Execution Visualization and Debugging in Three-Dimensional Visual Programming

Toshiyuki Okamura Buntarou Shizuki
Jiro Tanaka
Department of Computer Science,
Graduate School of Systems and Information Engineering,
University of Tsukuba,
{okamura,shizuki,jiro}@iplab.is.tsukuba.ac.jp

Abstract

To help programmers debug visual programs, we propose an animated execution method with supporting functions to be used with the animated execution. The animated execution animates state transitions as the program is executed. To make it easy for the programmer to interpret the animation, it is displayed in a manner similar to the visual program. The functions are used to narrow the possible locations of bugs and to test fixed components quickly. We have implemented animated execution and the functions on 3D-PP, our three-dimensional visual programming system.

1. Introduction

Unlike casual text-based programming, which represents programs with text, visual programming represents programs with graphs. Graphical representations are quite good at showing explicit relationships between components and data dependencies.

However, it is still necessary to consider how to represent the execution of visual programs for debugging. First, debugging a program usually requires the following three functions:

Monitoring data during runtime. Data are monitored to compare the input data of a component with the corresponding output as the basis for checking the correctness of the component. This is also used to narrow the possible locations of a bug to one or some components.

Monitoring the control flow of the execution. The control flow is monitored to determine whether there is buggy behavior. This also helps the programmer to deduce the type of bug. For example, if the execution terminates much more quickly than the programmer expects, she/he can guess that the termination is caused by a fault such as division by zero.

Testing a component. Once a component is modified or newly programmed, it is necessary to test the component with some input values. The programmer should be supported so that this test can be performed easily.

Moreover, it is desirable to provide functions for monitoring data and control flow in a user-friendly manner. If the execution is represented with text, the programmer is forced to associate the textual representation of the execution with the original visual representation of the program.

The goal of this paper is to provide the above three functions in an integrated, user-friendly manner. To achieve this goal, this paper proposes (1)*animated execution* and (2) additional functions for debugging that are used with animated execution. Animated execution provides a mechanism for browsing data and monitoring the control flow. It "*executes the program directly*"; i.e., it animates execution using the same representation as the visual program. Therefore, the programmer can easily understand what the animated execution shows. The additional functions provide mechanisms for testing components and for narrowing the possible locations of bugs.

We have implemented animated execution and the additional functions on 3D-PP[7, 8], the three-dimensional extension of our two-dimensional visual programming system PP[11].

2. 3D-PP

A 3D-PP program is a hierarchical three-dimensional graph composed of nodes and edges. The nodes correspond to *data*, *operators*, and *processes*. The shape of a node indicates its type: spheres represent data, such as integers and strings, or variables; inverted cones represent operators,

Proceedings of the Eighth International Conference on Information Visualisation (IV'04)
1093-9547/04 $ 20.00 IEEE

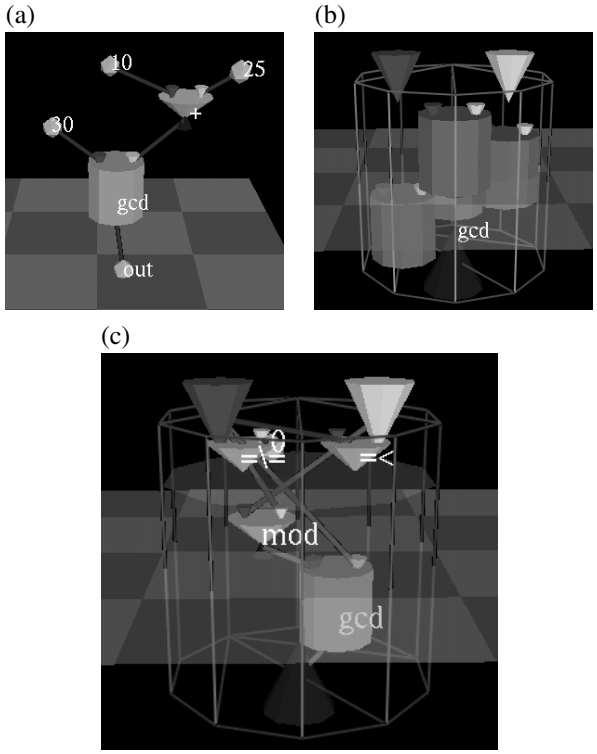

Figure 1. Programming constructs in 3D-PP

such as addition, modulo, and comparison; and pillars represent processes. An edge corresponds to data dependency.

