TiViPE – Tino's Visual Programming Environment

Conference Paper · October 2004

DOI: 10.1109/CMPSAC.2004.1342799 · Source: IEEE Xplore

1 author:

Tino Lourens
TiViPE
69 PUBLICATIONS **925** CITATIONS

SEE PROFILE

TiViPE - Tino's Visual Programming Environment

Tino Lourens
Honda Research Institute Japan Co. Ltd.
8-1 Honcho, Wako-shi, Saitama, 351-0114, Japan
E-mail: tino@jp.honda-ri.com

Abstract

TiViPE [4] is a component based visual programming environment (VPE) that enables users to build programs by construction of a network of components interactively. A single module (component), represented by a graphical icon, is a computational unit. Multiple icons can be connected to each other to yield a directed graph (a network) that represent a program. TiViPE is, in appearance similar to programs such as AVS [9], Vee [3, 1], OpenDX [6], Khoros [12], LabVIEW [7], NeatVision [10], and ViPEr [8], but presents some fundamental differences. TiViPE integrates documentation with an existing routine call (that has been programmed in C++, C, Fortran, or Java), and automatically generates C++ code that is compiled to stand-alone program. This program is able to execute the specified routine, provide a graphical icon, or give html-formatted documentation about the routine. Hence, within TiViPE there is no textual programming for the user. TiViPE strongly re-uses code, which is inherent to visual programming, and automatic code regeneration by compounding a network of modules to a single module, which leads to faster programming. TiViPE supports networking and parallel processing in a natural way, and allows the user to modify an activated network. TiViPE also aims at rapid prototyping which demands user friendliness, programming by existing modules for basic users, and focuses on the documentation of a module. TiViPE has been used in the field of computer vision, robotics, and computational neuroscience.

1 Introduction

Visual programming has been proven to be more efficient than classical textual programming. There is a strong increase in program development (up to five times faster) [1]. It increases the productivity of both researchers and application developers regardless of their programming experience [12]. One aspect of increased efficiency is that a visual notation provides a intuitive organization and can make information explicit [11]. It also is due to natural software re-use, i.e., by using the same (graphical) module multiple times. Another advantage is the possibility to bring existing programs and routine calls programmed in different textual programming languages, together into a consistent, standardized, and cohesive environment.

The number of users of visual programming languages, however, is negligible compared to the users of textual programming languages, like C++, C, Java, etc., despite the advantages. Domain specific visual programming languages (VPLs) such as LabVIEW which allows to import existing textual code [7] have been more successful than general-purpose VPLs like Prograph [2]. A VPL that conforms to a textual language standard has a much greater likelihood of acceptance. An important reason that visual programming has hampered is due to the migration process from a textual program to a visual program. The concept of TiViPE is to make such migration process simple. TiViPE wraps any routine into a graphical module by filling out a few forms about the routine. By pushing a single button all code is generated, compiled, and integrated in the environment, without additional programming. This makes the migration process fast and easy. It makes TiViPE a component based visual programming environment (like, for instance, Java studio) that employs an information flow with unlimited programming capabilities.

The paper is organized as follows: Section 2 provides an overview of several different VPEs. Section 3 introduces TiViPE, elaborates on the concept of visual programming, and gives an overview of the modules that are currently available within TiViPE. The paper finishes with a discussion.

2 Overview of Visual Programming Environments

A widely used graphical programming environment is LabVIEW [7] which is especially oriented to hardware. Its G programming language which is used to construct block

diagrams provides a powerful tool to interact with hardware directly. HP Vee [3], which is currently known as Agilent Vee [1] aims at a similar group of users as LabVIEW. Vee can execute modules by command line and has a flexible data structure. The graphical user interface is within the icon itself, which is helpful if modules are relatively simple and networks are small, otherwise the overview of the network is lost. IBM's OpenDX [6] focuses on visualization, and includes around 180 different tools for data analysis and visualization. OpenDX is open source software and is supported on many different platforms. The weak point in openDX is the user friendliness, for example no datatype is given between the connections, which implies that the documentation needs to be addressed extensively. Khoros [12] provides a programming environment with a small set of modules. Due to this small set of modules, new module development is a must. Khoros provides tools to construct the graphical user interface for a module, nevertheless still a considerable textual programming effort is needed to construct a working icon within VPE Cantata. A strong point of Khoros is that a network, constructed in Cantata, can be executed as a batch file (without the use of Cantata), since all modules are represented as executables. AVS [9] provides a programming environment with a large set of different modules. The construction of a working icon in AVS requires a considerable effort also. AVS has very strong visualization tools and has business solutions in many different areas. Drawbacks of AVS are its dependency on the environment and its inflexibility to user constructed data structures. ViPEr [8] is a visual programming language that uses Tcl and Python scripting. The main advantage of these scripts are that they are platform independent. ViPEr is clearly inspired on AVS and has a fixed set of data structures. Like AVS the data types are clearly distinguishable. NeatVision [10] is a Java based VPE meant for image processing. It currently contains around 200 different image processing and analysis functions. It is an intuitive environment that uses simple icons containing low-level routines, but it is suitable solely for image analysis.

