Juri Babkin

VISUAL PROGRAMMING LANGUAGES

Programming Languages Research Seminar (MTAT.03.271), 3 EAP

Tartu 2012
## Contents

Introduction ................................................................................................................................. 4
Visual representation of data .......................................................................................................... 5
  Main concepts ............................................................................................................................... 5
Visual programming ideas ............................................................................................................. 5
Visual data structures .................................................................................................................... 6
  Program visualization ..................................................................................................................... 6
Visual programming language family ............................................................................................ 8
  Paradigms ........................................................................................................................................ 9
    Dataflow ........................................................................................................................................ 9
    Object-Oriented ............................................................................................................................. 9
    Functional .................................................................................................................................. 10
  Programming-by-demonstration ................................................................................................. 10
  Multi-paradigm ............................................................................................................................. 10
Features ........................................................................................................................................... 10
  Abstraction .................................................................................................................................... 11
  Control flow ................................................................................................................................. 11
  Event and exception handling ..................................................................................................... 12
  Visual structures and data types ................................................................................................. 13
Classification .................................................................................................................................. 13
  Purely visual languages ............................................................................................................... 13
  Hybrid systems ............................................................................................................................ 14
  Constraint-oriented systems ........................................................................................................ 14
  Form-based systems .................................................................................................................... 14
  Complete classification ................................................................................................................. 14
Advantages ...................................................................................................................................... 16
Disadvantages ............................................................................................................................... 16
Overview of modern VPLs ............................................................................................................. 18
Scientific field .................................................................................................................................. 18
  LabVIEW ...................................................................................................................................... 18
  DRAKON .................................................................................................................................... 19
**Introduction**

Programming is the activity of representing algorithm based operations in a formal notation, which is known as programming language, with the purpose of executing it on the computer to achieve desired behavior and functionality. The history of computer programming languages starts in 1940’s with the advent of the first electronic digital computers. The programming languages themselves at that time were very complex and strictly based on the underlying architecture. However within the next decade were introduced new ways of programming called “problem-oriented” or “object-oriented” approach.

The vast majority of the programming languages are represented by the text. The structure of the casual program is based on the collection of strings (mostly linear). The code itself could be visually formatted using tabulation and different markers added by IDE, but it remains one-dimensional and considered as textual. With the growth of the computer graphics hardware and rendering software in the 1960’s it became possible to represent data more visually and in two- or three-dimensional way. This contributed to the beginning of the visual programming languages (VPLs).

The main scope of this thesis is to research the field of the visual programming languages and bring out basics of language family including different features, paradigms and classification, make an overview of the existing languages and bring some examples. In this scope will also be included benefits of using visual languages instead of textual ones and areas, where and with what purpose VPLs could be used.
Visual representation of data

Main concepts

It is very important to represent data or some sort of information in a completely comprehensible way, so user would understand as much as possible at the first sight. It is well-known fact that human’s visual perception is the strongest feeling and it gives better identification and interpretation of the information from the environment than any other sensory mechanism.

Basically there are two main methods of visual representation of data: graphically and textually. Often people are using combination of them in order to explain something. If we look at the science field, we would see that most of concepts or ideas in different areas are presented as text: books, studying materials, journal, articles, etc. Of course there could be pictures, tables and diagrams in them, which help to understand described subject more clearly, but the main carrier of the information is still a text. In the computer science field in the other hand is slightly different, by the mean of using computer the information could be represented in a very different way. Business models, algorithms, charts, graphs, spreadsheets etc. – it is possible to visualize all of them for better perception.

Textual programming languages have proven to be rather difficult for many creative, intelligent people or children to learn to use effectively [1]. However nowadays it is absolutely possible to teach those people the basic ideas of data manipulations and simple programming techniques by the using of the visual concepts and visual programming languages.

Visual programming ideas

The field of visual programming has grown from a confluence of work in computer graphics, programming languages and human-computer interaction. A visual language is a programming language which uses a primarily visual (graphical) notation. [2] It often referred as system that allows user to specify a program in a two (or more) dimensional way. Conventional textual languages are not considered as two dimensional, because their compilers or interpreters are using long one-dimensional streams.
The main idea of the VPL is that it allows user to create executable program by manipulating graphical elements, using visual expressions, spatial arrangements and relations of text and graphic symbols. The syntax or notation of each VPL could be different, but the common feature is using “boxes” as objects representation and “arrows” as connection of those objects, which represent relations.

