Chapter 4

Board Server

4.1 Painting on the Board

The code for displaying content on the board is shared between the client and the server-side board for maintenance reasons. With the client Applet required to run in a browser supporting only Java 1.1, this shared code cannot use more recent Java library additions.

The elements that are painted are represented as instances of the abstract class `echalk.shared.form.Form`: the subclass `StrokeForm` is used to represent a stroke composed of a sequence of connected line segments, a `TextForm` a typed text, an `ImageForm` an image on the board, and an `AppletForm` an Applet. A `Form` object stores its position on the board, and provides a paint method to draw itself and a method to test for an intersection with a given rectangle, which is used to determine the visibility of `Form` objects on the board for the current scroll offset. The `Forms` are stored sequentially in a dynamically growing array in order of creation.

The board uses double buffering to avoid having to call the paint methods of each visible `Form` each time a repaint request is issued by the window system, for example when a portion of the board becomes visible again after being obscured by another window. Whenever a new `Form` is added to the board content, it is painted both to the screen and to the board's offscreen buffer. On a repaint, the offscreen buffer is simply copied to the screen, resulting in a fast and flicker-free repaint.

A complete redraw of the offscreen buffer is only needed when scroll events happen or when a clear-all command or an undo is issued, see Section 4.5. In this case, all `Forms` have to be checked for their visibility given the current scroll offset, and the visible ones have to be painted in the order they were added to the board contents. The original brute-force approach ran sequentially through all `Form` elements, roughly about 2.000 objects per hour of recording. On the server-side board, the resulting redrawing behavior was still perceived as fast enough in real lectures.[1] However, rewinding and fast-forwarding the client

[1] This was even true for extensive tests running on a 500 MHz, 256 MB Pentium III Windows laptop, when the original data representation resulted in roughly 25,000-50,000 `Form` objects per hour of lecture. The original data representation created a `Form` for each line segment drawn. Now the board merges all line segments belonging to the same user stroke into a single `Form`, which creates less than a tenth of the original number of `Forms`, speeding up

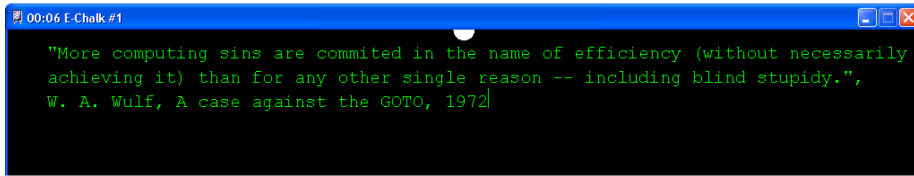

Figure 4.1: Typed text with the first line break created automatically and the second defined explicitly by the user with `[Shift]-[Enter]`.

board felt too slow. For fast redrawing, a data structure that allows fast access to all visible `Form` objects in the order they are to be drawn is now used. The virtual board space is separated into pages of the visible board area size. For each page, there is a (dynamically growing) array of references to all `Form` elements that intersect the given page, stored in their order of creation. For each given scroll offset, the visible board area is then contained in up to two pages and only the `Forms` referenced in the pages' arrays have to be checked for visibility. With the `Forms` of a single page being already ordered, all the relevant `Forms` can be retrieved in the correct order from the page arrays in linear time.

Elements which are shown only temporarily on the board, like image-placing frames and text-input cursors, are not painted to the offscreen buffer. Instead, they are painted directly to the screen using XOR paint and removed by repainting with another XOR paint.

For the board client, an instance of the `echalk.shared.DrawPanel` class is the component handling the actual display of the board content, implementing double buffering and processing board events into new `Forms` or changes to existing ones. The subclass `echalk.board.gui.ActiveDrawPanel` is used for the server-side board. The `ActiveDrawPanel` is capable of processing user inputs to create board events. To allow adding Applets to the `DrawPanel` (see Section 4.10), the `DrawPanel` class inherits from the container[2] `java.awt.Panel`.

4.2 Drawing Lines

The board feature used most is the plain drawing feature. A drawing action is achieved by a mouse-drag-like motion with a pointing device. The board receives the points of the drag movement and connects these points to a stroke in the current drawing color and drawing width. Eraser actions are handled like drawings in the background color.

A stroke to be painted is stored in a `echalk.util.form.StrokeForm` object. A `StrokeForm` stores the list of its vertex points, stroke width, and a color. The line segments forming the stroke are drawn as if the pen were a circle with the given width, i. e. round line joints and caps are used. A convenient way to draw these lines would be to use an instance of the class `java.awt.Graphics2D` for painting and set its drawing shape to circle with the `setStroke` method. Unfortunately, `Graphics2D` is not available in Java 1.1. Because the code for drawing

redraws considerably.

[2]In Java, GUI components that permit embedding of other components are subclasses of `java.awt.Container`.

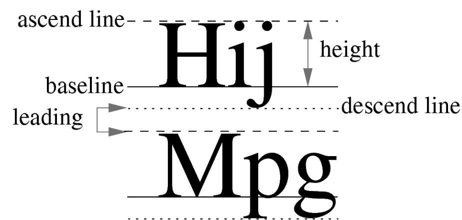

Figure 4.2: Vertical font metrics to consider for laying out text. Similarly, horizontal advance is composed into left side bearing, glyph width, and right side bearing.

lines should be the same on the server and client sides and the because the client side cannot be guaranteed to have the `Graphics2D` available, the drawing must rely on the more basic drawing methods provided by `java.awt.Graphics`. The board draws a line segment as a rectangle with circles at its ends.[3]

4.3 Typing Text

An `echalk.shared.form.TextForm` represents typed text by text color, font size, and position on the board. If it is still active, i. e. is still being changed by user key typing, it also displays a cursor in the text. The user can add all regular characters to the text as well as use delete, backspace, left and right text arrows to move the cursor in the text, and the home and end keys to jump to the start or the end of the text. Also, paste from clipboard with `Ctrl-V` and emacs style cut to clipboard (*kill* from cursor to end of line with `Ctrl-K`) are supported, as well as a text history accessible with the up and down keys. The history is kept between session. The number of entries in the history is determined by a property in the `echalk.conf` file. Key inputs that are duplicates to their predecessor are omitted from the history.

A `TextForm` splits the typed text in rows with dynamically created line breaks to avoid the text running out of the board area, see Figure 4.1. The vertical metrics of a font needed for the layout are illustrated in Figure 4.2. When painting, a call to the `java.awt.Graphics` method `drawString` is done for each row, because `drawString` does not handle newlines itself.

4.4 Images and Applets

An `echalk.shared.form.ImageForm` contains the board position of the represented image and the image data as `java.awt.Image`. Is paints itself by using the `java.awt.Graphics` method `drawImage`.

An `echalk.shared.form.AppletForm` instance encapsulate an Applet as an `java.applet.Applet` and its position on the board as rectangle. The painting

[3]Because the basic circle drawing with `Graphics` results in drawings with displeasing asymmetries for even circle diameters due to a Java bug (see IDs 4080106, 4151279, 4151636 and related in the Java bug database [43]), E-Chalk relies on its own implementation of the Bresenham circle drawing algorithm [Bre77].

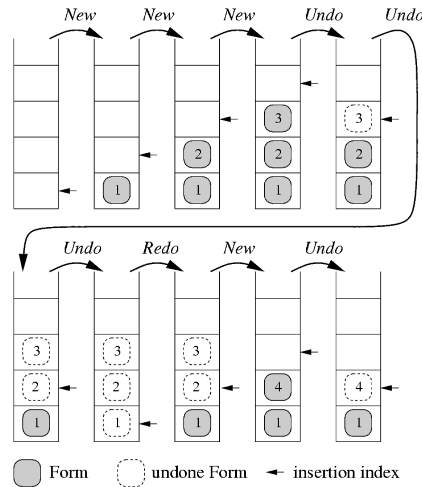

Figure 4.3: An example of handling a sequence undos, redos, and `Form` object creations. The example sequence is taken from the `UndoManager` example in [LEW+02].

is delegated to the `Applet`'s `paint` method. When the `Applet` is added to the board, it is registered as a `java.awt.Component` to the `DrawPanel`. For this reasons, the `DrawPanel` inherits from `java.awt.Container`, see Section 4.1. Whenever the `AppletForm`'s absolute position changes due to a scroll event, the `Applet`'s `setBounds(java.awt.Rectangle)` method is used to redefine its area. For description of logging AWT events of Applets for replay purposes, see Section 4.10.1.