Figure 1a shows an example program. The program adds 10 and 25. It also spawns a process labeled gcd with 30 and the result of the addition as the arguments. The result is assigned to variable out.

A process is defined as a set of *rules*. Figure 1b shows the definition of process gdc. It shows that the process has four rules. Each rule has two parts: a *condition* and a *body*. The condition is used to select one rule from multiple choices at runtime. This mechanism enables the programmer to define conditional behaviors. The body defines the rule's actual performance when the rule is selected. Each part is represented as a graph. Figure 1c shows one rule of gcd. The condition graph is located at the top and the body graph is located below it. The body graph consists of data, operators, and *sub-processes*, which are processes used in the body graph. Cones located above and below the rule represent arguments.

The execution of a program consists of parallel sequences of applications of *executable* operators and processes. An operator or process is executable when all of its arguments are available. Execution continues until there is no executable operator or process.

In 3D-PP, the programmer edits programs by manipulating three-dimensional graphs on the screen directly using pointing devices[6].

3. Animated Execution

To animate the execution of a program to support debugging, it is necessary to provide the following three items:

1. A visual representation of the execution state for each step of the execution to visualize data.
2. A visual representation of the step-by-step state transition as the execution proceeds systematically to visualize control flow.
3. Functions for controlling the above visualization to allow browsing.

The next two sections describe how animated execution achieves these three points.

3.1. Visual representation of execution state and state transition

This section describes how animated execution represents both the execution state and step-by-step state transitions using an example. Figure 2 shows snapshots of the animated execution of the program in Figure 1a.

The animated execution shows the application of an operator by moving the operator and its arguments closer and closer together, smoothly. The outputs from the operator are displayed by replacing the operator and its arguments with the output data. Figure 2a shows that operator + is selected for application; process gcd is not executable, since one of its arguments is not available in Figure 1a. Two integers, 10 and 25, are being moved toward the operator. In Figure 2c, the integers and operator + are replaced with the integer 35. Now, process gdc becomes executable.

The application of a process is animated by expanding the process to show all the rules of the process and to highlight the selected rule. The arguments of the rule are then connected to real data and the other rules vanish. Figure 2c is a snapshot taken after expanding process gcd. Animated execution renders wireframes of process gcd to show the four rules of gcd within the wireframes. A selected rule then blinks and enlarges until it is the same size as the original process, as Figure 2d shows. This figure shows the state after one rule has been enlarged and the others vanish. This figure also shows that the integers 30 and 35, which were connected to the argument of the rule, are now being reconnected to the body graph of the selected rule. The result is Figure 2e.

The animation continues to show the application of operator mod, as shown in Figure 2f, and another application of process gcd, as shown in Figure 2g in a similar way.