**Visual data structures**

In order to fully understand how VPLs are designed and structured, we need to get acquainted with basic data structures they are using for manipulating within a program. Described above features of “boxes” and “arrows” are abstract notions and could be specified further.

The primitive types of the VPL are often the same as in any traditional language. It is hard enough to imagine the representation of an integer value rather than as an ordinary number. As an exception, boolean type could be represented in more visual way, i.e. indicator of the some element would be green if it has true-value and red if it is false-value.

Representation of the composite or an abstract data types could be completely different from the ones that are used in a traditional programming language. For example, in case of hash-map or linked list, it is very easy to show relations between elements using arrows. A casual user, which started learning programming or do not understand it well, could not often imagine graph or tree structure. It is not clear right away by the look to i.e. Java or C++ code, how exactly nodes are connected together. If we are using graphical expression of a graph/tree it is quite simple.

**Program visualization**

Program visualization is a completely different concept of visual programming, but those two terms are often considered as VPLs. In real visual programming language the graphical elements are used to create program itself, but in the visualization the program is specified in a traditional textual manner and graphic is needed to show or illustrate aspects of it or the runtime execution. There are two characteristics that describe program visualization: what part of program could be illustrated (code, data or algorithm) and type of behavior (dynamic or static).
Data visualization systems show snapshots of actual data of the program. In a similar way code visualization system represent program code, by adding graphical marks to it or converting it to visual form (e.g. flowcharts or diagrams). Algorithm visualization could be very abstract - without showing user the code or actual data that corresponding function or method is processing by the moment and it is often simplified with animations for better understanding.

Dynamic visualization allows user to see the action of the program being processed as an animation, static uses ordinary pictures or static elements instead. [3]
Visual programming language family

Different visual programming languages were created for different purposes (general purposes, database oriented, image-processing, scientific visualization, user-interface generation [4]), but it is possible to distinguish general tendencies in all of them. The first and probably the most important goal is to make programming more accessible for some particular user audience. Most of VPLs are used to solve field or domain specific problems. The second goal is to improve speed of programming. The last one is simplicity to use or understand.

To achieve these goals, there are four approaches are used [5]:

1. Concreteness. Concreteness refers to the use of particular instances rather than using abstract ones. Abstractions could not be represented visually. Programmer should be able to specify some aspects of semantics on a specific object. In other words “visual code” of the program follows WYSIWYG principle (“What you see is what you get”).

2. Immediate visual feedback. Programmer should see automatic and immediate change of semantics of some object, when he edits “visual code”. Immediate feedback is sometimes described as program liveness [6].

3. Directness. Directness refers to concept of cognitive perspective of programming and means that there is a “short distance” between programmer’s action and system response to it. As an example programmer should be able to manage object (or its properties) directly to specify semantics rather than describing semantics textually. A good example of directness is a rule-based VPL in which the rule set directly represents how the output of a process or action is implied by detecting a condition in the rule domain, as in AgentSheets or Cocoa [9]. Another good example of directness is the graphical representation of class hierarchies in Prograph [10].

4. Explicitness. Semantics is explicit if it is stated in the environment and programmer should not infer it.
Paradigms
VPIs are using different paradigms. Paradigm is a way of conceptualizing, defining the organization and structuring of the calculations performed by the computer. In this section will be discussed the most important ones.

Dataflow
The dataflow paradigm is currently the approach to visual programming used most widely in industry. In the dataflow execution model all processing is performed by the instructions that are applied to values. An instruction could be executed as soon it gets all the required input-data [7]. This execution could be also defined as dataflow graph. Example of such graph is showed on the Figure 1. Using this paradigm it is possible to perform parallel calculations, which gives an opportunity of speeding up the whole program execution.

The dataflow model uses two different approaches - *Data driven approach* and *Demand driven approach*. Data driven model execution depends on available data at the input of instruction. In case of demand driven approach each instruction “ask” for input data from another instruction. Therefore, program execution starts from “the end”.

Visual dataflow technique could be represented by multidimensional structures that represent instructions or objects and links between them, which illustrate directions of the flow.

Object-Oriented
Object-oriented programming (OOP) is a paradigm that uses objects (as instance of a class), which are consisting of data fields and methods for their interaction. OOP provides set of features that could be used in a VPL, i.e. polymorphism, inheritance and encapsulation. Describing those principles is out of the scope of this research.

Figure 1. Example of the dataflow graph.
It is possible to represent OOP structure visually by the means of “Program Visualization” technique described in the first section. Class browsers, data structure visualizers or domain-specific execution visualizers could be considered as an example of this visualization [8]. Those approaches allow programmer to see object-oriented structures, hierarchics and interactions visually, but they are not purely visual object-oriented programming languages. Although, the latters are also exist.