4.5 Undo, Redo, and Clear All

To realize *Undo* and *Redo* actions, the board stores all `Form` events sequentially in an array and keeps an index to the next point of insertion into the array, like a stack.[4] An undo command decreases the index by one, a redo reinserts an undone element by increasing the index. When a new `Form` is added, redo of previously undone elements becomes unavailable. See Figure 4.3 for illustration.

A *Clear All* command is rarely used. It simply removes all elements from the `Form` array.

4.6 Custom GUI elements

The board features a few GUI elements specifically customized for the E-Chalk application. For choosing a drawing color, a subclass of `javax.swing.Checkbox`

[4]The Java package `java.swing.undo` provides an extensive collection of interfaces and classes for undo and redo features. Unfortunately, the undo/redo managing mechanism is not available with Java 1.1 and can therefore not be used in the code shared between the board server and client. Another shortcoming is that the implementation does not support infinite undo depth. For these reasons, the Java undo support is not used by the E-Chalk board.

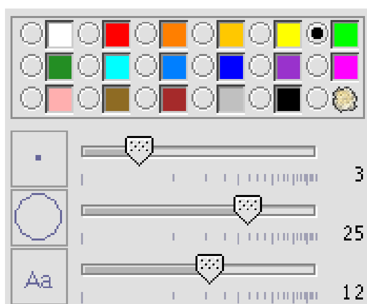

Figure 4.4: Custom color and size chooser elements from the board's tool dialog.

was implemented to allow graphical color labels (or an image of an eraser as a label for the eraser tool) instead of textual labels. Instead of a regular `javax.swing.Slider` an equivalent component with a logarithmic scale was used for selecting the size of the drawing tool and the font size for typed text, because the exactness of drawing sizes is the more important the smaller the drawing size is. See Figure 4.4 for an illustration of the chooser components. The implementation of the logarithmic slider relies on Java Swing `Components` delegating their painting and controls to a `ComponentUI` class, using the *model-delegate* variant of the *model-view-controller* (MVC) architecture. The board instantiates a standard Java `Slider` component and sets its `ComponentUI` to an `echalk.util.LogarithmicSliderUI`, which provides the logarithmic style in painting and control.

The `echalk.board.gui.URLDialog` allows the user to choose a bookmark. To make the Java Swing list component (an instance of the `javax.swing.JList` class) display the thumbnail icons saved with the bookmarks[5], an instance of a custom list cell-renderer class is set to the list.[6] The dialog also shows a summary of the bookmark properties in the tooltips of a bookmark, which can be especially helpful for bookmarks with parameters, namely for CGIs, chalklets, and macros. See Figure 4.5 for an example.

The bookmark lists for images and Applets are just named links to files without any extra parameter data needed to load the resource. For these bookmark lists, the displaying `URLDialog` also features an input line for the user to type in a known URL address. For convenient selection of local files, a file chooser can also be launched. Its file filters are preset to show only relevant files.

4.7 Board Resource Loading

The class `echalk.board.rc.RcLoader` handles access to external resource data, i. e. Applets, images, CGI calls, requests to computer algebra systems, as well as loading animations/chalklets. After loading, it passes the data on to resource-

[5]See Section 3.4.

[6]The class `echalk.util.CellRenderer` implements the `java.swing.ListCellRenderer` interface and is used here. It also implements the `java.swing.table.TableCellRenderer` interface to be used for rendering the thumbnails in the bookmark editors, where the bookmark entries are shown in `java.swing.JTable` objects.

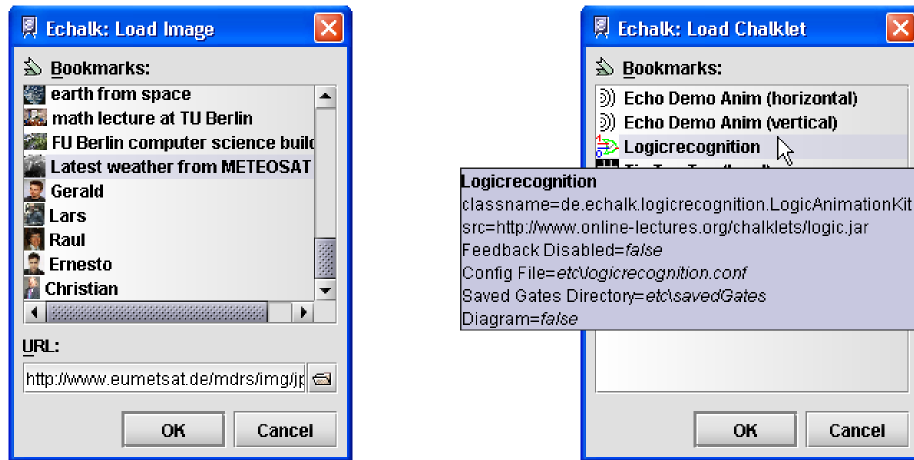

Figure 4.5: URL bookmark dialogs of the board, left with textual input line and button to launch a file chooser, right with tooltip for a chalklet bookmark.

type-specific *handler* objects which manage how they are integrated into the board content. Macros are not loaded via the `RcLoader`.

The resource loading *tasks* for the resource loader are triggered in the GUI's event handler thread, and because (potentially) longer actions should never be handled directly in the event handler thread[7], the `RcLoader` uses a separate thread to handle these tasks. As described in Section 2.4.2, a feedback dialog is shown when a load task needs more than half a second, and the dialog also provides a cancel action for the task.

Canceling a load task is not easily done: it is not possible, for example, to abort a read from a URL immediately. As long as the thread is IO bound, it cannot process the message to stop. As a workaround, the thread is not only sent a signal to die, but it is also marked as "invalid", so that any resulting data are discarded.

For efficiency reasons, not every task creates a new thread. Instead the loading thread is reused[8], except when the user selects to cancel the loading task. In the latter case a new thread is created to ensure that the user can immediately start a new resource loading task. This means the `RcLoader` is implemented according to the thread pool pattern [GHJV95], but with only a single worker thread being in the pool.

Resource requests are composed of two subtasks, represented in an abstract class named `echalk.board.rc.Task`. First, the raw resource data are loaded, like for example getting data of an image from its URL. In the second step, the loaded data is decoded to an object that can be handled by the board, i. e. the image byte data has to be converted to a Java `Image` object. After that, the image is passed on to a handler object which knows how to add the resource to the board. For example, the `ImageHandler` lets the user select the position of

[7]Responding to user input is done in the event handler thread. Having a longer action being handled in the event handler effectively renders the program non-responsive. It would not even handle cancel actions from the user until the pending action is finished.

[8]This is an implementation of the worker thread pattern, see [GHJV95].

an image on the board.

The `LoadTask` objects contain the resource loading and decoding implementation as well as the methods for accessing information about the current progress to display in the progress/cancel dialog: a textual message with a localized description of the step the loading process is currently performing and a numerical progress, if it is available.[9]

The loading subtask also tries to determine the MIME-type of the resource loaded. For most requests, like using image bookmarks, it is known in advance what kind of data will be returned, but for CGI requests it cannot be decided a priori which content decoding and handling is needed. The system then uses the MIME-type information to decide which decoder to use.

See Figure 4.6 for an overview of the classes involved in board resource loading.

4.7.1 URL Loading

Many of the board resources are loaded from URLs. In the case of CGI scripts, there can be also POST arguments given for the URL request.[10]

For a URL loading task, a URL connection is opened, any POST arguments are sent sent over, and the returned data is read from the connection. If the URL is a file URL, the `URLLoad` object does not use the Java `URLConnection` mechanism. Instead, the more efficient standard file read is used. An early implementation, which relied on the standard URL mechanism, suffered from a noticeable slowdown due to overhead introduced by the Java `URLConnection` mechanism.[11]

Java also allows access to URLs which need authentication. The programmer has to set a `java.net.Authentificator` to use the mechanism. The E-Chalk board defines the `echalk.board.util.LoginAuthentificator` as such an authenticator. When a user tries to access a protected URL resource, an authentication dialog is displayed. The dialog shows all the information available for the request, the authentication message from the remote side, and the address of the protected resource, and allows it to input the user's password without the password to be shown, see Figure 4.7. On a failed authentication, the dialog reopens until the user successfully authenticates or chooses to cancel. An authentication for a resource is only needed once per session since authentication data are cached by E-Chalk for the next request.

