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 support the programmer to debug a visual program, this paper proposes animated execution, and supporting functions that should be used in combination with animated execution. Animated execution animates state transitions while execution of the program proceeds. To make the animation easily recognizable for the programmer, the animation is displayed in a similar manner with the visual program. The functions are used for narrowing possible locations of bugs and for testing fixed components quickly. We have implemented animated execution and the functions on 3D-PP, our three-dimensional visual programming system.

1. Introduction

In contrast to casual text based programming, which represent programs with text, visual programming represent programs with graphs. This graphical representation is quite good at in explicitly showing relationship between components and data dependencies.

However, representing execution of visual programs for debugging is still waiting for some consideration. At first, debugging a program usually requires the following three functions:

Monitoring data at runtime. Monitoring data to compare input data of a component with corresponding outputs is a basis in checking correctness of the component. This is also used to narrow possible locations of a bug into one or some components.

Monitoring control flow of the execution. Monitoring control flow is used to recognize that there is buggy behavior. This also helps the programmer guess the type of the buggy behavior. For example, if the execution terminates very quickly than the programmer expects, she/he can guess that the termination is caused by some faults 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 to perform the test easily.

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

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

We have implemented animated execution and the additional functions on 3D-PP[7, 8], the three-dimensional exetension of our two-dimensional visual programming system PP[10].

2. 3D-PP

A program of 3D-PP is a hierarchical three-dimentional graph composed of nodes and edges. Nodes correspond to *data*, *operators*, and *processes*. The shape of a node represents its type; spheres represent data, such as integers and

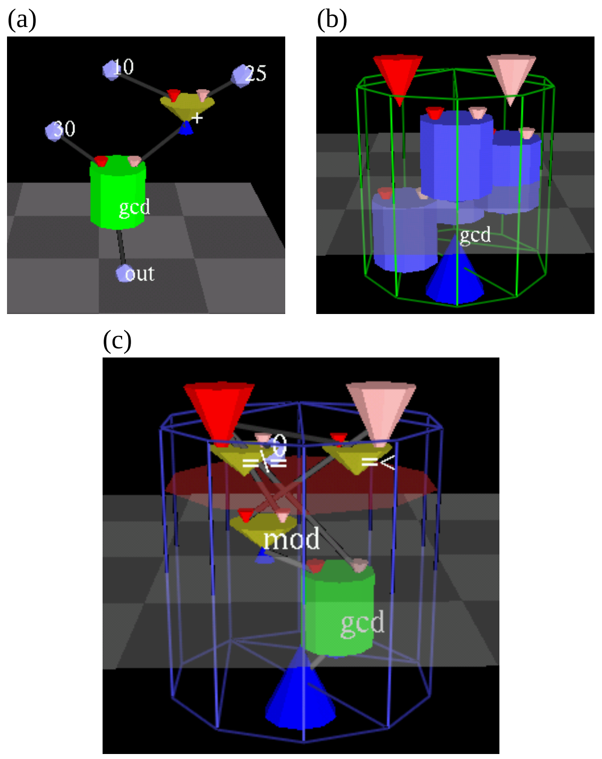

Figure 1. Programming constructs in 3D-PP

strings, or variables; upside-down cones represent operators such as addition, modulo, and comparison; pillars represent processes. An edge correspondts data dependency. Figure 1a is 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 will be 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. A rule consists of two parts, a *condition* and a *body*. The condition part is used to select one rule from multiple choices at runtime. This mechanism enables the programmer to define conditonal behaviors. The body part 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 upper side. The body graph is located lower side. The body graph is consists of data, operators, and *sub-processes* that is a process used in the body graph. Cones located at both of the upper and lower side of the rule represent arguments.

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

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

3. Animated Execution

To animate an execution of a program to support debugging, providing the following three points is necessary:

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

The next two sections describes how animated execution achieves the above three points.

3.1. Visual representation of execution state and state transition

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

Animated execution shows application of an operator by getting the operator and its arugments closer and closer, 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 being applied; process gcd is not executable since one of its arguments is not available in Figure 1a. Two integer, 10 and 25, are being moved toward the operator. In Figure 2c, the integers and operator + are replaced with integer 35. Now, process gdc becomes executable.

