Diploma Thesis:

Using Sketch Recognition to Enhance the Human-Computer Interface of Geometry Software

Dirk Materlik

December 2003

Supervisor:
Prof. Dr. Ulrich Kortenkamp

Freie Universität Berlin
Institut für Informatik

Takustr. 9
14195 Berlin

Author:
Dirk Materlik

Zossener Str. 16
10961 Berlin

materlik@inf.fu-berlin.de

Abstract

Recent advances in computer technology allow the use of Dynamic Geometry Software on new, pen-driven devices, such as Interactive Whiteboards and PDAs.

However, user interfaces designed for desktop use are awkward on those devices and do not take advantage of the characteristics of pen-based input. This thesis addresses adapting the user interface of the Dynamic Geometry Software *Cinderella* to pen-driven devices by using stroke recognition. Different usage scenarios are analyzed and two approaches to stroke recognition are implemented: *Scribbling* and *Cinderella Flow Menus*. It is shown that together, they significantly enhance the user experience in those scenarios.

Thanks

First of all, I thank my supervisor Ulrich Kortenkamp for insights into Cinderella code and geometry in general, for supplying me with all the hardware and gadgets I needed, a work place at the university, and a license for the excellent IDEA IDE. And of course for always answering my questions and advising me on the best way to proceed. Thanks to Jürgen Richter-Gebert for the idea and a prototype of a *scribble* mode and letting me use his idea for my diploma thesis. I thank Enno Brehm for discovering *flow menus* and pair-programming the Cinderella version with me.

Also a big thank you to the “beta-readers” who read earlier versions of this text and provided valuable opinions, namely Lukasz Pekacki, Hendrik Steller, Holger Materlik and most of all Miriam Busch.

Typographic Note

We use *emphasized type* to highlight technical terms that are used for the first time and explained nearby.

When talking about Java implementations, we use `a monospaced font` when referring to actual class names and `methodNames()`. When a class name is used in normal font, the normal English meaning of the word is intended.

Contents

1 Introduction

To familiarize the reader with the environment that this thesis was written in, we first give a short introduction to the research area of *Dynamic Geometry Software* in general. We then introduce *Cinderella* [6] in particular. Following that, we describe the topic of the thesis. At the end of the introduction, we outline how this document is structured.

1.1 Introduction to Dynamic Geometry Software

Think of a page in the Geometry chapter of a mathematics schoolbook [4]. It probably contains an illustration of some geometrical fact, e.g., the three angular bisectors of a triangle intersecting in one point. The illustration might look like figure 1. The geometric relationship holds for any triangle; however, the picture only shows a single example. For a textbook, there is no easy way to convey intuitively that this is true for all triangles. *Dynamic Geometry Software* utilizes computers to allow interactive exploration of a geometric construction, as visualized in figure 2. The free points, here the vertices of the triangle, can be moved with the mouse. The constructed elements, here the angular bisectors and their intersection, are moved accordingly. The user experiences that, no matter how the vertices are moved, the bisectors always intersect in one point.

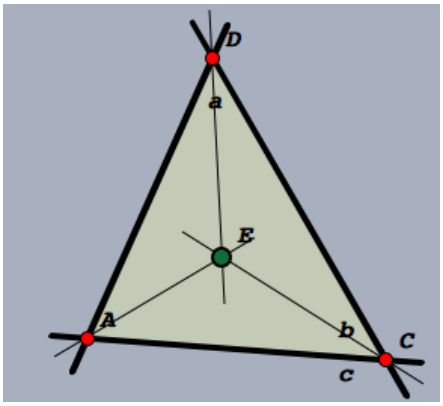


Figure 1: A static figure

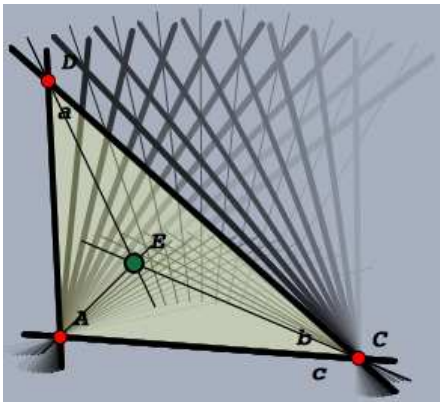


Figure 2: An “interactive” figure

The difference to vector-based drawing programs or CAD systems thus is that Dynamic Geometry Software internally stores relationships of the elements. This is done implicitly as the construction is created. When one of the starting points is now moved, the computer recalculates the positions of all dependent elements. The user is presented with the illusion of a continuous, smooth movement. This allows him or her to see how the movements of the starting points affect the other elements, and thus intuitively grasp geometric relationships.

We will describe typical elements of a construction in a Dynamic Geometry Software in section 2.2, when describing the planning phase.

An important problem for Dynamic Geometry Software is that in the computer context, time is not continual: when the user moves the mouse, the software is given a sequence of discrete pointer positions. Therefore, it must decide at every time step how to draw dependent elements on the screen. These should move along with the moving element in a way that "makes sense." What makes sense in the general case is surprisingly difficult do define mathematically [4, 5].

Cinderella [6, 16] is a Dynamic Geometry Software that solves the continuity problem well by using tracing. It uses projective geometry with complex coordinates. It supports more views on a construction than the classical Euclidean view, e.g., a spherical projection or hyperbolic views. A prover is integrated into the software. It proves incidences and notices when a new element is equal to one already in the construction. Another contribution of the software is the generalized handling of *loci,* the traces of elements when other elements are moved on a "road," like a line or a circle.

The program is implemented in Java, a cross-platform object-oriented language. The design of Cinderella follows the object-oriented paradigm.

Figure 3 shows the main window of Cinderella, with an Euclidean view on a construction defining a parabola using a locus. In the upper part of the window, there are two toolbars. The upper one contains actions that act on the whole construction, like *Load, Save, Undo, Select All* etc. The one below shows a selection of *modes,* e.g., the currently selected *Move* mode and the *Add a Point* mode next to it. Below the main

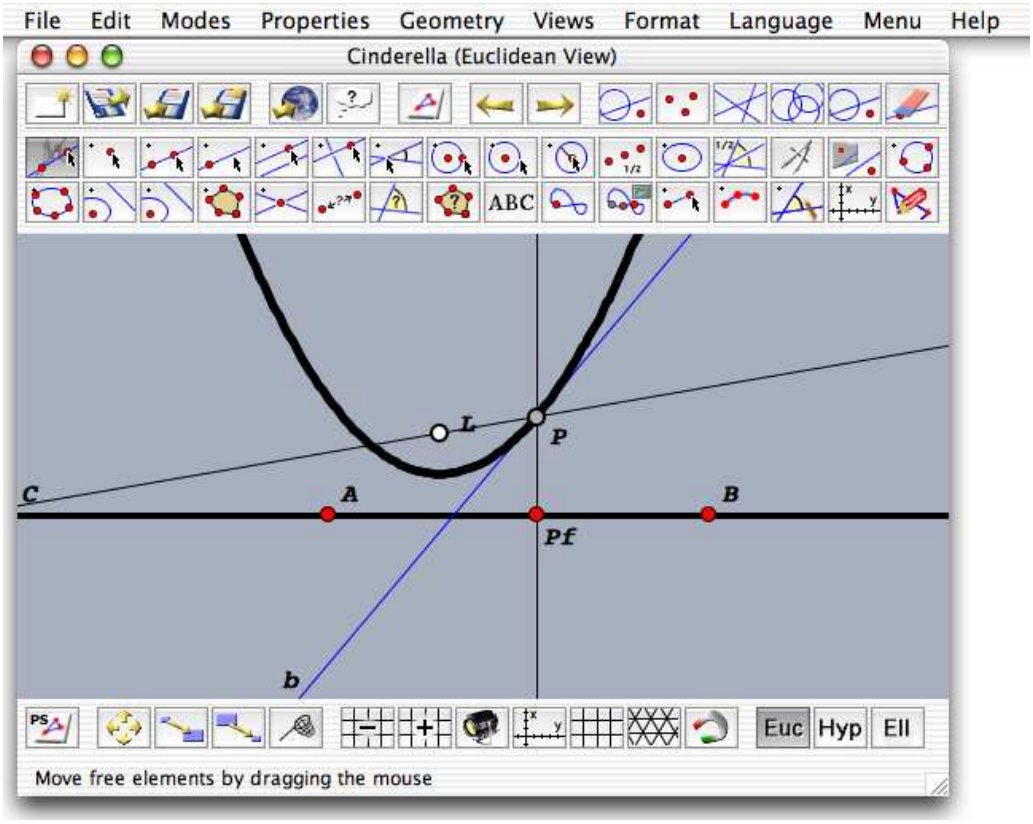


Figure 3: The main Cinderella window

construction area follows a toolbar that is specific to the view; here, modes for modifying the currently viewed part of the construction or rendering options like displaying a grid can be found. A menu bar is used for less common actions[1].

Cinderella is targeted at multiple audiences and usage scenarios. For one thing, it is useful to the academic who needs to visualize geometric relationships. It is also useful if the researcher wishes to give a presentation or lecture: because the geometry is dynamic, moving pictures can easily be integrated into the talk. Constructions can be created during the talk, facilitating comprehension.

It is also useful for teachers in classroom situations. Instead of struggling with the classic equipment of blackboard, chalk, and oversized rulers, a teacher can simply

[1]Since this screenshot was taken on a Macintosh, the menu bar is at the top of the screen; on other platforms, it is attached to the window.

draw an exact construction and use a projector to present it. Using moving constructions, lessons are bound to become more interesting. Dynamic Geometry Software also enables presenting more complex constructions, as the time it takes to draw a figure is reduced.

A more innovative use of Cinderella in the classroom is to let students themselves use the software. Interactive constructions allow students to develop a geometric intuition. They are rescued from doing manual constructions with pencil and paper that never come out exact. Instead, they study an arbitrary number of configurations of the construction and may learn through exploring.

Cinderella also allows creating *interactive exercises* that students can solve on their own.

1.2 Topic of this Work

Recent advances in computer technology enable the use of Dynamic Geometry Software in new scenarios. *Personal Digital Assistants* (PDAs) are palm-sized handheld computers originally developed for keeping track of addresses, appointments, and so on. They have become powerful enough to run Dynamic Geometry Software. In particular, Cinderella can be run as it is written in Java. PDAs are preferable to laptops or desktop computers in some situations because of portability. In other situations, e.g., in a classroom situation, they are preferable because they get less in the way of human-human interaction.

Interactive Whiteboards have become affordable enough to be in wide use. They can enhance the aforementioned presentation situation. Previously, the presenter had to concern him- or herself with the laptop computer while using Dynamic Geometry Software. Interactive whiteboards allow input from the natural position of the presenter, in front of the audience.

Both of these new devices are *pen-driven* – a physical object, the pen, is used to control the pointer position. This is very different from the conventional mouse-and-keyboard-based input model, because the logical position of the pointer always

corresponds exactly to the physical position of the user's hand. While these devices are compatible with programs that expect normal mouse input, users can benefit from adapting the software to this input model. Using the pens of these devices as we use regular pens naturally leads to *sketch recognition,* the analysis of freehand sketches executed by the user. We believe sketch recognition can play an important part in the adaptation.

This work focuses on changes to the Interactive Geometry Software *Cinderella* to enhance its usability using sketch recognition techniques, with a focus on, but not limited to, pen-driven devices.

1.3 Outline

In section 2, we analyze the usage scenarios of Cinderella that shall be improved. We then give an overview over previous work in sketch recognition, both academic and in specific software.

During the course of this thesis, two different approaches to sketch recognition were actually implemented: *Scribbling* and *Cinderella Flow Menus*. In section 3, we explain the design and implementation of the main new module in Cinderella, *Scribbling.* This module analyzes user-drawn sketches. The different ways that this new module is used are introduced in section 4; one of them is the creation of geometric objects for a construction.

In Section 5, we show the design and implementation of *Cinderella Flow Menus*, first explaining the previous work that they evolved from. The different places they are now used in Cinderella are introduced afterwards.

Finally, section 6 concludes this thesis by evaluating if and how these changes in the user interface improved human-computer interaction for the target scenarios. It then presents some ideas for future work and research before ending with finishing remarks.

2 Planning

This section describes the planning phase of the thesis. We first analyze the scenarios for which we wish to improve the human-computer interface. Afterwards, we describe common geometric objects of Dynamic Geometry Software, so we can plan which elements to support in the new interface. We then give an overview over previous work in stroke recognition, both from the academic domain and specific software. A discussion of the existing sketch recognition implementation for Cinderella follows. Finally, the last subsection explains our decisions about what to implement to improve the user interface in the target scenarios.

2.1 Analysis of Target Scenarios

2.1.1 Overview

This work aims to improve the Human-Computer Interface of Dynamic Geometry Software by using *stroke recognition*. A *stroke*, to us, is the movement of a pointing device. Usually a stroke is started by pressing the primary mouse button and ended by releasing it. Stroke recognition, then, means analyzing strokes to figure out the user's intention of a particular movement. Of course, the normal user interface elements like toolbars and menus can be seen as one way of stroke recognition; however, we use the term only for non-traditional user interface elements.

There are different ways that stroke recognition could be used advantageously in Dynamic Geometry Software. One possibility is recognizing elements directly, e.g., recognizing a circle from a sketch of a circle. Another is using a circle-shaped stroke to select a circle-creation mode.

The practical part of this thesis, actually implementing a stroke recognition facility, is done for Cinderella. To decide upon the way the stroke recognition facility is designed, implemented and integrated into Cinderella, we must first analyze the scenarios that we wish to enhance. We consider four different scenarios that could benefit from sketch recognition in different ways: presentations using Interactive White-

boards, PDAs, using desktop computers with graphics tablets and with regular mice. However, the first two are given priority because they are only poorly supported with the regular version of Cinderella.

In the following subsections, we first explain the scenario and then enumerate the special characteristics of the scenario that could affect the new stroke recognition facility. We explain the shortcomings of the current user interface in the scenario and finally give conclusions that should be kept in mind while implementing the changes.

2.1.2 Presentations with Digital Whiteboards

A *Digital Whiteboard* aims to replace conventional black- or whiteboards with computer-based devices. The picture from the computer is projected onto the whiteboard using a projector. The board reports the position of a special pen back to the computer, where this data is interpreted as regular mouse movements. All this normally should be transparent to the application running on the computer. Figure 4 shows a whiteboard made by Numonics, attached to a computer running the normal version of Cinderella [18, 19, 20].

The usage scenario we intend to support here is chiefly that of giving a lecture or talk. A mathematics teacher might use Cinderella for Geometry lessons, or a researcher could use it to demonstrate a point during a presentation. We want to preserve the advantages of traditional blackboards: the way a construction can be developed gradually while explaining it to the audience, the easy way corrections can be made and the possibilities of interacting with the audience while producing the construction.

On many whiteboards, there are at least two virtual mouse buttons available. The primary one is activated by pressing the pen down on the surface of the board and the secondary one by a button on the side pen. Also, these boards mostly support the differentiation between mouse drags and mouse movements, i.e. movements while touching the board or moving slightly above it. However, it should be noted that there are also touchscreen-based digital whiteboards. These can only simulate a single mouse button and cannot know where the pen is when it is not pressed down. The

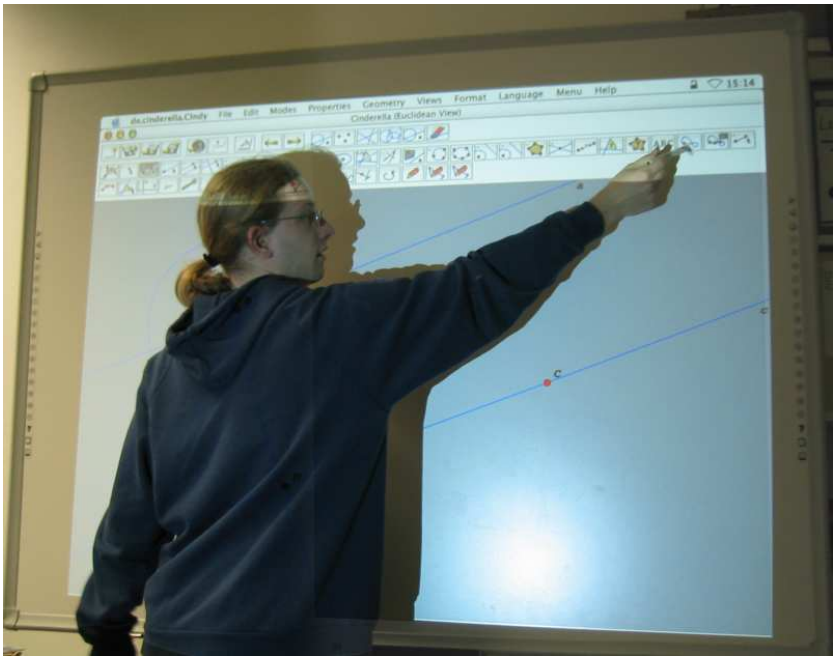


Figure 4: A Numonics Whiteboard attached to a computer running Cinderella with its normal user interface.

resolution of an interactive whiteboard is normally as good as that of the projector.

