4 Chalk Animator – Algorithmic Animation for an Electronic Chalkboard

Conventional systems for algorithmic animation handle only computer generated images as building blocks in an animation. Perfect rectangles, circles, arrows, etc. are drawn by the user using a graphical editor or are generated automatically by the system. Exceptions are Hundausen [02] who studied a system for animating sketches of objects produced by an editor, and Stasko, who studied the animation of objects along paths defined interactively by the user, also with an editor [Stasko 91a, 98b]. Strothotte [98] has also studied the techniques involved in animating non-photorealistic images rendered with the computer. In this chapter we consider a more radical alternative: the generation of algorithmic animations starting from sketches drawn by the user on an electronic blackboard, which is both the presentation tool and the user interface for the lecturer. Not only is this approach time-effective for the lecturer, but also the "look and feel" of animated sketches is very different from computer generated graphics. Sketch animation resembles best the kind of teaching done using a traditional chalkboard. If applied properly, sketch animation can result in a more interesting lecture.

This chapter is organized as follows. First we review the basics of the E-Chalk system, developed at the Free University of Berlin [Rojas 01, Friedland 02, 03a, 03b, 03c]. We then review how teachers make use of the chalkboard to produce what I call "static animations" (a term used in the "visual literacy" literature). Then we will see how to automatically produce sketch animations in E-Chalk with a sequence of images produced by a program running as intelligent assistant to the lecturer. Since a transcription of E-Chalk lectures can be printed, this is an important additional feature for the whole system. We discuss the protocol needed for synchronization between the animation program and E-Chalk and we finish giving some examples of animations.

4.1 The E-Chalk System

The main idea of the E-Chalk system is to provide the functionality of the traditional chalkboard using a large contact sensitive computer screen, but enhanced

with all the capabilities of a digital system. An electronic blackboard should be as easy to use as a traditional one. It should be in service 24 hours a day, seven days a week (that is, it should switch-on when touched). The only interface to the electronic board should be a stylus, instead of a piece of chalk.

E-Chalk is not only a software system but also a vision of our digital future. The large computer screens which are needed are not available yet at a price affordable for elementary schools or small universities. However, prizes are going down at such a pace that in a few more years, large computer screens will be ubiquitous. A day will arrive in which large flat computer screens, possibly made of organic materials, will be as cheap, or cheaper than traditional blackboards. There will be no reason for not going digital in the classroom.

The current implementation of E-Chalk was written in Java, mainly because the large computer screen is attached to a computer and remote viewers can connect to the system using an Internet browser. A Java enabled browser frees the viewer from having to install a plug-in to play the lectures being transmitted. Each lecture stream transports its own decoder as an Applet, so that users do not have to install and reinstall software every time the server software is changed.

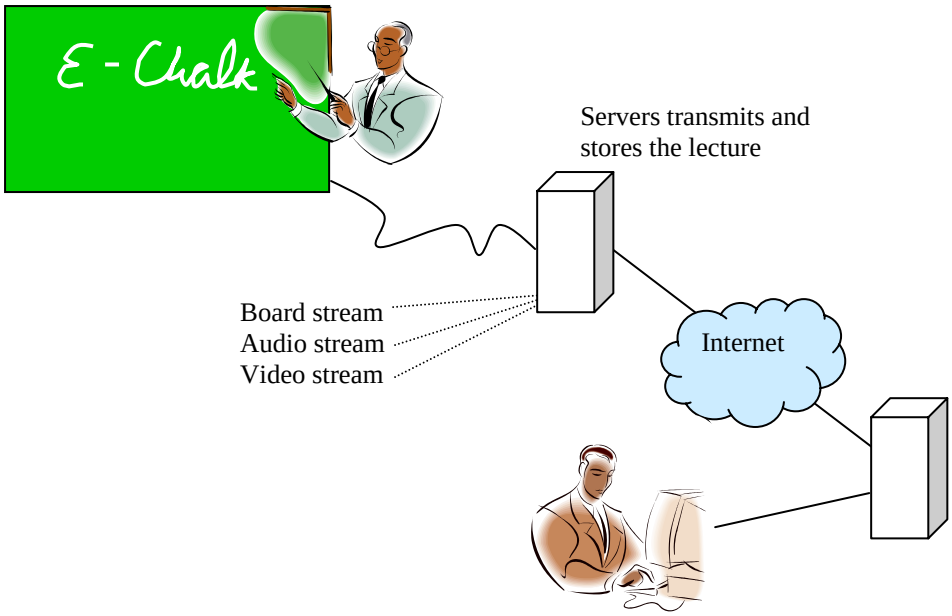

Figure 4.1 The E-Chalk system. A lecturer writes on an electronic blackboard. Audio, video, and board contents are stored and are streamed through the Internet. Remote viewers watch a lecture using a Java enabled browser.

When an E-Chalk session is started, the server computer starts storing and sending three streams: the board events, the audio channel, and an optional video channel. The three streams can be accessed from a Web page, by starting the E-Chalk cli-

ent, which is a collection of three client Applets, one for each stream. The streams are synchronized by the audio time stamp. Therefore, an E-Chalk session can be recorded completely, as on a video tape, but the quality of the reproduction of the board on the client side is much higher, since the board is repainted with the full resolution of the computer screen. Figure 4.1 above shows schematically a teaching scenario: a lecturer teaches to a live audience; the E-Chalk server transmits the audio, video and board streams and stores an archival copy. A remote viewer watches the class using an Internet browser.

Figure 4.2 shows a diagram of the main components needed for an E-Chalk transmission. Three servers synchronize on the sender side, one for each stream. The servers fill buffers, in order to provide a smooth transmission stream even when the Internet quality of service is changing. On the receiving side, when the viewer clicks on the URL of the lecture, three Applets are started, one for each stream. The three Applets receive and play the information sent by the server. In the case of the board Applet, the stream is composed of ASCII characters that encode the kind of action required: draw a line, paste a picture, write text, etc. The viewer hears and sees the lecture and can also scroll the window where the lecture is shown.

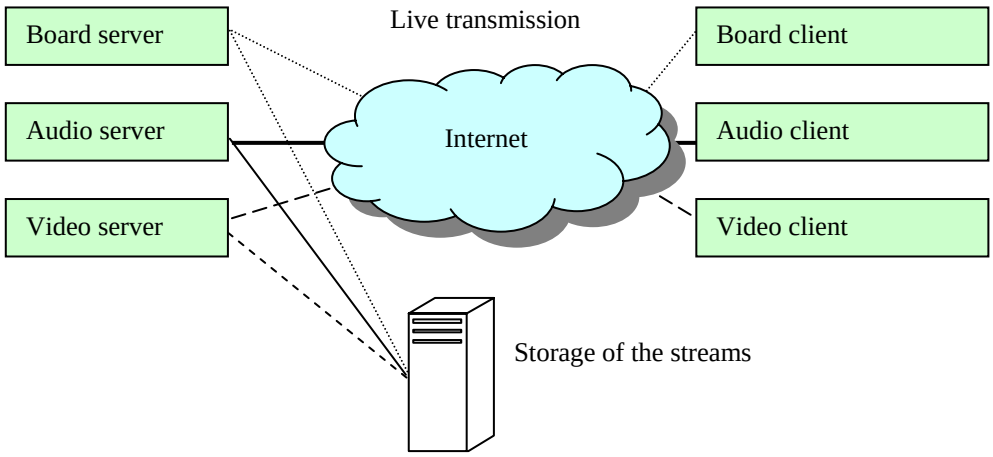

Figure 4.2 Three stream servers feed three Applet clients during an E-Chalk lecture (from the E-Chalk documentation)

Figure 4.3 is a screen dump of an actual lecture, as seen by a remote viewer in his browser. The look and feel of the screen is that of a lecture on a good blackboard. The use of color helps to emphasize some important aspects of the lecture.

The main feature of E-Chalk is to go beyond the original blackboard metaphor and provide “intelligence and information on demand”. This means, that a series of special programs is running in parallel with E-Chalk and is watching the user interacting with the screen. Certain programs can then become active when certain conditions are met. For example, a program can observe the handwriting of the user and if a mathematical formula is entered, and if the user writes a special stop

symbol, the program can interpret the formula. If it is an equation, it can solve it. This capability has been already implemented in E-Chalk, using Mathematica as the mathematical equation solver [Tapia 02, 03a, 03b]. Another example could be a program that interprets musical notation written on the blackboard. If the user gives a signal (that is, if she writes a special gesture symbol) the computer plays the music. Another example could be a program that plays tic-tac-toe with the user, as soon as the user draws a tic-tac-toe field and gives a signal. The possibilities for interfacing applications with the blackboard are endless.