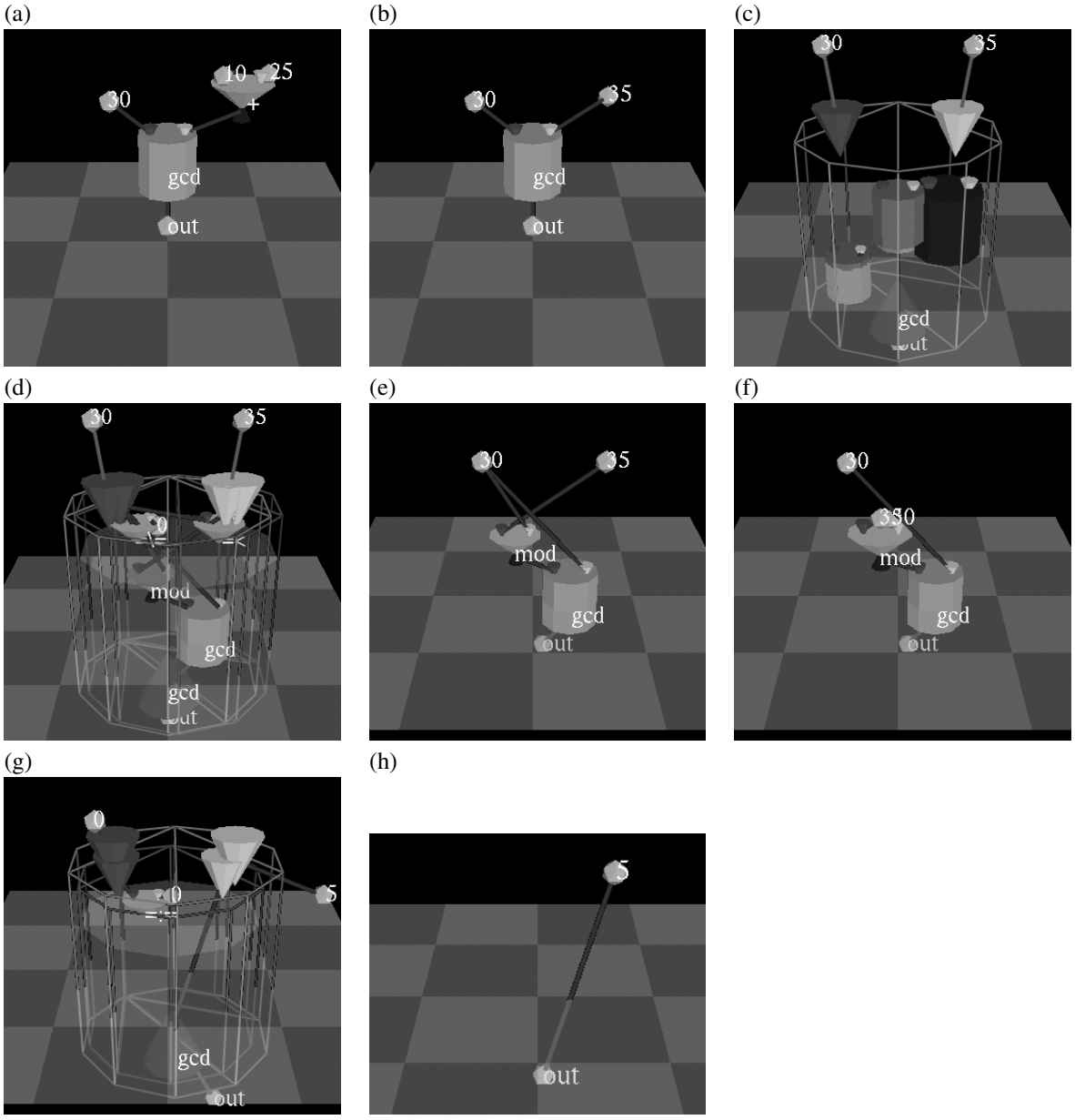

Figure 2. Snapshots of animated execution

Finally, computation of the graph shown in Figure 2g produces the graph in Figure 2h. This is the result of this execution. The graph is composed of variable out and integer 5. It indicates that the variable is assigned the value 5. Now, there is no executable operator or process, so the animated execution stops.

3.2. Functions for controlling animation

To enable the programmer to control animated execution easily, to browse the entire animation, or to examine a suspicious portion of the animation in detail, animated execution includes the following functions.

Suspend. Animation can be stopped whenever the programmer wants. The animation can be resumed from where it stopped.

Rewind. The programmer can undo the animation from any point in the animation. It is possible to rewind the animation after the execution has terminated.

Replay. The programmer can redo the above undo operation. Combining rewinding and replaying enables

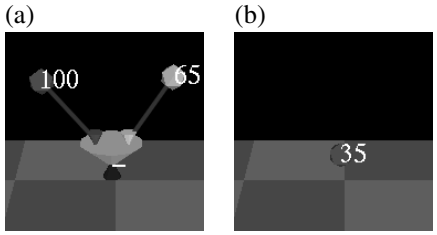

Figure 3. Marking data using normal propagation

the programmer to examine the execution in detail by playing the animation back and forth.