The standard URL handling of Java supports authentication only for HTTP requests. However, Java allows to replace the handling mechanism of URLs on a protocol basis (and to define handlers for new protocols) by a call to the `java.net.URL` method `setURLStreamHandlerFactory`. In the E-Chalk system, the FTP protocol handling was extended to support authentication.[12]

The HTTP protocol handling is also replaced in E-Chalk if it runs with a Java version lower than 1.4 to improve user messages in case of a resource access

[9] The progress cannot always be determined numerically. For example, if a task for loading a file from a URL is still waiting for the connection to the remote server, there is no way to tell how long it has to wait. In these cases, the dialog displays an indeterminate progress bar.

[10] GET arguments cannot only be specified for the CGI requests, but also for any other URL loading request, as they are encoded within the URL, see [FGM+97, FGM+99].

[11] So far, this speed was only tested in JRE 1.2.

[12] For Java 1.3 it also fixes a bug that prevents the specification of an alternative port in FTP URLs, see Java bug 4320992 [43]. In Java 1.4, this problem no longer occurs.

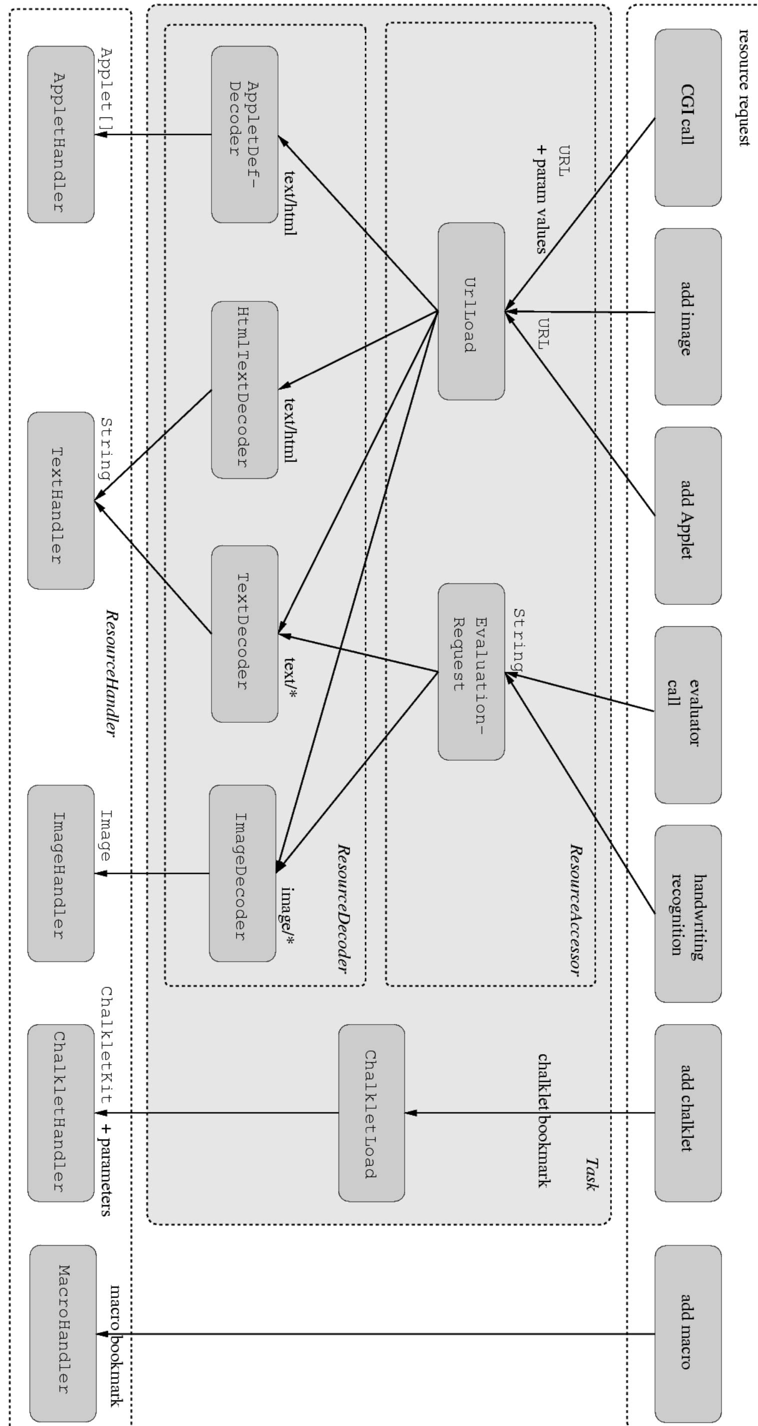

Figure 4.6: Overview of resource loading classes.

Figure 4.7: Authentication dialog shown for loading password-protected resources.

error. The `java.net.HttpURLConnection` has a method `getResponseCode()`, which would allow the `URLLoad` to return the HTTP request's response code to examine failed loadings in more detail, for example to distinguish between a document not found, a forbidden access, a timeout, or a user-canceled authorization. Unfortunately, this method does not always return the response code when the document was not successfully loaded. It often throws an exception due to a Java bug.[13] The HTTP handling mechanism used in E-Chalk fixes this problem.

For HTTP requests, the MIME-type is accessible via the `URLConnection`, as HTTP-Server sent this information with their reply (for HTTP versions since 1.0 [BLFN96]). For other protocols, this information is guessed from the extension of the URL's file-part, or, if an extension is not given, from the data content, using the *magic number* entry for file types.

4.7.2 Requests to Computer Algebra Systems

Another type of resource access are requests to computer algebra systems. E-Chalk's API provides interfaces to develop connectors for arbitrary system. The connectors for Mathematica and Maple are already provided as well as the connector to a simple built-in calculator.

Note that requests to computer algebra systems must be cancelable, as it is possible to send an infinite recursion or computing requests that are beyond the power of the computer (e. g. 9^{99}).

A connector is implemented by inheriting from the abstract class `de.echalk.util.EvaluatorKit`. It provides the localized setup information for the E-Chalk setup dialog described in Section 3.3 and a static factory method to build an object implementing the `de.echalk.util.Evaluator` interface. The `EvaluatorKit` also defines the parameter list for the factory method and how the parameters are set by user in the above-mentioned setup dialog. If an `EvaluatorKit` is activated, a connection to the computer algebra system is made by building an `Evaluator` at startup.

---

[13] To be more precise, in Java 1.3 URLs for which the server's response code was over 400 and where the URL file path does do not end in `/`, `.htm`, `.html`, or `.txt` trigger a `java.io.FileNotFoundException` whenever calling any of the methods `getResponseCode()`, `getResponseMessage()`, `getErrorStream()`, `getHeaderField(String name, long def)`, or `getInputStream()`. See Java bug 4160499 and the related 4150792, 4191207, 4222009, 4300174, 4314717, 4406592, 4492994, 4655826 [43].

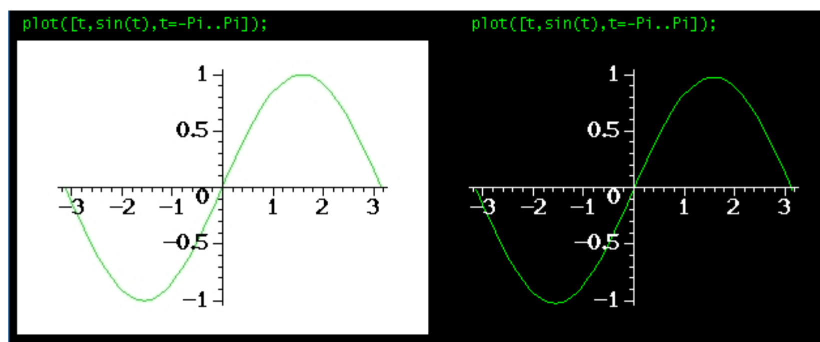

Figure 4.8: Maple plot without transparency (left) and modified for transparent background (right).

The `Evaluator` is notified of the board's background and is registered as a listener to the board's current foreground color. This allows for adjustments of returned images for the board's colors.

The `Evaluator` for Wolfram Mathematica uses a Java connection package provided by Wolfram, called JLink [46]. E-Chalk's implementation allows to connect to a locally-installed Mathematica Kernel and can handle both textual answers and graphical plots. As default plotting options, the plot color is set to the board's current foreground color and the plots background color is set to transparent. When Mathematica is connected, it is also used to evaluate the requests from E-Chalk's mathematical handwriting recognition, see Section 4.9.

