Visual Programming

Dražen Lučanin, 0036428867

2009./2010. summer semester

Contents

1 Introduction **2**
1.1 General idea . . . 2
1.2 Target population . . . 3
1.3 Abstraction as a helping tool . . . 3

2 Visual programming language characteristics **4**
2.1 Definition of a visual programming language . . . 4
2.2 The principles . . . 4
2.3 The challenges . . . 6
2.3.1 Representation issues . . . 7
2.3.2 Programming language issues . . . 8

3 Overview of existing visual programming languages **10**
3.1 Scratch, an educational VPL . . . 10
3.2 LabVIEW, a dataflow VPL . . . 12
3.2.1 Dataflow programming languages . . . 14
3.3 List of existing visual programming languages . . . 14

4 Conclusion **15**

Chapter 1

Introduction

The goal of this article is to present visual programming as an alternative to classical syntax-oriented "text editing" programming.

Today most programs are written in programming languages (for example C++, Java) as text files, compiled to an assembly language representing the computer architecture and executed as a series of binary instructions. These programming languages hide the complex computer architecture away from the developer so we say that they are a higher level of abstraction. They do require the developer to know the language's syntax. The syntax consists of a lexicon (reserved words, variable and constant definitions) and a grammar (rules for variable assignment, looping, branching, defining and calling functions, including libraries...) This approach will be referred to as classical programming.

An alternative to classical programming is visual programming where the environment hides the syntax away and instead presents the user with intuitive graphical elements to work with, giving him a better overview of the program semantics and easily understood feedback.

1.1 General idea

The main idea of visual programming is to eliminate the textual syntax of a programming language and replace it with a visual environment which will guide the programmer and help him specify the intended program.

As with operating systems, we can say that a visual programming language is doing it's job well when the user does not know of it's existence - when he is concentrating on the algorithm instead.

There lies a certain danger in this freedom. Classical programming languages are very concise and through their syntax it is usually very well defined what certain language structures and operations actually translate to in computer hardware. Is eliminating this element something we want? Although developers used to coding in classical programming languages, who like this "good definition" property, would maybe scorn languages of higher abstraction, such as visual languages, for not showing this layer (to such an extent at least), it does not mean that that it includes every developer. This good understanding of a program is not always necessary. A rising number of programmers would in fact be happy to be protected from it. Let us analyse this a bit further.

1.2 Target population

Imagine a person who is used to working on a computer, but does not know how to program on a computer. Let's say this person has achieved a state where he or she would like to change something in the way the computer behaves - would like to program it. Here we roughly described three types of people:

- *people who don't know programming* - children who are learning programming
- *people who don't want to know (any more) programming* - developers who already have some skill in programming, but don't want to know how a new technology works "under the hood", instead they only want to use it
- *people who don't need to know programming* - experts in other fields who work on computers and want to maximize their productivity by programming something

They are visual programming goal users.

1.3 Abstraction as a helping tool

Why would these people need visual programming? Well, they probably don't know so much about the way computers work. They more likely think of computers as black boxes. For this reason it is irrelevant to them whether variables are declared as *float* which occupies 32 bits or *double* which occupies 64 bits of memory. They don't want to think about casting variables before doing operations. In one word, there is no reason for them to know a concise syntax of a programming language which would let them control all these things. As a matter of fact, it would be a waste of their time to learn this big chapter to achieve what they want.

Professional developers know this is not always the ideal approach. For example, if we work with big decimal numbers we want to work with *double* rather than *float* variables (maybe even some sort of *decimal* type which does not round at all, but takes as much memory as it can), because it offers more precision. But in most cases our goal users don't mind if errors like this appear, since the program they are building is (probably) not a secret government missile launching application, where precision would be critical.

Now that we are motivated to use visual programming languages (or recommend their usage to someone who fits the goal user), we will see what they really are in the following chapter.

Chapter 2

Visual programming language characteristics

2.1 Definition of a visual programming language

According to [13] *a visual programming language (VPL) is any programming language that lets users create programs by manipulating program elements graphically rather than by specifying them textually.*

In other words VPLs use visual representations of abstract objects to accomplish what would otherwise have to be written in a classical textual programming language.

