FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Rapid Visualization Development based on Visual Programming

Developing a Visualization Prototyping Language

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computergraphik & Digitale Bildverarbeitung

eingereicht von

Benedikt Stehno
Matrikelnummer 0225175

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.-Prof. Dipl.-Ing. Dr. techn. Eduard Gröller
Mitwirkung: Dipl.-Ing. Dr. techn. Martin Haidacher

Wien, 20.10.2011

(Unterschrift Verfasserin) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Rapid Visualization Development based on Visual Programming

Developing a Visualization Prototyping Language

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computergraphik & Digitale Bildverarbeitgung

by

Benedikt Stehno
Registration Number 0225175

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.-Prof. Dipl.-Ing. Dr. techn. Eduard Gröller
Assistance: Dipl.-Ing. Dr. techn. Martin Haidacher

Vienna, 20.10.2011

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Benedikt Stehno
Geymüllergasse, 1180 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasserin)

Abstract

Over the years, many visualization tools and applications were developed for specific fields of science or business. Staying in the alcove of their field, these are highly suited and optimized for visualizing specific data, with the drawback of not being flexible enough to extend or alter these visualizations for other purposes.

Often, customers of such visualization packages cannot extend them, to fit their needs, especially if the software is closed source. But even using open source software does not solve this problem efficiently, since to extend the software, costumers need to have programming skills and are often forced to reimplement algorithms or visualizations which already exist, making rapid development impossible.

The goal of this thesis is to develop a dataflow visual programming language (DFVPL) and a visual editor for the rapid development of 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 visualization program. 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.

This programming language and its editor run platform independently to reach a high number of potential programmers, respectively users, to develop visualizations, since they are not bound to a specific platform. It is extendable, by means of self developed modules and data types to extend the language. Programming and editing a visualization is easy and fast, even for people with only little programming experience. The production cycle of the development of visualizations is reduced to a minimum. This is achieved by reusing and combining existing modules.

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

Kurzfassung

Seit geraumer Zeit wurden Visualisierungstools speziell für bestimmte Bereiche in der Forschung und für die Wirtschaft entwickelt. Diese sind oftmals für ihr Einsatzgebiet hochgradig optimiert, was es beinahe unmöglich macht, diese auf einfachem Wege für neue Anforderungen anzupassen.

Oftmals können Anwender solcher Visualisierungs-Softwarepakete diese nicht ihren eigenen, spezielleren Anforderungen anpassen, besonders wenn es sich um proprietäre Anwendungen handelt. Aber auch Open-Source-Software löst dieses Problem nicht zufriedenstellend, da die meisten Endkunden nicht über Programmiererfahrung verfügen. Oftmals müssen Algorithmen oder ganze Visualisierungen, die schon existieren, solchen Softwarepaketen hinzugefügt werden, um die gewünschte Funktionalität zu erreichen. Dieser Umstand macht eine schnelle Entwicklung unmöglich.

Das Ziel dieser Diplomarbeit ist es, eine visuelle Programmiersprache (Visual Programming Language - VPL), die auf datenstromorientierte Programmierung basiert, zu entwickeln. Anwender entwickeln Visualisierungen mit dieser Programmiersprache OpenInsightExplorer, indem sie vorgefertigte Module miteinander graphisch verbinden, anstatt Source Code zu schreiben. Jedes dieser Module stellt einen Teil der benötigten Funktionalität einer Visualisierung dar. Der Aufbau der Module ist so konzipiert, dass sie sich wie unabhängige Black Boxen verhalten und ausgeführt werden, sobald sie Daten empfangen. Dieses Design erlaubt es Module öfters zu verwenden und erleichtert die Programmierung dieser.

Die Sprache selbst als auch ihre integrierte Entwicklungsumgebung ist plattformunabhängig, um eine Vielzahl von möglichen Anwendern zu erreichen. Des weiteren ist diese leicht zu erweitern hinsichtlich zusätzlicher Module und Datentypen. Auch Personen mit wenig Programmiererfahrung sind im Stande, in dieser Programmiersprache Visualiserungen zu entwickeln. Ebenfalls wird die Entwicklungszeit neuer Visualisierungen reduziert, indem großteils nur fertige Module zu einer Applikation kombiniert werden müssen.

Um die Verwendbarkeit der Sprache zu testen, wurden zwei unterschiedliche Beispiel-Visualisierungen in der Sprache umgesetzt. Jedes Beispiel verlangte unterschiedliche Datentypen, Algorithmen und Renderingtechniken.

Contents

CHAPTER 1

Introduction

Many tools were developed to visualize data so far. They are all designed to transform data into meaningful visual images, that should allow people to gain insight and help to interpret the data. These tools help to understand data faster and in a more intuitive way. Furthermore, some visualization applications were developed to explore data sets and extract information in an interactive way. A great number of people could benefit from using visualization techniques for their every day life, to gain insight into various matters. But only experts use visualization software more or less exclusively. This can be reasoned by the following facts:

Most of the visualization software or tools were highly optimized for a specific field of science or business, with the drawback of not being flexible enough to extend or alter these visualizations for other purposes. For example, a highly optimized visualization package for flow rendering is often incapable of rendering a histogram of the stream line lengths being displayed. To mention yet another example, a graph based visualization of a social network lags the feature of rendering a pie chart representing the different connection types from a user in focus.

Therefore some visualization suits were developed which contain collections of different visualization methods and techniques. User can combine and/or configure to design custom visualizations. This combining and customization step is often limited to a certain level. Only a few aspects of a visualization can be configured and only certain visualization techniques can be combined at all.

Users could extend these visualization suits implementing certain needed functionality on their own. In most cases this is a rather impossible task, especially if the software is closed source. But even using open source visualization software does not solve this problem efficiently since to extend the software, costumers need to have programming skills. They are often forced to re-implement algorithms or visualizations which already exist, making rapid development impossible.

However some visualization suits or applications do support a plugin feature, which allows to extend the software with features or visualization techniques without altering the framework,

e.g., one visualization software only supports reading data out of specially formated files and lacks the ability to read the data out of a database instead. This feature may be added as a plugin to the software package. On the downside such plugins can actually only be developed by users with significant programming knowledge. Moreover the developers must be familiar with the programming language the visualization application is written in.

People have to choose a specific visualization software that hopefully supports all requested features they need. In the case it does not do that they have to deal with the problem that they cannot extend or configure the software easily. To summarize the problem: ***It is a rather complicated task to rapidly develop custom visualizations especially for people without any significant programming experience***.

To solve this problem we designed and implemented a rapid development framework for visualizations in which even little experienced programmers can develop visualizations. It is called ***OpenInsightExplorer*** and it is basically a programming language for visualizations. Users build custom visualizations with OpenInsightExplorer by arranging and connecting graphical elements, rather than writing source code. Using this so called ***visual programming*** paradigm allows even users with few empirical programming knowledge to develop visualizations. In addition, OpenInsightExplorer makes also use of the ***dataflow programming*** paradigm. In the case a language merges both paradigms together, it can be classified to the family of ***dataflow visual programming languages (DFVPL)***. OpenInsightExplorer is a member of this family.

OpenInsightExplorer's usability was evaluated by implementing two different example visualizations. Each example originates from different areas of visualization, therefore demanding different data types, data transformation tasks and rendering. These example implementations should ensure that the language is capable of covering distinct areas of information and scientific visualization.

This thesis is structured in the following way: The rest of this chapter will introduce the basic concepts of information visualization and each step of the visualization pipeline will be explained. Chapter 2 provides a short introduction to the concepts of visual programming and a more detailed one for the dataflow programing paradigm. Furthermore this chapter contains a section which deals with the historical development of dataflow visual programming languages. Starting with the first early members of the family all important successors get examined which introduced noteworthy essential features. The last section of chapter 2 gives a state-of-the-art overview of present dataflow visual programming languages. The following chapter 3 describes in detail the idea behind OpenInsightExplorer, its design decisions and the features it has. Chapter 4 features the implementation details of OpenInsightExplorer. Various aspects of the implementation of the framework and its features are explained in this chapter. The usability of the framework is evaluated in chapter 5. It depicts how the example visualizations got implemented with OpenInsightExplorer. Furthermore, it discusses on how intuitive these implementation tasks are. A discussion and future work section can be found in

chapter 6. It also deals with possible future improvements of the framework. The conclusion of this thesis is given in the final chapter 7. The conclusion exams how many preset goals of OpenInsightExplorer were finally achieved.

A ***User's Guide*** and a ***Programmer's Guide*** for the OpenInsightExplorer framework can be found in the Appendix of this thesis. Both can be read independently from the rest of this thesis and each other. They function as stand alone manuals. However, the thesis references to both manuals to provide further details at several times.

1.1 Visualization

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.

Visualization techniques can be subdivided into two main groups (by neglecting several other smaller subgroups). One subgroup is the ***scientific visualization*** branch, which concerns primarily visualizing three dimensional phenomena. This area emphases realistic rendering of volume data sets which perhaps have a dynamic time component. The other subgroup is the ***information visualization*** branch. This group deals with techniques to visually represent large quantities of high dimensional data, mainly large scale collections of non numerical data. Despite the fact that scientific and information visualization cover distinct areas both share the same processing steps which are described in the following section.

1.2 The Visualization Pipeline

Every visualization follows the concept of the visualization pipeline [5, 6], illustrated in Figure 1.1. Different subgroups of visualization may merge several processing steps together or simply skip individual steps. To generate the final visualization output, for example an image, the data gets transformed in each step of the pipeline. Users may interact with certain aspects of each processing step.

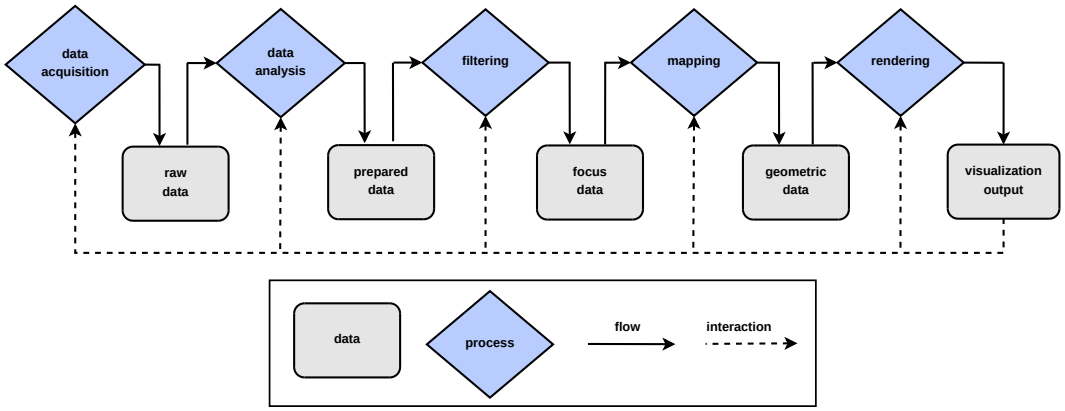


Figure 1.1: The visualization pipeline

The visualization pipeline contains the following successive processing steps:

- ***Data acquisition***
  In the first step the user defines the data source from which the data should be loaded and visualized. The data may get read out of special formated files, databases or various other sources like simulations or real time measurements. The result of the data acquisition is *raw data*.

- ***Data Analysis***
  The data is prepared for visualization in this step for example by interpolating missing values, applying a smoothing filter or correcting erroneous measurements. Normally, little to none user interaction is required for this step. The outcome of this processing step is *prepared data*.

- ***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. After this stage only *focused data* remains in the pipeline for the further processing steps.

- ***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*.

OpenInsightExplorer provides modules for each of these successive processing steps. The software gives users the opportunity to build their own costume visualizations by connecting graphical representations of these modules together, designing a custom visualization pipeline.

CHAPTER 2

State of the Art

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

This chapter begins with a short introduction to the concept of visual programming, followed with a more detailed one for the dataflow programming. The later introduction is executed in far more detail since it contains information which is necessary to understand the implications for the design of OpenInsightExplorer. Besides this chapter contains a section which deals with the historical development of dataflow visual programming languages. Starting with the first early members the section exams all important successors which introduced noteworthy essential features. At the end some examples of present members of visual dataflow programming languages are given and are examined.

2.1 Visual Programming

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 counter-parts using graphical metaphors. 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*** [15], where users can create, modify or extend parts of a software without any significant knowledge about programming.

Every visual programming language can be classified into one of these three basic categories: icon-based, form-based or (block) diagram-based, depending on which type of visual expressions are used.

Many visual programming languages follow the concept of ***boxes and arrows*** which belongs to the category of diagram based visual programming. Boxes (or nodes) represent entities which are connected by arrows (or arcs). Such connections express relations between entities. By connecting boxes together programming converts into the task of designing a graph which represents a program. Dataflow languages are also based on the boxes and arrows concept, which seems to be one reason why the visual programming and dataflow programming paradigms got combined.

OpenInsightExplorer is based on the boxes and arrows visual programming concept. Boxes represent modules or processing steps from the visualization pipeline 1.2 and arrows express paths, on which the data flows from each processing step to the next one.

2.2 Dataflow Programming

Dataflow programming languages consist of ***nodes*** and directed ***arcs*** (see Figure 2.1), following the boxes and arrows concept [1, 10, 12, 13, 23, 26].

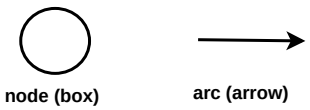


Figure 2.1: A ***node*** and an ***arc*** - the basic elements of a dataflow language.

Several of these basic elements are connected together to a graph or network, representing a program. Nodes can be understood as ***black boxes*** and they perform calculations on the data they receive such as primitive arithmetic or comparison operations (e.g., perform an addition or multiplication). Nodes send ***data tokens*** through these arcs to other connected nodes. Data tokens can be numbers, arrays or even pointers to objects. Arcs behave like unbound FIFO (first-in first-out) queues [22] between a sending and a receiving node. Arcs which flow to a node are called ***input*** arcs and the node can receive data tokens through them. In contrast arcs which flow away from a node are called ***output*** arcs and the node can send data through them.

Nodes perform calculations or tasks as soon as they have received all necessary data tokens for the execution. This principle is called the ***dataflow execution model***. Sometimes they only need to receive data on some of the input arcs and not on all of them for a specific task. A complete set which will trigger an execution of a specific task, is called a ***firing set*** [1, 7, 10]. Nodes can have several firing sets which will trigger different operations. On execution, the node removes the data tokens from the firing set input arcs and places the results of its computations onto its output arcs. Finally it waits until a new firing condition arises. Multiple nodes can become ***fireable*** at the same time and can therefore be executed in parallel. The dataflow execution model assures that operations are executed as soon as all vital data is available.

This scheme differs from the ***von Neumann*** execution model which most computers implement. Program instructions are written in a sequential order and are executed sequentially,

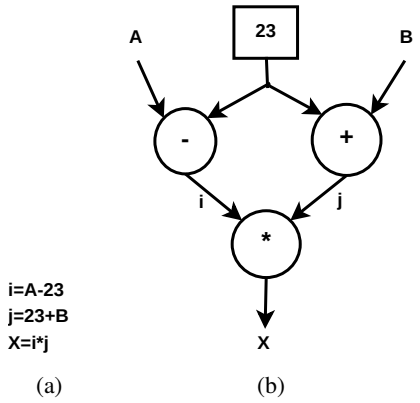


Figure 2.2: Comparison between a sequential program (a) and its equivalent dataflow graph/network (b).

one after another (see Figure 2.2 (a) for an example program). The execution of a program depends on the ordering of the instructions which might not be optimal. Some instructions may get executed later than they could have been since all vital operants were already valid at a earlier time. Using the classic von Neumann execution model no instructions are executed in parallel.

Figure 2.2 (a) shows a traditionally sequentially executed program and its equivalent as a dataflow graph (b). Further a constant (represented as a square) is a part of the graph. It has a forking output arc. This means it puts on each forked arc branch a token with the same value. The output gets duplicated. As soon as data is available at the input arcs of the subtraction (A) and addition (B), these nodes can operate. Their second operant is a constant (the square) which always provides data because it fills up the FIFO buffer of its output arcs with its constant value. Hence these nodes can are executed in parallel since their firing set depends only on one variable input. Furthermore when more data arrives from A and B, the subtraction and addition nodes can calculate intermediate results even when the multiplication node still operates on the first wave of data. This behavior is called ***pipelined*** dataflow [16, 37].

In contrast to this the von Neumann execution model (see Figure 2.2 (a)) takes three time units to finish (assuming one unit for each instruction), which are always executed in the same sequential order. No pipelining, no parallel execution and no intermediate results are calculated.

2.2.1 Determinism of the Dataflow Model

Dataflow languages are called ***well behaved*** in the case the nodes produce exactly one output set of data tokens for one input set. They generate new data tokens for their results and do not modify any consumed ones. This behavior makes a dataflow language become a ***pure*** dataflow language which is equivalent to a functional language. Functional languages treat computation as the evaluation of mathematical functions and avoid state and mutable data. Besides that the absence of a globally data storage makes it side-effect free, since all operations are based on local

data only (the sent and received tokens). Of course in that case the data tokens cannot contain any pointers to global stored data. On the same input tokens such a pure dataflow language produces ***always*** the same results and it becomes ***deterministic***. In some cases this can be a desired purpose.

2.2.2 Controlling the Flow of Data Tokens

Occasionally the result of one node's output arc will be needed as an input operand for more than one other node (e.g., see Figure 2.2). In that case the node will duplicate its result token and put it on every of its forked output arc branches. This preserves the data independence (side-effect freeness) and deterministic behavior since all input arcs FIFO buffers will have their own duplicated copy of the data token.

On the other hand, it is not permitted to simply merge arcs together. This could compromise the ordering how the data tokens arrive at a merged input and therefore the computation of a node itself and in turn the determinacy of the language. Additionally a mechanism is needed that allows to switch between destination nodes for the data tokens. Many dataflow languages use special ***gate*** nodes to control the flow of tokens in a dataflow network.

Gate Nodes

There are two types of gate nodes that allow to manipulate the flow of tokens without endangering determinacy. One covers the case of two or more input arcs which are merged together. The second one deals with the problem to steer the output to a specific destination.

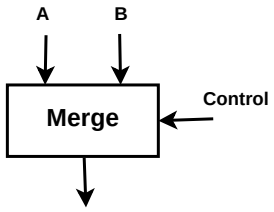


Figure 2.3: A ***merge*** gate.

The ***merge*** gate node, depicted in Figure 2.3, has three input and one output arc. This node reads first from the control input, which can only carry a ***boolean*** typed token. Depending on the received state of the value, it will delegate one token from its input to the output. In the case the control carried a ***true*** it will fetch from input labeled A, otherwise from B. The merge gate can be expanded to handle more than two input arcs. Of course, the control input will be typed to a numeric data type instead of boolean. In this case, the received number will decide which input arc is selected.

A ***switch*** gate, as illustrated in Figure 2.4, has one control input, a second input and two output arcs. It places the token from the input depending on the read control value on one of

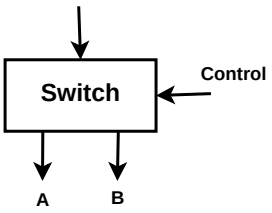


Figure 2.4: The ***switch*** gate.

both outputs. On ***true*** it places the input on A and on ***false*** it delegates the token to B. The switch gate can also be modified the same way like the gate node, to handle more output arcs.

2.2.3 Alternative to the Token-Based Model

An alternative to the token-based model was developed and referred to as the ***structured model*** [10]. It does not suffer from one drawback of the token-based approach. In the token-based model, nodes cannot access the incoming data randomly and have no history sensitivity. The structured model follows the same arc and node design as the token-based version. But instead of using a FIFO buffer, only one data object is placed onto an arc and remains there. This object will hold a structure that references data, instead of sending tokens across nodes in the dataflow network. This structures can be accessed randomly but can also hold infinite arrays which mirrors the functionality of streams in the token-based approach.

The structured model does not store data efficiently and introduces more complexity, since it has to keep track of which data can be released. It operates the same way as Java's garbage collector which frees data as soon as it is not referenced by a pointer anymore. This model only gained acceptance in the research area. Almost every commercial implementation of the dataflow model followed the token-based approach.

2.2.4 Dataflow Execution Architectures

There are two theoretical main approaches for the implementation of the dataflow execution model. The first architecture is the ***data driven*** approach, see Figure 2.5 [10–12]. A node stays inactive until a routine, called the ***driver***, determines, that on its input arcs a whole firing set is placed. The driver will execute the node, which will consume its input tokens and place its results onto all of its output arcs. This may cause other nodes to become fireable which will be executed subsequently. The driver can operate in additional threads which will scan independently for fireable nodes and can execute them in parallel.

Figure 2.5 illustrates an example of the data driven approach. This example is simplified so that every node only has one firing set. A node becomes only fireable if on all of its input arcs data has arrived. In the first step (a) the driver scans trough all nodes of the dataflow network to determine which are in the fireable state. It finds two nodes that are ready for execution, A and B. Since they have no input arcs at all, they are always fireable. In step two (b), the driver

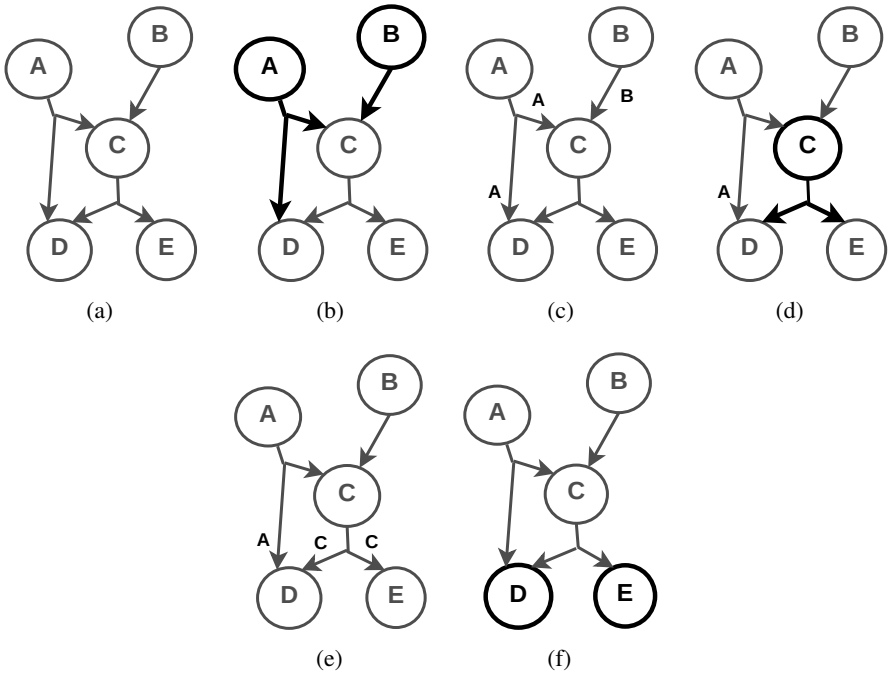


Figure 2.5: Example of a data driven dataflow network.

executes both nodes in parallel. This is emphasized in the figure by using bold outlines for the nodes and their output arcs. Both executed nodes place their results on their output arcs - as depicted in (c). Now the driver determines that the node C just became fireable, since on both of its input arcs a data token is available. Node D did not become fireable since its other input arc is still empty. Depending on the used driver algorithm it also may happen that the nodes A and B are fireable again too. But most implementations of drivers only execute always firable nodes like A and B only if no other execution-ready node was found. Therefore only node C gets executed as depicted in step (d). Again, the executed node places it result token onto its output arcs (e). Finally nodes D and E become fireable and get executed in the last depicted step (f).