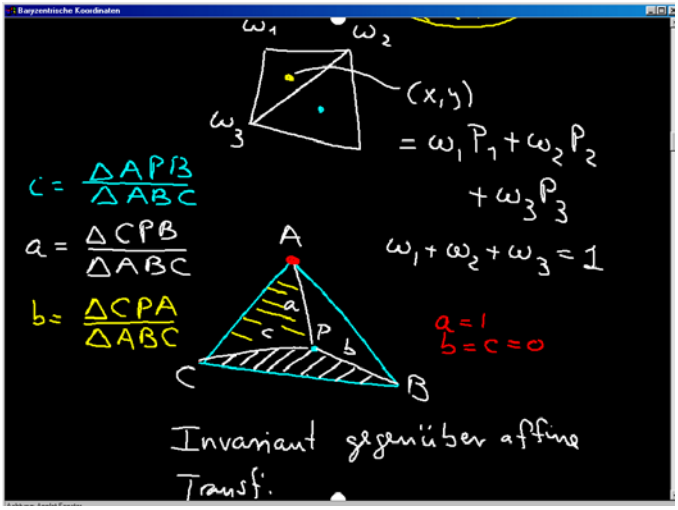

Figure 4.3 A real E-Chalk lecture about geometric concepts at the FU Berlin.

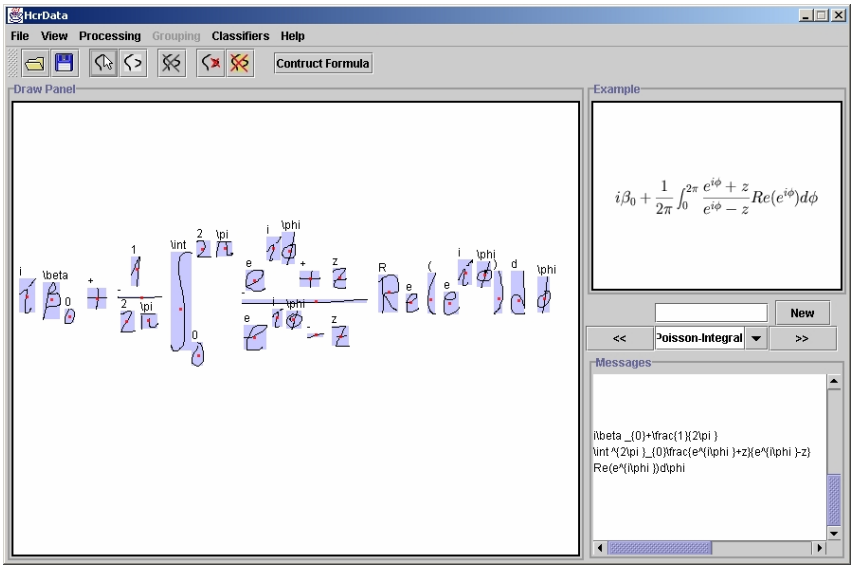

Figure 4.4 The mathematical handwriting recognition engine and editor.

Regarding algorithmic animations there are two things which come immediately to mind as possible extensions of the blackboard metaphor: a) the immediate execution of code written on the blackboard, b) the animation of algorithms started by the lecturer. Imagine a lecturer who writes the code for bubble sort in some concise pseudocode. Imagine that the code runs and delivers a numerical result, or still better, it animates an array in which the steps of the algorithm are clearly visible as exchanges of numbers.

In this thesis both problems are considered, whereby I concentrate on the second one: the animation of algorithms on an electronic blackboard, interactively with a lecturer. But before we proceed further with this investigation, we will look more closely at what constitutes a good "animation" of an algorithm on the blackboard.

4.2 Good chalkboard animations – good paper animations

There are many cases in which lecturers have to illustrate a dynamic process using a traditional chalkboard. Algorithms are such an example: if the steps of the algorithm are to be visualized, both the program and the data can be viewed "dynamically". There are several alternatives when using a chalkboard:

a) A diagram of the data model can be drawn and can be progressively erased and drawn again, to show the modifications.
b) A diagram can be drawn and subsequent modifications can be shown on a second version. For example, if a sorting algorithm is being discussed, the current state of the list of numbers can be displayed in successive lines ("stills," as defined in the ANIM system [Bentley 91a]).
c) A single diagram is drawn and temporal changes are shown using another color.

The first alternative, which is, animating by erasing, is confusing for the students. If a step is not understood, any subsequent step does not make sense. There is no way to go back to the previous state in order to connect it mentally with the present state. Some lecturers use this kind of static animation technique, but it should be avoided.

The second alternative is the one used in books, and usually the best that can be followed in class, when only a blackboard is available. Changes are shown in successive illustrations and the reader can go back and forth between the different steps of an algorithm. In fact, psychological experiments have found that illustrations of the individual steps of a process are superior to static illustrations. Such "dynamic" illustrations enhance the problem solving ability of students [Mayer 90].

However, it is difficult to generate "dynamic" illustrations in the classroom, since it requires too many redraws of the same data. An electronic chalkboard is therefore an ideal instrument for illustrating the dynamics of an algorithm. I call this a

"static animation". Each image by itself is static, shows a snapshot of the data or program state. But successive images show the state transitions. A "static animation" displays the temporal dimension across several pages, or across several images pasted on the screen. Static animation can be done in E-Chalk by pasting successive steps of an algorithm to the screen.

Figure 4.5, taken from an actual E-Chalk lecture delivered in 2002, shows how an algorithm can be "animated" using a high-level view of the computations involved. Quicksort is illustrated here by showing how a list is split into the elements smaller and the elements larger than the pivot (red arrows). Then, recursive calls sort both lists, which are appended, with the pivot in the middle, to yield the final result. We will return later to this problem of producing high-level, abstract illustrations of algorithms.

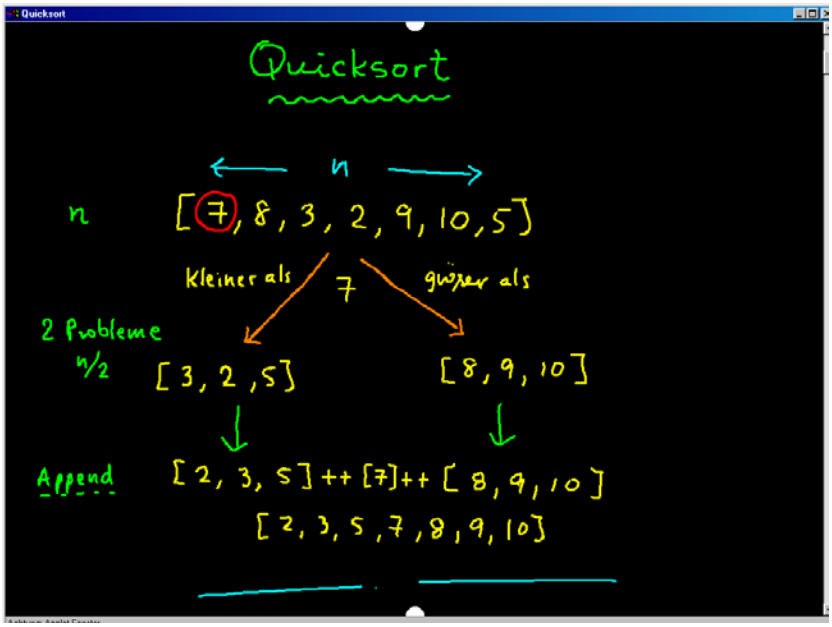

Figure 4.5 A "static animation" of Quicksort from a real lecture at the FU Berlin.

This "static animation" contains the essential information of the mechanics of the algorithm; it is almost "graphical code". Color has been assigned a meaning and the resulting composition is even esthetically pleasing. This could be called an algorithm explanation "without words".

My second example of a "static animation" is Figure 4.6. Here, a table for a Boolean function has been pasted four times on the electronic blackboard. The teacher can illustrate the individual steps involved in deriving a minimal Boolean expression for this table. By finding rectangular regions with ones, and enclosing them with a different color, it is easy to see where the final expression comes from. Such a manipulation could be done on a traditional blackboard, but is much easier to perform on an electronic blackboard. It is also very interesting to notice that when the costs of duplicating a diagram are so low, the possibility of spontane-

ously deciding to show a new example, given by a student, increases dramatically. A lecture can become more dynamic and interactive, when material and examples are produced on the fly, during a lecture.

