Algorithm Visualization System : An Overview of Internal Structure [1]

Somchai Prasitjutrakul
Dept. of Computer Engineering
Faculty of Engineering
Chulalongkorn University
Bangkok, THAILAND
email: somchaip@chulkn.chula.ac.th

Wittaya Watcharawittaykul
Division of Computer Science
School of Applied Statistics
National Institute of Development Administration
Bangkok, THAILAND
email: wittaya@as.nida.ac.th

Abstract

Algorithm visualization is a research area that explores presentation techniques for visualizing algorithm behavior. Complementing asymptotic analysis, algorithm visualization presents one with abstract behavioral models in terms of animated views, which helps viewers see behavioral aspects and leads to more understanding at a higher abstract level. This paper presents an internal structure of an algorithm visualization system under Microsoft Windows where multiple execution threads can be synchronized and behavioral view can be displayed simultaneously. The design aims to be structured, systematic, and easy to use.

I. Introduction

Algorithm visualization is a research area investigating presentation techniques for visualizing algorithm behavior. Interesting aspects relating to algorithm behavior involve dynamic aspects of data, program executions, and other computational operations. Direct measurement of these aspects usually provides absolute numerical values that do not reflect the algorithm's behavior. A better approach is to exploit the asymptotic analysis to analyze the algorithm complexity. Although this approach is currently widely used, it needs strong mathematical background and experience. Appreciating algorithm behavior is usually difficult because of the inability to see through mathematical functions.

Complementing the asymptotic analysis, algorithm visualization presents one with abstract behavioral models in terms of animated views. Each view represents an interesting aspects of the behavior. Because interesting aspects can be separated and shown simultaneously, overall understanding of the algorithm behavior can be achieved easily. One can try changing algorithm inputs, controlling algorithm and animation speed, and then observing the results. Moreover, one can even try visualizing more than one algorithm solving the same problem instance in order to compare algorithm performance.

This paper presents an internal structure of an algorithm visualization system under Microsoft Windows where multiple execution threads can be synchronized and behavioral view can be displayed simultaneously. The design aims to be structured, systematic, and easy to use. The rest of the paper is as follows. Section 2 presents the related work to algorithm visualization. A background on Microsoft Windows features is outlined in section 3. Section 4 presents algorithm visualization system components and how they interact. A brief demonstration is given in section 5. And then the paper concludes in section 6.

II. Related Work

The idea of having tools to help one understands how an algorithm works is dated back in the mid '70s. The first approach is to record the computer-generated images on film, for instances, Bell Labs' film on list processing [1], Booth's film on the associated algorithms of the PQ-trees [2], and Beacker's Sorting Out Sorting [3] which takes three years to make the 30-minute film. Algorithm movies show the film in the exact form where it was produced, so viewers can not interact or experiments with the algorithm they are watching.

The second approach is to have a system with the ability to graphically display internal program's data structures. Being able to observe what data are being kept and how they change helps viewer appreciates certain aspects of the algorithm. This feature is generally used for debugging programs where programmer want to know when the data are invalid. Incense [4], GDBX [5], PROVIDE [6] are examples of systems supporting this feature.

Although graphical display of data structures can reveal what data look like to the viewers, they do not show how data structures in a program related and do not exhibit program execution status. Algorithm animation system is another approach . It allows programmers to

[1]This research is supported by the National Electronics and Computer Technology Center.
Reprinted from Proc. of Third ASEAN Regional Seminar on Microelectronics & Information Technology, Aug. 1994

insert special functions to be called when certain aspects of program are worth observing. Viewers can change program input parameters, select appropriate views of data or programs, and then observe the results. This kind of systems includes Yarwood's [7], De Boer's [8], BALSA-I [9], Zeus [10], Tango [11], Pavene [12]

III. Windows™

Microsoft Windows is an operating environment running on top of DOS in IBM-compatible personal computers. The environment is currently the most popular for the PCs, this is because of consistent graphical user interface, device-independent graphics, multitasking capability, and the use of dynamic linking library.

Windows programs are event-driven. When an event occurs (e.g., mouse click, mouse move, timer tick) an associated message is generated. Associated with each application is an application message queue that keeps pending messages waiting to be processed by the application. That is each application consists of a main function and a set of functions corresponding to associated messages. When a message arrives, it gets dispatched to an appropriate function. The function reacts to the message it got, and then goes back to wait for other messages. Since Windows is a non-preemptive multitasking system, computationally-intensive processes must yield control to other applications from time to time in order not to hold CPU for its own [13].