Another approach is the ***demand driven*** architecture, see Figure 2.6 [10, 22]. Nodes issue demands to the relevant nodes connected to its input arcs. They propagate their request for data to nodes linked to their inputs. Requests can expand trough the whole dataflow graph. As soon as all requests from a node are satisfied (it received data on all inputs of its firing set) it gets executed. It places the result ***only onto the branch*** of the output arc where it was prior ***inquired for data***.

An example of a demand driven network is depicted in Figure 2.6. It makes the same simplification like the last one. Every node only becomes fireable as soon it has received data on each of its input arcs. The demand driven execution starts at nodes which have no output arcs. Every of these nodes alternately issues a request for input data. This example starts with node

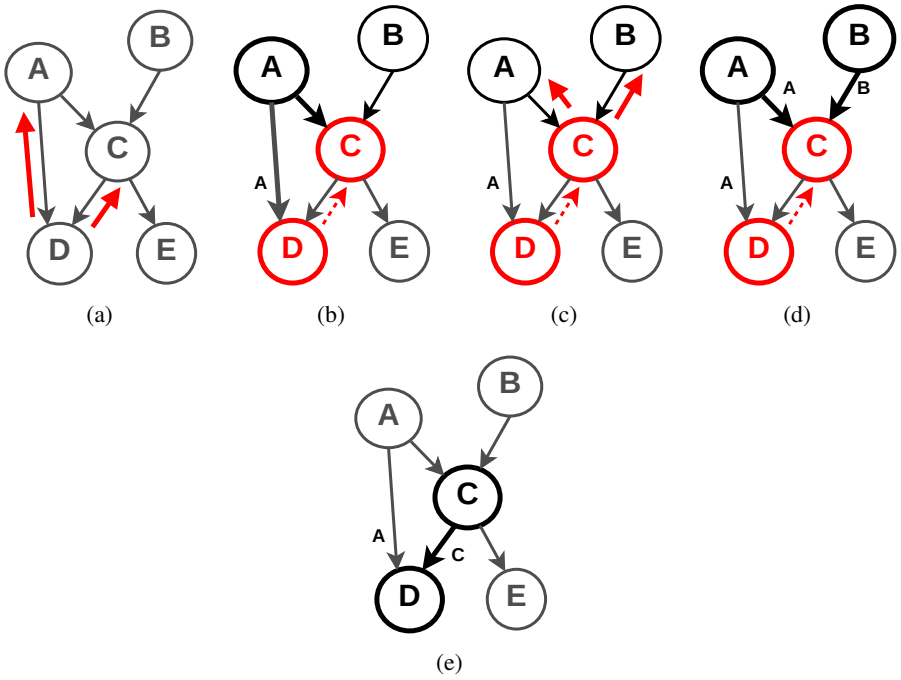


Figure 2.6: Example of a demand driven dataflow network.

D, but E would also be a possible candidate. In the first step (a) node D issues to all of its input connections a request for tokens. This is illustrated in the figure by using a red colored bold outline for the node and additional red arrows, which represent requests. In the second step (b), node A gets executed right away as it does not depend on any input data at all. It places ***only*** a data token onto its output arc which is connected to node D. ***Nodes only place data tokens onto branches where they were previously inquired for data in this execution model***. Node C recognizes the data request from node D (it's now depicted with a red outline). Since it depends on data from other nodes it cannot get executed right away, unlike it happened previously with A. Therefore node C issues requests to the nodes linked to its input arcs in step (c). Both partner nodes A and B can satisfy the request immediately because both are not dependent on any input data. They get executed and place their result tokens onto their output arcs as depicted in step (d). In the final step (e) node C becomes fireable and is executed. Its result token moves node D into the firable state (data rests on all of its input arcs), which gets executed shortly afterwards.

Both architectures have their benefits and drawbacks. The data driven approach does not suffer of the request propagation overhead which the demand driven method produces. On the other hand, with the demand driven approach it is possible to eliminate the need for certain gate node types (see Section 2.2.2) [10]. For example, the ***switch*** (see Figure 2.4) node becomes redundant, because only needed data is demanded at all. Either the node connected to the output arc labeled A or the one linked with B will request data, but not both.

2.2.5 Practical Realization of the Dataflow Model

In theory the pure dataflow model seems a promising approach but the implementation of it is a very difficult task [34]. This theoretical model makes assumptions which cannot be practically realized. First of all the model states that FIFO buffers are unbound in capacity which cannot be converted into reality since computer systems have limited memory. In addition the model states that an up to infinite number of instructions could be executed concurrently implying an unrealizable infinite number of processing elements. These restrictions dictate that the pure dataflow model cannot be implemented entirely. Thus the minor changes of the model (to enable an implementation) can result in deadlocks [1].

2.3 History of Dataflow Visual Programming Languages

This section is about the historical development of dataflow visual programming languages (DFVPL). Starting with the first early members it examines all important successors which introduced noteworthy essential features. Some of these features or concepts are explained in detail since they were essential to the design of OpenInsightExplorer and can be found in present dataflow visual programming applications.

One of the first pure visual programmable dataflow programming languages was ***Data-Driven Nets (DDN)*** [9]. It was developed by A. L. Davis in the 1970s and operates at a very low level on the token-based dataflow approach. Nodes perform rather simple tasks only. Programs were already rendered as graphs. Arcs contain FIFO queues and are restricted to a certain data type, introducing so called ***typed arcs*** to dataflow languages in this way. Figure 2.7 depicts a typed arc example. The arc is restricted to the data type boolean. Node A can only send boolean tokens through this arc and B can only receive such tokens. This feature ensures that both nodes will send and receive tokens of the same data type. Typed arcs proved to be an essential feature since they introduce a type-safety check mechanism to DFVPLs. A equivalent mechanism was implemented in OpenInsightExplorer to provide the same essential functionality like typed arcs.

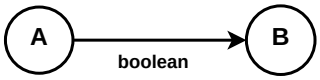


Figure 2.7: Example of a typed arc. Node A sends only tokens of the data type boolean to B.

Davis further developed a ***Graphical Programming Language (GPL)*** which proved to be a higher level language version of and derived from DDN [39]. It introduces structured programming to dataflow visual programming languages. Whole sub-graphs of a dataflow network can be expanded out of a single node (see Figure 2.8), structuring a program with the top-down refinement approach [40]. These sub-graph nodes can be defined recursively. As its predecessor DNN, GPL also uses typed arcs. GPL introduced facilities for visualization and debugging. Moreover, it provided the possibility for additional on-demand text-based programming.

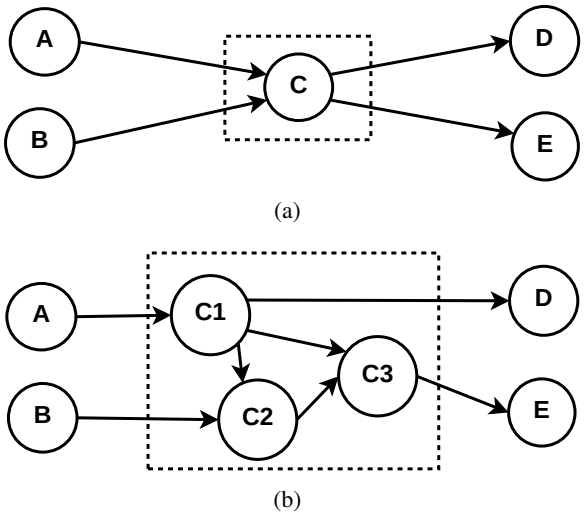


Figure 2.8: A whole sub-graph (the nodes C1, C2 and C3) of a dataflow network (b) can be collapsed to the single node C (a) and expanded again.

Functional Graph Language (FGL) [21] was created by Keller and Yen in the 1980s. It is a visual dataflow language that is based on the structured model instead of the commonly used token-based approach. Almost similar to ***GPL***, this language also supports top-down stepwise refinement for structuring programs.

In 1983 the first version of ***ProGraph*** [8,18,27], a more general purpose DFVPL developed by Acadia University, was released. It is a multi-paradigm programming language including the dataflow, visual and object oriented paradigm. It became a cross-platform supported software with executables/versions for Classic MacOS, Microsoft Windows and Mac OS X. The methods of each object are defined using dataflow diagrams (see Figure 2.9). ProGraph includes iterative constructs and permits procedural abstraction by condensing a graph into a single node. ProGraph has also been used as a subject in research.

The concept of programming consists of two tasks, emphasize Gerlernter and Carriero [17]. First programmers must express how computations are made and additionally how these computations are coordinated. This distinction becomes more and more important in the advent of distributed and heterogeneous computer systems. They suggested to separate these tasks into a computation and a coordination language. Dataflow graphs almost naturally fit the specifications of ***coordination languages***. Through their node and arc based concept, they express relations between computations (nodes) and coordination of their execution.

Most successors of ***FGL*** separated the tasks of computation and coordination. They use the dataflow model for coordination and as execution model only. Often their nodes are written in a different programming language but their execution still follows the rules of the dataflow principle. Additionally nodes became more complex and they execute more sophisticated tasks in comparison to those of previous dataflow languages. For example ***Vipers*** is a DFVPL that

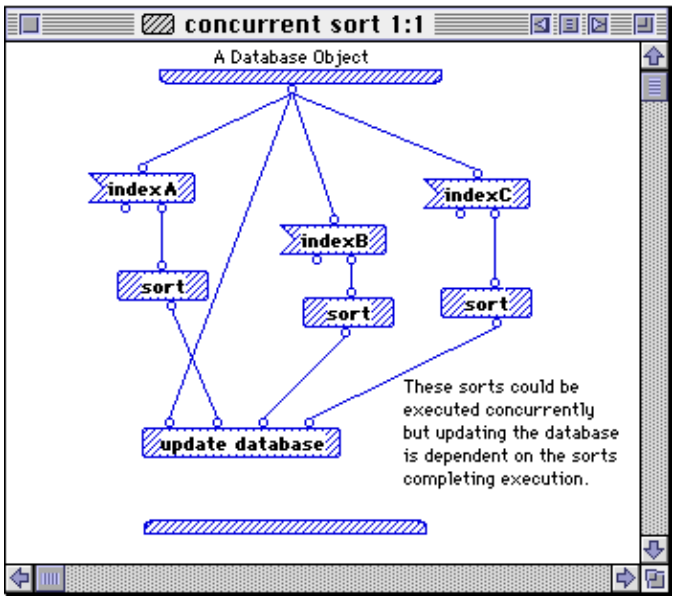


Figure 2.9: A ProGraph screenshot depicting a member function of an object.

uses the dataflow paradigm for coordination and as execution model [3]. Its nodes are written in the stand alone scripting language ***Tcl***.

Morrison [26] proposed a system where nodes can be written in ***any*** arbitrary programming language. The nodes can be arranged in a single visual network editing environment which controls the coordination of execution. His dataflow-based programming concept does not follow strictly the rules of the pure dataflow model. Moreover he also emphasizes a coarser grained dataflow approach which means that nodes should execute more complex tasks rather than primitive operations. He reported that his method seems to prove to be practical when applied in a real-world scenario. He also proposed the concept of ***token streams***. Instead of only sending independent data tokens through a dataflow network, nodes can group a series of data tokens together by surrounding them with special ***bracket*** tokens.

Granular Lucid (GLU) developed by Jagannathan [19] is based on Morrison's concept. The coordination task is written in ***Lucid***, a textual dataflow language. Whereas functions are written in another, properly sequential, language. ***GLU*** also promotes the more coarse grained dataflow approach as Morrison suggested. With ***GLU***s coarse grained approach, Jagannathan proved that it is feasible to achieve similar performances compared to conventionally developed applications for parallel processors.

OpenInsightExplorer follows Morrison's concept with the exception that the same programming language is used for the framework and its nodes. OpenInsightExplorer can be classified as a coarse grained dataflow visual programming language since most of its nodes

execute rather complex tasks. The framework supports the concept of ***token streams*** too allowing to group data tokens to a stream.

2.4 Present Dataflow Visual Programming Applications

This section addresses current commercial applications which use the dataflow visual programming paradigm. It examines how the visual programming is implemented and what additional features are supported.

2.4.1 LabVIEW

National Instruments released 1986 the first ***LabVIEW - Laboratory Virtual Instrumentation Engineering Workbench*** [18, 21] platform for Apple Macintosh. LabVIEW is still in development and several versions of the platform were released up to now. With this software users can build virtual instruments by connecting different function nodes within a block diagram by drawing wires (see Figure 2.10). Every virtual instrument has a visual front panel. Users can add controllers and output displays to it which will be represented as function nodes within the editor. Structural programming is provided by LabVIEW. Users can develop their own functional nodes and reuse them arbitrarily in their projects.

LabVIEW uses a programming language, referred to as ***G***. Function nodes are executed as soon as all their input data become available following the execution model of a dataflow language. In newer versions of LabVIEW execution-ready nodes are scheduled by a build-in scheduler of LabVIEW on multiple threads/processes. The software is developed to be cross-platform capable with support of Windows, Mac OS X and Linux.

LabVIEW demonstrated that large projects can be developed faster with a visual programming language in comparison to traditional text based programming languages. Jet Propulsion Laboratory, a NASA research and development center, reported that through the visual syntax of LabVIEW a large project was significantly faster developed than the same system in C [2]. LabVIEW became an industrial success and its benefits made it popular among researches.

2.4.2 KNIME

KNIME [4,24,33] is a visual data exploration and data mining tool, written in Java. Its graphical workflow editor is implemented as an Eclipse [14] plugin (see Figure 2.11). The user can arrange various nodes or modules and connect them visually in the workflow editor with pipelines generating a dataflow network for analysis and visualization.

It is possible to write own nodes to extend the functionality of KNIME. It also has built-in nodes to incorporate with existing tools, such as Weka [38], R [31] and JFreeChart [20]. Modules or nodes do communicate only with one data structure called *DataTable* [4]. Its structure is similar to an SQL data table having a unique row identifier (primary key). Additionally the DataTable holds meta information about the columns such as their types and names. The support of dataflow networks containing loops is currently in an experimental state.

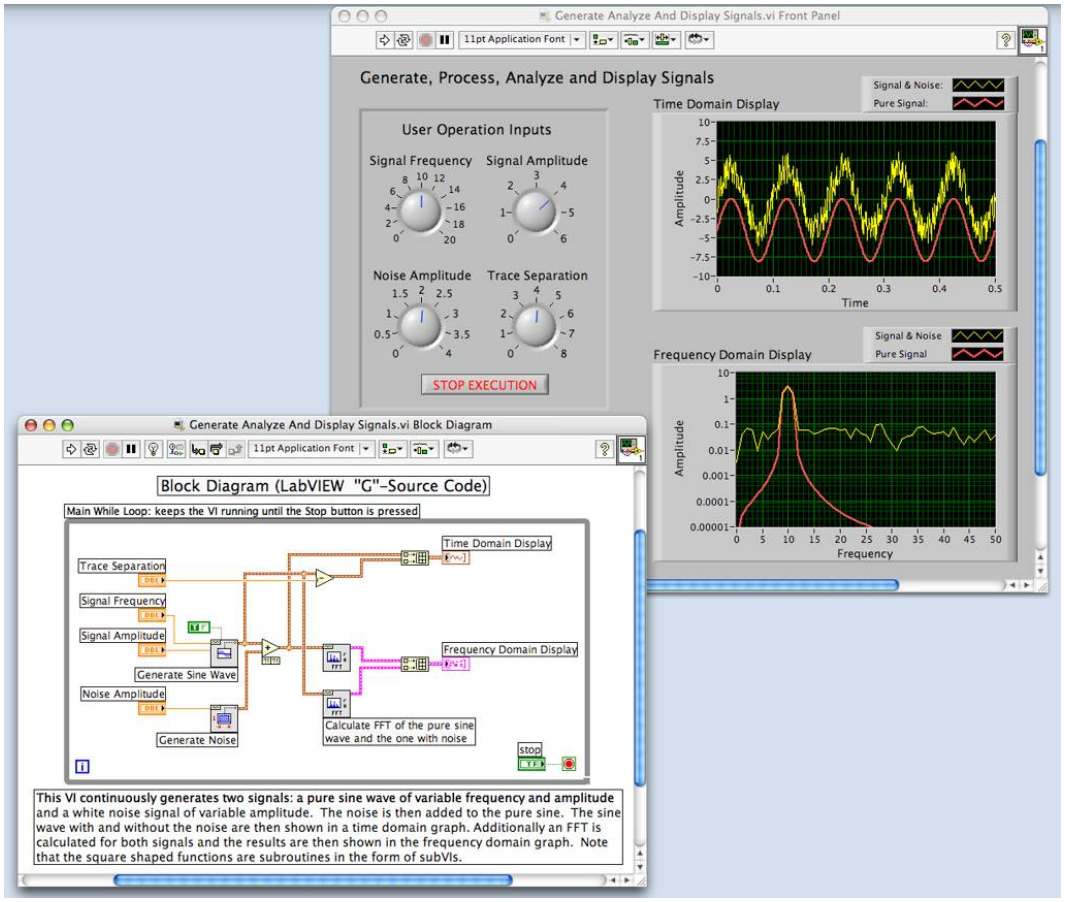


Figure 2.10: Example screenshot of a LabVIEW project, showing the blockdiagram source code (lower left window) and the visual front panel of a virtual instrument (upper right window).

2.4.3 OpenDX

OpenDX (Open Data Explorer) [28] is a cross-platform scientific data visualization software, developed by IBM. It can deal with different kinds of data such as scalar, vector or tensor fields. Visualizations can be programmed either using a scripting language or the visual program editor, depicted in Figure 2.12.

Each programmed visualization consists of connected modules, therefore following the dataflow principle. There are many ready-to-use modules of different kinds of visualizations (e.g., a streamline renderer), which only need to be configured properly.

Additionally OpenDX 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 determine automatically the minimum and the maximum of the dataset setting its boundaries appropriately. OpenDX is open source and user can expand its capabilities by programming their own visualization modules or using a build in scripting language.

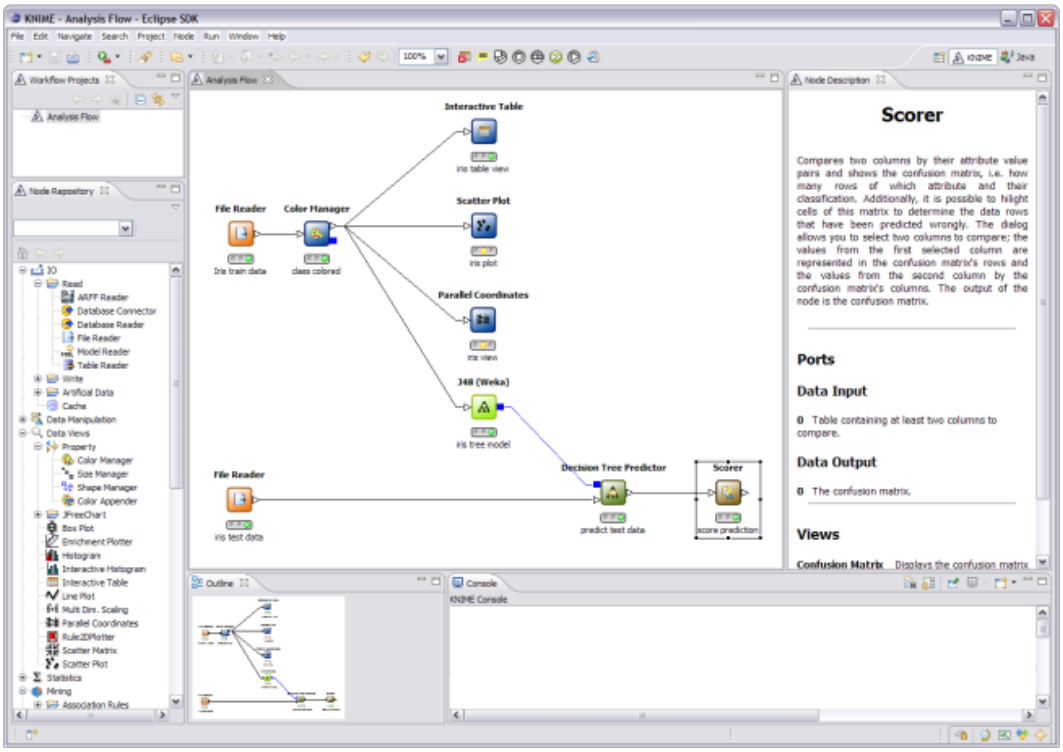


Figure 2.11: A screenshot of the graphical workflow editor.

2.4.4 Quartz Composer

Quartz Composer [30] is a visual based programming language for rendering and/or processing graphics. The user can arrange within a graphical editor nodes (called ***patches***) and connect them, generating the final program called ***composition*** (see Figure 2.13). These compositions can be played as a system screen saver, iTunes music visualization, as a Quartz Composer stand alone application or can be embedded into a Cocoa or Carbon application. Since this software is developed by Apple Inc., it is bound to a specific set of platforms and closed source. However it is possible to develop plugins and one's own specialized patches with new functionality. Since developers can not introduce new custom data types they have to emulate them with the already provided ones.

Quartz Composer supports following the native data types, which can be passed between the patches. It is not possible to extend these data types.

- Boolean
- Index
- Number (double precision floating point)
- String (unicode)

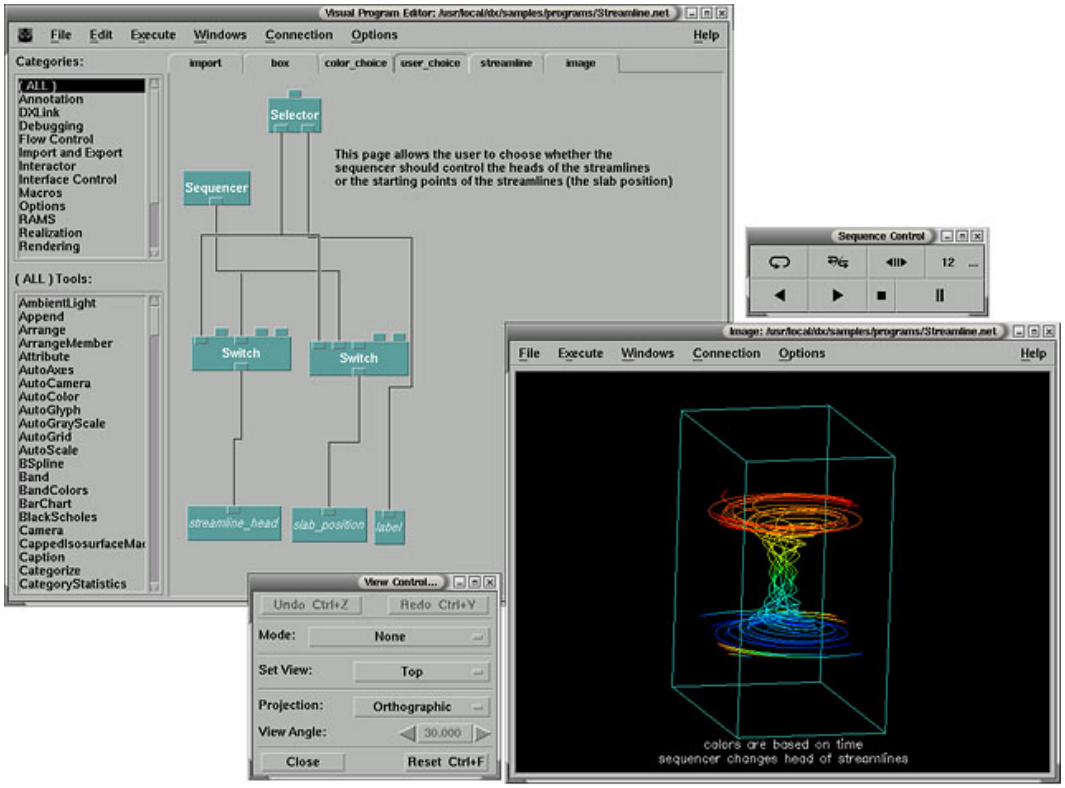


Figure 2.12: OpenDX screenshot.