3 TiViPE

The main drawback the author experienced by using AVS and Khoros in the past decade was that both environments need a considerable effort to construct a graphical module. It is one of the main reasons why users are not migrating to a VPE.

Much of the effort in developing TiViPE has gone in adding a *module* or a *data structure* together with their *documentation* to the environment. Wrapping existing textual routines, that are programmed in C++, C, Fortran, or Java, into a graphical unit is performed without additional programming. The conditions firstly are, that a routine has been programmed in a language that supports routines with data typed parameters, and secondly that the routine is available in a library. The construction of a module is a matter of filling out a set of forms. Six of these forms are about documentation, the other four are: the name of the routine, the used parameters of the routine, the names of the used libraries, and the names of the used include files.

An important difference with for example LabView is, that a user is able to add or delete a module (icon) from a running program, where a program is represented by a network of graphical icons. Depending on the structure of the network, modules can be restarted automatically. The latter is essential in robotics and real world parallel processes.

Strong focus has been on the user friendliness of TiViPE, i.e. all actions, parameter modifications, and the html documentation can be accessed within two button clicks on the graphical icon. TiViPE provides:

1. GUI construction and modification of modules and data structures.
2. automatic code generation and compilation to a stand-alone executable of a *module*; that is able to execute the routine provided by the user, provide html documentation both in TiViPE and on the command line, provide the graphical user interface to TiViPE, and give a usage message on the command line.
3. automatic code generation and compilation of a *data structure* to a library that contains a set of basic operations that can be performed on the data of the structure: initialize, set, check availability, delete, read, write, and retrieve html documentation about the structure.
4. simple icons with full functionality, given by 4 action buttons.
5. input and output data types that are distinguished by different colors.
6. build in networking and parallel processing facilities.
7. the possibility to compound a network to a new module by scripting or automatic code regeneration.
8. the possibility to have I/O through file or memory interaction.

TiViPE contains a small set of almost 60 modules for information processing, data exploration, and data visualization. With these (general) modules, problems can be solved in different application fields. TiViPE provides multi-dimensional, data manipulation operators including pointwise arithmetic, data conversions, data organization, and size operators, and includes a set of specific image processing routines.

TiViPE allows a user to run a module independently from the environment, therefore all modules are available as stand-alone executables. Compounding modules of a network into a single module will be performed by regeneration of code from the existing modules. The user will have the choice to make a selection between multiple processes

or threading. This is different from both AVS and Khoros that are designed to batch these modules. Hence these environments still contain a considerable computational overhead. Compounding modules by batching has the advantage that one can easily modify the structure, while for every modification in TiViPE of a compounded network code generation and compilation needs to be performed.

3.1 Environment

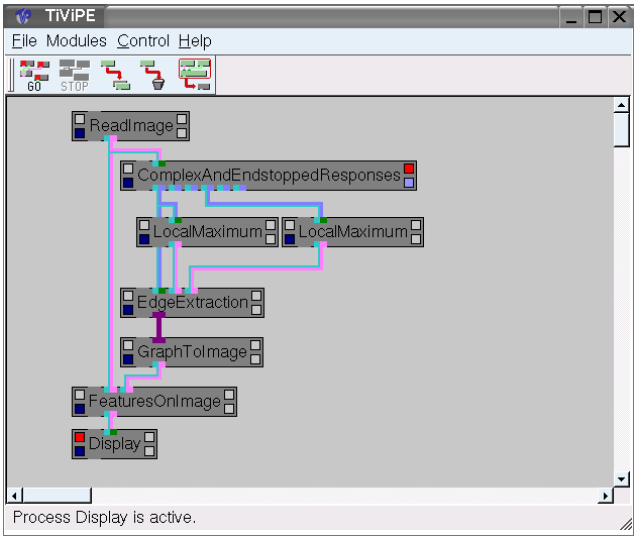


Figure 1. TiViPE with an example.

