Tools for Building Image Processing and Graphics Applications in a Workstation Environment

Gene L. Fisher
Frank J. Leahy
Laurie L. Lasslo

Division of Computer Science
University of California
Davis, CA 95616

Abstract

This paper describes a workstation-based environment for the development of image processing and graphics applications. The focus is on two graphical tools provided by the environment. The first tool is a graphical user interface based on the concept of a software control panel. Through a control panel, the user may interactively modify application program parameters and exercise fine-grain control over program execution. The second tool is a graphical block diagram constructor, which allows the user to build executable block diagrams based on standard application programs. Both of the graphical tools are integrated into the framework of an interactive programming environment, which provides overall support for applications development. While the tools are currently in use with image processing and graphics applications, they should be useful for a variety of other workstation-based applications.

1. Introduction

This paper describes the role of workstations in an applications support environment for an image processing and graphics laboratory. Software development in this laboratory entails the design and implementation of new imaging and graphics algorithms, as well as the development of applications programs to support instruction and research. The algorithms and applications programs are targeted for a variety of different imaging devices, which are in turn hosted by a variety of different processors and operating systems. As new applications are developed, they are incorporated into a library to provide an evolving base of reusable software.

A basic requirement in our setting is for tools that are standard in any good program development environment, together with an interactive interface to these tools. Such tools include language-sensitive editors, debuggers, compilers, etc. In addition, our setting has a requirement that the development tools uniformly support multiple languages. Many of the support applications and utilities are supplied by outside sources (such as vendors of imaging devices), making it infeasible to require a single language to be used for all applications.

Image processing applications are high-computation intensive. Users would prefer to spend as little time as possible waiting idly while an image is generated on a screen. Hence, a desirable feature for an integrated
environment is to allow the user to conveniently invoke multiple, non-cooperating image programs from the same workstation. Further, the code-level details of how a program is invoked on one machine or another should be as transparent as possible to the end user.

A final requirement relates to the manner in which image processing researchers typically design new algorithms. In this domain, a basic algorithm design is often sketched out in the form of a block diagram. The components of the diagram consist of a variety image processing primitives, such as functions to perform image filtering, scaling, linear combination of images, plotting, etc. A useful design tool is one which supports the construction of block diagrams based on standard components, including the ability to make the diagrams executable.

2. System Overview

Figure 1 shows the major components of the environment. The software development tools include those that are standard in any well-integrated programming environment: language-sensitive editor, compilers, interpretive debugger. The development tools also include a library installation utility for adding the hooks necessary to run an application remotely, under the control of the workstation user interface. The user interface tools include: a) a multi-language command interpreter, b) a graphical control panel interface, and c) an executable block diagram tool.

The multi-language interpreter serves two functions. It is a debugging interpreter for each of the languages in which applications are written. It is also the command language interpreter through which applications are invoked by users. Using a set of standard languages at the command shell level promotes overall uniformity of interface [13,17] and supports the concept of the "environment as a single tool" [8]. A user may interact at the command level with the same programming language that s/he uses to develop compiled applications, without having to learn yet another command language. The currently supported set of languages includes Pascal, C, Modula and Fortran. These are the languages in which applications are typically written in our laboratory.

A control panel tool [7] allows users to exercise fine-grain control over an executing application through a graphical window. The window includes a browsable list of program parameters as well as a number of function selectors to control the application's execution. A variety of graphical views can be provided for the parameters. These views include gauges and knobs for numeric parameters, spread-sheet-style windows for arrays, and other forms. Graphical views can be used for both modification and display of parameter values, and so provide a convenient mechanism for performing the repetitive testing typical with image processing applications. In this cycle, users continuously invoke a program, inspect the resultant image, change a few program parameters, and repeat the cycle. The control panel interface makes this reexecution process substantially more convenient than is possible with a "glass-tty" interface.