- Color
- Image
- Structure (named or ordered collection of objects, including nested structures)
- Mesh (vertices, vertex normals, texture coordinates, colors)
- Interaction

2.4.5 Visualization Toolkit

The Visualization Toolkit (VTK) is an open source C++ library for developing visualizations and image processing applications [32, 36]. It has several interpreted interface layers including Java, Tcl and Python. VTK is cross-platform and integrates GUI toolkits such as QT and Tk. The Visualization Toolkit supports a wide variety of visualization algorithms for scalar, vector, tensor, texture and volumetric data. It also contains an extensive information visualization framework and supports parallel processing.

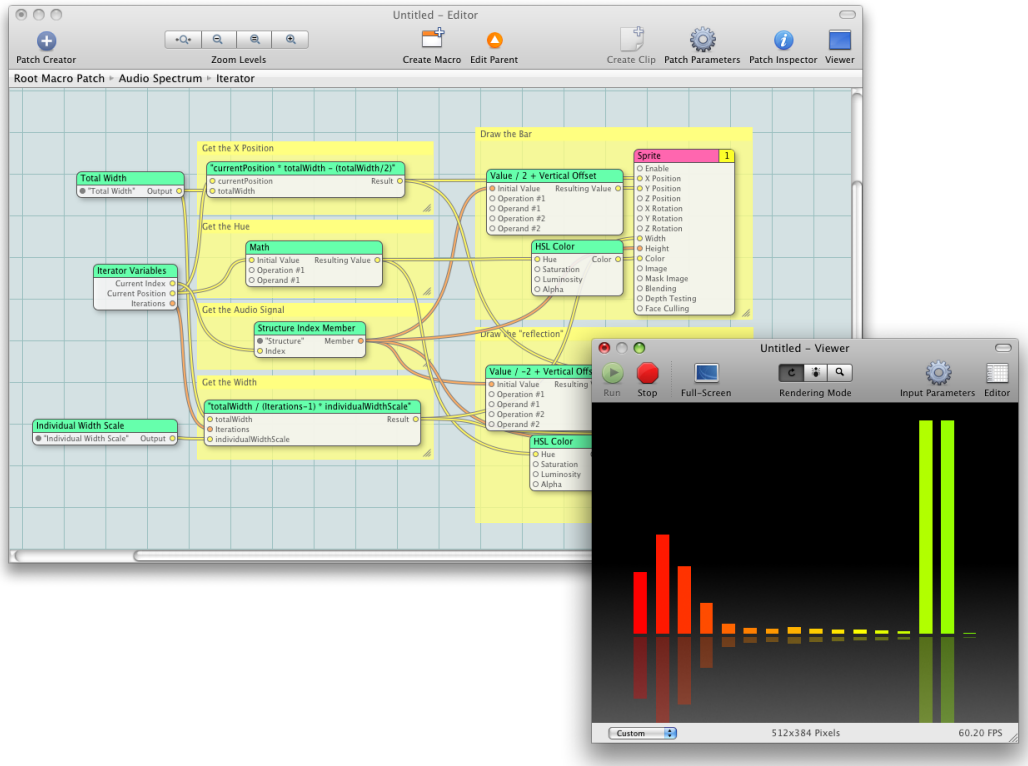


Figure 2.13: Quartz Composer screenshot.

The toolkit itself does not support ***any*** visual dataflow programming, but it was used as basis for many visualization applications that do. MeVisLab [25] for example integrates VTK modules in addition to its own ones. This cross-platform application framework allows users to program medical image processing applications and scientific visualizations with the visual dataflow programming paradigm (see Figure 2.14).

VTK's representation of a visualization pipeline follows the concept of dataflow programming. Its so called visualization model consists of independent modules which get connected together to a network. Each module performs algorithmic operations on the data as they flow through the network. The execution of the network is demand driven (each module requests new data from their inputs) or event driven in response to user input.

The modules can be further classified into three types: sources, filters and mappers. Source modules initiate the network and generate one or more output data sets. Filters require one or more inputs and generate one or more outputs. Mappers terminate the network and require one or more inputs.

The execution of a network is based on an implicit scheme. Each module maintains an internal modification and execution time-stamp. When output from a module A is requested, A compares its modification time and the modification time-stamps of its inputs against its last

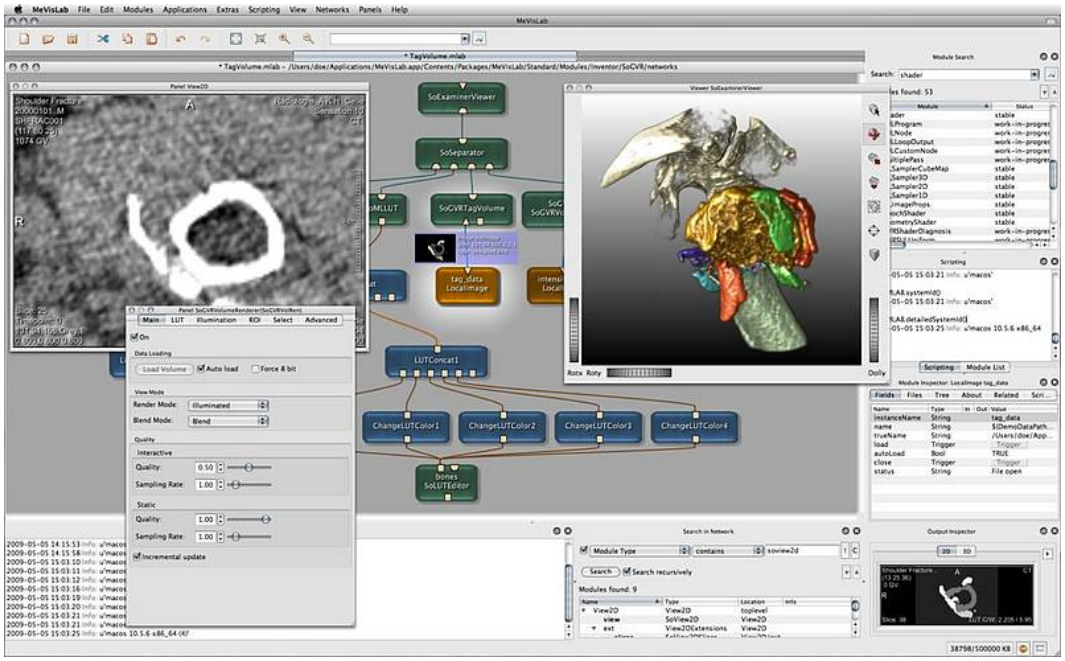


Figure 2.14: A screenshot of MeVisLab.

execution time. In the case A or one of its inputs was modified (more recently than A's last execution) it will be re-executed.

After examining the history of dataflow visual programming languages and the features of present state-of-the-art applications, the following chapter deals with OpenInsightExplorer. It describes the feature the languages has and the design decisions which were made for it.

CHAPTER 3

OpenInsightExplorer

OpenInsightExplorer is the rapid visualization prototyping language or framework we have developed. It is a solution attempt to the following problem: it is rather complicated to rapidly develop custom visualization especially for people without any significant programming experience. This chapter describes the idea behind OpenInsightExplorer, the design decisions for the framework and the (unique) features which are implemented into the software.

The basic idea of the framework is to combine the advantages of visual programming and dataflow programming (see Section 2.1 and 2.2). OpenInsightExplorer lets users 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 certain stage of the previously mentioned information visualization pipeline (see Section 1.2). 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 processing step to the next.

The modules are called ***patches*** in the OpenInsightExplorer framework. 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*** (Figure 3.1 depicts a graphical representation of the ***Box*** patch). Through its input ports a patch receives data from a previous stage of the visualization pipeline. It processes this data and passes its results to the next stage of the pipeline 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 (see Figure 3.2).

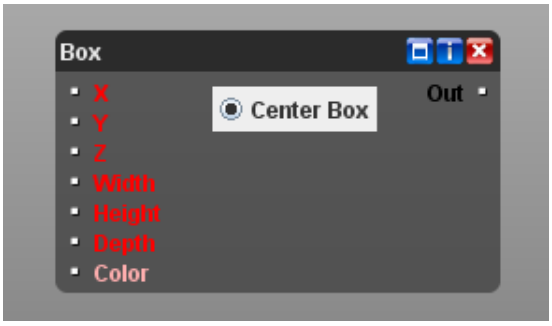


Figure 3.1: A patch named ***Box*** with input ports (named ***X***, ***Y***, ***Z***, ***Width***, ***Height***, ***Depth***, ***Color***) and an output port (***Out***).

Visualizations developed with the OpenInsightExplorer are called ***compositions***. A detailed composition programming tutorial can be found in section A.5 of this thesis. Using this simple visual programming concept allows users with little programming experience to program custom visualizations [18].

Patches are designed to function as dataflow nodes. They get executed as soon as data on their input ports arrives following the dataflow execution model. If a certain functionality or stage of a custom visualization pipeline is missing users with only a little programming experience can develop missing patches. Developers only need to implement a small Java interface to create a fully operational patch.

Since OpenInsightExplorer follows a modular approach for patches (each visualization is only a composition of independent patches) exchanging or adding new patches turns out to be simple. Also this modularization supports rapid development because patches can be reused for many different visualizations. Only missing new functionality must be implemented.

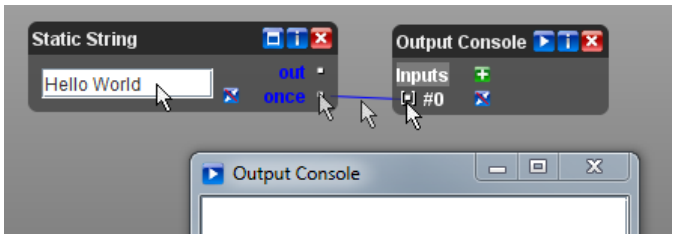


Figure 3.2: Programming with OpenInsightExplorer: two patches are connected together. The patch ***Static String*** sends its output to the patch entitled ***Output Console***.

3.1 Features of OpenInsightExplorer

This section lists some important and partially unique features OpenInsightExplorer supports.

- ***Open source and platform independence***
  OpenInsightExplorer is open source software. 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.

  However, the current version of OpenInsightExplorer can only be executed with the operating system Windows on machines with 32 bit or 64 bit architecture. The framework uses an OpenGL binding library (Jogl), which supports other platforms as well, but OpenInsightExplorer only implements the support of Windows binding of Jogl currently. Adding the Jogl support for other platforms to the framework should be a feasible task.

- ***Automatic Parallelization***
  OpenInsightExplorer is a visual dataflow programming language. Patches get executed like nodes of a dataflow language. The dataflow execution model automatically parallelizes the execution of nodes whenever possible (see Section 2.2).

- ***Streams***
  Instead of sending only individual data tokens between patches, OpenInsightExplorer implements the concept of Morrison's ***token streams*** (see Section 2.3). Patches can have special ***stream*** ports which enable to group data together to a stream (see Section 4.3.3).

- ***Growing ports***
  The growing ports mechanism of OpenInsightExplorer is a unique feature. It allows to add and remove ports dynamically to a patch while editing a visualization (see Section 3.2.4).

- ***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 will be discussed below). 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. This is comparable to the ***typed arcs*** (see Section 2.3) mechanism which was introduced through ***DNN***.

- ***Generic ports***
  To make patches more flexible, OpenInsightExplorer introduces a unique feature that allows 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 (see Section A.6 and B.10).

- ***Custom data types***
  Unlike Quartz Composer (see Section 2.4.4), for example, 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, etc.). This feature proved to be very useful for one group of our example visualizations (see Section 5.3).

- ***Classes as data type***
  In OpenInsightExplorer patches can send and receive instances of classes as data. These objects can contain (like any other Java class) methods and functions in addition to the data. For example, a class that represents a graph can have a method which returns the nodes of it. Furthermore such a class could implement many different interfaces and therefore represents multiple data types at once (e.g., 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 objects 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 enhances the usability of patches. For example: a patch renders graphs it receives. To render a graph it has to determine a camera position. Instead of implementing only one camera positioning model on its own it can delegate this functionality to a helper object. This helper object implements a function that returns a position for the graph. The patch receives this helper object on one of its input ports. Now multiple camera positioning patches can be implemented that will send helper objects with different implementations of the function. One may send an implementation that the whole graph is visible, another one sets the camera to zoom-in on the node with the most connections in the graph.

- ***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 location a developer can use 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 (see Figure A.6). The implementation of the GUI system is covered in section B.2 in detail.

3.2 Framework Design Decisions

This Section describes some important design decisions for OpenInsightExplorer.

3.2.1 Choosing the Development Language

One of OpenInsightExplorers conceptual goals is to be platform independent. Therefore the platform independent development language ***Java*** was chosen as development language. Other programming languages were also considered such as ***C++*** or ***C#*** to name a few. But only Java provides a runtime environment which is truly platform independent enough for the requirements of OpenInsightExplorer. Furthermore ***Java*** offers a built-in platform independent windowing toolkit and a huge runtime library.

Of course, other languages also provide such a functionality but mainly through other additional libraries which may or may not be platform independent. This in turn would force developers who want to extend OpenInsightExplorer to get familiar with these libraries. Also the project must be built for each release for every target platform separately. In contrast ***Java*** uses a virtual machine and just-in-time compilation to establish platform independence.

Moreover custom developed patches should be easy to add to an existing installation of OpenInsightExplorer, possibly even at runtime. Most of the other programming languages do not provide such functionality at all or in an inconvenient way. To achieve such a behaviour in those languages the platform independence often must be abandoned. Through the nature of ***Java***, loading new patches at runtime is a fairly straightforward task (see Section 4.2.1).

3.2.2 Dataflow Execution Architecture

One very important question was, which dataflow architecture OpenInsightExplorer should follow, the ***data driven*** or the ***demand driven*** approach (see Section 2.2.4). Both approaches possess distinct benefits and drawbacks. Two important factors had to be additionally taken into account. Which one of them would fit better to interactive and event driven GUI elements such as sliders or file-choosers because visualizations can be interactive and therefore contain such elements. Secondly which approach seems more suitable for developers who are only familiar with object oriented programming.

Dependent on the chosen architecture a way to handle the firing set of a patch had to be determined. The firing set is the condition under which a node of a dataflow network should be executed.

For example, a dataflow network contains a node representing a GUI slider (labeled S in Figure 3.3). This slider node sends values which are adjusted in the GUI slider element. Node A represents a simple addition node which reads from both inputs and adds the values together. The B node will send repetitively a set of data, embodying a filled buffer. According to this example setup a solution must be found for the question when the slider node reaches its firing state, since it does not depend on any other node in the dataflow network.

(a) (b)

Figure 3.3: Slider GUI element (a) and a dataflow network including a slider node labeled S (b).

There are several ways the slider node could reach the firing state and should send the current value of the slider depending on the chosen architecture.

- ***Only on value change***
  In the ***data driven*** approach, the slider node could send only data in the case the value has changed. This approach does not work, because the value is needed multiple times. In this scenario, node A would stop after the first addition, since it does not receive any more data from S. Of course node A could be implemented, so that it only reads once from the S input and adds the read value to the whole set of data from B. But as soon as node B sends the whole position set again, this approach will fail too.

- ***Always fireing***
  Node S could fire the whole time, filling up the FIFO buffer on the connection arc. This approach has a serious drawback. As soon as the value of the slider has changed, the following will happen: To one part of the data set from B the old value of the slider will be added, hence the values the FIFO buffer held up change of the slider's value. And to the other part the new value will be added. Furthermore, depending on the dimension of the buffer, this method will introduce lag to the whole visualization. The bigger the FIFO buffer is dimensioned the more of the old values must be consumed before the change appears.

- ***On demand***
  The slider node will only place the current value of the slider onto its output arc, as soon as node A tries to read from it. This is equivalent to the ***demand driven*** architecture. This will prevent introducing any lag but has the same problem we already had with the ***only on value change*** approach. Maybe parts of the set from B will be added with different values. Of course A can be implemented to read only once from S and add this value to the whole set from B. This approach does not fail in the case B sends the whole position set again. A can simply requests a new value from S for the whole set from B.

This little example even gets more complicated under the assumption that B should only send a set of data after the value of the slider has changed at all. This resembles the situation when a visualization should render only a new frame if one of the GUI elements were changed. This can be accomplished by adding an additional output arc to the slider node which will

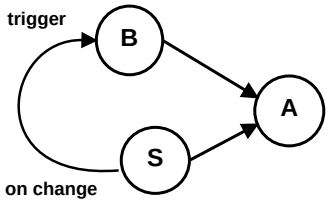


Figure 3.4: The slider example network with an additional trigger.

send a ***signal*** on a value change (see Figure 3.4). Further the buffer node B must receive an additional input, which will trigger it to send its buffered data.

To summarize this fairly simple example: Nodes behave very differently depending on the kind of functionality they implement or represent. Therefore they also have different firing sets and conditions on which they should be executed. And finally, it is possible to use ***signals*** to trigger events and more or less for synchronization purposes. The following conclusions for the OpenInsightExplorer can be deduced from this example:

- ***Patches need mechanisms to recognize if other patches need data from them***
  With this functionality interactive GUI control could send data only if it is really requested, following the ***on demand*** architecture.
- ***Patches should have the ability to recognize when data is sent to them***
  Patches must be able recognize if another patch wants to trigger some method of it (e.g., a buffer patch must provide a port which triggers the patch to send all buffered data).
- ***Patches must support multiple fire states that activate different functionality***
  Some patches are equipped with multiple input ports, e.g., a buffer patch. On one of the inputs it receives values to hold. The other input functions as a trigger for sending all data contained in the buffer so far over an output port.

OpenInsightExplorer uses a modified ***data driven*** approach extended with trigger and request functionality. With this design patches can have two kinds of ***ports*** (see Figure 3.5). Input ports which receive data and output ports to send data. A connection between an output and an input port corresponds to an arc in all dataflow graphs presented so far.

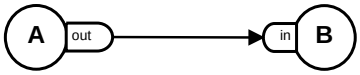


Figure 3.5: Patch A has an output port labelled ***out*** and patch B an input port named ***in***.

At the start of a visualization, a patch can spawn several threads which will run concurrently. When a patch reads from an input port, one of two possibilities can occur: If data is available,

it consumes the data token from the input port and the execution of the thread continues. In the case no data is available, the thread is suspended until a token has arrived at the port. This approach makes it unnecessary to define firing sets and to use a driver to determine which patch could be executed. Patches can decide ***on their own*** which data would be important for the current state and which may change during execution.

In the previous slider network example (see Figure 3.3) node A could be implemented as a patch for the framework as follows: It spawns a thread which will read alternately from both input arcs and add the received operands together.

In addition, output ports can recognize if an input port requests data. If an input port tries to read from an empty arc FIFO, the output port gets a notification. Developers can register a method in the output port which the port should call in such a case. This method is called ***listener*** function. The patch can recognize that data is requested and can respond to this demand. This conforms to the ***demand driven*** architecture. It can send data immediately if the value to be sent is already valid or known. Or it may request data itself from its input arcs for operands to determine the value, which will be sent afterwards. It should be noted, that the thread of the patch which triggered the listener, by reading from the empty input arc, will be suspended until the request has been satisfied.

The slider node in the mentioned example (Figure 3.3) can be implemented as patch by using this technique. Only if the node A reads from the arcs connected to the slider node and triggering thereby a request, the slider node will place a value onto the output arc.

An input port also provides the opportunity to register a listener function. This function will be called as soon as a token is put into a previous ***empty*** FIFO of an arc. This enriches patches and their ports with the trigger and signalling functionality. Of course patches could spawn threads which will read from input ports endlessly, providing the same functionality as using a listener on an input port. But the listener approach saves resources and seems easier to comprehend.

The trigger labeled input of node B (see Figure. 3.4) can be implemented by using a listener. No extra thread for reading the input port must be spawned, thus saving dispensable overhead.

3.2.3 Connectivity Scheme

OpenInsightExplorer only allows connections between exactly ***one*** input and ***one*** output port. Other dataflow languages, in contrast, occasionally feature arbitrary connection and relation schemes. Figure 2.5 illustrates a dataflow network with ***1:N*** support, for example. The output arcs from the nodes A and C are split up and their branches are connected to different nodes.

The decision, to prune the connection rules to ***1:1*** relations only, is based on several reasons. First, OpenInsightExplorer supports previously introduced request scheme. An optionally registered listener function of an output port would have to determine from which branch the

request came from, before sending a result exclusively down that branch. This would have caused only a minor modification of the source code.

Secondly if one input port is connected to multiple output ports (a ***N:1*** relation), such requests would have to be invoked in all of the multiple output port nodes. Their result tokens would be received in an arbitrary sequence at the input port which would possibly compromise computations. This would also be the case in a pure ***data driven*** architecture which supports ***N:1*** connection relations which do not feature any ***on demand*** support.

Therefore the decision was made to limit the connection scheme to ***1:1*** relations. If the output of a patch is required as input for more than one other patch, OpenInsightExplorer users can employ patches which function as ***gate*** nodes (see Section 2.2.2). These special patches are capable of controlling the dataflow in various ways. Some of them duplicate the received input tokens and send these through their multiple output ports. But OpenInsightExplorer is equipped with a feature that makes the usage of such special gate patches unnecessary in some cases. This feature is called ***growing ports*** and is described in detail in the following section.

3.2.4 Growing Ports

In some cases, it is desirable to dynamically add or remove ports to patches. For example, 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 (see Figure 3.6). To give another example, the slider patch should provide its values to more than one other patch by adding extra output ports. Adding extra output ports prevents the need for special gate patches.

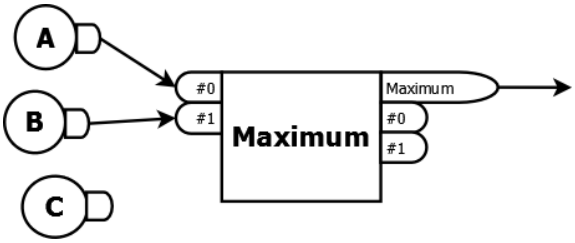
OpenInsightExplorer introduces a new unique feature that adds a lot of flexibility and can be used to avoid the occasional need of some special gate patches. Nodes can be implemented to add and remove ports dynamically. In the visual representation of nodes in the OpenInsightExplorer editor (see Figure 3.7), some ports will feature specific add and remove icons. By clicking on those icons, the node will add or remove ports dynamically.

The ports of nodes are organized in trees. These trees, one for the input and another one for the output ports, can be altered dynamically via the growing port mechanism (see Section B.5). A more detailed description to this mechanism and example screenshots are given in the ***User's Guide*** (see Section A.6.4) and the ***Programmer's Guide*** (see Section B.5) .