2.2 The principles

In the process of creating computer programs we can identify two main phases:

1. contemplation - designing the algorithm
2. implementation - importing the algorithm into the computer

The difference between classical programming and visual programming is in the implementation phase.

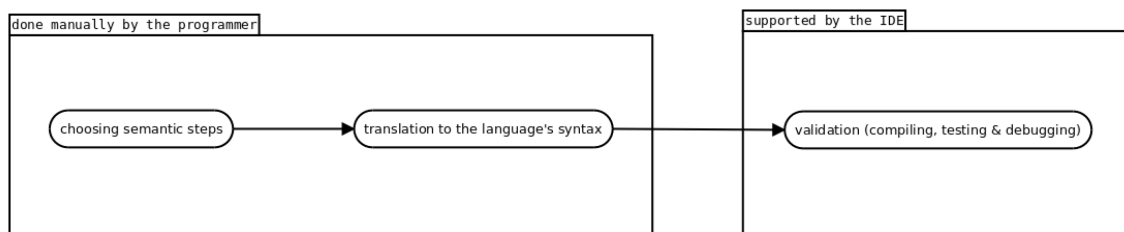

Figure 2.1: The development process in a classical programming language

As we can see in figure 2.1, in classical programming the implementation phase would consist of these 3 steps:

1. choosing semantic steps
2. writing them using the language's syntax
3. validating (compiling, testing and debugging)

The first step is necessary because the abstraction of human thought is usually of a higher level than the abstraction of a programming language. Therefore we need to divide the algorithm into semantic steps of our programming language's abstraction. This step is still necessary in visual programming, as it has more to do with the freedom allowed by a programming language, than with the philosophy of input.

The second step is what makes the biggest difference between classical and visual programming. In classical programming the developer has to enter the program as a series of commands inside a text editor. Of course, since text editors can be used for a lot of tasks, not just programming, there is lots of room for error here and it takes quite a bit of experience to master the skill of "coding". The whole idea of visual programming is to create an environment designed specifically for this task (unlike text editors, which are more general) that will guide the developer through better presentation and feedback to translate the semantic steps from his head to a computer program.

The third step consists of determining if our implementation works and if it does, does it do exactly what we imagined.

A successful virtual programming language is one which shrinks the barrier between the first and the third step to a minimum. We would say that ideally the developer would not have to know any syntax at all, but instead the environment would subtly enforce that syntax through the interface choices presented to him. This idea can be observed in figure 2.2, as a contrast to the idea shown in figure 2.1.

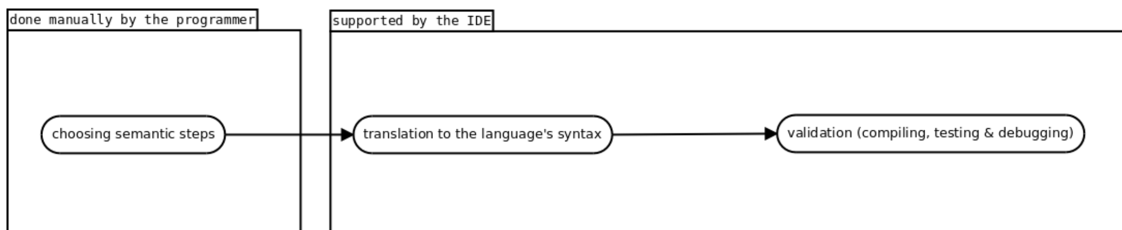

Figure 2.2: The development process in a visual programming language

In this idealisation we can omit the second step and get the target process of visual programming:

1. choosing semantic steps
2. validating (compiling, testing and debugging)

This is presented in figure 2.3.

