5 Interactive Computer Driven Animation of Sketches

In the previous chapter we saw how to generate geometric objects, such as bars, boxes or points, using the E-Chalk macro facility. While the annotations have the look and feel of the blackboard, the geometric objects are too “perfect”. We would like to preserve the “sketchy” character of the animations drawn on the blackboard. Sketches have an aesthetic appeal by themselves and properly used, let an animation look like magic.

The approach taken in this chapter is to let the computer ask the user for the image of the elements involved in the computation. In a graph algorithm, for example, the computer prompts the user for the drawings of the nodes and edges of the graph; then these images are used by the computer to produce its own rendering of the animation.

5.1 Sketches are esthetically appealing

Some researchers have explored the use of sketch animation techniques for innovative human computer interfaces. Thomas and Calder, for example, proposed several ways of handling menus and windows in a graphical user interface adopting some techniques from cartoon drawing [Thomas 01]. Frequent operations, such as dragging a window, making it larger, or opening a menu, were animated in such a way that the virtual objects seemed to oppose change. A window being dragged, for example, deforms elastically and the point of contact with the mouse is stretched. A menu which is opened does not appear instantly, but grows from a small rectangle to a large rectangle containing the menu options. A window being enlarged deforms like an object being pumped with air. The idea of the authors is that these techniques reinforce in the viewer the illusion of reality of the virtual objects because they behave like we would expect in reality. Sketches also have an intrinsic appeal because they lead to a stimulating esthetic experience [Strothotte 97, Deussen 00].

Thomas and Calder evaluated their new GUI using a form based questionnaire. Students seemed to enjoy the animations, for its novelty, and eventually used a setting for a drawing editor in which the animation was included. Although an-

imations should not be abused, the conclusion of this study is that sketch animation techniques have something to offer for human-computer interaction because of their esthetic appeal for the user. Also, producing such animations is not very difficult: sketches can be animated by providing the start and the end view, and computing the frames in-between [Kort 02].

5.2 Animations with sketched input

In order to test the above idea, I developed two animations systems: one based on E-Chalk, the second based on Macromedia Flash. In this section I will first describe the Flash prototype, because it is simpler and it shows the general technique used later in the full blown E-Chalk system.

Flash is an animation platform which has been in the software market for many years. It is a proprietary system from Macromedia, but it is also a de facto standard for high-quality animations in the World Wide Web. Animations can be hand-crafted by using the Flash animation editor, but there is also a script animation language that allows a user to produce her own programmed animations. The animation language is called ActionScript. A more detailed description of Flash and ActionScript will be provided in the next chapter. Here, we only need to know that Flash is a general purpose programming language with a built-in animation engine, which can be used to design graphical human-computer interfaces.

We saw in the previous chapter that animations can be synthesized for the E-Chalk system. A program is instrumented with extra code that produces an animation script, which in turn can be transformed into an E-Chalk macro. However, the input to the algorithm is given in numerical form or is fixed in advance. We would like the user to be able to provide her own input and be able to do this without leaving the chalkboard metaphor. The lecturer should be able to illustrate algorithms directly in the blackboard, handwriting or sketching the input. No algorithmic animation system provides this capability, only Hundhausen has tested the use of a graphical editor to sketch the input and then animate the resulting drawings [Hundhausen 00, 01]. However, editor and animator are separate entities and there is, as said in German, a “Medienbruch”: the lecturer abandons the teaching media for a moment, to use a secondary tool. This distracts the students.

In order to test the impact of lecturer provided input, in handwritten or sketch form, I wrote an E-Chalk clone in the Flash ActionScript language which I called my-E-Chalk. The clone provides the basic functionality of a drawing tool, without the additional features of E-Chalk (handwriting recognition, Internet calls, automatic transmission and recording of streams, etc.) My E-Chalk clone has the appearance of a blackboard, with an activation console for the animations. Figure 5.1 shows a screenshot of the my-E-Chalk drawing canvas. Different colors and

line widths can be used to draw on this canvas, the whole screen can be erased, and strokes can be undone.

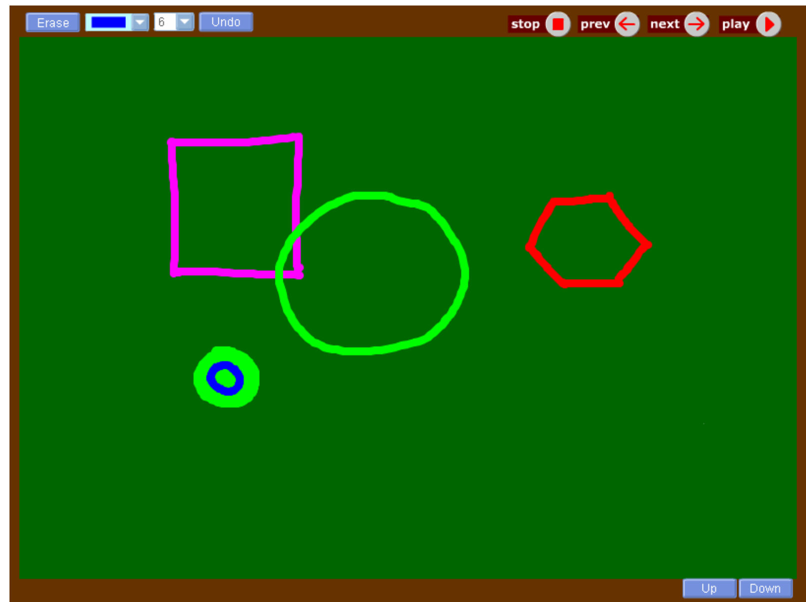

Figure 5.1 The drawing canvas and console of my-E-Chalk.

Some algorithms were written in the ActionScript language. They can be started using a menu option. Once started, the input that arrives in the form of strokes is interpreted by the algorithm as being the animation input. When the input has been drawn by the user, the animation console allows the lecturer to start the animation. The console has the buttons shown in Figure 5.2 An animation be started pushing the “play” button, it can be stopped (“stop”), it can be reversed (“prev”) or it can be played forward, step by step (“next”).

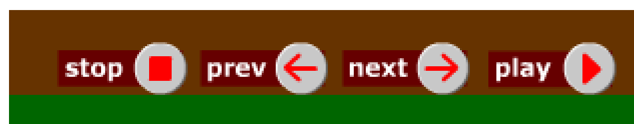

Figure 5.2 The animation console.

Fig. 5.3 shows an example for the classical bubble sort algorithm. The algorithm draws a line for the array, and sets two pointers to the beginning of the array. The input is entered in the form of vertical lines. The magnitude of the numbers in the array is proportional to the length of the lines. This is a very common visual metaphor for sorting algorithms. Figure 5.3 is a composite of four screenshots. The animation starts with the input in the upper left screenschot, continues to the right, and then on the next line. The animation is finished in the lower right.

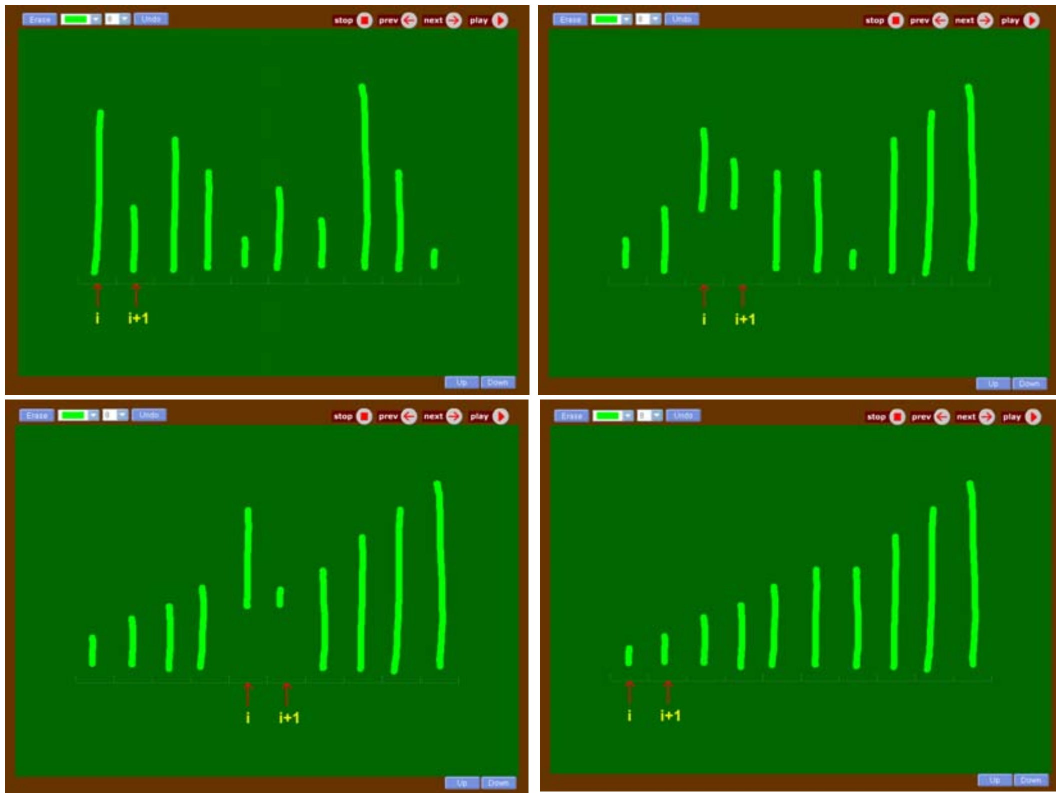

Figure 5.3 Animation of the bubble sort algorithm with user provided sketched input

Figure 5.3 also shows how highlighting was implemented: when two numbers are compared, they are moved upwards a few pixels, and then placed in position again. The indices move from left to right and show which elements are being processed.

