VIPR and the Visual Programming Challenge*

Wayne Citrin
Department of Electrical and Computer Engineering
Campus Box 425
University of Colorado
Boulder, CO 80309–0425 USA

Soraya Ghiasi and Benjamin Zorn
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309–0430 USA

February 22, 1998

Abstract

The Visual Programming Challenge (VPC) provides a framework for exploring visual programming language issues in a quasi-real-time environment and for comparing competing languages. VIPR is an attempt to bring the traditional strengths of textual imperative languages to a visual programming language. It leverages these strengths to make use of well understood design and abstraction methodologies while providing additional visual features, such as explicit representations, contextual information, and execution animation. VIPR uses these strengths and features to address the quasi-real-time problem posed by the VPC, whose requirements include interaction with the low-level vehicle manipulation code, map exploration and display, as well as additional requirements such as flexibility of solution and performance demands. Our solution does not directly address pedagogical issues or the value of VIPR's simple graphical semantics. It did point out weaknesses with VIPR's approach to solving the scalability problem in visual programming languages.

*This work was supported in part by a grant from the National Science Foundation: NSF CISE-IRI-9616242.

1 Introduction

The Visual Programming Challenge (VPC) is a competition between different visual programming language groups using a variety of visual languages to program an vehicle composed of LEGOs(tm). The car has a small onboard computer that controls the motors and sensors, and interacts with the visual language. The onboard computer can be manipulated to maneuver the car around a track composed of LEGO road plates. The desired end result is a display map of the road plate layout discovered by the car.

The Visual Programming Challenge provides an opportunity to test visual languages in a small subset of quasi-real-time control problems. It is intended to help focus visual language research on a single problem for easier cross comparison between languages. The VPC analyzes competing languages based primarily upon pedagogical issues, but also examines the utility of the languages [1].

VIPR was selected as one of the participants in the 1997 Visual Programming Challenge. VIPR is a containment-based visual imperative language. It attempts to extend Kahn's idea of a completely visual language [11] to an imperative programming language [4]. VIPR programs are composed of a small number of visual components [9]. These visual components are supplemented by the use of the Tcl scripting language for expressions and statements. We chose to enter VIPR in the VPC precisely because of the strengths of textual imperative languages.

VIPR has many of the strengths and weaknesses associated with its related textual languages. These strengths include support for abstraction, familiarity, and flexibility. Weaknesses of textual imperative languages include the lack of explicit representations and the inability to simultaneously maintain both a local and global view of the program (e.g., detail in context [2]). Imperative languages have been very successful in the textual programming arena. Can they be similarly successful in the visual arena? VIPR is an attempt to bring the traditional strengths of imperative programming to a visual language. It supports top-down design, integrates well with existing code, and can be used to construct reusable abstractions to support functional composition. The goal of explicit representation of relationships used by VIPR is to make programs easy to understand and to minimize the number of hidden dependencies [10]. The pedagogy of the simple graphical semantics of VIPR was not tested by the VPC. In addition to the features mentioned previously, VIPR also supports the notion that any program snapshot is also a program that can modified and later have its execution continued [8].

The VIPR programming environment supports a number of functions to ease the process of writing and debugging VIPR programs. VIPR's environment supports animated execution. VIPR attempts to address the issue of scalability in visual languages through the use of zooming and fish-eyeing [6, 7]. These features were added with the goal of enabling rapid construction of solutions.

In this paper, Section 2 discusses VIPR in more depth, focusing on its syntax and semantics. Section 3 examines the VIPR environment and how it provides additional support for visual programming. The

remainder of the paper looks at the requirements of the VPC and how VIPR addresses them. Section 7 summarizes our conclusions.

2 VIPR

VIPR is a graphical transformation language developed to avoid the problems of scalability and detail in context encountered by some other visual languages [2, 6, 7, 12]. It is an imperative language that uses the Tcl scripting language for statements and expressions, and visual constructs for its control, including procedures [4, 5]. VIPR combines the ease of programming a simple, but flexible, scripting language with understandable visual semantics and the extensibility of a procedural language. Typing and scoping in VIPR are determined by the underlying Tcl interpreter and are beyond the scope of this paper.