One reason that the usage of the normal user interface is awkward with these boards is the software's reliance on a central toolbar to select the current mode. The toolbar is in a fixed position at the top of the screen, and switching modes means having to move the hand a long way physically two times, to the toolbar and back again; the problem is clearly visible already in figure 4. This is very different to normal mouse usage, where all of the screen can be traversed quickly. The issue is exacerbated because using Cinderella normally involves switching modes quite often.

Another annoyance is the use of separate windows to change the properties of elements. Moving, resizing and closing windows is more cumbersome on a whiteboard, again because moving the pointing device means physical movement. Also, the nature of a window is that it is now possible to interact with the program in two separate ways; however, while giving a talk, the focus is usually on one task at a time, so these separate interaction possibilities distract the audience.

In summary, we should try to minimize the necessity of often moving the pointer large distances. One way to do this is to reduce the number of mode switches necessary to create a construction. Directly recognizing geometric objects from the stroke should be possible at least for the "basic" objects, such as lines, circles and points. Alternately, we could find a way to switch modes without moving the pen far from the area it is currently in. Additionally, we need a way to change the properties of elements without moving the pen far and preferably without opening new windows.

2.1.3 Geometric Pocket Calculator

PDAs such as the Sharp Zaurus shown in figure 5 feature hardware that is as powerful as that of desktop computers a few years ago[2]. They are fast enough to run a Java Virtual Machine and a software package such as Cinderella. This opens up new, interesting possibilities for the use of Interactive Geometry Software.

We see two main scenarios in which a PDA equipped with geometry software can be used where such software was previously not available. The first is as a *geometric pocket calculator*, i.e. a device that does for geometry what regular pocket calculators do for arithmetic. Using a handheld, portable, and thus highly available device, the user can create exact constructions and quickly visualize geometric relations. This is superior to using pen and paper, because the drawing will be exact and the software can prove relations. Furthermore, since the same software can be run on the handheld and desktop PCs, exchanging constructions is very easy. This makes it possible to later build upon ideas sketched on the PDA.

The second scenario in which a PDA could be a better choice than a notebook or desktop PC is for classroom use in schools. Normal computers establish a barrier between teacher and students due to large monitors that are directly in the line of sight and input devices that need a lot of desk space. When every student uses a PDA-style device, normal communication is preserved. Again, this is similar to the way standard

[2]Older Zaurus models have an ARM processor that runs at about 200 MHz, newer models have an Intel XScale processor with 400 Mhz. All models have at least 20 MB of RAM and 10 MB as permanent storage.

Figure 5: A Sharp Zaurus SL C-700 running Cinderella

calculators are used in schools. Additional interesting possibilities emerge when taking into account that these PDAs can be networked wirelessly[3]; however, that is not the focus of this work.

It should be noted that the current PDA models are not yet quite ready for use in schools; they are still relatively expensive and fragile. However, the direction of technological advance is clear, and it therefore makes sense to research productive usages now. Current models, e.g. the Sharp Zaurus lineup, are advanced enough to be used as prototypes and for initial on-site projects.

From a technical viewpoint, a PDA is more restricted than a normal PC. The screen is small and has a small resolution, typically around 320x240 pixels[4]. However, because the screen is so small, there are many pixels per length unit, which makes exact pointing at small user interface elements difficult. Also, the use of a touchscreen and

[3]The extension card on top of the PDA in figure 5 is a Wireless LAN compact flash card.

[4]The high-end model of the Sharp Zaurus series depicted here features a screen of 640x480 pixels. This is not the norm, however.

stylus means there is only a single mousebutton and no mouse movement, since the stylus can only be detected when depressed on the display. The processor is relatively slow and the memory smaller[5]. Due to these more limited resources, less mouse events are delivered than on a desktop.

The use of toolbars is awkward in this scenario as well. The screen space is much too limited to waste much of it on a toolbar. Using a menubar instead works, but requires too much exact pointing to be really comfortable. Using more than a single window is not advisable, because the window manager on a PDA, if there is one at all, is usually quite primitive. It is much harder to navigate from one window to the next, and manipulating them is more difficult than on a desktop because of the exact pointing required.

An important goal of sketch recognition on PDAs should be speed; the smaller resources available here must be kept in mind while developing the sketch recognition algorithms. Additionally, the sketching recognition should degrade gracefully when events get lost or less of them arrive per time interval.

Really complex constructions are not feasible on such a limited device. Therefore, directly recognizing strokes to geometric objects will be feasible for many of the constructions that make sense on a PDA.

2.1.4 Graphics Tablet

A scenario we will consider with lower priority is that of a normal computer with a graphics tablet attached. This is a peripheral device that detects the position of a pen on a surface and also the intensity with which it is pressed onto the surface. They usually have at least two buttons and a very high sampling rate and resolution.

We consider this scenario because it shares many characteristics with the previous two: chiefly, the use of a pen on a surface as an input method. Therefore, while the normal user interface is quite usable with a graphics tablet as well, sketch recognition might improve the human-computer interaction. For example, not having to move the

[5]We currently consider the abovementioned 200Mhz of processor speed and 32MB of RAM the absolute minimum requirements.

hand from the construction area to the toolbar to select another mode could save time and be preferable to some users.

Technically, this scenario is very similar to the normal mode of operation: the driver for the graphics tablet simulates mouse movements and button presses, and the resources of the computer running the application are the same as without the tablet.

2.1.5 Traditional Pointing Devices

Finally, with low priority, we also consider whether the stroke recognition developed for the other scenarios might also improve the user interface in desktop Cinderella usage. Other applications use *gestures* to initiate certain actions, e.g. opening a new browser window; we will experiment whether something like that is useful for Geometry Software as well.

It may be useful to distinguish between different classes of traditional pointing devices. In addition to mice, trackpads are in widespread use on notebooks. Since they are used with a single finger, the mode of operation seems similar to a pen-driven input device. Some notebook manufacturers use small sticks in the keyboard as pointing devices; it is probably not feasible to use any stroke recognition facility with those.

2.2 Typical Elements in Dynamic Geometry Software

In Dynamic Geometry Software such as Cinderella, we can expect to find the following basic geometric elements. Elements have one output and inputs. For some types of elements, the inputs completely define the element; others still have some degrees of freedom.

Points

Free Points: These are the basic free elements. They have no input elements and can be moved around freely.

Point on Road: A Point that is on another object, e.g. a circle, and can only be moved on that object. It thus has an input object but can still be moved.

Intersections: A Point that is defined by the intersection of two input objects, e.g., lines or circles. In general, there may be more than one intersection, the disambiguation required can be viewed as another input.

Midpoints: A Point that is defined by two input points, lies on the connecting line of the two inputs, and has the same distance to both.

Lines

By Two Points: A line that is defined by two input points. If the line is rendered only between the defining points, this element can also be called a **Segment**.

By Point and Angle: A line that is defined by a point and a slope. The input point is incident to the line and the angle at that point is given to a fixed reference axis.

Parallel: A line defined by a line and a point. The parallel's slope is equal to that of the input line. It passes through the input point.

Orthogonal: A line defined by a line and a point. The orthogonal has a right angle with the input line. It passes through the input point.

Angular Bisector: A line defined by two lines. It is incident to their intersection point and has an equal angle to both. There are two possible angular bisectors for any two lines, so the disambiguation can again be viewed as an additional input.

Circles

By Midpoint and Border Point: A circle defined by two points, one is the center, the other lies on the circle.

By Midpoint and Radius: A circle defined by a point and a radius.

By Three Points: Defined by three points that lie on the circle.

A Pair of Compasses: The output is a circle. It is defined by three points. One of them is the center, the radius is given by the distance between the other two.

Conic Sections by Five Points: Five points that lie on a conic section define it.

It is desirable to support as many of these objects as possible by recognizing them directly from a stroke. It may be difficult to find adequate stroke descriptions for those elements that are defined not only by other elements but also by a number, in particular "Line By Angle," "Circle By Midpoint and Radius."

"Compass" is challenging because some of its input objects can lie far away from the position of the output object. Conic sections are difficult because they can be disjoint in the Euclidean plane, e.g., hyperbolas.

However, the others should be recognizable directly.

2.3 Evaluation of the Previous Scribbling Implementation in Cinderella

2.3.1 Capabilities

Professor Jürgen Richter-Gebert of TU München already implemented a Scribble mode for Cinderella that works well for certain cases and has been demoed successfully at various opportunities. However, it also has certain deficiencies that made a fresh look at the problem seem a good idea.

To avoid confusion, the term *ScribbleJ* or *ScribbleJ mode* is used to refer to Richter-Gebert's implementation. When *Scribbling* is used alone, we mean the implementation developed in this thesis.

We will first give an overview over the strokes that the ScribbleJ mode recognizes. They are split into strokes that create a new object, listed first, and *gestures* that invoke some other function of the software.

Points

Free Points: These can be scribbled as a shape that is confined to a relatively small area. The area determines the size of a newly created point. There is a list of permissible point sizes. Drawing a point-defining shape over already existing points can increase their size; decreasing the size of points is not possible.

Midpoints: Selecting exactly two points before drawing a point anywhere defines a midpoint.

Lines

Segments: Drawing a scribble that is relatively straight creates a segment. The end points are created implicitly if they do not exist.

Orthogonals: Directly after drawing a segment that intersects another segment or line already in the construction, the area around the intersection can be annotated with a small scribble that crosses both segments. This will make the new segment a line that is perpendicular to the already existing line.

Parallels: Directly after drawing a segment, you can annotate first an existing line or segment and then the new segment with a small, relatively straight scribble. This will make the new scribble parallel to the already existing element.

Arrows: Drawing an arrow-like stroke over a line will set the arrow of that line close to the position the arrow was drawn.

Springs: Experimentally, code can be enabled that tries to recognize zig-zag-lines as Springs, from the *Physics Simulation* module of Cinderella.

Circles

By Midpoint and Radius: Drawing a scribble that is circle-shaped creates a circle. The midpoint will be automatically added; if there is already a point close to the new midpoint, that point is used.

Circle around an existing Point: Selecting an existing point and then drawing a relatively circle-shaped scribble around that point results in a circle around that point.

Gestures

Undo and Redo: Drawing a relatively horizontal line from right to left, spanning about a fourth of the window size, triggers an undo. Doing so from left to right triggers a redo.

Selecting: Clicking on an existing element toggles its selection state. It is possible to select multiple elements this way. Clicking on an empty area of the window deselects all elements.

Moving: It is possible to change the position of a selected point by dragging it. The radius of a selected circle can be changed in the same way. This is analogous to the regular *Move* mode when used on Circles defined by *Circle by Radius*.

Delete some elements: Selecting elements and then using an Undo-Gesture deletes the selected elements, and all elements that are defined by those elements.

Delete all elements: Drawing a scribble that covers most of the window, and that is not recognized as something else, deletes all elements.

While a stroke is drawn, the course of the pointer is drawn in red. This outline is not shown anymore as soon as the stroke is processed. Figure 6 shows a screen shot of ScribbleJ after a few elements have been sketched and a new circle is being drawn.

2.3.2 Advantages

The main advantage of this implementation is that it is functional and was available when this work started. After an initial learning phase, most objects are recognized

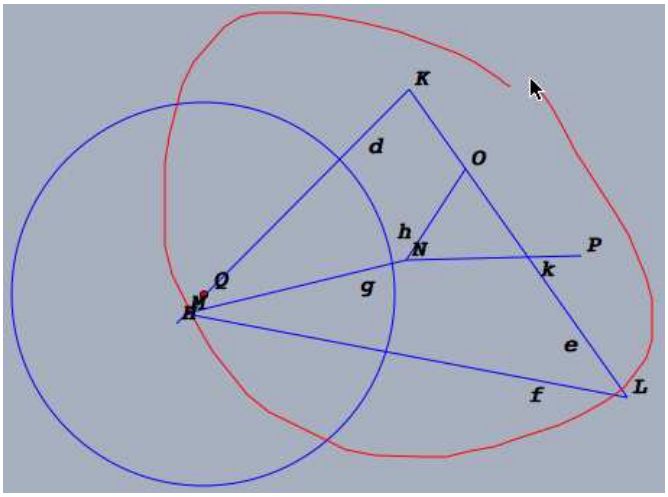


Figure 6: A screenshot of ScribbleJ

as expected. We have also deployed the ScribbleJ mode on Zaurus PDAs, and with a reasonably efficient Java VM it is usable there, too.

From the experience gained while using it, we can enumerate several aspects that work well in ScribbleJ.

The way that the center point of scribbled circles is calculated places the center close to where the user expects it. Limiting possible point sizes to a finite number of sizes has the positive effect of making the drawing look tidy. Some of the rules are intuitive, especially the way orthogonals and parallels are specified similar to the way mathematicians annotate their manual drawings.

The gesture chosen for Undo and Redo is natural and works quite well once you get used to it. The choices for selecting and moving points are both quick to execute and rarely get in the way.

2.3.3 Problematic Issues

However, other aspects do not work so well. For one thing, it is somewhat difficult to learn to use ScribbleJ.

One reason is that there are two mechanisms for specifying the relation of a new element to existing elements. Sometimes, elements must be preselected, sometimes they

must be annotated right after drawing the new element. These are not interchangeable; the user has to know which mechanism applies to the specific situation, e.g., for a midpoint the points must be preselected and for an orthogonal an annotation is necessary.

Another factor that makes ScribbleJ hard to learn is the lack of feedback; the user cannot see what is going to happen before he or she lifts the pen. Since only completed strokes are analyzed, someone learning to use ScribbleJ needs to draw many strokes to get the "feel" of how to draw certain shapes.

The way annotations must be drawn is somewhat difficult to get right: they must be drawn directly after adding the new element. If the first try fails, usually a short segment has been inserted. The creation of this segment has to be undone as well as that of the previous, correctly recognized element, to get a new try for an annotation.

Trying to recognize springs sometimes interferes with the recognition of ordinary segments.

From a software engineering standpoint, the code is hard to extend due to the fact that it is rather monolithic. It is also difficult to tweak recognition variables such as the minimum length of an undo-gesture. This makes it impossible to have different profiles for different usage scenarios or users. Because the interrelatedness of the various recognizer-methods, it is difficult to selectively disable certain strokes or gestures. Lastly, much logic is duplicated from the normal modes, and much code is in this class that really belongs elsewhere, e.g. deciding how to redefine a segment to an orthogonal.

2.3.4 Assessment of Relevance

Upon closer inspection, the existing scribbling code is not suitable as a basis for developing the sketch recognition envisioned in this work. This is mainly because the code is neither easily understandable nor extensible.

It is, however, highly instructive both as a proof of feasibility and as a prototype. It is clear that sketch recognition can work well with Geometry Software on pen-driven devices, and fast enough on a PDA. We were able to draw valuable lessons from seeing

what works well and not so well in the current code. Also, we expect that some of the algorithms developed for and used in ScribbleJ can be used in the future framework as well.

2.4 Previous Work

2.4.1 Sketch Recognition

Mahoney et al. describe concerns for sketch recognition technology[7]. The three main requirements they describe apply to this thesis as well:

1. It must cope reliably with the variability and ambiguity pervasive in sketches.
2. It must provide interactive performance or better.
3. It must offer easy or automatic extensibility to new shapes and configurations.

The focus of their work is on recognizing known configurations of elements, e.g. line drawings of humans. Unfortunately, reliable algorithms run in exponential time and are therefore not applicable to our problem domain.

Another difference is that we deal with only a few strokes at a time, and intend to give immediate feedback, whereas the algorithms used by the authors analyze finished drawings. Also, we do not want to recognize known configurations, but only known shapes. However, we need the exact coordinates where the user drew them.

Quicksketch is a project at TU Ilmenau [22]. It is aimed at pen-based computers and tries to recognize certain primitives from strokes. The system supports circles defined by midpoint and radius, circular arcs, lines defined by two points, and B-splines. Relationships between elements are deduced implicitly, e.g. a line is defined as an orthogonal to another line if it is drawn almost perpendicular to the existing line. It is possible to change elements after they were drawn by dragging on special control points attached to the elements; dependent elements are then updated. Relations that were found are maintained.

The main focus, however, is on 3D-models. Two-dimensional drawings can be extrapolated into the third dimension or rotated around an axis.

Because the constructions are supposed to eventually define three-dimensional bodies, the tool does not handle intersecting elements well. That simplifies recognition, as it is easier to exclude possible meanings of a stroke. We need to handle intersections however, as they are common in geometric constructions.

Another difference is that we have different ways to define elements, e.g. circles by three points or by point and radius.

Igaraashi et al. describe a system that can automatically infer geometric constraints from free-hand drawings [3]. The system handles only “vectors”, i.e. line segments in our terminology. It can recognize such things as parallels, segments of equal length, horizontal and vertical reflections and connections of vertices. These relationships are inferred automatically by the system. Strokes are analyzed one by one as they are drawn, just as we would like to do.