The beauty of this scheme is that many different kinds of sketch inputs can be handled with the same code. Figure 5.4 shows an example where the input has been entered as wiggled lines.

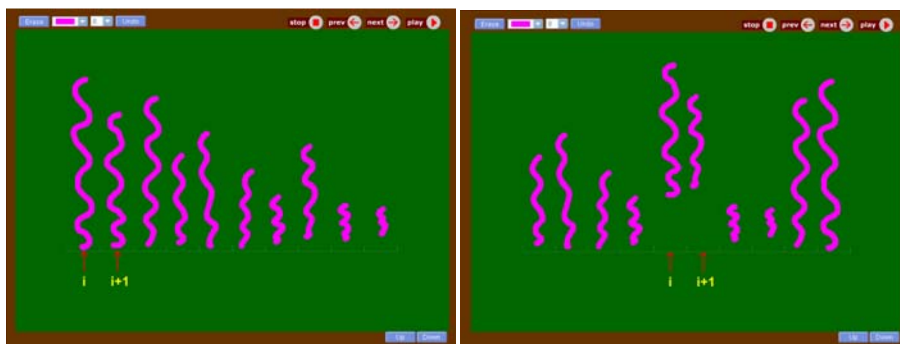

Figure 5.4 Wiggled lines as input to the animation.

Figure 5.5 shows an example where the drawing has a semantic corresponding to the sorting problem: the letters have been drawn with a height proportional to their position in the alphabet. When the animation runs the size perception is reinforced by our unconscious reading capability.

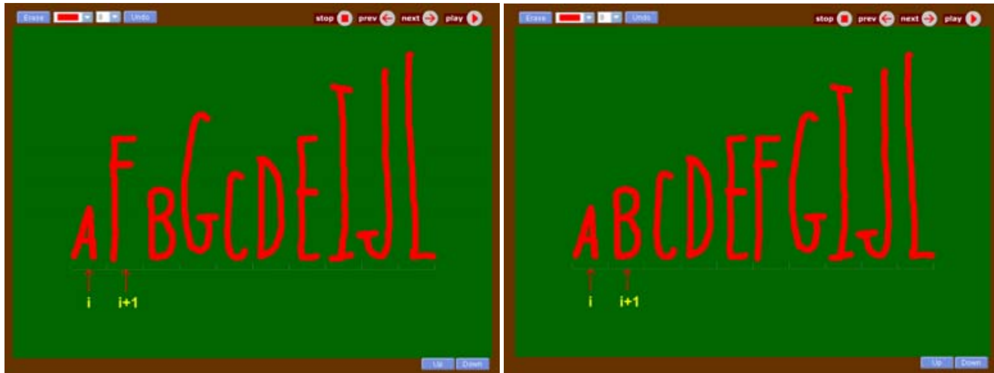

Figure 5.5 Sorting letters in my-E-Chalk.

Figure 5.6 shows an animation tested during a course for high-schools students at the FU Berlin. The animation engaged the students because of its esthetic appeal. Capturing students interest is the most important aspect of algorithmic animation, as has been shown by educational tests [Grissom 03].

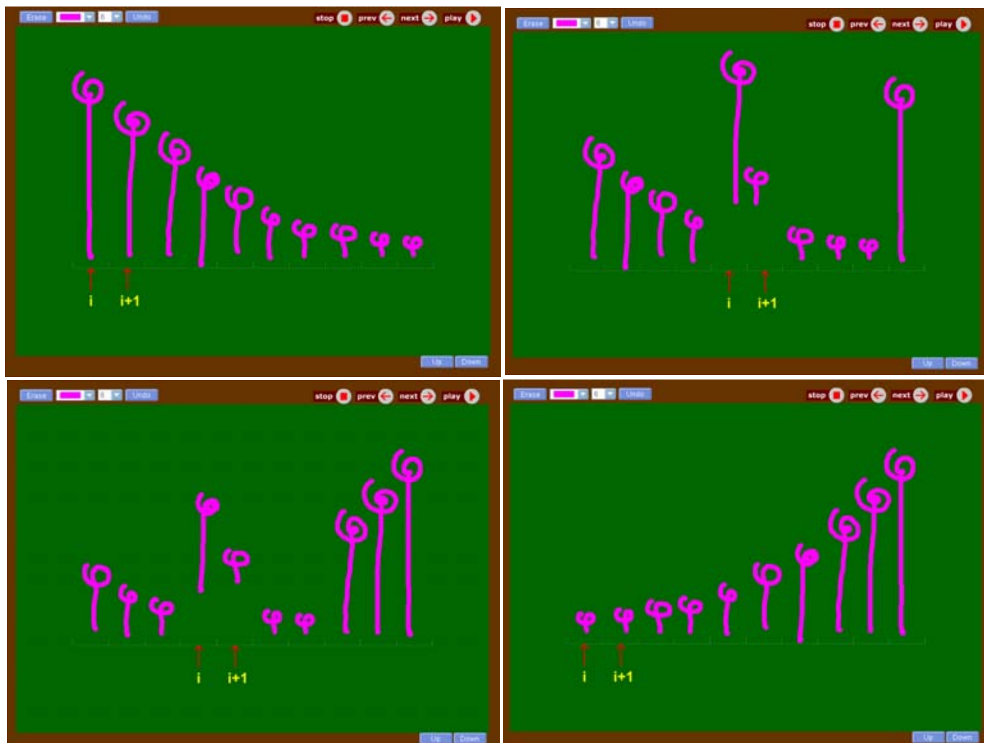

Figure 5.6 Animation of bubble sort produced in a lecture.

Finally, the last example shows again the reinforcement of the visual size impression with an input consisting of handwritten numbers. The animation, when it runs, is meaningful and the sorting method is easy to understand at every step.

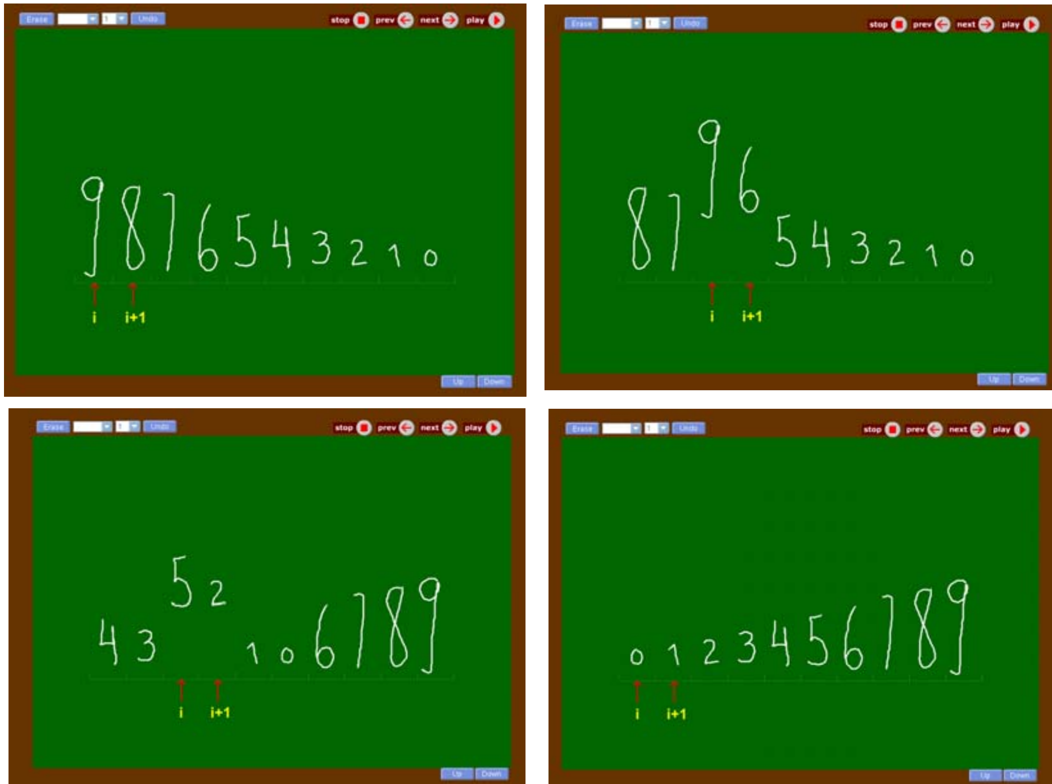

Figure 5.7 Sorting handwritten numbers in my-E-Chalk.

What these examples show, is that an animation which is started from the chalkboard, with user sketched input, can more easily attract the attention of students during a class. The chalkboard seems to be “intelligent”; it interprets the user input and handles the data immediately. There is no need to open another window or to start another tool.