Figure 1: A simple while loop in VIPR that counts down from 5

VIPR consists of a few graphical primitives that are used to compose a program. These primitives are rings, arrows, and parameter rings. Figure 1 illustrates a number of related VIPR concepts. The program sums the numbers from 5 down to 1 and prints the result. In the first statement (outermost ring), the program shown sets the value of a variable 'x' to 5 (set x 5) and 'Sum' to 0 (set Sum 0). Control then passes to a pair of sibling rings (nested inside the outer ring). Each ring supports a guard expression on the left and an action statement on the right. An empty or non-existent guard expression is assumed to be true. VIPR evaluates the guard expression on the left (expr $x > 0) and finds that it is true. VIPR then executes the action on the right (set Sum [expr $Sum + $x]) which adds the value of x to the current Sum. Next, the value of x is decremented (incr x - 1). Control then returns to the while loop (indicated by the arrow to the ring with the $x > 0 guard). This loop will execute until the guard is no longer true. When the guard becomes false, the sibling ring labelled 'otherwise' will execute and will print out the value of Sum (write $Sum).

VIPR's primitives can be combined to write programs of arbitrary complexity. Arrows are used for transferring control between various sections of the program. Control can be passed via arrows to other procedures or can be passed within a procedure to form iterative loops. Parameter rings are small rings that can be attached to a larger ring. They are used for passing parameters between procedures.

VIPR's semantics are based on the principles of sequencing, guarded execution, and substitution. Sequencing is seen in VIPR's ringed approach to its control syntax. The ringed approach in VIPR allows containment to determine the sequence of execution. As a VIPR program executes, each ring is successively merged into the outermost ring. This containment-based approach can be summed up as "Follow the rings deeper into the nest". Containment, or nesting, is used to build programs that do no suffer from the graph edge problem [6, 7].

Guarded execution is VIPR's method for allowing conditional execution. Each ring may have a guard (upper left portion of ring) and a statement for execution (upper right portion of the ring). Each guard has a boolean value. If no guard appears on a ring, it is assumed to be true.

Substitution is the remaining element of VIPR's control semantics. Arrows are used to connect sections of the program together. In addition, the arrows determine the section of program that will be substituted during execution. Although difficult to visualize without seeing a VIPR program execution, VIPR will execute a procedure by substituting the procedure into the current procedure. This effect is similar to the idea of inlining functions, but it is done during execution rather than during a compile phase. Substitution is needed to support editable and executable program snapshots.

VIPR's semantics can be summarized as follows:

- Follow the rings deeper into the nesting (similar to following successive statements in a C program).
- If there's a guard (expression on the left), follow its rings only if the guard statement is true.
- If there's an arrow, follow it as it transfers control.

Because even small programs involve a significant amount of nesting, deeply nested rings would pose a problem for readability if VIPR's environment did not support zooming. Zooming allows the programmer to specify a code fragment and enlarge it to the point where it may be easily read and manipulated. The programmer may shrink or enlarge the program as much as needed.

Figure 2: VIPR's graphical primitives support recursion.

VIPR's control primitives can be used to create arbitrary control structures. The primary control mechanisms in VIPR are iteration, in the form of while loops, and recursion. A more detailed example of a recursive factorial program is shown in Figure 2. The example shown calculates the factorial of 3 by recursively calling itself until the parameter passed in (n) reaches 1. The main procedure is on the set of rings with the bar attached. The procedure calls are those rings with 2 parameter rings and an arrow pointing to another ring. In the figure, we see two calls, one from inside main, and another, recursive calls, from within the factorial function. A parameter ring can be seen with the expression 'expr 3' attached. The initial call to the recursive function is made from the starting procedure. The factorial function evaluates the guard expressions to determine whether or not any work remains. If no work remains (expr $n == 1), control is passed back to the main loop through a parameter loop (the program continuation is explicitly passed as a parameter, seen here as the upper parameter ring in the function call). If work remains to be done (expr $n > 1), the procedure adds the current value of n to the running total contained in x (set x [expr $x + $n]), decrements n by one, and calls itself with the new value of n.

