

# On the Merits of Low (or No) Visualisation in Understanding Algorithms

Judith Bishop  
Computer Science Department  
University of Pretoria  
Pretoria 002  
South Africa

[jbishop@cs.up.ac.za](mailto:jbishop@cs.up.ac.za)

## Abstract

Many educators regard the visualisation of algorithms as an essential teaching tool for courses from first year up. Sophisticated and visually appealing systems exist to show the operation of algorithms from simple sorts to complex compression algorithms. These tools can take considerable effort to develop or even adopt and may end up being quite restricted in scope. An alternative type of tool is a line-animator which is a very simple program that emits text only, but still manages to show the progress of the algorithm. This paper investigates five such line-animators and explains the advantages of this somewhat neglected genre. These include being able to keep a trace for later perusal, and being able to understand the real working of the algorithm. The paper compares line-animators to graphic assistors (libraries of routines) and standalone visualisers, and concludes that each is useful in a certain context.

## 1. Introduction

It has become almost an axiom that visualisation of algorithms is an aid to understanding in the teaching of computer science. In any computer science education conference over the past ten years, several papers can be found describing tools which show, via animation and graphics, the working of an algorithm or set of algorithms. A sample of these is given by [Diehl, Khuri-1, Koldehofe, Rasala and Rößling]. Some of the tools are stand-alone, intended for one type of algorithm [Koldehofe]; others follow a pattern and can be adapted across a range of concepts from parsing to compression [Khuri-2], and others have developed a rich framework and language for specifying the visualisations [Stasko].

In these papers, one finds statements such as these, which underline the view that visualisation as a teaching tool is here to stay:

“Visualisation of computational models is at the heart of educational software for computer science and related fields.” [Diehl]

“... the use of toolkits in first year computer science is essential. Toolkits will ...enable students to be more effective and will provide a rich set of

examples that support the principles of computer science.” [Rasala]

Nevertheless, some doubt has been cast on the effectiveness of these tools [Kehoe] and only in rare cases does one find a tool written by one group being used and extended by another [Naps]. However, the prevalence of the web and of Java applets has undoubtedly enabled greater take up of these tools worldwide.

The purpose of this paper is not to enter into the debate as to whether visualisation tools are effective or not, nor to point out shortcomings of individual tools. It is rather to raise the awareness of another kind of teaching aid that has been overtaken by visualisation, but which in the author's opinion still has a valid place in the classroom and more so in the laboratory. This is the command line, text only, simple program which illustrates an algorithm or class of similar algorithms. Since there is no accepted term for these programs, we shall call them line-animators.

Section 2 sets the scene by describing a typical line-animator for a fairly advanced set of algorithms, i.e. garbage collection. Section 3 then outlines several of the other line-animators that we have developed over the years. Section 4 develops the properties of line-animators, and Section 5 places them in the context of their more common cousins, the graphical animators and visualisers. In so doing, we endeavour to pinpoint the advantages and disadvantages of each. Section 6 concludes the paper.

## 2. A Line-animator for Garbage Collection

In a course on compilers, students are taught about several methods of garbage collection [Appel]. Garbage collection is a satisfyingly visual activity, since one can see both the before and after state of memory, as well as the stages in between. Different algorithms treat memory completely differently. The most common, called mark-sweep, follows pointers from the stack and creates a free list of available space which can then be reused. The most efficient method actually copies all reachable nodes into a new part of memory, first copying roots then the structures they point to. Variations on these methods, as well as others such as reference counting also exist.

In order to bring home to students both the process and the efficiency of the algorithms, an animator would be ideal. At the time the course was being given (first half of 2001), we could not find one on the web. Time being short, we decided to write our own. The result was the line-animator here and available on <http://www.name.withheld>. The program to support a garbage collector method took three hours to develop and weighs in at 167 lines of Java. It is structured as follows:

1. three array data structures representing a symbol table, stack and heap;

```
class Collector {
    Hashtable table = new
Hashtable(100);
    int stack [] = new int[300];
    char heap [] = new char[2000];
    int sp = 0;
    int hp = 0;
    char dot = '.'; // for null
}
```

2. a method to read in data representing declarations of objects with certain sizes e.g. p 6 and store this information in the symbol table;
3. a method to execute instructions which manipulate the stack and heap, the instructions having the form:

```
p+ allocate p on the stack
qp assign p to q
q.4p assign p to field 4 of q
p- deallocate p
> print out everything
! Garbage collect
```