IV. Algorithm Visualization System

The main logic structure of the system consists of an algorithm module that accepts input from an input generator module and then produces image and animation corresponding to the program action and behavior to the view module (see Fig 1). In case that more than one algorithm is visualized (i.e., for performance comparisons among algorithms which are solving the same problem instance), the executions must be synchronized by the synchronizer. The synchronizer must be fair so that the comparison in terms of time is relatively correct.

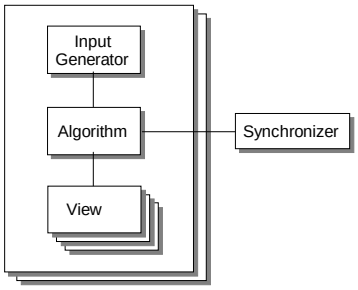


Fig. 1. Algorithm Visualization System Structure

Converter

From the structure presented above, the algorithm and the view are too closely tightened. That is the algorithm designer has to know exactly what kind of view they currently are. This makes additions or modification of views harder, since we need to modify the algorithm as well. To remedy this problem and clarify the responsibilities the algorithm designer and the view designer, we introduce an converter (see Fig. 2). The converter converts event messages from the algorithms to a sequence of commands each associated view reacts. Notice that there may be more than one view that accept the same commands e.g., a view whose purpose is to display value of a number can be either displaying the number itself, plotting a curve of a function of this number and time, or representing a value by an intensity of color.

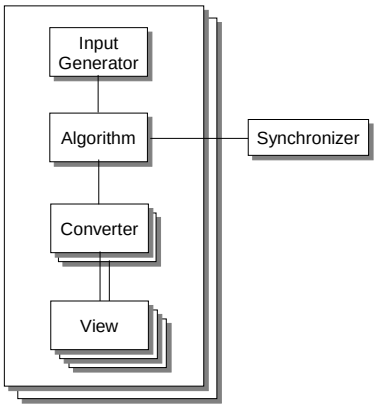


Fig. 2. Converter

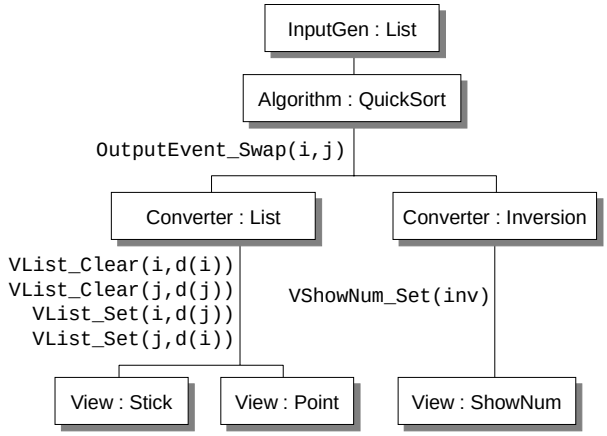


Fig. 3. Converting an output event to view's commands

Fig. 3. shows an example of converting an output event to view's commands. In this example, a user want to visualize Quicksort by observing the data contents through Stick and Point view and observing the number of inversion through ShowNum view. When there is any data exchange by the algorithm, it sends out an output event. Converters detect the event and then send out a set of view's commands to the associated views. So we can see that the algorithm designer only pays attention to

what interesting events of the algorithm are. And the view designer only pays attention to the details of how picture and animation can be done according to a certain set of specified view's commands. And it is the responsibility of the converter to glue both part together. According to this scheme, a view can be used by different algorithms for different purposes.

Binder

To further separate each module apart, we indirectly tighten each module via a binder (Fig. 4). The information about which algorithm, input generator, converters, and views the user choose are all stored in the binder local binding tree (an example is shown in Fig. 5). All messages generated will be sent to binder and will get rerouted to the associated modules. Therefore each module only waits for command messages, reacts on the commands, and sends command messages out (if any) and does not have to worry about where the command message it got is from and where the command message it sends out is to. The binding tree is built and updated according to selection choices chosen from the user via the control center.