The end result of their algorithms is a drawing, not an interactive construction – it cannot be changed afterwards.

One of the benefits of interactive geometry is being able to try what happens when elements change, for instance, when two previously almost parallel lines are not parallel any more. Therefore, inferring relations is not appropriate for our purposes, we must let the user specify constraints and consider those and only those.

One interesting feature of Igaraashi’s system is that multiple possibilities are generated for a stroke and the user is allowed to choose between them by tapping on the non-primary candidates. This could be a worthwhile feature for our sketch recognition as well, letting the user select the second-best candidate.

2.4.2 Gestures

The web browser *Opera* was the first widely used program to incorporate *gestures* [21]. Its gesture-mode is activated by pressing a secondary mouse button; the movement performed with the mouse while holding the button determines the action triggered.

Figure 7 shows some of the supported gestures, as presented on the website.

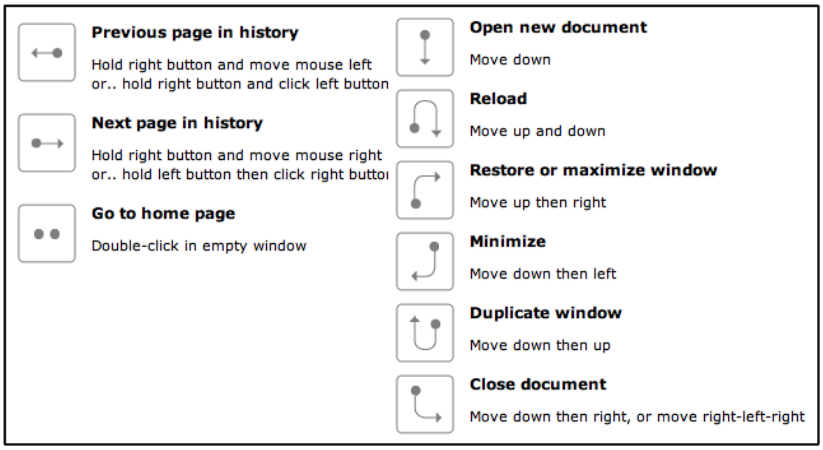


Figure 7: Sample Opera gestures

Without gestures, these actions could only be activated from the toolbar, the menu or with a keyboard shortcut. That requires either moving the mouse a long way or switching from mouse to keyboard and back. Therefore, the actions are quicker to execute with a gesture.

Many Opera users like gestures and use them all the time; others prefer the traditional options. But since they are non-intrusive, i.e., it is unlikely to use them by accident, gestures can be enabled by default. It should also be noted that gestures are a feature targeted at experienced users: it requires reading and understanding documentation, as well as some training, before it can be used.

Gestures have been widely copied; many browsers support them nowadays, at least through a third-party plugin [12]. Other applications have also started to provide this user interface [11].

They seem a worthwhile addition. The sketch recognition mechanism we implement should be able to provide gestures; we will keep the gestures shown above in mind when deciding on the kinds of shapes to recognize. As for the concrete gestures, the problem domain is too different to copy the operations.

2.4.3 Sketch3D

Sketch3D is a software aimed primarily at architects who wish to quickly visualize three-dimensional structures [17]. From the description on the website, it seems to offer sketch recognition to drawings.

In reality, its so-called “freehand mode” only allows the user to draw lines, however. Information about context, for instance when the new line is perpendicular to existing lines or parallel to an axis, is given to the user while the line is drawn. For other elements, such as circular arcs, different tools must be used. These tools are selected from a toolbar.

In more complex constructions, the relations of new elements to existing elements are not inferred correctly in all cases; in these cases, the elements that modify the new element can be preselected. For example, to draw a line parallel to a line far away in the construction, the other line must be clicked on before drawing the new line. The user then gets hints when the new line is parallel to the existing line. However, the relations established while drawing lines are not persistent; when the slope of a line changes, those that were parallel originally are not changed along.

After studying the demo version, it becomes clear that the goal of this program is too different from ours to emulate much of their user interface. Sketch3D aims to produce an exact drawing of a three-dimensional idea, while we intend to produce an exact mathematical construction.

2.4.4 Menus

It is possible to view *hierarchical context menus* as a form of sketch recognition [9]: on pressing a button, the user is presented with choices. The path then traversed by the pointer determines which submenus are opened and which choice is ultimately selected. This form of “sketch recognition” is more explicit because the possible choices are listed and displayed on the screen. However, it fits our requirements of minimizing the pointer movements necessary.

As visible in figure 8, there are some drawbacks: menus need a lot of space, require

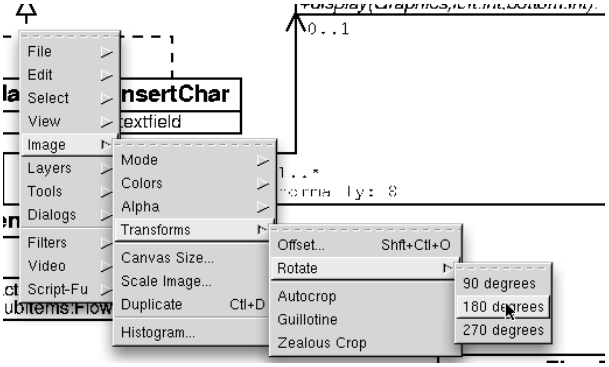


Figure 8: An example menu, from the GIMP [13]

exact pointing and can distract the audience. However, as we will explain in section 5, alternate types of menus have been proposed that alleviate these concerns and can be used advantageously for our scenarios.

2.5 Decisions

Looking at the core objects definable in a Dynamic Geometry Software and the prototype ScribbleJ implementation, it becomes clear that directly recognizing geometric objects from strokes is feasible. It also is useful on pen-driven devices, because it reduces the need to change modes often. To ease learning, we want to be able to give immediate feedback about the recognition to the user.

Gestures also look to be a promising way of enhancing the user interface on these devices. They minimize the necessary pointer movement to effect an action. However, it is not clear how to activate gesture-recognition on a device with only a single logical mouse button. Both direct recognition and gesture recognition should be handled by a single, flexible framework. Our version of such a framework, *Scribbling*, is described in the next section, section 3.

Finally, hierarchical context menus are a way to present many options to the user, more than can easily be recognized either through direct recognition or gestures. However, the traditional list-based context menus do not work well for pen driven-devices, because deeper hierarchies make long pointer movements necessary. This also presents

a problem for devices with very limited screen space, such as PDAs.

Finding a good way to present such a hierarchical menu to the user without long pointer movements and on a limited screen space can also be seen as recognizing strokes. We describe previous work and our solution to this problem, *Cinderella Flow Menus*, in section 5.

3 Scribbling

By *Scribbling* we mean a mode of operation in a Dynamic Geometry Software. In this mode, the user draws freehand sketches that are analyzed by the software. The action the user intended is inferred from the data in the scribble; it is then executed. We envision multiple uses of Scribbling in Cinderella, e.g., recognizing geometric constructions on a pen-driven device, or switching modes when using a mouse-like pointing device.

In this section, we analyze the requirements for a Scribbling framework in a Dynamic Geometry Software and give an overview over the design and implementation of such a framework for Cinderella. We then describe the different ways in which this framework is currently used.

3.1 Requirements

From the information gathered in section 2, we summarize the following design guidelines for the new sketch-recognition framework:

1. We want to give immediate feedback to the user about what is recognized while he or she is drawing.

2. Extensibility and flexibility in choosing recognized gestures is required, as we want to accommodate different usage scenarios.

3. It should be easy to tune recognition parameters, e.g., for different users or different hardware.

4. Analysis needs to be fast, since we intend to run it on a PDA as well.

The first point implies that we have to run the complete analysis every time that the platform sends us a mouse event. Therefore, an individual step of an analysis should only take $O(1)$ time. If the runtime depended on the number of events already processed, recognition would slow down the longer a stroke gets, which is not acceptable.

Extensibility means that it should be possible to easily switch certain gestures on and off.

There will be different stroke types that require similar sorts of analysis. Therefore, the framework needs to accommodate the sharing of intermediate results.

Generally, we want the new code to be as independent of Cinderella as possible. This is to reduce coupling, always good from a software-engineering standpoint, because this makes the code more maintainable and extensible. It also facilitates reuse; maybe other projects want to use our code to implement stroke recognition facilities.

3.2 Core Design of the Framework

This section explains the design of the scribbling framework. Technical complications as well as some details that are irrelevant for the concepts have been omitted. Therefore, the figures and explanations in this subsection do not accurately reflect the implementation, but explain the core design.

The basic data collected during a stroke consists of the positions the pointer was at and the time at which it was there. The class `Stroke` stores the data of one stroke, which we define as the sequence of events from mouse-button-down to mouse-button-up.

Since flexibility is desired when choosing what to recognize in a specific use of the framework, we represent one specific identification together with the appropriate action in an interface `Analyzer`. For performance reasons, we do not want every instance of a subclass of `Analyzer` to recalculate intermediate results all the time. Furthermore, there are intermediate results that can be used by more than one `Analyzer`. Therefore, we introduce an interface `Distiller`. Classes that implement this interface are notified every time the pointer moves and update intermediate results. Because intermediate results belong to the current `Stroke`, this class also is responsible for intermediate results.[6]

[6] As an alternative design idea, `Distiller`s themselves could store their intermediate results. Then, the `Analyzer`s would have to know the concrete `Distiller`s attached to a `Stroke`. Also, `Distiller` objects could not easily be reused for new `Stroke`s without losing their results for previ-

The setup and correct invocation of instances of all these classes is handled by an instance of a `ScribbleMode`. A `Mode` in Cinderella is a class that can receive mouse events and modify the state of the Cinderella kernel accordingly. Following the Model-View-Controller pattern, a `Mode` is the *controller* to the *view* presented by the Cinderella GUI and the *model* held in the Cinderella kernel.

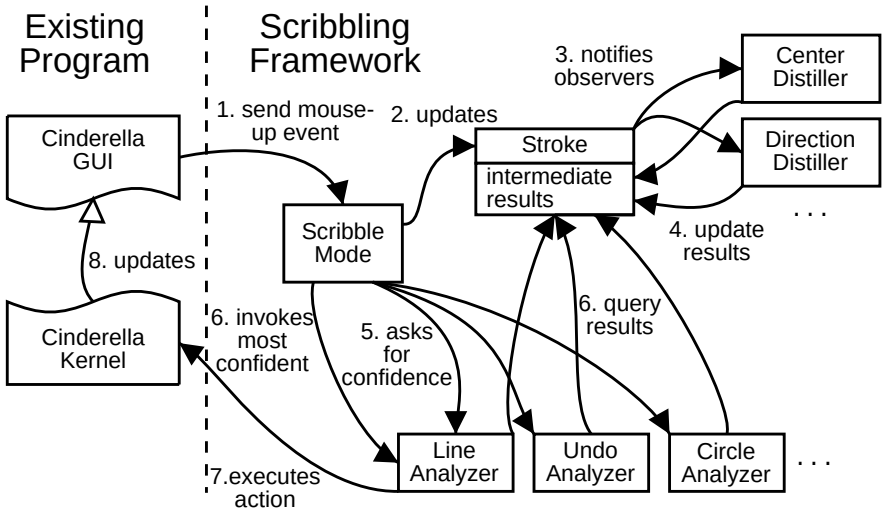


Figure 9: Overview of control flow during a stroke

Figure 9 shows the control flow when everything is set up correctly and a mouse event occurs – in this example, a mouse-up event. First, the GUI layer of Cinderella sends a mouse event to the `ScribbleMode`. The `Mode` updates its current `Stroke` instance; this automatically notifies the `Distillers` that were attached to the `Stroke` as observers, as per the *Observer* pattern [1]. Each `Distiller` updates the current set of intermediate results with the new, appropriate values. In the figure, only two `Distiller`s are shown: one that calculates the center point of the current stroke, and one that tries to identify the major direction of the stroke.

After appending the new point to its current `Stroke`, the `ScribbleMode` asks all the `Analyzer`s in use for a confidence rating. This confidence rating is calculated by every `Analyzer` by querying intermediate results and doing only very little

ous `Stroke`s. Since we support multi-stroke gestures, we chose the design described above, in which the data is stored in the `Stroke`.

processing, since results that are only calculated in `Analyzer`s cannot be shared. The `ScribbleMode` then selects the most confident `Analyzer` and asks it to either draw an icon indicating to the user what would happen if he or she ended the stroke now, or it requests that the `Analyzer` really execute the action, updating the state of the Cinderella kernel. In the figure, only three `Analyzer`s are shown. The `LineAnalyzer` is the most confident one, and since the event shown is one in which the stroke is finished, it is asked to execute its action.

On a mouse-down event, the `Mode` must set up an empty `Stroke` with all the necessary `Distiller`s attached to it.

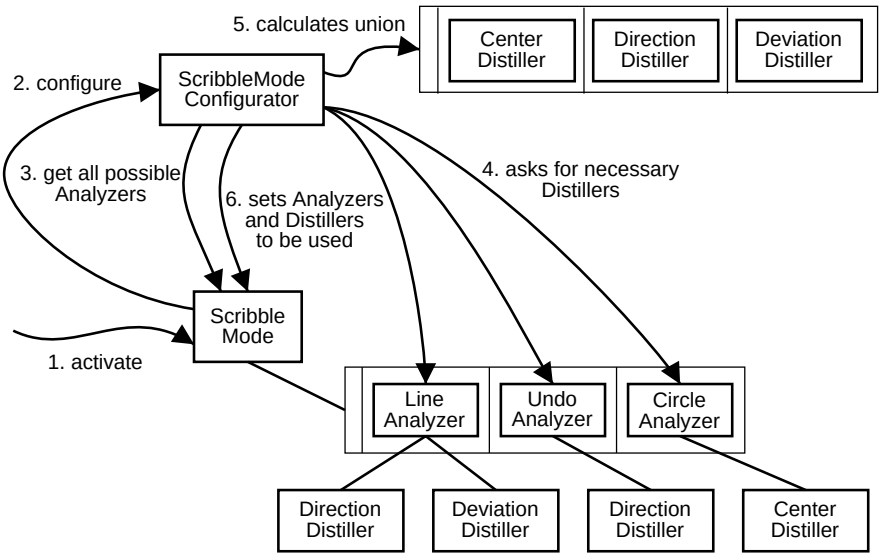


Figure 10: Setup when a mode is activated

Initial setup of the `Mode` is visualized in figure 10. It is accomplished with the help of a `ScribbleModeConfigurator`. `ScribbleMode` itself is an abstract class that is able to behave as previously discussed; the actual `Analyzer`s to be used are specified by concrete subclasses of `ScribbleMode`. When such a mode is instantiated, a `ScribbleModeConfigurator` is attached to it. On the first activation by the main program, the `Mode` asks the `Configurator` to configure the `Mode`. The simple `Configurator` shown in the figure retrieves all possible `Analyzer`s – defined in a subclass of `ScribbleMode` – from the mode. It asks all these `Analyzer`s

for the Distillers they need. Distillers have a unique key; this is used to calculate the set of all the Distillers any Analyzer needs. This procedure ensures that intermediate results are shared. The Configurator then sets the actual Analyzers and Distillers that the mode shall use.

This design allows for a high amount of flexibility: different concrete subclasses of ScribbleMode can be used for e.g. a geometry scribble mode or a physics scribble mode. An Analyzer only needs to declare the Distillers it needs and can then depend on their intermediate results being available.

Different ScribbleModeConfigurators can be used for different purposes. The basic one, shown in the figure, just activates all Analyzers the Mode knows as well as all Distillers needed by any Analyzer. This Configurator is suitable for normal operation. A different Configurator might be used during development, one that allows activation and deactivation of Analyzers at runtime. It might also allow the tuning of recognition parameters of the Analyzers. Furthermore, it might be useful to have a special Configurator for PDAs that will only activate Analyzers that do not depend on Distillers that are too costly to run on a limited platform.

Immediate feedback to the user is one of the design goals. Cinderella uses the concept of *hints*, graphical elements displayed by Hinters. A Mode can attach Hinters to the Cinderella GUI, and they will be asked to paint themselves anytime the window is repainted. Regular modes often use a Hinter to give a preview of the element that will be inserted.

Flexibility in giving feedback about what is happening to the user is desirable so we can test different approaches. Therefore, a ScribbleMode also has a list of Hinters that are currently active. The Mode is responsible for attaching the Hinters to the GUI layer of Cinderella and the current Stroke to the Hinters.

A Hinter declares which Distillers it needs – usually, it will only depend on one Distiller, the one whose intermediate result it visualizes. The Configurator also sets the Hinters to be used when it configures a Mode. In the simple case,

shown in figure 11, it just activates all the `Hinter`s whose requirements are currently fulfilled. Again, more sophisticated `Configurator`s could select `Hinter`s based on other context information. The development Configurator could also allow the user to turn `Hinter`s on and off at runtime.