**Functional**
Functional programming is a programming paradigm that treats computations as evaluation of mathematical expressions and avoids using states and mutable data. In VPLs functions could be represented as visual structures with I/O pins that interconnected together. Functions could also be defined as a “black boxes” and reused in other functions.

**Programming-by-demonstration**
Programming-by-demonstration is a programming paradigm in which programmer teaches system or robot new behavior by demonstrating the task to transfer directly instead of specifying actions through instructions or commands. In VPLs a programmer through user-interface manipulates visual objects and the system performs tasks according to those manipulations. Actions can be performed “live” or recorded and aggregated to some function and called in a different context.

**Multi-paradigm**
Most of VPLs are based on few paradigms. It helps to improve language by extending its boundaries and makes it easier to use and gives possibility to design more complex structures when it is needed. What paradigm (or combination) could be selected is often determined by application domain.

**Features**
Some of VPLs are using different features as an extension to the paradigm in order to extend programming facilities and simplify programmers work. The following features could be used: data or procedural abstraction, control flow, documentation, event handling, exception handling, data types and structures.
**Abstraction**

Abstraction improves scalability of the VPL. Low-level procedural abstraction allows programmer to combine visually represented logic into procedural modules. In high-level VPLs most of the abstraction logic is implemented by non-visual modules (hidden from programmers) combined as visual component. For example in LabVIEW [14], function could be represented as a “black boxes” with input and output streams, besides, those “boxes” could be enclosed one into another constructing nested chain of functions. General-purpose VPLs cover entire base of the abstraction using both approaches described above, allowing combining of the visual code into procedures, classes, libraries etc.

Data abstraction means that the programming language include user-defined data types, data inheritance, and data encapsulation, placing restrictions on the use of the operators, having rules that specify the visibility of data, and separating the specification from the implementation [7]. The notions of data abstraction in VPLs is very similar to the same notion in conventional programming languages, with the only difference, that abstract data types should be represented visually (in the form of icons, graphs, charts, boxes, arrows, etc.) instead of textually, and provide corresponding behavior.

**Control flow**

Visual programming languages cover both declarative and imperative notions. Declarative programming approach implies expression of the logic by computation rather than describing control of the flow. In the context of VPLs single assignment of the object is used when explicit state modification is avoided: a programmer creates a new object by copying an existing one and specifying the desired differences, rather than modifying the state of the existing object. Also, instead of specifying a sequence of state changes, the programmer defines operations by specifying object dependencies.

Imperative programming style on the other hand relies on the controlling the flow. Program constitutes one or more control-flow or data-flow diagram - how the thread of control flows through the program. This approach could give effective visual representation of the
parallelism. The main disadvantage of imperative approach is that programmer should deal with how sequencing of operations modifies the state of the program.

**Event and exception handling**

Event handling may be implemented in different ways in the VPLs. Possible implementations include single or multiple event structures. A single event structure can manage only one event at a time. Advanced event handling for multiple events is using an *event switch* technique and/or dynamic computation node (DNC) principles. Event firing mechanism could be either automatic (predefined state of object or property or system interrupts i.e. system clock ticks, input from a sensor port or other hardware-level interrupts) or based on user-computer interaction (clicks, selections, drag’n’drops, value changes, etc.) Example models of the event switch and DNC used in LabVIEW [14] are shown on the Figure 2 and Figure 3. [7]

![Figure 2. Event switch (LabVIEW)](image1)

![Figure 3. Dynamic computation node (LabVIEW)](image2)
Exception handling is the process of responding to exception occurrences. In VPLs like in conventional programming languages, it is possible to specify an exception for procedure or function and a subroutine to handle it. Both event and exception handling structures could be represented in the form of frames, icons or in some other visual way.

**Visual structures and data types**
Most of VPLs rely on visual structures and data types. Basic concepts behind primitive data types are the same in every VPL, but representation could be different. Specific data types and structures could be found in the domain-specific languages and designed according to requirements. In data-flow languages for example, there are always interconnected nodes that represent states of program and flow direction. In VPLs that support data abstraction programmer is able to create his own data types (user-defined data type) using encapsulation and/or inheritance.

**Classification**
Visual programming languages became a mature field in the IT sector in the past decades, therefore standardization and classification was needed to productively work in this area. Such classification helps to find related publications and gives overview for comparison or evaluation of different systems/languages. A significant contribution to the creating and improving such classification was made by Chang Shu [11] and Burnett [12]. Most distinctive categories are described below.

