VPL: An Active, Declarative Visual Programming System

**David Lau-Kee, Adam Billyard, Robin Faichney, Yasuo Kozato,
Paul Otto, Mark Smith, Ian Wilkinson**

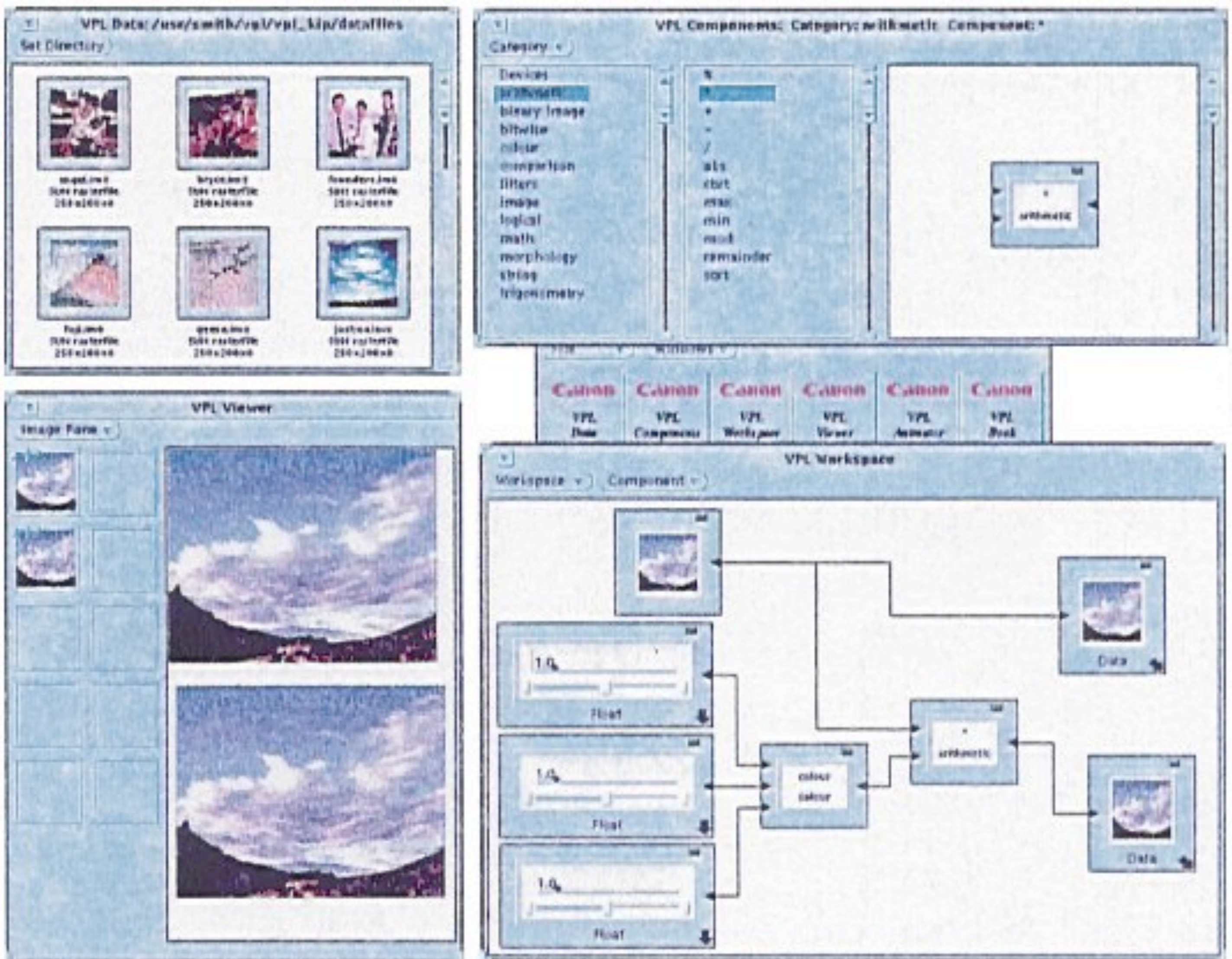
**Canon Research Europe,
Surrey Research Park,
Guildford, GU2 5YD, UK.**

Abstract

VPL is a visual programming language and environment for interactive image processing.

VPL uses a novel system architecture which separates interaction and computation components in order to provide a highly interactive visual programming user interface.

VPL is based on a declarative, demand driven dataflow model of computation. It is a practical, usable visual programming system, integrating tools for browsing, inspecting and editing components and documentation. VPL also provides data persistence and data import / export mechanisms. The visual programming model supports both function extensibility and higher order functions, allowing users to build their own program dataflow structures within the VPL environment.



1 Introduction

Many visual programming systems focus solely on the question of program representation, often to the detriment of supper for the actual task of visual programming. Some visual programming systems fail to integrate the execution of the program fully into the system (i.e., their concerns end with the production of an *artifact*; a visual program specification) or their level of functional sophistication is very low (either just sufficient to show that computation is possible but not comprehensive enough to be practical to a real end-user, or lacking in abstraction or extensibility mechanisms). VPL exemplifies an approach to visual programming which addresses these three central issues: support for the *task* of visual programming; support for visual programming as an *interactive process* rather than simply as the interpretation of a visual language artifact; provision of powerful mechanisms for *extending the functionality* (including a comprehensive library of functions).

2 Previous Work

The origin of visual programming is subject to debate, but the earliest example of what we would consider to be visual programming is William Sutherland's Graphical Programming Editor [5]. The dataflow programming approach outlined in Sutherland's work has been adopted by many visual programming systems, including VPL.

More recently, systems such as HI-VISUAL [6], apE [7], Khoros [8], VIVA[9] and LabVIEW [10] have used forms of dataflow as the computational model underlying their visual programming features. To place a perspective on our work, we will consider VPL in relation to aspects of two of these systems: the surface characteristics and functionality of Khoros, and the application domain and motivating design philosophy of VIVA.

In terms of surface similarity the Khoros system is perhaps closest to VPL. The Cantata module of Khoros [8] provides similar functionality to VPL's front end. However, VPL and Khoros differ significantly in many corresponding areas. For example: Cantata's program model is based upon non-declarative token dataflow, whereas VPL provides declarative dataflow; Cantata is data driven, VPL is demand driven; Cantata is level 2 live, whereas VPL is level 4, or fully, live (see ß4.1); VPL's functions are *first-class*, e.g., they can be passed as arguments to other functions, Cantata simply pipes data into functions; Cantata allows the user to save by taking a 'snapshot' of the entire state of the system, whilst VPL provides an extensible system complete with data persistence; Cantata allows a form of complexity hiding by using 'subprocedure workspaces', whereas VPL provides full and seamless procedural abstraction). Perhaps more importantly, the design philosophy behind VPL has produced a system which has a unique and specific *feel*: users manipulate and combine tangible, 'physical' components ; there is no 'underlying interpretation', VPL simply merges the edit-compile-execute cycle into an immediately active 'plug and go' mechanism.