3 VIPR's Environment

The VIPR environment provides users with program development support, including simple browsing through a direct manipulation interface. Additionally, it provides insertion, deletion and copying capabilities. The

Figure 3: VIPR's environment provides tools for manipulating a VIPR program

standard interface that VIPR programmers are presented with is illustrated in Figure 3. To get a feel for using the environment, we explain the toolbar here. Selection (the arrow in the toolbar), allows the program to select an item for deletion or duplication. The hand can be used to move a program around in the program window and is used frequently for adjusting fish-eyed programs. VIPR's toolbar also supports adding rings outside the current ring (add a parent ring), inside the current ring (add a child ring), and next to the current ring (add a sibling ring – used for conditionals). Parameters can be added when creating function calls. The tool icon with two small rings connected together allows the programmer to connect rings with arrows. The icon to the right of the arrow connection icon is used for introducing guard and action pairs. The stoplight starts program execution, while the stop sign halts it. The last row of icons are used for zooming in and out of a program.

VIPR's environment is an integral part of VIPR's attempt to reduce the graph-edge problem that plagues control-flow languages. The use of containment reduces the number of line crossings, but also means that program statements nested deeper in the procedure are difficult if not impossible to read. To remedy this problem, VIPR's environment supports zooming and fish-eyeing. Zooming allows the user to focus on code deep in a procedure, but at the expense of losing the surrounding context.

For situations where the user prefers to maintain the context in which the code occurs, VIPR's environment provides the ability to use a fish-eye view based on a spherical projection [6, 7]. The distortion causes one section of the code to be enlarged while maintaining enough additional contextual information for the user to still identify where the procedure was called.

VIPR's environment also allows full run-time animation of the program code as it executes. This allows the user to observe the program as it runs including watching as arrows are followed into procedures and the substitutions performed to maintain a copy of code that will be called again.

The run-time animation system also allows the user to step through the program, executing one program ring at a time. Because VIPR supports the idea that every program snapshot is a program in its own right, single-stepping can be used to debug programs. A program can be run by single stepping through it or can be stopped at any point in the execution. A program snapshot can then be edited and execution can be continued.

4 VPC Requirements

The Visual Programming Challenge imposes a number of requirements on the contestants. These requirements range from interaction with the low-level Handy Board commands to performance demands placed on the system by the need for quasi-real-time behavior.

4.1 Functional Requirements

The functional requirements imposed on a solution to the VPC are determined by the statement of the VPC and decisions made by the participants. At the lowest level, Handy Board commands allow the user to control everything from the threshold values used to differentiate black from white to the speed at which the motors turn the wheels. These commands also allow the user to retrieve information from the Handy Board detailing the current state of the sensors. Above the low-level control, commands to manipulate the LEGO vehicle and to determine what to do next are needed. The code that makes these decisions does not necessarily need to be exposed to the user, but we chose to make it visible. Alternative possible solutions range from implementing the lowest level in VIPR to encapsulating control and decision making operations as high level commands with most of the code written in a non-visual language.

The VPC poses additional functional requirements including the traversal of the road plates and the display of the course. These requirements led to the design of procedures that handled the initialization of the Handy Board as well as procedures for line following, plate recognition, and map composition and display. How these procedures are constructed and what information processing is done in each is determined primarily by how much information a particular solution chooses to expose to the user.

Simple data structures are required to keep track of the current position of the vehicle, the current state of the Handy Board including sensors, and as much of the map as has been discovered. VIPR does not support visual data structures and our solution does not use anything beyond the data structures available in Tcl, specifically associative arrays.

4.2 Additional Requirements

Additional criteria that VPC solutions must meet do not relate directly to specified functionality. For example, the VPC does not specify the speed at which the vehicle traverses the track. However, the latency between when a command is issued nad when it is executed must be small enough that the vehicle moves around the track in a quasi-real-time manner. It is not necessary for the vehicle to constantly be in motion, but it is unacceptable for the vehicle to take hours to traverse the track.

A number of factors contribute to latency. The motors move the vehicle by pulsing periodically and then reading the sensor values. Because the motors are not of equal power, the vehicle can drifts off-course and course corrections must be made. Additional processing is required to support plate identification. These

factors need to be balanced against the need for quasi-real-time motion. In the VIPR solution, we chose to implement all the code that handles this decision process directly in VIPR.