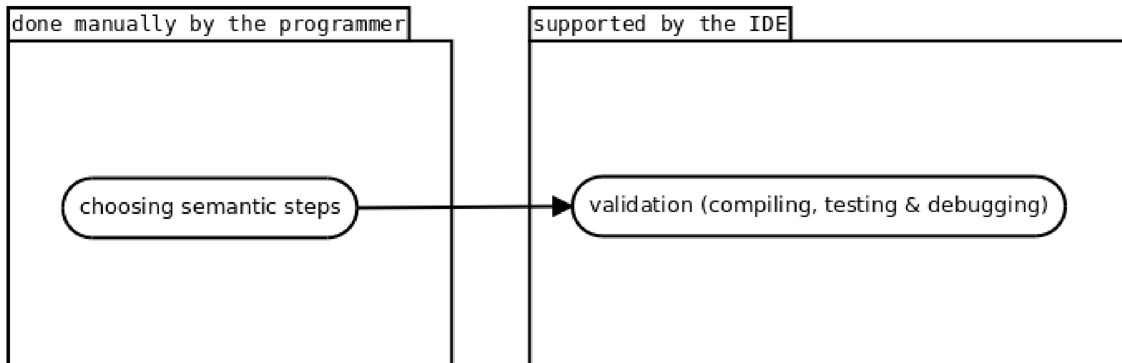

Figure 2.3: The development process in an ideal visual programming language

According to [2] improving the programmer's ability to express program logic and to understand how the program works is the main goal of visual programming languages. To achieve this, VPLs are designed using features with some of the following characteristics:

- Fewer concepts required to program - see section 1.3 on page 3
- Concrete programming process - in many VPLs, the programmer can see, explore and change specific data values or even sample executions
- Explicit depiction of relationships - dataflow diagrams are an example technique; this is our main point of interest - the ability to easily follow the algorithm through it's graphical representation rather than reading the syntax and imagining the flow of data
- Immediate visual feedback - after a programmer has edited some part of the program, many VPLs immediately display updated results to help him or her find errors as soon as possible

We have now seen what we want from a visual programming language - the good side. There also exist downsides to this simplified approach. There are things that are just better suited for textual programming languages. In the next section we will examine these potential pitfalls and see what can be done to overcome them.

2.3 The challenges

The best way to examine various common visual programming languages' features is to look at the causes of their creation - the issues that arose in their use. Most of this section was derived from the paper [2].

The biggest issue is the *scaling-up problem* - how to expand applicability without sacrificing the goals of better logic expression. It is actually quite a general definition, but it's because it contains a family of "smaller" problems. The term came from the size standpoint - it refers to the programmer's ability to apply VPLs in larger programs; in programs where the generality to connect several big modules is needed, but without sacrificing the ability to fine-tune those modules' behaviour.

The scaling-up subproblems can be divided by their nature to:

1. representation issues
2. programming language issues

2.3.1 Representation issues

Static representation

Static representation is analogous to syntax in textual programming languages. It is how we perceive the program "at rest". Typical representations can be dataflow diagrams or state-transition diagrams.

Problematic VPLs in these area are the ones defined dynamically (with no static representation) or languages with strange data structures. Let's see the examples of such languages:

- programming-by-demonstration languages where the user defines the program by doing a task manually and the environment extracts an algorithm from his actions.
- languages where attributes require nontextual references to data - by pointing at unlabeled data or icons, for example
- languages where values involve a time dimension or more than two spatial dimensions, such as animations and multidimensional data structures.

To successfully solve this issue, a VPL needs to stay consistent with it's original goals. This means that solutions such as:

- the developer required to take screenshots in various states
- translation to a standard textual language

are not useful. They just transfer the job to the user - to organise the view himself or to learn a textual language to be able to review and edit some of its parts.

A good static representation is one which offers *editability* and the capability to hide excessive visual details.

Effective use of a computer display

This is a challenge with most programs with GUIs. If display and navigation techniques are inefficient, the impact on a responsive VPL could result in very reduced usability of the language's good features, such as the ability to eye-scan the algorithm quickly and visual feedback.

This is part of the human/computer interaction field of research. In the future, we can maybe hope that progress done in this field could benefit to better graphical reasoning of VPLs and offer navigation power comparable to that now provided by search capabilities for word patterns in textual language editors.

One big advantage that VPLs can use to accomplish this is that the development environment has excellent access to the program's syntax and semantics (because the environment is responsible for creating them - they are not extracted from a text by parsing, like in classical programming). *Semantic information can control the way program information is depicted on the display.*

This issue is also closely tied to static representation.

Documentation

Documentation is more of an opportunity than a problem.