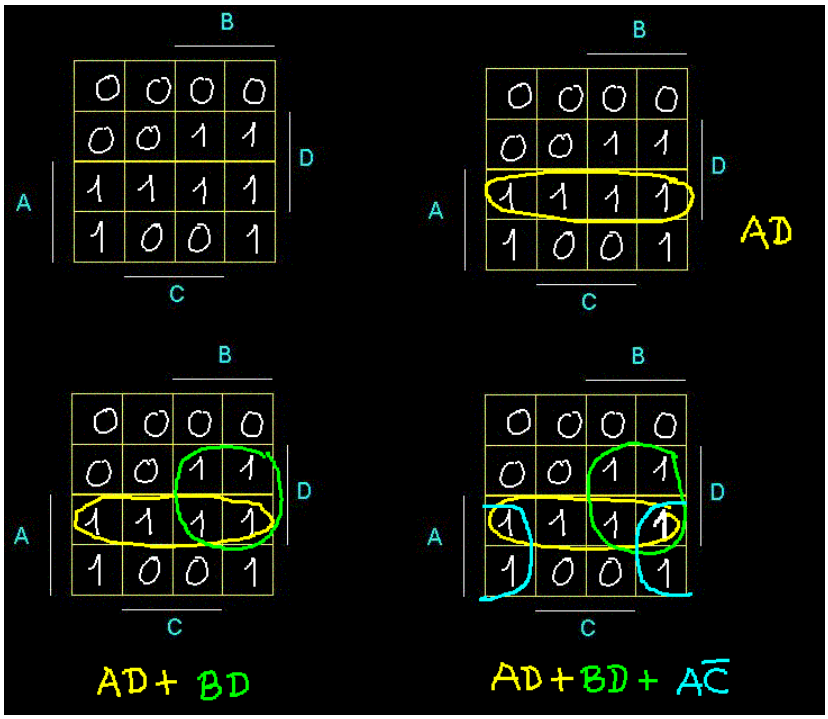

Figure 4.6 Reusing pasted diagrams in order to explain Karnaugh maps

As can be seen from this example, the temporality of the steps followed has been captured in a sequence of images. The color cue has been used to focus the attention of the viewer in the sequential production of the logical terms.

An electronic blackboard provides lecturers with additional possibilities for illustrating algorithm steps. In what follows, I will explain how my own animations exploit the features of E-Chalk.

4.3 Reuse of macros in E-Chalk

A “static animation” can be produced in E-Chalk in a simple way: the user starts the E-Chalk macro recording tool when preparing his or her lecture, draws a diagram the way this diagram should appear on the blackboard and saves the macro. Later on, the macro can be called during a lecture, from the E-Chalk menu, and can be annotated. A change of state can be shown by calling the macro again, pasting the original picture to the board, and annotating it. Through the reuse of macros, a sequence of images can be easily produced.

Figure 4.7 shows an example. The lecturer is explaining the different outputs of a circuit for the XOR function for all four cases of two inputs. Instead of erasing the first diagram, he can redraw the circuit (pasting the stored macro) and can annotate in a new way. Fig. 4.7 shows the evolution of the state of the electronic blackboard after each macro operation, and after the lecturer has labeled each diagram.

.

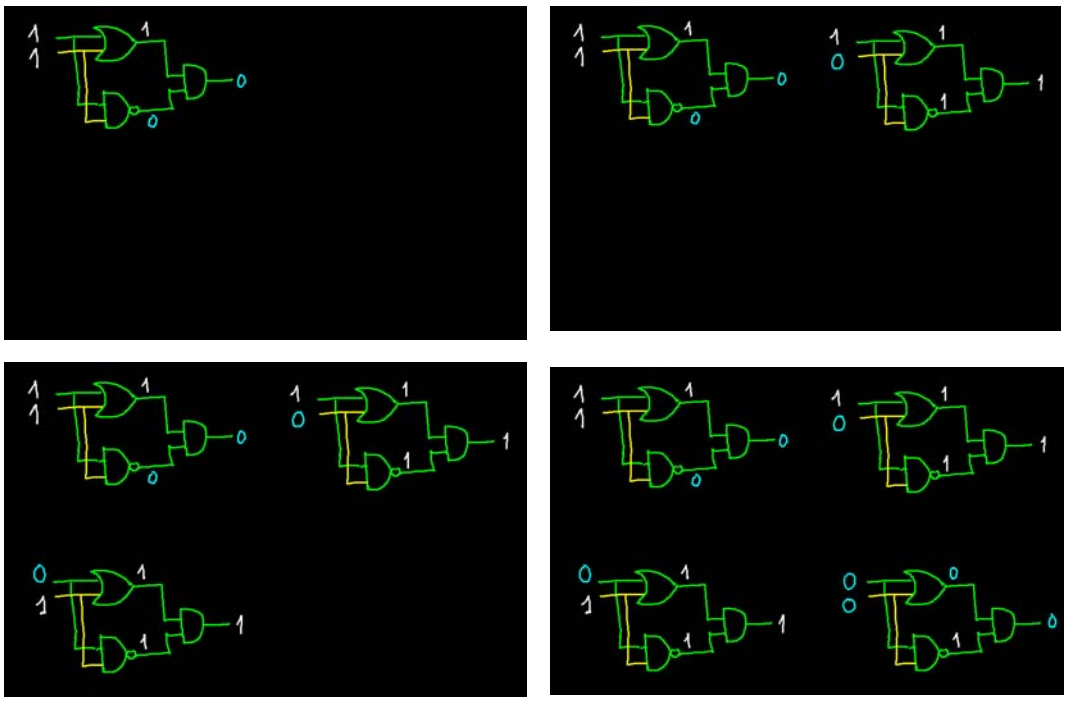

Figure 4.7 Static animation of a logic gate using the same pasted image

I call this approach *static animation by reuse of images*. The lecturer is an active participant, and is helped by the computer providing him with the ability to redraw sketches without loss of time and with minimal effort.

4.4 Animations in discrete steps

A second possibility of animating sketches in E-Chalk is by allowing the user to do the drawing and erasing himself, as preparation for a lecture. A macro couldbe recorded in E-Chalk, and a special “pause” button introduces a “pause event” in the stream recorded for the macro. When the macro is replayed in E-Chalk, the pause event could stop animating and provides the lecturer some time for explanations. The right-arrow key of the wireless keyboard used with E-Chalk could activate the continuation of the macro recording. Drawing resumes. Since any use of the eraser is also recorded in a macro, the user can proceed to erase parts of the

drawing and redraw them, possibly with some changes. The result is a stop-and-go animation of an algorithm, similar to the handcrafted animations mentioned in Chapter 2. Pauses will be included in E-Chalk macros in future releases of the system.

4.5 Reversibility

The E-Chalk macro facility can be used in reverse mode by just combining undo with redo steps. An animation which has been played as a macro is stored in the undo stack as a sequence of undo steps. For any stroke which has been painted by a macro, the corresponding inverse action is stored in the undo stack. It is possible to run an animation backwards by clicking on the undo button. Clicking on the redo button it can be made to run forward again.

We saw before that reversibility of an algorithm is an important aspect for any animation system. Here we obtain reversibility practically at zero cost because the undo stack is already a standard feature of the E-Chalk system. Reversibility allows the user to inspect an algorithm more closely, enhancing understanding.

4.6 The E-Chalk event format

The macro facility of E-Chalk was written by Lars Knipping, member of the E-Chalk team – it allows the system to load a macro file and play it, executing the board events stored in the file. For my own purposes, the format of the board events is important, since this is the data that must be produced by my algorithm animations.

Board events are dynamic events – the time of execution is given so that strokes are painted in a well-defined temporal order.

A macro file starts with a header:

```
ec1
1018
736
Macro #1
ff000000
0$Nop$created 03-10-30 11:46:57 GMT+01
```

Line 1 is a version identifier for E-Chalk lectures. Lines 2 and 3 give the size of the window in pixels (1018 horizontally, 736 vertically). The next line defines the name of the macro, and the last line is a NOP (No operation) containing the date and time of creation of the file.

The file continues with a definition of the strokes to be painted, for example:

```
708$Form$Line$64$96$64$72$ffffffff$2
708$Form$Line$69$96$69$6E$ffffffff$2
708$Form$Line$6E$96$6E$76$ffffffff$2
708$Form$Line$73$96$73$7A$ffffffff$2
708$Form$Line$78$96$78$7E$ffffffff$2
```

These five lines tell E-Chalk to draw five lines ($Form$Line command), all of them 708 milliseconds after starting the macro. The first line goes from the pixel with coordinates (64,96) to the pixel with coordinates (64,72). All numbers are in hexadecimal code, to save space. The color of the stroke is coded in 24 bits (last six hexadecimal digits of the long hexadecimal number). The code $ffffffff$ corresponds to white. The last number (2) is the width of the stroke.

As an example, consider the following event lines which are part of the macro that draws the rectangle with four colors shown on the right. The line width is 3. The first line starts at 120e hex milliseconds, the last line at 4353 hex milliseconds.

```
ec1
1016
734
Macro #3
ff000000
0$Nop$created 03-11-19 17:34:18 GMT+01
…
120e$Form$Line$78$36$7a$36$ffffffff$3
1222$Form$Line$7a$36$7b$36$ffffffff$3
…
22ac$Form$Line$dd$35$de$38$ffffff00$3
22c0$Form$Line$de$38$de$39$ffffff00$3
…
302c$Form$Line$db$90$d8$90$ff00ff00$3
3036$Form$Line$d8$90$d7$90$ff00ff00$3
433f$Form$Line$70$91$70$8f$ff0000ff$3
4353$Form$Line$70$8f$70$8d$ff0000ff$3
…
5f54$Nop$end 03-11-19 17:34:43 GMT+01
```

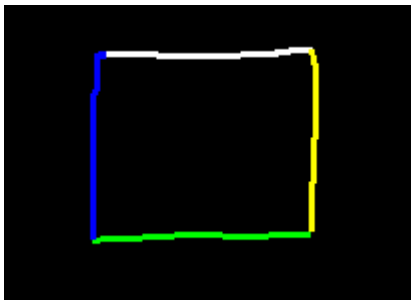

Figure 4.8 E-Chalk macro for a sketch of a square

When the macro is played, the lines are drawn smoothly, as when the macro was first defined.

