7 Conclusions and Future Work

7.1 Good algorithmic animations

Right at the beginning *of The Visual Display of Quantitative Information* Edward R. Tufte summarizes his rules for graphical excellence. This thesis is concerned also with graphical quality, but first and foremost, with algorithmic animation excellence. The difference is important: while Tufte only needs to consider rules for the esthetic and efficient display of *static* data, we are confronted with the more challenging problem of representing movement, changes, and individual steps of algorithms, that is, dynamics [Foley 86]. However, we can paraphrase Tufte's original rules, adapting them to the problem of algorithmic animation [Tufte 83]. What we obtain is a useful set of heuristic rules which can be found fully developed or just implicit in some sections of this thesis. Tufte's modified rules (my modifications are highlighted using italics) are now:

> Excellence in *algorithmic animation* consists of complex ideas communicated with clarity, precision, and efficiency. *Algorithmic animation* should
>
> - show the data *transformations*
> - induce the viewer to think about the substance rather than about methodology, graphic design, the technology of *rendering*, or something else
> - avoid distorting what the *algorithm* has to say
> - present many *steps* in small space
> - make large data sets coherent
> - encourage the eye to compare different *algorithm steps*
> - reveal the *algorithm* at several levels of detail, from a broad overview to the fine structure
> - serve a reasonable clear purpose: description, exploration, *learning* or decoration
> - be closely integrated with the verbal descriptions of *the algorithm*.

The animations discussed in this thesis try to put the spotlight on all data transformations, using explicit movement of data objects or highlighting them. The idea is always to convey the essence of an algorithm to the viewer, making her or him concentrate in the most important operations. The scripting language can deal with small and with large data sets. Small data sets were used extensively in the

Flash animations. Large data sets were handled as examples with E-Chalk Animator. The animations try to guide the eye of the observer, connecting a view of the pseudocode with views and overlays of the data. Algorithm steps can be reviewed, either by rerunning the algorithm or by letting it execute backwards. The algorithmic animation tools described in this thesis serve the main purpose of teaching students about such algorithms, and both E-Chalk Animator and Flashdance can be enhanced with sound and verbal descriptions.

The scripting language is a general purpose animation tool, and, of course, it can be misused. Bad animations can still be produced with the best animation engine available, in the same way that a blackboard can be used to give bad or good lectures. The animation engine is a clean slate in which the algorithm animator can imprint his or her understanding of an algorithm. The best animations, as explained in Chapter 3, are those in which the mental data structures proposed by the algorithm correspond best to the algorithms data structures. Or to put it in the words of Bertin: “The entire problem is one of augmenting this natural intelligence (of the user, ME) in the best possible way, of finding the artificial memory that best supports our natural means of perception” [Bertin 83].

I have tried to follow the principles for good algorithmic animations discussed in Chapter 3 and summarized above in Tufte’s words. It is interesting to know how others have applied my animation tools, as described in the next section.

7.2 Evaluation of algorithmic animations

The algorithmic animations developed with Chalk Animator and Flashdance have been used in the classroom. The handcrafted animations presented in Chapter 3, have been also used for teaching algorithms and data structures at the Technische Fachhoschule Berlin and at the Technische Fachhochschule Kiel.