**Purely visual languages**
Purely visual languages are characterized on the grounds of using visual techniques in the programming process. The programmer manipulates icons or other visually represented objects in order to create program, whereas debugging and execution processes are taking place in the same visual environment. The program is compiled directly from its “visual code” and is never translated into an intermediary text-based language. Examples of such completely visual systems include VIPR [1], Prograph [7], and PICT.
Hybrid systems

Hybrid languages (or systems) combine both textual and visual elements. Hybrid languages include both those in which programs are created visually and then translated into underlying high-level language and languages which involve the use of visual elements in a conventional textual language. An example of that kind of language includes Rehearsal World [15], where the user “teaches” the system to solve a specific problem by manipulating graphical “actors,” and then the systems generates a Smalltalk program to implement the solution [1].

Constraint-oriented systems

Constraint-oriented systems are widely used in the simulation design, in which a programmer models physical objects as objects in the visual environment which are subject to constraints designed to simulate the behavior of natural laws, like gravity force. Constraint-oriented systems have also found application in the development of graphical user interfaces (GUI). Thinglab [16] and ARK [1] are both primarily examples of constraint-based languages.

Form-based systems

Some visual languages inherited their visualization and programming style from spreadsheets. They could be classified as form-based languages representing programming as altering a group of interconnected cells over time and often allow the programmer to visualize the execution of a program as a sequence of different cell states which progress through time. Forms/3 [1] is an excellent example of a form-based language.

Complete classification

According to [12] a complete classification of VPL field could be represented as follows:

I. Environments and Tools for VPLs
II. Language Classifications
   a. Paradigms
      1. Concurrent languages
      2. Constraint-based languages
      3. Data-flow languages
      4. Form-based and spreadsheet-based languages
      5. Functional languages
      6. Imperative languages
7. Logic languages
8. Multi-paradigm languages
9. Object-Oriented languages
10. Programming-by-demonstration languages
11. Rule-based languages

b. Visual Representations
   1. Diagrammatic languages
   2. Iconic languages
   3. Languages based on static pictorial sequences

III. Language Features
   a. Abstraction
      1. Data abstraction
      2. Procedural abstraction
   b. Control flow
   c. Data types and structures
   d. Documentation
   e. Event handling
   f. Exception handling

IV. Language Implementation Issues
   a. Computational approaches (e.g. demand-driven, data-driven)
   b. Efficiency
   c. Parsing
   d. Translators (interpreters and compilers)

V. Language Purpose
   a. General-purpose languages
   b. Database languages
   c. Image-processing languages
   d. Scientific visualization languages
   e. User-interface generation languages

VI. Theory of VPLs
   a. Formal definition of VPLs
   b. Icon theory
   c. Language Design issues
      1. Cognitive and user-interface design issues (e.g. usability studies, graphical perception)
      2. Effective use of screen real estate
      3. Liveness
      4. Scope
      5. Type checking and type theory
      6. Visual representation issues (e.g. static representation, animation)
Advantages
There are many advantages of using VPLs instead of conventional textual programming languages. The main argument in the favor of VPL is the fact that human brain process graphical information easier and faster than textual. Text is one-dimensional structure, while pictures are multi-dimensional and allow representing information in compact encoding with the help of visual properties like color, size, form, size and direction. Pictures provide random access to any part, as well as detailed and overall views, while textual information always has sequential nature. Pictures provide a concrete notion, which from it is easier to acquire an abstract idea.

VPLs make programming more accessible to users. An ordinary user, who does not have much knowledge or experience in programming, data structures or algorithms, is able to create simple programs in a very short time. Children, students and theoretical-scientists are main audience, who successfully grasps basics of the programming by the use of VPLs.

Another advantage of using graphics is tendency to be a higher-level description of the desired actions (providing a higher level of abstraction) and may therefore make the programming tasks easier even for professional programmers. VPLs especially useful through the debugging process, where programmer can visually follow the program’s execution flow and see current state more detailed than in conventional language. Also, some complex programs with the concurrent processes (or even real-time processes) are difficult to describe textually, so graphical representation gives more appropriate overview. [3]

Since many VPLs use visual representation, editors for these languages usually have a “direct manipulation” user interface. This gives user an impression of more directly constructing a program rather than having to abstractly design it.