Perhaps the previous work closest to our own in terms of application domain and motivating philosophy is Steven Tanimoto's VIVA [4]. However, substantial differences in the architecture, the design approach, the level and extent of programmer support differentiate VPL from VIVA. A final distinction between VIVA and VPL is that VPL is in the final stages of completion, whereas, to the best of our knowledge, Viva exists only on paper as a set of proposals.

3 Overview of VPL

VPL is a programming environment that provides a set of tools for supporting the user in developing interactive, visual programs. The style of programming VPL encourages is exploratory and opportunistic. The application domain is the computationally expensive area of image processing. These factors originally led to the design of a novel system architecture, based around a client-server model, to support highly interactive, incremental visual program construction and execution. The actual image processing is performed by an integrated C++ 'back end' library, but VPL provides its own dispatcher to allow incremental program interpretation. By separating the three task groupings of: representation and interaction; parsing and function dispatching; and function evaluation, VPL succeeds in providing a very malleable and flexible visual language user interface which encourages incremental, opportunistic programming, whilst backing this up with a comprehensive and powerful library of functions.

The novel and interesting features of the system include:

- No program versus process distinction. In other words, the representation and behaviour of both interaction and execution are seamlessly linked, providing the end-user with a very simple system model.
- No hidden state. VPL is a WYSIWYG system.
- A declarative, demand-driven dataflow computation model. This straightforwardly minimises the computation to be carried out, since only when an output demand is made do the nodes leading to the output get evaluated. (This is a dataflow equivalent to lazy evaluation and is a very significant feature of the system since VPL's application domain is image processing where minimising computation has considerable time and space benefits).
- Support for higher-order functions. This means that users may create arbitrarily complex or refined dataflow structures.
- A client-server system architecture. VPL consists of a number of loosely coupled, communicating processes each responsible for a different task. This provides distribution of load, and allows different evaluation priority strategies as appropriate for the specific function of each unit. For example, at the user interface we are able to provide immediate feedback for syntactic, surface user actions irrespective of the computational requirements of the program being executed.
- Separate, communicating subsystems. For example, one process is responsible for maintaining and evaluating program graphs. The manner of evaluation (such as eager or lazy) is determined solely by this unit. Communication between units follows a message or request passing protocol, thus a different evaluation unit could be inserted whilst affecting little in the rest of the system. The front end runs on a Unix workstation with a colour bitmapped display, and the other parts of the system (see ß4) ; including the computationally intense image processing ; run as separate Unix processes, possibly distributed across a network.

4 Design History, Design Aims

VPL was originally intended as a simple, graphical user interface to the V-Sugar [1] image processing library. It soon became clear that if the full potential of a visual programming approach was to be harnessed then the user interface concerns of direct manipulation; consistent, coherent and persistent visual representations; declarative, modeless interaction; active components and immediate feedback; etc., would need to be addressed. The VPL prototype [2], written in Smalltalk-80 and C, explored many of these issues. Our experience with the prototype led to the following design aims:

4.1 Liveness

Studies on visual programming must address more than simply program representation (i.e., more than the *power* or *representational abilities* of a visual language, or the visual program as an *artifact*); in particular, issues concerning user interaction (manipulation and feedback of components of the visual language) and issues concerning the behaviour and representation of executing visual programs both require investigation.

Myers [3] splits visual languages into two categories, visual programming and program visualization, which could be thought of as visual programming in the contexts of *specification* (what the artifact stands for) and *execution* (what the artifact does). We suggest that the manner in which the user builds and interacts with visual programs should also be considered, in other words visual programming should be thought of in a context which includes *interaction*.

An important aim for VPL was that it should be highly interactive, and that it would combine the stages of program development (editing or construction), program compilation and program evaluation. In other words, VPL makes no distinction between a program and its value. VPL intertwines the stages of specification, compilation and execution, which can be described after Tanimoto [4], whose notion of 'liveness' captures our own intent. Briefly, level 1 is an "informative" level (meaningful to the user), level 2 is a "significant" level (meaningful to the machine), level 3 is "responsive" (user actions trigger visual changes), and at level 4, creation or modification causes the computation taking place to change.

VPL is level 4 live (or fully live). Programs are constructed by dragging the components into a workspace. As the components are plugged together a declarative program relating the components is evaluated

giving direct and immediate feedback. Manipulation of the program (by unplugging and re-plugging components), or manipulation of the devices (e.g., dragging the 'slider bar') causes direct and immediate re-evaluation.

4.2 Tool Integration

As well as the Workspace, which is used for constructing visual programs, VPL provides a number of separate tools to support the program development process (such as the Component Browser, and the Book). Although each component runs as a separate process they are all designed around a common representation and communication protocol. The communication channels connecting the devices are extremely rich: communication can be synchronous or asynchronous, and arbitrarily complex data or, indeed, functions can be passed along them.

A consistent and general procedure for user initiated communication between tools is provided using a 'drag and drop' interaction mechanism. The semantics of a 'drag and drop' action are a function of the triple <source, object, destination>. In other words, the meaning conveyed by dragging an object between two locations is dependent upon those locations. For example, dragging a component from the Component Browser to the Book gives the user an interactive page describing the component.

We find that the integration of contextual information in the interpretation of the user action gives a task vocabulary quite rich enough for VPL's requirements.

4.3 Separation of Concerns

The requirements of the platform supporting the VPL front end mainly concern handling input and output devices (as is the case with any highly interactive, graphical user interface). Conversely, the requirements of the image processing module concern (in our case) efficient manipulation of large numerical arrays. The architecture of VPL reflects this difference in objectives by separating the concerns of the user interface and the image processing modules. The front end benefits since it does not have to 'waste' time on processing anything not concerned with the representation and manipulation aspects of the system, and the back end benefits in not having to incorporate and service graphics and interaction handling routines.

The architecture of VPL is described further in ß4.

4.4 Extensibility

A system designer cannot predict everything that a user might want to do, so in order to be of practical use, programming systems must be extensible.

VPL provides procedural abstraction by allowing programs to be 'packaged up' as composite functions, which can then be stored in the Components Browser for later reuse. The components of a composite function can include data objects as well as functions. On quitting VPL, newly produced components and data are saved automatically, and are available for use in later sessions.

VPL also allows higher-order functions (i.e., it allows functions to be passed as arguments to, or returned as results by, other functions. This allows the set of dataflow structures available to be extended. In ß6.1.4 we provide an example to show the power of this facility for a visual programming language.

5 The Architecture of VPL

5.1 The Hub

The central component of the system is the 'Hub'. The Hub straddles the border between front end and back end, providing a communications-handling interface between other units (see Figure 1).