The next step is to transport this technique to the E-Chalk system, where we can also make use of the handwriting recognition features already integrated into the system. Using handwriting recognition, sorting of letters or numbers can be done without having to encode the letter or number order in the size of the input. But before considering the way animations with user sketched input were implemented in E-Chalk, let us consider the problem of coding on the blackboard in the next section.

5.3 Executing a programming language in E-Chalk

In this section I describe a further educational innovation implemented for the E-Chalk system as part of this thesis. I wrote a simple interpreter for the programming language BASIC, and the interpreter was coupled with the handwriting recognition machinery of E-Chalk. The lecturer can now write a BASIC program on the chalkboard and request its immediate execution. The electronic chalkboard complies. This implementation provides a glimpse of what will become possible in the future.

5.3.1 Motivation for handwriting-programming

The authors of the E-Chalk system have written that their main motivation in developing an electronic blackboard was to provide an adequate teaching medium for disciplines such as mathematics or physics, which make heavy use of the traditional blackboard. Computer science could also profit if it could be possible to write a program on the electronic board and let it execute. This would allow the teacher to provide the students an immediate example of some algorithms.

A handwriting recognizer is needed, if code written on the electronic blackboard is to be executed. The E-Chalk handwriting recognizer is a pattern recognition system developed by Ernesto Tapia at the FU Berlin [Tapia 02].

Symbols are recognized by processing line strokes and extracting relevant features. Such features are, for example, the length of the line stroke, its centroid (in a unitary square), the distance between the first and the last point of the stroke, divided by the total length. Also, the coordinates of a few points along the stroke can be added to the feature vector. The most relevant points for the shape are selected using a shape simplification algorithm. Once a symbol has been transformed into a feature vector, it is given to a neural network or support vector machine, which has been trained previously to recognize this symbol. The recognition itself is user independent, but this depends on the quality of the database used for training the models.

5.3.2 Tiniest BASIC

As a proof of concept for a programming system based on handwriting recognition, I defined a minimum subset of BASIC, which is nevertheless general purpose and universal. BASIC has always been a popular language for education in schools (this was the whole purpose of the language when it was defined by Kemeny and Kurtz) and Tiny BASIC variations were written for the first microcom-

puter systems which appeared in the 1970s. My own version of BASIC is called *Tiniest BASIC* and consists of only two interpreter commands and six types of instructions. A definition of the language follows.

Variables are denoted in Tiniest BASIC by a single letter (A, B, C, etc.) and are of floating point type. A Tiniest BASIC code line begins with a line number and contains one of six possible instructions. The instructions (and examples) are the following:

LETI

Command used to set a variable to a constant value.
Example: 10 LETI A = 1

LET

With this command, an addition, subtraction, multiplication, or division of two variables can be performed. Only one operation per line is possible.
Example: 20 LET B = A + C

PRINT

Print the value of a given variable.
Example: 30 PRINT A

GOTO n

Transfers execution to line *n*.
Example: 50 GOTO 10

IF <var> **n**

Transfers control to line n, if the variable <var> is greater or equal to zero.
Example: 80 IF A 30

STOP

Stops execution of the program

As an example, the program below adds the two numbers 3 and 5 and prints the result, before stopping.

```
10 LETI  A = 3
20 LETI  B = 5
30 LET   C = A + B
40 PRINT C
50 STOP
```

When the interpreter is started, any line beginning with a number is treated as code. If a line has the same number as one previously entered, the new line supersedes the old one.

The two instructions “RUN” and “LIST” can also be entered. RUN starts the program at the first line of code. LIST provides a listing of the current code lines. Non-trivial programs can be written with this language.

Tiniest BASIC was implemented in Java. The goal was to have a nontrivial programming language, with a simple syntax and easy to edit using the electronic chalkboard. The instructions are entered using handwriting recognition and the program can then be started. The result of a program run is given back as an ASCII string. There is a timeout for the maximum execution time. A program which does not terminate in the allotted time returns the ASCII string “timeout”. This avoids having to wait for a program which gets trapped in an infinite loop.

5.3.3 Tiniest BASIC execution

Figure 5.8 is a screenshot of the Tiniest BASIC interpreter running in E-Chalk. The program given above was entered by hand and then the command RUN was written. The handwriting recognizer was set to work with the lower-case equivalents of the different commands.

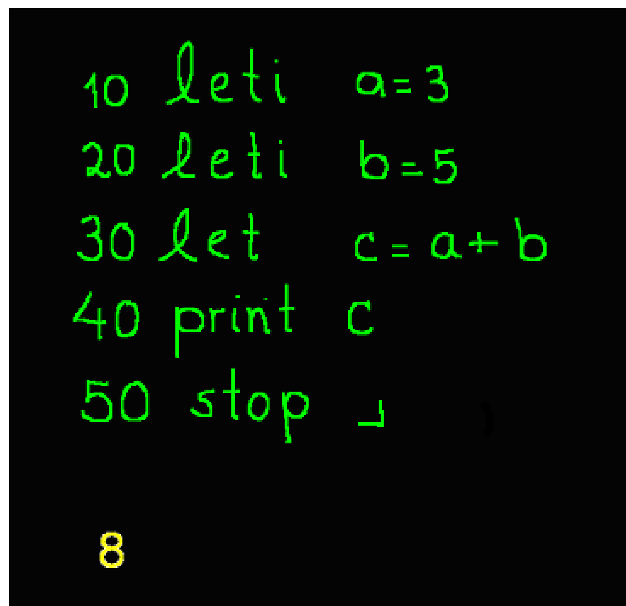

Figure 5.8 Screenshot of handwritten code which has been executed.

As can be seen from the screenshot, a line with an angle closes the handwritten input. The lines are recognized, and they are passed to the Tiniest BASIC inter-

preter. This executes and provides a string as result, which is passed to E-Chalk for display using typewritten output.

Figure 5.9 shows JMATH, the program developed by Ernesto Tapia to train the character recognizer [Tapia 03b]. A Tiniest Basic program has been entered. Each character is surrounded by a box and an identifier of the character which has been recognized. After writing the “LIST” command, the user closes the input by pressing on the B button (BASIC). The window to the lower left shows the output of my Tiniest BASIC interpreter.

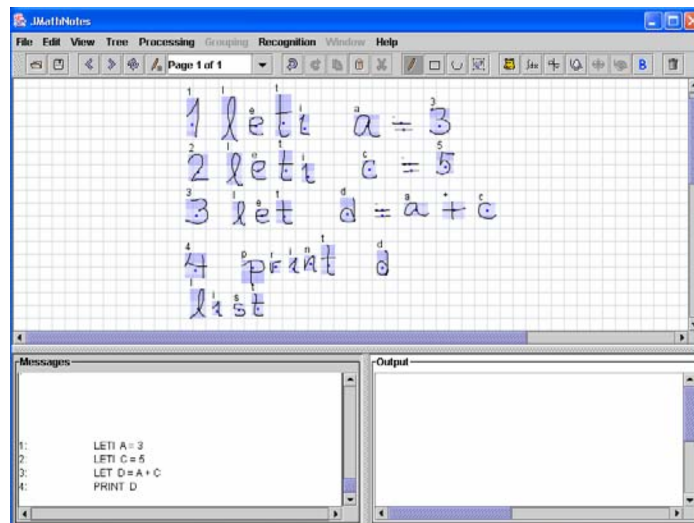

Figure 5.9 Screenshot of JMATH, the editor and training program for formula recognition developed by E. Tapia.

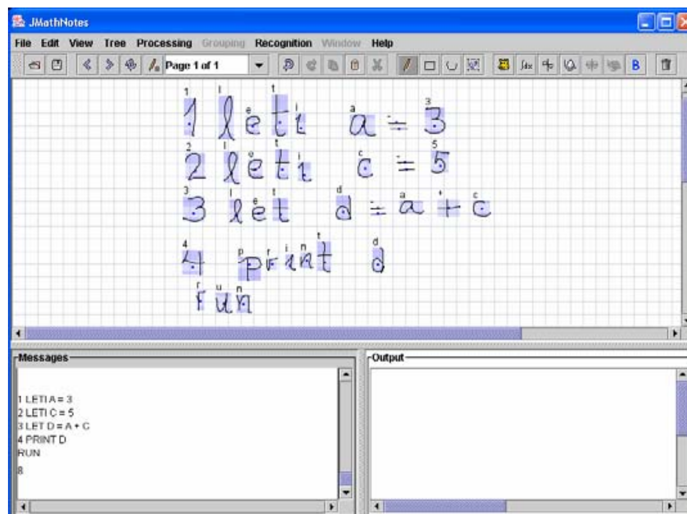

Figure 5.10 Screenshot the Tiniest BASIC program after it has been executed.

Figure 5.10 shows the same program, but now the command "RUN" has been entered. The output of the program is visible in the lower left window, it is the constant 8.

Ideally, many of the options of the JMATH editor will be available in future releases of E-Chalk. Symbols can be erased, for example, by scribbling rapidly on them. Symbols can be moved from their position. If a gap is needed between two lines, the symbols below the first line are selected and moved down. This gesture recognition could allow editing and annotating programs written in more complex programming languages, as explored in the next section.