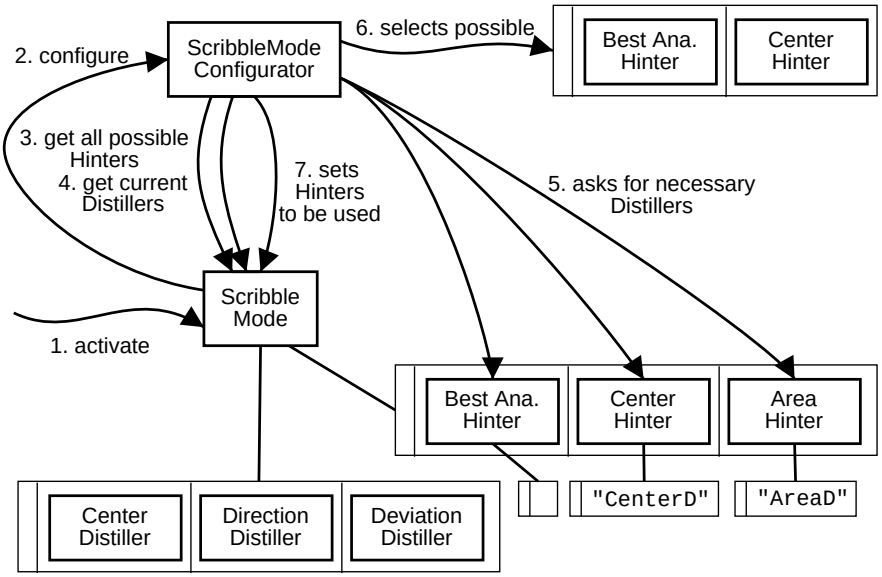


Figure 11: Setup of Hinters

Figure 12 shows the UML diagram of the core design. In addition to the classes and operations described above, a class `StrokePt` is introduced to represent a single position of the pointer on the screen. An additional interface `StrokeObserver`, used here only as a superinterface of `Distiller`, makes the use of the *Observer* pattern clear to everyone looking at the design or the code.

An `Analyzer` can have parameters, e.g., for the `Analyzer` recognizing an Undo-gesture, these might be the minimum length of the stroke or its maximum time. To accommodate the adjustment of these parameters at runtime, we introduce the methods `setParameter()` and `getParameter()`. An `Analyzer` must take care of converting the `String` argument to the type actually used. Multiple parameters are separated by semicolons. If an illegal argument is given to `setParameter()`, the `Analyzer` is expected to not change its parameters and log an error mes-

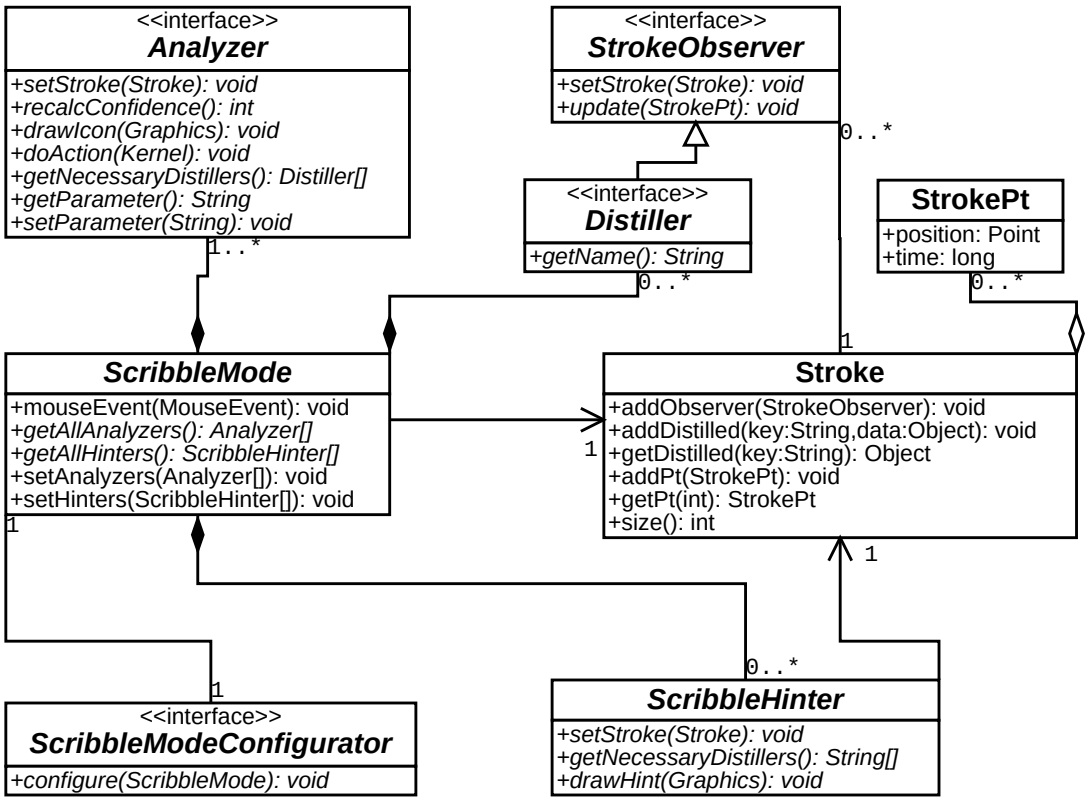


Figure 12: UML diagram of the design model

sage.

A development `ScribbleModeConfigurator` should also provide interactive access to these parameters, to facilitate tuning and testing. For deployment, different profiles for different environments could be managed by special `ScribbleModeConfigurators`; user profiles could also be managed through a properties-aware `Configurator`.

3.3 Implementation of the Framework

When implementing the design introduced above, some technical issues had to be addressed. Instead of explaining every detail, we name the areas in the implementation that differ from the design described above. The reader is referred to the source code for more detailed insights [15].

Since multi-stroke gestures are possible in principle – e.g., for annotating parallels – a class `StrokeSequence` bundles multiple `Stroke`s.

Moving elements is a special case because it is necessary to initiate an action – moving an element and recalculating dependent elements – every time that the mouse is moved, not only on button-up. There are other differences from the normal operation as well, e.g., no `Hinter`s are used. The concept of `Behavior`s is introduced to solve this issue. A `ScribbleMode` has multiple `Behavior`s; on button-down the one for the next `Stroke` is selected. The sequence of events discussed in the previous subsection now is in the responsibility of the default `Behavior`, the `ScribblingBehavior`.

The type of the intermediate results that a `Distiller` sets in the `Stroke` is dependent on the concrete `Distiller`. We adopt the convention that a `Distiller` defines an inner class, an instance of which is added to the intermediate results. A `Distiller` must also define a static method, usually called `getData()`, to retrieve an object of the correct type from a `Stroke`. If the intermediate results are complex, using a top-level class as intermediate result is also permissible. This convention avoids problems when a `Distiller` changes something about the type of its result object; it also avoids class casts in `Analyzer`s. `Analyzer`s need not concern themselves anymore about the `getDistilled()`-method of `Stroke`, because the concrete `Distiller`s provide a type-safe wrapper.

As an aid to development, the `Analyzer` interface is expanded by a `getParamDescription()`-method that returns a `String` suitable for displaying a reminder about the format of its parameter.

Since `Analyzer`s sometimes recognize multiple possible objects, e.g. lines defined by two points as well as parallels, we allow them to have an internal state. This state is reset by `setStroke()` and modified only by `recalcConf()`. The behavior of the `display()` and `doAction()` methods may depend on that state.

3.4 Interactive Fine Tuning

In this subsection, the interactive `ScribbleModeConfigurator` is discussed. It is mainly a development tool, but may also be useful to advanced users who wish to tweak the settings of the sketch recognition.

For developing concrete `ScribbleMode`s, it is highly useful to be able to activate and deactivate `Analyzer`s at runtime. Also, in order to develop `Distiller`s and their algorithms, specialized visualizations are valuable debugging aids. The concept of `ScribbleHinter`s fits this need nicely; we therefore need to be able to turn `Hinter`s on and off at runtime as well, so the display does not become too cluttered. Figure 13 shows an example of a `Hinter` that allows the developer to immediately see whether the center distiller calculates the center satisfactorily. Of course, the display is continuously updated while drawing, which increases this effect.

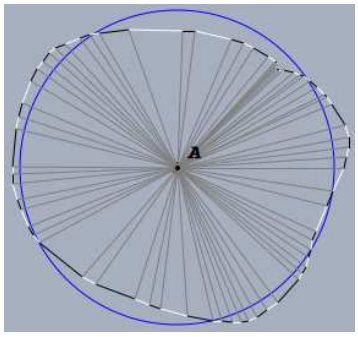


Figure 13: A Hinter visualizing the Center Distiller

Finally, tuning a `ScribbleMode` is an interactive task: many different variations must be tried to find out what works best on a given device. Therefore, setting the parameters of the active `Analyzer`s should be supported as well.

The interactive `ScribbleModeConfigurator` opens a separate window that allows tuning the `ScribbleMode` independently of the main program. Figure 14 shows a screen shot of that window. In the top part, the active `Analyzer`s can be chosen in a list that supports multiple selections[7]. The `Analyzer`s available here are

[7]The “deselect”-button is a workaround for a bug in some AWT implementations that do not allow deselecting elements in such lists.

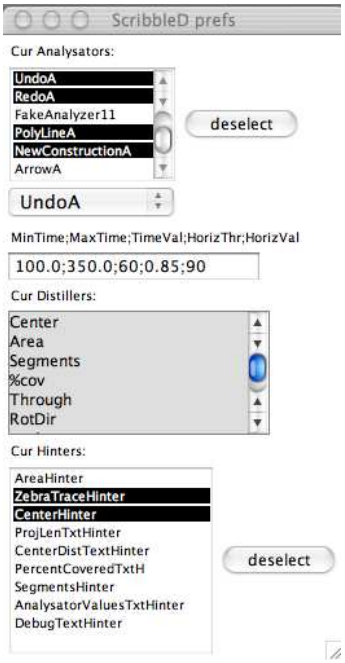


Figure 14: The interactive ScribbleModeConfigurator

those that the `Mode` returns in `getAllAnalyzers()`; initially they are activated as per `getAnalyzersDefaultOn()`.

The choice below selects the `Analyzer` whose parameter can be edited in the textfield. Shown in the figure is the parameter of the `Analyzer` for the undo gesture. Above the text field a short reminder of what the parameter means is presented; the `Analyzer` returned this reminder from its `getParamDescription()` method.

Then, the `Distiller`s that the active `Analyzer`s need are listed. This list is not modifiable nor selectable. In the bottom part of the window, the `Hinter`s that have their requirements fulfilled by the current set of `Distiller`s are listed. As with `Analyzer`s, multiple selections are supported and a “deselect”-button is provided. The list of possible `Hinter`s is returned by the `Mode` from `getAllHinters()`; their initial state is determined by `getHintersDefaultOn()`.

Whenever anything is changed in this window, the mode is instantly updated accordingly. It is possible to select the active `Hinter`s even after a stroke is complete, which is a rather useful feature while debugging.

Currently, this `ScribbleModeConfigurator` is activated when a concrete `ScribbleMode` is specified with "debug" as configuration parameter in the Cinderella configuration file. If the development of more `Configurator`s becomes necessary in the future, a more sophisticated mechanism will be used.

3.5 Calculating Common Intermediate Results

3.5.1 Overview

In this subsection, we will explain the core `Distiller`s that can be used by many `Analyzer`s. Those are the ones that calculate results that are useful for recognizing basic shapes, such as circles or lines.

3.5.2 Area Distiller

Information about the area a scribble moved over is useful for recognizing points, or large gestures such as the delete-all gesture.

This distiller maintains the minimum and maximum *x* and *y* coordinates that a stroke has traversed. It also calculates the percentage of the window's width and height that the stroke has spanned. Updating these bounds only takes four comparisons and is thus not runtime-critical.

3.5.3 Center and Distance from Center Distiller

The information about the center point of the stroke is, of course, necessary for inserting circles. It is, however, also used for other things, e.g., selecting elements.

This `Distiller` calculates the *center point of the stroke*, as well as the maximum, minimum and average distances from that center. The definition of *center point* is interesting; the naive approach is to simply calculate the arithmetic average of the *x* and *y* coordinates of all `StrokePt`s $pt_1$to pt_n:

$$\text{center} = \frac{\sum_{i=1}^{n} pt_i}{n}$$

However, this leads to unintuitive centers, because the parts of the stroke in which more events occur have a stronger influence on the result. The problem is that the events are not always delivered in regular time intervals. Therefore, we adopt the approach also used in ScribbleJ: consider the midpoints between neighboring `StrokePt`s. We scale these with the distance between the two `StrokePt`s and take the arithmetic average of these points:

$$\text{center} = \frac{\sum_{i=1}^{n-1} (\frac{pt_i + pt_{i+1}}{2}) \, |pt_i, pt_{i+1}|}{\sum_{i=1}^{n-1} |pt_i, pt_{i+1}|}$$

It is possible to update this point in $O(1)$ time by keeping the running totals of the sums and not only the result.

Unfortunately, calculating the average distance of the `StrokePt`s from the center requires iterating through all of them. This is because the position of the center changes with every new event, which also changes the distance of every previous `StrokePt`. Therefore, this `Distiller` currently needs $O(n)$ time for *n* points.

We also calculate the minimum and maximum distance of any `StrokePt` from the center. How far these are off from the average distance can give an indication of how close to a circle a stroke is. This data can easily be calculated while calculating the average distance.

3.5.4 Rotation Direction Distiller

A uniform rotation direction indicates a circle-shaped gesture.

This `Distiller` determines the direction in which the points rotate. This means, whether the points, viewed from the center point calculated above, move only left or only right. Another way to look at this is to determine whether the points are all clockwise, all counter-clockwise or neither.

To do this, the `Distiller` calculates the relative vectors of all neighboring `StrokePt`s to the center calculated by the Center `Distiller`[8]. It then calculates

[8]Therefore, this `Distiller` depends on the Center `Distiller` being available. A class using the Rotation Direction `Distiller` must ensure that the Center `Distiller` is also available, and

the cross product of those vectors for every two neighboring points. If the sign of this cross product is the same for all neighboring pairs, the stroke has a single direction and is thus possibly a circle.

In every step, we need to calculate one cross product and do one comparison. This is in $O(1)$ time per step.

3.5.5 Segmentizer

An important problem when analyzing strokes is finding segments, i.e., linear parts of the stroke. This `Distiller` is probably the most important `Distiller` included in the Scribbling framework; it is also the most complex. While finding the segments, it also calculates some information about the segments and the parts between them, which can potentially be meant as points. The data calculated in this `Distiller` can be used by a wide variety of `Analyzer`s. It is of course suited to detecting sequences of segments. Furthermore, it can be used by `Analyzer`s such as the undo-detecting one. It can check that only one segment was found and it is close to a westerly direction.

The `Segmentizer` finds stretches of adjoining `StrokePt`s that are considered linear. A stretch is linear if the distances from the start to the points projected onto the line from start point to end point are always increasing. Figure 15 shows a stretch that would be considered linear from S to E; figure 16 shows one which is not linear.

The Segmentizer also calculates certain characteristics of both linear stretches and non-linear stretches. For linear stretches, we keep two numbers `westward` and `northward` that are between -1 and 1, that are an indication of the direction of the line between the start point and the end point. They are obtained by calculating the scalar product of the normalized vector from start to end and the vectors $\binom{-1}{0}$ and $\binom{0}{-1}$, respectively.

We also keep the maximum distance that a point deviated from the direct connec-

called before this one. An exception is raised if this assertion is not fulfilled.

Since this is the only instance of a `Distiller-Distiller` dependency, using assertions suffices to ensure their fulfillment.

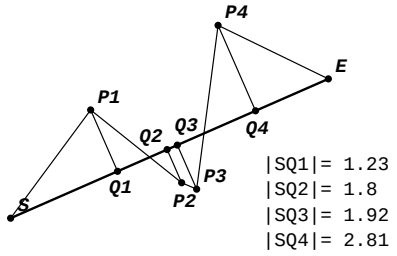


Figure 15: A linear sequence

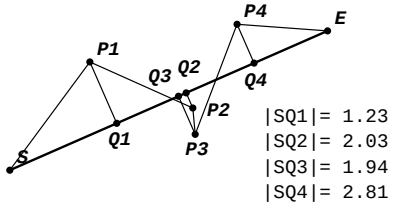


Figure 16: A non-linear sequence

tion from the start and end points. That distance is scaled to the length of the direct connection, so the allowed deviation is larger for longer strokes. This makes sense because it becomes increasingly more difficult to draw the line straight the longer it becomes.

For non-linear stretches, we keep the minimum and maximum x and y coordinates.

We also gather some data for the stroke as a whole: the maximum height and width of a non-linear stretch and the maximum scaled deviation that a point in a linear stretch had from the direct line.