It is also possible to include other type of events in macros. One important event is pasting text on the screen. Individual characters are pasted one after the other, and the sequence is opened with a $Form$Text$ command. The characters follow, each at a specific time, and the whole sequence is finished with a $Text$End$ command. In the example below, the square has been embellished now with the word “square” to the right.

```
100$Form$Text$$f6$5a$ff00ff00$14
6d2$Text$Char$s
7b7$Text$Char$q
846$Text$Char$u
9be$Text$Char$a
10de$Text$Char$r
11fe$Text$Char$e
11ff$Text$End$square
```

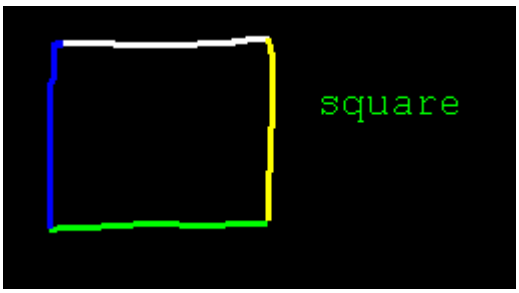

Figure 4.9 A square with text.

It is now very easy to take the next step. An algorithm can be animated by inserting the appropriate events in the source code of the algorithm. This can be done in any programming language, as shown in the next section.

4.7 Animation with sequences of stills

We can now produce our first E-Chalk animation using programs that produce the E-Chalk events. For the first experiment we will use Matlab code, extended with write instructions that produce the appropriate animation strokes.

Figure 4.10 shows the result of running the bubble sort algorithm in Matlab. The Matlab code was annotated to produce the header of the macro and each of the strokes, as well as the text. The screenshot shows each one of the ten iterations of the algorithm, as columns, the displacement of the element being compared (red line) in each iteration, and which elements have been already sorted (in green). The magnitude of an element of the array is represented by the length of the stroke.

When the macro runs, each array appears one after the other (not all at the same time), so that the viewer can see the algorithm running and going from one iteration to the next. It is not the same visual impression as when an array is being sorted in-place, but the history of the algorithm remains on the screen and can be reviewed later at a glance.

The animation has not been heavily annotated with text, and it would be preferable to avoid any lettering, because the intention is to use these animations during a class. The lecturer should do the labeling and annotations to the animation, so that there is a more direct interaction between lecturer and blackboard. When the lecturer writes her annotations, she explains the algorithm, and this reinforces the visual impression from the animation, as postulated by the dual channel coding theory.

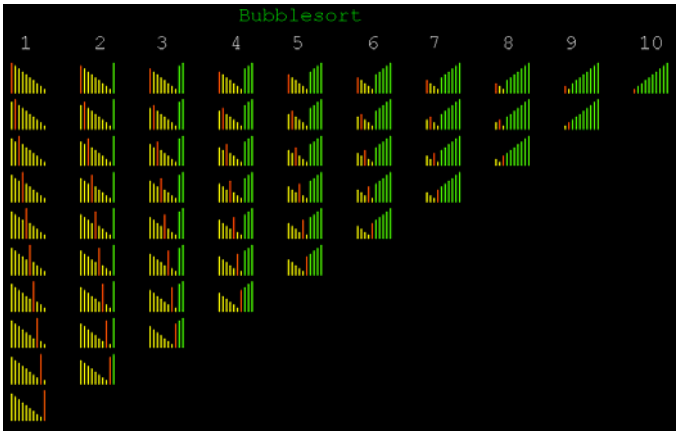

Figure 4.10 The history of ten bubble sort iterations. The pivot has been colored red, sorted numbers green.

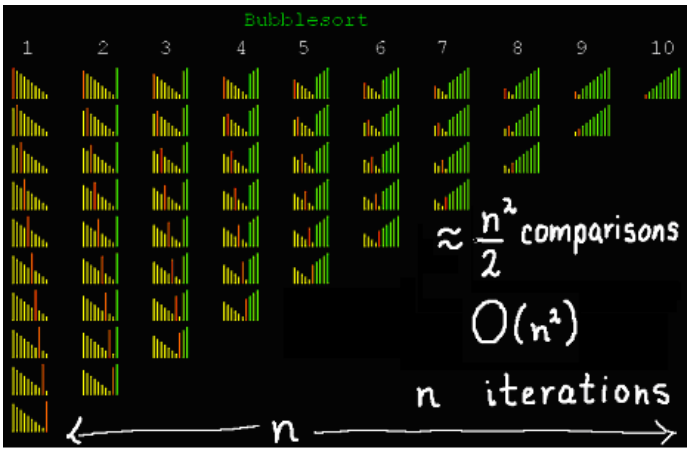

Figure 4.11 Discussing the complexity of bubble sort

The code inserted in the Matlab definition of bubble sort has the following form:

```
x=dec2hex(95*it+5*j);
y=dec2hex(100+i*50+50);
y2=dec2hex(100+i*50-4*a(j)+50);
tt=dec2hex(time+1800+(i-1)*20);
fprintf('%s$Form$Line$%s$%s$%s$%s$ff55ff0%i$2\n',tt,x,y,y2,x4,1);
```

The coordinates for the line to be drawn are computed, also the time at which the line should appear, and then the $Form$Line$ command is written to the macro file. The *it*, *i*, and *j* variables define the iteration (two loops) and the array element we are painting. The constants were calculated to allow displaying the whole table in one screen.

Of course, the whole algorithm instrumentation could be made easier by defining Matlab functions which take care of the details. This was not done for Matlab, but for Java, as will be explained in the next sections. The Matlab experiment was

performed to illustrate macro animation in the E-Chalk system, which is language independent. Any high-level language can be used to generate macros.

4.8 Overlaying images – pseudoanimation

It is also possible to produce an in-place animation of an algorithm in E-Chalk. For this, the macro file should just avoid translating the array on the screen. The array is repainted after every iteration (covering the previous strokes with black). Since in E-Chalk the velocity of the macros can be regulated, it is possible to play a sequence of images at a rate that gives the illusion of a smooth animation.

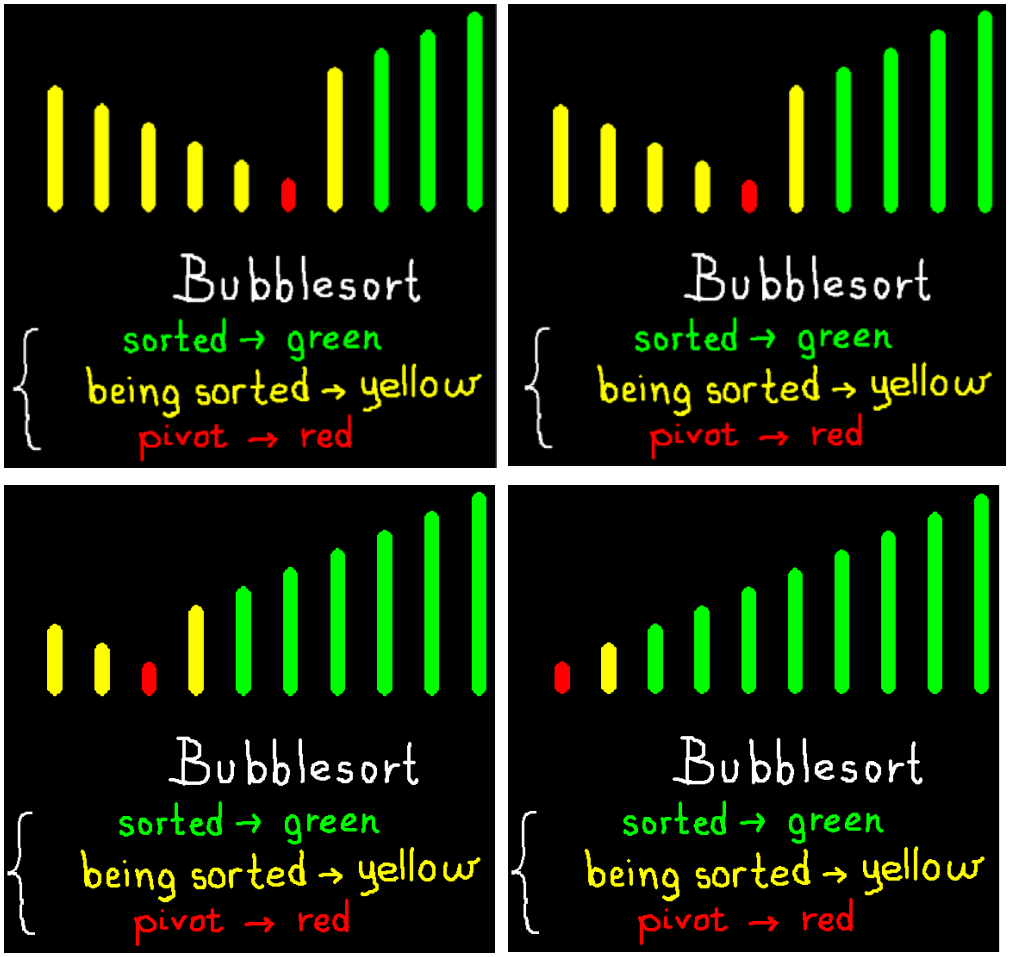

Figure 4.12 In place animation of bubble sort in E-Chalk. The four screenshots show four different temporal states of the animation.