4. an execute method which interprets the commands:

```
void execute (char left, char right) {
    Objects ptable, qtable;
    int pstack, pheap;
    if (right == '+') {
        // Create stack entry and fill
        // heap with the character
        . . .
    }
    else
    if (right == '-') {
        // mark as null in the stack
        . . .
    }
    else {
        // alter the stack pointer
        pstack = ((Objects)
table.get(new Character
(right))).stackaddr;
        pheap = stack[pstack];
        qtable = (Objects)
table.get (new Character(left));
        stack[qtable.stackaddr] =
(char) pheap;
        table.put(new Character
(qtable.name), qtable);
    }
}
```

```
}
}
```

If we run the Collector program, the line animator will show us the data which forms the program as input, the state of the table, stack and heap as the allocations and deallocations in memory are performed, and the state just before we can call a garbage collector. An example of such a sequence is:

```
p +
q +
r +
s +
The table
s 3 4
r 3 3
q 6 2
p 6 1
The Stack
0 1 2 3 4
0 0 6 12 15
The Heap
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
p p p p p p q q q q q r r r s s s
p q
r -
The table
s 3 4
r 3 3
q 6 2
p 6 1
The Stack
0 1 2 3 4
0 6 6 . 15
The Heap
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
p p p p p p q q q q q r r r s s s
Program complete
```

In the laboratory, the students were asked to write a garbage collector algorithm of their choice and show the result. The pseudo-code in the book (which is very similar in format to any book on algorithms) looked like this for one of the methods (copy-collect-scan):

```
function Forward (p)
if p points to from-space
then if p.f1 points to to-space
then return p.f1
else for each field f1 of p
    next.f1 = p.f1
    p.f1 = next
    next = next + size of record p
return p.f1
else return p
```

It was fairly straightforward for them to translate this code into what was required in the Java program and to get the output of:

```
The table
s 3 4
r 3 3
```

```

q 6 2
p 6 1
The Stack
 0 1 2 3 4
 0 2020 . 26
The Heap
 0 1 2 3 4 5 6 7 8 91011121314151617
 p p p p p p20 q q q q r r r26 s s
202122232425262728293031323334353637
 q q q q q s s s

```

Copy-collect makes a compacted copy of the heap into a new area of the heap, here starting from 20. We can see that q and s are retained and p and r have been excluded from the new heap.

In adding their garbage collectors, students have full control of the memory as represented in the stack and heap data structures. Immediately, various real issues are brought home to them, as they try to program their algorithms in a typed language:

1. In copy-collect, the first phase uses the heap to point back to the uncopied elements in the stack. But the heap is declared of type char. So it was necessary to change the type of the heap to include a "bit" to indicate whether the contents of a word was a pointer or data.
2. In the reference count algorithm, each element of the stack retains a count (of, say, a byte long). Once again, the "memory" as represented here had to be altered to accommodate this information.

The recognition of such changes is important for students learning about the algorithms. However, the provision of the line-animator at least makes their task easier in that the input, interpretation and output of data is handled for them.

### 3. Other line-animators and their features

We give examples of four other line animators that we have used with good effect over the years. Each illustrates a particular point of the genre.

#### Example 1. Quicksort

A very common candidate for animation is quicksort. By simply adding an intelligent line writer in the algorithm itself, one obtains the following output.

```

**** Testing Quicksort ****
27 91 42 23 40 74 96 28 31 59
27 31 28 23
27 23 28
   27 28
     28 31
       74 96 42 91 59
         42 96
           96 74 91 59
             91 96
23 27 28 31 40 42 59 74 91 96

```

Both the movement of the elements and the division into sublists is immediately obvious. No special

software is required to provide such output, and the student is aware all the time of the algorithm at the programming level.

#### Example 2 Queue data structure

Moving on to data structures, we may wish to show how a queue can be used in a concrete example. Here a simulation is being run, and the state of the queue is shown at each stage.

```

* Doctor's Waiting Room Simulation *
There are 7 chairs
Arrivals on 2,4,6; patients seen on 1,2

```

| Time | Choice | Patient numbers     |
|------|--------|---------------------|
| 2    | 1      | Empty               |
| 3    | 1      | Empty               |
| 4    | 6      | Empty               |
| 5    | 5      | 4                   |
| 6    | 4      | 4                   |
| 7    | 4      | 4 6                 |
| 8    | 5      | 4 6 7               |
| 9    | 5      | 4 6 7               |
| 10   | 1      | 4 6 7               |
| 11   | 4      | 6 7                 |
| 12   | 4      | 6 7 11              |
| 13   | 6      | 6 7 11 12           |
| 14   | 4      | 6 7 11 12 13        |
| 15   | 1      | 6 7 11 12 13 14     |
| 16   | 4      | 7 11 12 13 14       |
| 17   | 6      | 7 11 12 13 14 16    |
| 18   | 6      | 7 11 12 13 14 16 17 |
|      |        | Waiting room Full   |
| 19   | 1      | 7 11 12 13 14 16 17 |

The simulation can print the status of the queue very simply if it has a toString method.