Pseudocode for the Segmentizer is given in algorithm 1. The Segmentizer is in one of two states: either a linear stretch is currently running, or not. Initially, it is not. To switch to the line-running state, the distance between the points indexed by `lineStart` and `cur` has to be larger than the threshold value, `MIN_LINE_LENGTH`. If it is, `lineStart` is increased by one until the stretch from `lineStart` to `cur` is linear, or `lineStart` is equal to `cur`. If the distance between the points indexed by the new `lineStart` and `cur` is still larger than the threshold, we have found a linear stretch and switch state.

If the Segmentizer is currently in line-running state, it stays there as long as the stretch from `lineStart` to `cur` is linear. When it is not anymore, a line has been identified from `lineStart` to `cur-1`.

While finding the stretches of `StrokePt`s that are linear, we of course also find the stretches in between.

This algorithm does not always find the longest possible linear stretches. It finds

Algorithm 1 Pseudocode for the Segmentizer

```
class Segmentizer{
int cur             = -1;
int lineStart       = 0;
boolean lineRunning = false;
void update(){
  cur++;
  if (lineRunning){
    if (!linear(linestart, cur){
      lineFound(lineStart, cur-1);
      lineRunning=false;
      lineStart=cur-1;
    }
  } else {
    if (distance(lineStart, cur) < MIN_LINE_LENGTH)
      return;
    while (!linear(lineStart, cur) && lineStart < cur){
      lineStart++;
    }
    if (distance(lineStart, cur) >= MIN_LINE_LENGTH){
      lineRunning=true;
    }
  }
// ...
}
```

those above `MIN_LINE_LENGTH` that occur first, as it does not backtrack once a linear stretch has been found.

Determining whether a stretch of n `StrokePt`s is linear requires $O(n)$ time, since the projected distance must be calculated for all points between the potential end points. Therefore, this algorithms takes $O(n)$ time per update. The algorithm above is efficient, however, in that it only looks at the last linear stretch and does not reconsider older `StrokePt`s. Hopefully, that will be enough to make it usable in real-life scenarios.

The actual code is more involved than the pseudocode above, because we do not know when the stroke ends. In effect, the data about found linear stretches and the

properties of lines and points has to be kept in a way that the last update may have been the last to come. Also, updating the additional information about linear and non-linear stretches has to be done at multiple points in the code.

3.5.6 Traversed Objects

Knowing which objects have been traversed by a stroke is valuable context information for many Analyzers, e.g. for defining a circle by three points.

This `Distiller` checks for every mouse event, whether any element *is hot* at that position. An element *is hot* if it is displayed at that position. There are some ways that an element can become unable to be hot; in general, if it is hot, it can be used for defining subsequent elements. All the elements that were hot during the stroke are saved in a list[9].

This information can be used by `Analyzer`s to modify the type of element they create. E.g., a circle-like gesture that passed exactly three points could define a "Circle by 3," the circle that contains those three points, instead of the normal "Circle by Radius," which is defined by a center point and a radius.

3.6 Dropped Approaches

Some ideas seemed good in theory but were not useful enough in practice. For some of these, this subsection explains why.

Timing-Based Segmentizer

In contrast to other projects trying to recognize drawings, we have information about how and how fast the drawing was sketched by the user. It seems likely that it should be possible to detect vertices, and thus find segments, in strokes such as the one shown in figure 17 by examining the timing information – the user is slower near the vertices.

[9] All the other `Distillers` discussed in this section have been programmed to be independent of Cinderella; they could thus easily be reused in other projects. Of course, a `Distiller` that looks at existing objects in the construction must interface with Cinderella directly. Therefore, this `Distiller` can only be used with Cinderella.

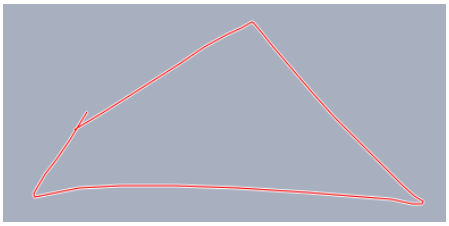


Figure 17: A sketched sequence of segments

However, it turns out that the temporal resolution of many input devices, most notably PDA touchscreens, is not high enough to yield information that is useful as the sole indicator. After implementing additional recognition logic, as explained above in subsection 3.5.5, it turned out that the other logic alone is quite sufficient for recognizing sequences of segments and thus the timing-based analysis was dropped.

Integrating Segmentizer

Another interesting idea is that vertex detection might be accomplished by integrating the point sequence.

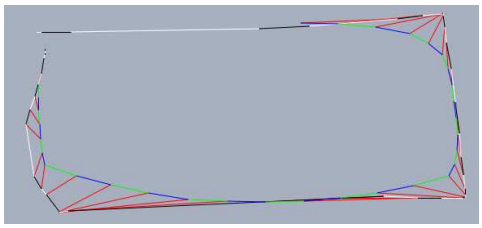


Figure 18: Averaging 11 neighboring points - Whiteboard

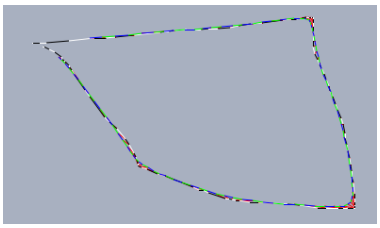


Figure 19: Averaging 11 neighboring points - Graphics tablet

The easiest way to do that is averaging the discrete points we are given in the mouse events. This is cheap to calculate, and the first results, as illustrated in figure 18, seem to confirm the idea – when the distance between a point on the white-black line and the corresponding point on the green-blue line of averages becomes large, a vertex is likely. However, this is greatly dependent on the timing – figure 19 shows a polygon when drawn on a different device. The data is meaningless in this context. It would be necessary to try averaging different amounts of neighboring points, negating the advantage that this is fast to calculate. Furthermore, giving immediate feedback would become much more difficult, because the best number of neighboring points could change during the stroke. So, this approach yields interesting pictures but is not really usable for our purposes.

Similar results and pictures can be obtained by integrating over the length of the sketched outline. That is no longer timing-dependent, but takes longer to calculate – in particular, a square-root operation is necessary for every segment of the outline. The information gained also is not exact enough to really localize vertices correctly in all cases.

The correct way really is to integrate over time – that way, the stretches of the stroke where the user was slower are given a higher value. However, this is not fast to calculate either. It also is not sufficient for vertex detection, because users sometimes pause when they do not mean an edge and a practiced user hardly pauses at the vertices when sketching a triangle, for example. Furthermore, this method requires sufficient events to be valuable – more than we can rely on on PDAs.

In summary, while integrating is the “correct approach” from a mathematical standpoint, the computationally simpler approach from subsection 3.5.5 is sufficient in practice.

Screen Size Covered

Initially, we used the screen size as a basis for the length of undo and redo gestures. This has the advantage that a stroke that has been recognized on a device once will

always be valid on that device, even when the window size is different. However, in practice, the user intuitively links the required length for these gestures to the current window size. Also, this could potentially make undo unusable in very small windows. Thus this approach proved to be more confusing and was dropped.

Line

Originally, a separate `Distiller` was used to determine whether a stroke could be considered a line. This `Distiller` calculated parameters like westwardliness and maximum distance from the direct connection. The advantage of a separate `Distiller` is that these somewhat expensive calculations are not made anymore once the stroke clearly cannot be a single line. However, once `Analyzer`s started to require directional information for linear stretches beyond the first one, this functionality was put into the `Segmentizer`.

4 Applications of the Scribbling Framework

In this section, we present the concrete applications of the sketch recognition framework. In effect, an application of the framework is a set of `Analyzer`s; some that can be used in different circumstances and some unique to the application. Currently, there are three applications of scribbling: recognizing geometric objects, recognizing physics objects and switching modes using scribbles.

4.1 *ScribbleD* – Recognizing Geometric Objects

4.1.1 Design Principles

ScribbleD is the `Mode` that was the primary focus of the implementation. Its task is the same as ScribbleJ's: recognize geometric objects from scribbles. A description of this `Mode` from the user's perspective can be found in appendix A. This subsection covers ScribbleD from a more technical standpoint. We describe the design principles and the `Hinter`s and `Analyzer`s it uses. We then evaluate its capabilities.

As a general principle, preselection is used as a modifier for new objects. E.g., to create a parallel, the line to which the new one should be parallel must be selected. While this principle must be explained to a new user, it makes the mode consistent and its behavior predictable.

The `Analyzer`s expose all the parameters that can be fine-tuned via the standardized methods defined above. If there are multiple such parameters, they are separated by semicolons in the parameter string.

4.1.2 Visualizing Progress with Hinters

Several Hinters are available for visualizing the `Stroke` currently being drawn. The `BestAnalyzerHinter` displays a small picture in the upper middle of the construction area that shows a pictorial representation of the currently most confident `Analyzer`. This is accomplished by calling its `draw()` method. This `Hinter` is always active; if no `Analyzer` has a positive confidence, it displays nothing. After

a timeout of 1.5 seconds after button-up, the picture is not displayed anymore. One example of this `Hinter` in action is shown in figure 21 on page 48.

The outline of the stroke is displayed by the `ZebraHinter`; it alternates the color of the lines between `StrokePt`s between black and white. This gives an immediate feedback about the number of points that are processed. This `Hinter` is always active.

A `RedLineHinter` is provided for those who prefer a quieter look. It is modeled after the ScribbleJ interface; the start point of the stroke is marked by a small red point and the outline is drawn in red.

Various `Hinter`s were implemented for debugging; some of them do produce nice pictures, but they are of no relevance in practice. Interested users can access them through the interactive `ScribbleModeConfigurator`, discussed in 3.4.

4.1.3 Selecting and Moving

Selecting and moving works the same way as in ScribbleJ. Clicking an element toggles its selection state. If the button-down event occurs on a selected element that is movable, no `Stroke` is recorded, but the element can be moved around. Since the Cinderella kernel decides whether an element is movable, this automatically works not only with free points, but also with all other elements that can be used in the regular "Move" `Mode`. E.g., circles that are defined by midpoint and radius, or lines that are defined by point and angle can be changed.

To recognize clicking, a maximum time and a maximum length of the stroke are configurable.

Moving is implemented as a `Behavior` of ScribbleD. This is necessary because the normal processing of events must be inhibited.

4.1.4 Creating Geometric Elements

Circle

A circle is recognized when the rotation direction is uniform and the distance from the first point to the last point is not larger than a parameter. Also, the difference between

the minimum and maximum distances from the center point must be below a threshold.

To determine the type of circle to be created, this `Analyzer` looks first at the preselected and then at the traversed points. If exactly three points on the circle were found, the new circle is defined by those three points. If only one is found, a circle through that point, and around another point, is created. If the calculated center point is close to an existing point, that point is used. A preselected point may be further away from the calculated center point to still be used as the midpoint of the new circle.

If no modifying points are found, a circle around a new point with the calculated average radius is created.

Line

A line is recognized when the `Segmentizer` reports only one linear stretch and the end points as reported by the `Segmentizer` are smaller than a maximum size. The maximum deviation of a point in the linear stretch from the direct connection of the end points may not be too large.

The type of line depends on the preselection; if a line is selected and the stroke is relatively parallel or relatively orthogonal to that line, a parallel or orthogonal is created. If the stroke went through an existing point, that point is used as the point still needed for definition. Otherwise, for parallels the center point of the stroke is used or for orthogonals, the intersection point with the existing line is used. The maximum permissible deviation of the angle of the new line to the existing line from 0 or 90 degrees is controlled by a parameter.

If two lines are preselected and the stroke moves from one of the sectors defined by those two lines to the opposite sector, an angular bisector will be inserted.

If none of this applies, a line through two points is created. Preexisting points close to the stroke are given priority; when not enough are found, new points at the beginning and end of the stroke are created and used. If the line spanned a large fraction of the window (by default, 80%), it is created as a line that extends to infinity; otherwise, it is rendered on the screen as a segment between the defining points.

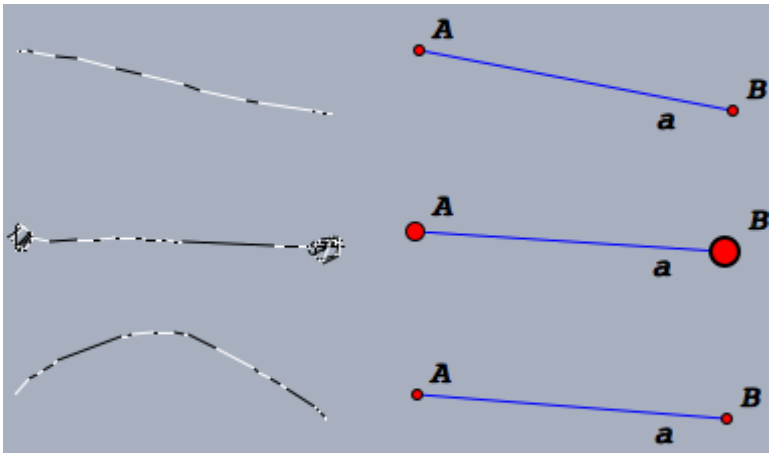


Figure 20: Recognizing lines

If new points are created, their size depends on the size of the relevant part of the scribble. The legal point size is determined as for points, see there.

Figure 20 shows three example strokes and the lines defined by them. The one on top is just a simple line; the end points are of the minimum size. The middle one has explicitly drawn end points, and the newly created points reflect the size of the scribbled end points. The bottom scribble illustrates that because of our definition of linearity, curved scribbles are accepted as lines as well. The curvature shown here is about the maximum allowed by the default whiteboard parameters, a more strongly curved line would have a too large maximum distance from the direct connection.

PolyLine

This `Analyzer` recognizes line sequences that are connected. This is often desired to draw polygons such as triangles. The work is mainly done in the `Segmentizer`; this `Analyzer` only checks whether the maximum deviation of a line from the direct connection is below a threshold and whether the maximum point size is acceptable. Both of these values are parameters. For the case of a single line, this `Analyzer` defers to the line `Analyzer`. Figure 21 shows a few examples of polylines recognized correctly.

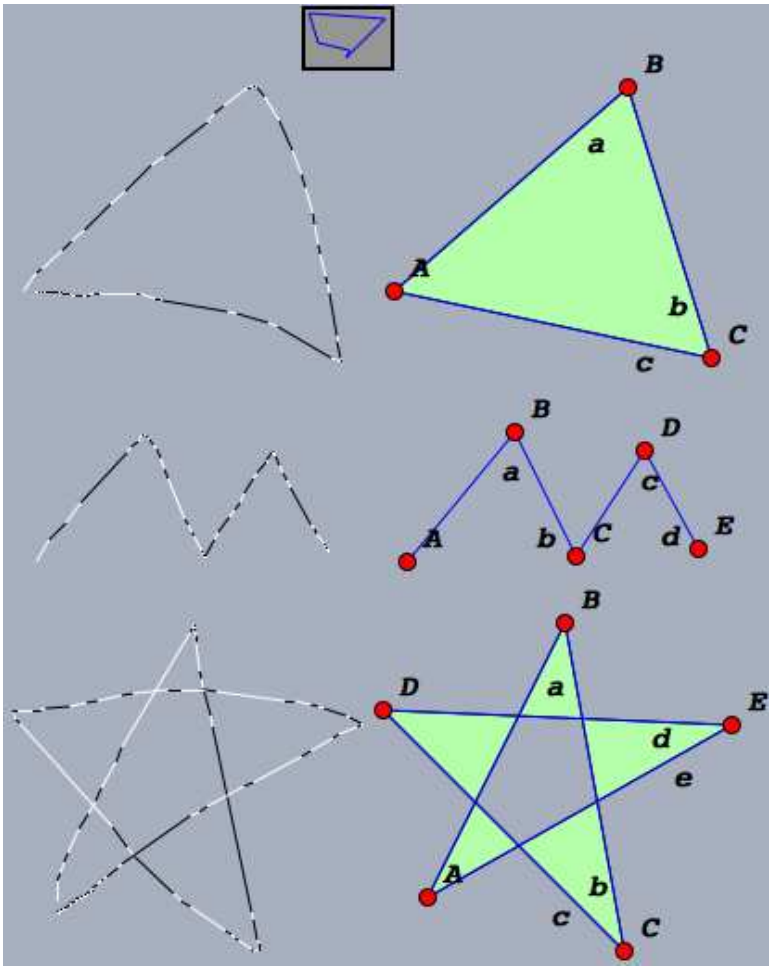


Figure 21: Recognizing polylines

The vertices are first consolidated, so that close vertices are created as a single point. Then new points are created for those vertices where no existing point can be found in the construction. Finally, the connecting lines are created, always cut off at the defining points. If the final consolidated point equals the first point, a polygon element is also created; this gives the effect of filling the polygon with color. The newly created points always are of default size; otherwise, the new line sequence looks rather untidy.