Figure 4.12 shows four screenshots of a running animation of bubble sort. The numbers in an array are represented by the height of the bars. The pivot of the previous iteration is shown in red (possibly after having been exchanged with a

larger element). The unsorted numbers are shown in yellow, and the numbers sorted in the previous pass are shown in green.

The animation was produced with few annotations in the Matlab code, which is essentially the same as was used to produce the animation of the history table of the execution of the algorithm. The lines below show the code for painting one of the number bars. The code is inserted in the main execution loop.

```
x  = dec2hex( 100+50*j );
y1 = dec2hex( 300 );
y2 = dec2hex( 300-20*a(j) );
tt = dec2hex(time+1800+(i-1)*20);
fprintf('%s$Form$Line$%s$%s$%s$%s$00000000$10\n',tt,x,y1,x2,100);
fprintf('%s$Form$Line$%s$%s$%s$%s$00ffff00$10\n',tt,x,y1,x2,y2);
```

As can be seen from Fig. 4.12, the lecturer has done the textual annotations herself. Before the animation runs, the lecturer can explain the coding to be used and annotate the blackboard. Then the macro is started and the lecturer provides an explanation. The macro can be stopped at any time by pressing a key on a wireless keyboard.

Our experience in the classroom with this kind of algorithmic animations is that embedding them in the electronic blackboard provides a feeling of immediacy. Since the appearance of the animation is made to look like a sketch, the lecture seems more spontaneous and students are driven to ask more questions. There is a community within the computer graphics field that has actively explored naturalistic and non-photo realistic rendering in the last few years. The animations possible with E-Chalk go in this direction, but as we will see in the next chapter, there is still room for additional improvements.

4.9 Audio for the animations– explanatory narration

The E-Chalk system allows the producer of a lecture to store the signal from a microphone when an animation is running. This is relevant for us because algorithms taught in class with an animation automatically become a narration. It is possible to bring the animation on-line for viewers from the Internet. We already saw in Chapter 2, that some algorithm animators consider sound a valuable and important addition to the visual material [Brown 98d].

Psychological research has confirmed how important a synchronized narration is for understanding algorithms. Mayer and Anderson, for example, conducted experiments with a simple learning task: students had to learn the operation of a tire pump and a brake system with the help of a multimedia system. In one group, an animation was shown, and a narration was provided before, or after the animation. In another group, the animation was shown coupled with the narration. The later

group performed better in problem solving tests, although there was no difference in memory retention tests. The authors conclude that images and words are most effective when used contiguously in time and space [Mayer 92]. This shows that the concurrent presentation of audio and animations helps the student to build references between both, which provide an advantage later in creative problem solving [Mayer 91].

However, one word of caution is needed here. When the narration and the material on the screen compete for the attention of the viewer, this can lead to a split-attention situation, in which the actual assimilation of the material is impaired. The best effect is obtained when the narration and the action on the screen are correctly synchronized. What is being narrated should be visible on the screen [Moreno 02] and should not confuse the viewer. This is a problem of staging which cannot be solved automatically by a computer but where we rely on the experience of the lecturer.

4.10 Python as pseudocode language

In the rest of this chapter, the programming language Python will be used as the implementation platform for algorithms. There is an important reason for this: the syntax of Python is very similar to the kind of algorithmic pseudocode used in [Cormen 90]. Several algorithms from this standard book can be transformed into Python with small effort.

Python is by now a relatively old programming language, but from its origins of not being a not very well-known system its use has increased significantly in the last years. Python was first defined in 1990 by Guido van Rossum, who wanted a scripting language for the Amoeba distributed operating system developed at CWI in Amsterdam. Van Rossum's intention was to have a simple language with a clean syntax. In this endeavor he was influenced by the programming languages ABC and Modula 3. The first release of Python was posted in February 1991 and since then the language has evolved, adding new capabilities, but preserving its original simplicity. Python is an interpreted language, which can be used interactively, in a similar way to Matlab. Python is object oriented and can call external libraries.

An example can illustrate the similarity between Python and the pseudocode used in [Cormen 90]. The example was written by Pai Chou for a Web page about " Algorithm Education in Python." To the left, we see the definition of the insertion sort algorithm as in [Cormen 90], to the right, equivalent Python code.

```
Insertion-Sort(A)
1 for j <- 2 to length[A]
2   do key <- A[j]
3      i <- j - 1
4      while i>0 and A[i]>key
5            do A[i+1]<-A[i]
6               i <- i - 1
7      A[i + 1] <- key
```

```
def InsertionSort(A):
1 for j in  range(1,len(A)):
2    key = A[j]
2    i = j - 1
4    while(i>=0)and(A[i]>key):
5         A[i+1] = A[i]
6         i = i - 1
7    A[i+1] = key
```

Figure 4.13 Comparing pseudocode with Python code

As can be seen, there are no begin-end blocks in Python. Indentation groups statements in blocks. A difference to the pseudocode is that Python arrays start with the index zero, and this has to be taken into account when using the length of an array as a parameter (as in the algorithm). Other than that, we can see that there is almost a one to one correspondence between both types of coding. Python has, of course, many other instructions that go beyond pseudocode, but for the purpose of algorithmic animations we are interested only in the similarity to pseudocode.

The next example is Quicksort in-place (code below by David Eppstein). Quicksort operates with two pointers that traverse the array in opposite directions, moving elements lower than a key towards the lower indices, elements greater than a key to the larger indices. This algorithm was instrumented to produce the E-Chalk animation discussed in section 4.12.

```
def qsort(L):
   quicksort(L,0,len(L))

def quicksort(L,start,stop):
    if stop - start < 2: return
    key = L[start]
    e = u = start
    g = stop
    while u < g:
          if L[u] < key:
                    swap(L,u,e)
                    e = e + 1
                    u = u + 1
          elif L[u] == key:
                    u = u + 1
          else:
                    g = g - 1
                    swap(L,u,g)
    quicksort(L,start,e)
    quicksort(L,g,stop)

def swap(A,i,j):
    temp = A[i]
    A[i] = A[j]
    A[j] = temp
```

Python has been very popular for writing scripts and part of the Google code has been developed in this language. Now, Python is also gaining increasing acceptance in education as a first programming language and it is therefore very interesting as a platform for algorithmic animation. Another language which can be also used for algorithmic animation is Haskell. Instrumenting a program in Haskell is non-trivial, due to its declarative functional nature, but one of the examples in the next section was produced using Haskell.

4.11 A scripted animation language

In order to minimize the effort needed to generate an animation for E-Chalk, I defined a new scripting language for algorithmic animation. The script language has been defined primarily with Macromedia Flash productions in mind, and therefore I call it *Flashdance*, true to the tradition of naming algorithmic animation languages after a kind of dance.

The algorithms discussed in the next section were produced by generating a Flashdance script. The script is then processed by an interpreter written in Java which produces the appropriate E-Chalk macros. The macros can then be played in E-Chalk during a lecture. It would be possible to play the animations in any other animations system (Quicktime, a Java Applet). In that case an interpreter for the script language has to be written, which produces the appropriate Quicktime or Java commands.

Flashdance will be extensively covered in Chapter 6. Here, I only list the commands available in the macro language.

The commands have a name, reproduced in bold face, and arguments. Optional arguments are enclosed in square brackets.

new *object-type object-name x y width height* [*label*] [*colour*]

View *view-name x y width height* [*background-colour*]
String *object-name x y width height* [*label*] [*colour*] [*style*]
Line *object-name* x_1 y_1 x_2 y_2 [*line-width*] [*colour*]
Rectangle *object-name x y width height* [*line-width*] [*colour*]
Oval *object-name x y width height* [*line-width*] [*colour*]

remove $object\text{-}name_1$ … $object\text{-}name_n$
removeAll
change *object-name* $property_1$ $value_1$ $property_2$ $value_2$ …
exchange *object-name object-type* [*label*]

moveTo *object-name$_1$ x_1 y_1 . . . object-name$_n$ x_n y_n*
animTo *object-name$_1$ x_1 y_1 . . . object-name$_n$ x_n y_n* [*path*]
highlight *highlight-type object-name$_1$ . . . object-name$_n$*
swap *object-name$_1$ object-name$_2$* [*path*]

setView *view-name*

setTime *interval*
stop [*label*]

See section 6.5 for a complete description of the Flashdance language.

For the animations in section 4.12, I only had to use the “new Line” command in order to generate points and lines. Points were generated as lines going from one point on the screen to itself, with a certain appropriate line width. The command “swap” was used to generate the animation of an element swap. In the macro event code for E-Chalk, the swapped data points are erased and painted again at the new location.