A criteria that is mentioned in the VPC requirements, but not given equal weighting with pedagogy, is that the solution be flexible [1]. This requirement implies that the solution, and the language, support code reuse or that the solutions be trivial to construct. The VIPR solution has substantial flexibility due to abstraction and procedure reuse. The scope of the VPC did not present us with the opportunity to fully explore this dimension of VIPR.

Another requirement of the VPC is that the solutions should be easy to construct. Because the VPC does not specify for whom these solutions should be easy to construct, the question of how much information the programmer/user should be expected to know arises. A solution intended to be programmed by young children will be considerably different and require much less knowledge of the low level system than a solution designed for robotics hobbyists. We selected Pascal/C programmers as our target audience. Because VIPR is not explicitly intended for use by non-programmers, the level of programming experience required is substantially higher than some other entries in the VPC.

Although the VIPR solution strives to meet all the VPC requirements, there are still aspects of VIPR that the VPC solution does not test. VIPR is specifically designed to support scalability. The extent of the problem posed by the VPC does not stress the VIPR environment support for scalability enough to determine whether or not this support is adequate or useful. The VIPR solution translated into C would consist of roughly 100 lines of code, not including variable declarations, spread over 5 procedures.

The issue of ease of use is related to that of simple graphical semantics used by VIPR. To understand whether VIPR's simple graphical semantics actually make VIPR an easier language to learn imperative programming, tests using naive users could be constructed, but have not been thus far.

5 Design

While the requirements of the VPC influenced the VIPR solution, the strongest influences came from the nature of VIPR. The imperative nature of the language supports traditional design approaches. The explicit nature of representations show relationships clearly.

Because VIPR is a visual imperative language, it is possible to bring the strengths of imperative coding to a visual environment. In this case, a traditional top-down design method was used to break down the problem and implement the solution. This approach allowed the programmer to examine the problem as a whole, and then determine the best method to isolate portions of the problem, leading to a standard layered design. The high-level architecture, shown in Figure 4, shows the problem breakdown used in the VIPR solution. In addition to a strong tradition of top-down design, many imperative languages support

Figure 4: The high level architecture of VIPR's VPC solution

code reusability. The various procedures written in both VIPR and as Tcl extensions in C are reusable. The potential exists to develop VIPR libraries for either or both approaches. It is also possible to develop reusable abstract data types. The selection of data types is limited, but is determined as much by the functionality provided by Tcl as it is by the needs of the particular solution. In this case, an associative array was used.

VIPR has an advantage over some more traditional imperative languages in that it explicitly shows interfaces. In a textual language, there are certain markers that indicate that a new procedure has been called. In VIPR, this relationship is made explicit through the use of arrows. These arrows denote the connections between the various procedural abstractions. This explicit representation provides a "map" of the code and how it will execute that may be more easy to follow than a text-based imperative language.

These factors lead to a striking similarity between the code architecture and the coded solution shown in Figures 5 and 6. The contributions of top-down design and explicit interfaces are particularly important because they allow a VIPR program diagram to maintain similarity to the original high-level architecture of the problem solution.

Figure 5: The code architecture shows the end result of a top-down design

Figure 6: The VIPR VPC solution reflects the code architecture

6 Implementation

The actual implementation of the VIPR solution hinges on the design decision delimiting the boundary between low level support code and VIPR code. Low level support code is the code used by a solution to a problem that is written in a language other than the visual language. The VIPR solution chose to mimic the original Lisp functions provided by the VPC Committee. These low level functions handle the actual serialization of commands to the vehicle and provide a simple interface to call these functions. These functions were translated out of Lisp and into C so that VIPR could call them as Tcl extensions.

VIPR exploits the ability of Tcl to be modified through Tcl extensions. Tcl provides a simple framework for making quick extensions to the language. These extensions are written in C and then called as Tcl functions. The simple Tcl API used is described by Ousterhout [13]. All interactions are done between VIPR and Tcl or between Tcl and C. VIPR calls a Tcl extension that relies on the underlying C code to execute the command. The ability to quickly add Tcl extensions provides VIPR with an extremely flexible boundary between support code and VIPR code. This boundary can be adjusted on a per application basis and is chosen by the programmer instead of dictated by the language.