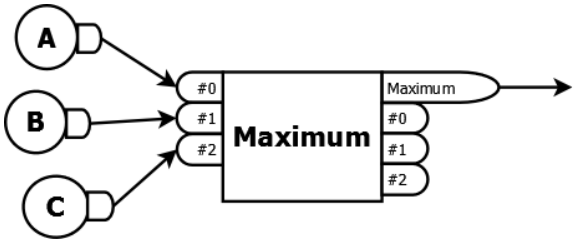
3.2.5 Data Types and Side Effects

One of the objectives of OpenInsightExplorer is to be a rapid prototyping language for a broad range of various visualizations. To accomplish this target, the language must not be bound to a predefined set of data types. Programmers must be free to use and introduce new data types to their projects, since different visualizations may build upon different data types.

This decision has a drawback. OpenInsightExplorer loses the ability to be side-effect free, which the pure dataflow model requires (see Section 2.2.1). Developers can introduce data types that hold references to objects. Accessing a referenced object by different nodes of a dataflow



(a)



(b)

Figure 3.6: Schematic diagram of a two value maximum node (a) and the through growing ports extended version (b).

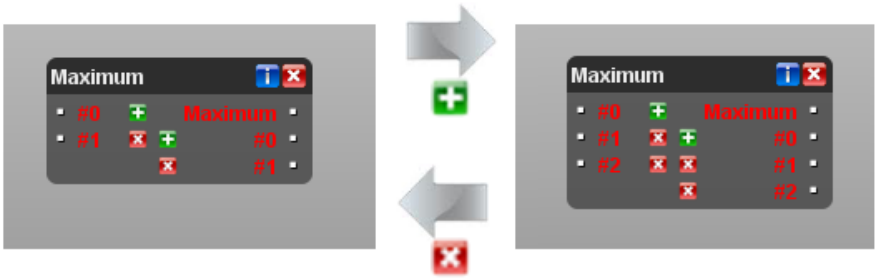


Figure 3.7: Screenshot of OpenInsightExplorer depicting the maximum node and a version of it with an additional port.

network concurrently can cause side-effects, because the manipulation happens on global data. This is a small price to pay for the flexibility which is gained through this design decision (see Section 5.2 for an example). Also some other state-of-the-art languages, e.g., Quartz Composer are not side-effect free and even tied down to a handful of native data types.

This chapter summarized the features of OpenInsightExplorer and all the design decisions which were made for it. The following chapter deals with the actual implementation of the framework and its features.

CHAPTER 4

Implementation

This chapter addresses the implementation of the OpenInsightExplorer framework. It describes how patches and ports and the dynamic loading process of patches and libraries are implemented. Furthermore this chapter contains a section which deals with hiding the frameworks internals from the developers and providing a clean programming interface.

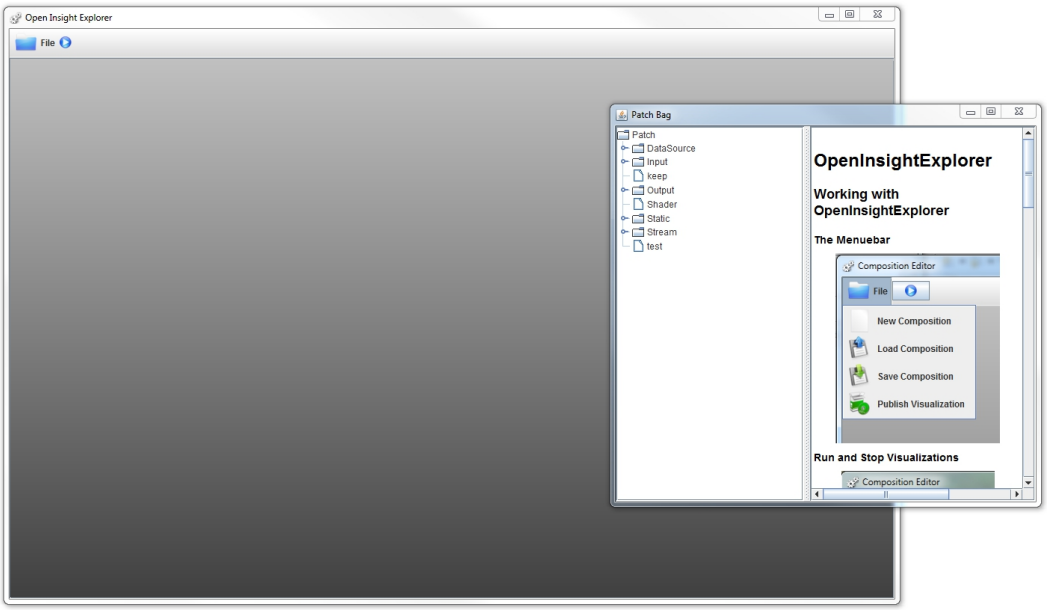


Figure 4.1: The visual editor of OpenInsightExplorer.

4.1 General Structure of the Framework

OpenInsightExplorer is a stand-alone Java application compiled to a ***jar*** file. It is started by double clicking on the jar file. After startup users see the graphical editor of the framework, the so called ***workbench*** (see Figure 4.1). Within this editor users can program visualizations by dragging patches into the workbench and connecting them together. This process is described in detail in Appendix A.1 of this thesis.

The whole framework is designed to follow a modular concept. OpenInsightExplorer itself only provides the visual editor and all necessary functionality to execute a programmed visualization. All other functionality is provided by patches, which are described in detail in the following section.

4.2 Patches

Every visualization made with OpenInsightExplorer consists of several patches, which represent certain stages of the visualization pipeline. The functionality of OpenInsightExplorer can be extended by developing new custom patches. They are designed to work as ***black boxes***. All patches implement the same Java interface, called ***Patch***. It defines all necessary functions which OpenInsightExplorer needs to graphically render patches, display vital information about them and let users connect them to a visualization pipeline. Building on only one interface raises the modularity of the framework. A detailed description of the patch interface can be found in the Appendix B.2 of this thesis.

4.2.1 Loading Patches at Runtime

To add custom developed patches developers must only copy the compiled files of it into the appropriate directories. The binary files ("***.class***") are stored in the ***/Patch*** directory. The files can even be placed in sub directories, reflecting the naming of the package. Therefore every added patch descends from the root package called ***Patch***.

OpenInsightExplorer scans at the start all files and sub directories of the patch directory. It adds every class it finds which implements the ***Patch*** interface. This process is described in detail in the introduction of the ***Programmer's Guide*** (see Section B.1). Since the software finds and instances patches with the ***Class.forName()*** method and the standard ***ClassLoader*** the classpath must contain the current working directory.

This is achieved by setting the classpath in the manifest file of the OpenInsightExplorer.jar (see Listing 4.1). Additionally the manifest tells the JRE where it can find the main class in the jar file. This adds an auto-run capability to the jar file. This means that most operating systems will start OpenInsightExplorer by double clicking its jar file. User are not bothered any more to start the software in a command prompt.

```
Manifest-Version: 1.0
Main-Class: OpenInsightExplorer.OpenInsightExplorer
Class-Path: .
```

Listing 4.1: The Manifest.mf file of the OpenInsightExplorer.jar

4.2.2 Loading Jar Files at Runtime

Some patches will need to access additional libraries, stored in separate jar archives to operate flawlessly. Furthermore, different versions of a library may exist depending on which operating system OpenInsightExplorer is being executed. Sometimes even the computer architecture must be taken into account. For example, if a patch performs ***OpenGL*** rendering, the ***Jogl*** library must be loaded beforehand.

To address this problem, the OpenInsightExplorer installation contains a sub directory called ***lib***. In that folder all additional multi-platform and architecture independent jar archives are stored. Also the Dynamic Link Libraries (DLL) they may access are placed in this directory. This ***lib*** directory contains two sub directories, an ***amd64*** and an ***x86*** named folder. The ***amd64*** folder contains all the 64-bit libraries and DLLs and the ***x86*** folder the 32-bit ones.

If a patch relies on a library which is not a default element of the OpenInsightExplorer software package, it can be shipped with the patch all together. Patches should place their libraries in the ***lib*** directory and all jar archives are loaded which the framework finds in that directory at startup. Additionally the architecture of the computer is determined and only content of the specific architecture sub folder is loaded too.

Usually all jar archives which may be accessed by a program must be specified to the JRE before program execution. This is done by adding them to the classpath. Since the software does not know which libraries were installed before execution of OpenInsightExplorer these jar archives must be loaded manually at runtime. The Listings 4.2 and 4.3 depict the needed source code for this process.

```
public static void loadAllJarsInDir(String dirname){
  File f= new File(dirname);
  if(f.exists())
    for(String s: new File(dirname).list(
      new FilenameFilter() {
        public boolean accept(File dir, String name) { return name.endsWith("
            .jar"); }
      } )) addJar(dirname+s);
}
```

Listing 4.2: Scanning a directory for jar archives.

```
@SuppressWarnings("deprecation")
public static void addJar(String s) {

  URLClassLoader sysloader = (URLClassLoader)ClassLoader.getSystemClassLoader
      ();
  Class<?> sysclass = URLClassLoader.class;

  Class<?>[] parameters = new Class[]{URL.class};

  try {
    Method method = sysclass.getDeclaredMethod("addURL", parameters);
    method.setAccessible(true);
    method.invoke(sysloader,new Object[]{ new File(s).toURL() });
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

Listing 4.3: Loading a jar file at runtime.

Some libraries access Dynamic Link Libraries during execution via JNI. Of course these files must be found by Java during execution.
Depending on the operating system the JRE tries to find these library files in different locations. On Windows for example the JRE will search through all directories which are set in the ***PATH*** and ***USRPATH*** variables and finally in the current working directory per default. On Unix based operating systems the library path is specified in the ***LD_LIBRARY_PATH*** variable. Users can add locations where to search for DLLs via setting the ***java.library.path*** at startup of the JRE. It is not possible to set this parameter within the manifest file of a jar.

OpenInsightExplorer determines the architecture of the platform it runs on at startup and adds the appropriate directories at runtime via the ***System.setProperty()*** method (see Listings 4.4 and 4.5).

```
public static void addDir(String s) {
  try {
    Field field = ClassLoader.class.getDeclaredField("usr_paths");
    field.setAccessible(true);

    String[] paths = (String[])field.get(null);
    for (int i = 0; i < paths.length; i++) {
      if (s.equals(paths[i])) {
        return;
      }
    }

    String[] tmp = new String[paths.length+1];
      System.arraycopy(paths,0,tmp,0,paths.length);
      tmp[paths.length] = s;

      field.set(null,tmp);

      System.setProperty("java.library.path", System.getProperty("java.
          library.path") + File.pathSeparator + s);

    } catch (Exception e){
      e.printStackTrace();
      System.exit(1);
    }
}
```

Listing 4.4: Adding a directory to the java.library.path at runtime.

```
private final String filesep=System.getProperty("file.separator");
private final String userdir=System.getProperty("user.dir");

String arch=System.getProperty("os.arch");
String lib=userdir+filesep+"lib"+filesep;
String lib_arch=lib+arch+filesep;

OpenInsightExplorer.addDir(lib);
OpenInsightExplorer.addDir(lib_arch);
OpenInsightExplorer.loadAllJarsInDir(lib);
OpenInsightExplorer.loadAllJarsInDir(lib_arch);
```

Listing 4.5: Loading libs and architecture dependent libs.

4.3 Ports

As mentioned before in chapter 3 patches exchange data through their input and output ports. This message passing follows the concept of the ***producer consumer problem***. There are patches which produce data and others which will consume. Since patches should run concurrently the communication between them must be synchronized. The consumer can only process data which was sent by the producer so far. If no more data is available, the execution of the consumer must

be suspended until new data has arrived. Also the producer should not wait until the consumer has read this data.

The solution to this problem is to let the producer write the data to a buffer from which the consumer reads. OpenInsightExplorer uses a simple ring buffer implementation with a fixed buffer size at runtime. Since the buffer has a fixed size no performance penalties can happen by reallocating additional storage space.

A producer patch thread gets suspended if the ring buffer is totally filled leaving no space for additional data. On the other hand a consumer thread gets halted when there is no more data available to be read.

This design provides the opportunity to enhance the functionality of this dataflow language. In OpenInsightExplorer the ring buffer is implemented to send signals to ***listeners*** in the case of a buffer under-run. Producer patches can be implemented so that they only provide data on request, following the ***on demand*** architecture. Additionally patches can register listeners on receiving data, implementing a ***trigger*** functionality.

4.3.1 Custom Data Types

Some visual dataflow programming languages support only a small set of built-in data types which nodes can exchange, e.g., Quartz Composer (see Section 2.4.4). Working with only a limited set of data types would make the development of visualizations (which may need more complex data structures and data types) become a very tedious task. Needed data structures and types must be emulated with the given limited set. Languages which do not support custom data types, such as Quartz Composer, often provide a specific data type which allows to circumvent this limitation. This data type allows to combine basic data types to a structure and even add other structures to it.

This approach has an essential drawback. Accessing members of such a data structure can only be done in a non type-safe way. OpenInsightExplorer also provides such a structure data type to work with. It proved to add a lot of flexibility to a dataflow programming based language.

OpenInsightExplorer is designed in such a way that ports can exchange any data type a developer may desire. Furthermore this design concept enriches OpenInsightExplorer to use every class the standard Runtime Library (rt.jar) of Java offers and classes of other libraries too. It even gives developers the opportunity to develop their own data type classes specifically fitted to the given type of visualization.

4.3.2 Generic Port Class

To enable custom data types the classes which represent different ports must be generic so developers can specify which data type the port should send or be able to receive. There is only one reason why a port should be aware of the data type it was constructed with: type-safety. The port must know its generic type at runtime, so it can validate a connection to another port providing the same functionality as ***typed arcs*** (see Section 2.3). Both must be constructed with compatible data types.

During the development of OpenInsightExplorer it transpired that this is almost an impossible task to accomplish. Usually Java only keeps track of the generic type of a class at

compile time and not, as necessary, at runtime with ***one exception***. The reason for this is that Java compilers use a technique called ***type erasure*** to be downwards compatible with older JREs. In this process all information about the generic related type parameters and arguments is removed within a standard class or method. At runtime its impossible that a class with a generic parameter can determine the type of the parameter since the compiler replaces all generic types by the type ***Object***.

One solution attempt to this problem is to use a technique called ***type tokens*** which turned out to be insufficient. The following example depicts the reason why: To the constructor of a generically typed class an extra parameter is added, which is unveiling the actual type of the generic parameter (see Listing 4.6).

```
public class SimpleTypeTokenExample<E> {
  private Class<?> type;
  private E data;

  public SimpleTypeTokenExample(Class<E> type){
    this.type=type;
  }

  public set(E data) {
    this.data=data;
  }

  public E get() {
    return this.data;
  }

  public Class<?> getType() {
    return this.type;
  }

  public printType(){
    System.out.println("my type is "+getType().toString());
  }
}
```

Listing 4.6: Simple type token example.

This technique sounds promising, since it solves the problem of determining the actual type of the generic parameter. But as depicted in Listing 4.7, a mistake can easily happen and the wrong type gets assigned. Also the programmer must specify twice which generic data type a class has which seems fairly clumsy.

```
SimpleTypeTokenExample<Integer> stte=new SimpleTypeTokenExample<Integer>(
    Integer.class);

stte.set(10);
int myint=stte.get()+20;

stte.printType();

SimpleTypeTokenExample<String> stte_failure=new SimpleTypeTokenExample<String
    >(Integer.class);

stte_failure.set("hello");
String mystring=stte_failure.get()+" world";

stte_failure.printType();

// obviously wrong
if(stte.getType().equals(sttefailure.getType())
  System.out.println("they have the same type");
```

Listing 4.7: Simple type token fail example.

As mentioned earlier Java removes all generic type information at compile time from standard classes and methods. Only ***anonymous inner classes*** will keep their generic type parameter information even after compile time.

Reflection can be used to exploit the fact that anonymous classes will keep their generic type information even after compile time. This solution enables to get rid of type tokens and programmers must not specify twice the type of data the port should operate with. The only drawback of this technique is that it must be ensured that ports are constructed as anonymous inner classes. This seems rather simple, because a custom Exception (***AnonymousException***) can be thrown in the case a port is not constructed as an anonymous class (see Listing 4.8).

```
public SimpleGenericPort<E>{

  Class<?> type;

  public SimpleGenericPort() throws AnonymousException {
    try{
      type=(Class<?>)((ParameterizedType)  this.getClass().
          getGenericSuperclass()).getActualTypeArguments()[0];
    } catch (java.lang.ClassCastException E) {
      // a ClassCastException is thrown, if the class was not constructed as
          an anonymous inner class
      throw new AnonymousException("Simple Generic Ports must be constructed
          anonymously and with a generic type parameter");
    }
  }

  public void send(E data) {
    //... code that will send data to a connected port ...
  }

  public E get(E data) {
    //... code that will return received data ...
  }

  public void connect(SimpleGenericPort<?> port) throws ConnectionException {

    // check if both data types are compatible
    if(!this.getType().equals(port.getType())) throws new ConnectionException
        ("Data types are incompatible");

    // ... connection code ...
  }
}

...

try {
  SimpleGenericPort<Integer> myintport=new SimpleGenericPort<Integer>(){};

  // this does not compile
  SimpleGenericPort<String> mystringport=new SimpleGenericPort<Integer>(){};

  // this will throw an Exception at runtime ...
  SimpleGenericPort<Integer> noanonymousport=new SimpleGenericPort<Integer>()
      ;

} catch (AnonymousException ae) {
  //  thrown if the Port is not constructed as an anonymous inner class
}
```

Listing 4.8: Implementation of a simple generic port.

4.3.3 Stream Ports

So far ports can be generated which operate on any desired data type. These ports are able to send only individual chunks of data. Some visualizations will need data structures, which will allow to group individual data together.

For example, a file reader patch reads the content of a file, line by line, and sends this information to another patch. The second patch should be able to realize the start of the whole record of lines and the end of it. It maybe counts the number of lines the file consists of.

The solution to this problem should not subvert one of the basic design principles of dataflow programming: nodes should work concurrently. Therefore sending an array of all lines seems suboptimal. The second patch can only start working as soon as the file reader patch has read the whole content of the file. Even if the second patch only wants to know the first line of the file, it has to wait until the first patch processed the whole file. In this case Morrison's ***token stream*** model provides a suitable alternative (see section 2.3).

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. These starting and end tokens are called ***open*** and ***close*** brackets [26], which will surround the actual data. As illustrated in Section 4.3, ports can be implemented to follow the producer consumer problem. Output ports represent producers and input ports consumers (see Listing 4.9).

```
// the Producer
public class OutputPort<E>{
  // Connected is the InputPort to which this Output Port is connected to
  public InputPort<E> Connected;

  public void send(E data){
    // send a data element to the Input Port "Consumer"
    Connected.receive(data);
  }
}

// the Consumer
public class InputPort<E>{
  E[] ringbuffer=new E[buffersize];

  public void receive(E data){
    // here comes code which adds data to the ring buffer
    ringbuffer[writepos]=data;
  }

  public E get(){
    // returns an element from the ring buffer
    return ringbuffer[readpos];
  }
}
```

Listing 4.9: Example implementation of ports that follow the producer consumer problem.

Obviously, ports which should operate on streams must be able to send and receive open and close tokens additionally to the real data elements. These tokens and data elements must be added to the ringbuffer in sequence as they were received. Apparently these bracket tokens cannot be added to a ringbuffer designated to a generic data type.

To solve this issue data elements are packed within so called ***Information Packet***s or shortly ***IP***s [26]. They may contain actual data elements or else they represent these open and close bracket tokens instead (see Listing 4.10).

```
package OpenInsightExplorer.IP;

import OpenInsightExplorer.Stream.Close;
import OpenInsightExplorer.Stream.Open;

public interface Ip <E>{
  E get() throws Open, Close;
}
```

Listing 4.10: The information packet (IP) interface.

All three kinds of information packets will implement this generic ***IP*** interface. Packets who contain actual data, will return the data element which they contain on calling the ***get()*** method. The bracket packets are also implementing the interface, but will not carry any data itself. In the case their ***get()*** method is called they will either throw an ***Open*** or a ***Close*** exception depending on which control token they should represent.

Some extensions to the port implementation must be made to equip it with streaming capabilities. Two additional send methods are added to the ***StreamOutputPort*** in comparison to the basic ***OutputPort***, which will send the control token IPs. Real data elements are packed into a ***IPData*** packet before sending to a connected ***StreamInputPort***. The ***StreamInputPort*** class contains a ringbuffer which holds IPs of the generic type the port should receive. Additionally the ***get()*** method throws now ***Open*** and ***Close*** exceptions (see Listing 4.11).

```
// the Producer
public class StreamOutputPort<E>{
  // Connected is the InputPort to which this Output Port is connected to
  public StreamInputPort<E> Connected;

  public void send(E data){
    // send a data element contained within an IPData packet to the Input
        Port "Consumer"
    Connected.receive(new IPData<E>(data));
  }

  public void sendOpen(){
    // sends a control token Open to "Consumer"
    Connected.receive(new IPOpen<E>());
  }

  public void sendClose(){
    // sends a control token Close to "Consumer"
    Connected.receive(new IPClose<E>());
  }

}

// the Consumer
public class StreamInputPort<E>{
        IP<E>[] ringbuffer=new IP<E>[buffersize];

  public void receive(IP<?> data){
    // here comes code which adds an IP to the ring buffer
    ringbuffer[writepos]=data;
  }

  public E get() throws Open, Close{
    // returns data or in the case of a stream control token throws an
        exception
    E data=ringbuffer[readpos].get();
    return data;
  }
}
```

Listing 4.11: Implementation of ports with streaming capabilities.

Throwing exceptions in the case a ***StreamInputPort*** reaches a control token becomes very handy. Developers do not need to check IPs if they are control tokens or unpack data IPs. They only have to call the ***get()*** method and handle the ***Open*** and ***Close*** exceptions. The following code snippets demonstrate the streaming mechanism (see Listing 4.12 and 4.13. This example features the file content reader and the line counter mentioned in the beginning of the section 4.3.3.

```
StreamOutputPort<String> output=new StreamOutputPort<String>();

// output gets connected to the StreamInputPort

FileReader fr = new FileReader("MyFile.txt");
BufferedReader br = new BufferedReader(fr);
String s;

output.sendOpen();
while((s = br.readLine()) != null)  Output.send(s);
output.sendClose();

fr.close();
```

Listing 4.12: Streaming the content of a file line-by-line and signaling the start and the end of the file.

```
StreamInputPort<String> input=new StreamInputPort<String>();

// input gets connected to the StreamOutputPort

int linecount=0;
while(true){
  try{
    System.out.println("line #"+(linecount++)+":"+input.get());
  } catch (Open o){
    System.out.println("File has started");
    linecount=0;
  } catch {Close c){
    System.out.println("File ended and has "+linecount+" lines");
  }

}
```

Listing 4.13: Reading the stream and outputting the content of the stream to the console.

4.4 Hiding the Framework Implementation

The previous sections described how to create streaming and non-streaming ports which can operate on any desired data type since they get instanced with a generic type parameter.

One of the design goals of OpenInsightExplorer is that it should be fairly easy to extend even for average skilled developers. At a deeper look on the current state of the port classes, programmers are confronted with many methods and functions which should not be accessed. The access should be limited to the framework itself. For example, the ***receive()*** method of the ***InputPort*** and ***StreamInputPort*** classes must be declared public because output ports call these methods on sending data. Patch developers on the other hand should not even realize these methods exist since they may get called by accident. Developers should only be confronted with the minimal set of functions and methods they really need to use a port. The real implementation should be hidden from them.

One possibility to accomplish this is to apply the ***cheshire cat*** programming pattern, often also called ***pimpl idiom*** [35]. OpenInsightExplorer uses this pattern and adapted it a bit since it does not provide the entire desired functionality.

4.4.1 Hiding Behind Proxy Classes

The basic idea of the cheshire cat programming pattern is to hide the class that really implements the whole functionality behind proxy classes. Each proxy class can even provide different methods and functions, depending on how the real implementation should act like. These methods in the proxy classes will only redirect calls to the real implementation. Therefore only one real implementation for all kinds of ports is needed. It has all the functionality for input and output and of course streaming, the ***RealPort***.