#### Example 3. Turing machine

Elaborate Turing Machine simulators have been written [Decker]. Our 117 line Java program accepts input of the form shown in this piece of a palindrome tester program:

```

1 a/.,R 2
1 b/.,R 7
1 ./.,L 6
2 a/a,R 2
2 b/b,R 2
2 ./.,L 3
3 a/.,L 4
3 b/b,L 5
3 ./.,L 6

```

The central part of the program is the method which acts as the Turing Machine engine. Students can see how it works by reading the code.

```

void runtheMachine () throws IOException {
    getTape();
    int state = 1;
    int input;
    Instructions ins;
    do {
        input = convertToInt(tape[head]);

```

```

        ins = transition[state][input];
System.out.println(String.valueOf(tape)
    +"\t"+state+' '+tape[head]+'/'+ins);
tape[head] = ins.outsymbol;
if (ins.direction == 'L')
    head --;
else head++;
state = ins.nextstate;
} while (state > 0);
System.out.println(String.valueOf(tape)
    +"\t"+state+' '+tape[head]+'/'+ins);
if (state != 0)
    System.out.println ("Ended in error");
else
    System.out.println("Ended
successfully");
}

```

#### Example 4: Merge sort and Find Min Max

These two algorithms were taught to first years as examples of divide and conquer. Because the course was emphasising object-oriented programming, we wanted the students to use objects in the parameters passed in the recursive routines. We gave them findMinMax as follows:

```

void findMinMax (MinMaxInfo info) {
    System.out.println
        ("At beginning :" + info);
    if (info.n == 2) {
        info.min = minOf(a[info.left],
            a[info.right]);
        info.max = maxOf(a[info.left],
            a[info.right]);
    } else if (info.n == 1) {
        info.min = a[info.left];
        info.max = a[info.left];
    }
    else {
        // Create info about the sublists
        MinMaxInfo leftList = new MinMaxInfo
            (info.left,info.midpoint,
            info.min,info.max);
        MinMaxInfo rightList = new
MinMaxInfo
            (info.midpoint+1,info.right,
            info.min,info.max);
        // Process them
        findMinMax(leftList);
        findMinMax(rightList);
        // Use the results
        info.min =
            minOf(leftList.min,
rightList.min);
        info.max =
            maxOf(leftList.max,
rightList.max);
    }
    System.out.println("At end " +
info);
}

```

The point we wanted to get across was not so much the algorithms of merge sort or find-min-max, but how you would program these in an object-oriented way. Without reference parameters, the solution was not

immediately obvious to many students who had previously used Pascal or C. Given the structure above, they were asked to change as little as possible to make it work for merge sort. The similarity in structure can be seen from the central part of the method which becomes:

```

        // Create info about the two
sublists
MergeInfo leftList = new MergeInfo
    (info.left,info.midpoint);
MergeInfo rightList = new MergeInfo
    (info.midpoint+1,info.right);

// Process them
mergeSort(leftList);
mergeSort(rightList);

// Use the results
a = merge (leftList, rightList);

```

The animator is providing the array or objects and the ability to write it out, leaving the student free to concentrate on the actual coding of this quite complex algorithm.

#### 4. The advantages of line-animators

Line animators have numerous advantages.

2. They can be **created** in very little time to suit specific needs.
3. Frameworks, such as the one in Section 2, can be **adapted** to several algorithms..
4. They are small, **platform-independent** and require no special facilities to run.
5. They **reveal** to the student the process of the algorithm at a level very similar (or even almost identical) to that in a text book being followed.
6. By deft use of integers and characters, output can be made easily **readable** (for example, the practice of creating the object "p" on the heap with "p" in each of its fields.
7. The output provides a **trace** of the algorithm which is a starting point for the student to study what happened for given input. The output can be printed and perused later: it is not constantly being overwritten as would be the case in most visualisors.
8. They can be certainly be made not only **data dependent**, but **programmable**, as shown in Section 2.

#### 4. A classification of animators

In comparsion to other more well know animators, how do line-animators stand up? We classify animators and visualizors into three groups:

1. **line-animators** which concentrate on a few algorithms in a group and have textual output

2. **graphic assistors** which can be combined with a group of similar algorithms and called to provide graphical output of algorithm progress
3. **visualizers** which provide a complete environment, programmable perhaps, for showing the progress of an algorithm graphically.

Line animators do not have to be as primitive as shown in this paper. The same concept of simply reporting in text format what is happening is used with a window-based animator for parsing [Kaplan]. A problem with window-based systems is either each window replaces the previous, so that valuable history is lost, or there are too many windows and the student becomes confused..