To name a few good ways of handling documentation:

- annotation of dataflow graphs
- user's ability to control when documentation is displayed (selectable levels of displayed documentation detail, "mouse over" text boxes etc.)
- drag'n'drop documentation support - an object can be dragged to windows with different capabilities, one of which displays the object's documentation
- whenever the program is changed, example output is computed giving the developer some insight into the new functionality

As we can see, viewing the documentation becomes very similar to the actual programming process - the separation between source code and documentation begins to disappear.

2.3.2 Programming language issues

Procedural abstraction

This issue is mostly solved today. Representative solutions include:

- the programmer can select and iconify a section of a dataflow graph which adds a node to a library of function nodes
- using a form as a grouping mechanism for calculations
- recording and generalizing a sequence of manipulations

It is interesting to note that historical solutions existed that corrupted the VPL's scalability. The user had to write procedures in a textual language, like C. This approach put the VPL in the role of merely a support language and was therefore not good.

Interactive visual data abstraction

This is basically the encapsulation feature. A VPL should offer some sort of visual process to define a new data type.

Type checking

Like in other programming languages there exist three ways to perform a type check:

1. static - good for detecting some errors at compilation time, executable code efficiency, helpful as documentation
2. dynamic type checking - simpler to use, but they lack feedback - some errors can get through unnoticed for a long time
3. static type checking with implicit types - as in functional languages - user does not enter type declarations, but instead they are inferred from constants; keeps advantages from both sides

Persistence

Data persistence in VPLs is possible in these four ways:

1. explicitly handling file input/output
2. database language capabilities
3. semitransparent approach - the programmer selects data that is to remain persistent, but does not explicitly save or load it
4. entirely transparent approach - all data is transparent

According to [2] the first approach is inconsistent with making programming easier and clearer, but to my opinion that's not true. Many beginners' first programs revolve around files. I would even say that their motivation to learn a programming language often comes from dealing with files and the desire to automate the task. To name a few examples:

- scanning through a file to find some string using regular expressions
- organising large amounts of textual data (from e-mail, sms, instant messaging...) in some way
- converting data from one format to another

From this it can be seen that it is quite useful to have methods for dealing with files inside a VPL.

The second possibility draws from the extensive work for visual database access.

Semitransparent approaches can enhance VPL simplicity.

An example of the fourth approach are spreadsheets.

Efficiency

Because one of the main principles of VPLs is responsiveness environment efficiency is critical.

Two main parts of the environment that need to be considered are:

- efficient language translation and program execution
- efficiency of graphical display

Chapter 3

Overview of existing visual programming languages

In this chapter an overview of some existing VPLs will be given. Scratch and LabView are languages that are actually used today and are very popular.

3.1 Scratch, an educational VPL

Also known as "The Youtube of interactive media", Scratch is an educational VPL. It is targeted at children and it allows them to build interactive, media rich applications inside the Scratch visual programming environment and share them with an on-line community.