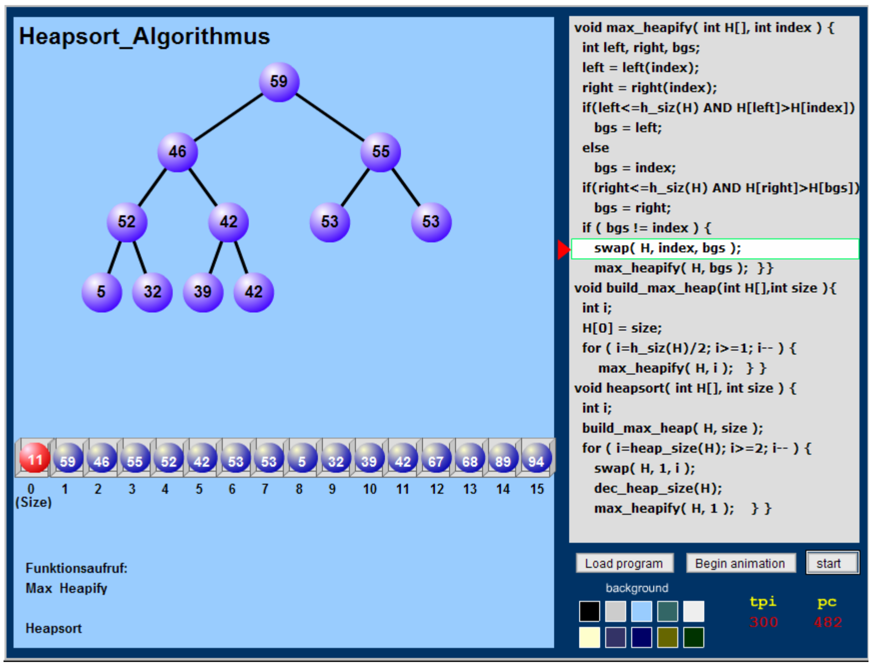
From our discussion in Chapters 2 and 3, about the value and impact of algorithmic animations, it should be clear that algorithmic animations can be used in two different settings. In the *intructivist* framework, the information for the students is arranged and digested in order to give it a form easy to grasp and remember later. Chalk Animator and Flashdance can be used in this framework. The first, especially, is geared towards teaching in the classroom, while Flashdance animations are suited for delivery through the Internet.

In the *constructivist* framework, students learn the material but they also learn to build their own representations. By experimenting, they construct an internal model which is arguably superior to a model provided from outside. Learning by doing is the strategy proposed by constructivists.

I applied partially the constructivist approach during an Algorithms and Data Structures course at the Fachhochschule Kiel in the winter term 2003. For example, the animations presented in Chapter 6 were tested intensively by my students. The Flashdance animation language is so simple, that as an experiment, I let students in Kiel develop their own animations as part of the homework. Groups of two students wrote C programs and produced the animations described in what follows.

Heapsort

The animation produced by the students is rather complex. The C code is visible on the right window in Figure 7.1. The visualization of the heap is the classical one, as binary tree. The array being sorted is visible below. Four numbers at the end of the array have already been sorted. Unfortunately, the animation does not highlight these sorted numbers using another color, for example, green. The instruction being executed is highlighted with white. The specific function being called is shown with additional text at the bottom of the left display.



```
void max_heapify( int H[], int index ) {
 int left, right, bgs;
 left = left(index);
 right = right(index);
 if(left<=h_siz(H) AND H[left]>H[index])
   bgs = left;
 else
   bgs = index;
 if(right<=h_siz(H) AND H[right]>H[bgs])
   bgs = right;
 if ( bgs != index ) {
   swap( H, index, bgs );
   max_heapify( H, bgs );  }}
void build_max_heap(int H[],int size ){
 int i;
 H[0] = size;
 for ( i=h_siz(H)/2; i>=1; i-- ) {
    max_heapify( H, i );   } }
void heapsort( int H[], int size ) {
 int i;
 build_max_heap( H, size );
 for ( i=heap_size(H); i>=2; i-- ) {
   swap( H, 1, i );
   dec_heap_size(H);
   max_heapify( H, 1 );   } }
```