4.12 Examples

In this section I will describe some algorithms which were implemented in Python, producing scripted code that was then transformed into E-Chalk macros. The macros have been collected in a library of illustrative examples which can be used during a class with the electronic chalkboard. The lecturer does not have to change the user interface metaphor; all animations are started from within E-Chalk.

4.12.1 Seeing Quicksort

My first example is Quicksort. The code is the one described in section 4.10. The actual code produces two macros, one which draws the initial data, enclosing it in a frame. Figure 4.14 shows an array, with the index going from 0 to 99 in the horizontal direction. The distance of a point from the horizontal axis represents the magnitude of the number stored in the array. The numbers were selected randomly in the range 1 to 100. When the lecture about Quicksort begins, the lecturer can first let the data be drawn.

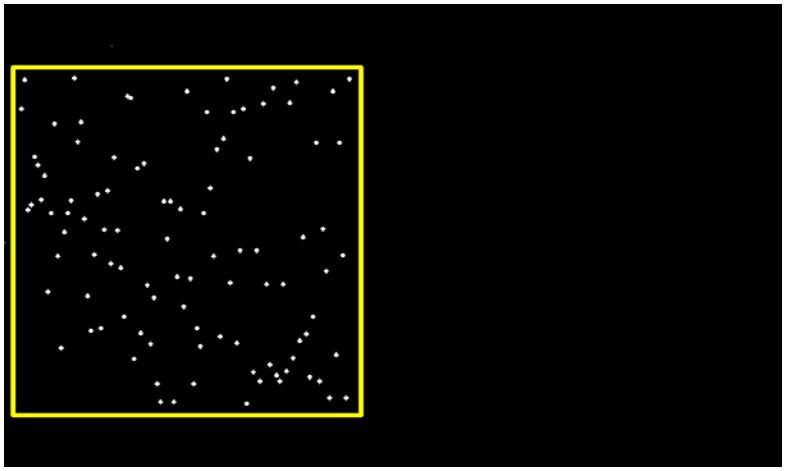

Figure 4.14 A macro draws the original unsorted data. The horizontal direction represents the position in an array of 100 elements. The vertical height is proportional to the value of the element at each position in the array.

The next screenshot, Figure 4.15, shows how the lecture has progressed. The lecturer writes the name of the algorithm, and explains the data. To the left a small example of the recursion used in Quicksort has been drawn. The code shows that the initial list is partitioned into two lists, the lists of elements smaller than the pivot, and the list of elements larger than the pivot. Two recursive calls sort the data, which provides the final answer after appending the lists. In the case of in-place sorting, the append operation is implicit in the calls.

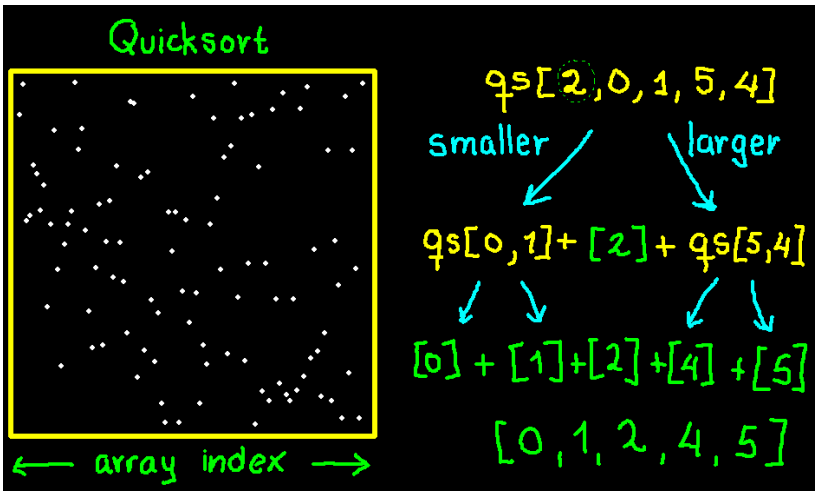

Figure 4.15 The lecturer has annotated the blackboard.

The next screenshot, Figure 4.16, shows the state of the electronic chalkboard once the second macro is started. This macro erases and repaints the swapped elements of the array. The sorted elements are painted in green. One can see that the algorithm has more or less grouped the input, and that the first array elements are already sorted, that is, they are green.

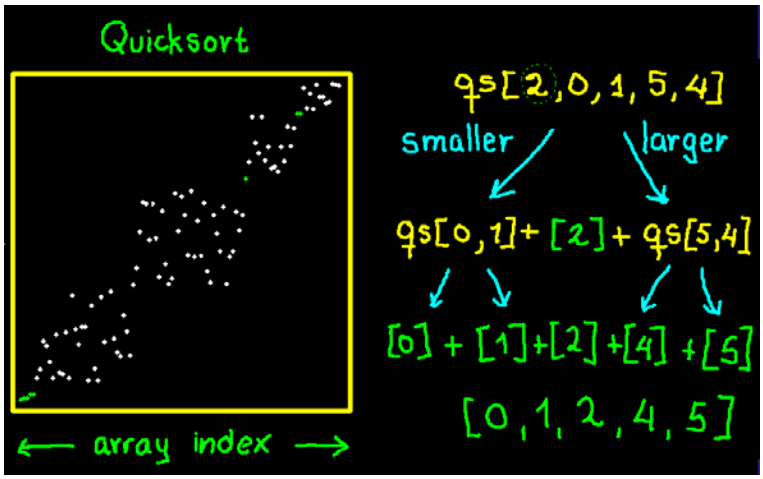

Figure 4.16 Progress of the animation

In Figure 4.17 (left) we can see the further progression of the animation, now almost all numbers have been sorted. Figure 4.17 (right) shows the final result.

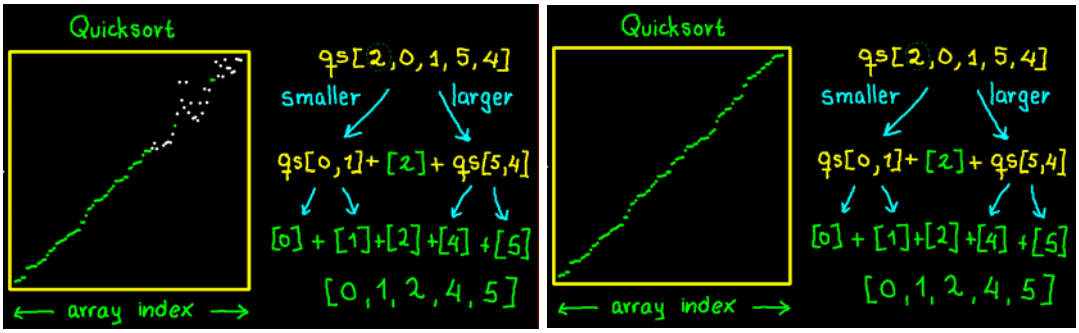

Figure 4.17 The animation ends, the data has been sorted.

The lecturer can now offer several more examples: Figure 4.18 shows a worst case for Quicksort: an ordered list of numbers. The screenshot on the right shows the slow progression of the algorithm. If the algorithm is too slow, the macro can be stopped by the lecturer by clicking on the screen.

Now, the lecturer can ask an interesting question, namely, which numbers have been swapped to which array positions. An animation with an overlay can be run. The overlay shows all array positions and which numbers have occupied them during the algorithm run. For the worst case example, the pattern shown in Figure 4.19 provides the answer. One can see that there are some repetitive patterns, which better illustrate the algorithm dynamics. Some array positions are used to store sequential numbers and this yields the vertical lines in the pattern.

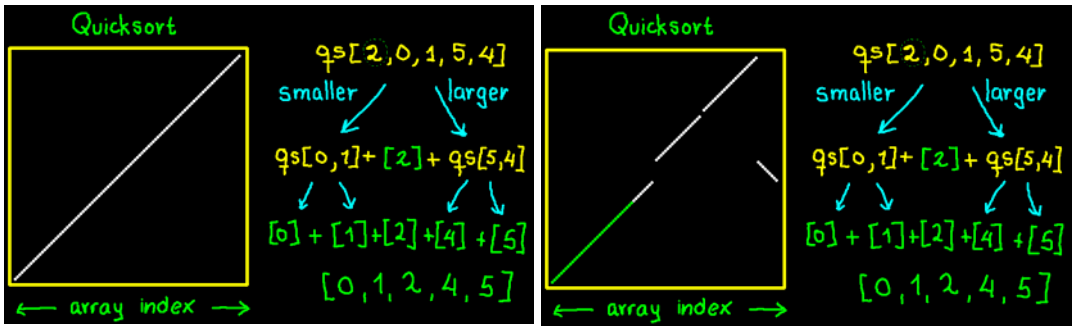

Figure 4.18 One worst case for Quicksort: a sorted list is the input. To the right we see the slow progress of the algorithm.