The block diagram tool allows users to construct executable block diagrams composed of standard applications. Execution is defined in terms of a data-driven interpretation of the diagram, where each node executes as a normal sequentially-coded application chosen directly from the library. One of the functions of the environment's installation utility is to include source code hooks so that compiled applications can be executed from within a block diagram.

![Figure 1: Environment Components.](image-url)
The interactive tools communicate with an applications server that runs the compiled applications on potentially remote hosts. From the user's perspective, the remote execution of an application is largely transparent. The server handles communications with the interactive tools via a remote procedure call mechanism, which transmits parameter packets to and from the executing applications.

The overall user interface to the environment is through a multi-window workstation display. Figure 2 shows a sample display screen with a variety of windows. Clockwise from the lower left corner, the windows are:

- an interactive command interpreter, currently running a Pascal front end (physically, the interpreter may run inside an editor window, so that convenient editor functions are available such as scrolling, textual pick, etc.)
- a Pascal-based editor, containing the Pascal source code for an imaging application named "LINCOM"
- a block diagram tool, containing a simple executable diagram to perform scaling (SCALE), linear combination (LINCOM), and display (PLOT, DUMP)
- a control panel for the LINCOM program
- control gauges for the LINCOM coefficient parameters

The remainder of the paper will focus on the environment components that use the special capabilities of the workstation -- the control panel interface and the block diagram tool. Other features of the environment are described elsewhere [8, 9].

3. Software Control Panels

For electrical engineers, the concept of using a control panel is a standard part of testing and operating electronic devices. The panel provides a general-purpose interface to the changeable parameters of the device. When a control panel is part of a debugging scope, the panel provides readouts for specific probes that can be inserted at points within the device.

The concept of a control panel in our environment is an effort to adapt the well-used idea from electrical engineering to software applications [7, 11]. That is, the panel will provide an interface to an application through which parameters can be set and displayed, both for end-use and debugging purposes. A high-resolution bit-mapped screen permits the modeling of control panel gauges and knobs. The gauges are used to report changing values as a program executes; the knobs can be turned using a mouse or other pointing device.

The structure of an application control panel is shown in Figure 3. Physically, the control panel is a window on the workstation screen, containing the following component subwindows:

- The bottommost subwindow is a scrollable list of program parameters and their current values.
- To the left of the parameters is a view-indicator subwindow that specifies which forms of alternate view are currently displayed for which parameters. The indicators are iconic depictions, such as a small gauge for an integer or real, a magnifying glass for an array or record zoom, etc.
• Immediately above the parameters is a control function subwindow where the functions are selected by mouse click. From left-two-right, the functions are:
  o Run -- run the program, returning control to the panel
  o Run&Done -- run the program once only, returning control to the command interpreter
  o Abort -- return control to the command interpreter without running the program
  o PanelHelp -- display instructions for how to use the control panel
  o ProgHelp -- display additional help for the entire program (supplied as optional header comment above a program definition)
  o ParmHelp -- display help information for a parameter (supplied as optional comment in the program definition next to each parameter)
  o AltView -- select a graphical alternate view for a parameter
  o Inc/Dec -- select an increment/decrement accelerator for a scalar parameter (e.g., integer, real); inc/dec is the digital equivalent of turning a knob, which some users may prefer
  o Zoom -- zoom in on a parameter of a composite data type (e.g., array or record)
  o Default -- reinstate the default parameter value
  o History -- reinstate a previous parameter value (successive clicks move back in history ring)
  o Undo -- undo the last change/function
  • the top two subwindows in the control panel contain information about the status of the program, detailed help information about the currently selected parameter, and error messages.

In addition to the functions on the control panel itself, auxiliary windows provide more detailed control and display of program parameters. These auxiliary windows are selected using the 'AltView' and 'Zoom' functions for specific parameters. The auxiliary controls include various forms of knobs and gauges for scalar parameters, graph plots of vectors, and spread-sheet-style access to matrices and higher-dimensional arrays. Further alternate view details are shown in Figure 4.