```
public class RealPort<E> {

  boolean isInputPort;
  RealPort<E> Connected=null;
  IP<E>[] ringbuffer=new IP<E>[buffersize];

  public RealPort(boolean isInputPort) throws AnonymousException {
  /* code to find out which generic type the Port is instanced of
  * see section "Circumvent Type Erasure"
  */
    this.isInputPort=isInputPort;
  }

  public void Connect(RealPort<E> port) throw Exception{ /* Connect this Port
       with another Port */ }
  public void receive(IP<E> ip) { /* receives IPs */

  public void send(E data){ /* sends data to a connected Port */ }
  public void sendOpen(){ /* sends an Open token */ }
  public void sendClose(){ /* sends a Close token */ }

  public E get(){ /* returns received data */ }
  public E getStream() throws Open, Close { /* returns received stream data,
      see section "'Stream Ports"' */ }

  public void setName() { /* set the Name of the Port */ }
  public String getName() { /* get the Name of the Port */ }

}

public abstract class Port{
  public abstract Port setName();
  public abstract String getName();
}
```

Listing 4.14: Implementation of the ***RealPort*** and the abstract ***Port*** class.

All proxy classes extend an abstract class called ***Port***. This abstract ***Port*** class defines a minimum set of methods each proxy must at least implement to be understood as a port. For illustration purposes in a reduced example the functionality is limited to ***get()*** and ***set()*** methods for the name of a port (see Listing 4.14).

On the construction of the proxy class ***InputPort***, it instantiates a ***RealPort*** and stores its reference in the variable ***hide***. This variable hide is declared private therefore developers cannot access the whole functionality of the ***RealPort***. Since it is not an output port and has no streaming capability, it only provides the one method which is necessary to function as an input port, the ***get()*** method (see Listing 4.15).

```
public class InputPort<E> extends Port {

  private RealPort<E> hide;

  public InputPort<E>() throws AnonymousException {
    // tell the RealPort that it is an Input Port
    this.hide=new RealPort<E>(true);
  }

  // set Name and return a self reference, overriding the "Port setName()"
  public InputPort<E> setName(String name) {
    hide.setName(name);
    return this;
  }

  public String getName(){
    return hide.getName();
  }

  // get a received data element
  public E get(){
    return hide.get();
  }

}
```

Listing 4.15: Implementation of the ***InputPort*** proxy.

The output port proxy is implemented in a similar way, with two exceptions. It tells the ***RealPort*** constructor to act as an output port by setting the ***isInputPort*** parameter to ***false***. Furthermore, it has no ***get()*** method and provides a ***send()*** method instead (see Listing 4.16).

```
public class OutputPort<E> extends Port {

  private RealPort<E> hide;

  public OutputPort<E>() throws AnonymousException{
    // tell the RealPort that it is an Output Port
    this.hide=new RealPort<E>(false);
  }

  // set Name and return a self reference, overriding the "Port setName()"
  public OutputPort<E> setName(String name) {
    hide.setName(name);
    return this;
  }

  public String getName(){
    return hide.getName();
  }

  // send data
  public OutputPort<E> send(E data){
    hide.get();
    return this;
  }

}
```

Listing 4.16: Implementation of the ***OutputPort*** proxy.

Up to now the implementation follows exactly the cheshire cat pattern. Developers cannot access the real implementation of a port, the ***RealPort***, with their reduced proxy interfaces. Unfortunately the framework is also unable to deduce the hidden ***RealPort*** behind a given proxy class pointer. But OpenInsightExplorer must be able to unveil it: For example, the ***Patch*** interface contains the methods ***getInputPorts()*** and ***getOutputPorts()*** which only return pointers to proxy ports. To connect input ports and output ports of patches the framework must access the ***RealPort*** hidden behind these proxies.

Apparently an abstract method to the abstract ***Port*** interface could be added forcing all proxies to implement it which will return this reference. But this destroys all the efforts to hide the implementation of a patch from developers, because they could call this method as well as gaining access to the implementation.

A better solutions is to add a private static final ***HashMap*** to the ***RealPort*** class. Also the constructor gets modified with an additional parameter which holds the reference to the proxy. The constructor adds every time it is called an entry to the ***HashMap***, with the proxy as key and itself as value. The ***RealPort*** class implements a public static ***get()*** method, which returns a reference to the ***RealPort*** object if it is called with the proxy port as parameter.

```
public class RealPort<E> {

  private static final WeakHashMap<Port, WeakReference<RealPort<?>>> map=new
      WeakHashMap<Port, WeakReference<RealPort<?>>>();
  private Port myProxy=null;

  public static RealPort<?> get(Port port){
    WeakReference<RealPort<?>> wr=RealPort.map.get(port);
    return (wr!=null?wr.get():null);
  }

  public RealPort(Port<E> proxy, boolean isInputPort) throws
      AnonymousException {
  /* code to find out which generic type the Port is instanced of */
    this.isInputPort=isInputPort;
    this.myProxy=proxy;
    map.put(proxy, new WeakReference<RealPort<?>>(this));
  }

  ...

  // no more references to this RealPort and the proxy? remove HashMap entry
  public void finalize() throws Throwable{
    RealPort.map.remove(this.myProxy);
  }

  ...
```

Listing 4.17: Using a ***WeakHashMap*** to reference all ***RealPort*** objects.

More precisely a ***WeakHashMap*** is used instead of a ***HashMap***. WeakHashMaps can hold references to objects which are not visible to the Garbage Collector, a ***WeakReference***. As soon as all other references are nullified, the objects will be finalized (see Listing 4.17).

Patch developers are not aware of the ***RealPort*** class. The framework hides this class successfully behind proxy classes and itself can still access it (see Listing 4.18).

```
// create a Console Patch and access the RealPort of the InputPort root node
Patch p=(Patch) new Console();
p.init();
System.out.println(RealPort.get(p.getInputPorts()).toString());
```

Listing 4.18: The framework can still access the ***RealPort*** object by using the classes' ***get()*** method.

4.4.2 Proxy Port Listeners

Many ports in OpenInsightExplorer give programmers the ability to set ***listeners***. These listeners are interfaces which declare a reference to callback functions that the port calls in the case a specific event occurs.

For example, it is possible to set an ***InputPortReceiveListener*** for an ***InputPort***. When the port receives data it will call the ***Receive()*** method of listener with a reference to itself

as a parameter (see Listing 4.19). This gives the programmer the ability to use only one ***InputPortReceiveListener*** for multiple ***InputPorts*** if it is desired to distinguish the port on the basis of the given reference. Therefore developers may not create for each port an ***InputPortReceiveListener*** for itself and can furthermore access the referenced ***get()***-method immediately.

Since there are different input port classes with different capabilities (different ***get()*** methods) they all have a suitable receive listener of their own. For the ***InputPort*** an ***InputPortReceiveListener*** interface is available which declares a ***Receive()*** method with an ***InputPort*** as parameter. On the other hand for the ***StreamInputPort*** the ***StreamInputPortReceiveListener*** is appropriate since it declares the ***Receive()*** method with a ***StreamInputPort*** as reference parameter (see Listing 4.19).

```
// method in InputPort class
public InputPort<E> setReceiveListener(InputPortReceiveListener<E> listener){
  new InputPortReceive(listener, this, hide);
  return this;
}

...

// Implementation of the Receive Listener proxy
public class InputPortReceive implements CallBackReceive {

  RealPort<?> hide;
  InputPortReceiveListener<E> listener;
  InputPort<E> port;

  public InputPortReceive(InputPortReceiveListener<E> listener, InputPort<E>
      port, RealPort<?> hide){
    this.listener=listener;
    this.hide=hide;
    this.port=port;

    if(listener==null) hide.setReceiveListener(null); else hide.
        setReceiveListener(this);
  }

  public void Receive() {
    listener.Receive(port);
  }

}
```

Listing 4.19: Example implementation of the listener functionality.

4.4.3 Redirecting Exceptions

In Section 4.4.1 it was mentioned that the ***RealPort*** class may throw several exceptions. For example, if the port is not constructed as an anonymous class. Every exception gives the developer the opportunity to analyze its stacktrace, following up to the point where it had been thrown.

This would reveal the hidden ***RealPort*** class and that should be avoided. Every exception that the ***RealPort*** class can throw, manipulates its stacktrace in the constructor (see Listing 4.20). They remove all elements at the top of their stacks which could lead to the real causer, our hidden class. They keep the rest of the stacktrace untempered, giving the programmer the ability to find the line of code which caused the exception up to the point where he accessed the framework in the wrong way. For example, the top of an ***AnonymousException*** stacktrace references to the line of code where the developer constructs a port class like a normal class instead of being constructed as an anonymous class. All stacktrace elements were removed which would guide him to the line of code where the exception was thrown in reality.

```
package OpenInsightExplorer.Exceptions;

public class AnonymousException extends Exception{

  private static final long serialVersionUID = 1L;

  public AnonymousException(String s){
    super(s);

    int removecount=3;
    StackTraceElement[] stack=this.getStackTrace();
    StackTraceElement newstack[]=new StackTraceElement[stack.length-
        removecount];
    for(int i=0; i<newstack.length; i++) newstack[i]=stack[i+removecount];
    this.setStackTrace(newstack);
  }

}
```

Listing 4.20: Removing elements from the stacktrace.

4.4.4 Proxy Port Return Statements

Proxy port methods are designed to return a reference to themselves instead of being declared ***void***. This behaviour allows to construct and configure ports in one line of code (see Listing 4.21).

For example, it is possible to construct an ***InputPort*** and set its name, several listeners and even add some child ports in one compact statement. Sometimes a reference variable can be spared by using this self reference scheme. Of course this technique is optional and developers can still use reference variables to configure ports.

```
// written as one line code using self reference
try{
  adderInput.add(new InputPort<Integer>(){}.setName("Number").
      setReceiveListener(this).setRemoveListener(this));
} catch (AddException e) {}
catch (AnonymousException e) {}

// traditional way
try{
  InputPort<Integer> number=new InputPort<Integer>(){};
  number.setName("Number");
  number.setReceiveListener(this)
  number.setRemoveListener(this);
  adderInput.add(number);
} catch (AddException e) {}
catch (AnonymousException e) {}
```

Listing 4.21: Using self reference to configure a port in one line of code.

CHAPTER 5

Results

To evaluate the usability of the OpenInsightExplorer framework two example visualizations were implemented with it. The first example is a volume renderer and 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 basic volume renderer based on raycasting. A volume renderer displays a 2D projection of 3D discretely sampled data sets. These data sets are typically acquired by computer tomography- or a magnetic resonance imaging scanners. The volume usually gets sliced into a series of pictures of the same resolution, defining a regular volumetric grid. Each element of this grid, a so called ***voxel*** (volumetric element), represents a density value or other properties of the volume.

To generate a picture of the volume a camera position relative to the volume needs to be defined. Additionally a color gets assigned to each possible value of the volume. This mapping, from values to colors, is called ***transfer function***. The renderer can look up a color for each value in this transfer function.

All example renderers use the ***volume ray casting*** technique for image generation. A ray is generated for each image pixel, emanating from the eye point of the camera and passing through this image pixel. The volume gets sampled at regular intervals along this ray. For each sampled density value of the volume the renderer looks up the assigned color in the transfer function. A compositing function combines these gathered color samples to the final color of the image pixel. Using different composition functions results in different volume visualizations (e.g., first hit, accumulate, maximum intensity projection, etc.).

The goal was to use only off-the-shelf patches of the framework whenever possible for each example volume visualization. Only two custom patches (the ***TF Editor*** and the ***Volume File***

Loader patch) had to be implemented in addition because of their rather specific functionality. The examples use GPU hardware acceleration for image generation. This is achieved using GLSL fragment shader programs, which implement the ray sampling and composition functions. The first volume renderer generates x-ray like pictures of the volume (see Figure 5.1). This example composition contains only 20 patches in total (see Figure 5.2).

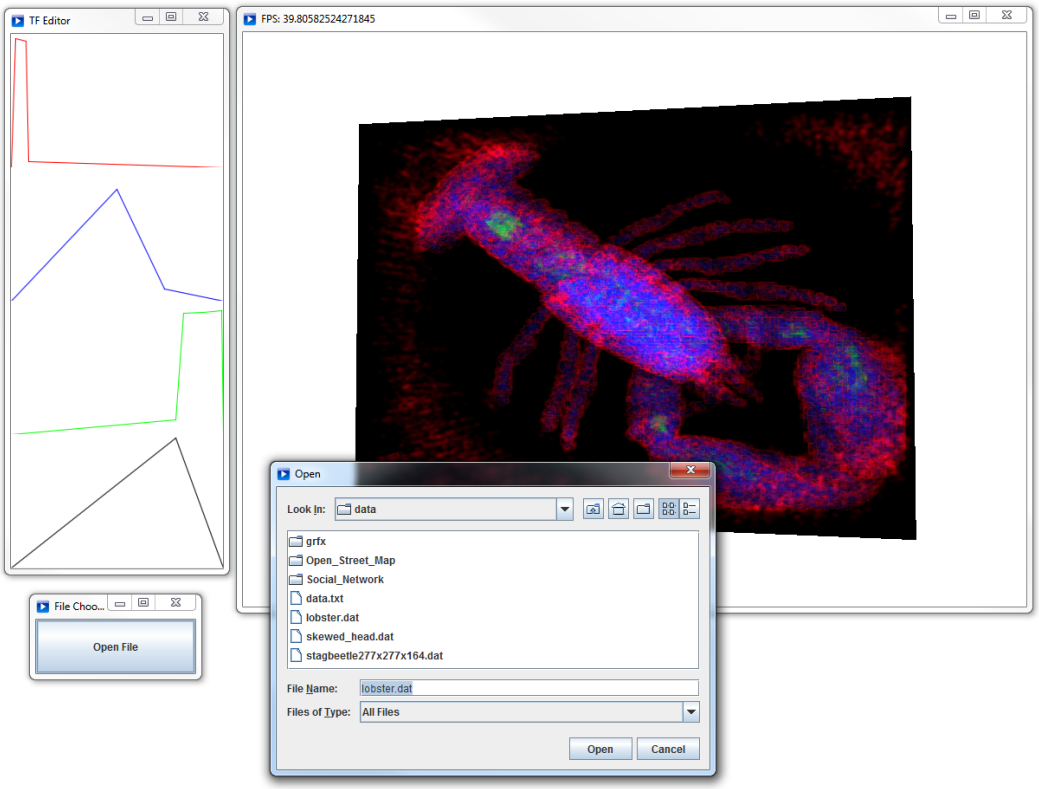


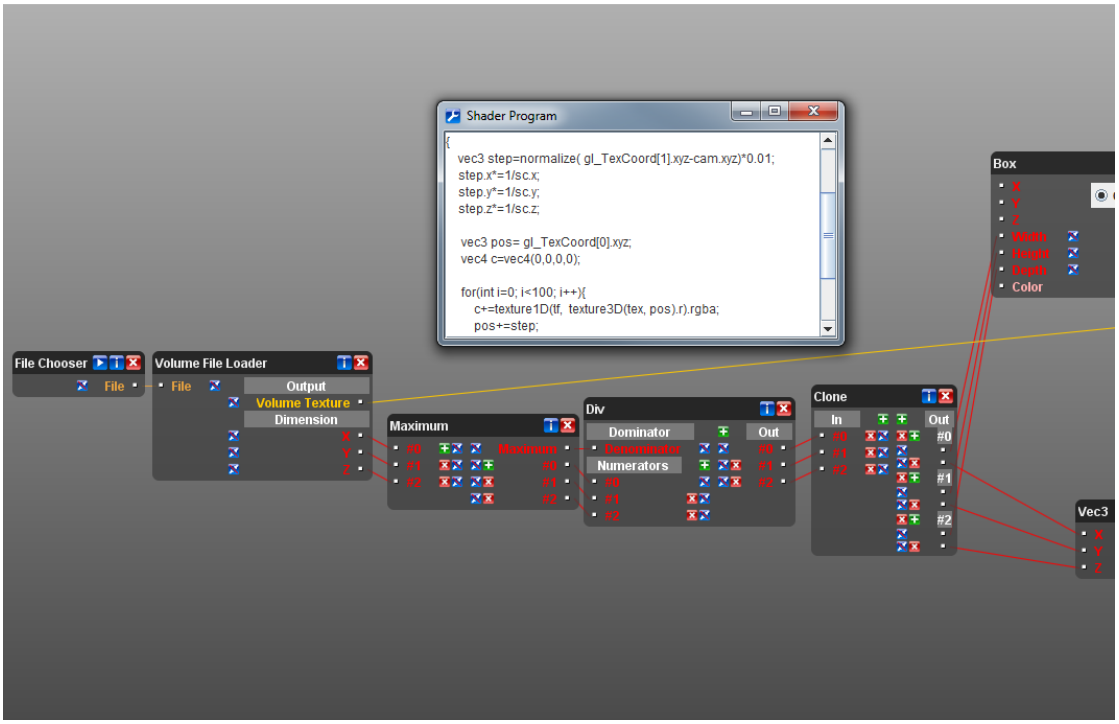
Figure 5.1: A screenshot of the GUI for the volume renderer. On the left is the transfer function editor depicted and in front the file open dialogue of the ***File Chooser*** patch is shown. The right window is the rendering output window, displaying the volume visualization.

All volume renderer compositions share the same visualization pipeline and therefore the same data acquisition step. For this processing stage a custom ***Volume File Loader*** patch was implemented. This patch takes a file handle as input, loads the volume data, converts it into a ***texture3D*** OpenGL texture and sends it to the next stage. Additionally it outputs the dimensions of the volume too. This patch is connected with the ***File Chooser*** patch, which allows to load different files during the visualization with an interactive file dialogue. All data analysis steps were omitted, but additional patches could be developed for this purpose (e.g., a patch that will generate a histogram of the volume or calculate the gradients of the volume and transforms them into texture3D for the renderer to use).

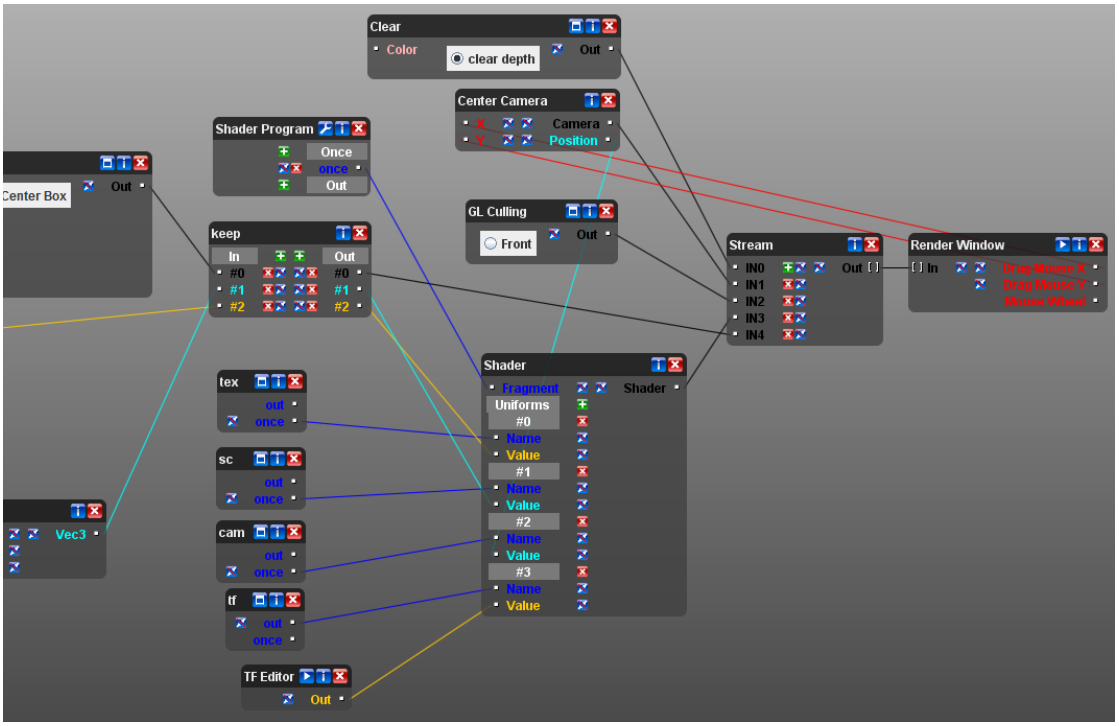
The user can filter and map color attributes to the density values of the volume with a transfer function editor (see Figure 5.1). The editor is provided by the ***TF Editor*** patch. This patch embodies the filter and mapping stages of the visualization pipeline. The ***TF Editor*** is the second patch that was specially designed and implemented for the volume rendering compositions. It provides a GUI where users can edit functions for each color channel (red, blue, green) and alpha and outputs a one dimensional OpenGL texture. The final rendering step of the pipeline is done by the ***Shader*** patch, which takes a string containing a GLSL fragment program as input. It compiles the program and executes the shader with the values provided to its other inputs (in this case the texture3D containing the volume, the transfer function texture and a three component vector with the eye position of the camera).

Based on the raycasting volume renderer, various other composition functions were implemented. Some of them provide additional functionality, e.g., they take the density gradients or light sources for shading into account. All those renderers originate from the same base version and are only slight modifications of it. Most of the time only the GLSL shader program was adapted with a different composition function. For some examples an additional patch (providing a light source position) was connected to the ***Shader*** patch.

Volume rendering techniques belong to the field of scientific visualization. A basic volume renderer was implemented to evaluate the OpenInsightExplorer's capabilities to use GPU hardware acceleration and to determine the overhead which the framework's dataflow execution implementation produces. The example renderer achieved high rendering framerates, which primarily depend on the performance of the used graphic card rather than the execution model. This example also supports OpenInsightExplorers modularity concept. Only two additional patches had to be developed. This was accomplished in a short timespan which was significantly shorter than implementing a volume renderer from scratch.



(a) Left half of the composition.



(b) Right half of the composition.

Figure 5.2: The composition of the raycasting volume renderer.

5.2 OpenStreetMap Visualization

The second example visualizes data from the ***OpenStreetMap (OSM)*** [29] 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 OpenStreerMap database.

5.3 The OpenStreetMap XML File Format

OpenStreetMap maps can be exported to special format XML-files. These files are build on only three simple elements: ***node***, ***way*** and ***relation***. Each element may have an arbitrary number of properties (a.k.a. ***tags***) which are key-value pairs (e.g., highway=primary).

- ***Node***
  Nodes are the basic underlying element of the OSM scheme. Nodes describe a single geo-spatial point with a pair of ***latitude*** and ***longitude*** coordinates. Nodes are commonly used to define a way. Furthermore a node can also be a standalone unconnected point, representing something like a telephone box, a pub, a place name label, or all kinds of other points of interest (POI). Standalone nodes have at least one tag.

- ***Way***
  A way is an ordered interconnection of at least 2 nodes that describe a linear feature such as a street or similar. Way elements are also used to define outlines of areas and buildings. They contain references to the nodes they consist of.

- ***Relation***
  Relations are used to group all sorts of elements together. They can contain references to nodes, ways and even other relations. For example a relation describes the whole transportation network of a city. This relation would have references to all relations which represent a single service line of the network. On the other hand those relations will reference each way they need to represent a service line.

Since OpenInsightExplorer allows users, as a feature, to introduce arbitrary data types, the example maps these elements to specially developed classes (see Listing 5.1). Each of them can hold the tags (the previously mentioned key-value pairs) of the XML elements and if specified, the references to other elements with ***ArrayLists*** within a ***HashMap***.