A successful example of a graphic assistor is given in [Magee]. In this text book, the authors examine numerous concurrency algorithms, and illustrate them via standard rotating pies whose speed is governed by sliders. The graphic routines are called from the plain Java code for a monitor, a semaphore or whatever. The student gains considerable leverage in visualising the progress of the algorithm, without having to program the somewhat complex graphics. The standardised nature of the output also aids in comparison of similar programs, for example the dining philosophers with or without a deadlock-free adjustment. In addition to output, it is important to retain the input on the screen so that progress can be related to it, and the data can be tweaked for the next run. [Bishop] proposes a simple display system which has input and output windows constantly on screen. Calls to these input and output on such a display are very similar to line i/o, and the authors illustrate how sophisticated numerical algorithm routines, or indeed sorts and searches as well, can be illustrated with minimum effort.

Visualisers were covered in Section 1, and comprise most of the references here. Now we can consider a comparison of these three groups, with a view to putting line-animators in context. The criteria for comparison are derived from some of the references and from the points in Section 4.

| Criterion                      | Line-animat or | Graphic-assitor | Visual izer |
|--------------------------------|----------------|-----------------|-------------|
| trace available                | yes            | no              | no          |
| extensible to other algorithms | mostly         | limited         | can be      |
| effort to create               | small          | medium          | large       |
| programmable algorithms        | limited        | yes             | can be      |
| exciting                       | no             | medium          | yes         |
| reveals the algorithm          | yes            | yes             | can         |

## 6. Conclusion

The table shows that line-animators can hold there own as a useful tool. While admittedly not exciting or visually appealing, they take very little time to create and can within limits be just as extensible and programmable as other more sophisticated tools. They have the two big advantages: they can reveal the algorithm to the student and they can produce a trace output which can be printed and studied.

While line-animators cannot cover all areas of computing adequately, they can be used with great effect in second level data structures course, compilers, numerical analysis and some computation areas. It is hoped that they will receive more attention in the future. A web site with all the line-animators that we have is on <http://www.name.withheld> and they have been available for use in one way or another for some time.

In general, we do seem to regard the link between exciting technology and good learning as a given. Line animators buck this trend. They free the instructor of a heavy development cycle (or adoption, which can be as problematic) and they make the understanding of the algorithm firmly the student's responsibility, while providing sufficient assistance for experimentation at the data and even program level.

## References

- Appel A, Modern compiler implementation in Java, Cambridge University Press, 2000.
- Bishop J and Bishop N, Object-orientation in Java for scientific programmers, 31<sup>st</sup> SIGCSE Technical Symposium on Computer Science Education, 362-366, Austin 2000
- Decker R and Hirschfield S, The Analytical Engine, PWS Publishing, Boston, 1998
- Diehl S and Kerren A, Levels of Exploration, 32<sup>st</sup> SIGCSE Technical Symposium on Computer Science Education, 60-64, Charlotte 2001
- Kaplan A and Shoup D, CUPV – a visualisation tool for generated parsers, 31<sup>st</sup> SIGCSE Technical Symposium on Computer Science Education, 11-15, Austin 2000
- Kehoe C, Stasko J and Taylor A, Rethinking the evaluation of algorithm animations as learning aids: an observational study, Technical Report, Georgia Institute of Technology, March 1999.
- Khuri S and Hsu H, Interactive packages for learning image compression algorithms, 5<sup>th</sup> SIGCSE ITiCSE, 73-76, Helsinki 2000
- Khuri, S, Educational tools, on <http://www.mathcs.sjsu.edu/faculty/khuri/animati on.html>, visited 13 November 2001
- Koldehofe B, Paptriantafilou M and Tsigas P, Distributed algorithms visualisation for educational purposes, 4<sup>th</sup> ACM ITiCSE, 103-106, Cracow 1999
- Magee J and Kramer J, Concurrency: 1999, Wiley, Chichester, 1999.

- Naps T, Norton L, Eagan J, JHAVE – an environment to actively engage students in web-based algorithm of accessible educational software, 31<sup>st</sup> SIGCSE Technical Symposium on Computer Science Education, 109-113, Austin 2000
- Rasala R, Toolkits in first year computer science: a pedagogical imperative, 31<sup>st</sup> SIGCSE Technical Symposium on Computer Science Education, 185–191, Austin 2000
- Rößling G, Schüler M and Freislebel B, The Animal algorithm animation tool, Proc 5<sup>th</sup> SIGCSE ITiCSE, 37-40, Helsinki 2000
- Stasko J, Algorithm Animation, at <http://www.cc.gatech.edu/gvu/softviz/algoanim/algoanim.html>, visited on 13 November 2001
- Stern L, Søndergaard and Naish L, A strategy for managing content complexity in algorithm animation, Distributed algorithms visualisation for educational purposes, 4<sup>th</sup> ACM ITiCSE, 127-130, Cracow 1999