5.3.4 Animated pseudocode

The significance of the Tiniest BASIC exercise is that it shows what could be done in the future. It would be possible to directly execute handwritten code written in compact languages such as Perl, Lambda Calculus, or Haskell.

Cormen et al. [90] use a very powerful pseudocode to define algorithms. The algorithm's code is compact and easy to read. As we saw in section 4.10, Python is a good match for this kind of pseudocode. It would be possible to write a parser and interpreter for the handwritten version of the pseudocode (extending the Python interpreter). The user would then handwrite a program and the electronic chalkboard could immediately execute it.

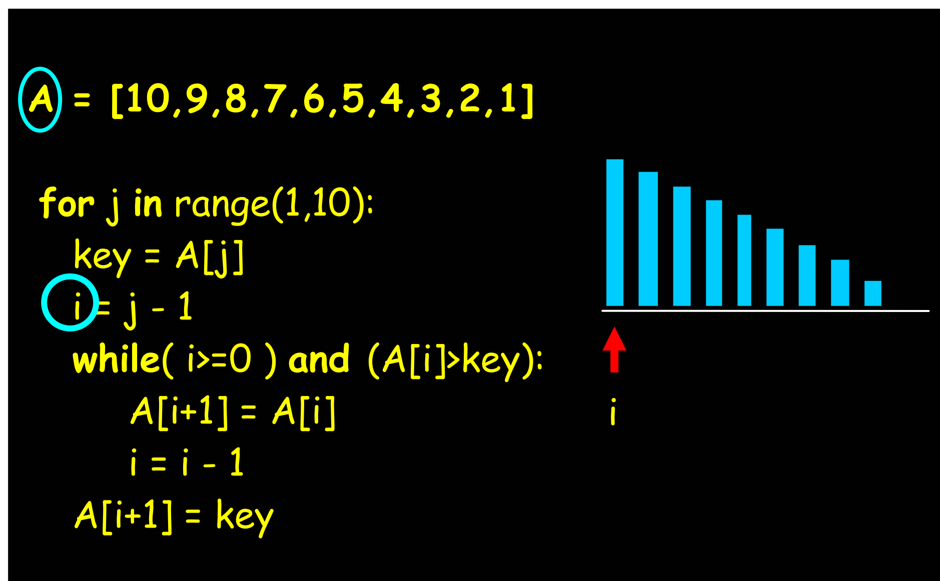

Figure 5.11 A vision of the future. Pseudocode has been handwritten. The user signals to the system that the array A and the index i are to be visualized (by encircling them). The system responds by automatically producing the animation shown to the right.

It would be very interesting to provide automatic animation of pseudocode operations. From Chapter 2, we know that declarative algorithmic animation is difficult and that not many systems have been developed. Nevertheless, it would be possible to annotate handwritten pseudocode in such a way that some portions of it would be animated.

An algorithm handling an array (for example a sorting algorithm) could produce an animation of the array, so that when the code is executed step by step (and this would be shown on the screen) the contents of the array are made visible. The selection of the variables to be visualized could be simply done by underlining or circling their names in the handwritten version of the code. The screen-shot in Figure 5.11 is a rendering of this idea, which falls outside the scope of this thesis, but which is worth pursuing in the future. In the figure, the lecturer has handwritten the Python code for insertion sort, which is applied to an array A. The name of the array and of the variable *i* have been encircled, to tell the system that we want to follow all changes to them. The default view of the array is a bar graph. The computer knows that *i* is an index to the array A and draws it as an arrow. The algorithm is executed and the result is visible to the right. Of course, we assume that the lecturer is collaborating with the system and that she is taking care of carefully handwriting the code. We also assume that the interpreter is able to make some deductions about the kind of variables being used.

5.4 Animated algorithms in E-Chalk with sketched input

The general technique described in the previous section was ported to E-Chalk using the handwriting recognition interface. In E-Chalk, there is a special color which can be set at the beginning of a session. Strokes drawn with this color are processed by the handwriting recognizer engine. The engine receives all the strokes (as sequences of lines), processes them, and gives the result of the recognition to another application (in the case of mathematical formulas, to Mathematica from Wolfram Research). The application processes the input and gives back an ASCII string to E-Chalk or a picture in GIF or JPEG format. This application output (a number, a graph, a drawing, etc.) is pasted to the blackboard.

For our purposes this is not enough. An animation produces multiple frames, which have to be pasted at the same position. If only the history of an algorithm is being drawn, then the handwriting interface is all that is needed, but the output can only be a static image. I talked to the developers of E-Chalk and the handwriting recognition interface was modified to provide an API for third-party applications, such as my animations. The API should allow applications to write directly on the board, giving them control over the rendering of parts of the blackboard. The API for the E-Chalk system was developed so that my animations could be

interfaced to the system. As a workaround while the API was being written, E-Chalk macros were used.

The workaround is the following: stroke input drawn with a selected color is passed to the shape recognition engine. The shape recognition engine groups strokes according to proximity or overlap (some digits, for example, consist of more than one stroke) and recognizes the shape among a library of symbols. This information is passed to my own application, not to Mathematica. My application is an animation engine for E-Chalk, which has loaded an algorithm written in Java. The Java algorithm receives the input from the shape recognition engine, runs the algorithm, and produces a Flashdance animation script. The animation script is processed as a stream by the animation script interpreter, which in turn produces the code for a macro. After entering the input, the user can then go to the macro menu and start the new macro, which will be played as directed by the animated algorithm [Esponda 04b]. The diagram below shows schematically the information flow in the E-Chalk system. All steps are transparent for the user, who only has to call the appropriate macro at the end. Once the E-Chalk API is finished, the macro call will be performed automatically by the system.

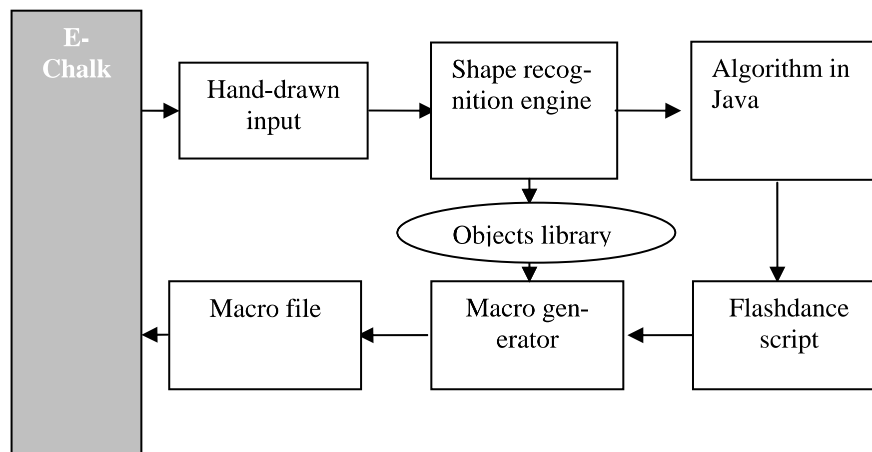

Figure 5.12 Information flow for the animation of sketched input.

The shape recognition engine generates an object library. This is needed, because if we want to reuse the shapes entered by the user, the shapes must be stored by the recognition engine. They are assigned an object number, which can then be used by the animation algorithm generating the script.

5.5 Extracting the input from a macro

As a proof of concept, a simpler experiment was performed before the E-Chalk-API was complete, in order to allow the user to enter sketched input before start-

ing an algorithm. The user writes her input in a macro which produces a file containing the line strokes drawn by the user. A program is used to read this macro, extract the line strokes, and use this line strokes as the objects in the animation library. The coordinates of the line strokes are recomputed so that the lower left point is the origin of coordinates for each object.

The extractor program receives as input the file with the line strokes drawn by the user. A simple heuristic is used to group strokes, computing the spatial and time distance between groups of strokes. The extractor can be used to filter out from the user diagram any kind of shapes. A graph, for example, can be filtered and the nodes can be separated from the edges, based on the shape of the line stroke. Fig. 5.13 shows an example, where the user has entered the alphabet and ten digits. The extractor transforms every shape into a library object, which can then be used again, for example to write a synthesized text in the handwriting of the user. In Figure 5.13 the user input is colored green, the computer output, that is, the words "it works fine", in yellow.

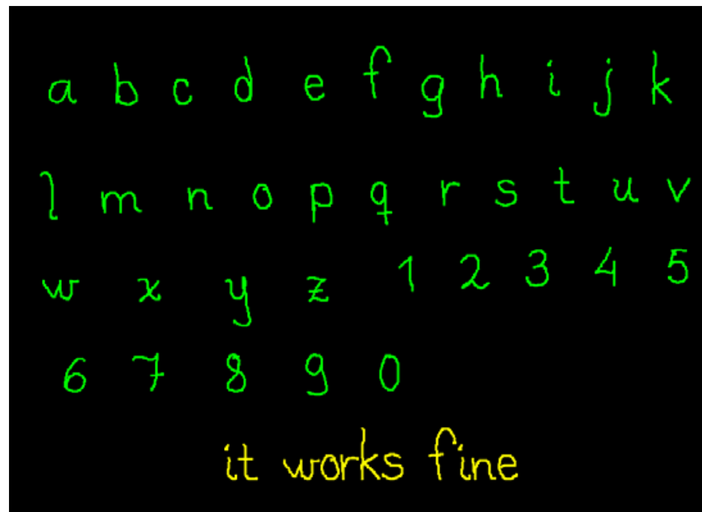

Figure 5.13 Extraction of the individual shapes of the alphabet letters and synthesis of a text by the computer (in yellow).