The way that linearity is defined by the Segmentizer implies that obtuse angles cannot be reliably recognized, as visualized in the top example of figure 22. However, this can be overcome by drawing the points explicitly as shown in the bottom example

of that figure.

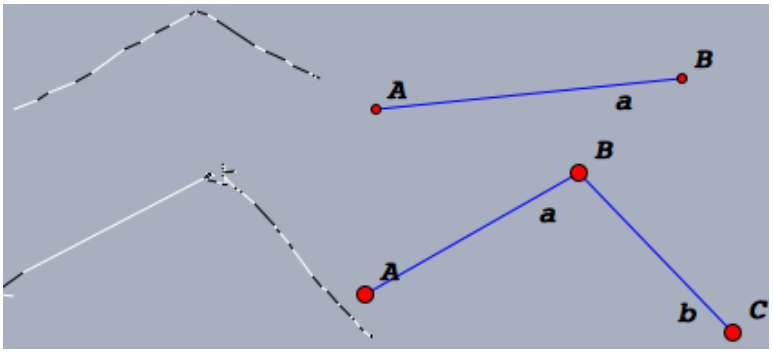


Figure 22: Polylines with obtuse angles

Point

A stroke that remains confined to a small area and is not recognized as anything else is recognized as a point. Its size is one of the legal sizes, determined by the area the stroke covered. If there already is a point at the same place, its size is changed.

If two points are preselected and the new point is close to the midpoint of those, a midpoint is created.

4.1.5 Gestures

Undo / Redo

The `Analyzer`s for undo and redo look for strokes that took longer than a minimum time but not longer than a maximum time, are relatively horizontal and linear and also span a higher percentage of the window than a threshold. All of the mentioned values are configurable as parameters.

This gesture conflicts with drawing lines. However, short lines are not affected. Drawing long lines usually takes longer than the time threshold, because the user is drawing them more exact than a quick undo-gesture. Therefore, the two Analyzers normally function together well.

New Construction

A clear-all gesture is recognized when a large percentage of the window is covered by the stroke, yet it is executed quickly. Traversing many objects increases the confidence of this Analyzer.

Inspect Properties

A gesture that has two linear stretches, the first one relatively southward and the second one relatively northward, is executed within a certain time and whose end point is close to its start point is recognized as a request for information.

Currently, the Cinderella *Inspector* is opened as a result. This component allows the inspection and manipulation of properties elements export, e.g., their color.

Right Click

The inverse gesture to the Inspect gesture, i.e. north then south, simulates a click of the right mouse button. This is useful in environments in which there is no right mouse button. What happens then depends on the Cinderella configuration; a context-menu could pop up or a menu could allow the selection of actions not possible through pure scribbling.

Rename

A double click allows the user to change the name of the element that is double-clicked. This is implemented in an `Analyzer` for non-movable elements; however, the click on a movable, selected element triggers the move `Behavior`. Thus, the move `Behavior` also had to be adapted to allow this feature, which creates a closer coupling between it and the double-click `Analyzer` than is really desirable.

4.1.6 Evaluation

The scribbling mode works as intended on both whiteboards and PDAs. The immediate feedback functions as envisioned: at all times, a pictorial representation of the action that would happen if the stroke ended now is displayed. Tuning recognition variables via the interactive `Configurator` allows searching for good defaults on the whiteboard; it is of less use on the PDA as it opens a second window. However, it turns out that the whiteboard configuration is functional on the PDA as well.

Most of the functionality of ScribbleJ is supported by ScribbleD as well. A notable exception are the annotations of lines directly after drawing them. This is not supported because redefining existing elements is not trivial, and not yet possible in the "regular" (non-scribbling) Cinderella. Therefore, the infrastructure is not present. Implementing that infrastructure requires more knowledge of Cinderella internals than the author of this thesis currently has. Arrows, too, are not yet included because modifying the arrows of an existing line has no easily usable interface. When these shortcomings in Cinderella core code are addressed, this functionality should be relatively easy to implement.

Deleting elements is not yet supported; reusing the undo gesture as ScribbleJ does would be possible but is somewhat confusing.

4.2 *ScribbleP* – Interfacing to Simulations

4.2.1 Goal and Design

ScribbleP aims to allow the user to sketch physical experiments, insofar as supported by Cinderella. This is a separate `Mode` from ScribbleD because it is rarely useful to mingle geometric and physics objects in the same construction. Using a second `Mode` also means that scribbles can be reused for other elements, which facilitates both the implementation and the actual use.

Physics in Cinderella is not calculated mathematically correct as the geometry part is. Instead, elements can be assigned *simulation behaviors*. When the user starts

the simulation, these simulation behaviors determine how elements move from one time step to the next. When the user stops the simulation, the elements return to the position they had before it started. The simulation therefore is discrete and only useful for visualizing experiments. It is less useful for actually calculating how elements will behave. In particular, it is fairly easy to create a situation in which the computational accuracy is not sufficient and elements behave differently from how the things they simulate would in the real world.

4.2.2 Similarities to ScribbleD

The ScribbleP application of the stroke recognition framework is similar to ScribbleD: new objects can be created and manipulated. Therefore, the `Behavior`s and some of the `Analyzer`s can be reused here. In particular, selecting and moving elements stays just the same. We also keep the gestures for deleting all elements, for undo and redo, for inspecting properties and for simulating a right button click.

4.2.3 Creating Physics Elements

Free Mass With Velocity

A new element is the “Free Mass With Velocity.” It can be scribbled by drawing a point above a certain minimum size and then a line. The line is interpreted as the velocity vector of the mass. When the mass is drawn on top of an already existing mass, that mass’ velocity is replaced with the new one.

Technically, we rely on the `Segmentizer` and check the relevant point sizes and number of linear stretches.

Figure 23: Scribbling a “Free Mass With Velocity”

Bouncers

"Bouncers" are walls: masses will deflect of them, but they themselves are immovable, as shown in figure 25. ScribbleP recognizes bouncers similar to the way ScribbleD recognizes polylines. Any scribble that has linear stretches that do not deviate too far from the direct connection can be recognized as a sequence of bouncers, see figure 24. The points between the bouncers are always created in a small size and black color.

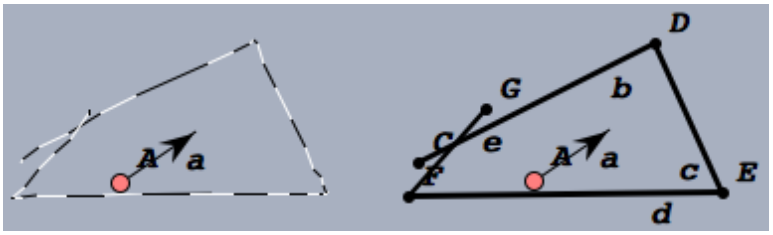


Figure 24: Scribbling Bouncers

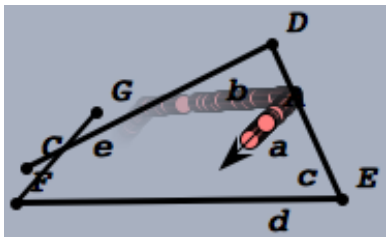


Figure 25: The effect of Bouncers

Sun

A "Sun" is similar to a "Free Mass With Velocity" in that it has a mass and attracts other masses. However, its mass is so high that it does not actually move when the simulation runs. It can be defined by scribbling a small circle, above a minimum size and below a maximum size.

The circle is recognized just as in the circle `Analyzer` for ScribbleD.

Floor

A “Floor” is a horizontal line that stops any physics elements. Moving masses will bounce off it. A Floor has an inherent friction. Recognizing a floor is fairly easy: a relatively horizontal line that spans much of the window, as determined by suitable parameters.

Springs

“Springs” are segments that try to maintain their initial size when the simulation is running. When the point at one end of the spring is moved, the other one moves accordingly. They can be stretched and compressed a bit, like springs in real life.

To scribble one, the user must sketch a linear scribble that zigzags over the direct connection of the start and end points multiple times, as shown in figure 26. The masses at the end points are created if they do not exist yet.

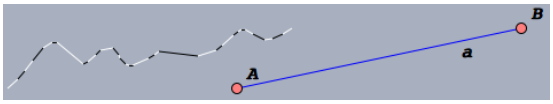


Figure 26: Scribbling a Spring

Rubber Bands

“Rubber Bands” are very similar to Springs. They are like Springs that try to reach a size of zero. In effect, the masses at their two ends are drawn towards each other.

To scribble one, an oval must be drawn that starts and ends at one of the points. We check that there are exactly two linear stretches, that the rotation direction `Distiller` says that it is monotone and that the start and end points are close to each other. This gesture is similar to drawing a rubber band around the two masses. These will be created if they do not exist yet, as shown in figure 27.

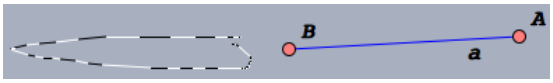


Figure 27: Scribbling a Rubber Band

4.2.4 Starting and Stopping the Simulation

In the current user interface of Cinderella, running a simulation is a separate `Mode`. It must therefore be selected from the toolbar or the Menu; this starts the simulation. Stopping the simulation again is accomplished by selecting a different `Mode`. This clearly is not suitable for ScribbleP, because the normal way to switch modes with the toolbar is not available. Instead, we adapt the Cinderella concept of *Port Buttons*, small rectangular areas that are displayed as regular elements in the construction. They can perform an action when clicked on in "Move" mode. To enable port buttons in scribbling `Mode`s, the `Analyzer` responsible for selecting elements had to be modified to first check if the clicked element is a port button, and if so, execute its action instead of toggling selection state.

A "Play Button" is created when the user scribbles a triangle with a vertical edge and the third vertex to the right of that edge – just like the pictogram on any play button on a real-world consumer device. When clicked on, that button switches the primary `Mode` between ScribbleP and Simulation; this gives just the desired effect of allowing the user to start and stop the simulation at will without resorting to toolbar or menu.

The triangle is recognized using the `Segmentizer`. We check that there are three linear stretches, that they have the correct slope and that the end point is close to the start point. Figure 28 shows how to draw a play button.

4.2.5 Evaluation

ScribbleP is not yet as mature as ScribbleD. Its capabilities are somewhat limited and some of the recognition algorithms may need further tweaks. However, it is already possible to scribble most of the physical elements incorporated into Cinderella. Creating and running an experiment such as the planet simulator shown in figure 29 is

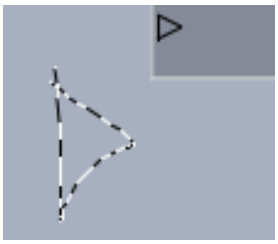


Figure 28: Scribbling a ”Play Button”

reasonably straightforward and possible without ever having to move the pointer a long distance over the screen.

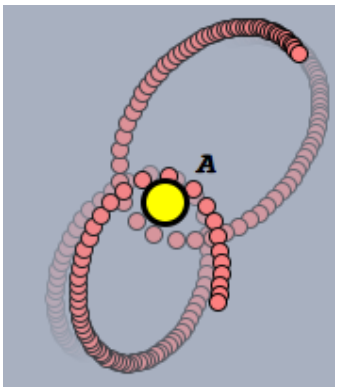


Figure 29: Simulating the orbits of two planets

More importantly, ScribbleP proves that the term “framework” is justified for our Scribbling infrastructure. Implementing ScribbleP took no longer than three working days, and much of that spent figuring out how to correctly interface to Cinderella’s physics module. Much of the existing code, in particular the `Distiller`s, was reused. The interfaces proved flexible enough to allow even peculiar functionality such as the Play-Stop-Button.

4.3 Mode Switching

This application of Scribbling lets the user switch the mode using gestures, when pressing the middle or right button. In this way, all the traditional Cinderella modes such as

creating lines or circles, can be activated without reaching for the toolbar or menu.

The functionality is implemented as a `Mode` that allows the switching of the primary `Mode` by using gestures. Cinderella was extended to support of secondary and tertiary `Mode`s, activated using the middle and right mouse buttons[10], respectively. The `ScribbleModeSwitcherMode` is designed to be configured as secondary or tertiary `Mode` and then allows rapid switching of the `Mode` used with the left mouse button.

For this mode, a new `Hinter` was implemented that darkens the window while the mouse button is pressed and leaves a transparent white trace of the pointer, so the user can see what he or she draws.

Additionally, another `Hinter` observes the primary mode and, whenever the primary `Mode` changes, displays and fades the name of the new mode at the top of the construction window, as shown in figure 30. This ensures that the user gets feedback about the new mode, because it is not certain that a toolbar that contains the new mode is displayed.

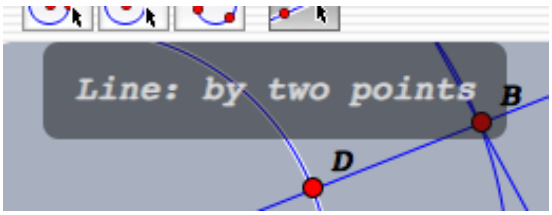


Figure 30: Visualizing mode switches

The available primary modes are grouped and an `Analyzer` is specified for the group. Whenever that `Analyzer` recognizes the gesture, the primary mode is set to the first mode of the group. Relatively horizontal lines to the left and right select the previous and next mode in the group, respectively.

[10]Modern operating systems allow their users to switch the left and right mouse buttons, primarily for the benefit of left-handed users. So to be exact, we should be speaking of secondary and tertiary mouse buttons; however it is slightly confusing that the tertiary button is physically situated between the primary and secondary buttons. Therefore, we will continue to just use the somewhat inexact terms of left, middle and right mouse buttons.

The mode switcher is configured via the central Cinderella configuration file. Therefore, different configurations are possible depending on the modes possible in the scenario.

Because the recognition routines implemented for ScribbleD can be reused, creating `Analyzer`s for new groups for the mode switcher is fairly straightforward.

5 Flow Menus

Flow Menus are an alternate way to present menu choices to the user. This section discusses first the previous work that our flow menus evolved from, then the design and the implementation of our version of flow menus, *Cinderella Flow Menus*. We then introduce the application of these menus in Cinderella.

5.1 Previous Work

Many applications use *context menus.* These menus pop up at the current position of the pointer and display a list of actions pertaining to whatever is at the pointer position. They are most often activated by the press of the right mouse button. Context menus have become almost universal, and are usually organized as a vertical list of options. The advantage of context menus over other user interface elements is that the user can select the desired action at the focus of attention, without moving the pointer far.

Pie Menus were introduced in the late 1980s [14]. They are a kind of context menu, but the menu items are arranged radially around the pointer position. Their advantage is that the distance to all of the options is equal. The pointer movement can be less exact, since every menu item has a larger area of the screen allocated to it. Furthermore, after a few interactions, the user will learn the directional gesture needed to select a certain option.

Pie Menus can be hierarchical, the selection of an option becoming the start point for a further selection. More complex zigzag gestures correspond to these nested menu items. The user will learn these more complex gestures, too. Many implementations of pie menus do not display the menu immediately, but only after a timeout.

Quikwriting was proposed as an alternative input method for PDAs, particularly Palm Pilots [9]. It is similar to pie menus in that it uses radial selection as a way to distinguish choices. The area used for character input is divided in several zones: one *central zone* and eight radial *outer zones*. A character is selected by dragging the pen from the central zone to an outer zone, possibly to another outer zone, and then back to

the central zone. The outer zone entered from the central zone determines three or five possible characters that can be the outcome of this stroke. If the pen is moved right back to the central zone, the middle one of those characters is selected. If it is moved one zone clockwise, the next character to the right of the middle one is selected, and so on. Figure 31 (from [9]) illustrates the entry of the single letter “f” and the word “the.” To input multiple characters, the pen does not have to be lifted off the input area; the return to the central area at the end of one character can be the start for the next.

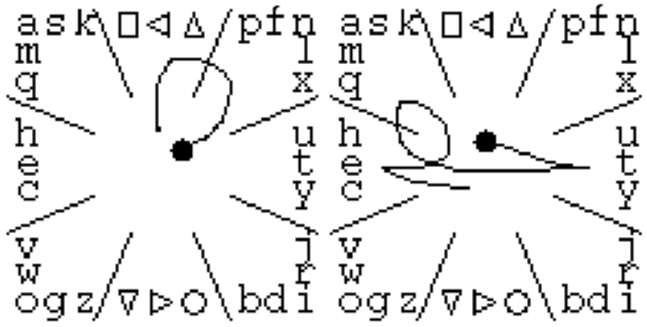


Figure 31: Quikwriting an “f”’ and the word “the”

In the top and bottom outer zones, various cursor control commands and shifts to different character sets, e.g., capital characters and punctuation, are located. Text entry using quikwriting is fast, also because users learn common words as a gesture. The letters are arranged such that common letters require less movement than rarer ones.

Flow Menus combine pie menus, quikwriting, direct manipulation and a system for continuous parameter entry [2]. *Direct manipulation* means that when the option “Move” has been selected, the selected element can be immediately moved without lifting the pen. *Continuous parameter entry* is achieved by using the outer ring of zones as a virtual knob, changing the value every time the pointer crosses a zone boundary.