Figure 1 illustrates TiViPE with a sample network. This network extracts a contour graph from an input image by using modeled complex and endstopped cells of monkey primary visual cortex, areas V1 and V2, and visualizes its result. For details about the model, see [5].

A single module is a stand-alone executable with input, local, and output parameters. Every module contains four internal buttons, see Figure 3a:

- The *left upper* button is the *execute* or *stop* module button. Red denotes active, grey inactive.
- The *left lower* button indicates on which *machine* the module is being executed. Dark blue denotes the local host, other colors that are provided by the user represent a different machine or processor.
- The *right upper* button pops up or closes the *parameter window* upon pressing; Red indicates a visible parameter window, grey means invisible.
- The *right lower* button pops up or closes the *module information* window upon pressing; Grey means no information, light blue means information available, purple denotes an aborted routine. Red indicates a visible information window.

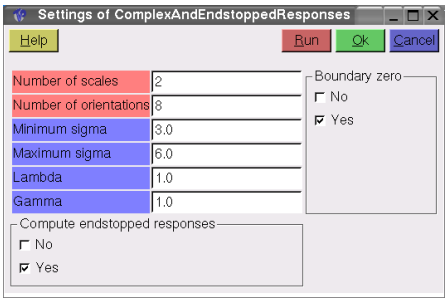


Figure 2. Parameter window of module ComplexAndEndstoppedResponses, which is illustrated in Figure 1.

A parameter window for the module *ComplexAndEndstoppedResponses*, which contains 8 parameters, is illustrated in Figure 2. The window gives a quick overview of all internal parameters that can be modified within the module. TiViPE supports a small set of parameters, three of them (integer, float, and toggle) are given in this example.

On top of every parameter window are four buttons. The run button saves the parameters and executes the module when the module is finished it will automatically activate the subsequent connected modules, the ok button saves the parameters only, and a cancel button resets the parameters to the formerly saved state. These three buttons yield rapid prototyping in a network as well as rapid testing for the module itself. The help button pops up a window with html documentation about the module. Its content has been partly provided by the user through the available documentation tabloids.

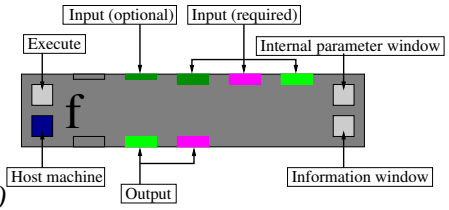
3.2 Visual Programming

The TiViPE concept of visual programming is to wrap a routine that is constructed in a textual programming language into a graphically represented unit. The main reason for this concept is that there are many software libraries available, but due to lack of documentation and a clear structure most of these libraries remain unused. By making all routine calls available as graphical units together with appropriate documentation these routines can be made accessible through TiViPE.

In a C or C++ program a routine call is typically given by a name and between brackets its parameters. These parameters can be divided into 3 groups: *input* parameters that are obtained from another routine, *internal* parameters that are needed within the routine only, and *output* parameters that are used by at least one of the subsequent routine calls. Such routine in its general form is as follows:

$$f(i_1,\ i_2,\ \ldots,\ i_N,\ o_1,\ o_2,\ \ldots,\ o_M,\ p_1,\ p_2,\ \ldots,\ p_L)\ , \tag{1}$$

where f is the name of the routine, and i, o, p the input,



a)

```
// Read input data
i0 = guiGetInput (”Input0”, ANYTYPE, OPT);
i1 = guiGetInput (”Input1”, ANYTYPE);
i2 = guiGetInput (”Input2”, UCHAR);
i3 = guiGetInput (”Input3”, LONG);
p0 = guiGetInteger (”P0”, LONG);
p1 = guiGetFloat (”P1”, UCHAR);
// Routine call
f (i0, i1, i2, i3, p0, p1, o0, o1);
// Write output data
writeData (guiGetOutput (”Output0”), o0);
writeData (guiGetOutput (”Output1”), o1);
```

b)