The `EvaluatorKit` for Maple was integrated due to user requests. Like the Mathematica connection, the Maple integration can handle both text output and graphical plots. It directly starts Maple as a process and realizes the data exchange as reading and writing text data via the process's standard IO streams and plots data via temporary files. Maple does not support transparent background in plots, but E-Chalk's `MapleKit` optionally allows transparency. This is realized by parsing the plot's GIF image data and setting its background color to transparent.[14] Also, any color that is equal to or very near to the board's background color is set to black (on light backgrounds) or white (on dark backgrounds), so that they remain visible. See Figure 4.8 for an example. This color modification is not possible for some older Maple versions, as they only allow to plot in JPEG format, where transparency is unavailable.

To evaluate the mathematical terms identified by the handwriting recognition[15] without Mathematica being available, a simple calculator program is part of E-Chalk as an `Evaluator`. It can handle terms with integers and fraction numbers as operands, brackets, and the operators $+$, $-$, $*$, $/$, and exponentiation. A student has extended this calculator to handle function calls, definitions of function, constants, and basic function plots for two and three dimensions, see Figure 4.9 for example output.

With the E-Chalk API it is easy to integrate other computer algebra systems

[14] GIF images have a color table of up to 256 colors. One of the colors can be marked to be fully transparent.

[15] The mathematical handwriting recognition is described in Section 4.9.

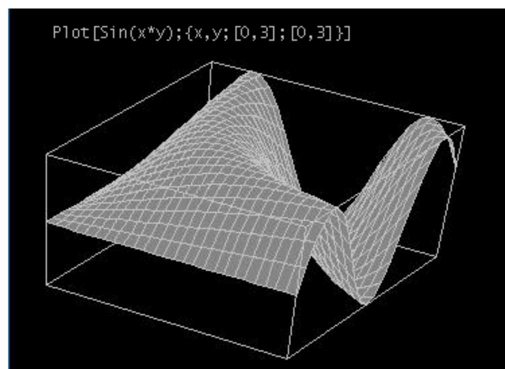

Figure 4.9: Example plot from E-Chalk's built-in computer algebra system.

like MuPAD or use different ways to connect, like using a remote Mathematica server via a TCP socket connection. So far only Mathematica and the built-in calculator can be used to process the results of the mathematical handwriting recognition. In the future, the communication between the `Evaluator`s and the handwriting recognition should become part of the API.

4.7.3 Decoding

When the resource is loaded as raw byte data, it is transformed to data the board can handle. Text data is decoded according to its character encoding.[16] For `text/plain` MIME-type the character is directly passed directly to the board. For `text/html` the text is parsed and converted to plain text by the `echalk.board.util.HtmlDoc` class. The transformation is similar to the formatting done by a text-only browser like Lynx [57], though only a subset of HTML is currently supported. See Figure 4.10 for an example.

An alternative would be to convert the HTML to an image by using the HTML-rendering capabilities of Java. The drawbacks are that using images would need more bandwidth for the transmission, that the Java HTML interpretation often fails due to non-standard HTML data, that its implementation is very heavy weight and perceivably slow.

Images are decoded to an `java.awt.Image` object by using the standard Java image creation methods. Before the image is given to the board, the raw image data are saved in the image resource directory[17] in order to make the image accessible to any E-Chalk clients.

An Applet resource is given by an HTML page where the Applet is embedded and its parameters are defined. The resource's bytes are decoded by transforming them into HTML text, taking the character encoding into account and parsing all `<applet>`-blocks with the corresponding parameters using the class `echalk.shared.applet HtmlAppletBlock`. After storing the HTML data into the Applet resource directory (to make it accessible to E-Chalk clients, see Section 7.2.4), the referenced Applets are loaded using a custom class loader

[16] For data returned from an HTTP request the encoding may be specified in the `Content-type` header field. If unspecified, it defaults to iso-8859-1 (ISO Latin1) [FGM+99].

[17] See Section 4.11.2.

```
*Lars* *Knipping's* *Home* *Page*

[photo: Lars Knipping]

******Lars Knipping******

Freie Universität Berlin
Institut für Informatik

Takustraße 9
D-14195 Berlin

Room:    147 (1st floor)
Fon:     ++49 (0)30 838-75 149
Fax:     ++49 (0)30 838-75 193
email:   knipping@inf.fu-berlin.de

[horizontal bar]

****Projects****

  o Electronic Chalkboard
  o Robocup
  o Map Labeling
  o Estima Ratio

****Publications****

  o Helmut Alt, Lars Knipping, Gerald Weber :
    /An/ /Application/ /of/ /Point/ /Pattern/ /Matching/ /in/ /Astronautics/,
    Technical Report B-93-16, FU Berlin, Institut für Informatik, November 1993.
    Downloadable as Gzipped PostScript (216 kB).

  o Gerald Weber, Lars Knipping, Helmut Alt :
    /An/ /Application/ /of/ /Point/ /Pattern/ /Matching/ /in/ /Astronautics/,
    Journal of Symbolic Computation, 17(4), pp. 321-340, April 1994.
    Downloadable as Gzipped PostScript (193 KB).

  o Lars Knipping:
    /Beschriftung/ /von/ /Linienzügen/,
```

Figure 4.10: The author's home page converted to plain text on the board, only top section of converted page shown.

and initialized. The class loading process also stores all class data in the Applet resource directory. See Section 4.10.1 for further details.

4.7.4 Chalklets

When the user selects a chalklet bookmark, the `ChalkletKit` referenced by the bookmark is loaded as a resource load task. To access the kit class from the class path URL, the loading process uses the `java.net.URLClassLoader` and Java reflection. The `ChalkletKit` is asked for the minimum area size its chalklets need. Next, the `echalk.board.rc.ChalkletHandler` lets the user select the board area the chalklet should live in, as described in Section 2.4.2. The construction of the concrete chalklet instance is done in the handler following the area selection, because the user-selected board area is needed in a parameter to the chalklet building method. In addition to the parameters given by the chalklet bookmark[18], the factory is passed a `echalk.board.rc.ChalkletContext` instance, which serves as a communications interface between chalklet and board. The `ChalkletContext` can be queried about information on the environment the chalklet lives in, for example the location of the chalklet area and the background color of the board, and provides methods for the chalklet to send stroke paintings to the board. When the chalklet instance is constructed, its area is outlined in gray on the board and it is registered as a stroke listener to user strokes drawn in its area. See Section 4.8.1 for a detailed description of the mechanisms for strokes being passed from the server-side board to chalklets and vice versa.

In contrast to images and Applets, the server does not have to mirror the chalklet, because the client does not have to load the chalklet itself. It only receives the strokes produced by the chalklet and saved as board events.

[18] See Section 3.4 for setting chalklet parameters.

4.7.5 Resource Handling

The echalk.board.rc.TextHandler pastes the text to handle directly a text row below the request that produced the text.[19]

The echalk.board.rc.ImageHandler lets the user place the image resource to the board as described in Section 2.4. Applet resources are placed similar by the echalk.board.rc.AppletHandler, see Section 4.10 for the details of Applets instantiation.

The echalk.board.rc.ChalkletHandler lets the user select the chalklet's area and marks it by drawing the area's border. Once the area is determined, the de.echalk.util.ChalkletContext for the new Chalklet can be constructed and the ChalkletFactory is used to build a new Chalklet instance.

The new object is registered to the echalk.board.rc.StrokeSender queue, which handles the forwarding of user-stroke events to the stroke listeners, see Section 4.8.1. After that, the board returns to the default user paint mode.

The echalk.board.rc.MacroHandler lets the user select where the macro should start and then starts to play the macro on the board in an extra thread.[20] Any user action on the board stops the macro. The board behaves like a client while in macro play mode, as pre-recorded events are delivered to the board.

4.8 Stroke Delivery

4.8.1 Chalklet Strokes

The board sends all its users line drawing events as well as its undo and redos of line strokes to the associated echalk.board.rc.StrokeSender instance.[21] Whenever a full stroke has arrived, e. g. when the a drawing is terminated by a mouse-up event, or a stroke is undone, the StrokeSender sends this to its echalk.board.rc.StrokeReceiver. Each registered StrokeListener like a Chalklet is encapsuled StrokeReceiver, which checks the stroke events for their relevance to the listener. In the case of a Chalklet, a stroke is considered relevant if it occurred in the chalklet's observed area and if the stroke was not created before the chalklet became active. The latter restriction is important for consistency when undos and redos occur. Any relevant stroke event is passed on to the listener. That stroke is handled in a separate thread for each of the listeners to avoid having a StrokeListener instance blocking the board. See Figure 4.11 for an illustration of the stroke flow.

When a Chalklet wants to draw into its board area, it sends Strokes to its ChalkletContext.[22] The strokes are put into the board's echalk.board.-

[19]The implementation of the TextHandler also allows to handle text resources without a specified request position. In that case the handler would let the user place the text as in the add text action described in Section 2.4. For the time being, there is no way to produce such text resources.