Change speed. It is possible to change the frame rate of the animation.

Change view. When suspended, the programmer can change viewpoints by panning, zooming, and rotating. It is also possible to move objects, such as data and processes, to examine objects occluded by other objects.

4. Functions for Debugging

To support the programmer during test processes and to reduce the number of suspected buggy processes, three debugging functions are available with animated execution: rewriting to speed up testing and modifying processes; marking data for tracing data; and marking rules for narrowing the number of suspicious implementations.

4.1. Rewriting data and rules

While the animated execution is suspended, the programmer can edit the state at runtime and the program itself, including the value of data and the connection edges of rules. When the animation is resumed after editing, the programmer observes the animated execution derived using the edited data and programs. In addition, the edited values or programs are recovered when the animation is rewound.

This mechanism, in combination with animated execution, is a powerful tool for testing and debugging processes quickly. Suppose that some suspicious behavior of a program is caused by a buggy implementation of a process in the program. When the programmer sees the suspicious behavior during its animated execution, the programmer stops the animation and rewinds to the point where the behavior begins. Note that the point can be found easily by playing the animation back and forth. After the programmer finds the buggy process, the programmer rewrites the rules and restarts the animation from where it was suspended. Now, the programmer can observe whether the modification is correct. Moreover, the programmer can test the modification further by using different input values by rewriting the data and restarting the animation.

Modifying programs using this mechanism does not modify the original program. The programmer must explicitly commit the modification to the original program when necessary. Therefore, this mechanism allows the programmer to perform "trial-and-error" testing and debugging safely and easily.

4.2. Marking data

The programmer can mark arbitrary data while the animated execution is suspended. The marked data are colored differently. Moreover, the marks are propagated by operators. There are two types of propagation: normal and reverse. In normal propagation, if marked data are assigned as the input arguments of an operator, the result of the operator is also marked in the animated execution. In reverse propagation, if marked data are the result of an operator, the input arguments of the operator are marked while rewinding the animated execution of the program.

Marking data using normal propagation is used to trace data along a timeline. One example is to highlight an integer that is used as a counter.

Figure 3 shows an example of marking data using normal propagation. In Figure 3a, one of the two arguments of a subtraction operator, the integer 100, is marked (in red on the screen). The other argument, the integer 65, is not marked, and is displayed in the usual color. The result of this marking is intuitive; integer 35, the product of the subtraction, is marked in red in Figure 3b.

Marking data using reverse propagation is used to trace suspicious data back along the timeline in reverse in order to discover where and how the data were produced.

Figure 4 shows an example. Suppose that integers 2400 and 130 were produced as the result of the execution shown in Figure 4a. When the programmer marks integer 130 (in blue on the screen) and rewinds the animated execution, the programmer will observe the snapshots shown in Figure 4b and then in Figure 4c. This clearly shows that the integer 30, located leftmost in Figure 4c, does not affect the calculation of the first marked integer 130.

Note that reverse propagation should be activated in isolation, since it conflicts with other functions, such as rewriting data and marking using normal propagation.

4.3. Marking rules

The programmer can mark arbitrary rules for processes at any time. Marked data are colored differently (in red on