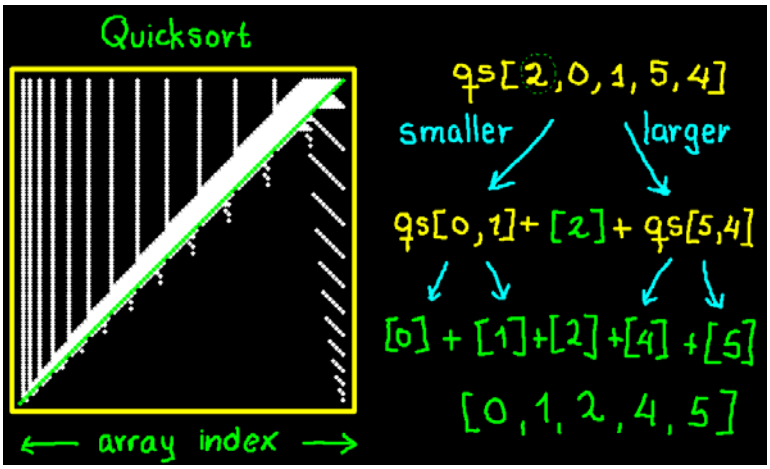

Figure 4.19 All positions occupied by elements in the array have been marked. The white points give an idea of the number of exchanges performed for this data.

In the average case, that is, for randomly selected numbers, the pattern is still more intriguing. Instead of vertical lines, now horizontal lines dominate. Some numbers are copied sequentially through several array positions. The pattern is interesting – it looks somewhat fractal. One can see that the points are organized in boxes of different sizes, a typical visual pattern produced by the Quicksort filter operation.
The most astonishing property of the animation cannot be reproduced with these images: it is the visual surprise of seeing a blackboard come alive to play an algorithm.

The lecture about this specific algorithm can be stored on the Web, and the explanation of the lecturer is integrated with the animation. Any interested student from all over the world can access the animation, hear the explanation, and see the numbers being sorted.

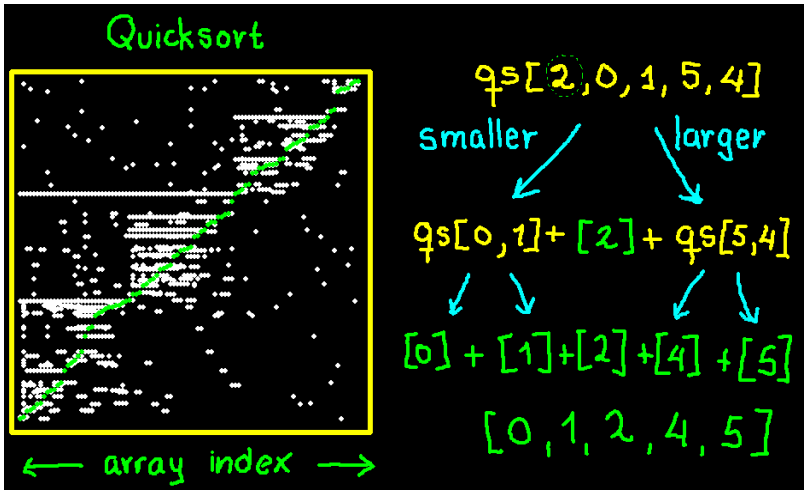

Figure 4.20 History of Quicksort swaps for random data.

The time needed to prepare this animation was negligible: only a frame had to be drawn, and point objects were painted on the screen. The rest, the textual material, was added by the lecturer on-the-fly, during a class, and completes the algorithmic animation.

4.12.2 Convex hull algorithm

The next example is a geometric algorithm for computing the convex hull of a set of points in the plane. The algorithm is quite simple, it is known as the Graham scan. It works by generating a stack of points. Each new point pushed into the stack is tested for membership in the convex hull. If a point does not belong to the convex hull, as evidenced later by the consideration of new points, it is popped out from the stack. A description of the algorithm can be found in [Cormen 90]. The code below, in Python, is from the Python cookbook repository and was written by D. Eppstein. The algorithm computes the upper and the lower convex hull of a set of points. The code is very elegant and short, an excellent example of the pseudocode flavor of Python.

```
def orientation(p,q,r):
    return (q[1]-p[1])*(r[0]-p[0]) - (q[0]-p[0])*(r[1]-p[1])

def hulls(Points):
    U = []
    L = []
    Points.sort()
    for p in Points:
        while len(U) > 1 and orientation(U[-2],U[-1],p) <= 0: U.pop()
        while len(L) > 1 and orientation(L[-2],L[-1],p) >= 0: L.pop()
        U.append(p)
        L.append(p)
    return U,L
```

The algorithm was instrumented and the Flashdance script was generated. From this, the macro file was generated. Figure 4.21 shows the algorithm starting from the left. The upper convex hull is being formed with a white polygon, the lower convex hull with a yellow polygon. At the beginning, only a few points have been tested and the white and yellow polygons almost touch.

Figure 4.21 convex hull algorithm starting in E-Chalk.

Figure 4.22 and Figure 4.23 show the further progression of the algorithm. Lines which have been tested for membership in the convex hull are painted green, using a fine line. As can be seen, both convex hulls are built slowly and many lines have been tested during the computation. Fig. 4.23 shows the final result, further annotated by the lecturer, who has drawn arrows pointing to the upper and lower convex hulls.

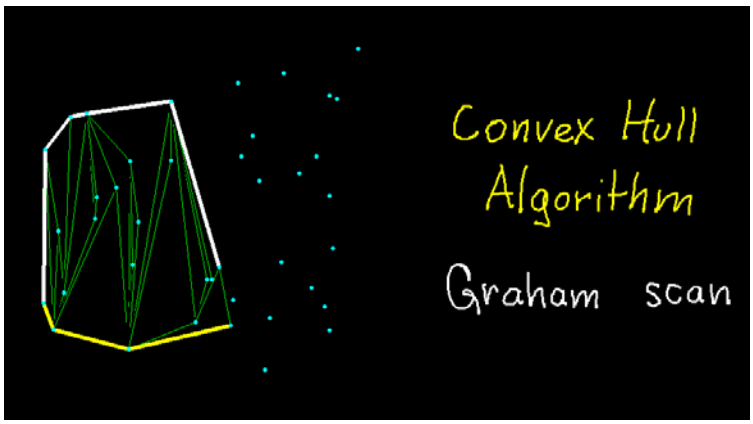

Figure 4.22 Progress of the convex hull algorithm. Lines tested and rejected are shown in green.

Figure 4.24 shows another run of the algorithm, played at the same position on the screen. Now the lines which have been tested have not been highlighted in green. This is an overlay which can be turned on or off when the animation is translated into an E-Chalk macro.

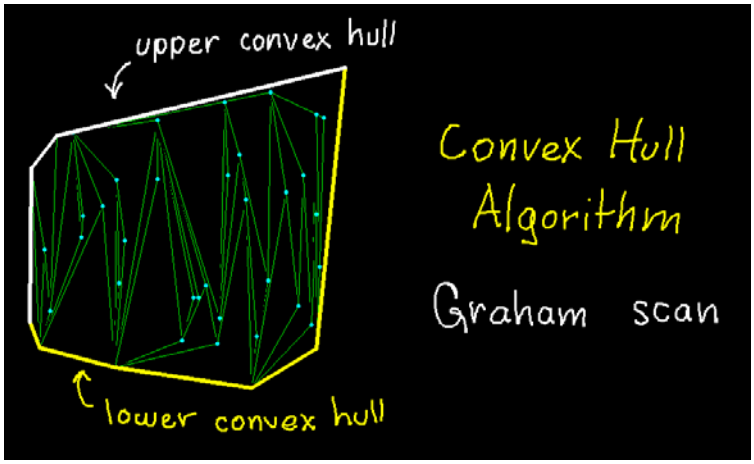

Figure 4.23 The final result. The lecturer has annotated the upper and lower convex hulls.

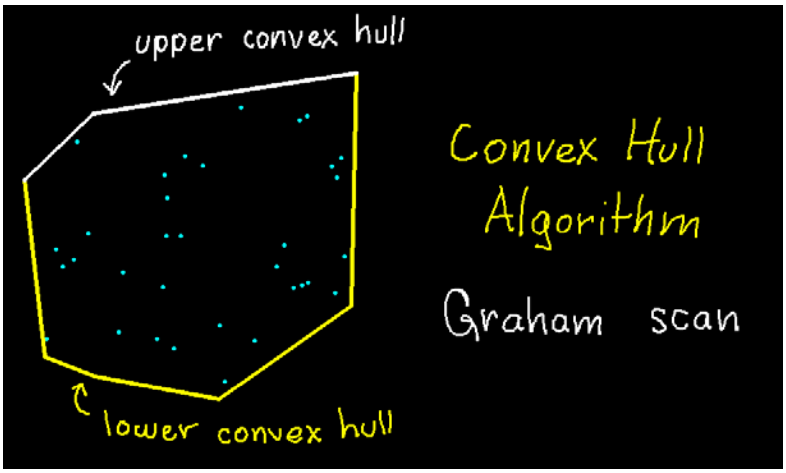

Figure 4.24 Another example where only the convex hulls and the points have been drawn.