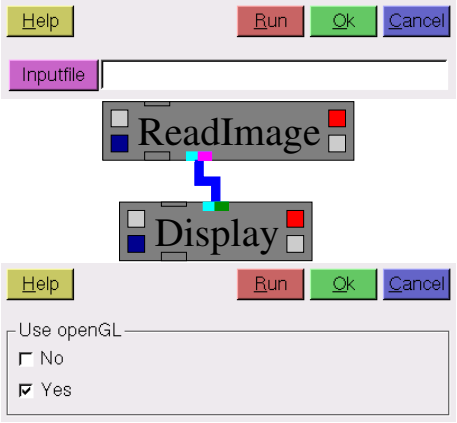
Figure 3. *a)* **Graphical representation of the routine given in (1) with an explanation of the differently positioned buttons.** *b)* **C or C++ equivalent, modified from the code generated by TiViPE and reduced to the relevant calls.**

output, and internal parameters. A graphical representation of this routine call is given in Figure 3a. A module is graphically represented by an icon which is a gray rectangle with an appropriate name, which is mostly the name of the routine call. A number of $N + 1$ small colored rectangles are given at the top of the rectangle, they represent the input parameters. The additional left most unobtrusive grey rectangle is a control flow parameter to direct the activation flow. The control flow might be needed when there are no input or output parameters, or when parallel processing flows need to be avoided. At the bottom of the rectangle are $L + 1$ colored rectangles that represent the output parameters. Again, the first rectangle is used to control the activation flow. Four internal rectangular action buttons complete the icon.

The advantage of a graphical representation over the textual routine call is that the three types of parameters are naturally subdivided. Also different data types are immediately visible due to their difference in color. The internal parameters can be visualized by clicking the upper-right internal button of the icon, a parameter window will pop up, see also Figure 2, for an example.

Different routines in a textual program are given by a sequence of routine calls, in a graphical programming environment these modules are connected by a directed graph,[1] where the vertices represent the icons and the edges represent the connections between the icons. An example program of reading and displaying an image in TiViPE represented by two connected icons is illustrated in Figure 4a. Its textual equivalent, as generated by TiViPE, in Figure 4b. This figure demonstrates the differences between the two ways of programming. The icon's name indicates the functional role of the routine. The execution order of the icons is determined by the way the modules are connected, i.e., output is connected to input. The icons hide all unnecessary details of a textual program, which makes a network of icons much more intuitive than a textual program.

[1] A graph is a set of vertices and a set of tuples of vertices that are called edges.



a)

```
// ReadImage
TVPgen_ReadImage m1 (m1argc, m1argv);
m1.g = new GUI (m1argc, m1argv);
m1.v1 = m1.g->guiGetIO (”Inputfile”);
m1.cn = new ImageIO ();
m1.v0 = m1.cn->ReadImage (v1);
// Display
TVPgen_Display m2 (m2argc, m2argv);
m2.g = new GUI (m2argc, m2argv);
v1 = m2.g->guiGetToggle (”Use openGL”);
m2.cn = new TVPDisplayInterface ();
m2.cn->TVPDisplayData (m1.v0, v1);
```

b)

Figure 4. *a)* **Graphical program that reads and displays an image. The “ReadImage” parameter window on top allows the user to specify the filename. The “Display” parameter window at the bottom allows the use of openGL.** *b)* **Textual (C++) equivalent of the network, modified from the code generated by TiViPE.**

3.3 Architecture

The functional architecture of TiViPE is presented in Figure 5; modification of a *module* is at the left and modification of a *data structure* on the right. Since both use a similar concept only module construction or modification will be discussed in more detail.

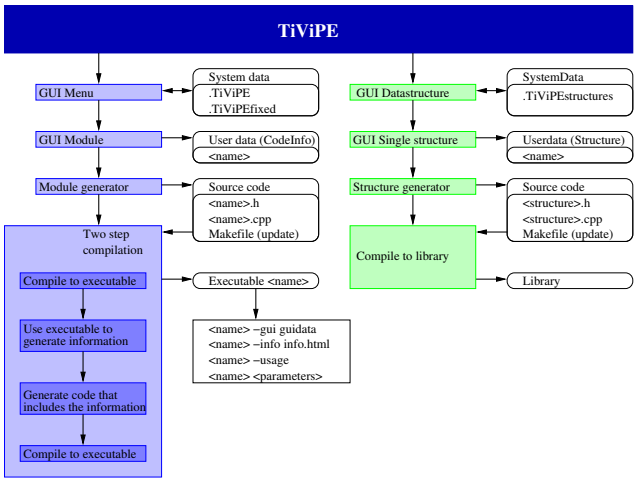


Figure 5. Functional architecture of TiViPE.