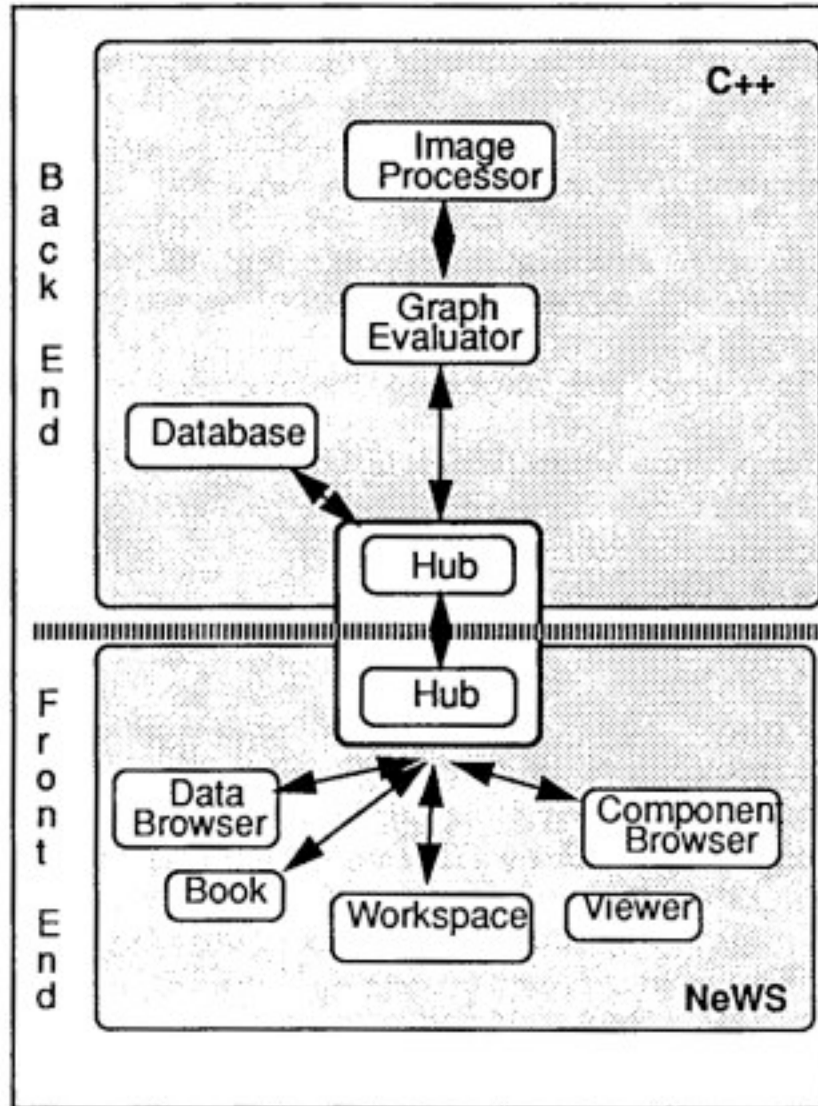


Figure 1

The VPL front end uses the OpenWindows XNeWS server. The Hub is a C++ program which downloads communications-handling code to the XNeWS server. The front end components are NeWS programs running as separate light-weight processes in the XNeWS server, and which use the NeWS event mechanism for communication between each other and the Hub communications module.

The Hub XNeWS client side handles both synchronous and asynchronous requests between the Graph Evaluator, Database and Hub XNeWS server-side processes. It is the Hub's responsibility to ensure that requests are serviced correctly, integrity is maintained, and that updates, invalidations and change notices are propagated throughout the system.

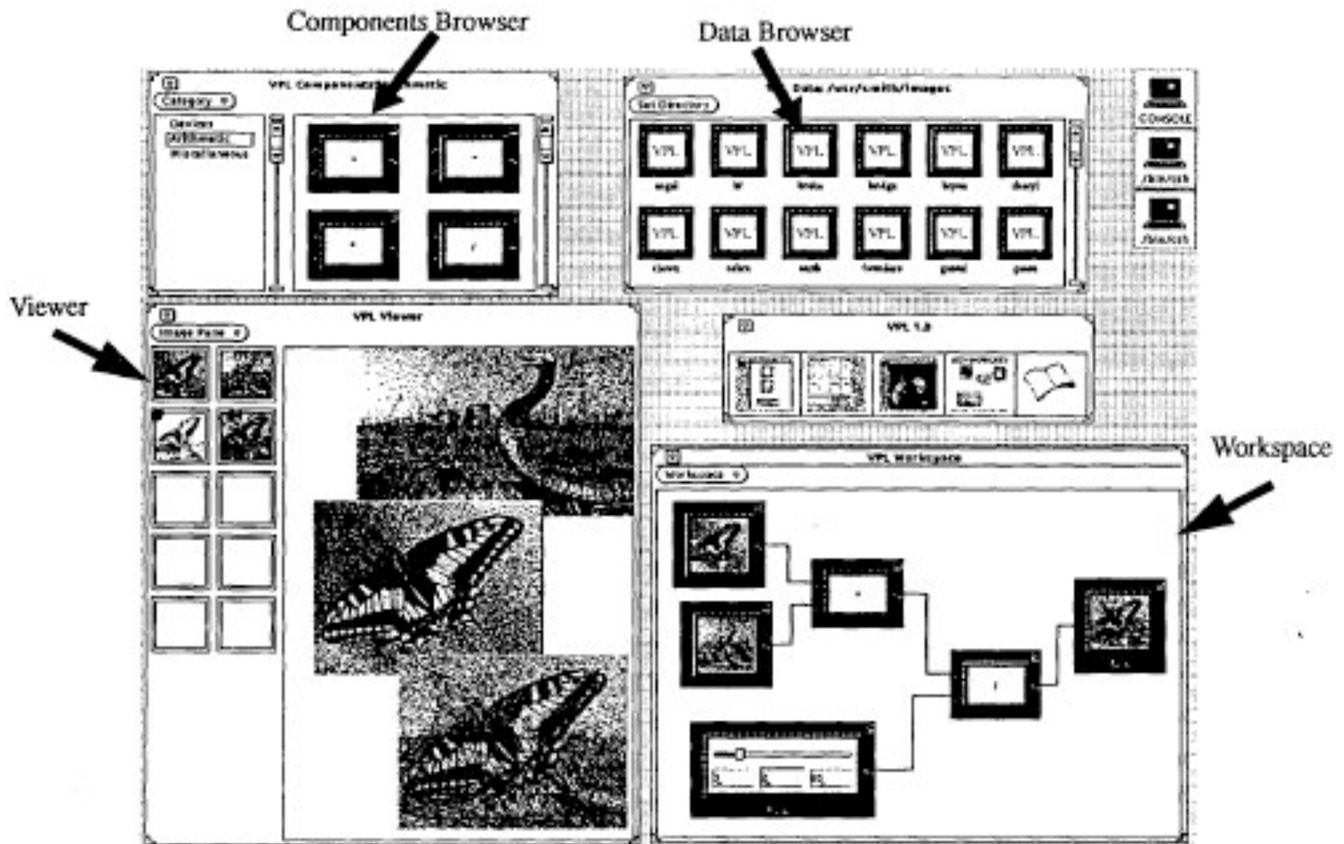


Figure 2

5.2 The Database

The Database process handles requests for information on the structure and form of components and data. The Database is responsible for coordinating access to the filesystem, and presenting a persistent store for VPL.