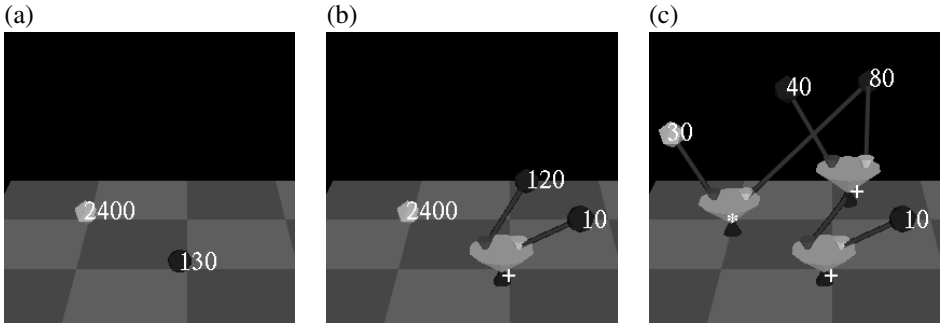

Figure 4. Marking data using reverse propagation

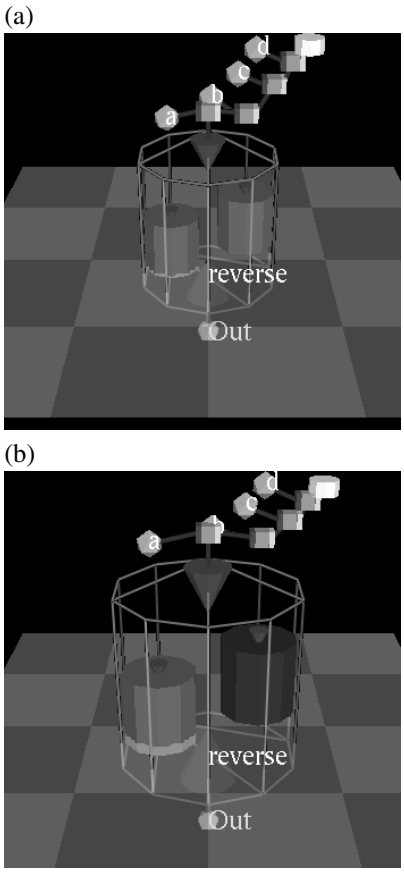

Figure 5. Marking rules

the screen) from other rules. During the animated execution, the marked rules appear in the same color. Therefore, the programmer can concentrate on checking unmarked rules.

Figure 5 shows a marked rule as an example. reverse is defined using two rules. The right one is marked.

This mechanism is used to distinguish suspicious rules from trusted rules. First, the programmer marks rules that are believed to be correct. Once the programmer has confirmed that a rule performs correctly, by rewinding the animation or by testing, then the rule should be marked. When all the rules of a process have been marked, then the process can be trusted.

5. Debugging Strategy

This section outlines the standard strategy used to debug a program with animated execution using the above functions. First, the programmer starts the program and observes the animated execution. There are three types of buggy behavior:

1. The execution terminates normally, but the results are incorrect. This necessitates searching for the point where the incorrect data were produced. A combination of marking the incorrect data with reverse propagation and rewinding the animation will help in this search.
2. The execution terminates due to a fault. The fault occurred for one of the following two reasons:
   (a) There was a buggy process.
   (b) Incorrect input fed into a (correct) process.

   Rewinding the animation slightly will identify the reason. In case (a), modify the rules of the process. In case (b), search for the location where the wrong input was produced.
3. The execution does not terminate; it might fall into an infinite loop or become deadlocked. Stop the animation. Rewind and replay the animation to browse for suspicious behavior, concentrating on checking processes that have unmarked rules. Try to test the behavior of the rules while browsing. If you are certain that a rule is implemented correctly, mark the rule.

Before modifying a rule, unmark the rule that is to be modified. After modification, test the modified process using various sets of arguments by rewriting data.

6. Related Works

Some two-dimensional visual programming systems support the animation of execution. Examples are Pictorial Janus[4, 5], Visulan[14], VIPR[1], and KLIEG[12, 10]. Of these systems, the first was Pictorial Janus[9]. It is a two-dimensional visual programming system that visualizes a concurrent logic programming language called Janus. Its animation is similar to a recorded video. That is, the programmer cannot alter runtime states by changing values or changing the program. Even when the programmer wants to test components using different input values, the entire animation must be re-created after modifying the program.

There are also several animation systems available for three-dimensional visual programming, such as Toontalk[3], 3D-Visulan[13], and SAM[2]. SAM, the newest of these systems, is a synchronous, parallel, state-oriented, general-purpose programming language. A SAM program is composed of a message, agents with ports, and a rule with a precondition and an action sequence. On executing SAM, one rule is selected according to the conditions and its action is executed. SAM animates processes in a manner similar to our animated execution. It animates rule selection by making the rule larger and moving arguments into the action. However, the execution of SAM does not support functions such as rewriting data during suspension, rewinding the animation, or marking with propagation.