Funktionsaufruf:
Max Heapify

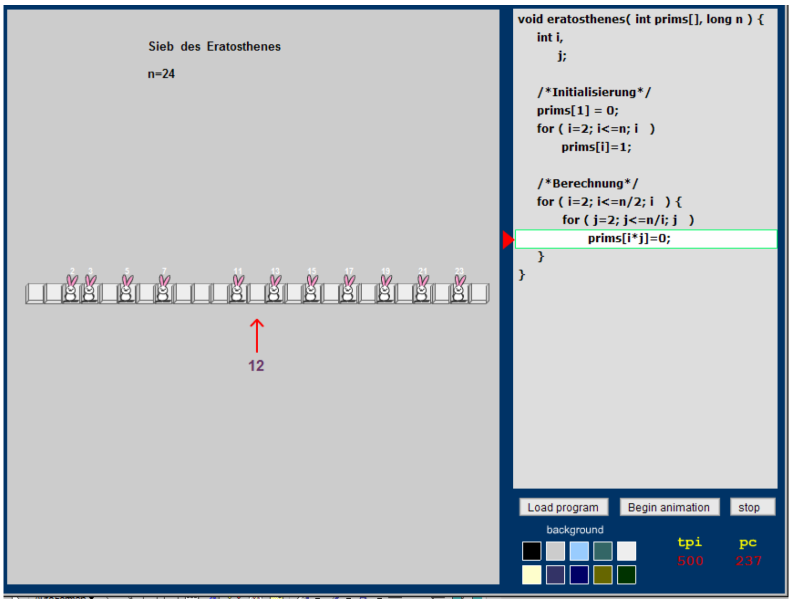
Heapsort

Figure 7.1 Student produced animation of Heapsort.

This animation is a good example of what students taking a first programming course can produce after a single lecture in which Flashdance was introduced.

Sieve of Erathostenes

The next example is an animation of the classical sieve of Erathostenes algorithm (Figure 7.2). The students decided to represent numbers by their position in an array, and each element by a rabbit, which falls from the array when “touched” by an index jumping along the array with steps of length 2, 3, 5, etc. In this particular animation, the number of data points is very limited. It would have been probably better to use a larger array, representing the data by single points. The C code is small, it is highlighted while being executed and the fall of the rabbits makes for an interesting animation. The color of the background has been poorly chosen, but it can be modified by the user while running the animation.



```
void eratosthenes( int prims[], long n ) {
    int i,
        j;

    /*Initialisierung*/
    prims[1] = 0;
    for ( i=2; i<=n; i   )
        prims[i]=1;

    /*Berechnung*/
    for ( i=2; i<=n/2; i   ) {
        for ( j=2; j<=n/i; j   )
            prims[i*j]=0;
    }
}
```

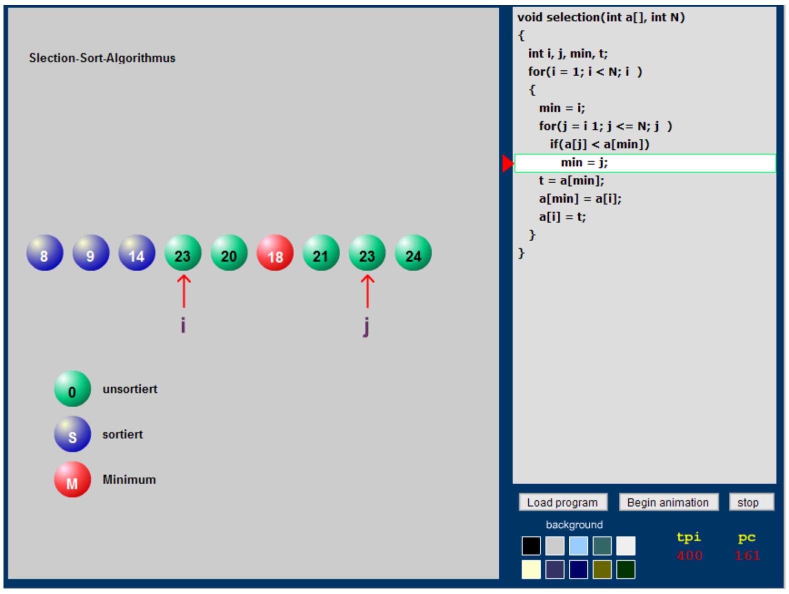
Figure 7.2 Student produced animation of Erathostenes’ sieve.

Selection Sort

This animation looks good on the screen (Figure 7.3). The students animated different actions using different speeds, and the color code is explained below the array being sorted. The C code is highlighted while being executed. If the students had used smaller spheres, more data could have been shown.