All module entries start with "GUI Menu" (see Figure 5), where a specific module is selected from an existing list that has been obtained from the "System data" files. When a single module is selected "GUI Module" will be activated. It provides the user with a set of ten forms (6 for documentation and 4 to embed a routine call) that can be edited. The data of these forms is stored in, and retrieved from a so called CodeInfo file. The name denotes a blend of *source Code* and *Information*.

Renaming, deleting, and moving a module starts with the "Module generator", which generates or/and deletes files, depending on the type of action. When files are generated, the content of the CodeInfo file is used to generate the C++ source code. After code generation has been completed a two step compilation takes place which results in a stand-alone executable.

The file structure used by TiViPE is given in Figure 6. Contributions of different users are expected, hence the file structure needs to be divided into two parts: a *private* (user) part and a *public* (general) part. Users initially develop modules in their private directories given by $MYTVPHOME. When these modules are tested and approved, they can be moved to the public $TVPHOME directory which is accessed by multiple users.

The names of all public and private modules are stored in the *.TiViPEfixed* and *.TiViPE* files, respectively. These files contain the full menu structure.

The *Modules* directory holds the information and source code of all modules. The *bin* directory hosts an executable for every module. When a module is modified TiViPE updates the files and directories automatically. Over time, it is expected that many modules will be made available for TiViPE, hence a clear and flexible way to structure files is necessary. In TiViPE the user can specify a *main entry name* and up to 10 levels of sub-menus. The main entry is meant

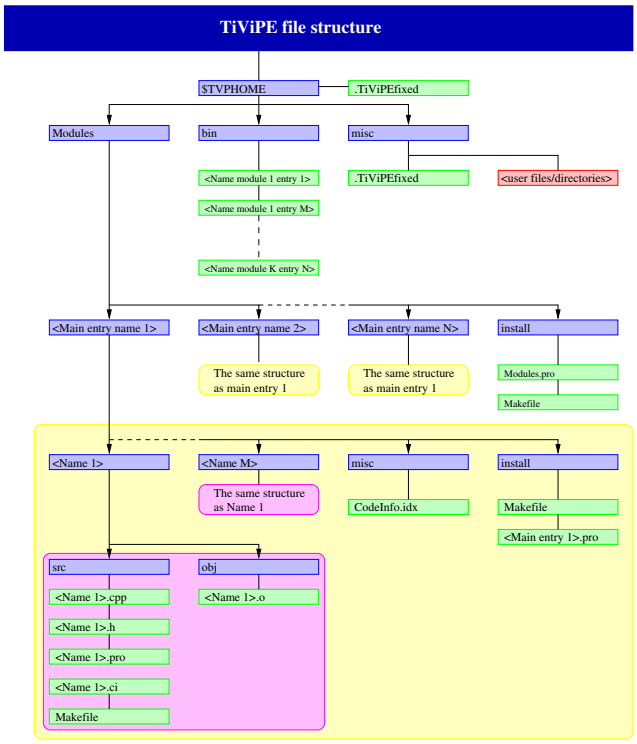


Figure 6. File structure as used by TiViPE. All files, are maintained by TiViPE.

to specify a *topic*, e.g, image processing, neural networks, finance, mechanics, etc. The sub-menus can be used to go into more detail of the given topic. At a single level, it is useful to specify first the modules in alphabetical order and next to give the sub-menus in alphabetical order. This is most convenient for searching a specific module.

Every module has a separate directory in the main entry directory. Such a directory contains sources (<name>.ci, <name>.h, and <name>.cpp) in the *src* directory and object file(s) (<name>.o) in the *obj* directory.

The file *CodeInfo.idx* in the misc directory keeps a list of all Modules and stores the makefile directives that are used by the module generator to construct, modify, or update the project file in the src directory. A *Makefile* is generated by *qmake* and is used by the program *make* to compile and install the generated source code.

3.4 Available modules

TiViPE is set up in such a way that initially the environment is empty, i.e., there is not a single module available in the environment. The environment provides the means to construct a module, that is used as stand-alone executable, with minimal effort.

Currently around 60 different modules are available, most of them operate on the Data5D structure (a vector

specified by 5 dimensions and one out of 10 supported data types), they are subdivided into five different groups. The first four groups cover the general aspects of arithmetic, data manipulation, input/output, and visualization.