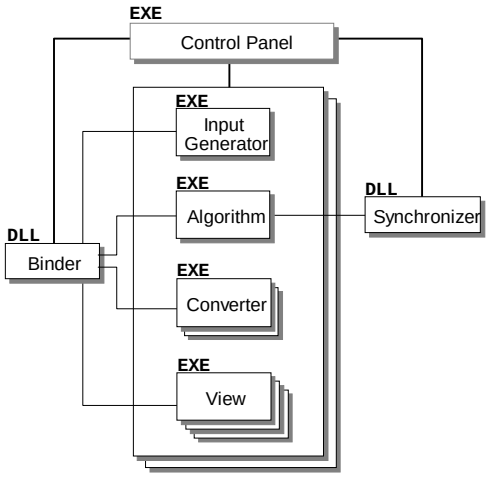
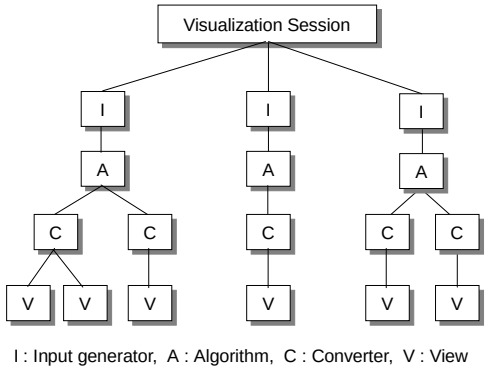


Fig. 4. Binder



I : Input generator, A : Algorithm, C : Converter, V : View

Fig 5. Binding Tree

How does the relationship among modules in the binding tree come from ? This relationship is specified by the algorithm designer in a form of configuration file. An example of the configuration file for a sorting program is shown below. The file lists all modules which can be selected for the algorithm. Here, there is one input generator and four views available. Each view consists of the name of executable files for the view and its associated converter. By default, pre-selected modules specified in the `[default]` section are loaded upon selecting the algorithm.

```
[directories]
; search path for each components
inputgen=c:\avis\inputgen
view=c:\avis\view
converter=c:\avis\sort\convertr

[inputgens]
1=List, glist.exe

[views]
1=Point, clist.exe, vdot.exe
2=Stick, clist.exe, vstick.exe
3=Inversion, cinverse.exe, vshownum.exe
4=#comparisons, cnumcmp.exe,
vshownum.exe

[default]
inputgen=List, glist.exe
view=Point, clist.exe, vdot.exe
view=Stick, clist.exe, vstick.exe
```

Parallel Call Mechanism

Multiple views can be shown for visualizing an algorithm. Therefore updating views needs to be done simultaneously (at least to the user's perception). If the binder sends out commands to each view sequentially, the views will be jerkily updated. To achieve smooth view update, the system provides a parallel call mechanism. It allows the binder to send out command in parallel to associated modules, then waits for all the actions to be completed before returns control to the caller. The following code illustrates how the parallel call is done under Windows.

```
static int   count;
PSendCommand( Alg, Command  )
{
 count = 0;
 for each convertr Cnv assoc w/algorithm Alg {
   PostMessage( Cnv, Command );
   count = count + 1;
 }
 while ( count != 0 ) YieldControl();
}
PRespond()
{
  count = count - 1;
}
```

Suppose an algorithm wants to send out command to its associated converters, it calls the `PSendCommand` function when the function returns, every converter must complete its job. The PSendCommand actually sends out command via the `PostMessage` Windows function, which puts the command message to the target converters' application queues. Notice that we have a variable named `count` that increases its value by one for each command posted to a converter, and decreases its value by one when any converter finishes its job (when a

converter finishes the command received, it must call the `PRespond` function). Therefore, right after posting the command to all the associated converter, the `PSendCommand` waits for value of `count` to reach zero. and then returns control to the caller algorithm. During the waiting loop we need to call `YieldControl` function to let other processes of the system run since Windows is non-preemptive.