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 the real data. Other rules are vanished. Figure 2c is the snapshots after expanding process gcd. Animated execution renders the wireframes of process gcd to show the four rules of gcd within the wireframes. A selected rule is then blinked and enlarged until it has the same size as the original process as Figure 2d shows. This figure shows the state after one rule is enlarged and others are vanished. This figure also shows that two integer 30 and 35 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.

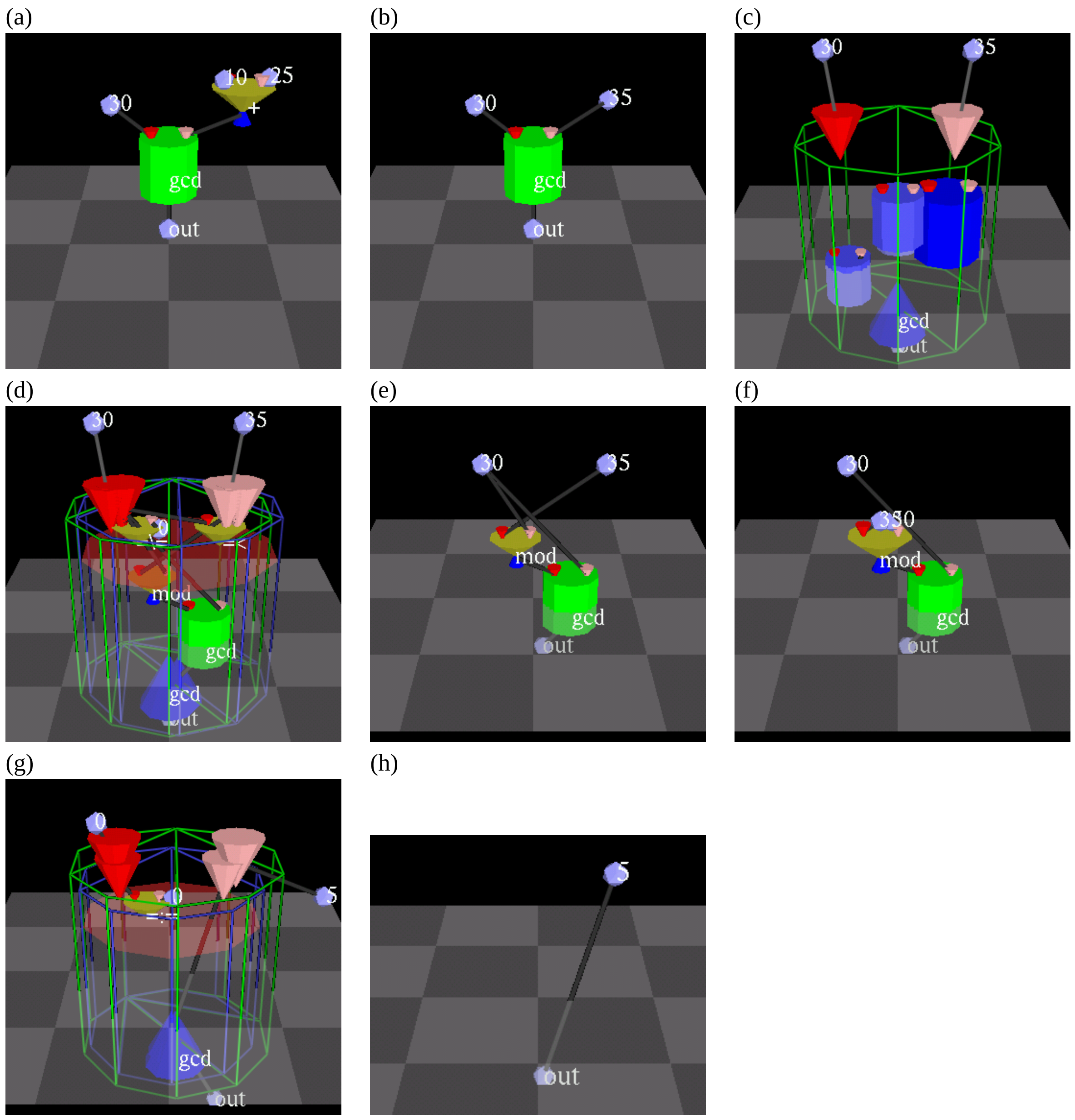

Figure 2. Snapshots of animated execution

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

3.2. Functions for controlling animation

To enable the programmer to control animated execution easily for browsing entire animation and for examining suspicious portion of the animation in detail, animated execution provides the following functions.