```
public class Node {
  public long ID;
  public double lat;
  public double lon;
  public HashMap<String, String> tags=new HashMap<String,String>();

  public Node(long ID, double lat, double lon){
    this.ID=ID;
    this.lat=lat;
    this.lon=lon;
  }

  public String toString(){
    String s="Node ID+"+ID+" lat:"+lat+" lon:"+lon+" \n";
    for(String k:tags.keySet()) s+="\t tag:"+k+" "+tags.get(k)+" \n";
    return s;
  }
}

public class Way {
  public long ID;
  public HashMap<String, String> tags=new HashMap<String,String>();
  public ArrayList<Node> nodes=new ArrayList<Node>();

  public Way(long ID){ this.ID=ID; }

  public String toString(){
    String s="Way ID+"+ID+" \n";
    for(String k:tags.keySet()) s+="\t tag:"+k+" "+tags.get(k)+" \n";
    return s;
  }
}

public class Relation {
  public long ID;
  public HashMap<String, String> tags=new HashMap<String,String>();
  public ArrayList<Relation> relations=new ArrayList<Relation>();
  public ArrayList<Node> nodes=new ArrayList<Node>();
  public ArrayList<Way> ways=new ArrayList<Way>();

  public Relation(long ID){ this.ID=ID; }

  public String toString(){
    String s="Relation ID "+ID+" \n";
    for(String k:tags.keySet()) s+=k+":"+tags.get(k)+" \n ";
    return s;
  }
}
```

Listing 5.1: The classes which map the OSM elements scheme.

For the data acquisition step of the OpenStreetMap (OSM) example visualization, a special patch called ***OSM File Reader*** was implemented. It loads all data from the OSM XML file and maps the elements to the data type classes. As soon as the complete file is loaded it streams the data through its output ports. The patch can be triggered to stream all elements again with its trigger labeled input ports. Various different filtering patches were implemented which allow users to select only parts of the data set that are of interest (e.g., buildings, trams, etc.). Three different patches (***Render Relations***, ***Render Ways*** and ***Render Nodes***) map the basic OSM elements to rendering primitives such as points and lines with the additional attributes color and size.

The stream of render primitives is sent through the ***OSM Camera*** patch to the final rendering window. This patch buffers the stream and adds a camera to it. If the user interacts with the rendering window (e.g., mouse dragging) the camera patch will update the position of the camera and send the previously buffered stream again. This enables scrolling and zooming into the rendering of the street map.

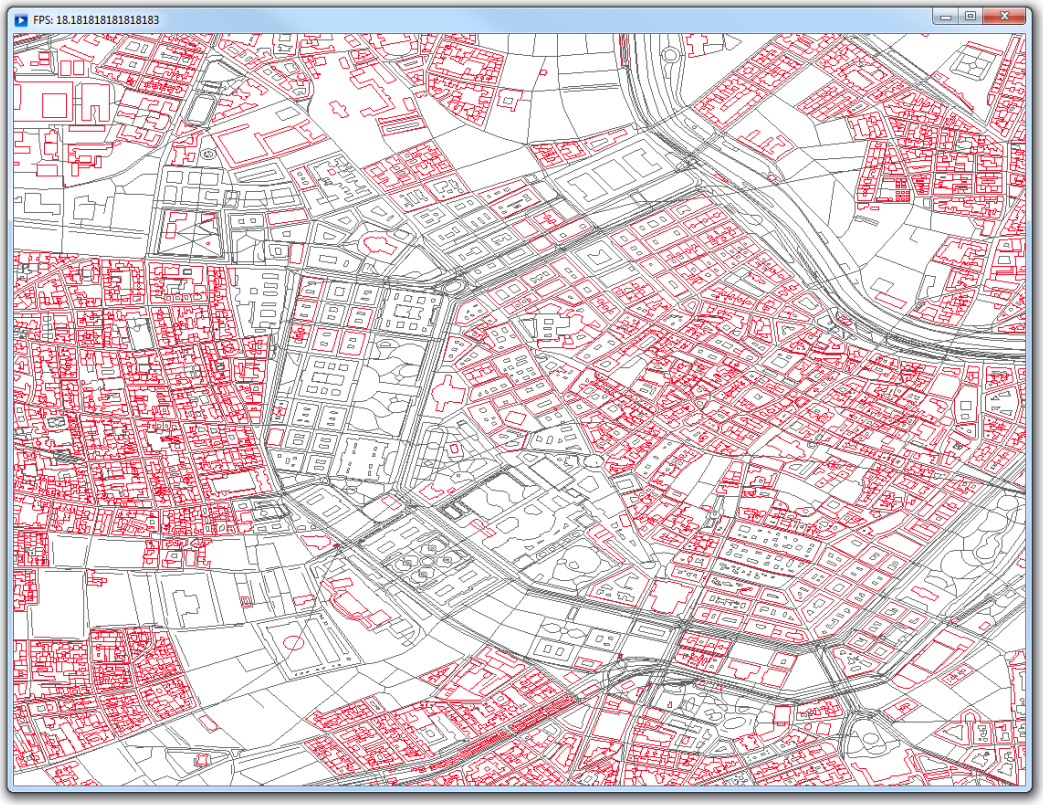


Figure 5.3: Rendering all ways (gray) and buildings (red).

The visualization in Figure 5.3 shows all ways and buildings of Vienna. The corresponding composition contains only 10 patches in total (see Figure 5.5). The ***OSM File Reader***'s way output stream is split with a ***Filter Buildings*** patch into two separate streams. One stream contains way elements which represent buildings and another one which does not. Both streams are converted to render primitives of different colors with ***Render Ways*** patches. Both of the streams are merged again with a generic ***Stream Merge*** patch which adapts its ports to the render primitive type.

Another implemented OpenStreetMap visualization renders routes of a map for example (see Figure 5.6). The user can select interactively the routes to be visualized (see Figure 5.4). For example, public transport service lines, bicycle routes or any other routes of interest can be selected. All selected routes get sorted by their lengths and are rendered with a different color. Additionally, a bar chart of the lengths is generated and displayed in a different output window.

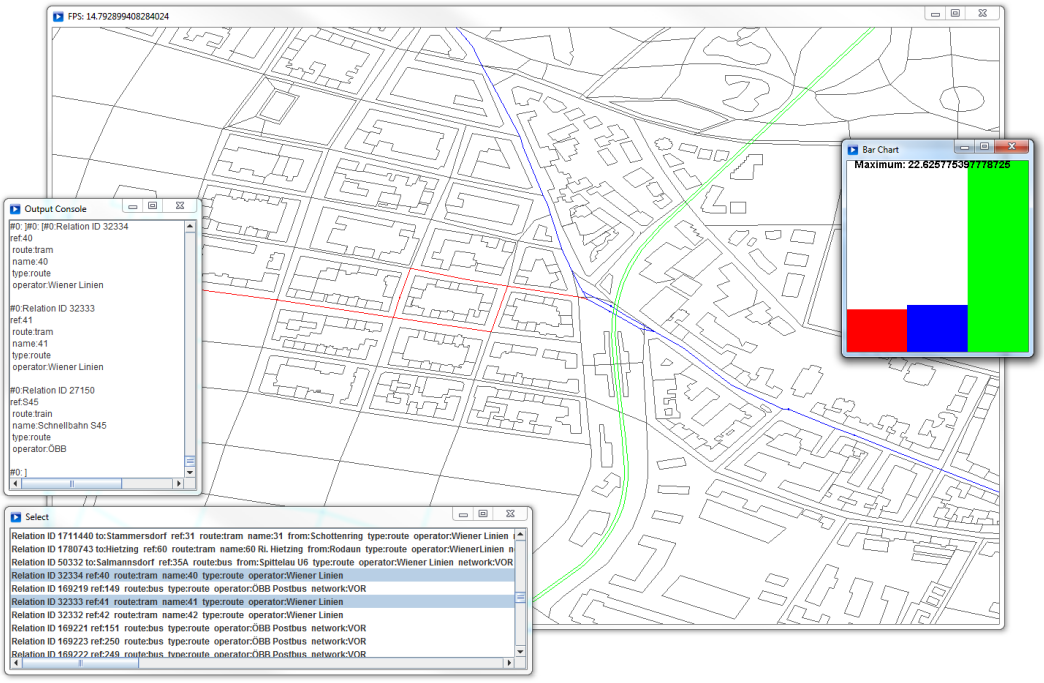
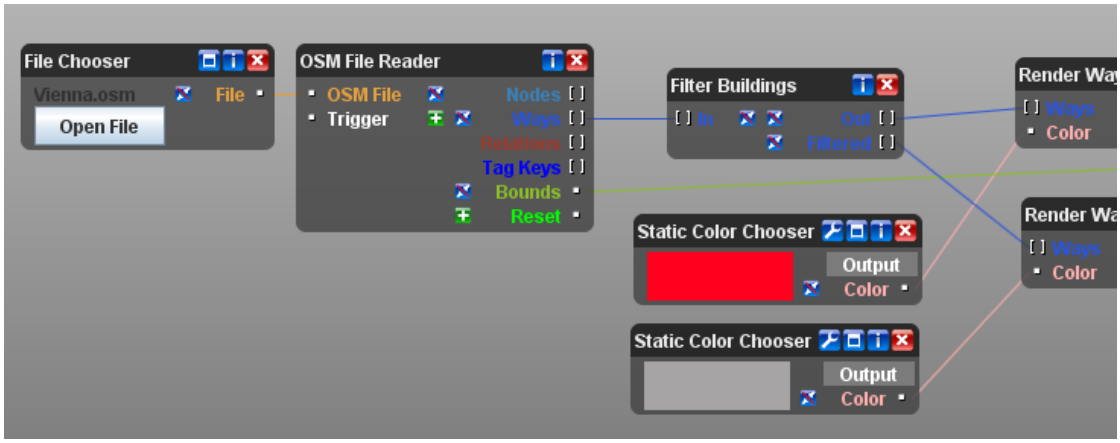
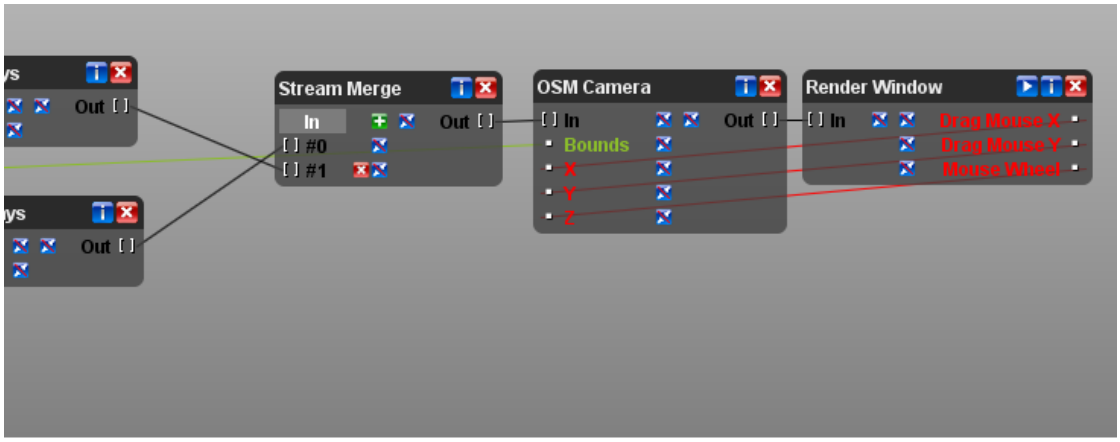


Figure 5.4: Rendering selected routes sorted by their lengths and displaying a bar chart depicting their lengths. To each route a different color gets assigned. In the lower left window users can select interactively the routes they want to visualize.

This example demonstrates the capabilities and the benefits of the usage of individually implemented data structures to create patches (especially the filter patch). 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 and re-usage.

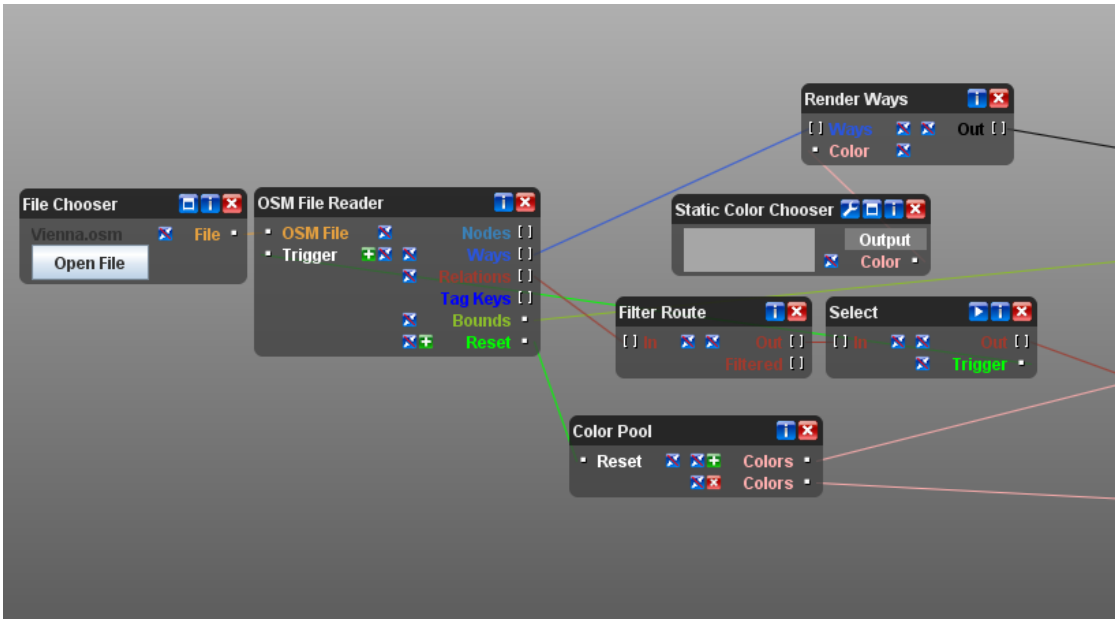


(a) Left half of the composition.

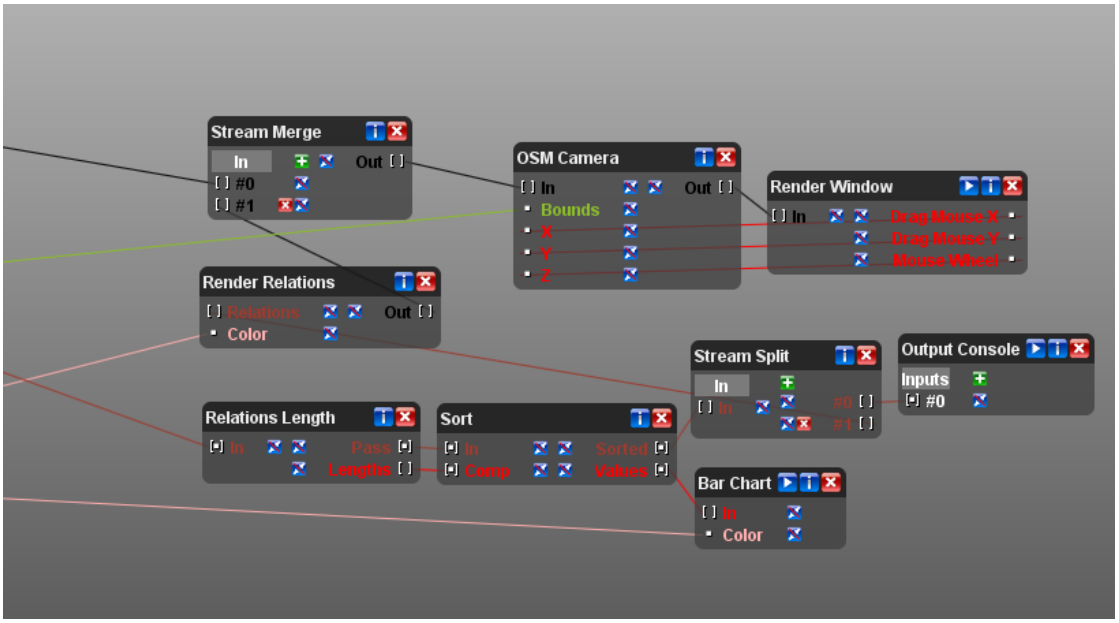


(b) Right half of the composition.

Figure 5.5: An OpenStreetMap composition which visualizes buildings and ways.



(a) Left half of the composition.



(b) Right half of the composition.

Figure 5.6: A OpenStreetMap composition which visualizes lengths of user selected ways.

CHAPTER 6

Discussion and Future Work

This chapter discusses the features which OpenInsightExplorer introduced to the field of dataflow language research and those which are obviously and rather painfully missing. The framework was evaluated with the development of the example visualizations. This happened after the completion of the framework. Most insights gathered during the production of the examples were not used to further improve the framework. Many of the proposed improvements would lead to major design changes and therefore code alterations. These improvements were not implemented because of time constraints.

6.1 Growing Ports and Generic Ports

The growing port mechanism indicates to be an admirable feature for upcoming visual dataflow languages. It simplified the development process of the example visualizations, since some patches could be adopted to the number of needed input and output ports without large efforts.

Generic ports demonstrated their practical usefulness in the example visualizations too. With the implemented generic port approach, the framework can offer many off-the-shelf patches such as the ***Stream Merge*** patch that can adapt itself to a certain data type dynamically (see Figure 5.5). Such patches can be reused more frequently because they can operate with many different data types. Without the generic ports feature, e.g., a special version of the ***Stream Merge*** patch needs to be implemented for each data type it should merge.

6.2 Structured Programming

The current version of OpenInsightExplorer does not provide any mechanism for structured programming. It was assumed, that the prototyped visualizations would not become complex enough to justify such facilities. This assumption turned out to be wrong, as soon as the OpenStreetMap example was implemented. Some of the compositions in this example fill up multiple screens in the editor, making it very hard to keep the overview. Like other visual dataflow languages OpenInsightExplorer should provide a mechanism where users can collapse whole sub-graphs to a single node and expand it on demand again.

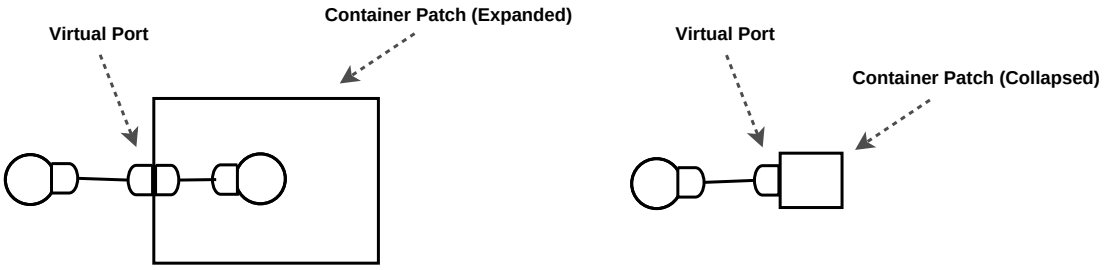


Figure 6.1: Schematic diagram of a the intended ***Container*** patch.

To accomplish this, a special ***Container*** patch must be developed (see Figure 6.1). Users can drag patches into it. The container can be collapsed and expanded, hiding or revealing the embedded patches. If a patch, which is embedded within such a container, gets connected to a patch lying outside of the container the ports of the embedded patch is virtually duplicated and will be renderer at the enclosing container border. Both patches connect to the virtual port instead of connecting to each other directly.

In the case of OpenInsightExplorer the implementation of this feature becomes a lot more complex in comparison to other visual dataflow languages, since only OpenInsightExplorer provides the unique growing ports mechanism. The virtual duplicates need to have the same add and remove port functionalities, like their real counterparts. It seems necessary to duplicate the parent ports on some occasions too since they may have add and remove listeners which will operate on their childs.

6.3 Debugging

The framework lacks a debugging toolkit. Debugging in dataflow languages means to find the node which is not producing (any) output and the reason why. Most often, input ports are not connected in the right manner, so they do not reach the firing state as expected.

In the current version of OpenInsightExplorer, users have to debug their programs by temporarily connecting output ports to the ***Console*** patch to investigate if a patch produces output as desired. By repeating this task on several patches, it is possible to track down the error. Moreover, each time the users have to reconnect the output ports to their previous connection

partners again. This is a very exhausting task, since it becomes more a trial-and-error approach to find the patch which does not behave like indented.

A future version of OpenInsightExplorer should possess a sophisticated debugging toolkit. This toolkit should allow to run visualizations in a special debug mode. In this mode the editor window will not disappear and users can point with the cursor on patches and their ports. Pointing on a port will result in displaying the content of the FIFO buffer and the information if a thread is halted caused by an empty or full buffer. Additionally, an ***inspection*** window will exist which shows all patches and ports which are currently halted by an attempt to read from an empty input port or an output port which tries to send to a full FIFO buffer.

6.4 Data Types and Side Effects

As previously addressed in section 3.2.5, OpenInsightExplorer allows developers to introduce new custom data types. This is accompanied with the loss to be side-effect free. This is a small price to pay for the flexibility which is gained through this design decision. Also some other state-of-the-art languages, e.g., Quartz Composer are not side-effect free and even tied down to a handful of native data types.

6.5 GUI

The GUI of OpenInsightExplorer could be improved in several ways on closer examination:

- The disconnect buttons of ports can be entirely removed. Instead clicking on a port or a connection wire with the right mouse button could provide the same functionality and would result in a cleaner interface.
- All patches should provide a minimize icon in their titlebar. A patch window would collapse to the point where only its titlebar would be visible any more. All connection wires of the input ports would end on the left boundary of the titlebar and all output connections on the right end.
- Connection wires should be rendered more like wires instead of straight lines. This can be accomplished by using splines.
- Stream port connection wires should be rendered bolder than other wires.
- It should be studiously avoided that wires cross other wires or reside behind patches if possible.
- Users should be able to relocate wires manually.

CHAPTER 7

Conclusion

This thesis presents OpenInsightExplorer a new visual dataflow programming language. It was designed for rapid prototyping visualizations. It allows to connect modules in a visual editor that represents individual processing stages of the visualization pipeline. These modules are executed by the dataflow execution model, which almost naturally supports parallel execution.

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. To extend 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 introduces new unique features to the field of visual dataflow programming research. Features 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. It seems worthwhile to further investigate the dataflow execution architecture of OpenInsightExplorer. It uniquely combines the two main dataflow execution models and simplifies the development of modules.

To summarize the results of this thesis and to make a conclusion: OpenInsightExplorer clearly failed to be a universal tool for non-programmers for developing arbitrary visualizations in the current development state. The visual programming paradigm of OpenInsightExplorer is

still too complex for users without any programming experience. The example visualizations proved that OpenInsightExplorer cannot provide all necessary modules off-the-shelf. All missing modules must be implemented by the users. 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. OpenInsightExplorer's concept is not flexible enough to support the development of complex or arbitrary visualizations. Nevertheless OpenInsightExplorer introduces new unique features, which could bring great benefits to other visual dataflow languages. They are worthwhile to be adopted by current state-of-the-art languages to improve their usability.

APPENDIX A

User's Guide

A.1 Introduction

OpenInsightExplorer is a visual dataflow programming language for developing visualizations. This type of programming language consists of nodes called ***patches*** in OpenInsightExplorer. These modular units get connected together in a graphical editor of the framework. Depending on how patches are connected they represent a visualization application. This guide gives an introduction on how to find the right patches for a visualization and to compose them together to an application.

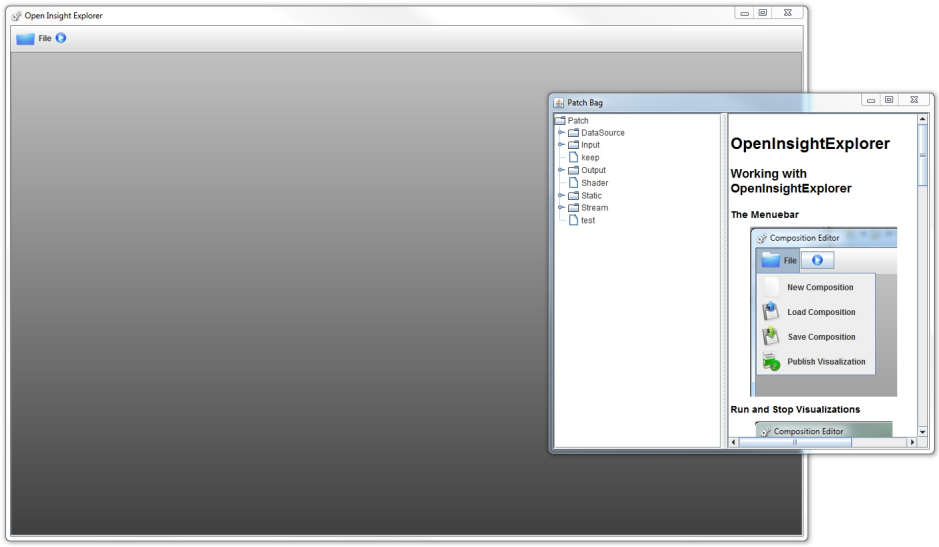


Figure A.1: Main GUI after starting OpenInsightExplorer.

A.2 The GUI

After starting OpenInsightExplorer two different windows are visible on the screen (Figure A.1). The big main window is the ***workbench*** editor (entitled ***Open Insight Explorer***) and the smaller one the so called ***Patch Bag***.

A.2.1 The Workbench

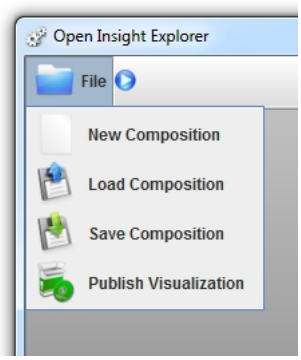


Figure A.2: The workbench's menubar.

With the workbench a user is capable to load, store and run visualizations and of course build them. Visualization programs are called ***compositions*** in OpenInsightExplorer, since they are composed of individual patches. Clicking on the play-button in the menubar (see Figure A.2) runs a composition ( ). The ***File*** menu allows to load and store compositions. When a composition is started the workbench and patch bag window will disappear to make space for GUI elements of the visualization. To stop a running composition and return to the editing mode of the editor a user has to click on the stop button that will be displayed in the upper-left corner of the screen (Figure A.3).