5.5.1 Bubble sort revisited

After the object library has been defined by the user (by entering her input), the animation algorithm uses this objects in the animation. In the case of the example shown in
Figure 5.14, bubble sort was animated by using ten objects, that is, the digits from zero to nine. The digits were written with a size proportional to their magnitude. The object number corresponds to the digit which has been drawn.

The instrumented algorithm produces an animation script, which after processing yields a macro for E-Chalk. The animation runs on the electronic blackboard.

Figure 5.14 shows the start of the animation and the input written by the user (upper left picture). The upper right side of
Figure 5.14 shows the progression of the algorithm. The pivot for the comparison is painted red; the other number being compared is painted pink. After a number has reached its final position it is painted green, with a thicker line.

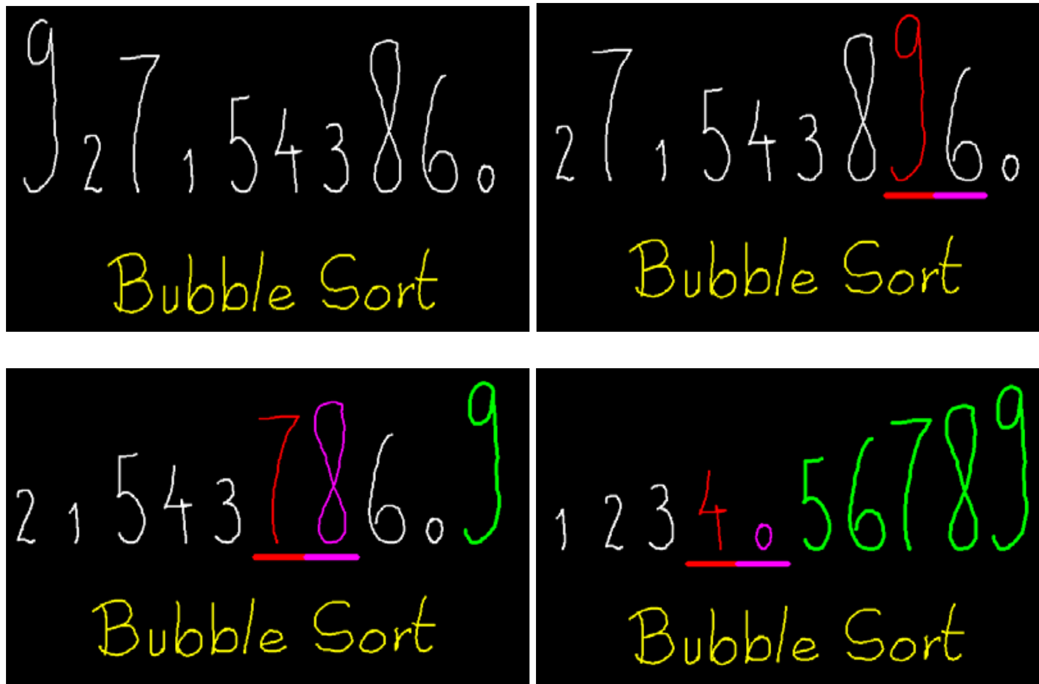

Figure 5.14 The bubble sort algorithm animated with user input

This experiment is also interesting, since multiple visual cues have been used to reinforce the idea of magnitude. The size of the digits is proportional to their value. The value itself is read by the user. Sorted numbers are shown in green, a color which automatically recalls the association of something being right or correct. The action is taking place at the place where the pivot is, which is painted red to signify work. The position of the two numbers being compared is also marked by two horizontal bars which slide across the array.

The animations produced in this manner can be easily collected as macros and can be replayed by an instructor during class.

In the example above, the separation of the strokes was made by considering only connected objects made of one single stroke sweep. This property is easy to test when given the coordinates of the strokes points.

An interesting aspect of this and all other macro animations mentioned above is that the animations are automatically reversible. Once a macro has been played, the user can go backward and again forward, by using the undo and redo buttons in the E-Chalk menu. All board events occurring after the macro starts are stored

in the undo and redo stacks, and the user can navigate at will, back and forth, through the animation frames. E-Chalk offers reversibility at no cost for the algorithm animator.

The difficulty of implementing reversible animations in E-Chalk lies in the fact that all graphic activity is pixel based. When an object is drawn on the screen, in front of another object, it is necessary to store the background so that it can be restored in case of an undo. It is not trivial to design an effective and fast data structure for this task. Fortunately this has been done by L. Knipping and forms part of the standard distribution of E-Chalk.

5.5.2 Prim's Algorithm for minimum spanning trees

The next algorithm is a more spectacular example of the kind of animations which become possible once the user is empowered to enter her input using a pen computer. I wrote a program which implements the popular Prim algorithm for the computation of a minimum spanning tree. A spanning tree is a subset of the graph's edges which touches all nodes of the graph without producing cycles. The minimum spanning tree has the minimum total weight (each edge has an associated weight) of all possible spanning trees. Figure 5.15 shows how the user enters her input: by drawing the nodes and edges of the graph. In this case the extractor program was coded to expect nodes as white elements, edges as green elements. It would have been possible to detect automatically which elements are edges and which circles, but in this case I settled for a simple extractor. The weight of an edge is its total length. The reader can imagine that these are cities connected by roads of different length.

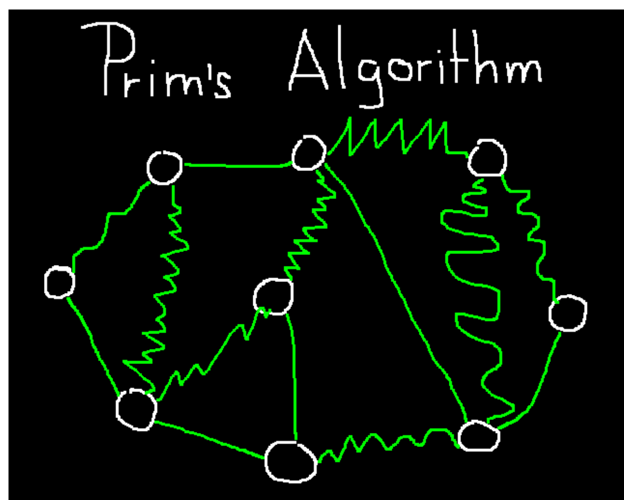

Figure 5.15 The user enters the input. Nodes are painted white, edges are green.

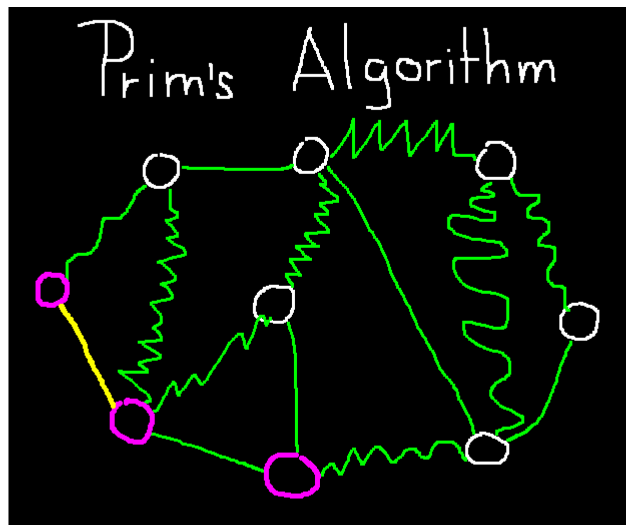

Figure 5.16 An edge has been added to the spanning tree (yellow edge). The next node to be visited is painted pink (lower middle).

Figure 5.16 shows the further progress of the animation. Nodes in pink have been selected and yellow edges already belong to the spanning tree. The next edge selected is the shortest edge touching the pink nodes. Figure 5.17 shows the animation some frames later. The edge to the middle right is being painted yellow after being selected.

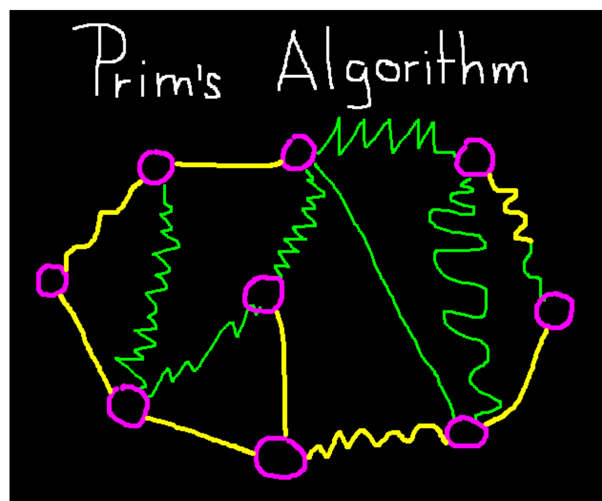

Figure 5.17 The last edge is being painted yellow.