5.3 The Graph Evaluator

The Graph Evaluator is responsible for ensuring that the user interface (or any other external interface) is always supplied with up-to-date computational results. It is a separate module so that it is easy to ensure that only the necessary computation is done, and so that memory usage can be optimised.

The front end notifies the Graph Evaluator of any changes to the set of program graphs in the Workspace (see ß6.4). The graph evaluator checks to see whether any existing values have been invalidated by the changes; if they have, it notifies the user interface at once. This ensures prompt user feedback, even in cases where the true result will take seconds or minutes (or even hours!) to calculate. Then the graph evaluator calculates the updated values, and notifies the user interface.

In the current application (image processing) the overheads of this process are negligible in comparison with the time required by the image processing itself.

5.4 The Image Processor

The "back end" of the system is the part responsible for doing the real computation (from adding two numbers to performing a Fast Fourier Transform on an image). It has two main components: the image processing library and the dispatching code.

The image processing library is an arbitrarily large C++ library of subroutines. In the current version of the system, the V-Sugar library from VIEW-Station [11] is used. The dispatching code ensures that, given the types of the arguments, the correct version of any routine is called.

C++ provides *ad hoc* overloading of functions, and a complex set of automatic coercions. Unfortunately, C ++ compilers will only generate the appropriate code if the types of the arguments are known at compile-time, whereas in an interactive system like VPL the argument types are not known until run-time. This means that we have to provide our own code to handle the dispatching, in effect dynamically interpreting and typing the programs and dispatching the C++ functions as appropriate. We machine-analyse the raw C++ image processing library and then automatically generate this dispatching code; partly to reduce the likelihood of errors in the dispatch code, and partly so that it is easy to upgrade the subroutine library (or substitute a different library).

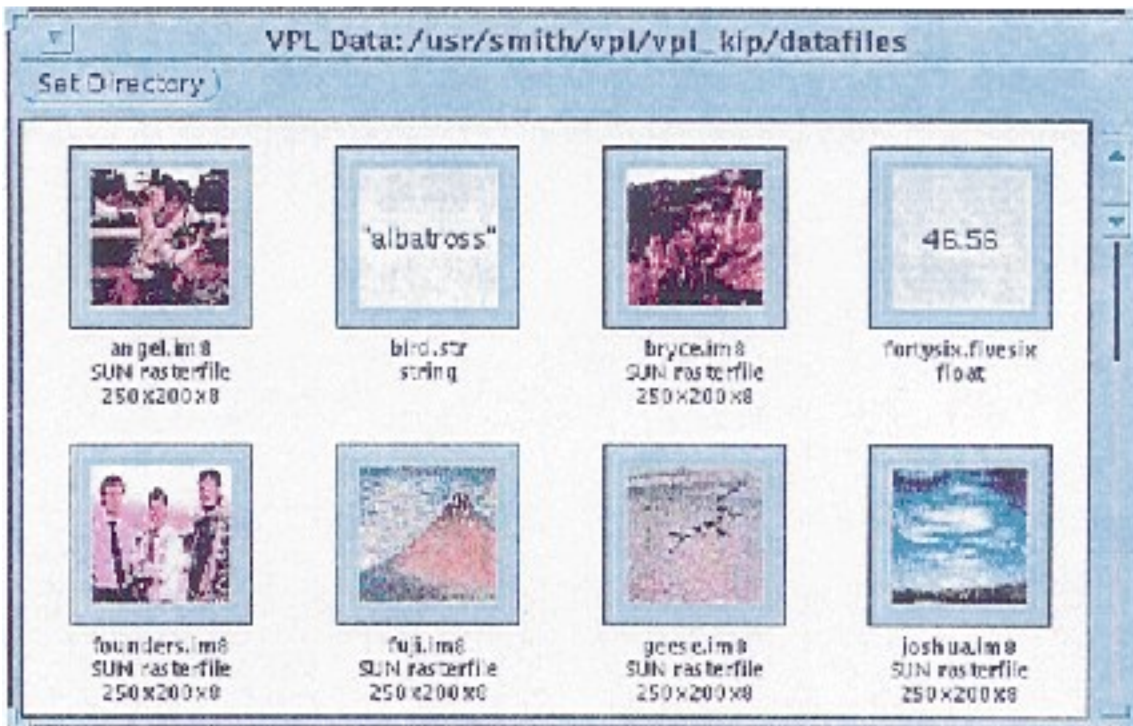
6 The Front End

6.1 The Data Browser

The Data Browser provides a view onto VPL data items. It acts as VPL's equivalent to the Apple Macintosh's Finder, for VPL specific data.

Users alter their view on the filesystem by changing the Data Browser's target directory. Data objects visible from this viewpoint are collected by the Database and passed to the Data Browser for representation and manipulation.

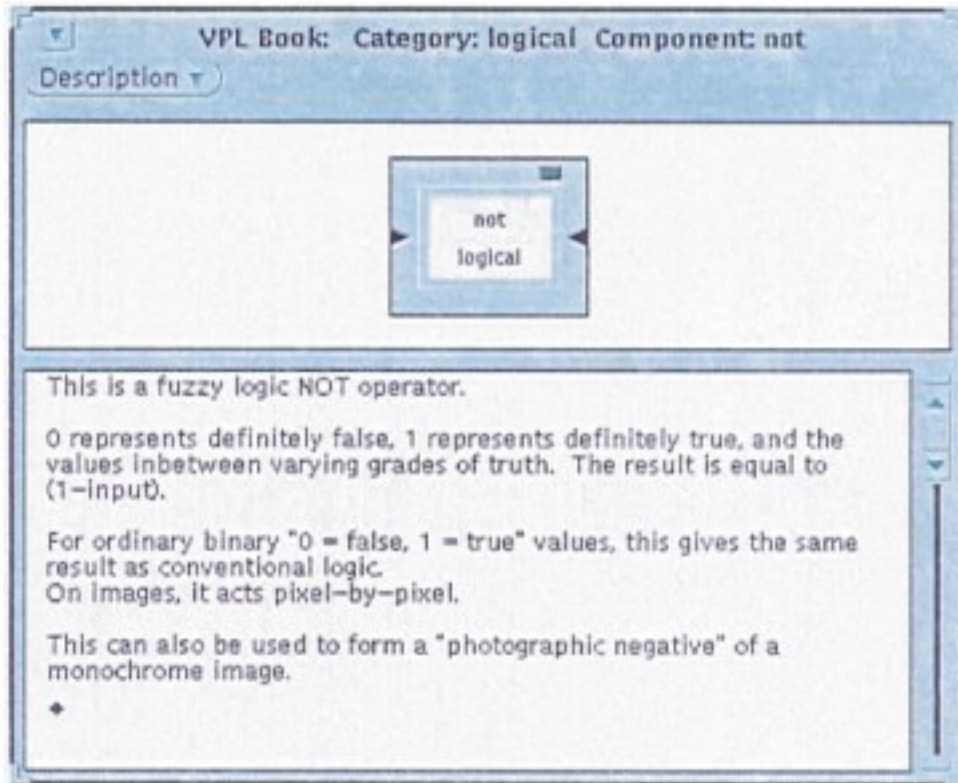
Dragging from the Data Browser to the Workspace copies the data and makes it 'live' and available for processing. Dragging to the Book presents a description of the data. (Although the data format we use is specific to VPL, we are publishing the format definition and will provide several conversion filters for some common data formats, such as PICT, Sun rasterfile, PPM, etc.)