Figure A.3: The stop icon in the upper left corner of the screen.

A.2.2 The Patch Bag

The smaller window is the patch bag, see Figure A.4. On the left side is a tree containing patches which are modules with different functionalities grouped in libraries. They are arranged and categorized in a tree by the features they provide. The tree consists of categories and even sub-categories.

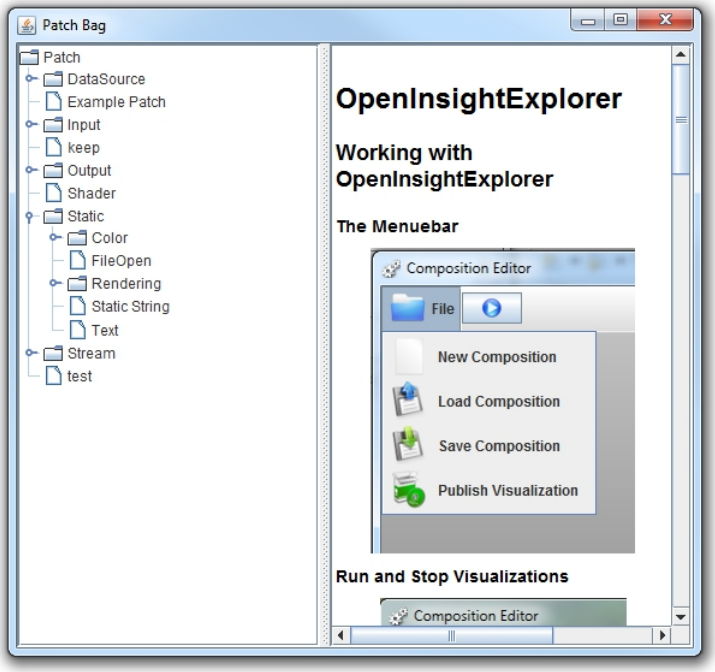


Figure A.4: The patch bag with a selected ***Slider*** patch, showing information about the patch.

There are several categories like ***Input***, ***Output*** and ***Rendering*** for example. The input library consists of patches which provide GUI elements like sliders and textfields. Rendering contains modules to perform OpenGL rendering. The right side of the window provides information about the patches if they are selected in the tree. It shows information about how to use and configure these patches in a meaningful manner. To develop visualizations with OpenInsightExplorer the user drags patches from the patch bag and drops them into the workbench editor window as depicted in Figure A.5. In the editor a new graphical representation of the patch will appear. This little window has different functionalities which will be explained in the next section.

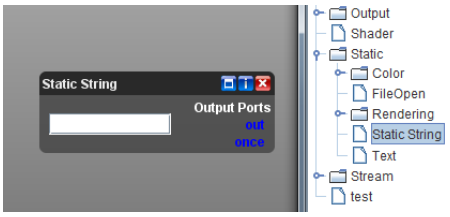


Figure A.5: Dragging patches into the workbench.

A.3 Patches

Patches are displayed as little windows in the workbench editor (see Figure A.6). They can be moved around by dragging them on the titlebar. The name of the patch can be altered via double clicking on it and editing the appearing textfield. Depending on the provided functionality of a patch its titlebar may contain several icons on the top right corner.

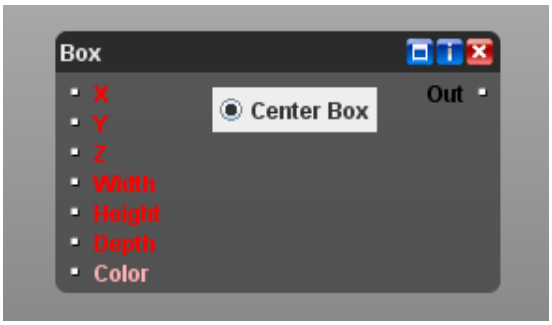


Figure A.6: Patch with input ports (named ***X***, ***Y***, ***Z***, ***Width***, ***Height***, ***Depth***, ***Color***) and an output port (Out) and a bound GUI (the checkbox labeled ***Center Box***).

Almost every patch has so called ***ports***. The ports placed on the left (below the titlebar) are called ***input*** ports. With these ports the patch is able to receive data. On the right side of the patch are the ***output*** ports. The patch uses them to send data to another patch. The names of the ports are printed in different colors depending on the data type the ports are able to send/receive data. Different patches can be connected by clicking on their ports which should be connected together.

A.4 The Patch Titlebar Icons

- ***Delete***
  Removes a patch from a composition. All connections to other patches will be removed automatically. Also all windows the patch owns are terminated.

- ***Info***
  Shows information about this patch in the patch bag window.

- ***Hide Bound GUI***
  Some patches have a GUI element to configure its own behavior. This GUI element is placed in the middle under the titlebar of the patch. It is called bound GUI because it is directly ***bound*** to the graphical representation of the patch in the workbench. With this icon it can be hidden and the patch will have a smaller size (see Figure A.7).

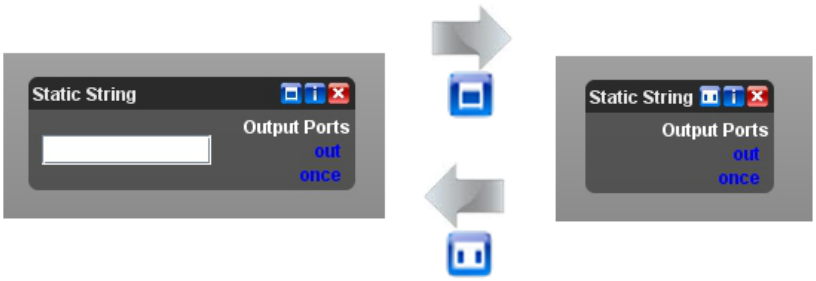


Figure A.7: Examples of a shown and hidden bound GUI (the textfield) of a patch.

- ***Show Bound GUI***
  Shows the bound GUI of a patch.
- ***Show Configuration GUI***
  This icon will bring up a window where the user can configure the patch (see Figure A.8).

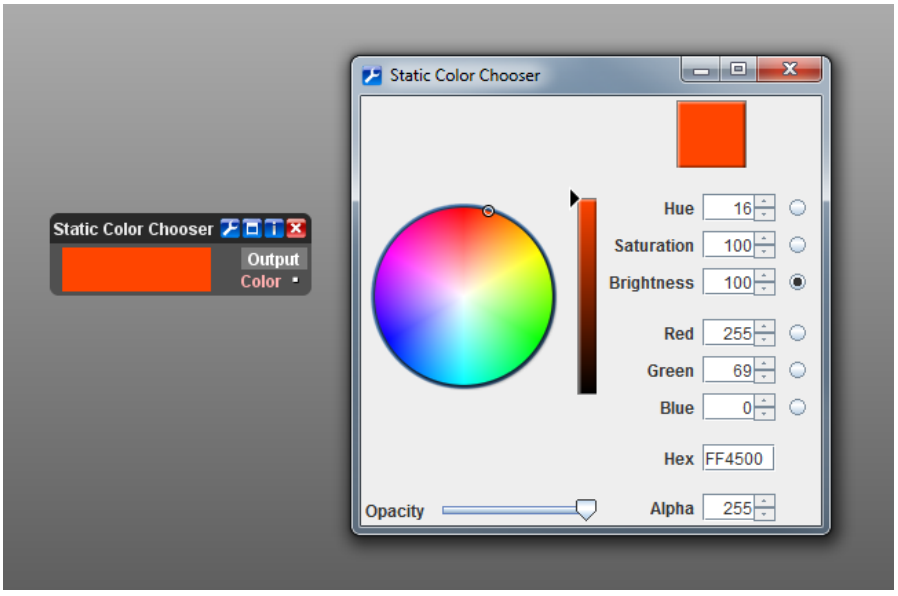


Figure A.8: A color chooser with its configuration window visible.

- ***Show Running GUI***
  Will show the running GUI, the window the patch owns that will be visible when the composition runs (see Figure A.9).

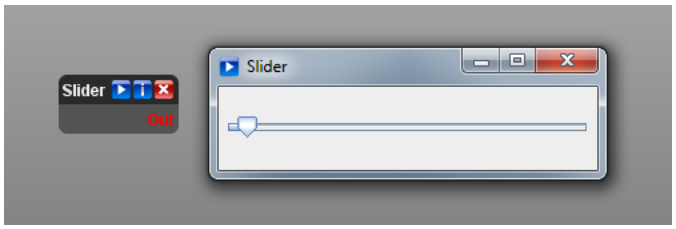


Figure A.9: A patch with its running GUI visible.

A.5 A “Hello World“ Tutorial

This section depicts a little basic step-by-step tutorial on how to program with OpenInsightExplorer. This tutorial implements a simple “Hello World“ program. It explains how to add patches to a composition, connect them and finally run the composition.

Composing “Hello World“

- Start OpenInsightExplorer or in the case it is already running, create a new composition by clicking on ***New Composition*** in the ***File*** menu at the workbench window (see Figure A.10).

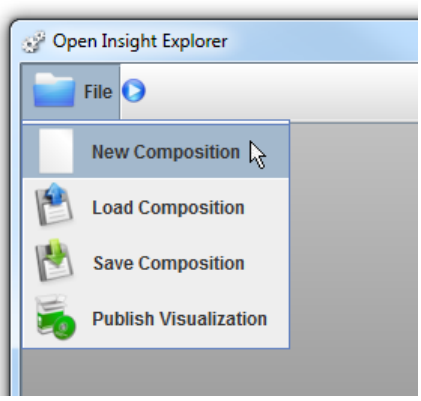


Figure A.10: Starting a new composition.

- Expand the sub category ***Static*** in the patch bag window by clicking on it. The library tree will now expand and show new entries.

- Drag the entry named ***Static String*** from the patch bag window into the workbench. A new window entitled ***Static String*** will appear in the workbench, as depicted in Figure A.11.

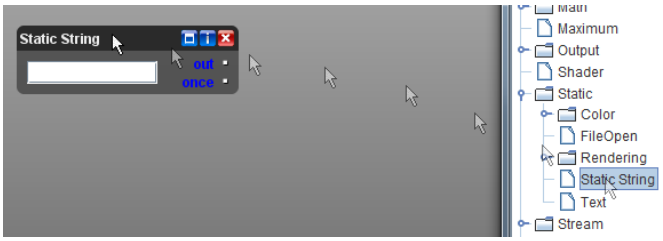


Figure A.11: Dragging the ***Static String*** patch into the workbench.

- Expand the ***Output*** library in the patch bag and drag the ***Console*** patch into the workbench (see Figure A.12). In the workbench a small window entitled ***Console*** will appear. Also a new window called ***Output Console*** will pop up. This window is the running GUI of the ***Console*** patch. It will display all data which is send to the patch.

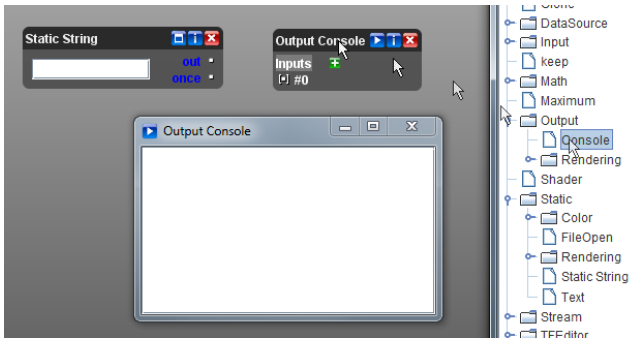


Figure A.12: Dragging the ***Console*** patch into the workbench.

- The user can drag patches in the workbench on their titlebar.
- Connect the ***Static String*** output port entitled ***once*** with the ***Console*** input port labeled ***#0*** by clicking on ***once*** and later on ***#0***. Some icons near those ports will appear (see Figure A.13).

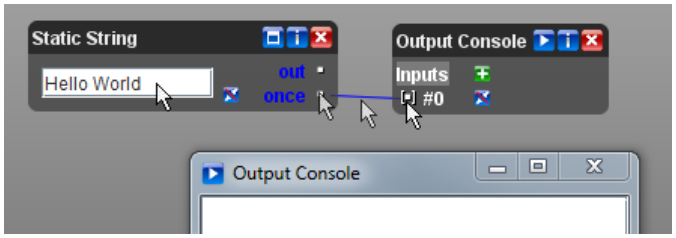


Figure A.13: Dragging the ***Console*** patch into the workbench.

A.5.1 Running the Application

To run the "Hello World" program just press the play ( ) icon in the workbench's menubar. The editor and the patch bag vanish and all running GUI windows will appear. In this example it is the ***Console*** window, depicted in Figure A.14. Everything that was entered in the textfield of the ***Static Text*** patch will now be displayed in the ***Console*** patch's output window. To stop the application just click on the stop button displayed in the upper left corner of the screen.

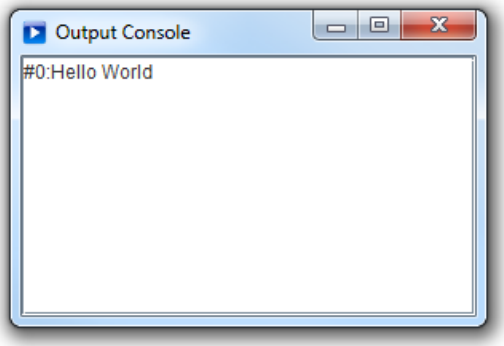


Figure A.14: The ***Console*** patch outputs the string which was send to it.

A.5.2 Loading and Saving Compositions

Compositions can be loaded and saved. Under ***File*** in the menubar of the workbench are two entries for that purpose, ***Load Composition*** and ***Save Composition***. In both cases a file dialog will appear (see Figure A.15). Compositions carry the file postfix "***.cmp***". Either it is a dialog for choosing a composition file to load or specify a name and a location to save a composition. When a composition is loaded the current composition will be lost if not saved prior.

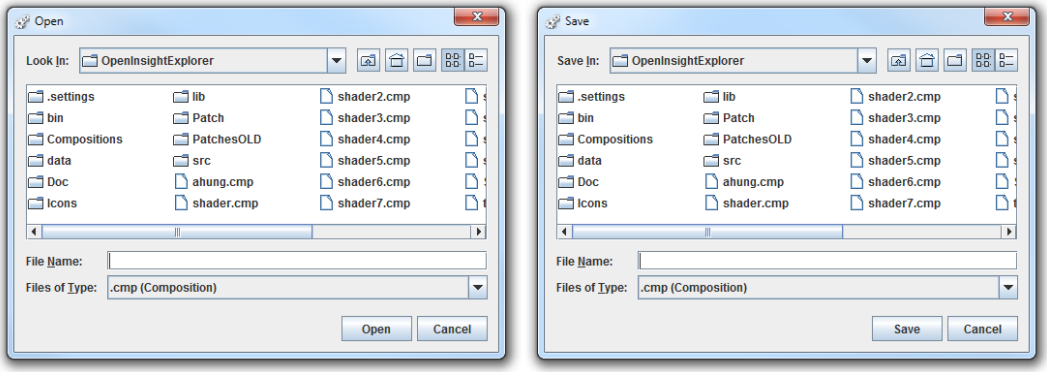


Figure A.15: A composition loading and saving dialog.

A.6 Ports in Detail

This section takes a closer look on ports and exposes the rules when they will accept or decline connections. It describes the icons of the ports and the coloring of their names.

A.6.1 The Different Port Types

There are three different types of ports in OpenInsightExplorer. The little icons near the name of the port allow a distinction of the type of the port.

- ***Basic ports***
  Input and output ports receive and send individual chunks of data. This type of port can only be connected to other basic ports or to mixed mode ports.
- ***Stream ports***
  These ports exchange data as a stream. Stream ports only accept connections to other stream ports or mixed mode ports.
- ***Mixed mode ports***
  Mixed mode ports can handle data from basic and stream ports. Depending on their connection partner they will change their own type dynamically.

A.6.2 Connection Rules

- It is only possible to connect input with output ports. This seems almost straightforward since only output ports provide data and only input ports are able to receive data.
- Ports allow only one-to-one connections. It is impossible to couple an input port with more than one output port or vice versa.
- Ports can only be connected to ports of the same type except they are connected to mixed mode ports.
- The different colors represent different data types. A port can only be connected to another port of the same color because it should understand and handle the same data type. This rule has exactly two exceptions:
  - ***The port has a white color***
    This means that the port can handle a generic data type. It will either dynamically adapt its type to the one it is connected to or will refuse a connection if the patch cannot handle this data type. In some cases the patch even changes the type of multiple ports to adapt itself to the new data type. For example the ***Console*** patch has such ports because it should print out information about the data it receives, regardless which type the data has (see Figure A.16). Another example is the ***Filter*** patch. Depending on the data type it receives on the input ports it will send out data of the same type on the output port.

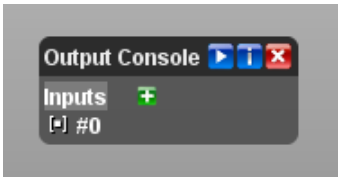


Figure A.16: ***Output Console*** with a generic port.

– ***Sub data types***
In some cases data types are build upon other data types. The coloring of the ports suggests the least common thread the port can handle. The information about what data types and sub types it may or may not accept is displayed on the right side of the patch bag. This can be either invoked by clicking on the info icon ( ) in the patch's titlebar or by selecting the patch in the patch bag. The port will refuse a connection upon attempt with an unsupported data type and the editor will display a notice message.

A.6.3 Port Icons

- ***Disconnect***
This icon triggers the disconnection of both involved ports of a connection. After the dissolve of the connection, both ports are ready to get newly and independently connected again. A example is depicted in Figure A.17.

Figure A.17: Example how to disconnect two patches.

A.6.4 Growing Ports

OpenInsightExplorer includes a feature called ***growing ports***. It provides patch developers the ability to write patches that can dynamically vary the number of ports a patch provides (see Figure A.18). With growing ports it is possible that a user can signal a patch that more or less input ports are needed, depending on the number of operands the patch should process. The growing port approach is not limited to adding ports of the same kind. Signaling a patch to add ports may result in adding simultaneously input and output ports or even a collection of ports of different data types.

- ***Add***
  The add icon causes the patch to add new ports or even a collection of ports to itself.
- ***Remove***
  Removes a port or a collection of ports. Connections of removed ports will be suspended automatically.

Figure A.18: Example of the growing ports mechanism. Everytime the ***Add*** icon is triggered the patch adds a set of input ports (Name, Value) to the ***Uniforms***. In this example it was triggert twice. Clicking on the ***Remove*** icon removes just the set the icon belongs to.

APPENDIX B

Programmer's Guide

B.1 Introduction

The functionality of OpenInsightExplorer can be extended by developing new ***patches***. This guide gives a brief introduction how to achieve this.

OpenInsightExplorer was designed to be easy extendable. Patches are only Java classes which implement a special interface called ***Patch***. The Program scans through the ***Patch*** directory and its sub directories to find such classes. Patches can be appended to an OpenInsightExplorer installation by just copying the compiled class files into these directories.

This approach has a major drawback: OpenInsightExplorer must call the standard constructor of each class in order to be able to verify the implementation of the interface. No resources of any kind should be allocated or constructed within the standard constructor.

As soon as a patch is dragged from the patch bag into the workbench, the editor will construct a new instance of this patch class. Afterwards it calls the ***init()*** method of the instance to signal that it became ***life***. Within this ***init()*** method all resources such as ports, ***JPanels*** and ***JFrames*** should be created. After calling this method the editor will fetch all references to GUI elements only ***once***. These references should be static for the whole lifetime of a patch.

B.2 Patch Interface

In order to write a new patch, the ***Patch*** interface must be implemented. This interface consists of a set of methods making the patch working. The Listing B.1 depicts the ***Patch*** interface followed by a detailed description of each method.

```
package OpenInsightExplorer;

import java.io.Serializable;
import javax.swing.JFrame;
import javax.swing.JPanel;

import OpenInsightExplorer.Ports.Interfaces.InputPort;
import OpenInsightExplorer.Ports.Interfaces.OutputPort;

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 B.1: The ***Patch*** interface.

- ***void init()***
  Is called when the patch gets dragged into the workbench window and becomes alive. All initialization must be done here exclusively. The standard constructor should not be used for the allocation of any resources.

- ***void start()***
  Threads and port listeners (which the patch needs) should be constructed and registered within ***start()***. It is being called every time a composition is started.