The decision about where to define the boundary between VIPR and Tcl extensions strongly affects the overall solution to the VPC. It determines how much information the programmer needs to know before

coding a solution. The decision to limit the amount of support code to only direct manipulation was due to the choice of Pascal/C programmers as the target audience and to our desire to explore the flexibility of VIPR as much as possible within the confines of the VPC. To understand the VIPR solution to the VPC, we walk through the code shown in Figure 6 and then explain one of the procedures, Line Following, in greater detail.

The high degree of similarity between the code architecture (Figure 5) and VIPR's VPC solution (Figure 6) is due to the implementation of the solution suggested by our top-down design. VIPR accesses the low level commands in limited locations in the VIPR solution, including the initialization code. The initialization procedure simply sets a number of parameters including the threshold for differentiating between black and white and the current command number. These parameters are then passed on, via a Tcl extension, to low level code that manipulates them into the byte codes expected by the Handy Board run-time interpreter and sends them via a serial port to the Handy Board. Similarly, the actual commands to move the car by pulsing the motors and the commands to retrieve sensor values are called only in the Line Following routine. This corresponds to the suggested architecture where only Initialization and Line Following call low level code to manipulate the car.

The code architecture defines the interface between the code modules. With only minor exceptions (global variables that are needed in multiple locations), information that is local to one procedure is not accessed from another. This distiction is not as well maintained between Plate Identification and Line Following, the reasons for which will be discussed later.

The VIPR solution consists of 4 procedures. Initialization handles the set up of the car and serial port. Mapping creates and maintains the explored map. It also has responsibility for correctly placing the just explored road plate on the map. Plate Identification keeps track of information, such as turn indicators, which it then uses to determine which type of plate has been traversed and the current orientation of the car. Line Following is responsible only for reading sensors and pulsing motors. When the Line Following routine encounters situations that require knowledge about the plate, information is passed back to Plate Identification.

The VIPR solution contains run-time decision making procedures. These are the road plate identification procedure and the line following procedure. These procedures are closely related and somewhat intertwined.

Map management is handled by the map procedure. It takes information provided by the road plate procedure and simply keeps track of the road plate type, its position in a grid and unexplored road plates adjacent to explored road plates. The procedure could be made more intelligent since it is possible in many cases to determine what type an unexplored road plate is by the surrounding plates. The map management procedure then uses a Tk-based display function to display the map to date.

Before the VPC, VIPR supported very limited display capabilities. VIPR maintains a small display window which shows the current values of variables, but this is incapable of showing any graphics. Similarly, VIPR can display a small input or output window, but again, this is incapable of anything beyond simple

ASCII graphics and interferes with the continuation of the program because it requires active dismissal of the window by the user. In order to solve the larger problem of display in VIPR, a Tk display capability was added. Tk runs as a separate process and receives Tk commands via a socket connection from VIPR. These simple Tk extensions tell Tk to display a bitmap of a a road plate in a certain orientation and position. This allows the program to display any information it needs without conflicting with the event loop used by VIPR. Simple Tk extensions can again be written to enable more complex display such as the VPC map.

6.1 VIPR Line Following and Plate Identification

Line following is a simple procedure that attempts to follow a line based on the information provided by the sensors. It performs simple corrections in direction by pulsing the motors different amounts when the need arises. It has very little ability to determine behavior beyond this simplistic line following technique. It relies primarily on the notion that the center belly sensor should always read black due to the black tape denoting the center of the road plate path. Because motor pulsing leads to small incremental motions, it is possible to correct the direction based on the absence of a black reading in the center sensor and the presence of a black reading in either the left or right sensor.

This simplistic approach can lead to difficulties, however, because T and + intersections have additional tape markings that confuse the line following procedure. In the situation where all sensors, belly and wing, read black, line following assumes that the end of a plate has been reached, irregardless of whether or not it has. In such situations, the line following procedure returns control to the plate identification procedure. The plate identification procedure then examines the data structure returned by the line following procedure. This structure keeps track of the turn markers seen by the wing sensors and whether or not it is possible to have reached the end of the plate yet.