6.2 The Book

The Book is a text viewer and editor. It provides details on all primitive and composite functions and components, as well as automatically generated details documenting VPL data objects.

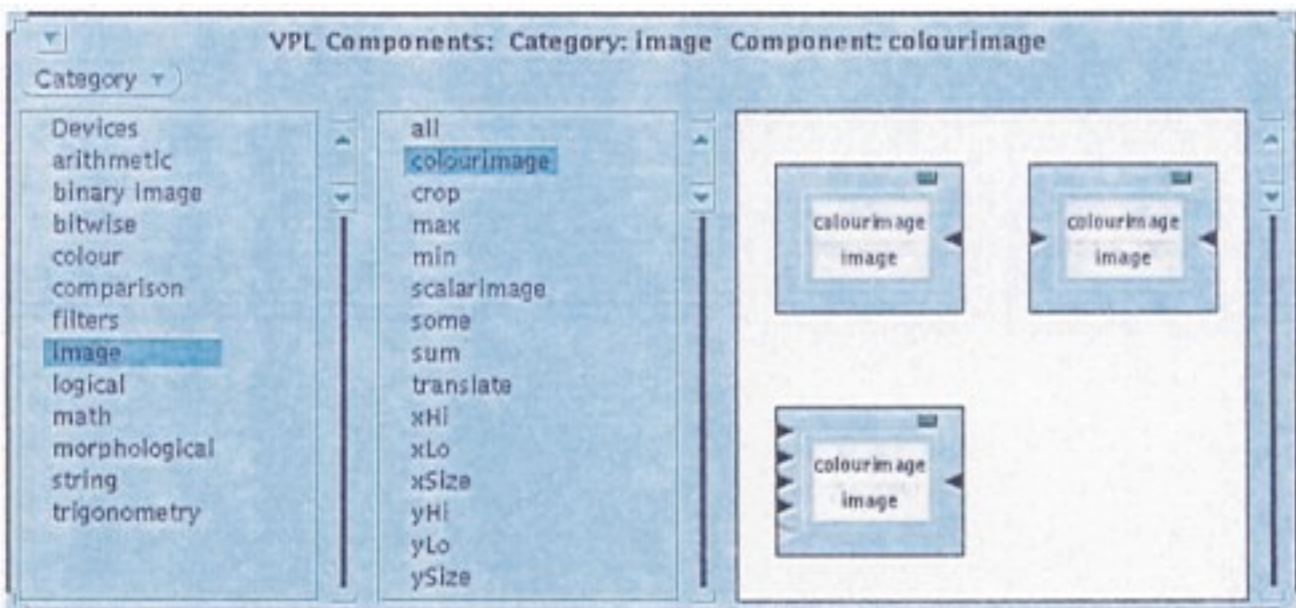
The Book can use visual representations in its descriptions. When they are present, they are active and can be dragged to other destinations.



6.3 The Component Browser

The Component Browser presents a view of part of the Database, and allows it to be interrogated and updated in a structured manner. The presentation of components is structured according to their grouping into categories. On starting up, the Component Browser requests the list of current categories from the Database, then allows the user to alter its view of the Database by moving between categories.

Components can be re-categorised, added or removed from the Database using the Component Browser. Dragging components copies them to other tools.



6.4 The Workspace

The Workspace allows components and data items to be combined, where the combination of objects describes a declarative, demand-driven, dataflow diagram. As components are added, removed, connected, disconnected and modified these activities are communicated to the Hub which forwards them, asynchronously, to the Graph Evaluator. As results are computed, they are sent back to the Workspace, via the Hub, for incorporation into a front end representation.

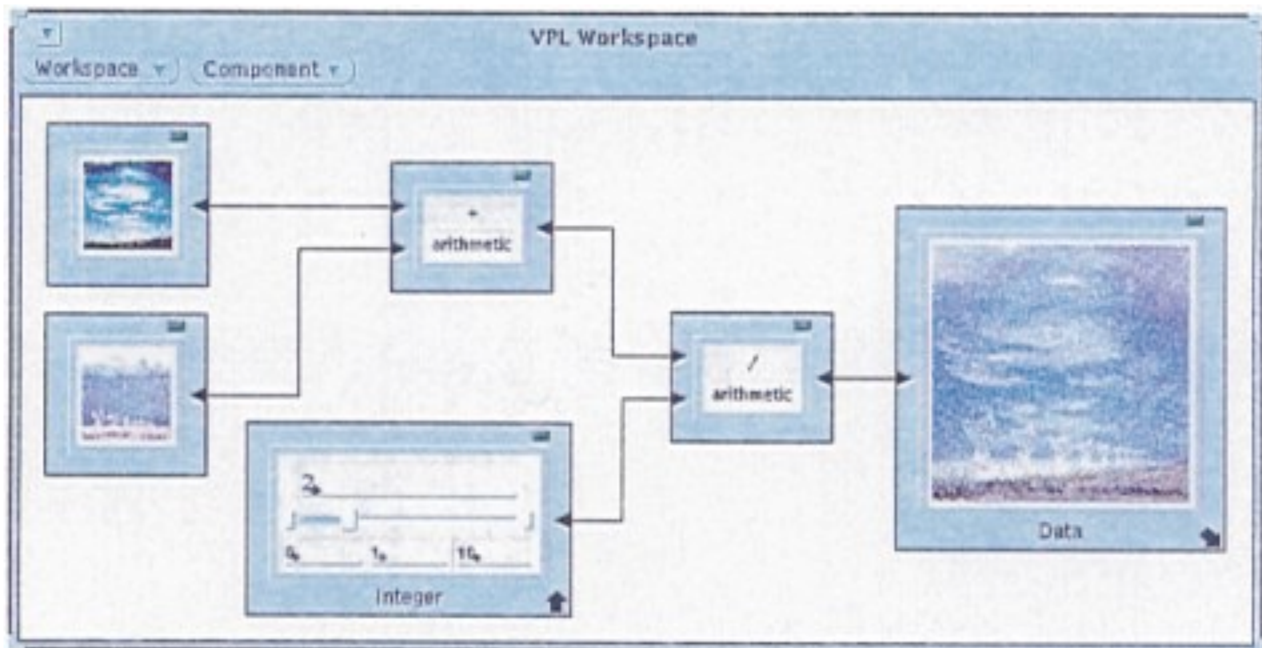
Data from the Workspace can be copied to the Book, where a description of it is given, the Data Browser, for storage, or to the Viewer.