Control panels and the associated gauges are generated based on the source text of an application program. Programs are written without special hooks for panel generation. The user may supply help information in the form of standard header and parameter comments in the program text. If specified, this information will appear in the panel information window as appropriate. The type of alternate view generated for a particular parameter is based on the parameter's data type and default setting.
An illustration of control panel use is shown in Figure 4. The figure contains a snapshot of a session where a user is controlling the execution of a graphics program. Counterclockwise from the upper right, the windows on the screen are:

- A program edit window, containing Pascal source text for the definition of a graphics program to plot a Bezier triangle
- A control panel for the Bezier program
- A defaults editor for Theta camera angle gauge
- Graphical views for each of the seven Bezier parameters (below the control panel along the bottom of the screen)
- The graphical output of the Bezier program
- The interactive command interpreter, running Pascal

Running in the command executive window is the interactive interpreter for one of four supported languages. The user may select the language at any point that a new command is entered. The currently selected language is indicated by the prompt ('P>' for Pascal, 'C>' for C, etc). The selection also affects the language used by the editor for prompting templates and other language-specific editor functions. The remainder of this example will use Pascal.

From within the command window, users may enter any normal Pascal declaration or statement. Thus, to invoke the Bezier program, the user may use the standard Pascal calling form of name(actual parameter list). When invoked in this manner, the program executes normally, returning control to the command interface upon completion. Alternatively, the user may invoke a procedure with the actual parameters underspecified. Underspecified means that one or more actual parameter values are left blank or specified as a question mark. For example, in Figure 4, the call of Bezier is "Bezier(3,2,30,30,?,?,CP1)." Invoking a program with an underspecified actual parameter list will activate a control panel for the program.

When the control panel becomes active, the program is suspended at the point of parameter binding. The user enters, modifies and browses actual parameter values in the bottom subwindow of the panel. For non-scalar parameters, the user may zoom out using additional subpanels to show the parameter's components. For example, the scrollable window near the lower right of Figure 4 is a zoom on the ControlPoints matrix.

Once parameter values have been satisfactorily entered, the user may select the "Run" function button in the control panel, whereupon the program will continue its execution, returning control to the panel when it has completed. In this way, the user may continue to enter new parameter values and reexecute in order to inspect the behavior of the program. As the program...
environments and its associated gauges. This form of animated display is comparable to the visualization capabilities of other environments [16,14,3].

Figure 4 shows a total of eight alternate data views for the Bezier parameters. These were selected using the 'AltView' function in the control panel. Based on the data type of the selected parameter, AltView provides a graphical depiction of the data. In addition, the user may modify certain features of the graphical view, using a type-specific defaults editor. Overlaid on the upper right of the control panel in Figure 4 is a defaults editor for the Theta control gauge.

One of the key features of this environment is the capability to use alternate data views as input devices as well as for display. When parameter values are changed in the control panel, no values are transmitted to the interpreter until the 'Run' button is selected. To permit more immediate execution, any of the alternate views can be set to operate in accelerator mode. In this mode, the user may position a pointing device on the hand of one of the control knobs, turn the hand, and release the button, whereupon the program will be immediately reexecuted, without having to select 'Run'. This form of control is particularly useful for controlling the parameters of programs that generate image or graphic displays. As an input parameter is changed in the control panel, the resulting effect is shown immediately on the output image. When the control panel knob is used, this effectively animates the output display. For example, turning the knob associated with either of the Bezier camera angles will rotate the displayed triangle (not necessarily in realtime).

To summarize the use of the control panel, consider the sequence of user interactions that leads to the final configuration in Figure 4:

1. Initially, only the command and edit windows are present on the screen. To invoke the control panel, the user loads and invokes the Bezier program in the command window, using the calling form "Bezier(3,2,30,30,?,?,CPI)." The question marks correspond to the ShowGrid and ShowTriangle parameters.

2. At this point, the control panel appears, without any gauges or other alternate views.

3. Now inside the panel, the user sets both 'ShowTriangle' and 'ShowGrid' to true and selects the 'Run' function.

4. The procedure then begins execution, the graphics output window appears, and the Bezier triangle is drawn with parameters = (3,2,30,30,30,?,?,CPI).