These few examples show that students are able to make creative use of the scripting language after a short introduction in class. My experience was that the students became more involved with their assignment, once they could see the algorithm running. The students worked many hours on their projects and this in a class, in which students traditionally have not gone further than learning the basics of C. In this class, algorithmic animation helped to elevate the level of learning and achievement. My experience was thus similar to that reported by Stasko

[97]. Algorithmic animation can easily become part of the curriculum when the appropriate tools are available and the time invested in the production of the animations is drastically reduced.



```
void selection(int a[], int N)
{
  int i, j, min, t;
  for(i = 1; i < N; i  )
  {
    min = i;
    for(j = i 1; j <= N; j  )
      if(a[j] < a[min])
        min = j;
    t = a[min];
    a[min] = a[i];
    a[i] = t;
  }
}
```

Figure 7.3 Student produced animation of selection sort.

7.3 Abstract interpretation and algorithmic animation

Something should be said now about the future. Many algorithm animators have struggled with the problem of generating automatic animations for many years. Brown, for example, discusses Quicksort and the way it could be recoded to provide an automatic animation. Indices, he notes, could be increased not by writing "`j = j+1`", but by making the operation explicit with an increment function, which at the same time, updates the position of an arrow on the screen [Brown 88c]. Such kind of recoding should be done automatically by the compiler or interpreter; the programmer should not carry the burden of rewriting her code.

Code transformations require a higher level of intelligence by the compiler, that is, a higher level of abstract understanding about the operations involved and which of them are "interesting events" for an animation. The way to provide this capability could be by performing an abstract interpretation of program code in order to find such interesting code sections.

In abstract interpretation, code is run by not using actual data, but a representation for the possible data types. In an abstract interpretation run, we are interested in examining properties of a program, such as whether the code can be parallelized or not, or whether data structures can be absorbed by the garbage collector or not. It would be feasible to examine code for the presence of indices and arrays, or linked data structures. The code could be transformed in order to make pointer or index operations explicit, like in the Quicksort example mentioned above. The user could then declaratively select which of the data structures should be visualized and a compiler would generate the appropriate code.

A first promising example in this direction, which at the same time deals with the problem of representing large data structures on a screen, is the work of Braune and Wilhelm [Braune 00]. Using techniques borrowed from the compiler construction field, they propose to focus the display of data structures only on those special nodes that are relevant for the visualization. In a sorting algorithm, for example, two nodes in a list could be referenced by the pointers *p* and *q*. Figure 7.4 shows a possible visual transformation of a standard list, to one in which only special nodes are rendered individually. The nodes in-between the first node and special nodes are summarized using a box pointing to itself, an iconic representation of a recursive data structure. We could as well have rendered three dots, to illustrate repetition of the main pattern.

This example shows how shape analysis, that is, an analysis of the data structures and their shape in a visual representation, can help to simplify the animation and reduce the cognitive load on the viewer. This shape analysis can be done automatically and represents a first glimpse of the advantages of automatic animation of algorithms. The same techniques can be applied to trees and any other recursive data structure.

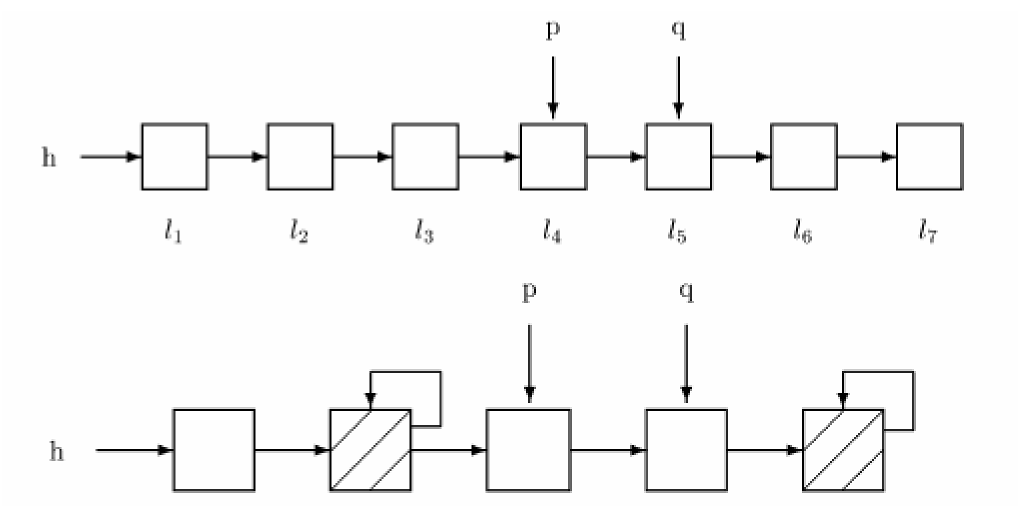


Figure 7.4 Abstract graphical display (below) of a long linked list (above).