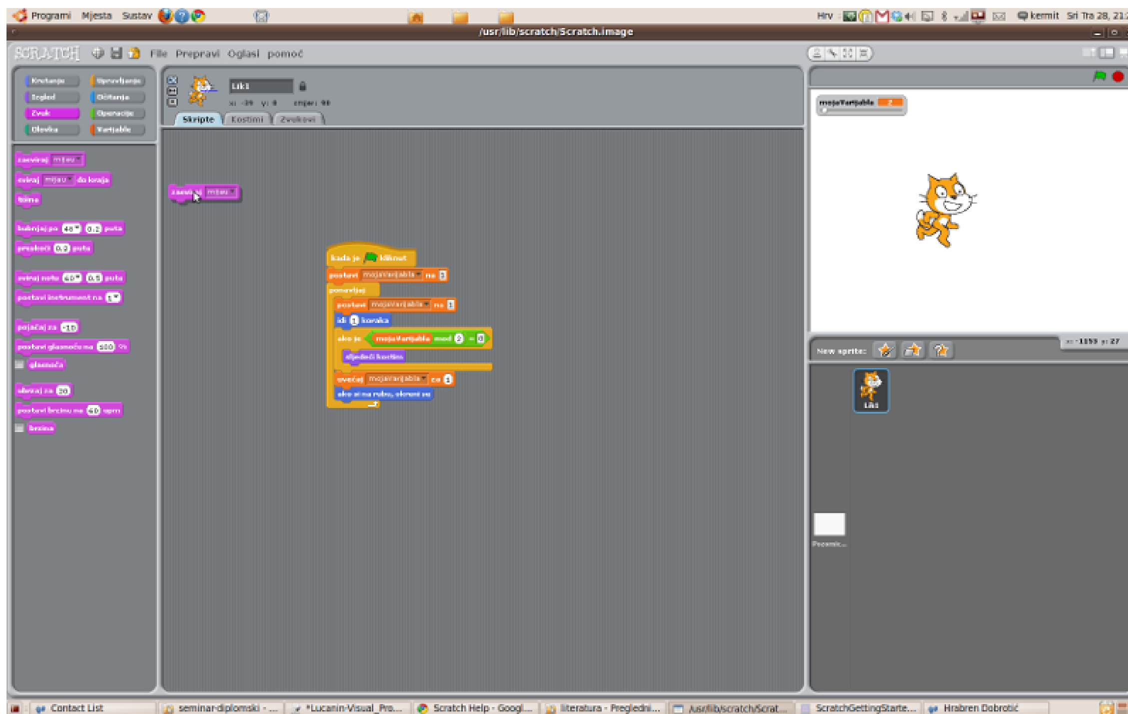

Figure 3.1: Scratch visual programming environment

Using Scratch, many types of applications can be built:

- video games
- interactive newsletters
- science simulations
- virtual tours
- birthday cards
- animated dance contests
- interactive tutorials
- ...

Users are mostly children between the ages of 8 and 16, but people of other ages too.

The designers' primary is to help children develop creative, systematic thinking and to teach them basic mathematical and computational concepts. They decided to encourage young people who already are fluent in using digital technologies (SMS, playing video games, browsing the web) to start creating new ones. It is as if they can "read", but not "write" - digital fluency should be both. [7]

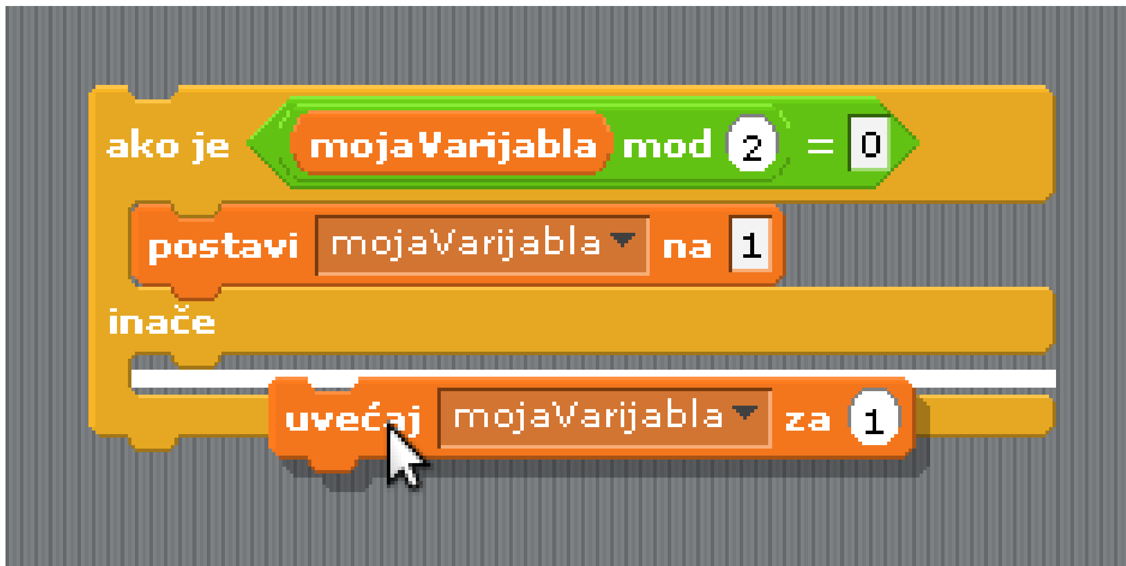

Figure 3.2: Creating an "if loop" by dragging and dropping in Scratch

In Scratch we can observe how exactly is the syntax hidden away from the user. As we can see in the picture 3.1, when we are dragging a command we want to add to the program, the environment highlights the appropriate place in the program structure. This way the programmer does not need to know that the syntax is *if <condition> then <block of commands> else <block of commands>*. He merely needs to look at the design of the graphical elements, see where they fit and try to put them there. The program can then be executed right away to see the effect.