The *arithmetic* group contains, bitwise, comparison, single and double operand, trigonometry and standard mathematical function operations. On complex data the following modules are available: splitting of real and imaginary part, polar-complex coordinate conversion, complex conjugate, and any normal datatype to complex conversion.

The *data manipulation* group contains routines for conversion from one datatype to another, and to normalize, reduce, append, cut, and pad data.

The *input/output* group contains routines, to read and write the Data5D data structure and images in 8 different formats.

Visualization contains modules to display or animate data.

Around 30 different modules for image processing are available. These modules were used for research purposes, they include color manipulation, data manipulation, filtering, graph extraction, image operations, image input and output, and early vision operators.

4 Discussion

TiViPE is a visual programming environment, it provides both an application environment and a software development environment for data processing, analysis, exploration, and visualization. The TiViPE environment has been designed to enable efficient collaboration among people with different backgrounds and interests. Within the environment this is achieved by emphasizing on ease-of-use, scalability, and extensibility. For example, all applications used within TiViPE, including those generated by you and others with the TiViPE environment, can transparently access arbitrarily data structures, distributed across a network.

TiViPE aims at different user groups:

1. All users will be able to program by using ready modules and constructing a network of these modules.
2. Programmers will in common avoid to use a graphical environment if the effort is too big, to wrap an existing routine. The only effort needed in TiViPE to wrap an existing routine is to fill out a set of 10 forms. TiViPE will then generate the code needed for the icon and compile the code to a stand alone program. Programmers will benefit from the existing modules in TiViPE that will give them the opportunity to re-use existing code in a simple way.
3. Researchers will find TiViPE very beneficial for performing experiments, since it allows simple and rapid parameter modification.

TiViPE supports parallel programming and allows interaction with a running program. Its graphical structure gives a developer a natural debugging tool, therefore TiViPE is suitable for critical processes that can be applied in fields like robotics and computational neuroscience.

The graphical programming environment TiViPE will make programming faster and easier! TiViPE makes other library routines accessible through its graphical environment. Construction of a network is a matter of selecting icons from a menu and connecting them. Compounding such a network to a new module is performed by pressing a single button, TiViPE will re-use the code of all the modules in the network and generate a new module. The ease of creating a new module and the re-use of existing code will lead to the availability of a respectable number of modules, that in turn can be used by others. This is the way to have many developers using TiViPE as their environment and sharing the constructed modules.

References

[1] Agilent Technologies Inc. *Vee Pro User's Guide*, 2000.

[2] P. T. Cox, F. R. Giles, and T. Pietrzykowski. Prograph: A step toward liberating the programmer from textual conditioning. In *IEEE Workshop in Visual Languages*, pages 150–156, 1989.

[3] R. Helsel. *Visual Programming with HP VEE*. Prentice Hall PTR, 1997.

[4] T. Lourens. http://www.dei.brain.riken.go.jp/emilia/Collaboration/Tino/TiViPE/index.html. TiViPE Download, 2004.

[5] T. Lourens and R. P. Würtz. Extraction and matching of symbolic contour graphs. *International Journal of Pattern Recognition and Artificial Intelligence (IJPRAI)*, 17(7):1279–1302, November 2003.

[6] B. Lucas, G. D. Abram, N. S. Collins, D. A. Epstein, D. L. Greesh, and K. P. McAuliffe. An architecture for a scientific visualization system. In *Proceedings of Visualization '92*, pages 107–114. IEEE Computer Society Press, October 1992.

[7] National Instruments. *LabView User Manual*, 2003.

[8] M. F. Sanner, D. Stoffler, and A. J. Olson. Viper a visual programming environment for python. In *10th International Python Conference*, February 2002.

[9] C. Upson, J. T. Faulhaber, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam. The application visualization system: A computational environment for scientific visualization. *IEEE Computer Graphics and Applications*, 9(4):32–40, July 1989.

[10] P. F. Whelan and D. Molloy. *Machine Vision Algorithms in Java: Techniques and Implementation*. Springer-Verlag, London, 2000.

[11] K. N. Whitley and A. E. Blackwell. Visual programming in the wild: A survey of labview programmers. *Journal of Visual Languages and Computing*, 12(4):435–472, 2001.

[12] M. Young, D. Argiro, and S. Kubica. Cantata: Visual programming environment for the khoros system. *Computer Graphics*, 29(2):22–24, May 1995.