Synchronizer

Since the system allows multiple algorithm visualizations, it must provide an algorithm synchronization mechanism. This is to guarantee that algorithms running at the same time each has a share equal opportunity to execute in the environment. Relative time spent by algorithms must reflect algorithm speed. That an algorithm finishes earlier than the others means it is faster. This job is taken care by the synchronizer with helps from the algorithm designer. Before writing a program to implement an algorithm, the designer needs to define what basic operation of the algorithm is. The basic operation generally is an operation of the algorithm whose operation count reflects execution time. For example, a basic operation for a sorting algorithm would be the comparison operation. When the basic operation is defined, whenever the program executes any basic operation, it must call `SyncNotify` function inform the synchronizer that it passed one more basic operation as shown below.

```
Alg_SelectionSort_Code( hAlg )
{
   int   i, n, iMaxVal, iMaxPos, iData[MAXN];

   n = InputEvent_HowManyKeys();
   for (i=0; i<n; i++)
     InputEvent_ReadKey( iData[i] );
   for (i=0; i<n; i++)
     OutputEvent_NewKey( i,iData[i] );
   for (i=n-1; i>=1 i--) {
     iMaxPos = i;
     for (j=0; j<=i; j++) {
       if ( iData[iMaxPos] < iData[j] )
          iMaxPos = j;
       SyncNotify( hAlg );
     }
     swap( iData, i, iMaxPos );
     OutputEvent_Swap( i, iMaxPos );
   }
}
```

Each algorithm has a time slot in the synchronizer. The time slots are initialized to a number represent time quantum at the beginning and when all algorithms have executed one more basic operation. A time quantum decreases its value by one if its associated algorithm passed one basic operation. Whenever the quantum of an algorithm reaches zero, the synchronizer holds the execution of the algorithm and yields control to others. The holding will be released if all the slots are zero and time quantums are reloaded and the algorithms resume their work. (See the code below.)

```
SyncNotify( hAlg )
{
  if ( Quantum[hAlg]-- != 0 ) return;
  NumAlgsLeft--;
  while ( NumAlgsLeft != 0 ) YieldControl();
  SyncInit();
}
```

Since the synchronizer acts just like a conductor giving a rhythm to all the algorithms. It, therefore, indirectly controls speed of all the algorithms . This is done by controlling the holding duration. If the holding only releases when user click a certain button on the control panel, it becomes a single-step control. So pausing the execution is achieved by holding the execution forever, and single step is done by releasing the holding and go to the pausing stage in the next round.

V. Examples

To demonstrate how algorithm visualization helps viewers appreciate the insight of algorithms (albeit the fact that views cannot be animated on paper), we choose various sorting algorithms to be presented here in this section since the sorting problem is very well-defined and well-known. A list of four hundred data is to be sorted. A view named `Points` is used as a window presenting the list. Each datum is represented by a point where its y-coordinate is proportional to its numeric value and its x-coordinate is its position in the list. In the example, we assume that initial value of the data are randomly generated and ascending sort order is expected.

Let take a look at Heapsort algorithm in Fig. 6. The principle of Heapsort is to iteratively select a datum whose value is maximum among all the data not yet selected and then put the datum on the right-most position among all the positions not yet having data. The data not yet selected are kept in a Heap, a data structure very well-suited for deleting the maxima. Starting from the initial positions shown in Fig 6a. Time must be spent on building the heap illustrated in Fig. 6b where the heap is partially built. Building heap is completed in Fig. 6c. (Notice the left-most element is always the maximum one.) Then the maximum selection process is iterated as shown in Fig. 6d and 6e. Finally, the sorting is finished in Fig. 6f. We can observe how data are partitioned into a heap on the left side (which gets smaller and smaller) and a sorted data on the right side which gets longer and longer. The interesting thing is if an already sorted list is sorted using Heapsort, the corresponding visualization views are shown in Fig 6. It seems like the Heapsort is wasting its time building a heap which destroys the initial sorted order of data, and then relocates data to form a sorted data. The sequence of view snapshots from start to finish are Fig. 6f, 6g, 6h, 6i, 6j (here the heap is completely built), 6i, 6h, 6g, and Fig. 6f.

VI. Conclusion

This paper describes the internal structure of an algorithm visualization system running under Windows operating environment. The input generator module

generates a problem instance to the algorithm module which in turns produces events. The events are interpreted and converted to a list of commands for the view modules by the converters. All the mentioned modules are glued together via the binding tree and parallel call services offered by the binder module where data, events, and commands are routed to the correct destinations. For a multiple algorithm visualization session, the synchronizer controls the execution of each algorithm. The research is currently exploring the abstract behavioral models for various graph, numerical method, computational geometry, and VLSI design algorithms. A good algorithm visualization system can be a powerful tool for studying algorithm analysis and design.