[20]The rate of board events sent by the macro thread is restricted to 100 Hz to avoid bandwidth bloats. See Section 4.11.1.

[21]Line drawings that are not drawn by the user, like the strokes produced by Chalklets are not collected. The stroke forwarding is purely meant for interpreting human drawing actions. This also prevents infinite loops that might easily occur if Chalklets are allowed to react to Chalklet's output with drawings.

[22]Similar to an AppletContext for Java Applets, the ChalkletContext is the communication interface for a chalklet to its environment it runs in. The ChalkletContext for a chalklet is provided when it is built, by passing it on as a parameter to the ChalkletKit's chalklet build

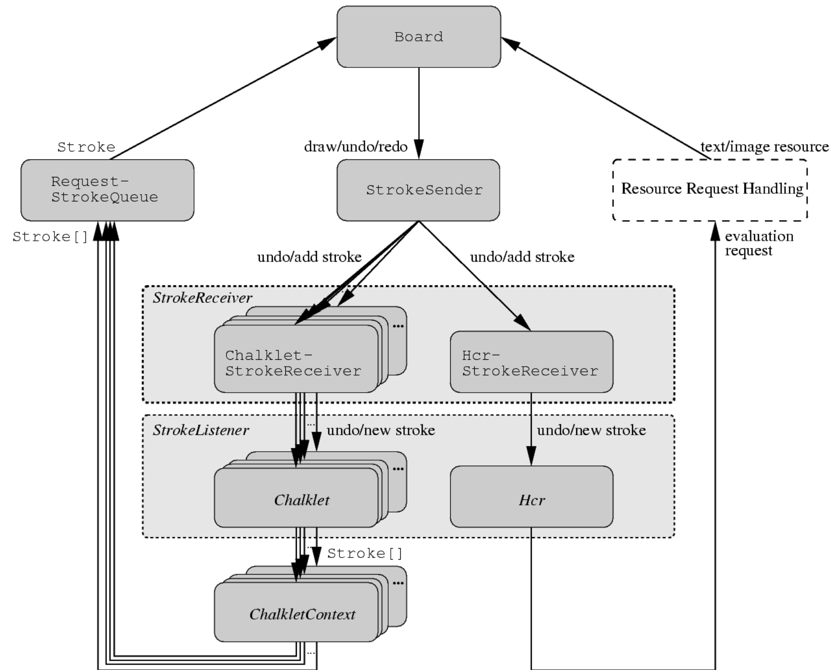

Figure 4.11: Communication between the E-Chalk board and its stroke listeners, the handwriting recognition, and active chalklets.

`rc.RequestStrokeQueue`. This queue has an extra thread which takes the line-drawing data from the queue and sends them to the board. The queue processes strokes using a FIFO strategy. The line segments a `Stroke` object is composed of are associated with timestamps. If a timestamp is in the future, the queue waits for the given time before passing on the line-draw event, otherwise it is processed immediately. If the user is drawing, the draw event is delayed until the user finishes his or her stroke. Users drawings can also split the synthetic strokes into two parts. This is important to keep the complete control of the board to the user. Handling the two drawing actions concurrently would not be a suitable alternative, as this would destroy the connection of the user strokes for undo/redo purposes.

The `RequestStrokeQueue` does not send the strokes' data at a higher rate than one line segment per ten milliseconds to limit the bandwidth taken up by chalklets. As a comparison, user drawings with a typical USB mouse generate 125 line segments per second, while the chalklets are bound to 100 per second. See Section 4.11.1 for a more detailed discussion.

When the board area is scrolled, the position of all active chalklets is checked. When the area of a chalklet is no longer completely visible, it is deactivated. It is unregistered from the `StrokeSender`, its pending strokes in the `Request-StrokeQueue` are removed, and the `Chalklet` instance itself is notified of its deactivation, so it can release any resources it still holds.

---

method.

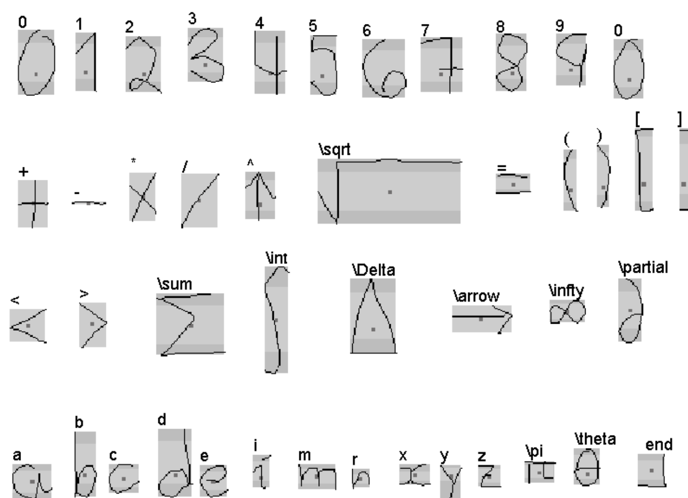

Figure 4.12: Recognized symbols.

4.8.2 Handwriting Recognized Strokes

If the handwriting recognition is activated, it gets the stroke events from listening to the `StrokeSender`, like chalklets do. The main difference is that its stroke receiver tags all strokes as relevant which have the color which is associated with the recognition, see board settings description in Section 2.4.1.

The results of the handwriting recognition are requests to an `Evaluator`, see above. An overview of the included handwriting recognizer is given in the following section.

4.9 Mathematical Handwriting Recognition

The first on-line handwriting recognition developed for the E-Chalk system used a k-nearest neighbor classification. The sequence of strokes was grouped into symbols, based on intersections of their bounding boxes and a constraint that symbols can be only formed by one or two strokes. The recognizer returned its result once a special end symbol was recognized. The approach worked quite well for simple, calculator-type terms. However, the classification performance decreased dramatically once a greater number of symbols were introduced and it was not capable to handle spatial relationships between the mathematical symbols, which are fundamental to the interpretation of more complex formulas.

For this reason, the current recognizer engine uses a two-step strategy similar to well-known approaches [BG96, LW97, CY00]. The first step is the classification of the stroke data, now relying on Support Vector Machines (SVMs) [TR02], whose non-linear classifications [Vap98] yield superior classifications for the increased number of symbol classes. The classification now also handled symbols composed of three strokes. The second step is to apply a structural analysis technique to obtain a hierarchical structure of the expression which describes

the mathematical relationships among the symbols. A detailed description of the on-line mathematical handwriting recognizer can be found in [Tap04].

Data Preprocessing

Strokes delivered by the E-Chalk board are considered part of the same symbol if their distance (considering the strokes brush radius) is below a certain threshold. The online data is then preprocessed [TR03, FKST04] in a standard way for handwriting recognizer. First the direction of the stroke data is normalized. A fixed number of points is then taken from the stroke's list of support points[23], the points are smoothed by averaging the coordinates with those of their neighbors[24]. The points are then resampled equidistantly: the sequence of points $(p_1, \ldots, p_n)$ is replaced by a sequence of points $(q_1, \ldots, q_n)$ having the same distances between successors on the polygonal chain formed by $(p_1, \ldots, p_n)$. Finally, the stroke data are scaled and translated to get a normalize size and position for the bounding box of all strokes belonging to a single symbol.

The feature vector used as input for the classifier is constructed from a number of local and global features. The local features for a stroke of length ℓ formed by the preprocessed points $(p_1, \ldots, p_n)$ are the coordinates of the points p_i, the turning angle[25] between the two consecutive segments (p_{i-1}, p_i) and (p_i, p_{i+1}), the points' relative length position defined as $\ell_i = \sum_{k=1}^{i-1}(||p_k - p_{k+1}||)/\ell$ for the point p_i, and the change of the turning angle[26] for two consecutive turning angles. Global stroke features represented in the input vector are the center of gravity of the stroke points, the length ℓ of the stroke, the relative stroke length ℓ/d, where d is the distant between the first and last point of the stroke, and the accumulated angle, i. e. the sum of all turning angles.

The SVM classifier

Three classifiers are used, depending of the number of strokes that form the symbol. For symbols that there composed of more than one stroke, the feature vector of each stroke is constructed and the concatenation of the feature vectors content forms the full input vector which represents the symbol.

The classifier implementation uses DAG-SVMs [PCST00] as a multi-class SVM-classifier with an RBF kernel for all the classification nodes and the first modification of the *Sequential Minimal Optimization* (*SMO*) algorithm from [Pla99, KSBM99] with $\gamma^2 = 0.001$.