Suspension. Animation can be stopped whenever the programmer wants. The animation can be resumed from the stopped point.

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

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

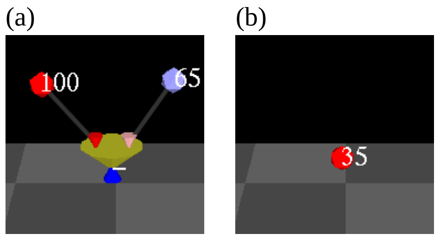

Figure 3. Marking data with normal propagation

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

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

Changing view. During suspension, the programmer can change viewpoints by panning, zooming, and rotating. It is also possible to move the location of objects such as data and processes to examine occluded objects by others.

4. Functions for Debugging

To support the programmer to test processes and to narrow possible buggy processes, three kinds of functions for debugging are available along with animated execution: rewriting for speeding up testing and modification of processes, marking data for tracing data, and marking rules for narrowing suspcious implementation.

4.1. Rewriting data and rules

While the animated execution is suspended, the programmer can edit the state at runtime and the program itself, such as values of data and connection of edges of rules. When the animation is resumed after editing, the programmer observes the animated execution that is derived from the editted data and programs. In addition, the edited values or programs are recovered when the animation is rewinded.

This mechanism, in combination with animated execution, is a powerful tool to test and debug a process quickly. Suppose that some suspicious behavior of a program is caused by a buggy implementation of a process of the program. When the programmer sees the suspicious behavior during its animated execution, the programmer stops the animation and rewinds it back to the point where the behavior begins. Note that finding the point is easy by playing the animation back and forth. After the programmer finds the buggy process, the programmer rewrites rules and restarts the animation from the suspended point. Now, the programmer can observe whether the modification is correct or not. Moreover, the programmer can test the modification further by giving some different input values to the process by rewriting data and then restarting the animation.

The modification of programs with this mechanism does not modify the original program. The programmer must explicitly commits the modification into the original program when necessary. Therefore, this mechanism provides the programmer with safe and easy “try-and-error” testing and debugging.

4.2. Marking data

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

Marking data with normal propagation is used in tracing data along by timeline. An example usage is to highlight an integer which is used as a counter.

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

Marking data with reverse propagation is used in tracing back suspicious data along the timeline in reverse order for finding out where/how the data were produced.

Figure 4 shows an example. Suppose that two integer 2400 and 130 were produced as the result of an 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 snapshot of Figure 4b and finally the snapshot of Figure 4c. Now, this figure clearly shows integer 30, located leftmost in Figure 4c, does not affect the calculation of the firstly marked integer 130.