References

[1] Kenneth C. Knowlton. *L6: Bell Telephone Laboratories Low-Level Linked List Language*, two 16mm black and white sound film, 1966

[2] Kellog S. Booth, *PQ Trees*, 16mm color silent film, 12 minutes, 1975

[3] Ronald M. Baecker and David Sherman, *Sorting Out Sorting*, 16mm color sound film, 30 minutes, 1981

[4] Brad A. Myers, "Incense: A System for Displaying Data Structures," *Computer Grpahics*, Vol 17, No.3, July 1983, 115-125

[5] David B. Baskerville, "Graphic Presentation of Data Structures in the DBX Debugger," *Report No. UCB/CSD 86/260*, University of California at Berkeley, Berkeley, CA, October 1985

[6] Thomas G. Moher, "PROVIDE: a Process Visualization and Debugging Environment," *Technical Report*, University of Illinois at Chicago, Chicago, IL July 1985

[7] Edward Yarwood, *Toward Program Illustration*, M.Sc. Thesis, Dept. of Computer Science, University of Toronto, Toronto, ON, 1974.

[8] James M. De Boer, *A System for the Animation of Micro-PL/I Programs*, M.Sc. Thesis, Dept. of Computer Science, University of Toronto, Toronto, ON, 1974.

[9] Marc H. Brown, *Algorithm Animation*, The MIT Press, Cambridge, MA, 1988.

[10] Marc H. Brown, "Zeus: A System for Algorithm Animation and MultiView Editing," *Proc. IEEE Wokshop Visual Languages*, IEEE CS Press, CA, 1991, pp. 4-9.

[11] J. T. Stasko, "Tango: A Framwork and System for Algorithm Animation," *Computer*, Vol. 23, No. 9, Sept. 1990, pp. 27-39.

[12] G.-C. Roman et al., "Pavene: A System for Declarative Visualization of Concurrent Computations," *J. Visual Languages and Computing*, Vol. 3, No. 2, June 1992, pp. 161-193.

[13] Charles Petzold, *Progamming in Windows 3.1 3rd Edition*, Microsoft Press, 1992

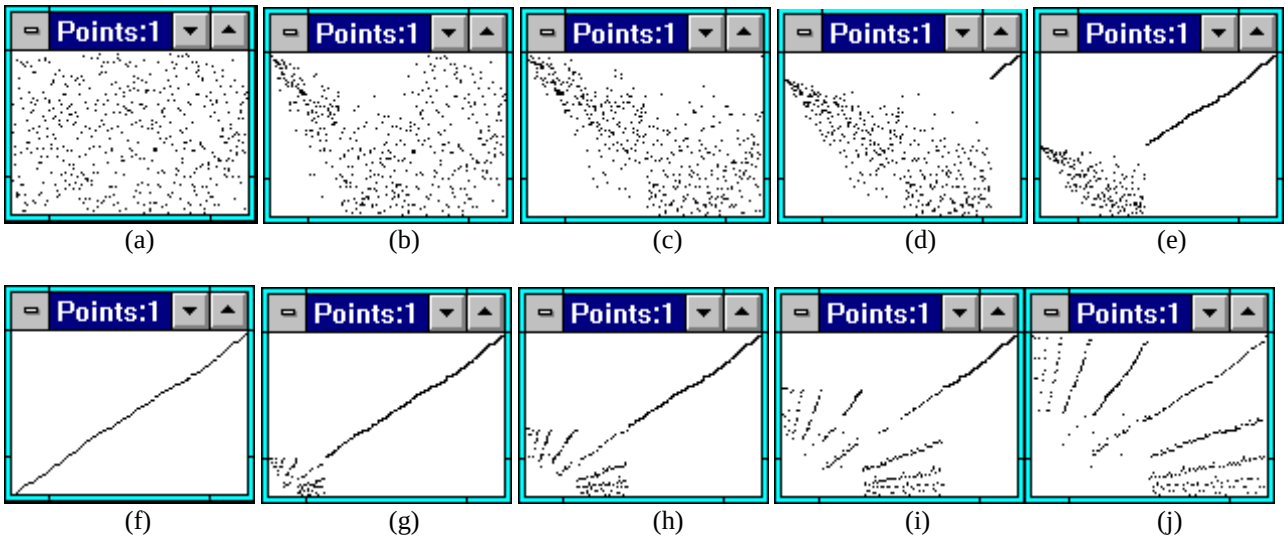


Fig. 6. A visualization of Heapsort algorithm.