Figure 5.18 shows the end result. All nodes have been visited and the "road map" is the shortest possible tree connecting all cities.

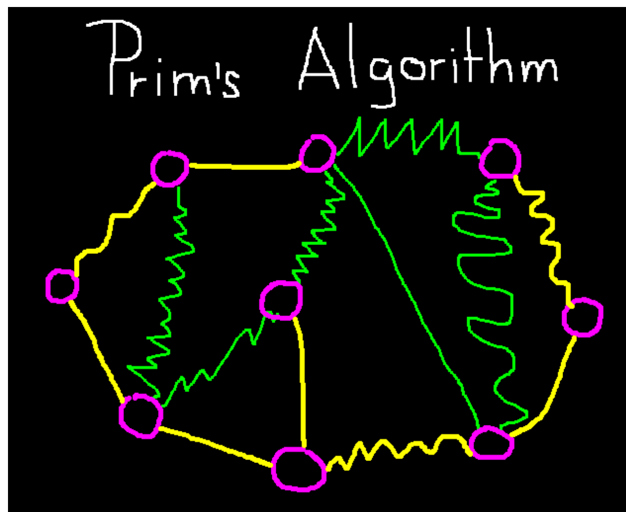

Figure 5.18 Final minimum spanning tree.

The more impressive results from the animation are obtained when the user changes the input and the animation runs automatically.
Figure 5.19 shows four screenshots of the same algorithm running on a new graph sketched by the user.

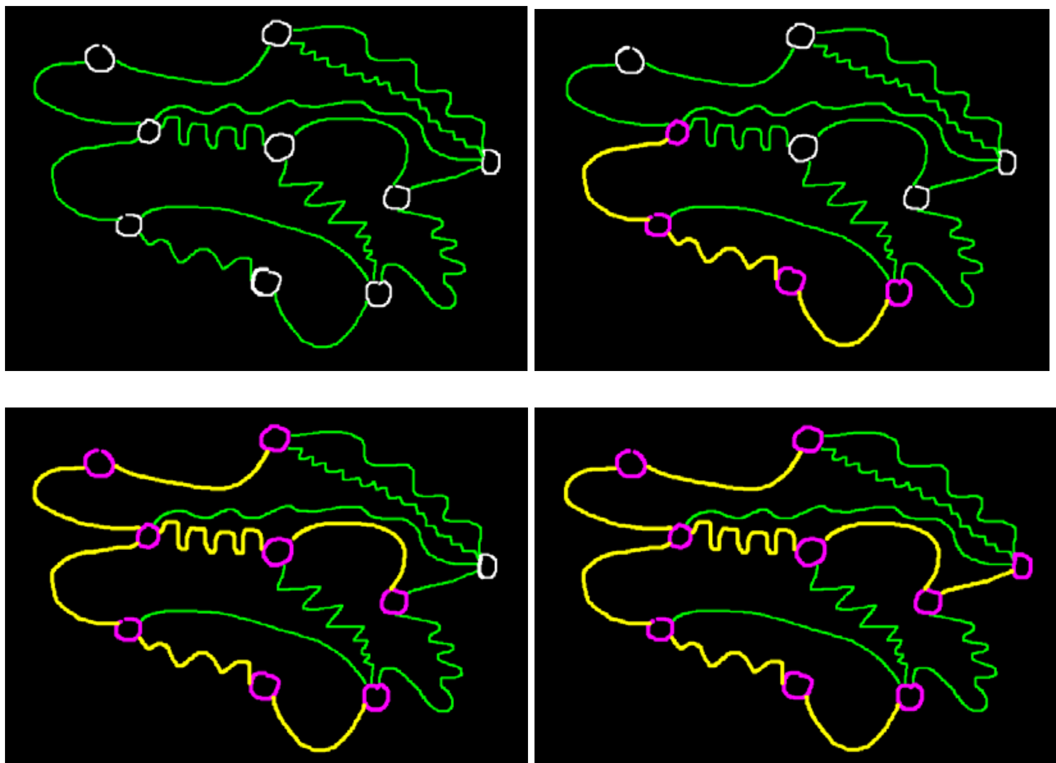

Figure 5.19 Prim's algorithm working on another graph (from upper left to lower right).

In the case of this animation the technique of representing the weight of an edge by its length gives an intuitive feeling for the correctness of the decisions taken at each step. Some algorithmic animation systems represent the weight of an edge sometimes by the width of the edge, but this is the first attempt to represent the weight using a wiggled line. It is also clear why: other algorithmic animation systems render graphs using lines and circles. It is very difficult to draw esthetically appealing graphs using this convention (length of edge equals weight), but not when the user can sketch on a blackboard. Here we find a definitive advantage of the blackboard compared to simple graph drawing programs. The animation produced by the Chalk animator looks almost like an abstract painting.

Once an extractor for graph objects has been written, it can be used to construct the library for many other graph algorithms.

5.6 The E-Chalk Interactive Application Interface

After having shown my macro animations to the E-Chalk developer team, it was decided that computer animation should be directly supported through an Application Interface for E-Chalk. Lars Knipping wrote as part of his forthcoming PhD dissertation a specification for the E-Chalk API that allows developers to seamlessly interface their own applications to an electronic whiteboard.

The menu of E-Chalk was expanded with a new button. When the button is selected, a list of applications appears. The user selects an application, dragging and positioning a frame in the E-Chalk screen. This rectangle is a portion of the electronic chalkboard reserved for the application. All strokes drawn on this frame are passed to the application and the application can draw new strokes inside the rectangle. When the user draws a stroke outside the frame, the application is stopped and does not consume any CPU time. The lecturer can continue with other topics.

In the E-Chalk API, applications “listen” to strokes. Any stroke drawn in the application window is pushed into the application, which provides the method for the push operation. The stroke is passed to the application as a Java E-Chalk object. The E-Chalk methods can then be used to process this stroke in any desired way. An application listening to strokes can recognize the shape of strokes, which trigger different operations and also the creation of new strokes to be drawn on the screen. In the next section, I review the E-Chalk API in more detail.

5.6.1 Description of the E-Chalk API

An animation which can be plugged to E-Chalk is defined as a subclass of the E-Chalk “animation” class. When an animation is selected from the menu and is

started it waits for strokes coming from its own region of the chalkboard. The user animation has to provide two methods: pushStroke and removeLastStroke. When a stroke is drawn in the animation window, E-Chalk calls the pushStroke method for forwarding the stroke to the animation.

The pushStroke method, written by the user, can analyze the passed Java stroke object using the stroke handling methods provided by the API. It is possible, for example, to compute the bounding box of the stroke calling a method, or to obtain the coordinates of the first pixel in the stroke, etc. The API also provides methods to get some important parameters from the screen, such as the size of the animation window and the color of the background. The coordinates of the left upper corner of the region can be read also, an important step before sending any strokes to E-Chalk, which must be encoded using absolute screen coordinates.

The removeLastStroke method is activated by E-Chalk anytime the undo function of E-Chalk is called (pushing on the undo button). E-Chalk deletes the stroke from the screen, but also informs the animation that a stroke has been deleted. This is important in order to manage a consistent view of the screen in E-Chalk and in the user animation.

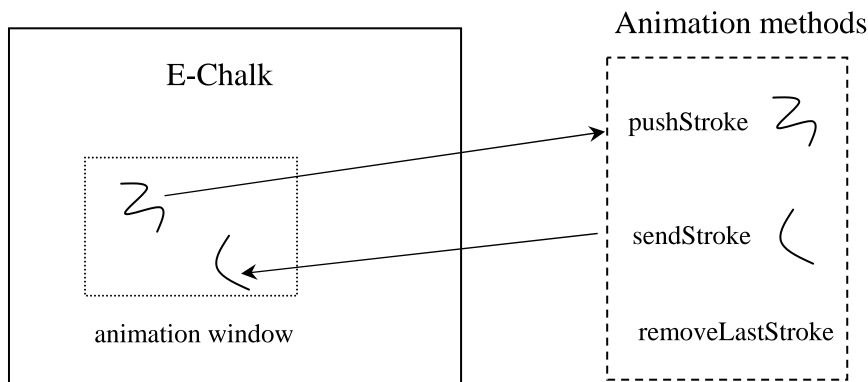

Figure 5.20 : Interaction between E-Chalk and a user animation.

The sendStroke method is provided by the API and allows the animation to generate a stroke by itself and send it to E-Chalk for display on the screen. The API provides stroke constructor methods. With them it is possible to copy a stroke, change its parameters, such as color or width, or just build a stroke point by point. The animation never draws by itself on the screen. All screen updating is made by E-Chalk itself and the only way to alter the screen appearance is by sending strokes to E-Chalk.

Figure 5.20 shows a diagram of the interaction between the E-Chalk API and a user animation. The pushStroke method is called by E-Chalk when a stroke is drawn. The stroke is passed to the animation as an object. The animation can gen-

erate a stroke and send it to E-Chalk using sendStroke. The method removeLastStroke keeps the animation screen state consistent with the visible screen managed by E-Chalk.
Writing an animation for E-Chalk consists essentially in providing the pushStroke method, and in generating strokes using the API stroke constructors. An animation can only be started by calling pushStroke at least once. The method sendStroke can be called repetitively from inside a pushStroke method. When pushStroke finishes, the animation can only be restarted by drawing a new stroke on the animation screen.