For example, a vehicle traversing a + intersection should see two full sets of turn markers before exiting the plate. The turn marker sighting can be differentiated from the center of + plate marker by the combination of the "distance traveled" and the fact that all sensors will read black as it crosses through the center of the plate. While turns are more difficult, similar reasoning can be applied. The road plate identification procedure determines whether or not the plate has been traversed. If it has not, it again calls the line following procedure, potentially with a flag telling the procedure to ignore certain values for a number of pulses. This can be used to clear intersections and make turns.

To get a feeling for how the VIPR solution looks and how it works, we walk through the VIPR code for line following. Additional details relating to Plate Identification are also exposed due to our implementation decisions. A series of 4 pictures of Line Following and 1 picture of Plate Identification are presented to accompany this discussion. The Line Following pictures (Figures 7-10) represent zoomed in images of the previous picture. In each case, the zoomed in section in one picture is surrounded by a dotted lines in the preceeding picture.

Figure 7: The entry point into Line Following

Figure 7 shows the entry point into Line Following. Line following has two parameter rings, one for returning to Plate Identification and the other for passing the $car structure in and out of Line Following. The code shown in Figure 7 is concerned primarily with initializing the values in $car to 0 before an inquiry of the car is made. The commands shown (set car(LW) 0, etc) expose Tcl's method for constructing data structures. set car(LW) 0 sets the LW (left wing sensor) element of $car to 0. These initializations are done only to ensure that the same behavior can be expected immediately upon entrance and while executing the while loop.

Figure 8: The main while loop in the line following procedure

Figure 8 is an enlargment of the dotted line box in Figure 7. This is the high level view of the main while loop that drives the car. The uppermost left ring (labelled with the guard expr $ignore == 0 && (expr $car(LW) > 0 || expr $car(RW) > 0)), handles the special case where Line Following can no longer determine what action to take and must return control to the Plate Identification procedure which maintains information about the plate. The expr $ignore == 0 portion of the guard allows our solution to simulate a don't care state. Line Following will return if and only if it has not been told to ignore the wing sensor values during this pulse of the motors. The upper right set of rings are executed in all cases where expr $ignore == 0 or neither of the wing sensors is on. Ignore is decremented by one if it is greater than 0 and then control is transferred to the remainder of the program (lowermost set of rings). This behavior is coded as a nested if-then-else.

The lowermost set of rings begins the actual work of manipulating the car. The current synchronization number is obtained by querying the car with the setCar Tcl extension. In some situations, the synchronization number is needed to ensure that a given command has executed. The Tcl extensions are done at a low level and this is reflected in the knowledge the user must have to work with the car. In this particular case, the command takes 3 arguments, synchronization number, speed and maximum distance. Speed and maximum distance were determined through trial and error. The status command is then called. An array of values (Tcl does not support multiple return values) representing the current state of the car's sensors plus some additional information is returned. Some of this information will be used later to determine what commands will be used to move the vehicle. Other information, such as the left and right wing sensor values, is useful for loop control (see Figure 7) and for passing back to Plate Identification.

Figure 9: The current sensor readings are obtained and read into the $car structure.

Figure 9 shows the VIPR code for filling in the structure given the array of values representing the car's state. It is shown primarily to provide an example of how structures in Tcl, and hence VIPR, work. The

code shown copies information from the current status array into the $car data structure for later use. The innermost circle visible in Figure 9 is expanded in Figure 10. Figure 10 shows the VIPR code that drives the car. Decisions are made based upon the values of the left and right belly sensors. The simplest line following routine we could think of was that the central belly sensor should be on and the right and left belly sensors should be off. Our code is a reflection of this. Figure 10 illustrates an if-then-elseif-else structure in VIPR. If the right sensor is on ($car(RB) > 0), the car has drifted to the left, and a resume command is issued to correct this. Similarly, if the left sensor is on ($car(LB) > 0), the car has drifted to the right and a resume command with different parameters is issued. If neither sensor is on, then both motors are pulsed. Resume takes two arguments, one for each motor. A argument value of 8 to resume means pulse the motor at full speed, while a value of 0 means don't pulse.