When a program is seen to be functioning as required, it can be 'packaged up' (see Figure 5) and saved via the Components Browser as a new function.

The Workspace responds to an action of dragging and dropping a data object (from, for example, the Data Browser) by creating a special component called a Producer. A Producer has a central pane which gives a graphical representation of the data, and a single output port (which allows the data to flow from the Producer).

A Consumer is another special component, which is something like a Producer in reverse. A Probe is a Consumer which has a small window which gives a visual representation of the data flowing into it. Connecting a Probe into a graph has the effect of anchoring demand at that point, and so triggering evaluation (which propagates the demand back down the dataflow graph). When that segment of the graph has been evaluated, a data object is passed to the Probe for display and manipulation. The data in the Probe can be dragged to the Workspace (it becomes a Producer), to the Data Browser (it is stored as a VPL data item), to the Book (it is described) or to the Viewer.

In common with the other types of VPL window, several Workspaces can be open at one time. The Graph Evaluator simply sees a number of disjoint program graphs, and leaves the Hub to control communications routing.



6.5 The Viewer

With some modification, VPL, with its strong emphasis on back-end/front-end separation, could be applied to a number of different application domains. However, the Viewer (see Figure 3) is an application specific tool, which we introduced because of the need for a flexible and fast image viewing and manipulation mechanism. Referring back to Figure 1, the Viewer is somewhat separate from the other front end tools in that it does not require the Hub/ back end to carry out its basic function.

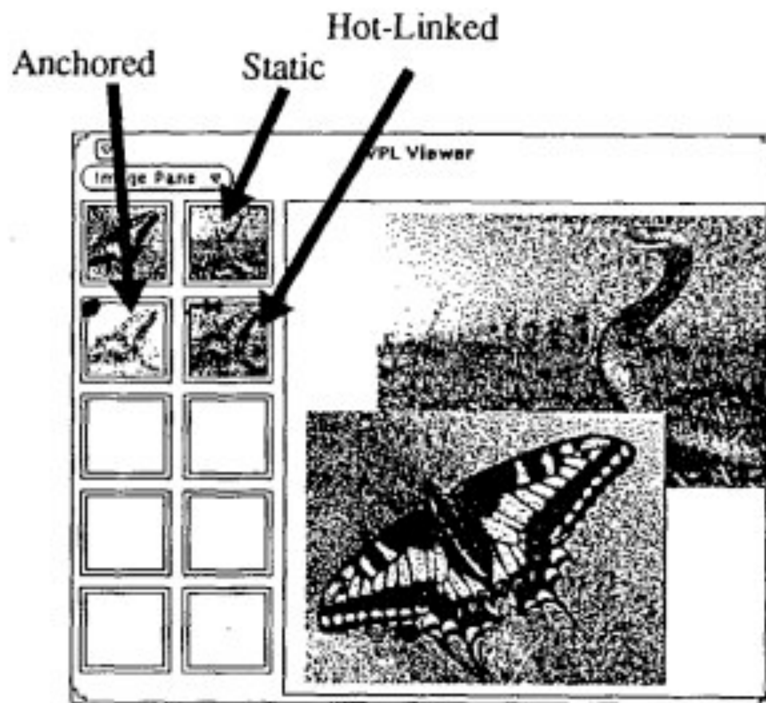
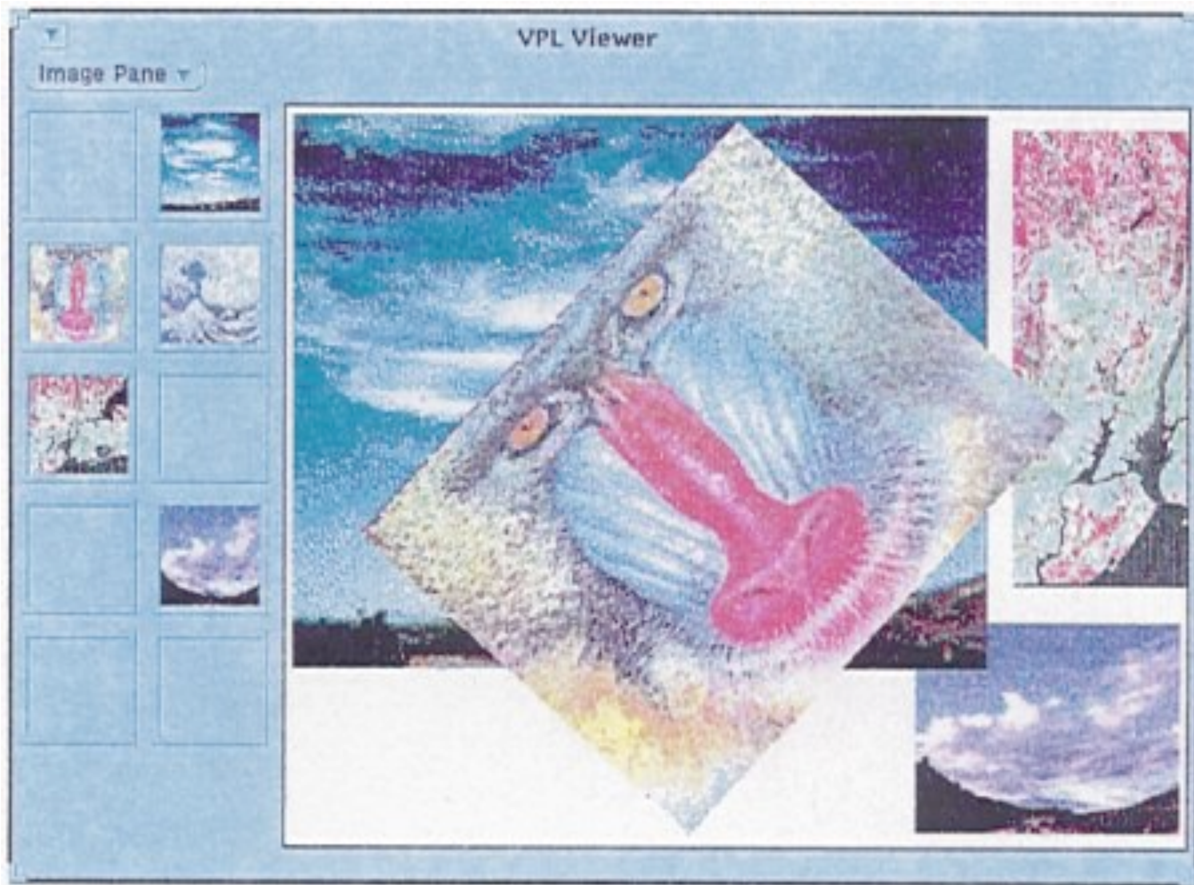


Figure 3

Image data objects can be dragged over to the icon dock on the left side of the Viewer, and dropped as a thumbnail copy. Clicking on the icon displays a full-size version of the image in the Viewer. Multiple images can be viewed together, allowing side by side comparison. Within the Viewer, individual images can be interactively zoomed, panned and rotated.