5.6.2 Classes and methods of the E-Chalk API

In this section I review the most important methods and classes contained in the E-Chalk API. For more detailed information the reader is referred to the API documentation written by Lars Knipping and included in release 1.11 of E-Chalk.

Animation

An E-Chalk *animation* has a StrokeListener which receives any stroke drawn by the user inside its animation frame. The animation can request strokes to be painted via its AnimationContext. An E-Chalk animation should implement the following three methods from the StrokeListener interface and the Animation interface.

pushStroke This methods is called when a stroke is painted on the animation area. A new stroke is pushed into an input buffer. It can be a stroke repainted with the redo button.

removeLastStroke Remove the last stroke which was pushed into the input buffer (this method is called when the undo button is pressed).

endAnimation Aborts the animation at the request of E-Chalk (when the user draws a stroke outside the animation frame, for example).

AnimationContext

The AnimationContext interface provides a set of methods for obtaining information about the animation frame pasted by the user on the board, and methods to send strokes to be shown in the animation frame. The following methods are defined on this interface.

getAnimationArea Returns the parameters of the animation frame rectangle

getBackgroundColor Returns the color of the background in the board.

sleepUntil This method is used to pause when an animation is running

The more important methods in AnimationContext are those used to paint strokes in the animation frame. These are:

sendStroke Requests E-Chalk to paint a stroke in the animation frame. A stroke is a Java object constructed using the API constructors. It contains information about the points in the stroke, color and width of the lines, and a time stamp for drawing segments between points.

sendStrokes Similar to sendStrokes, but an array of strokes is passed to E-Chalk

getPendingStrokesCount With this method the animation can ask if all strokes have been painted.

animationEnded This notifies E-Chalk that the animation has ended. E-Chalk stops sending strokes to the animation.

BoardStroke

The class BoardStroke is used to construct strokes for the board and set their parameters. The *BoardStroke* constructor allows the animator to build strokes from arrays of coordinates, including timestamps, and specifying parameters such as line width, color, and drawing duration. There are 21 BoardStroke get methods, which can be used to extract parameters from a stroke. The method getColor, for example, allows the user to retrieve the color of a stroke passed by E-Chalk. There are methods to extract the array of point coordinates from a stroke, the bounding box, the time stamps, etc.

AnimationKit

The animation kit class is the factory kit for producing animations for E-Chalk. The class contains important methods such as:

getAnimation Provides a link to the user animation
getAnimationName Returns a localized name for the animation

getMinimumAnimationSize Returns the minimum size of the frame that can be used by this animation.

In the following sections, I show some examples of the animations produced using the E-Chalk API. The animations are fully interactive and can operate on sketched input entered by the user.

5.6.3 Example: bubble sort from sketched input

Our first example is just a reimplementation of the bubble sort animation discussed at the beginning of this chapter using the my-E-Chalk clone of E-Chalk programmed in Flash. My-E-Chalk was important for this thesis, historically, because only after the E-Chalk team saw my animations in this clone of E-Chalk, they decided to produce the E-Chalk API for interfacing animations to the system.

The two screenshots in Figure 5.21 show the start of the animation. The instructor is explaining bubble sort and has written the name of the algorithm on the board. Then the animation frame is pasted to the board. Several strokes are drawn inside this frame, one after the other. When enough strokes have been drawn, the user draws a small angled segment on the right upper corner. This is the signal for the animation to start sorting the strokes, according to their length. The angled stroke is a gesture which is analyzed and recognized by the pushStroke method, which then starts the animation. The animation itself consists of many calls to the sendStroke method, which erase and repaint strokes at their appropriate positions.

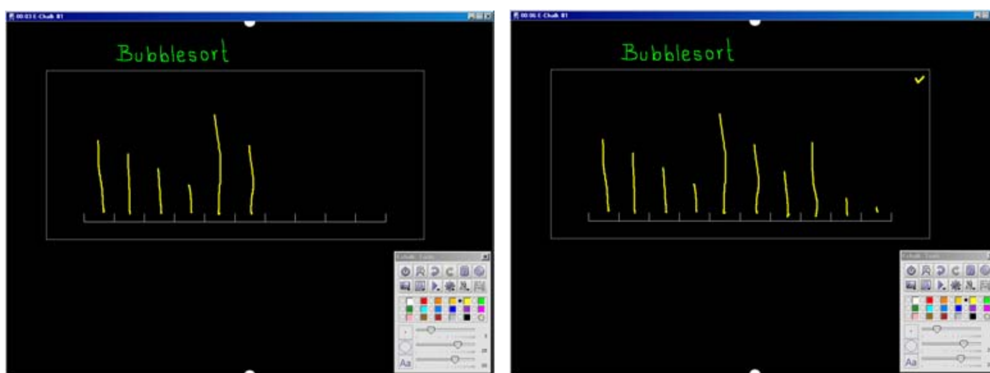

Figure 5.21 : The user enters strokes to be sorted according to their length (right screenshot). An angled stroke in the upper right corner starts the animation (left screenshot).

Figure 5.22 shows the progress of the animation: several strokes have been sorted. Two of them are moving to their new positions.

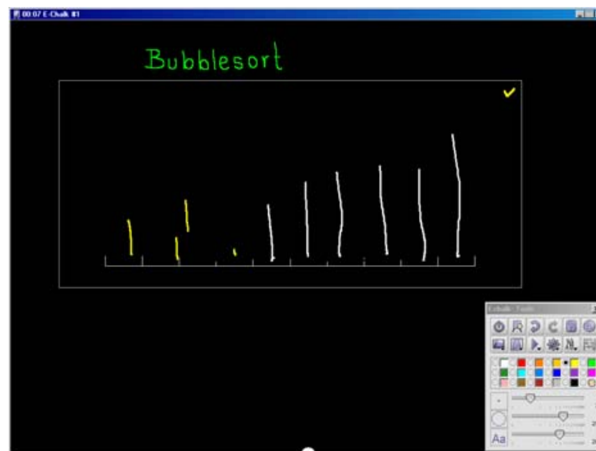

Figure 5.22 Strokes being sorted in the animation window.

Figure 5.23 shows that many kinds of strokes can be drawn, wiggled or not wiggled. The bounding box is the parameter used for the sorting. My experience in the classroom is that students like to experiment with different classes of input and are attracted to the system due to this flexibility.

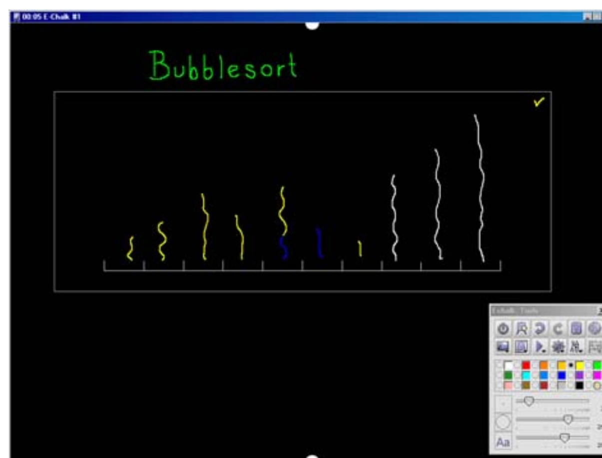

Figure 5.23 Other kind of input for the bubble sort algorithm.

5.6.4 Dijkstra's algorithm

A more involved example of the kind of animations possible with the E-Chalk API is shown in the next figures.

Dijkstra's algorithm is a well-known method for computing the shortest path between a source node and all other nodes in a graph. In Fig. 5.24 we see how a graph is entered for processing. In this particular example, nodes are drawn using green strokes, edges using yellow strokes. The length of an edge is the number of pixels the stroke contains. Therefore, wiggled edges are longer than straight edges. The algorithm starts by marking all edges as "unused" (using white) and all

nodes as having an infinite distance to the source node. The source node itself is marked with a zero (it has zero distance to itself).