5. To experiment further, the user then selects an alternate view for each of the procedure's parameters, the result being the row of gauges, etc. along the bottom of Figure 4.

6. Particularly revealing interactions include rotating the camera angles 'Theta' and 'Phi', plus modifying individual control point values in the array 'ControlPoints.' The final display shown in Figure 4 is the result of having rotated Theta one full turn from 30 degrees and then up again to 60 (=420), increasing the level of subdivision to 4, and changing ControlPoint[5,3] to stretch it from the center of the triangle over to the left edge of the graphics output display.

At present, the user must invoke the application from within the context of the command interpreter in order to use the control panel. We are currently working on a menu-based interface that will allow applications to be selected and executed directly from a menu of available library programs. Through this interface, the control panel will appear on the screen immediately upon menu selection, without requiring the end-user to enter the command interpreter at all. This will be a substantial enhancement for users who would prefer to remain unaware of the application's program-level details. In particular, the user need not be aware of the language in which the program was written, nor any of the details of command interpreter operation.

The menu-based interface will also provide a means for end users to invoke multiple, independent programs. Each menu-selected program will have a separate control panel process to allow the user to view and control multiple executions on an individual basis. At present, a control panel is invoked synchronously from within the command interpreter, which means that the interpreter waits until control is returned from the panel in order to proceed with another command. In this synchronous form of operation, multiple control panels may be active if applications contain nested calls to other applications that in turn use a control panel. However, only one control panel can be active at a time, since the nested procedures are executing synchronously, not as independent processes.

The menu-based interface overcomes the limitation of synchronous operation by bypassing interpreter control entirely. Another means to remove the limitation of synchronous operation is to add a general concurrency feature to the interpreter. This solution would involve a significant modification to the semantics of the supported languages, since neither Pascal, C, nor Fortran supports concurrency at the language level. We are currently investigating how general concurrency can be added to the interpreter in a graceful manner.

4. **Block Diagram Tool**

The design tool in our environment embodies the general principal of the dataflow diagram. There are a wide variety of forms for expressing such diagrams, including DeMarco's structured analysis technique [4], SADT diagrams [15], and others. While specific details in each of these techniques vary, the general principle is
the same -- the high-level functional processing of a software system is specified as a network of processes interconnected with named, typed data links.

Recent software engineering environments have begun to provide graphical interfaces for constructing dataflow diagrams [18] as well as graphical dataflow interpreters [5, 10]. These environments allow users to express the diagrams in graphical forms that are significantly more natural and convenient than earlier textual notations.

In addition to the use of dataflow techniques for specification and design, much work has been done on dataflow languages and their translation [12]. In general, dataflow languages express small-grain details of program logic, but are not necessarily suited for expressing high-level, abstract design. Work on the translation of dataflow languages does provide insight into how high-level dataflow designs can be translated into executable form, however. Two systems for large-grain dataflow translation are described in [1] and [2]. The translation in these systems is "large-grain" in the sense that standard sequential programs are connected together as the nodes in a larger dataflow network. In both these systems, the dataflow translator functions as a preprocessor for textually-specified dataflow programs, producing compiled executable code.

The diagram tool in our environment combines features that have appeared separately in other systems:

- a graphical user interface
- large-grain, interpretive execution
- support for dataflow nodes written in several standard application languages

Figure 5 is an example of the dataflow design tool. The tool interface is through a window with the same generic structure as the control panel. Clockwise from upper left, the subwindows are as follows:

- A function button subwindow, with the following functions:
  - **Program** -- load a program node and display its movable icon on the canvas; the name is entered in the upper right text entry subwindow
  - **File** -- load an input/output image file and display its movable icon; the image file is the most common form of constant input data to the head nodes of the dataflow program
  - **Tty** -- allocate a tty subwindow and display its movable icon; a tty subwindow is used to display raw data in textual format, typically for the purpose of debugging
  - **ImageDev** -- allocate (i.e., attach) an image device and display its icon; image devices are typically remote from the workstation that runs the user interface
  - **Pipe** -- connect two nodes with a data pipe; endpoints are selected by entering names in the text entry subwindow or by pointing on the canvas; details of data interconnection can be specified by zooming out on a program icon, as described below
  - **Tee** -- display a tee node, which is used to split (i.e., duplicate) an outgoing data stream or merge (i.e., multiplex) an incoming data stream
  - **Delete** -- remove a displayed icon
  - **LibSelect** -- select the default library from

![Dataflow Design Tool Diagram](image-url)
which selected program nodes will be loaded; more than one library may be active at one time, in which case an ordered library search will be performed.

- Go -- commence data-driven execution of the dataflow diagram; details of the execution mechanism are given below.
- **Suspend** -- suspend execution of the entire program (a stop signal is sent to each of the executing nodes); individual nodes can be suspended selective pointing; the suspend button is a toggle, which switches to "resume" when clicked on "suspend".
- **Stop** -- terminate the entire program (a kill signal is sent to each of the executing nodes).
- **Save** -- save the textual representation of the program graph for later reloading.

- A text entry subwindow. Each of the functions that requires one or more textual arguments will prompt in this subwindow. Error messages will be issued here.
- The dataflow canvas subwindow. Program and other icons appear here as they are selected.

In addition to the function buttons, a pop-up menu is available in the diagram canvas to perform the following functions:

- Open a canvas icon to its enlarged form; table 1 indicates the corresponding open and closed forms for each type of the canvas item.
- Move an icon on the canvas.
- Delete an icon from the canvas.
- Delete a connection between two icons.

The dataflow pipes shown on the diagram canvas can be thought of as line "trunks" in the sense that they indicate one or more specific data connections between two nodes. The diagrams are depicted using trunks rather than individual lines because applications may have large numbers of incoming and outgoing parameters. If the individual connections were shown on the canvas, a large number of such connections could become unwieldy to display clearly. Given that trunks show only large-scale interconnection, the user is given the capability to zoom on individual program icons in order to view the interconnection details. The zoom view of a program icon is in the form of a control panel, as shown in Figure 6. The figure shows the details of how the SCALE and LINCOM nodes are interconnected. Specifically, the sixth parameter of the upper SCALE node is connected to the fourth parameter of LINCOM; the sixth parameter of the lower SCALE node is connected to the fifth parameter of LINCOM. The tool automatically tags duplicate nodes so that external references can be disambiguated; the convention is to label the first node to appear on the canvas with an 'a' suffix, the second 'b', and so on.

The dataflow diagram depicted in the tool canvas is fully executable. The nodes represent standard compiled applications, loaded from the library. The execution proceeds by treating each node as a data-driven subprogram. Specifically, a node executes as a separate (UNIX) process. Prior to beginning execution, the process performs a blocking read on each of its input data lines. When all reads have completed, the sequential body of the node proceeds to execute. Upon completion of the normal body, the process writes to each output data line.

In general, any of an application's formal parameters may be connected to a dataflow port. In image-processing applications, the connections are typically between parameters of type ImageFile. In order for an application to execute, all of its parameters must have specified values. This is where the control panel again comes into use. Using the control panel parameter entry window, values for all non-connected parameters are specified as constants that will be sent directly to the application when the dataflow 'Run' function is selected.

Hence, each application will explicitly read all incoming data lines from other applications, and receive values for the remaining unconnected parameters directly from the dataflow diagram tool.

This model of dataflow execution can be considered "large-grain" in that nodes are internally a standard sequential application, externally connected in a dataflow network. To the dataflow interpreter, the sequential body of each node is a black box. At present, the dataflow diagram is interpreted in a strictly data-driven manner. That is, each node must wait until all the input data have arrived on all input ports before execution will commence. Having received all data, the node executes normally, as if it had been called from any other context, such as the sequential command interpreter for example. Upon completion, the node writes on each of its output ports.