Scratch allows for some advanced features too - programs can be edited while they are running. This is good for experimenting and fast debugging.

3.2 LabVIEW, a dataflow VPL

LabVIEW is an example of a professional visual programming language. It is used commercially and therefore represents a VPL of good scalability.

LabVIEW (short for Laboratory Virtual Instrumentation Engineering Workbench) is a platform and development environment for a visual programming language. The graphical language is named "G".

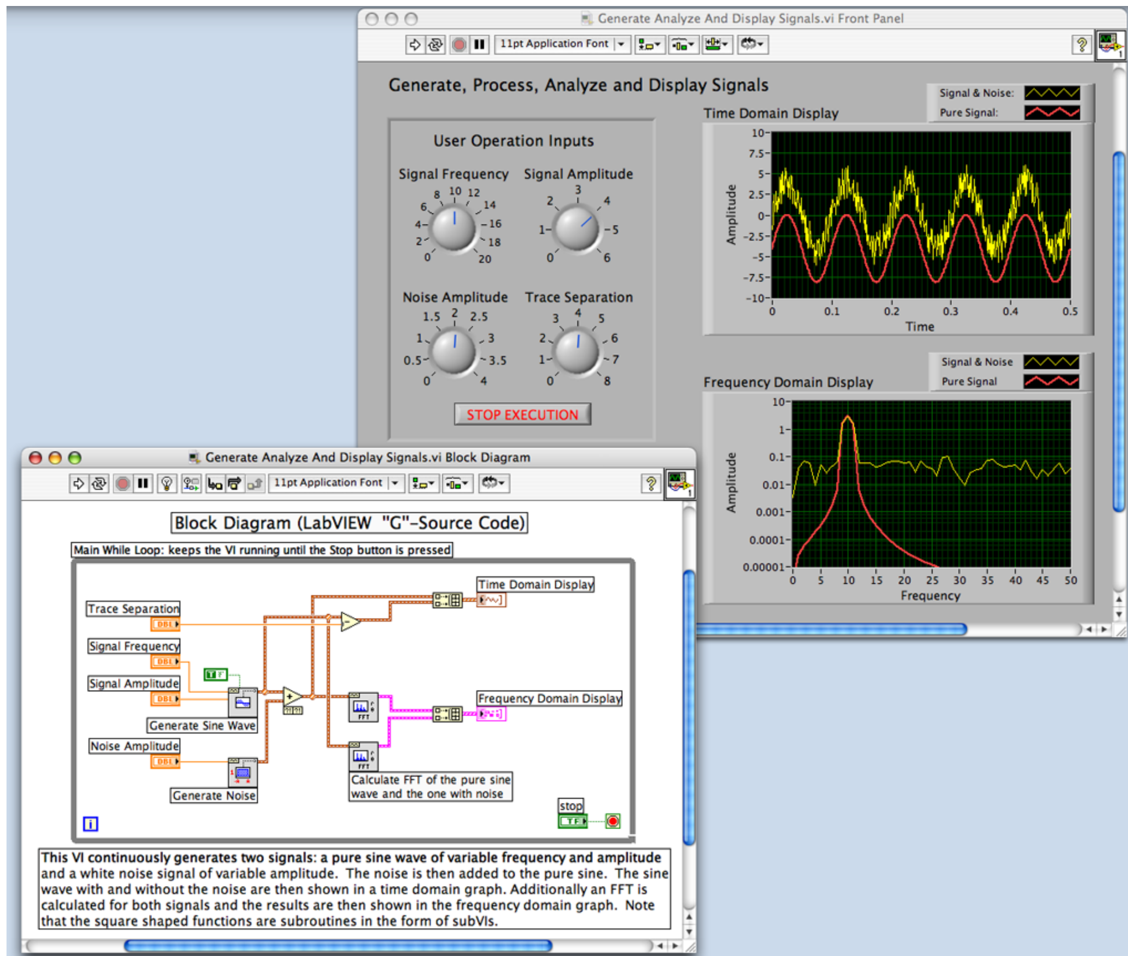

Figure 3.3: a simple LabVIEW program that generates, synthesizes, analyzes and displays waveforms

The programming language used in LabVIEW, also referred to as G, is a dataflow programming language. Execution is determined by the structure of a graphical block diagram on which the programmer connects different function-nodes by drawing wires. These wires propagate variables and any node can execute as soon as all its input data become available. Since this might be the case for multiple nodes simultaneously, G is inherently capable of parallel execution. Multi-processing and multi-threading hardware is automatically exploited by the built-in scheduler, which multiplexes multiple OS threads over the nodes ready for execution.