- ***void reset()***
  OpenInsightExplorer calls this method before ***start()*** every time a composition is started. Developers should reset or clear data structures like lists within this method.

- ***void stop()***
  Is called when a composition is stopped. Within this method ***Receive*** and/or ***Wait*** listeners of ports should be unsubscribed if necessary and threads spawned by the patch should be stopped.

- ***String getInfo()***
  This function should return a ***String*** containing a description of the functionality of the patch. This information will be displayed in the right half of the patch bag window and if the  gets clicked in the titlebar of the patch. HTML-tags can be used to describe the look of the patch bag information page and even pictures and hyperlinks can be included (see Listing B.2).

```
public String getInfo(){
  return "<html><h1>My Patch</h1>My Patch does ... <br> and looks like <
      br><img=" ./myPatch.jpg"></html>";
}
```

Listing B.2: Example implementation of the ***getInfo()*** method.

- ***String getName()***
  Returns the name of the patch used for the patch bag entry and the titlebar of the patch.

- ***void load(Serializable o)*** *and* ***Serializable save()***
  On saving a composition in OpenInsightExplorer the ***save()*** method is called for every patch and the framework writes the results in a serialized form into the save file. Therefore it is possible to reproduce the state of each patch on loading. Each object will be deserialized on loading and given as an argument for the ***load()*** method.
  The following Listing B.3 excerpts the loading and saving of a patch which represents a graphical slider.

```
private JSlider jsl;

public void load(Serializable o){
  // set the slider to the right value after loading
  jsl.setValue((Integer)o);
}

public Serializable save(){
  // read the state of the slider and save it
  return (Integer) jsl.getValue();
}
```

Listing B.3: Load and save example.

- ***Port getInputPorts()*** *and* ***Port getOutputPorts()***
  All input and output ports of a patch are organized in trees. ***getInputPorts()*** and ***getOutputPorts()*** should return the root nodes of each tree (see Section B.5). In the case the patch has no input or output ports the method should return ***null***. The functionality of ports will be explained in detail in a later section of the guide (see Section B.3).

Patches can have GUI elements which will be shown in the OpenInsightExplorer editor. The following methods are only called once (after the patch has been constructed and ***init()*** was called). All the references to these Swing objects should be static for the whole lifetime of the patch.

- ***JPanel getBoundGUI()***
  Should return the reference to a ***JPanel*** representing the bound GUI. It is used to display only a few GUI elements to configure properties of a patch. This ***JPanel*** will be fitted centered within a patch representation in the workbench window. It can have its own ***Layout Manager*** or should be set to a fixed size.

- ***JFrame getConfigurationGUI()***
  This ***JFrame*** is visible while editing a composition in the editor. Its purpose is to display GUI elements that configure a patch. Patches should use this configuration GUI if the bound GUI would become too large and/or overloaded with elements.

- ***JFrame getRunningGUI()***
  Patches can have windows that will be visible when a composition runs. These ***JFrames*** should be used to display the output or controlling elements of a visualization.

B.3 Ports

Patches exchange information via their ports. Most important there are two types of ports. Input ports which receive data and output ports which send data. Ports allow only one-to-one connections. It is impossible to couple an input port with more than one output port or vice versa. Ports can be further classified to the following types:

- ***Basic ports***
  They can only send or receive one individual chunk of data at a time.

- ***Stream ports***
  These ports operate on a data stream. They signal a start of a stream, send some data and finally signal the end of the data stream. Streams can be empty and layered within other streams of the same data type.

- ***Mixed mode ports***
  Mixed mode ports can handle data from basic and stream ports. Depending on their connection partner they adapt their own type.

Additionally developers must specify which kind of data type a port should operate on. The following code snippet in listing B.4 demonstrates how to create ports for different kinds and data types.

```
try {
// input port with data type Double
  InputPort in=new InputPort<Double>(){}.setName("Number");

// input stream with data type String
  InputStreamPort names=new InputStreamPort<String>(){}.setName("Names");

// input mixed mode port with data type Point
  InputMixedPort coord=new InputMixedPort<Point>(){}.setName("Coordinates");

// output port with data type Double
  OutputPort max=new OutputPort<Double>(){}.setName("Maximum");

// output port with data type String
  OutputStreamPort lines=new OutputPort<String>(){}.setName("File lines");

} catch (AnonymousException e) {
// in the case a port was not created as an anonymous class
}
```

Listing B.4: Creating different ports with different data types.

All three kinds of ports can be declared to work as ***generic*** ports. That means that they can change their data types dynamically at runtime (depending on the connection partners). An introduction to this topic is given in section B.10.

B.4 Port Labels

These ***label*** ports are not designed for message passing. They have only structural purposes for the port tree GUI and cannot by declared with any data type (see Listing B.5). Their usage is explained in more detail in the following section.

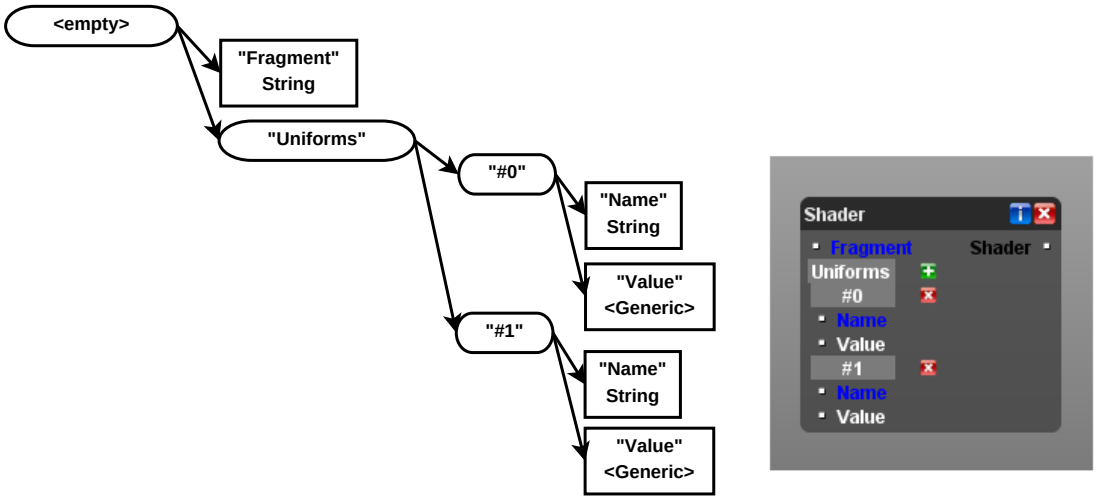
```
// input port label
InputPortLabel inroot=new InputPortLabel().setName("Inputs");

// example of a not rendered output port label
OutputPortLabel out=new OutputPortLabel();
```

Listing B.5: Using ***label*** ports to structure the port trees.

B.5 Port Trees

OpenInsightExplorer uses two trees to organize the ports of a patch. These trees (one for the input and another one for the output ports) can be altered dynamically via the ***growing port*** mechanism. Patches return the references to the root nodes of both trees via the ***getInputPorts()*** and the ***getOutputPorts()*** methods (see Section B.2).

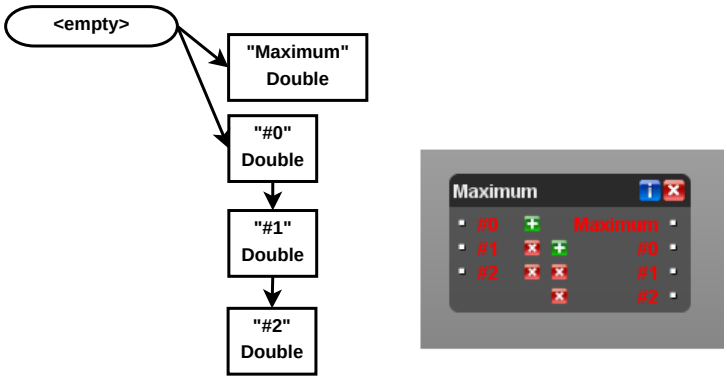
Figure B.1 illustrates how the patch ***Shader*** organizes its input ports. Boxes with round corners symbolize label ports (see Section B.4). They are only for structural purposes and do not exchange any data with other ports. Ports which return ***null*** via the ***getName()*** method are not rendered in the OpenInsightExplorer GUI, like the input tree root node of the ***Shader*** patch.



(a) (b)

Figure B.1: Example of an input port tree of the ***Shader*** patch and a screenshot depicting how the OpenInsightExplorer editor renders that tree. Note that the root node is unnamed and therefore not visible.

To each port a child can be added via the ***addChild()*** method. Furthermore ports can also be organized as lists (see Figure B.2). Ports are added to a list using the ***add()*** method. ***Only ports of the same kind and data type can be added to the list.***



(a) (b)

Figure B.2: Example of an output port tree, were some ports are organized in a list. Note that the root node is unnamed and therefore not visible.

B.6 Spawning Threads

The OpenInsightExplorer framework provides functionality to spawn and execute threads for a patch. Developers need not create and control the threads of their patches manually. By calling ***OpenInsightExplorer.execute()*** method programmers can pass an implementation of the ***Runnable*** interface which will be executed by the framework automatically.

Reading from ports and sending data should happen ***exclusively*** within spawned threads or port ***Listener*** functions. Sending and receiving function-calls may block in the case a receiving buffer is empty or full. Hence calling an input port's ***get()*** or an output port's ***send()*** method within any in the ***Patch*** interface declared methods can result in a deadlock of the visualization. In the case a thread reads from an empty input port it is suspended until data has arrived. If meanwhile the composition is stopped by the user an ***InterruptedException*** is thrown. Within its ***catch*** statement developers can exit the ***run()*** method and therefore expire the current thread (see Listing B.6).

```
  public void start() {
    OpenInsightExplorer.execute(this);
  }

  public void run(){
    while(true){
      try{
        System.out.println(In.get());
      } catch (InterruptedException ie){
        return;
      }
    }
  }
}
```

Listing B.6: Threading example.

B.7 Data Request

Output ports of OpenInsightExplorer are capable to determine if their connection partner ***requests*** data. This feature was implemented to support interactive GUI elements that are only providing data upon request. This technique is demonstrated with a simple patch that sends the current running time of a composition (see figure B.3).

This ***Running Time*** patch only has one output port called ***Out*** (see Listing B.7). In the ***init()*** method a ***WaitListener*** is registered to the port. The registered callback function ***Wait()*** of the listener is called every time the connected input port tries to read data from it.

In this function the difference of the current system time and the value of the variable ***starttime*** is send through the port ***Out***. This variable contains a timestamp when the composition was started. This was determined within the ***start()*** method of the patch.

Figure B.3: Screenshot of the ***Running Time*** patch.

```
public class RunningTime implements Patch, OutputPortWaitListener<Long>{

  OutputPort<Long> Out;
  long starttime;

  public String getInfo() {
    return "Running Time";
  }

  public String getName() {
    return "Running Time";
  }

  public Port getOutputPorts() {
    return Out;
  }

  public void init() {
    try {
      Out=new OutputPort<Long>(){}.setName("Millis").setWaitListener(this);
    } catch (AnonymousException e) {}
  }

  public void start() {
    starttime=System.currentTimeMillis();
  }

  public void Wait(OutputPort<Long> port) {
    Out.send(System.currentTimeMillis()-starttime);
  }
}
```

Listing B.7: A source code snippet of the ***Running Time*** patch. Note that only essential methods are depicted in this listing. All other methods of the ***Patch*** interface are empty or return ***null***.

B.8 Trigger Functionality

Developers can register a ***listener*** to an input port. This listener will call a function in the case the port currently receives data. This provides a ***trigger*** functionality. This mechanism is demonstrated with a little example patch. The ***Even*** patch will determine if the number is even (see Figure B.4).

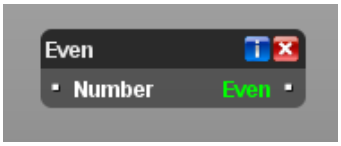


Figure B.4: Screenshot of the ***Even*** patch.

This patch has two ports (see Listing B.8). An input port called ***In*** and the output port ***Out***. A ***ReceiveListener*** is registered to the input port ***In*** which triggers the callback function ***Receive()*** every time ***In*** receives data. Within this method the currently received data from ***In*** is read and it is determined if the number is even. The result is finally sent with ***Out***.

```
public class Even implements Patch, InputPortReceiveListener<Long>{

  InputPort<Long> In;
  OutputPort<Boolean> Out;

  public String getInfo() { return "Even"; }

  public Port getInputPorts() { return In; }

  public String getName() { return "Even"; }

  public Port getOutputPorts() { return Out; }

  public void init() {
    try {
      In=new InputPort<Long>(){}.setName("Number").setReceiveListener(this);
      Out=new OutputPort<Boolean>(){}.setName("Even");
    } catch (AnonymousException e) {}
  }

  public void Receive(InputPort<Long> port) {
    try {
      Out.send(In.get()%2==0?true:false);
    } catch (InterruptedException e) {}
  }

}
```

Listing B.8: A source code snippet of the ***Even*** patch. Note that only essential methods are depicted in this listing. All other methods of the ***Patch*** interface are empty or return ***null***.

B.9 Growing Ports

The ***growing ports*** approach of OpenInsightExplorer and how to iterate over a list of ports (see Section B.5) is demonstrated with a simple example patch in this section. This patch adds numbers together. The user should be able to specify how many numbers he/she wants to add. Therefore this ***Add*** patch must be flexible on the number of input ports (see Figure B.5).

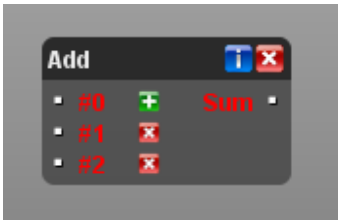


Figure B.5: Screenshot of the ***Add*** patch after two additional input ports were added by clicking twice on the plus icon.

Two ports (an input port named ***In*** and an output port ***Out***) are constructed with the data type ***Double*** in the ***init()*** method (see Listing B.9). An ***AddListener*** of ***In*** is defined which will call the ***AddRequest()*** every time a user clicks on the plus icon. Within this method an additional input port gets constructed and is added to the list of ***In***.

Since the user may remove some previously added input ports a ***RemoverListener*** is defined for each additionally constructed port. The ***RemoveRequest()*** will remove the port from the list of ***In***. The port argument of this method is a reference to the port where the user has clicked on the remove icon. A ***RemoveListener*** was never assigned to the ***In*** port, hence the user cannot remove this port and it can be used safely as the root node of the input port tree. In this case the tree becomes only a flat list. Every time a port is added or removed the ***renumber()*** method is called. It iterates trough the list of input ports of ***In*** and renames them with "#" followed by a number which is incremented at each list element.

In the ***start()*** method a thread is spawned which loops endlessly. In every pass of the loop this thread iterates over the input port's list of ***In*** and tries to read a value and adds it to the sum. If no data is available on one input port the thread gets suspended until a value is received. When the iteration is completed the sum is sent over to the output port.

When the user stops the composition, the currently executed ***get()*** method call of the input port will throw an ***InterruptedException***. This exception is used to break out of the endless loop and stop the execution of the patch.

```
public class Add implements Patch, Runnable, AddListener, RemoveListener{

  InputPort<Double> In;
  OutputPort<Double> Out;

  public String getInfo() { return "Add"; }
  public Port getInputPorts() { return In; }
  public String getName() { return "Add"; }
  public Port getOutputPorts() { return Out; }

  public void init() {
    try {
      In=new InputPort<Double>(){}.setAddListener(this);
      Out=new OutputPort<Double>(){}.setName("Sum");
    } catch (AnonymousException e) {}
    renumber();
  }

  public void start() { OpenInsightExplorer.execute(this); }

  public void run() {
    while(true){
      double sum=0;
      for(InputPort<Double> port:In)
        try {
          sum+=port.get();
        } catch (InterruptedException e) { return; }
      Out.send(sum);
    }
  }

  public void AddRequest(Port port) {
    try {
      In.add(new InputPort<Double>(){}.setRemoveListener(this));
    } catch (AddException e) {} catch (AnonymousException e) {}
    renumber();
  }

  public void renumber(){
    int c=0;
    for(InputPort<Double> port: In) port.setName("#"+(c++));
  }

  public void RemoveRequest(Port port) {
    port.remove();
    renumber();
  }

}
```

Listing B.9: A source code snippet of the ***Add*** patch. Note that only essential methods are depicted in this listing. All other methods of the ***Patch*** interface are empty or return ***null***.

B.10 Generic Ports

Ports can change their data type dynamically. This feature is called ***generic ports*** in OpenInsightExplorer. It allows developers to design patches, which can be used with multiple data types. This mechanism will be demonstrated via a simple generic buffer patch (see Figure B.6). It can adapt itself to store any data type a user may desire. Therefore no special buffer patch need not be developed for every data type.

Figure B.6: Screenshots of the ***Simple Generic Buffer*** patch, depicting it in the generic state and adapted to the ***Double*** data type.

All output and input ports can be put into the generic state by setting its data type via ***setType()*** to ***null*** (see Listing B.10). This will tell the OpenInsightExplorer framework that on a connection attempt it should give the patch the opportunity to adapt itself to the data type of the potential connection partner. Ports will recognize such an attempt by setting the ***ConnectListener***. The listener calls the method ***Connect()*** in the case of such an event. Additionally, ***Disconnect()*** gets called if the port gets disconnected.

Within the ***init()*** method the ***In*** and ***Out*** ports are set to the generic state and the ***ConnectionListener*** is registered to both of them. As soon as one of the ports ***In*** or ***Out*** is connected the ***Connect()*** method is called. Within this method it is determined if the patch is still in the generic state. In this case ***every*** generic port of the patch is set via the ***setType()*** method to the data type of the potential connection partner. If now the other port is connected the framework can check if the data types are matching, since it was set to a data type previously.

The ***Disconnect()*** method is used to determine when the patch should reset the ports to the generic state again. As soon as all generic ports are disconnected their types are set to ***null*** again.

```
public class SimpleGenericBuffer implements Patch, InputPortReceiveListener<
    Object>, ConnectListener{

  OutputPort<Object> Out;
  InputPort<Object> In;
  InputPort<Object> Trigger;

  Vector<Object> buffer;
  Class<?> type=null;

  public String getInfo() {
    return "Simple␣Generic␣Buffer";
  }

  public Port getInputPorts() {
    return In;
  }

  public String getName() {
    return "Simple␣Generic␣Buffer";
  }

  public Port getOutputPorts() {
    return Out;
  }

  public void init() {
    buffer=new Vector<Object>();

    try {
      Out=new OutputPort<Object>(){}.setName("Out").setConnectListener(this).
          setType(null);
      In=new InputPort<Object>(){}.setName("In").setReceiveListener(this).
          setConnectListener(this).setType(null);
      Trigger=new InputPort<Object>(){}.setName("Trigger").setReceiveListener
          (this);
      In.add(Trigger);
    } catch (AnonymousException e) {
    } catch (AddException e) {
    }
  }

  public void reset() {
    buffer.clear();
  }

  public void Receive(InputPort<Object> port) {
    synchronized(buffer){
      try {
        if(port.equals(In)) {
          buffer.add(In.get());
        } else {
          Trigger.get();
```

```
            for(Object o: buffer) Out.send(o);
          }
        } catch (InterruptedException e) {}
      }
    }

    public void setType(Class<?> type){
      this.type=type;
      In.setType(this.type);
      Out.setType(this.type);
    }

    public boolean isAnyGenericPortConnected(){
      return (In.isConnected()||Out.isConnected());
    }

    public boolean stillGeneric(){
      return (this.type==null?true:false);
    }

    public void resetToGeneric(){
      setType(null);
    }

    public void Connect(Port here, Port to) throws ConnectException {
      if(stillGeneric()) setType(to.getType());
    }

    public void Disconnect(Port here) {
      if(!isAnyGenericPortConnected()) resetToGeneric();
    }

}
```

Listing B.10: A source code snippet of the ***Simple Generic Buffer*** patch. Note that only essential methods are depicted in this listing. All other methods of the ***Patch*** interface are empty or return ***null***.

Bibliography

[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] M. Bernini and M. Mosconi. Vipers: a data flow visual programming environment based on the Tcl language. In *Proceedings of the workshop on Advanced visual interfaces*, Advanced Visual Interfaces, pages 243–245, New York, NY, USA, 1994. ACM.

[4] M. R. Berthold, N. Cebron, F. Dill, G. D. Fatta, T. R. Gabriel, F. Georg, T. Meinl, P. Ohl, C. Sieb, and B. Wiswedel. KNIME: The Konstanz information miner. In *proceedings of the workshop on multi-agent systems and simulation mass, 4th annual industrial simulation conference (ISC)*, pages 58–61, 2006.

[5] 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.

[6] 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.

[7] D. Comte, G. Durrieu, O. Gelly, A. Plas, and J. C. Syre. Parallelism, control and synchronization expression in a single assignment language. *SIGPLAN Notices*, 13:25–33, January 1978.

[8] P. T. Cox and T. J. Smedley. A visual language for the design of structured graphical objects. In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, pages 296–303, Washington, DC, USA, 1996. IEEE Computer Society.

[9] A. L. Davis. The architecture and system method of ddm1: A recursively structured data driven machine. In *Proceedings of the 5th annual symposium on Computer architecture*, International Symposium on Computer Architectures, pages 210–215, New York, NY, USA, 1978. ACM.

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

[11] A. L. Davis and S. A. Lowder. A sample management application program in a graphical data-driven programming language. *In Digest of Papers Compcon Spring*, 14:162–165, 1981.

[12] 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.

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

[14] Eclipse. http://www.eclipse.org. Accessed: 2011-02-01.

[15] 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.

[16] G. R. Gao and Z. Paraskevas. Compiling for dataflow software pipelining. In *Selected papers of the second workshop on Languages and compilers for parallel computing*, pages 275–306, London, UK, UK, 1990. Pitman Publishing.

[17] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35:97–107, 1992.

[18] 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.

[19] R. Jagannathan. Coarse-grain dataflow programming of conventional parallel computers. In *Advanced Topics in Dataflow Computing and Multithreading*, pages 113–129. IEEE Computer Society Press, 1995.

[20] JFreeChart. http://www.jfree.org/jfreechart/. Accessed: 2011-02-19.

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

[22] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475. North Holland, Amsterdam, 1974.

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

[24] KNIME. http://www.knime.org. Accessed: 2011-06-03.

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

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

[27] M. Mosconi and M. Porta. Iteration constructs in data-flow visual programming languages. *Computer Languages*, pages 67–104, 2000.

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

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

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

[31] R. http://www.r-project.org/. Accessed: 2011-06-02.

[32] 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.

[33] C. Sieb, T. Meinl, and M. R. Berthold. Parallel and distributed data pipelining with KNIME. *The Mediterranean Journal of Computers and Networks*, 3(2):43–51, 2007.

[34] J. Silc, R. Borut, and T. Ungerer. *Asynchrony in parallel computing: from dataflow to multithreading*, pages 1–33. Nova Science Publishers, Inc., Commack, NY, USA, 2001.

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

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

[37] W. W. Wadge and E. A. Ashcroft. *LUCID, the dataflow programming language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[38] Weka. http://sourceforge.net/projects/weka/. Accessed: 2011-03-22.

[39] P. Whiting and R. Pascoe. A history of data-flow languages. *Annals of the History of Computing, IEEE*, 16(4):38–59, 1994.

[40] N. Wirth. Program development by stepwise refinement. *Communictaions of the ACM*, 14:221–227, 1971.