The diagram tool performs type checking on each of the connections specified in the canvas. It also performs structural consistency checks to disallow inconsistent connections; for example, two outputs cannot be connected to one input port. The checks also disallow circular connections, although we are at work on an implementation that will remove this restriction. The tool does not perform any internal checks to verify that the connections are semantically meaningful; any of an

<table>
<thead>
<tr>
<th>ICON</th>
<th>OPEN FORM</th>
</tr>
</thead>
<tbody>
<tr>
<td>named program node</td>
<td>control panel for program</td>
</tr>
<tr>
<td>image data file</td>
<td>read-only textual display window open on file</td>
</tr>
<tr>
<td>image device</td>
<td>workstation preview of image</td>
</tr>
<tr>
<td>tty subwindow</td>
<td>read-write textual display window</td>
</tr>
<tr>
<td>data line</td>
<td>information window displaying</td>
</tr>
</tbody>
</table>

Table 1: Corresponding Open forms for Canvas Icons.
application’s parameters can be treated as an input or output port. This could allow meaningless connections, such as using a read-only input parameter as a dataflow output. To avoid having a node connected in a meaningless way, the application developer can indicate in the parameter help information which parameters should be used for inputs or outputs in a dataflow context. This help information will appear in the control panel to assist the user in constructing meaningful dataflow programs.

A particularly noteworthy feature of the dataflow interpreter is that individual nodes consist of standard application programs from the library. The programs must be processed by the installation utility prior to compilation. The utility adds the necessary process declaration and read/write logic to permit execution in a dataflow context. The additional code is provided wholly by the utility and remains transparent to the application programmer and end user.

At present, the control panels are used only to specify wiring details and to supply constant parameter values. We have designed and are implementing a natural extension to the use of control panels that will permit panels to be used to control dataflow execution in the same manner they are used to control execution of an application invoked from the interpreter. Consider again Figure 6. A natural point of control in the execution is the central LINCOM node. If a control panel, such as the one for LINCOM, remains open after execution commences, its presence will indicate a break point at that node in the diagram. Through the open panel, the user will select control parameter values, through knobs if desired, and then use the panel Run button to continue execution. Closing the control panel during execution will signal a removal of the breakpoint, so that dataflow execution can proceed uninterrupted.

Also in the design stage is the full integration of the Pascal-class interpreter into the dataflow context. At present each of the nodes in the diagram is a compiled program selected from the library. For debugging purposes, it would be convenient to allow nodes to execute interpretively within the context of the dataflow program. Then, if a particular node was not fully debugged, the debugging interpreter could be run on the node at a controlled breakpoint within the dataflow execution. This would provide both small- and large-grain debugging capabilities for a dataflow program and its nodes.

5. Conclusion

This paper has presented an environment for use by developers of scientific and engineering applications programs, with particular emphasis on graphics and image processing applications. The design has been driven by the specific needs of researchers in a graphics and image processing laboratory. The primary requirements in this setting are for a multi-language environment, with a convenient end-user interface to the applications, and a higher-level design tool. The three major environment subsystems address these requirements.

The general requirements in our setting are common to other scientific and engineering development environments. For the initial prototype development, we are focusing on the specific needs of our own user community. As the environment becomes stable, we plan to export it to other suitable sites.

A noteworthy contribution of the environment is
the support for software reuse. We have made a concerted effort to have newly developed tools usable with an existing library of applications. The multi-language interpreter supports standard languages so that existing software need not be converted to a new language. The control panel tool operates on standard programs; programmers are not required to insert special hooks for the control panel interface. The dataflow tool allows standard sequential programs to be interconnected, through standard parameter specifications, to operate in a large-grain dataflow context. Finally, the program modifications required to install an application are performed at the source code level by a preprocessing installation utility. This makes details of program installation transparent to the user and permits the use of existing compilers for the generation of efficient application code.

In general, the support for reuse will help users to migrate more easily from a tty-based environment to one that is workstation based.

References