7. Conclusions

To support programmers debugging visual programs, this paper proposes animated execution. Animated execution animates state transitions while execution of the program proceeds. The animation is displayed in a manner similar to the program, so that the programmer can interpret the animation easily. In addition, this paper describes three functions that should be used in combination with animated execution: rewriting, to speed up testing and modifying processes; marking data, to trace data; and marking rules, to narrow the number of suspicious implementations. We have implemented animated execution and its functions using 3D-PP, our own three-dimensional visual programming system.

References

[1] W. Citrin and C. Santiago. Incorporating Fisheying into a Visual Programming Environment. In *Proceedings of 1996 IEEE Symposium on Visual Languages*, pages 20–27. IEEE Computer Society Press, Sept. 1996.

[2] C. Geiger, W. Mueller, and W. Rosenbach. SAM - An Animated 3D Programming Language. In *Proceedings of 1998 IEEE Symposium on Visual Languages*, pages 228–235. IEEE Computer Society Press, Sept. 1998.

[3] K. Kahn. Programming by example: generalizing by removing detail. *Communications of the ACM*, 43(3):104–106, Mar. 2000.

[4] K. M. Kahn. Concurrent Constraint Programs to Parse and Animate Pictures of Concurrent Constraint Programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 943–950. ICOT, June 1992.

[5] K. M. Kahn and V. A. Saraswat. Complete Visualizations of Concurrent Programs and their Executions. In *Proceedings of 1990 IEEE Workshop on Visual Languages*, pages 7–15. IEEE Computer Society Press, Oct. 1990.

[6] H. Mitsunobu, T. Oshiba, and J. Tanaka. Claymore: Augmented direct manipulation of three-dimensional objects. In *Proceedings of Asia Pacific Computer Human Interaction 1998 (APCHI'98)*, pages 210–216. IEEE Computer Society Press, July 1998.

[7] T. Oshiba and J. Tanaka. "3D-PP": Three-dimensional visual programming system. In *Proceedings of 1999 IEEE Symposium on Visual Languages (VL'99)*, pages 189–190. IEEE Computer Society Press, Sept. 1999.

[8] T. Oshiba and J. Tanaka. "3D-PP": Visual programming system with three-dimensional representation. In *Proceedings of International Symposium on Future Software Technology (ISFST'99)*, pages 61–66, Oct. 1999.

[9] V. A. Saraswat, K. Kahn, and J. Levy. Janus: a step towards distributed constraint programming. In *Proceedings of the 1990 North American conference on Logic programming*, pages 431–446. MIT Press, Oct. 1990.

[10] B. Shizuki, M. Toyoda, E. Shibayama, and S. Takahashi. Smart Browsing among Multiple Aspects of Data-Flow Visual Program Execution, Using Visual Patterns and Multi-Focus Fisheye Views. *Journal of Visual Languages and Computing*, 11(5):529–548, Oct. 2000.

[11] J. Tanaka. Visual Programming System for Parallel Logic Languages. In *The NSF/ICOT Workshop on Parallel Logic Programming and its Program Environments*, pages 175–186. the University of Oregon, Mar. 1994.

[12] M. Toyoda, B. Shizuki, S. Takahashi, S. Matsuoka, and E. Shibayama. Supporting Design Patterns in a Visual Parallel Data-flow Programming Environment. In *Proceedings of 1997 IEEE Symposium on Visual Languages*, pages 76–83. IEEE Computer Society Press, Sept. 1997.

[13] K. Yamamoto. 3D-Visulan: A 3D Programming Language for 3D Applications. In *Proceedings of Pacific Workshop on Distributed Multimedia Systems*, pages 199–206, Hong Kong, June 1996. The Hong Kong University of Science and Technology.

[14] K. Yamamoto. Visulan: A visual programming language for self-changing bitmap. In *Proceedings of International Conference on Visual Information Systems*, pages 88–96, Melbourne, Australia, Feb. 1996. Victoria University of Technology (in cooperation with IEEE).