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

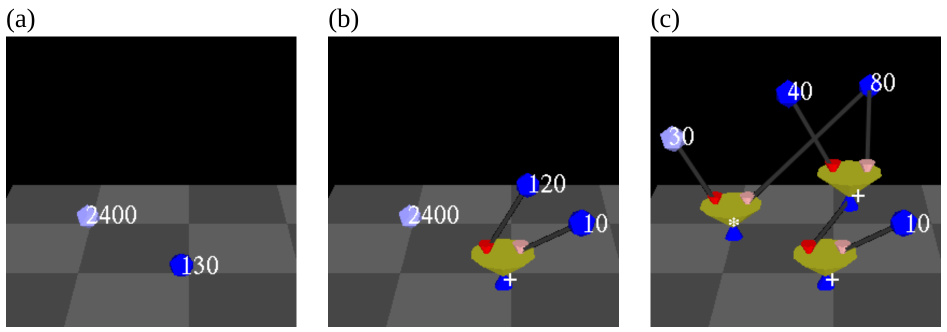

Figure 4. Marking data with reverse propagation

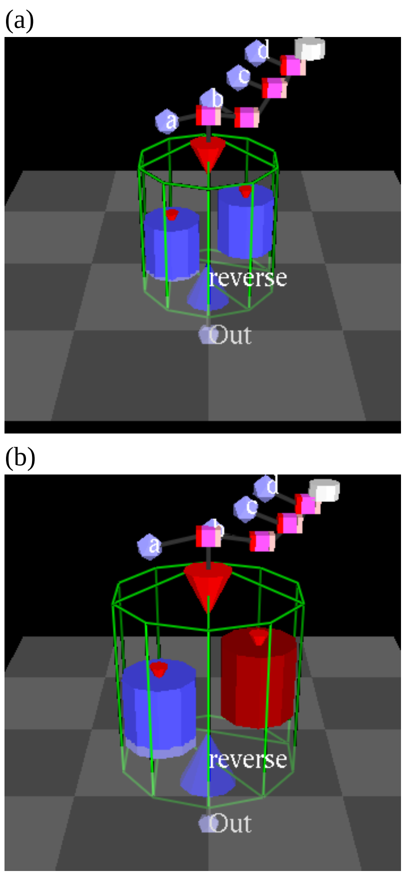

Figure 5. Marking rules

4.3. Marking rules

The programmer can mark arbitrary rules of processes at any time. Marked data are differently colored (in red on the screen) with other rules. During the animated execution, the marked rules appears 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 with two rules. The right one is marked.

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

5. Debugging Strategy

This section outlines the standard strategy of debugging of a program using the above functions with animated execution. Firstly, starts and observes the animated execution of the program. There are three types of buggy behavior:

1. The execution terminates normally but the results are incorrect. Search the point where the incorrect data were produced. Combination of marking the incorrect data with reverse propagation and rewinding the animation will help this search.

2. The execution terminates due to a fault. The fault was caused by one of the following two kinds of reason:

   (a) A buggy process directly.

   (b) A wrong input fed into a (correct) process.

   Rewinding the animation a little bit will determine which was the reason. In the case of (a), modify the rules of the process. In the case of (b), search the location where the wrong input was produced.

3. The execution does not terminate, possibly falling in infinite loop or in deadlock. Stop the animation. Rewind and replay the animation to browse a suspicious behavior, concentrating on checking processes that have unmarked rules. Try to test the behavior of the rules during the browse. If you can assure that a rule is implemented correctly, mark the rule.

Before modifying a rule, unmark the modified rules. After modification, test the modified process by giving various sets of arguments by rewriting data.

6. Related Works

There are some two-dimentional visual programming systems that support animation of execution. Examples are Pictorial Janus[4, 5], Visulan[13], VIPR[1], and KLIEG[11, 9]. Among these systems, the most pieoneering one is Pictorial Janus. Pictorial Janus is a two-dimensional visual programming system which visualizes a concurrent logic programming language called Junus. However, its animation is similar to a recorded video. That is, the programmer cannot change runtime states by changing values and changing the program. Even when the programmer wants to test some components by giving different input values, the entire animation must be re-created after modification of the program.

There are also several systems that provide in the field of three-dimentional visual programming, such as Toontalk[3], 3D-Visulan[12], and SAM[2]. SAM, the most recent one of the above systems, is a synchronous parallel, state-oriented, general purpose programming language. A SAM program is composed of message, agents with ports, and rule with a precondition and a sequence of action. At the execution of SAM, one rule is selected by condition and its action is executed. SAM animates produces in a similar way as our animated execution. It animates selection of rules by growing it and passing arguments into the action by moving them. However, execution of SAM does not supports such functions as rewriting data during suspension, rewinding the animation, and marking with propagation.

7. Conclusions

To support the programmer to debug a visual program, this paper proposes animated execution. Animated execution animates state transitions while execution of the program proceeds. The animation is displayed in a similar manner with the programs, thus the programmer can easily read the animation. Moreover, this paper describes three kinds of functions that should be used in combination with animated execution: rewriting for speeding up testing and modification of processes, marking data for tracing data, and marking rules for narrowing suspicious implementation. We have implemented animated execution and the functions on 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] 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.

[10] 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, 1994.

[11] 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.

[12] 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.

[13] 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).