Images dragged into the icon dock from the Workspace are 'active': a 'hot link', providing change communication, exists between the Probe it was dragged from and the thumbnail image (and any full-size scaled and rotated version). As components in the Workspace are replaced or manipulated, images hot-linked to the Workspace are updated dynamically. Hot-linked images can be anchored to prevent changes to the source affecting the image.

Of course, docked images can be dragged to other VPL tools (the Workspace for reprocessing; the Data Browser for storage; the Book for description).



7 Programming Language Issues

VPL is a powerful programming language in its own right. To illustrate the expressive power of VPL this section presents a worked example of how to build a control structure using recursion and higher-order functions.

7.1 Essential Primitive Components

VPL provides three components to increase the power of its programming model:

- an If component
- a Generic Producer component
- an Apply component

Several frequently used dataflow structures are also provided as non-essential primitives but, as we show in ß6.1.4, even these can be constructed from the essential primitives.

7.1.1 The 'If' Component: The 'if' component is the equivalent of C's built-in '?:' operator. ('?:' is an 'if then else' expression, whereas C's 'if' is a statement.)

We handle 'if' specially because it cannot be defined as a C++ function (in the back-end library), since all C++ functions are strict in all their arguments. (All user-defined operators are also strict in C++, unlike the built-in "?:", "&&" and "11" operators.)

The If component takes three inputs and produces one output. When demand is placed upon the output the If component is evaluated. The effect of the evaluation is as follows:

- If there is no value on the first input port then propagate demand for it.
- If the value of the first input is 'True' then propagate demand on the second input port and forward the value returned as the output of the component.

- If the value of the first input is not 'True' then propagate demand on the third input port and forward the value returned as the output of the component.

Note that this is a demand driven evaluation sequence; only if there is a need for the value of each individual input is demand propagated along that input link.

7.1.2 The Generic Producer Component: The topic of visual programming languages is interesting in that it presents the language designer with a severe problems concerning the representation of concepts and constructs. For example, the notion of higher-order functions is very well known to language theorists, but in visual programming we obviously cannot use the theory until a mechanism for visually representing the use of a higher-order functions is defined.

VPL provides a special device, called a Generic Producer, which can be used as a mechanism to convert a function (call it 'a') into another function ('b') which, upon evaluation, produces the original function 'a'. See Figure 4. The effect of this simple device, the Generic Producer, is to provide a mechanism for passing a function as an argument to another function.

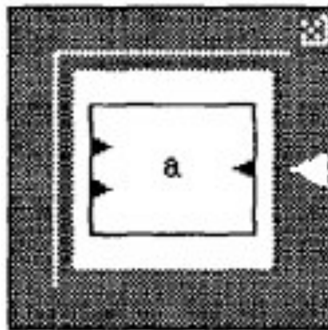


Figure 4

7.1.3 The Apply Component: In certain ways, the Apply component acts as the converse of the Generic Producer. The purpose of the Apply component is to take its first input (a function) and apply it to a second input, passing the result as its output.

Note that VPL provides support for higher-order functions, thus both the inputs and the output to the Apply component could be functions.

7.1.4 Building Control Structures in VPL: The literature on visual programming appears to spend much time on the choice and representation of control structures. In VPL we have chosen to address the problem from an alternative angle by providing the visual programming language with the power to describe its own control structures.

To illustrate this, consider the control structure 'Repeat', which repeats a section of code with a given argument a fixed number of times. (Repeat is so often used by programmers that we in fact provide it as a built-in.) However, the following example shows how VPL users could create a function which performs the same task themselves.

Stage l: Construct a dummy entry

The first step would be to create a dummy entry in the database for the Repeat function. The Workspace is cleared, and 'Create Composite' is selected from the Workspace menu (this places a thick border around the Workspace, which allows input/output ports to be added to the composite being built). Three input ports and one output port are added (by clicking on the border at the desired locations), and 'Save Composite' is selected from the Workspace menu. After a short dialogue to establish category and name, a database entry for Repeat is created, and the Components Browser is automatically updated to reflect this modification.

Stage 2: The real Repeat function is built

The Workspace is cleared again, and several components are dragged in from the Components Browser (including the Repeat dummy, an Apply component and an If component). These are linked together as

shown in Figure 5. (Note that the Workspace is continually active, if the user wishes to try out a section of code while building the new component then adding appropriate Data Producers and Probes will trigger evaluation).

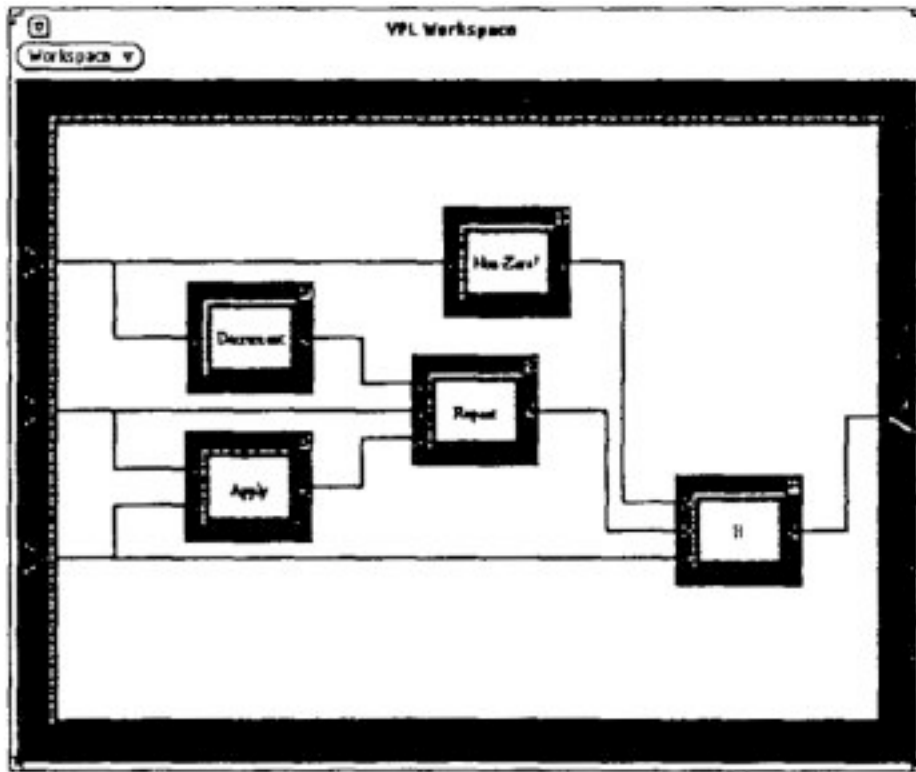


Figure 5

Stage 3: The new composite function is saved

Finally, 'Save Composite' is again selected from the Workspace menu. This time, the dummy database entry is replaced with the newly constructed, recursively defined, higher-order 'Repeat' control structure. The inputs to the function are an integer count, a function to be repeated and an argument to be passed to the function.