Experimental Classification Results

Experimental data were obtained by having a volunteer writing 50 samples of each of the symbols shown in Figure 4.12 with a Wacom *Graphire 2* tablet. The number of symbols was then artificially increased ten times by means of transformations related to the tangent distance [SM96]. The data were randomly split into three sets. 50 % of the samples were used for training, 25 % for testing, and the remaining 25 % for validation. For comparison, an Artificial

[23] In the current implementation, 16 points are taken.
[24] Smoothing is done using the Kernel $(0.25, 0.5, 0.25)$.
[25] The angles are represented as their sine and cosine values.
[26] Again, the difference angles are represented as their sine and cosine values.

classifier type	strokes	support vectors	error train	error test	error validate
Support Vector Machines	1	29.65 %	0.31 %	0.93 %	0.76 %
Artificial Neural Networks	1	-	0.72 %	1.17 %	1.27 %
Support Vector Machines	2	36.73 %	0.00 %	0.62 %	0.70 %
Artificial Neural Networks	2	-	0.69 %	1.31 %	2.09 %
Support Vector Machines	3	39.41 %	0.00 %	0.00 %	1.17 %
Artificial Neural Networks	3	-	0.00 %	1.17 %	1.17 %

Table 4.1: Experimental classification rates for Support Vector Machines and Artificial Neural Networks with classifiers built for symbols composed of one, two, or three strokes. The column *support vectors* gives the share of training vectors stored as support vectors in the SVM representation.

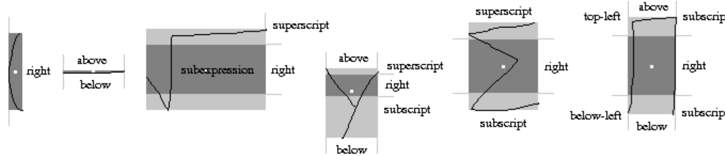

Figure 4.13: Regions, thresholds, and centroids of different symbol types.

Neural Network was trained with the RPROP algorithm applying the standard sigmoid and its standard parameter values [RB93]. The number of hidden units for each one were the same of the dimension of the feature vector. Table 4.1 summarizes the results obtained for each of the classifiers.

Structural Analysis of Mathematical Expressions

Relations in mathematical notation are defined explicitly or implicitly by the position and size of symbols in the expressions [Cha70]. Depending on a symbol's meaning, it can have relations to symbols in regions classified as *top-left*, *above*, *superscript*, *right*, *subscript*, *below*, *below-left*, and *subexpression*. Figure 4.13 shows such regions. For example, the operands of the fraction operator, a horizontal bar, are the numerator and denominator and are expected to lie in the regions above and bottom, respectively.

Mathematical notation is represented by a recognizer in a data structure similar to the one used in [ZBC02], as a hierarchical structure of nested baselines, where a baseline is a list of symbols which represent a horizontal arrangement of symbols in the expression. Each symbol has links to other baselines, symbol lists which satisfy the spatial relations mentioned above.

The method in [Mat99] describes a minimum-spanning-tree construction for groups of strokes, considering the centers of the stroke bounding boxes as nodes of a completely connected weighted graph with a custom weight function. The approach is robust in terms of stroke grouping, but not in terms of structural analysis, so the algorithm is used in the recognizer to build the baseline tree, which is then recursively changed to take the constraints of the above-described spatial relations into account [TR04]. See Figure 4.14.

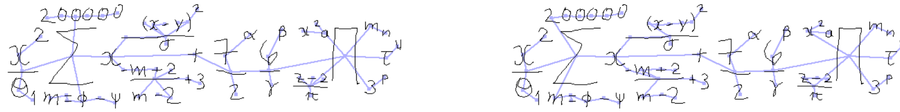

Figure 4.14: Left: Baseline tree after the first recursion. Right: Final tree of spatial relations.

4.10 Applet Control

As mentioned in Section 4.7.3, any Applet added to the board content is loaded by an instance of a custom `ClassLoader`. This object saves a local copy of the loaded classes to make them available for the client-board Applet, see Section 7.2.4. The board then acts like a kind a Java-enabled browser for the Applet: It constructs an instance of the Applet using the default (i. e. the parameterless) constructor, and then sets a `java.applet.AppletStub` by calling the Applet's `setStub` method. The `AppletStub` and the `java.applet.AppletContex` accessible through the stub serve as interface between the Applet and the "browser" it is running in. For example, the Applet accesses its parameters and any other Applet on the same Web site through these classes.[27] Therefore E-Chalk provides its own `AppletStub` and `AppletContext` classes for the Applets to run with.

Actually E-Chalk has two classes implementing the `AppletContext` interface, because Java 1.4 made a change to the interface which is not backward-compatible to Java 1.1. In 1.4 the two methods `getStream` and `setStream` where added to the interface. Unfortunately, a Java 1.4 compatible implementation of the interface will cause an exception when loaded in a Java 1.1 environment with JIT (just-in-time) compiling. The JIT compiler looks at all the classes methods, including the new ones, which use the class `java.io.InputStream` in their signatures. Because this class is not available in Java before 1.2, the process will fail. Using such a class will cause the exception, even in code parts not run due conditional execution. To be able to provide an `AppletContext` to the JVM on both the server side running on 1.4 and the client side, which may run in any Java version from 1.1 on, the appropriate `AppletContext` is loaded dynamically, utilizing reflection and therefore preventing the JVM from JIT compilation of an inapplicable class.

After the `AppletStub` is set, the `init()` method of the Applet is called, where all the initializations depending on parameters should be made by the Applet.[28]

Then the Applet is placed by the user on the board and added as a component to the boards `DrawPanel`, see Section 4.4 The Applet's position and size are set by calling the Applet's `setBounds(java.awt.Rectangle)` method. The Applet's execution begins by calling its `start()` method, after adding a number

[27] A distinct `AppletStub` instance is used for each Applet running in a browser, while an instance of `AppletContex` exists for each HTML page with Applets. The Applets parameter are accessible through the stub and the other Applets can be communicated with through the context.

[28] These cannot be done in the constructor, because at that time an Applet has no access to parameters via the `AppletStub`.

```
for all Applet definitions from HTML page
    load given Applet class c
    instantiate c with default constructor
construct AppletContex (with access to all Applets instances)
for all Applets a
    construct AppletStup s from definition and context
    call a.setStub(s)
for all Applets a
    call a.init()
for all Applets
    user places Applet a on board to Rectangle r
    call dp.add(a) for DrawPanel dp
    call a.setBounds(r)
for all Applets a
    call a.start() in a new thread
```

Listing 4.1: Steps to initialize a number of Applets given in an HTML page.

of event listeners, to be able to record the lecturer's interaction with the Applet, see below.

For multiple Applets from a single HTML page, each of these steps has to be executed for all Applets before the next step is performed, realizing a simple barrier synchronization. For example all Applets must be constructed and made accessible via the stub setting before the Applets are initialized, where they may try to access other Applets. See Listing 4.1 for details.

When an Applet is removed from the board, for example by a clear-all event or by rewind on the client, it is removed from the component list of the `DrawPanel`, its `stop()` and `destroy()` methods are called, the thread the Applet were started in is stopped, and all references to the Applet and its classes are cleared. According to Java's implementation advises on Applets, the `destroy()` method should release any resources the Applet used (like extra threads) and they can then be garbage collected. Unfortunately, it seems that few of the Applets around in the Web adhere to Java's implementation guide.

4.10.1 Applet Events

To be able to transmit the user interaction to the remote clients, the AWT events on the Applet's components have to be logged. To do that, AWT listeners are registered to the Applet and recursively to all its subcomponents. When a listener is notified of a low-level AWT event happening, it registers the event to the board for storing[29], thus the remote client can recreate the event to be handled.

While the lecturer's basic interaction with the Applet can be reproduced, this strategy has a number of problems with achieving an exact replay:

- Changes to the Applet's component hierarchy are not handled.

  The event listeners are only registered to Applet components that exists after the Applets `init()` method returned. When the Applet adds new

[29] See Section 4.11.2 for the event format for logged Applet events.

components afterwards, events delivered to them will be lost in replay. This problem can be solved by adding a `java.event.ContainerListener` to the Applet and all its subcomponents that inherit from `java.awt.-Container` (and can thus contain other components). The listener is then notified whenever an adding or removal of components happens and can add and remove the event listeners accordingly.

- Extra windows are not handled.