They have been developed for the *Stanford Interactive Mural*, a system similar to the whiteboards we focus on, only larger. The basic idea is that the user can execute almost any action as one continuous stroke of the pen. Flow Menus are different from pie menus in that an action is executed when the pointer returns to the central area, rather than when the mouse button goes up again. Submenus are selected when

the pointer moves from the central zone to an outer zone, or the other way around. Gestures for nested commands therefore look like the ones for Quikwriting.

5.2 Adaptation for Cinderella

These types of context menus fit the goal of this thesis quite well: they provide a gesture-based, quick and intuitive way to select from multiple options. They are well suited for pen-driven devices, much more so than traditional context menus that require exact pointing and dragging.

We call the type of radial menus that we implemented *Cinderella Flow Menus*, because they share most of the characteristics with the Stanford flow menus.

Some design decisions of Cinderella flow menus differ, however. We adopt the rule that a menu item in a Cinderella Flow Menu is only selected when the pointer returns to the center zone. A submenu, however, is activated when the pointer crosses the boundary between the center zone and another zone, in any direction.

Once a Cinderella Flow Menu has been activated, we do not care about the state of the mouse buttons anymore. This improves handling on interactive whiteboards, because this behavior enables the user to navigate a flow menu without keeping the pen pressed to the board. To cancel a menu selection, the pointer can be moved farther away from the menu than twice its radius.

The menus themselves are implemented in a way that allows an arbitrary number of items in the menu. However, usually we will stick to having eight outer zones, as in the Stanford implementation.

Incorporating “direct manipulation” does not seem necessary, as Scribbling already allows moving objects. In contrast to the Stanford implementation, which displays the selected choices from all parent menus, we always show only one level of menus and highlight the current selection.

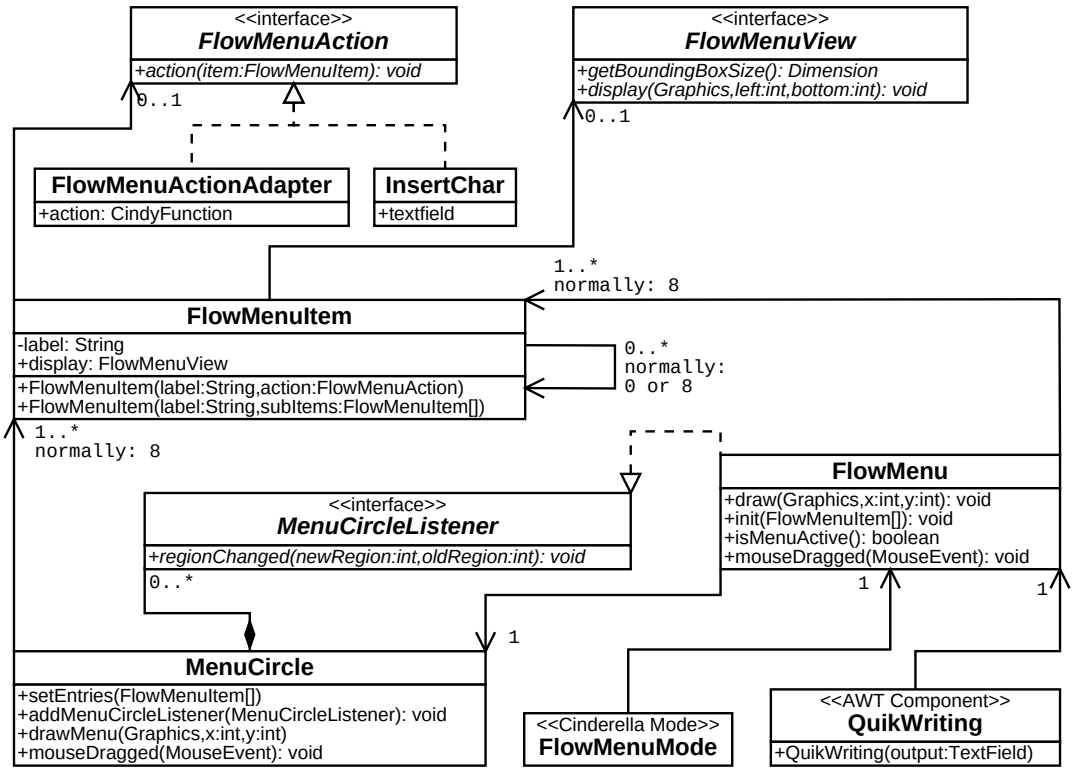


Figure 32: UML diagram of *Cinderella Flow Menus*

5.3 Design and Implementation

Figure 32 shows a slightly simplified UML diagram of the Cinderella Flow Menu implementation.

One option in a menu is represented by an instance of the class `FlowMenuItem`. A `FlowMenuItem` can be a submenu; in this case, it has several submenu items. If it represents an action, it does not have submenu items, but a `FlowMenuAction`. A `FlowMenuItem` in any case has a label. This label is sufficient to display the item to the user. If an item needs specialized rendering, it may have an object of type `FlowMenuView`.

These classes discussed up to here, except `FlowMenuView`, constitute the model of a Cinderella Flow Menu.

The class `MenuCircle` is the view that can display a set of `FlowMenuItem`s. It can handle an arbitrary number of items. The first item is drawn in the “north” zone,

the others follow in mathematical positive direction. The numbering is shown in figure 33 for the normal case of eight outer zones and in figure 34 for three outer zones.

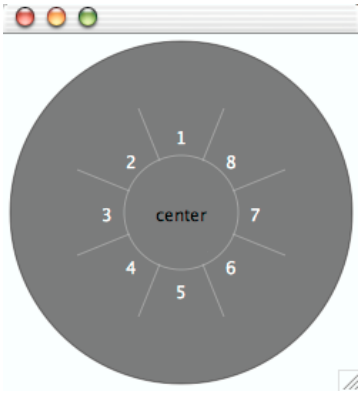


Figure 33: 8 outer zones

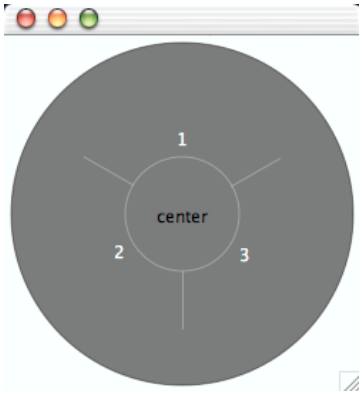


Figure 34: 3 outer zones

The currently selected item, i.e., the one the pointer is over, is highlighted. The `MenuCircle` is responsible for drawing the items at the correct positions; if an item has a custom `FlowMenuView`, that is used for the actual rendering. The `MenuCircle` communicates zone changes of the pointer via the `MenuCircleListener` interface. When determining which zone the pointer is in, it ignores the state of the mouse buttons, only the position is considered.

`FlowMenu`, which implements the listener interface, is the controller. When its `MenuCircle` reports a zone change, it checks whether a submenu was activated, and if so, updates the entries in the `MenuCircle` accordingly. If the zone change means that an action was selected, `FlowMenu` executes that action and will afterwards return false from its `isMenuActive()`-method. The `FlowMenuItem`s given to `FlowMenu` in the initializing method represent the top-level menu. A `FlowMenu` also provides a special `FlowMenuItem` that is displayed as an empty string and can be used to move up one level in the menu hierarchy.

Currently, two different classes use a `FlowMenu`. The first is the `FlowMenuMode`, which is a `Mode` in the Cinderella sense. It initializes a `FlowMenu` and installs a `Hinter` to display the `FlowMenu`. It then grabs all mouse events until the `FlowMenu` reports itself as no longer active. This can happen when an action has been selected or

when the pointer moves too far away. The `FlowMenuMode` utilizes a realization of `FlowMenuAction` called `FlowMenuActionAdapter` to link flow menu actions to Cinderella actions and mode switches.

The other class using `FlowMenu` is the one implementing quikwriting, called `QuikWriting`.

5.4 Flow Menu Applications

5.4.1 Mode Switching

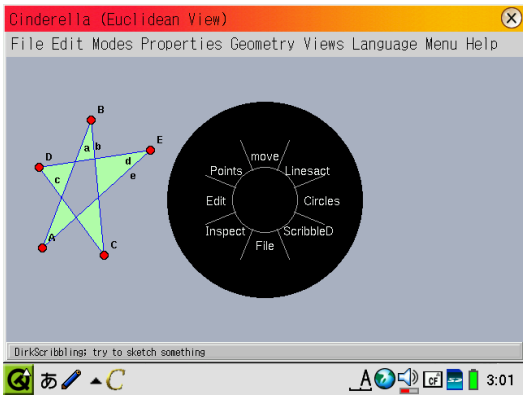


Figure 35: Switching modes

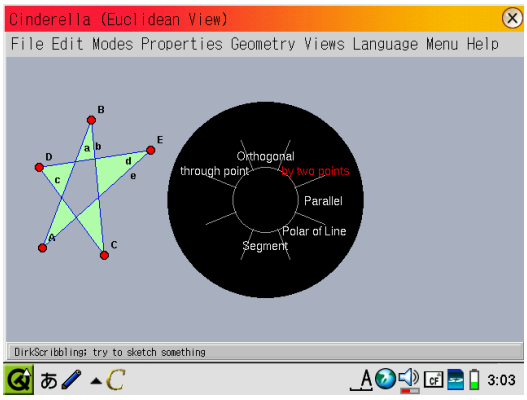


Figure 36: A modes-submenu

This application uses a Cinderella Flow Menu to allow switching the primary `Mode`. An instance of `FlowMenuMode` is configured from the central Cinderella configuration file. In the file, the possible modes are divided into submenus, with the most often used modes requiring shorter strokes. This instance of `FlowMenuMode` is configured as secondary or tertiary mode, so that it can be activated with the middle or right mouse button. Figure 35 shows the top-level menu when activated on a PDA. When the user then moves the pen from the center to, e.g., the “lines” menu item, the appropriate submenu is activated, as shown in figure 36. Moving the pen back now activates the highlighted menu entry “by two lines” and will change the primary `Mode` to creating lines defined by two points.

Flow Menu Mode Switching can additionally be activated via a menu item, if only one mouse button is available on the platform. The menu item simulates a right click in the middle of the screen, activating the `FlowMenuMode`. Since the `FlowMenuMode` then grabs all mouse events until something has been selected or the menu has been canceled, this works even on a PDA where the subsequent mouse events will be generated with the primary button down.

5.4.2 Quikwriting

We support quikwriting as a platform-independent, alternate way of entering text. To the user, Cinderella quikwriting presents itself similar to the original quikwriting shown in figure 31, but using Cinderella Flow Menus as introduced above.

We currently use the Cinderella quikwriting component in the window that allows a user to edit the label of an element. A screenshot of this is shown in figure 37. Here, quikwriting is used on a PDA to rename the midpoint of the circumference of the triangle.

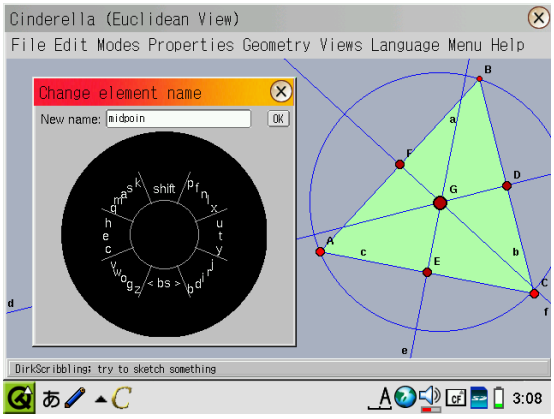


Figure 37: Using QuikWriting

The Cinderella quikwriting component uses a `FlowMenu`. It is an AWT `Component` that is suitable to be added into any AWT container. In its constructor, it is given a `TextField` that receives the text that the user enters via `QuikWriting`. `QuikWriting` uses an `InsertChar` realization of `FlowMenuAction`. This action

inserts a character at the current caret position into a text field. The component also uses some other actions that are not shown in the UML diagram, e.g., to move the cursor or simulate a backspace.

The top-level menu items are rendered with a specialized `FlowMenuView`, so that the characters appear in a circle. How the characters are distributed depends on internationalization, the English version uses the same layout as [9]. The German version currently does as well, but has the umlauts added in previously unused positions. It may be better to redistribute the letters according to their frequency in each language.

5.4.3 Context Menu / Inspecting

A flow menu is also a good replacement for the context menu that Cinderella normally uses, at least in environments in which windows are awkward. However, the necessary changes in the infrastructure to allow us to construct a context-sensitive flow menu are not yet complete.

We could use a flow menu to display and change the properties of any visible object. E.g., the color of a point could be changed or whether lines should be cut off at the end points or continue to the end of the window. The *Flow Menu Inspector* is currently in development stage, as the mechanisms in the Cinderella kernel that allow generic inspection of elements are not yet in place.

However, a proof of concept already works. In figure 38, a line already has been selected and the user executes the “Right-Button-Click” gesture. For this example, Cinderella has been configured such that the Flow Menu Inspector is activated by the right mouse button. After navigating to “Appearance” and “Line Color,” the desired color can be selected as shown in figure 39. The result can be seen in figure 40.

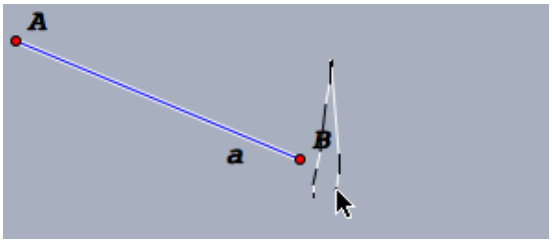


Figure 38: Activating the flow menu Inspector

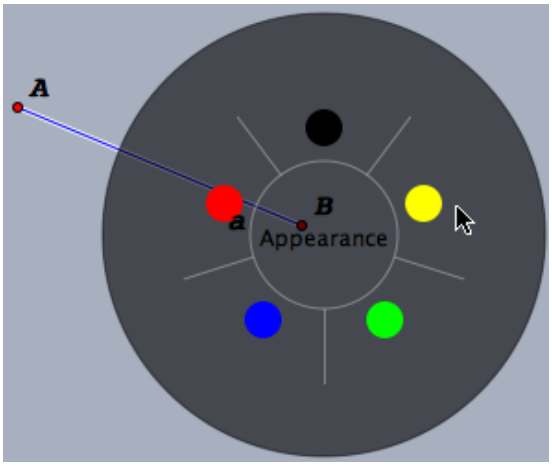


Figure 39: Selecting the desired color

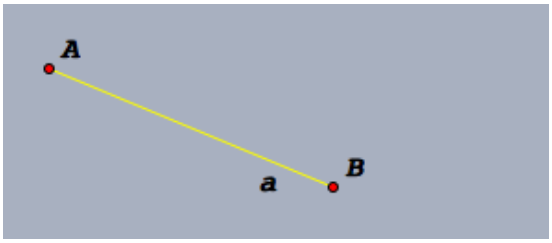


Figure 40: Result

6 Results

In this last section, we revisit the target scenarios and analyze whether the work done improves them. We then present ideas for future work before concluding this thesis.

6.1 Evaluation of the New User Interface

6.1.1 Presentations with Digital Whiteboards

Figure 41 shows a picture of the new version of Cinderella used with a Numonics Whiteboard. The differences to figure 4 on page 1 are visible even on a small picture like this: the toolbars are gone. Instead a large, uncluttered construction area is presented. The presenter and the audience can concentrate on what is actually being constructed.

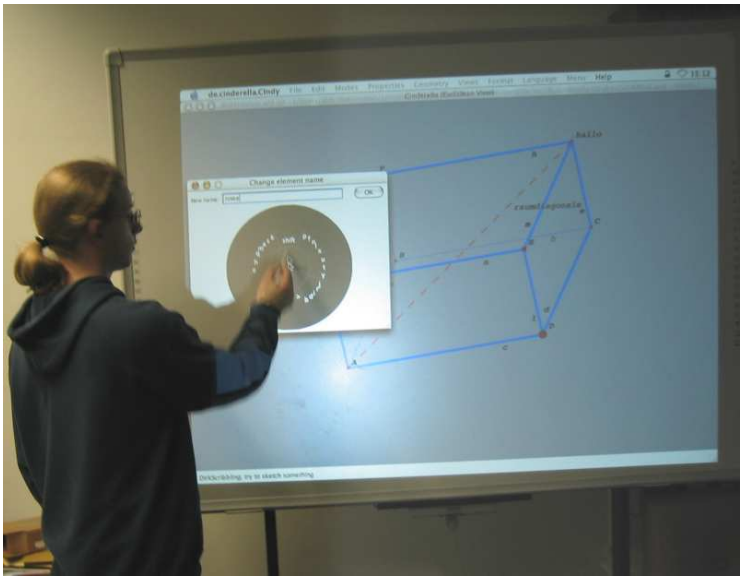


Figure 41: A Numonics Whiteboard with Cinderella and ScribbleD

The new ScribbleD mode works well in this scenario. Today's laptop computers are easily powerful enough to quickly run the calculations involved for recognizing all of the geometric objects we currently support. After a short introduction, users are able to scribble the constructions they envision. Interacting with the large, projected

construction soon feels natural. The new mode allows an interaction as we envisioned before, simply drawing what one wants to see with ScribbleD recognizing the intention and producing an exact, interactive image.