Figure 6 shows how the new function might be used to vary the level of 'enhancement' of an image.

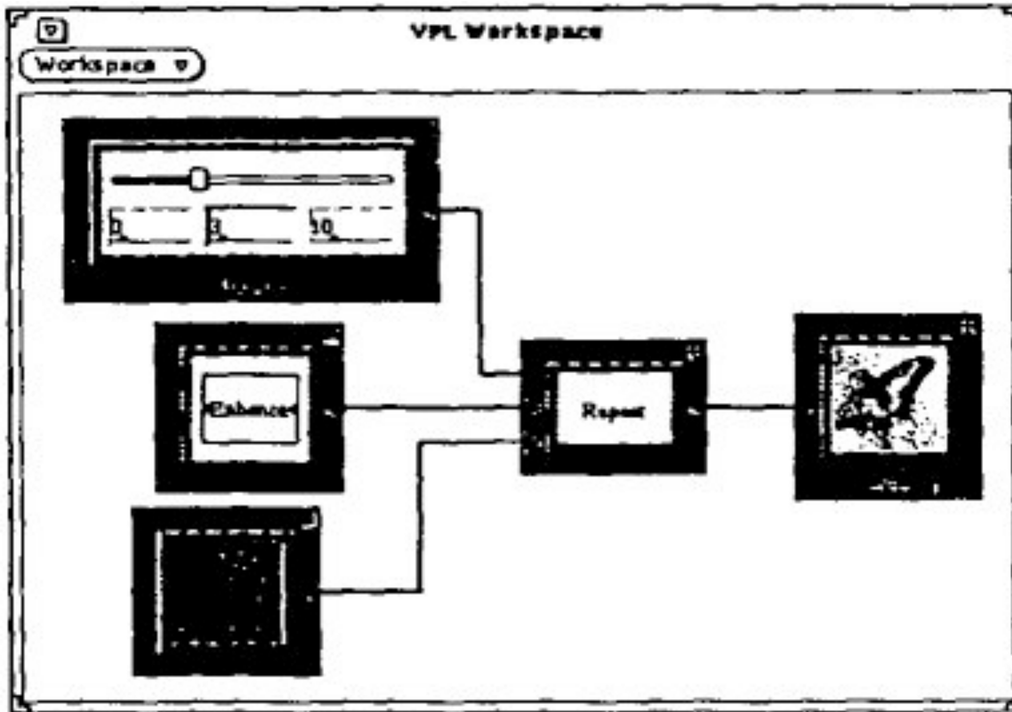


Figure 6

8 Summary and Conclusion

VPL is an interactive visual programming environment for image processing. For processing the constructed programs VPL makes use of a C++ back end image processing library called V-Sugar. However, the library's semantics are hidden from the user so that, for example, run-time polymorphism is gained from the provision by VPL of its own dispatch code.

VPL is a fully live system which executes as several Unix processes. Using a combination of synchronous and asynchronous requests in its communication protocol the system runs as a set of interacting, loosely coupled services.

The VPL front end provides a visual programming language interface in which dataflow diagrams are constructed. The nodes in these diagrams are *components*; functions and interactive devices; which, linked together, form a set of possibly disjoint, continually active program graphs. As soon as a component is dragged to the Workspace it becomes active, and the evaluation engine (running as a separate process) examines the newly modified program graph looking for parts that might now require evaluation.

VPL takes the view that for visual programming to be successful, it must take into account user interaction issues. In particular 'liveness', tool integration, and support for exploratory programming must be provided. Visual programming is more than a static, graphical program representation: it also concerns how the artifact is *created* and how the artifact *behaves*.

Acknowledgements

VPL is a part of the VIEW-Station image processing project being carried out at Canon Information Systems Research Center in Japan. We gratefully acknowledge the support of Dr. H. Tamura and the VIEW-Station team. Harold Thimbleby made very helpful comments and suggestions on the text for which we are grateful.

References

1. H. Sato, H. Okazaki, T. Kawai, H. Yamamoto, H. Tamura, "The VIEW-Station Environment: Tools and Architecture for a Platform-Independent Image Processing Workstation," Proceedings of the 10th International Conference on Pattern Recognition, Atlantic City, NJ, June 1990.

2. D. Lau-Kee, Y. Kozato, G.P. Otto, H. Tamura, "Software Environment of Vision and Image Engineering Work-Station - A Prototype Visual Programming Language for Image Processing," Proc. 41st National Conference of the Information Processing Society, Japan, 1990.

3. B.A. Myers, "Visual Programming, Programming by Example, and Program Visualization: A Taxonomy," Proc. CHI '86, April 1986.

4. S.L. Tanimoto, "VIVA: A Visual Language for Image Processing," J. Visual Languages and Computing, 1(2), June 1990.

5. W. Sutherland, "On-line Graphical Specification of Computer Procedures," Ph.D. Thesis, M.I.T. Cambridge, Mass., 1966.

6. M. Hirakawa, I. Yoshimoto, M. Tanaka, T. Ichikawa, S. Iwata, "HI-VISUAL Iconic Programming," Proc. IEEE Workshop on Visual Languages, pp. 34-43, 1986.

7. Ohio Supercomputer Center, "apE: Providing Visualization Tools for a Statewide Supercomputing Network," Proc. 24th Cray Users Group meeting, pp. 237-241, Trondheim, 1989.

8. C.S. Williams, J.R. Rasure, "A Visual Language for Image Processing," in Proc. EVE Workshop on Visual Languages, October 1990.

9. S.L. Tanimoto, "Towards a Theory of Progressive Operators For Live Visual Programming Environments," in Proc. IEEE Workshop on Visual Languages, October 1990.

10. G.M. Vose, G. Williams, "LabVIEW: Laboratory Virtual Instrument Engineering Workbench," BYTE, pp. 84-92, September 1986.

11. H. Tamura, H. Sato, H. Yamamoto, H. Okazaki, T. Kawai, "The Software Architecture of an Image Processing Workstation," Proc. 6th Scandinavian Conference on Image Analysis, pp763-769, Oulu, Finland, June, 1989.