My expectation is that techniques borrowed from the compiler construction community will serve one day to analyze code and generate animations in semi-automatic form. An adequate user interface could allow the user to provide some insight or manipulate the graphical and esthetical part of the representation, the portion of the art of algorithmic animation impenetrable to computers.

7.4 The future of algorithmic animation

Much more complex algorithmic visualizations will be possible on the electronic blackboards of the future. Therefore, in the closing section for this chapter, I would like to speculate about what will be possible years from now by just describing a computer science lecture taking place in 2015. The story is fiction, but informed fiction (see for example [Cypher 93]).

Professor Müller arrived to his class at 8:00 AM, ready to deliver his lecture about heaps, the next data structure to be covered in the Algorithms and Programming course. He went slowly to the front of the classroom, every one of his sixty eight years weighting heavily on his every step. Sometime around the turn of the century, retirement age was increased from 65 to 70 years, and here he was, still working at the trenches.

The blackboard looked like any other blackboard from a distance, but on closer examination one could see that this blackboard was smooth and shiny. Prof. Müller was a computer scientist from the old school. He never adopted the fashions of the day. Never bought a cellular telephone, never used the old PowerPoint system championed by the long extinct Microsoft Corporation. He stuck to the chalk and was very proud of the way he could fill many blackboards during an entire lecture. Every one of them was, when finished, a piece of art. It would have been possible to hang each of them at the Berlin Museum of Modern Art, to great advantage for the museum.

A few years earlier, in 2013, the traditional blackboard had eventually disappeared. It was too expensive, compared to the new organic screens being built for a few Euros per square meter. When technologists finally mastered the production of organic screens in 2006, production skyrocketed and slowly displaced old technologies. Nobody hung pictures on the wall anymore. An organic display could update the image on the screen every minute. Apartments without windows could now be sold. People just hung a large organic screen, connected to a video camera in any city in the world. They could have a virtual window with a real-time view of Paris, Manhattan or the ocean.

Prof. Müller never bought such a window, and was rather speechless when his old beloved blackboard disappeared. After some days of mourning, he started to experiment with the new blackboard, and to his surprise it was exactly like the old one. The chalk was now a piece of plastic, but he could write perfectly on the screen and that was all he needed. He became an expert in a few days.

Prof. Müller started his class by drawing an array of fifteen elements as a box with fifteen compartments. He then explained the logical structure of a heap and how it could be stored in an array. At his command, the array smoothly transformed into a tree, showing how each entry in the array was mapped to a node in the tree. Prof. Müller proceeded to explain the heap property – he erased the number in the root and wrote a smaller one. The heap then reorganized automatically by smoothly exchanging values down the logical tree, until the heap property was fulfilled again. Prof. Müller tried again, the result was the same.

He then explained the main idea of Heapsort and extracted manually the largest element from the heap, exchanging it with the last element in the array. The heap reorganized again, and he kept exchanging the root with the last node, until there were no elements left. They had been ordered.

The students asked the professor for the pseudocode of the algorithm. Prof. Müller could have written it down himself, but he just asked the electronic blackboard to produce the pseudocode of Heapsort automatically. The blackboard had generalized the manual sorting procedure Prof. Müller had followed and showed neatly typed pseudocode on the screen. Prof. Müller then asked the machine to please sort 1000 random numbers using this algorithm and show the computation graphically, to which the machine complied without a glitch.

Very satisfied, Prof. Müller said to the electronic blackboard "Thank you very much, HAL that is all for today. Please produce a transcription in electronic format for all students and put it in their electronic mailboxes".

Prof. Müller left and on his way home he muttered to himself "HAL, this Heuristically Programmed Algorithmic Computer is better every day. I wonder if it likes my algorithms class".