4.12.3 The Heapsort algorithm

The last example in this chapter is the Heapsort algorithm. Heapsort builds a max-heap of numbers using an array. The first element (the root) is then taken out of the heap (because it is the maximum number in the array). The rest is heapified and the algorithm continues. The four screenshots shown in Figure 4.25 illustrate a run of the algorithm, when the numbers have been selected randomly between 1 and 100. The visualization used is the one introduced before in section 3.4.

The animation shows clearly, how the heap becomes smaller and smaller. The numbers which have already been sorted are colored green. The heap cloud is restricted to a smaller and smaller rotated square, until all numbers have been sorted.

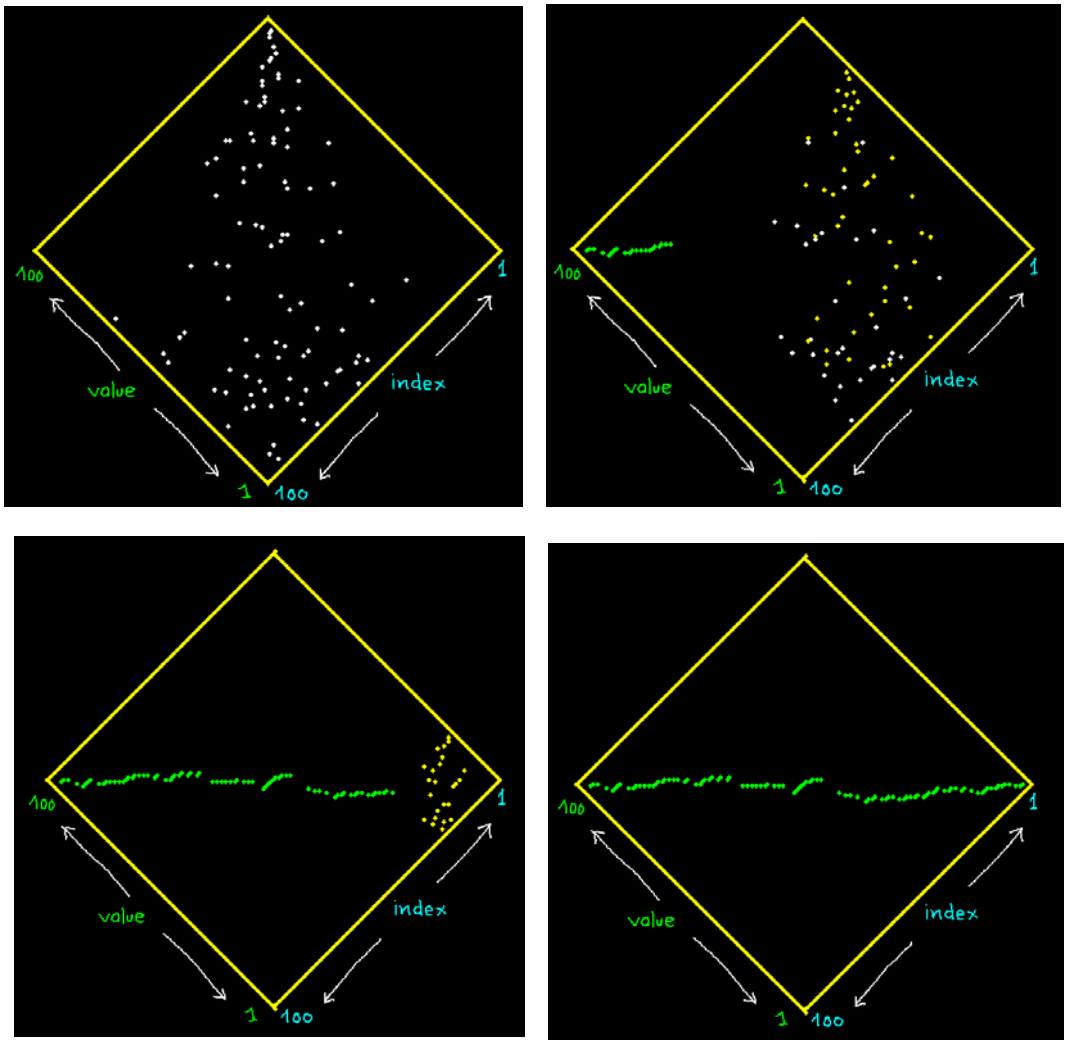

Figure 4.25 The Heapsort algorithm running.

Figure 4.26 shows all numbers which have been stored in every entry of the array. The number of white points is a good indicator for the number of substitutions computed during the sorting algorithm. It is easy to see that the number of substitutions is much lower than order n^2.

4.13 Summary and discussion

In this section I have introduced the E-Chalk system for electronically enhanced classroom teaching. E-Chalk was developed mainly in order to improve classroom teaching. The transmission and storage components are secondary derivatives of the technology. E-Chalk is based on the concept of “intelligent assistants” working in the background, helping the lecturer to deliver her class. The algebraic

server, for example, is used to compute algebraic expressions or to plot functions on demand. The teacher does not have to leave the chalkboard metaphor when using any of these intelligent tools.

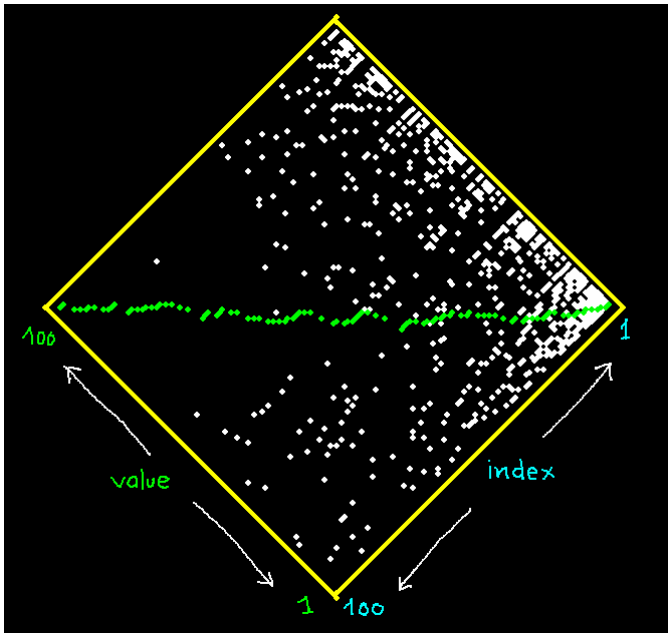

Figure 4.26 A trace of the Heapsort algorithm. All values which were stored at every array position have been marked with a white point. The greatest activity has been registered at the lowest indices.

In this chapter, I have reviewed the type of teaching that can be done with a blackboard. Using a blackboard is an art and not everybody can do it correctly in a class. A static description of an algorithm or process should make visible the main steps involved. I showed that an electronic blackboard makes possible the reuse of diagrams, impossible in a classical blackboard, and that a kind of static animation by reuse of pictures becomes possible. This type of explanation is extremely important in computer science.

I have shown above that real algorithmic animations can be integrated in an elegant way in electronic blackboard teaching. Although E-Chalk has the capability of calling Applets, which are pasted to the blackboard, and these Applets can be algorithmic animations, my approach of generating macros is more natural for a class given with a blackboard. The teacher never changes the interaction interface with the system – it is always a blackboard.

The effort needed to instrument algorithmic animations for E-Chalk, following my approach, is negligible. All textual annotations can be done by the teacher while giving her class. The Quicksort example is very illuminating in this respect. The code had to be annotated just in the swap function and every time a point had

been sorted (to paint it green). Since the teacher writes the rest, the text and diagrams for an explanation, this does not have to be included in the macro. Different experiments can be conducted in the same region of the blackboard or in another, if desired.

I showed also, that there are two possible ways of showing an animation: as an animation of the history of the algorithm or as an animation in-place. Both kinds of animation fulfill different needs. The static animation through a history helps the teacher to slowly go from one step to the next. The student can review the whole history of the animation with her eyes.

I also introduced briefly in this chapter the Flashdance script animation language. Using this script notation, instrumenting algorithms becomes very easy. The script language will be described fully in Chapter 6. The examples described in this chapter were written in Matlab and Python. This shows how easy it is to change the coding language. Python is especially attractive, since algorithms written in Python resemble pseudocode of the type used by [Cormen 90].

The sequences of screenshots presented in this chapter should give the reader an idea of the kind of result obtained from my Flashdance algorithmic animation system for E-Chalk. The visual impression during class is one of surprise and immediate interest. Although macros were initially integrated into the system as a way of storing definitions or small handwritten diagrams, in order to speed up the pace of a lecture when those definitions or diagrams are given. I have shown in this chapter how to exploit the macro event format for the production of algorithmic animations. Animations using this approach have already been produced and have been made available through the Web, together with an oral explanation of the algorithms. With my approach, the lecturer has a partner in the computer, who helps to produce the visual effects. Likewise the lecturer is a partner of the algorithmic animation, because she annotates and embellishes the animation.

In the next chapter, I will show that the level of coupling between lecturer and computer can be further increased by allowing the user to enter her own data in handwritten form, even the code of an algorithm in a simple language.