Disadvantages
The main issue of the VPLs is support of growing programs. For example, some of the visual mechanism used to achieve characteristics such as explicitness can occupy a great amount of space, what makes a program harder to maintain. It is also hard to use some of programming
techniques developed and improved for traditional languages, because doing so often results in reinvention of complexities that VPLs are meant to reduce. [3]

VPLs need special editors and environments. Most of the conventional programming languages allow writing programs in ordinary text editors, only compiler and possibly debugger are needed. Currently each of VPL requires its own editor and environments, because there are no general purpose Visual language editors or even visual semantics. Programs created using graphical language cannot be easily open without appropriate software, it means that editable source code is hard to send by an e-mail or share on the web page. Programming solutions taken from one VPL cannot be reused in another due to incompatible structure and syntax.

Most of VPLs are interpreted, and in many cases, the interpreter operates on the same level (or very close to it) as picture representation. This results in poor performance of the runtime i.e. low execution speed.

Most of existing VPLs are latently typed, which means that they check whether an operator (visually represented element) receives value of a correct type only when this operator is applied (during runtime). Verification of the type mismatch not performed during compile-time leads to error detection only by trial-and-error technique, which is very inconvenient. [2]
Overview of modern VPLs

Scientific field

VPLs are widely used in the science area. The main reason for this is simplicity to understand programming methods and develop simple solution for specific problems. Scientists often do not have enough time resources to learn conventional programming on the sufficient level, so special visual environments were developed to satisfy requirement of a simple and easy-to-learn techniques.

LabVIEW

LabVIEW is the most popular solution and very powerful tool for data acquisition, instrument control, and industrial automation aimed to be used by technical engineers. It is a commercial product used by different organizations, institutes and universities to perform work with data from the external environment.

LabVIEW, sometimes also referred to as G, is a purely visual dataflow language. Execution flow is determined by graphical block diagrams on which user interconnect function-nodes by wires. Wires propagate data like variables, exceptions, events. Node can start execution as all input wires are propagated with data. G is also capable of parallel execution. Multi-processing and multi-threading hardware are utilized automatically with the built-in scheduler, so
user does not have to specify parallel processes manually.

Variety of features is available for programmer to use. Many libraries with a large number of functions for data acquisition, signal generation, mathematics, statistics, signal conditioning, analysis, etc., along with numerous graphical interface elements are provided in several LabVIEW package options. LabVIEW modularity allows code re-use: as long as the data types of I/O are consistent, two sub-VI$s$ (nodes, representing “black boxes”) are interchangeable. Environment allows creating stand-alone executable files that do not need any external libraries or run-time engines (those could be included to executable).

**DRAKON**

DRAKON (acronym from Russian for “Friendly Russian algorithmic language that guarantees clarity”) is an algorithmic visual programming language developed for the Buran space project. Its development, started in 1986. The goal was to replace specialized languages PROL2, DIPOL and LAKS used in the Buran project with one universal language.

DRAKON rules for creating diagrams are cognitively optimized for easy comprehension, making it a tool for intelligence augmentation.

**Robotics field**

Robotics field is quite young but it has very complex basis underneath, including physics and mechanics along with differential calculus. Visual language gives a good way to introduce pupils or students to robotics and teach them easy principles of programming robots without touching complex mechanics or hardware.
NXT-G

NXT-G is a graphical programming environment that comes bundled with the NXT - Lego Mindstorm Robotics Kit. NXT-G is used for real-world programming for robots, allowing programmer to specify actions and behavior of the hardware components like motors, sensors, actuators, etc. The language supports virtual instruments for all LEGO branded and most 3rd party sensors/components.

Microsoft Visual Programming Language

Visual Programming Language is an integral part of Microsoft Robotics Developer Studio environment, which makes possible to build, test and debug robotics applications, without having to make any assumptions about the underlying hardware. VPL provides a relatively simple drag-and-drop graphical tool that helps make it easy to create and maintain robotics applications. VPL also provides the ability to combine a collection of connected blocks and reuse them as a single block elsewhere in your program. VPL is also capable of generating human-readable C# code.
Summary

Visual programming languages and visual environments are aimed to satisfy the process of programming by incorporating images and other meaningful graphical representation of data, algorithms and tools. VPLs use graphics as part of their syntax to express data constructs and relationships between them, while environment manipulates the symbols and notations of the language.

Using visual representation instead of textual has many advantages, including better understanding of programming techniques for non-programmers, direct data manipulation, visual debugging and some others. Many VPLs are domain- or problem-specific, which makes impossible to create general-purpose programs. This is the main weakness of visual languages.

This research shows the wide range of programming paradigms, features and principles used in VPLs, which describes ability to be exploited in the different areas and enhance the power of computer programming in general.
References