  Applets opening their own windows do not only break the board metaphor, but the windows and their subcomponents go unnoticed by the described recursive event listener registration. All events on these components will be lost.

  On the server side, it would be possible to prevent the Applet from accessing the frame display methods via a `java.lang.SecurityManager`, throwing a `java.lang.SecurityException` on an access attempt. The `SecurityManager` can also be used to prevent the Applets from increasing the priority of their threads or accessing E-Chalk's system threads.

  Unfortunately this mechanism cannot be used on the board client to achieve identical results for the replay: the board client is an unsigned Applet and is therefore not allowed to change the system's `SecurityMa-nager`.[30]

- Applets may modify their listener registration.

  The Applet logic may remove E-Chalk's event listeners, for example by removing all event listeners to replace them. Also, the logic of the Applet may rely on the registered event listeners, an Applet state which we changed on the server side.

- Platform or time-dependent Applet behavior.

  If the behavior of the Applet is dependent on random numbers given by the Java system or on execution time or information about the platform, like OS name or Java version, the Applet's behavior on the server side may differ from that on the client side.

  In addition, the different thread scheduling-mechanisms of the underlying operating system may introduce race conditions and result in platform-dependent replay behavior.

  Finally the Applets themselves may rely on special versions of the underlying Java system, for example by using library classes not available in older versions, by relying on running as a signed Applet, or even by depending on the occurrence of bugs fixed in later Java version.

- Different execution speed on server and client.

  The execution speed of the Applet on the client side and on the server side can be expected to differ, as they usually run on different hardware and share resources with different processes. For example when the lecturer

[30] While this security mechanism does not really help for Applets on the board due to its restriction to the server side, it would be worth to integrate it for the use of chalklets, as these are only running on the server side, while the board client only handles their stroke events.

clicks on a moving target displayed in an Applet at the server board and the target moves with a different speed on the client board, the mouse event might miss the target in the replay.

Even worse, the board has no appropriate way to handle time-dependent states in replay for navigation in time, leading to inconsistencies when the user uses fast forward or rewind on the replay.

- Applets reading from the network.

  When the Applet reads from a network connection, the data read will usually not be available for the replay. For example the lecturer might use a chat client Applet, that reads the communication data from a chat server. The chat server must run on the same server the chat Applet came from, as the security model of Java does not allow Applets to connect to different locations. During replay, the chat Applet can only connect to the side where the board replay came from, and usually there will be no appropriate chat server.

- User AWT events generated on client side.

  The remote user cannot be prevented from interacting with the Applet in the replay. By having Applets receive both kinds of AWT events, the lecturer's recorded events and the input at the client side, the Applet may easily reach inconsistent states, for example if it keeps track of the mouse pointer, assuming these is only one mouse while receiving events from two. Even if the internal logic can cope with the remote users events, it will not resemble a replay of the Applet's behavior during recording.

As a summary, replaying Applets in general is a hard problem. Many of the problems disappear when the reproduction of the AWT events is dropped and the Applets are more used as an interactive element in the lecture to be operated by the individual learner. However, this does not fit well to the overall philosophy of E-Chalk lecture recording, where remote participants should get the same information as the listeners in the class room.

For these reasons, the handling of Applets in E-Chalk stayed at a very experimental stage.

4.11 Event Storage

4.11.1 Encoding

All board events are concurrently stored to a file named `board/events` in the lecture recording directory. Applets and images are referenced indirectly in the events file and stored in extra directories, see below. An event file is organized by lines.[31] Everything is stored in plain text (ISO Latin-1 encoding). This makes the events files readable to humans, a great advantage in debugging. On the other side, this results in higher bandwidth use for transmission than for a binary representation. In practice, the bandwidth consumed by the board

[31] E-Chalk's event parsers allow any of the platform-dependent end-of-line markers, single newline, single line feed, or a new line followed by a line feed. The board always saves events with Unix-style line breaks.

is negligible compared to the one used by the audio transmission. As shown in Table 4.2, real lectures need less than 1 kbps in average. The peaks are in the range of 3 to 5 kbps, as standard mouse devices generate between 50 and 125 mouse events per second.[32] Up to one line segment event is stored per mouse event. Considering the syntax of a line form event described below, and assuming x coordinates up to 4095 (hexadecimal FFF), y offsets with scrolling less than 64*1024, lecture duration of up to 4.5 h (less than 16^7 milliseconds), and Unix-style line breaks (single newline character), a line form event needs up to 46 bytes.[33]

Note that for embedded image and Applet resources total bandwidth may peak higher, although these data are loaded concurrently to the event stream and thus do not block the board event's loading.

With the introduction of macros and chalklets, the bandwidth may also peak higher if no limit for their line events production rate is introduced. For this reason, the stroke-handling queue currently constraints the chalklet output[34] to a line-event rate of 100 Hz, roughly the same rate as produced by drawing with a USB mouse. For the same reason, the event rate for a macro is limited to 100 events per second.[35]

Another approach to keep the bandwidth produced by chalklets low enough is to change the event format to a more compact representation. The format should be changed to a binary one with a differential encoding being applied. The event data will compress very well using a differential encoding, because it consists largely of long sequences of line segments with its data entries only slightly changed (position on board) or even identical (color and stroke width). An additional entropy encoding like *gzip* can be applied afterwards, except for live transmission where the events cannot be gathered in packets as they have to be transmitted without delay. While compression will still not allow for an unlimited rate of chalklet events, it permits the limit to be relaxed. As a drawback of using compressed event encoding, the task of providing a repair tool for damaged event files will become much more difficult, see Section 6.6.

For the text-encoded event file, one may consider to use XML encoding. The main advantage of using XML is the availability of generic parsers for the overall structure. Also, one gains a self-documentation of the format with the DTD for the XML data.

While Java includes XML parsing support since version 1.3, there is no XML parser in 1.1. Because the events need to be parsed on both the client and server sides, E-Chalk needs to implement its own event parsing anyway.[36] Therefore

[32]Serial mice report at a rate of 33 Hz, PS/2 mice default to 40Hz on Win98/ME and to 60 Hz on WinNT/2000, but can sometimes be speeded up to a rate of up to 200 Hz. USB mice have a default sampling rate of 125 Hz, wireless mice often below that. The data rates of commercial pen-input devices tested by the author (*Numonics IPM* whiteboard, Hitachi *StarBoard*, different Wacom Tablets) range from 50 Hz to 80 Hz. Specifications for high-end graphic tablets (Wacom *Intous2* and *Cintiq*) state up to 205 Hz, but in practice rates of about 110 Hz are produced.

[33]Logging mouse-move events on Applets may produce even bigger events, as the size depends on the depth of the Applet's component tree, see below for Applet event formats. Because Applet support in E-Chalk is rather rudimentary, it is not analyzed here in detail.

[34]See Section 4.8.1.

[35]Macros are nevertheless able to produce undesirably high data volumes by rapidly loading images. A mechanism to limit the bandwidth in this scenario still needs to be introduced to the server board.

[36]A parser would be especially difficult to realize for the lecture repair tool described in

topic	length min:sec	events total	events /s	bytes total	bytes /s
bubblesort (capsule) [81]	12:16	8124	8.3	335297	455.5
quicksort (capsule) [82]	20:51	12910	10.3	528955	422.8
wavelets [83]	73:27	58661	13.3	2570274	583.2
colors [84]	71:00	38535	9.0	1663785	390.6
two's complement [85]	16:43	14352	14.3	596287	594.5
eigenvalues [80]	102:12	70039	11.4	2975178	485.2
eigenvalues [87]	103:18	52897	8.5	2244007	362.0
eigenvalues [111]	102:51	77080	12.5	3292172	533.5
optics [96]	94:03	47334	8.4	1985525	351.9
thermodynamics I [97]	91:57	42202	7.6	1829194	331.6
disc. Fourier-transformation [6]	89:24	88233	16.5	3832611	715.8
polynomial interpolation [7]	79:38	78190	16.4	3362383	703.7
surface integrals [98]	99:11	90992	15.3	3978159	668.5
stokes' theorem [99]	89:23	89913	16.8	3919085	732.1

Table 4.2: Recorded E-Chalk sessions from courses at Freie Universität Berlin and at Technische Universität Berlin. Capsules denote short modules recorded without listeners, the other entries are produced as full lectures held with E-Chalk. Horizontal lines separate recordings from different authors.

it makes more sense to implement a simple custom format that is both better readable for humans and simpler to parse by programs. Also, using XML would introduce a structuring overhead that would noticeably increase the size of the representation.