Figure 10: This code calls the resume command with various arguments. Resume causes the car to pulse its motors.

Line Following performs only the necessary functions to move the car along. Decisions that involve knowledge of the plate are made by Plate Identification. Figure 11 shows a subsection of the plate identification code. It presents a brief introduction to the methodology used to identify road plates and how long to ignore inputs while passing over additional line markings. The code examines the current value stored in $plate(leftIndicator) to see how many left turn indicator marks have been seen this far. Each of the conditions of the if-then-else then updates the $plate structure and sets a variable, ignore, with the number of pulses to ignore. This variable allows the line following procedure to ignore the wing sensor values for the specified number of pulses which enables it to pass over road plate markings.

Figure 11 is one of many cases contained inside a while loop with a guard condition of (expr $car(LW) > 0 || $car(RW) > 0). The loop is executed until all the necessary updates to the $plate structure have been done and then line following is called again. We chose to give only a brief overview of Plate Identification due to the complexity of the code.

Figure 11: This small subsection of Plate Identification analyzes the case where the left wing sensor is on.

6.2 Non-functional requirements and VIPR's solution

The non-functional VPC requirements also played a large role in the final VIPR implementation of the solution. The importance of reducing the latency between when a command is issued and when it is executed led to an analysis of the contributing factors because we felt this latency was initially too large. The largest contributor to this latency was the animation of program execution. This led to a modification in VIPR's run-time environment that allows the user to turn off execution animation. Run-time animation is still important, but it detracted from the final solution. The decrease in command execution latency allowed

much more rapid maneuvering. With animation, it took over 30 seconds between motor pulses. Without animation, this dropped to a much more acceptable 1 second per pulse.

VIPR was able to address the non-functional requirement of flexibility without any code modifications. VIPR supports the requirement of flexibility through reusable code. This code can be either in the form of existing VIPR programs which can then be edited to support additional functionality, including insertion of new procedures and functionality, or in the form of new Tcl functions (Tcl extensions). VIPR's approach to a flexible VPC solution incorporates both elements.

6.3 Experiences

The experience of using VIPR in the VPC has pointed out a number of positive aspects that make VIPR a potentially useful language. The combination of execution animation and snapshot capability provide VIPR with the ability to display values and program state dynamically as the program executes. This information, and the ability to stop execution when something goes wrong, allows the user to exploit program snapshots. Because each program snapshot is also a program and because VIPR is an interpreted language, it is possible to correct an error in a snapshot and then continue execution.

There are also negative aspects of the current implementation of VIPR that detract from its overall appropriateness as a language for the VPC. Containment addresses some aspects of the scalability problem, but still has disadvantages over a flat text representation. It is not currently possible to quickly scan through code or to search for a particular symbol. This problem could be addressed by the expansion of editing support. For example, a search tool would enable the programmer to specify a symbol and the VIPR environment would search through its own internal representation of the program diagram to find the appropriate symbol. It could then fish-eye the appropriate section of code for the programmer's perusal.

Zooming somewhat alleviated these problems. It was both usable and necessary for maneuvering through a program diagram, but it too has disadvantages. Currently, the programmer must know where things are to find them. While this was acceptable for something as small as the VPC solution, it quickly becomes unusable as programs become larger or have multiple contributors. Fish-eyeing is somewhat better than simple zooming because it allows the focus to be moved without zooming in and out, but currently distorts more along the edge of the sphere than is necessary [3]. The selection of a different projection may lessen this distortion.

7 Summary

VIPR's traditional imperative language approach coupled with additional visual features can be used to address needs ranging from the initial design to the final debugging and displaying of a solution. VIPR's strengths lie primarily in the fact that imperative languages have a well developed body of information and techniques that allow them to work well in a variety of situations. VIPR attempts to translate these strengths

into a visual language including support for data and procedural abstractions. The additional contributions made by the "Visual" in "Visual Imperative PRogramming" allow the programmer to more explicitly see relations between functions and to maintain a better understanding of contextual information.

These contributions reflect that the target audience of VIPR is expected to be able to understand and make use of these advantages. While this is not necessary, it does influence the types of solutions that will be constructed. VIPR's solution to the VPC relied heavily upon the top-down design process and data hiding through data and procedural abstraction. The layers of abstraction necessary for this approach are easily seen in the program diagram.