LabVIEW is commonly used for data acquisition, instrument control, and industrial automation. [12]

3.2.1 Dataflow programming languages

Dataflow programming languages are very popular amongst VPLs. They are based around functional programming languages and center around inputs and outputs (how the system transforms data), not so much on the internal data structers and algorithms used to achieve the transformation. Explaining them would stretch the topic too much. More details can be found in [4].

3.3 List of existing visual programming languages

In this section, a list of notable visual programming languages will be given with references. This is not a complete list, since new languages are adopted (and old are abandoned) daily. The languages in the list were mostly found on [13] (a bigger list can be found there; I just extracted the languages that I think are most notable).

- Alice - educational object-oriented language [1]
- Executable UML - a profile of the UML, that graphically specifies a testable and compilable system [14]
- FxEngine Framework - Open C++ dataflow programming for audio, video, signal, ... [3]
- G - the language based on data flow diagrams used in LabVIEW [12]
- NXT-G - a visual programming language for the Lego Mindstorms NXT robotics kit, also created by LabVIEW [5]
- Quartz Composer - a language for processing and rendering graphical data in OS X [6]
- Simulink - a commercial tool for modeling, simulating and analyzing multidomain dynamic systems, integrated with MATLAB [8]
- Scratch - educational VPL [7]
- SynthMaker - an audio programming VPL that allows creation of sounds, virtual instruments and effects [10]
- SourceBinder - a node based visual development environment for Flash 10+ [9]
- VSXu - an OpenGL-based, modular programming environment with its main purpose to visualize music and create 3D effects in real-time [11]

Chapter 4

Conclusion

Visual programming is an exciting, still developing field that has many potential. Visual programming languages are not merely a proof of concept, they are used in all fields - from education (Scratch) to professional programming (LabVIEW).

With the evolution of human/computer interaction new possibilities appear daily and it is important that the process of programming does not stay behind. Technologies like visual programming languages are a way to encourage this evolution of programming.

Bibliography

[1] *Alice*. URL: http://www.alice.org/.

[2] Margaret M. Brnett et al. "Scaling Up Visual Programming Languages". In: *Computer* 28 Issue: 3 (1995). ISSN: 0018-9162. URL: http://ieeexplore.ieee.org/Xplore/login.jsp?reload=true&url=http://ieeexplore.ieee.org/iel1/2/8390/00366157.pdf%3Farnumber%3D366157&authDecision=-203.

[3] *Fx Engine*. URL: http://www.smprocess.com/.

[4] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. "Advances in Dataflow Programming Languages". In: *ACM Computing Surveys* 36 (2004), 1–34. URL: http://synthmaker.co.uk/about.html.

[5] *NXT-G*. URL: http://en.wikipedia.org/wiki/Lego_Mindstorms_NXT#NXT-G.

[6] *Quartz Composer*. URL: http://developer.apple.com/graphicsimaging/quartz/quartzcomposer.html.

[7] Mitchel Resnick et al. "Scratch: Programming for All". In: *Communications of the ACM* 52.11 (2009).

[8] *Simulink*. URL: http://www.mathworks.com/products/simulink/.

[9] *SourceBinder*. URL: http://www.mathworks.com/products/simulink/.

[10] *SynthMaker*. URL: http://synthmaker.co.uk/about.html".

[11] *VSXu*. URL: http://vsxu.com/.

[12] *Wikipedia - LabVIEW*. 2010. URL: http://en.wikipedia.org/wiki/LabVIEW.

[13] *Wikipedia - Visual Programming Language*. 1010. URL: http://en.wikipedia.org/wiki/Visual_programming_languages.

[14] *xUML*. URL: http://en.wikipedia.org/wiki/Executable_UML.