4.11.2 Structure

The first five lines form the header of the file, starting with the *magic* file marker `ec1` in the first line, followed by width and height of the board in decimal ASCII representation, a line containing the lecture's title, and finally the background color. All colors in the event file are stored as hexadecimal α-RGB values, even though α-values are not yet supported. Any α other than `FF` (intransparent) is currently ignored.[37]

Each of the following lines contains an event, given in the order of creation time. An event line is separated into tokens by the character `$`. The first token is always the timestamp of the event, given in milliseconds relative to the lecture's start time. It is saved in hexadecimal formar, like all numerical values in the event file if not stated otherwise.[38] After the timestamp, tokens specify the type of event, followed by tokens for the event-specific parameters. Any extra trailing tokens are ignored. In the following events, these optional entries are only given for the NOP operation, where they are used for commentaries, see

Section 6.6, which then would have to be able to parse files in an erroneous format.

[37] Transparency for images is supported, but the color values of images are not saved directly in the events file, see below on image events.

[38] Any usage of decimal numbers is due to historical reasons. Examples are the numbers for image and Applet resource file and the decimal encoding for board width and height entries in the header.

```
ec1
680
420
Example Lecture
ff000000
0$Nop$created 03-10-28 14:50:36 GMT+01
ed3$Form$Line$48$2a$48$59$ff00ff00$3
efe$Form$Line$48$59$47$65$ff00ff00$3
[...]
31e9Form$Line$218$117$218$118$ff00ff00$3
4d2e8$Nop$end 03-10-28 14:50:52 GMT+01
4d2e8$Nop$append 03-10-28 15:05:13 GMT+01
5325$Scrollbar$6
5340$Scrollbar$16
[...]
54a5$Scrollbar$13c
6567$Form$Text$$42$19e$ff00ff00$c
6e0c$Text$SetTxt$plot([t,sin(t),t=-Pi..Pi]);
7406$Text$End$plot([t,sin(t),t=-Pi..Pi]);
83b6$Form$Image$0$3e$1a8
908a$Undo
[...]
```

Listing 4.2: Excerpt from a board event file.

below.

Nop, Scroll, Clear All, Undo, Redo, Terminate

The simplest event is NOP, having no mandatory parameters:

> *timestamp*`$Nop`{`$`*plain ascii text*}

The board creates these at the start and end of a recording as commentary entries: date and time are stored as a commentary token, as well as the creation mode (created or appended). See Listing 4.2 for an example.

The events for clearing the whole board, for undo and redo also do not need any parameters, and scrolling events use the single parameter *scrolloffset*, the new vertical offset of the board's top position:

> *timestamp*`$RemoveAll`
> *timestamp*`$Undo`
> *timestamp*`$Redo`
> *timestamp*`$Scrollbar$`*scrolloffset*

The terminate event is exclusively used in live transmissions to signal to a client the end of transmission, see Section 4.12:

> *timestamp*`$Terminate`

Line Segments

A line segment from point (x_0, y_0) to point (x_1, y_1) drawn with stroke radius r in color c has the form:

$timestamp\texttt{\$Form\$Line\$}x_0\texttt{\$}y_0\texttt{\$}x_1\texttt{\$}y_1\texttt{\$}r\texttt{\$}c$

with all numbers given in hexadecimal. Again, see Listing 4.2 for an example.

Images

An image added to the board with (x, y) being the position of the upper left corner is represented as

$timestamp\texttt{\$Form\$Image\$}id\texttt{\$}x\texttt{\$}y$

with the image itself is stored in the file `images/`*id*`.dat` in the lecture session directory. The *id* parts are created as ascending decimal numbers, but the event-file parsers do not rely on this.

Applets and Applets Events

An Applet with (x, y) being the position of the upper left corner is represented as:

$timestamp\texttt{\$Form\$Applet\$}id\texttt{\$}x\texttt{\$}y$

with the HTML file containing the definition being stored as `applets/`*id*`.html`. Again, the file names are created as ascending decimal numbers.

Input events on Applets are also encoded. The event

$timestamp\texttt{\$AppletEvent\$Mouse\$}n\texttt{\$}id\texttt{\$}mod\texttt{\$}x\texttt{\$}y\texttt{\$}clicks\texttt{\$}popup\{\texttt{\$}comp\}$

encodes a mouse or mouse-motion event for the Applet, internally numbered n. The integer entry *id* is the Java event identifier for distinguishing between mouse pressed, mouse released, mouse moved, etc. and *mod* is the integer modifier key mask for Shift, Ctrl, Alt, and/or Meta key down. The values (x, y) give the position of the mouse event in its triggering component, *clicks* is the click count for this event, for example for identifying double clicks, and *popup* is `true` or `false`, marking the mouse event as a pop-up trigger on the platform where the event was created.[39] The originating component is encoded by a trailing list of indices: the sequence $n_0\texttt{\$}\ldots\texttt{\$}n_{k-1}\texttt{\$}n_k$ means the event is on the n_kth subcomponent of the n_{k-1}th subcomponent etc. of the n_0th subcomponent of the Applet.[40] If the list is empty, the mouse event is on the Applet itself.

Action events and key events have the form

$timestamp\texttt{\$AppletEvent\$Action\$}n\texttt{\$}id\texttt{\$}actionname\texttt{\$}mod\{\texttt{\$}comp\}$
$timestamp\texttt{\$AppletEvent\$Key\$}n\texttt{\$}id\texttt{\$}keycode\texttt{\$}mod\{\texttt{\$}comp\}$

[39] The pop-up-triggering mouse event is platform-dependent. For example, on Mac OS X, the pop-up menus are triggered by the mouse button being pressed down, on Windows on the mouse button being released.

[40] To identify the event component, the client relies on the subcomponent enumerating methods returning the elements in the same order. As the Java API does not specify this order, it may differ among different Java environments, even though this has not been observed in tests of E-Chalk so far.

with *n*, *id*, *mod*, and component sequence given, the same as for mouse events. The entry *actionname* is the command string associated with the action event, *keycode* is the integer key code of the key typed, pressed, or released with the event.

Text and Text Input

The event for creating a new text form takes the following arguments: *string*, its initial value; (x, y), the inital baseline position for the text; *c*, the text color; and *size*, the font size. The event is defined by

> *timestamp*`$Form$Text$`*string*`$`*x*`$`*y*`$`*c*`$`*size*

where the string is MIME-encoded to ensure encoding-independent representation. As long as the text is in edit mode, it features a text-input cursor and accepts editing changes from the user. The text-creating event positions the cursor at the end of the text. When the user presses the *enter* key, the text form is closed by a

> *timestamp*`$Text$End$`*string*

event and the cursor is removed. The *string* text stored in the event is the MIME-encoded final content of the text form including any line breaks from the board's dynamic line breaking. The client board does not need this information for replay, but it is used by the PDF converter. With this extra field, the PDF converter does not have to recalculate the text layout from the automatic line breaking done by the board.

A text is also automatically closed when a new `Form` is created. or when an `Undo` event happens. This means there can never be more than one `TextForm` receiving key input.

A key input of the printable character *c* for an open `TextForm` is represented as

> *timestamp*`$Text$Char$`*c*

with *c* being MIME-encoded. Some control characters, *backspace* and *delete*, and the cursor-moving *left* and *right arrow* keys have their own events:

> *timestamp*`$Text$Backspace`
> *timestamp*`$Text$Delete`
> *timestamp*`$Text$Left`
> *timestamp*`$Text$Right`

When the user sets the whole text content using the text history (with the *up* and *down arrow* keys), the event

> *timestamp*`$Text$SetTxt$`*string*

is used to set the `TextForm`'s content to the content of the MIME-encoded *string*. For pasting from the clipboard, a *string* is inserted at the current cursor position. The event

> *timestamp*`$Text$Str$`*string*

with the MIME-encoded *string* is used. Finally, for bringing the cursor to the beginning of the text with the *home* key or to the end with the *end* key, the event

timestamp`$Text$Cursor$`*idx*

allows to move the cursor to an arbitrary character index *idx*.

4.12 Live Server

The board's live server thread listens at the TCP server socket for incoming connections. When a client connects, a connection thread is spawned for the client and the offset to the board's initial time index (the offset to event timestamp 0) is sent as hexadecimal ASCII followed by a newline. This enables the client-board Applet to synchronize its timing with the server-board timestamps. Then the current content of the event file is flushed over the socket connection. The connection is maintained until the server or the client terminates. Whenever a new event is created on the board, is it sent over to all open client connections. For a description of the client side, see Section 7.2. When the board session ends, it sends clients a single terminate event, see Section 4.11.2.