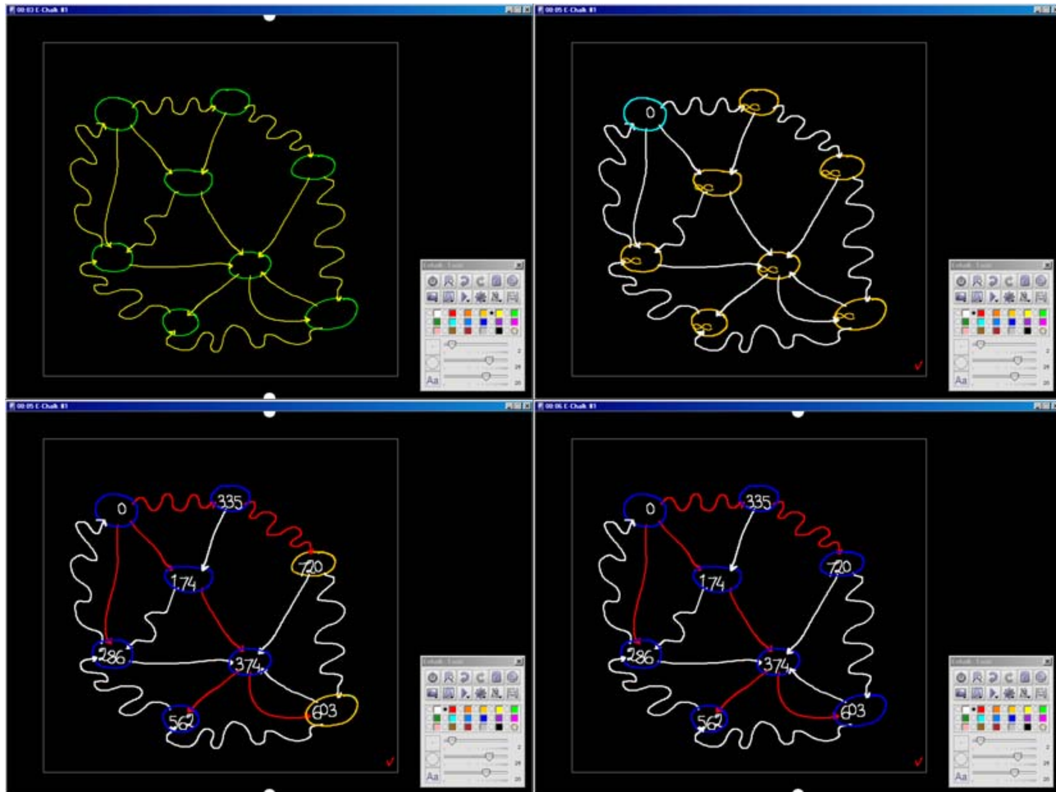

Figure 5.24: Dijkstra's algorithm running on a hand-drawn graph. The algorithm starts with the input shown on the upper left. All nodes are marked with an "infinite" distance (upper right). The algorithm gradually relaxes the distances. Finally, the markings on the nodes show the shortest distance to the start node and the edges selected (colored red, lower right).

The two lower screenshots in Fig. 5.24 show the further progress of the algorithm. Starting from the source node, all nodes connected to it are relaxed. This means that their distance to the source is updated by selecting the minimum of the current mark on the node, and zero plus the edge length. Note that the edges are directed, they point from one node to the other. In the next step, the node with the smallest mark (smallest distance to the source) is expanded by looking at the neighboring nodes. Expanded nodes are colored blue. The marks in the adjacents nodes are updated: if the mark on the source node plus the length of the edge to the neighbor is smaller than the mark on the neighbor node, the mark in the neighboring node is updated to the new value. The node is marked as non-expanded. The process is continued by always expanding the node with the smallest mark, until all nodes have been expanded. This guarantees that all possible paths through the graph have been considered.

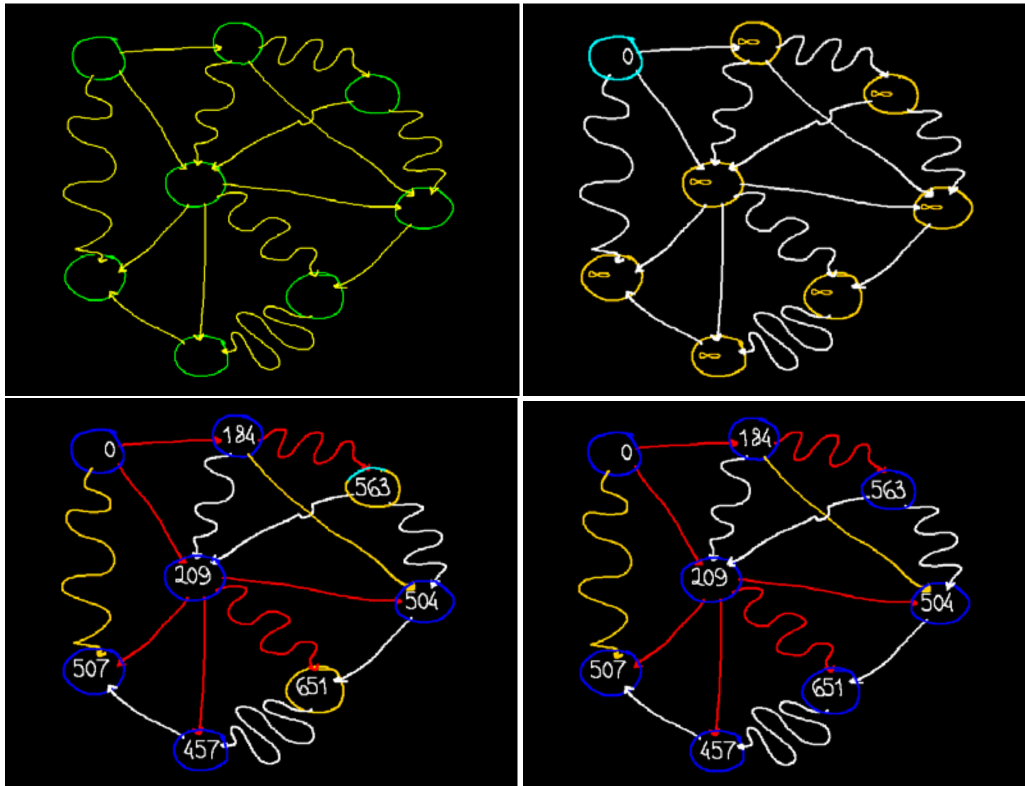

Figure 5.25: A second example of Dijkstra's algorithm running in E-Chalk.

In this particular example, the recognition of nodes and edges was simplified by using specific colors for each. This can be configured by the programmer easily. Also, the marks were generated using stored letters and numbers in the user's handwriting. This simple handwriting synthesis is easy to implement and avoids spoiling the "look and feel" of the sketched input with incongruous characters.

This example concludes our presentation of some classical algorithms programmed using the E-Chalk Animator. The algorithms, once programmed, are easy to run and easy to explain in class. The lecturer can run examples with different kinds of graphs and the input time is reduced to a minimum. The students are also encouraged to experiment with input of their own. E-Chalk Animator is an animation tool that fosters critical thinking and makes experimentation as natural as drawing a sketch.

5.7 Summary and Discussion

I can best summarize this chapter by saying that we have boldly gone where computer animators have never gone before: we have considered the interaction of the user with algorithms and animations directly, using the possibilities of pen computing in the classroom.

This chapter started by stating that sketches are esthetically appealing and can attract the interest of students. Animations are for learning. An instructor in a classroom needs a more natural interface with the teaching instrument, i.e. the blackboard, and this is what we have investigated in this chapter. Although many sketches are used to illustrate algorithms or books, few experiments have been conducted towards coupling sketches and algorithmic animations, the exception being the "low fidelity" algorithms of [Hundhausen 01] and the non-photorealistic animations described by Strothotte [98].

For my research I first developed a clone of the E-Chalk board using Macromedia-Flash. It was an exercise in fast prototyping, but the graphical result is excellent. The strokes drawn on the electronic board are handled as vector graphics by Flash and the animation engine helps to produce smooth and very appealing animations. The same algorithm can be run with different inputs from the user, as shown in Section 5.2. The user interface is very natural: editor and animation medium are not separated – the blackboard provides a unifying metaphor. The animations attracted widespread attention when first shown at the FU Berlin.

Given an electronic blackboard, it is natural to ask if the algorithms could be directly written on the blackboard and could be interpreted by the computer. I have called this "handwriting programming". My experiment with the Tiniest BASIC interpreter shows that it is indeed possible to couple a handwriting recognizer with an interpreter, in order to provide this functionality. I wrote the Tiniest BASIC interpreter in a few hours. Future work will be oriented towards recognizing pseudocode languages, such as Python, which offer a very powerful medium for handwriting algorithms. It is my belief that the experiments recorded here are among the first conducted towards developing a handwriting coding and animation system.

The second half of the chapter deals with E-Chalk and how to implement a two way communication channel between the user and the algorithm to be animated. The E-Chalk API was created with this objective, but similar functionality can be obtained through the use of macros. I showed that a user can enter his or her input as a macro, which is then analyzed by a symbol recognizer. The symbol recognizer builds a library of symbols. An animation written in my scripting language can then access these objects and use them during the animation. I showed that this method can be extended even to the labels of the animation, which can be produced by text synthesis, that is, by reproducing the handwriting from the algorithm animator. This preserves the sketch form of the diagrams, even when they contain text.

The release of the E-Chalk API is significant, because it allows developers to interface arbitrary programs to E-Chalk, which can accept sketch or handwritten input and operate with it. There is no "Medienbruch" and the user interface remains homogeneous. E-Chalk Animator was one of the first programs to take ad-

vantage of the new E-Chalk API and provided some of the pressure from the user community towards its definition by the E-Chalk programmer team. In the last sections of this chapter I illustrated the use of the E-Chalk API by animating bubble sort and Dijkstra's algorithm. Now that the E-Chalk API has been released, some of my animations are being distributed as an extension of E-Chalk in version 1.1. The results described in this chapter have been presented at international conferences [Esponda 04b, 04c].

In the next chapter we will close our investigation by going to the opposite extreme: not sketches but the fine graphical rendering provided by the Flash animation engine will be the delivery medium for my algorithmic animations.