VIPR addresses the requirements of the VPC by relying on the strengths inherent in its underlying design. The extensibility and flexibility of reusable code and Tcl extensions allowed VIPR to address the VPC on a low level basis. The direct interaction with the low level Handy Board commands focuses attention on VIPR's intended audience: existing imperative language programmers. However, this focus removes attention from the less experienced audience that many other VPC solutions were directed towards. Without taking the difference in audiences into account, VIPR may appear pedagogically weaker.

While VIPR did address many of the issues raised by the VPC, some of the remaining issues were not well defined in the VPC or were not explored by VIPR's solution to the problem. The VPC did not directly address the ease of learning the participating languages. The target audience would either need to be specified or cross comparisons between each of the languages and each of the target audiences would need to be performed. User tests would need to be constructed and performed to fully evaluate which languages provide the least pedagogical impediments under which circumstances.

VIPR's designers intended that VIPR would support scalability and simple visual semantics. Neither of these issues was tested by the VPC. The VPC was not a large enough problem to stress VIPR's scalability. It did show some of VIPR's weaknesses, however, by drawing attention to the lack of an effective search tool and to the extent of distortion around the edge of the spherical projection used for fish-eyeing.

Acknowledgments

This work was supported in part by a grant from the National Science Foundation: NSF CISE-IRI-9616242. The authors would also like to thank the members of the Visual Programming Committee, the referees, and especially Allen Ambler for their guidance, support, and suggestions.

References

[1] Allen L. Ambler, Thomas Green, Takayuki Dan Kumura, Alexander Repenning, and Trevor Smedley. 1997 visual programming challenge summary. In *Proceedings of the 1997 IEEE Symposium on Visual Languages*, pages 11–18, Capri, Italy, September 1997.

[2] Margaret M. Burnett, Marla J. Baker, Carisa Bohus, Paul Carlson, Sherry Yang, and Pieter van Zee. Scaling up visual programming languages. *Computer*, pages 45–54, March 1995.

[3] M. Sheelagh T. Carpendale, David J. Cowpertwaite, and F. David Fracchia. Making distortions comprehensible. In *Proceedings of the 1997 IEEE Symposium on Visual Languages*, pages 36–47, Capri, Italy, September 1997.

[4] Wayne Citrin, Michael Doherty, and Benjamin Zorn. Formal semantics of control in a completely visual language. In *Proceedings of the 1994 IEEE Symposium on Visual Languages*, St. Louis, MO, October 1994.

[5] Wayne Citrin, Michael Doherty, and Benjamin Zorn. A graphical semantics for graphical transformation languages. *Journal of Visual Languages and Computing*, 8(2):147–173, April 1997.

[6] Wayne Citrin, Richard Hall, and Benjamin Zorn. Addressing the scalability problem in visual programming. Computer Science Technical Report CU-CS-768-95, University of Colorado, Campus Box 430, Boulder, CO 80309, April 1995.

[7] Wayne Citrin, Carlos Santiago, and Benjamin Zorn. Scalable interfaces to support program comprehension. In *International Workshop on Program Comprehension*, Berlin, March 1996.

[8] Wayne Citrin and Benjamin Zorn. *VIPR Reference Manual*. Campus Box 430, Boulder, CO 80309, January 1997.

[9] Wayne Citrin and Benjamin Zorn. *VIPR Users Manual*. Campus Box 430, Boulder, CO 80309, January 1997.

[10] Thomas Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[11] Kenneth M. Kahn and Vijay A. Saraswat. Complete visualizations of concurrent programs and their executions. In *Proceedings of the 1990 IEEE Workshop on Visual Languages*, pages 7–15, Skokie, IL, October 1990.

[12] E. Howard Kiper and C. Ames. Criteria for evaluation of visual programming languages. *Journal of Visual Languages and Computing*, 8(2):175–192, 1997.

[13] John K. Osterhout. *Tcl and the Tk Toolkit*, chapter 30, pages 293–303. Addision-Wesley Professional Computing Series. Addison Wesley, Reading, MA, 1st edition, 1994.