It should be noted, however, that ScribbleD is not usable without initial directions. This is not a big problem, however, because in this scenario of giving a lecture or talk, only the lecturer needs to know how to use it. Although audience members could not immediately use scribbling, they are not confused by the presenter's actions because they look natural.

Scribbling mode therefore is primarily targeted at experienced users. It should be noted that use of such advanced techniques increases the gap between presenter and audience. This is not a problem in talks, where there is a clear separation of roles anyway. However, this effect should be considered in tutorial-like situations like workshops or classrooms. Usability studies are required to decide how much of a problem this is.

Because we can enable antialiased graphics in the Java environment and use double-buffering without using too much memory, the image on the whiteboard looks nice.

For whiteboard use, we usually configure the right mouse button to display the mode-switching flow menu. The "right" mouse button in this context is a button on the side of the whiteboard pen. Using that button, the flow menu can easily be activated from anywhere in the construction area.

The mode-switching flow menu then is highly useful: it allows the quick selection of modes that produce elements not accessible with scribbling. The visual feedback of the menu means that users quickly learn the gestures required for the auxiliary modes they often need. Switching back to ScribbleD is done using the same menu.

Many common actions, e.g. loading a file or deleting an element can be accessed from this flow menu as well. This eliminates the need to use the menu bar for these actions. The menu bar is still necessary for some advanced functions like opening a spherical port or exporting the construction. It should be noted, however, that expert users that often need any functionality not available in the default flow menu can

customize it to add these actions.

Not having a toolbar means that there is less visual feedback about the currently selected mode, only the status bar at the bottom of the window indicates it. This is not such a big problem, because switching modes is much less necessary when utilizing scribbling.

The scribbling-based mode switcher is not very useful on the whiteboard. There are less possible modes and actions available than in the flow menu mode switcher. It also tends to require longer strokes than the flow menu, which is awkward on a large whiteboard.

Using the scribbling mode switcher appears a bit like magic to the audience. In contrast, with the flow menu mode switcher, it is obvious that the presenter is using a menu. However, drawing the menu may distract the audience from the point under discussion. Again, a usability study is needed to gain more insight into this matter.

The *Inspect* gesture of ScribbleD is a very nice feature. Since the Inspector window appears next to the pointer, the user does not have to move far to change elements' appearances. Because it can easily be called up again, the user can simply close the extra window after each interaction. Previously, on a whiteboard, users often moved the Inspector window next to the main window so it could easily be accessed again. This is not optimal because the main window then no longer fills the whole screen, and longer movements are necessary.

QuikWriting (shown in figure 41) is essential in many setups, and nice to have in the others. It is essential when the platform and the whiteboard drivers do not supply any text-entry widget; in these cases, only the built-in QuikWriting saves the user from having to walk to the laptop to change an element's name. Even if another widget is available, it will require interacting with another program in another window. Entering long texts with quikwriting requires some training, but is not necessary anyway in this scenario. Smaller texts suitable for element names can be entered reasonably quickly even by a user not familiar with quikwriting.

ScribbleP shows promise that physical simulations can be created using scribbling

as well. This aids presentations including such simulations.

Overall, the user experience is much improved by scribbling and flow menus. Keyboard use is not necessary any more. The presenter no longer needs to move away from the current focus of interest to select a different tool, change the appearance of an element or execute a common action.

6.1.2 Geometric Pocket Calculator

Figure 42 (as well as figure 5 on page 10) shows a Sharp SL-C700 running the new version of Cinderella. There is no toolbar, but the menu is still available.

Figure 42: A Sharp Zaurus C-700 running Cinderella

ScribbleD is usable on this PDA. All the objects implemented can be recognized. However, the latency between drawing and recognizing is noticeably higher on this platform than on an average desktop computer. Using Cinderella as a geometric pocket calculator requires some patience and concentration, but is possible. The immediate feedback is very useful in this regard, allowing one to abort gestures that cannot be recognized correctly anymore.

The display is not as visually pleasing as on a desktop. The platform[11] does not provide antialiasing, making the objects look jagged. Transparency for hinters or flow

[11] Java 2 Micro Edition, with personal profile. Published by Sun Microsystems.

menus is not available either. Also, the limited main memory on this PDA runs out very quickly when enabling double-buffering and doing a non-trivial construction. It is therefore turned off as well, which leads to a moderate amount of flickering.

Although PDAs normally provide some facility for text entry, QuikWriting remains a useful option. On the C700, for example, the text entry widget takes up about a third of the screen space when activated. Therefore, the user usually wants to deactivate it again after text entry is finished; integrating the text entry widget in the dialog disposes of it along with the dialog automatically. If the user chooses to employ QuikWriting, it can be used even quicker than on the whiteboard, because the pen has to move smaller distances. It works largely as expected.

Calling up a flow menu with the simulated right click discussed in section 4.1.5 allows accessing many actions like loading a file or selecting all elements without having to use the menu. Doing it this way is much easier to accomplish, because less exact pointing is required. However, while it is possible to switch to another mode from ScribbleD, one cannot switch back using the flow menu since there is no way to activate it. We only have one mouse button, and the gesture cannot be recognized if a different mode is active. As a workaround, a menu entry to activate the flow menu as well as one to activate ScribbleD is provided in the menu.

The Inspector is currently not functional on PDAs. Even if it was, it would probably be too large to be really useful. Cinderella flow menus do have the potential to replace the Inspector in this regard. The flow menus are usable since they offer less options at a time.

In summary, the current implementation on today's PDAs is a prototype for a geometric pocket calculator. It is usable by dedicated users willing to put up with a few annoyances, but with this hardware, it is not yet quite ready for the mass market. We expect that to change in the next few years, however.

6.1.3 Graphics Tablet

ScribbleD is pleasant to use with a graphics tablet. The high rate of events generated by a graphics tablet, coupled with a fast computer, means that the software gets many events and recognition is usually quite good. When a user utilizes a tablet anyway, scribbling is probably superior to the classic modes, because one does not have to switch modes so often.

Graphics tablets provide at least two buttons, so it is possible to use the mode switching modes for the other button(s). Many graphics allow recognizing the “back” tip of the pen as a second mouse button, usually used as an eraser in drawing programs. Some pens also have a button on the side, similar to the whiteboard pens by Numonics.

Mode Switching using flow menus is much quicker than the traditional toolbar method. The gestures for the mode a user needs most often are quickly remembered and the necessary gestures can be executed fluently then. The scribbling mode switcher is usable, but the flow menu version is probably preferable, as the scribbling version needs extra instructions before it can be used.

Whether a user prefers to select modes via the flow menu and use the traditional modes or rather scribbles most of the time on a graphics tablet is a matter of personal taste.

The quikwriting is of course usable, but if one is sitting in front of a regular computer anyway, it is probably not that beneficial.

6.1.4 Traditional Pointing Devices

Using scribbling with the mouse is possible with training, but offers no real benefit over the normal mode of operations. A mouse user is familiar with toolbars, and moving the pointer to its location is no real problem. Conversely, the exact execution of motion sequences, especially round ones, is rather hard to get right with a mouse. As expected, the current user interface is perfectly all right for mouse users.

However, gestures are useful. This is especially true because mice today almost universally offer a scroll wheel that also functions as a middle mouse button. That

button is currently unused by Cinderella, so a mode-switching `Mode` can be configured there. The scribbling mode switcher actually works well when configured this way. If the groups are linked to toolbar groups, the feedback of "previous/next mode" is immediately obvious.

The flow menu is also a possibility for the middle button. Whether it is preferable is again a matter of personal preference; all its functions can easily be executed in other ways as well.

On the trackpad of a laptop, it is even harder to do coordinated gestures beyond simple linear ones. Therefore, the mode switching modes are not really usable. Laptops do not offer a third mouse button anyway, so the right button would have to be used, losing access to the context menu.

6.2 Future Work

So far, these new sketch recognition facilities were used only people familiar with the project. The next step is a beta test with actual of the software[12].

Naturally, there are many types of objects that scribbling does not recognize yet and we have several ideas on how to integrate them. However, we will first seek the opinions of our users.

Originally, we said that we wanted to avoid using more than a single window. However, the current version still uses a second window for changing element names. This is not a big problem because the window is only visible during the time the user interacts with it, and during that time, the main window is not important. However, eliminating this window is a worthwhile goal especially for the PDA scenario and for consistency. Doing so will require relatively intrusive changes in Cinderella core code, because we will have to implement something like a window manager for the construction area.

As noted earlier, Cinderella Flow Menus have the potential to be used as an alter-

[12]The beta test is scheduled for the end of 2003. To become a beta tester for Cinderella 2.0 and get access to a version including sketch recognition, please visit `http://lists.spline.inf.fu-berlin.de/mailman/listinfo/cindybeta`.

nate Inspector. This would eliminate the other extra window we still use. It would be beneficial on platforms with limited screen space, or when a second window is cumbersome. To accomplish this, Cinderella core code will have to be generalized and cleaned up a bit.

For the lecture scenario, it might be beneficial to allow switching the mode for one action only. So, using the flow menu mode switcher, one could switch to "Define Locus" mode and define a locus, but be automatically back in scribbling mode afterwards. This might make the work flow more fluent, because less manual switching is required for the common case. To accomplish this, however, again core code will have to be modified. It is currently not possible to be notified when something has been inserted.

Revisiting the argument that scribbling widens the gap between presenter and audience, it might be worthwhile to implement a possibility to use the flow menu without displaying it. That way, it too looks like magic and does not distract the audience. Probably, a timeout before displaying it would be a good solution.

Another area worth pursuing further is to push sketch recognition deeper into Cinderella. The most obvious benefit to that would be the possibility to recognize gestures, e.g. for simulating a click of the right mouse button, in all modes. That way, gesture recognition would become universally available on platforms that support only one button.

Implementing this change would entail a restructuring of the scribbling framework as well, because we do not want to analyze a stroke twice – once for recognizing gestures that can happen in all modes, and another time in a scribbling mode if that is the currently active one. So the concept of `Stroke`s would have to be moved into the core code, with the scribbling modes using the global `Stroke`.

It is still an open question whether generic recognition parameters are sufficient or different profiles for different scenarios are beneficial. So far, the same parameters work in all scenarios. A systematic search for optimal parameters for PDAs needs to be undertaken. If profiles are needed as a result, the scribbling framework will have to be modified accordingly, providing appropriate `ScribbleModeConfigurator`s.

A GUI for preferences is also very desirable, not only for the scribbling framework. It should be possible to configure e.g. secondary and tertiary modes, toolbars and flow menus using a graphical user interface. This is a not quite trivial project because it should encompass all of the configuration possibilities Cinderella currently offers.

We already mentioned worthwhile areas for usability studies above. In general, usability studies are desirable concerning the different usage scenarios of Cinderella.

6.3 Conclusion

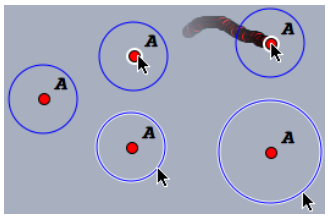
In this thesis, the differences between traditional pointing devices and pen-driven devices were analyzed. We identified *stroke recognition* as a worthwhile approach to enhancing the usability of Dynamic Geometry Software on these devices. Two complementary approaches to stroke recognition, *Scribbling* and *Cinderella Flow Menus*, were implemented.

The changes made in the practical part of this thesis are incorporated into the current development version of Cinderella and will be included in the upcoming 2.0 release. The source code developed for the scribbling framework is available from [15], because it might be adapted to other Java programs.

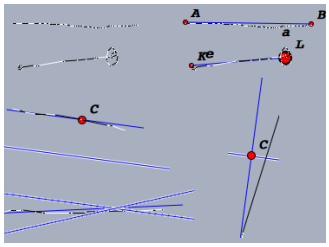
In summary, using Dynamic Geometry Software in the target scenarios has been significantly improved by the efforts of this thesis. Recognizing elements directly from sketches works reasonably well on pen-driven devices. The new scribbling mode surpasses the old one in functionality already and we expect to extend it further. The immediate feedback makes it easier to learn. Even if the scribbling mode is not capable of the next action the user wishes to do, the user never has to move the pen far away from his or her current area of attention, due to the new Cinderella Flow Menus.

A User's Guide to Scribbling

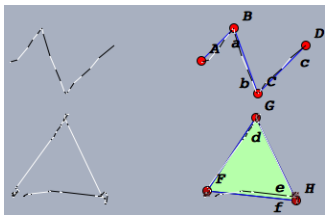
Scribbling is a new user interface for creating Cinderella constructions. It is based on the idea of directly recognizing sketches to the geometric objects you intended to draw. It is most useful on pen-driven devices such as interactive whiteboards, PDAs or graphics tablets.



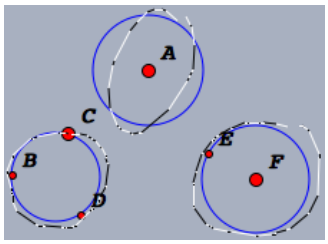
Selecting and Moving
Tap an object once to select or deselct it. Tap in an empty space to deselect all. Move a selected object by dragging it.
This applies to points as well as other free elements, such as *Circle by Radius.*



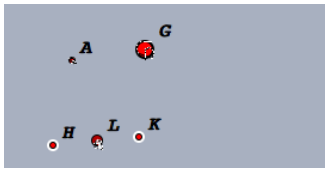
Lines
Draw a line to create a line through two points. The endpoints are created if they do not exist. Short lines are cut off at the endpoints, lines over 80% of the window are not.
Preselect another line and draw a scribble parallel or orthogonal to it to create a *Parallel* or *Orthogonal*.
Preselect two lines and draw an *Angular Bisector* to create one.



Polylines
Draw multiple lines to create a sequence of segments.
If the last point is near the first point, a *Polygon* is created.



Circles
Draw a circle to create a *Circle By Radius.* A midpoint is created if it does not exist. Preselect a midpoint for greater tolerance.
Pass exactly 3 existing points to create a *Circle By 3 Points.* Pass exactly 1 point to create a *Circle By Midpoint and Border Point.*



Points
Draw a small scribble to create a *Free Point.* Preselect two points and draw a small scribble in the middle of those to create a *Midpoint.*

Undo and Redo
Draw a horizontal, straight, quick scribble over 30% of the window width to the left to undo the last action; to the right to redo an action.

Edit Label
Double-click an element, i.e., tap it twice within a short time, to edit its label.

Delete All
Draw a quick, large gesture, crossing out all of the construction, to delete all elements.

Inspect
A straight line down, then back up to the startpoint opens the *Inspector.*

Right Click
A straight line up, then back down simulates a right mouse button click. Usually, this brings up the context menu.

List of Figures

References

Books and Papers:

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns. Addison-Wesley 1996.

[2] François Guimbretière, Terry Winograd: FlowMenu: Combining Command, Text and Data Entry. Stanford CS Technical Report CS-TR-2000-01. May 2000.

[3] Takeo Igaraashi, Satoshi Matsuoka, Sachiko Kawachiya, Hidehiko Tanaka: Interactive Beautification: A Technique for Rapid Geometric Design. In: Symposium on User Interface Software and Technology, pages 105-114, 1997.

[4] Ulrich Kortenkamp, Jürgen Richter-Gebert: Grundlagen dynamischer Geometrie. In: Zeichnung – Figur – Zugfigur, pages 123-144, Franzbecker, 2001.

[5] Ulrich Kortenkamp: Foundations of Dynamic Geometry. Dissertation at ETH Zürich, No. 13403. 1999.

[6] Ulrich Kortenkamp, Jürgen Richter-Gebert: The Interactive Geometry Software Cinderella. Springer-Verlag 1998.

[7] James V. Mahoney, Markus P. J. Fromherz: Three main concerns in sketch recognition and an approach to addressing them. In AAAI Spring Symposium on Sketch Understanding, 2002.

[8] Michael Moyle, Andy Cockburn: Analysing Mouse and Pen Flick Gestures. Proceedings of the SIGCHI-NZ Symposium On Computer-Human Interaction, pages 39–46, 2002.

[9] Ken Perlin: Quikwriting: Continuous Stylus-Based Text Entry. In Proceedings of UIST ’98, pages 213-214, 1998.

[10] Michael Trinder: The Computer’s Role in Sketch Design: A Transparent Sketching Medium. In Computers and Building, Proceedings of CAADfutures99, Atlanta, pages 227-244.

Online Resources:

[11] Generic Gestures for Mac OS X:
`http://www.bitart.com/CocoaGestures.html`

[12] Gestures-plugin for the Mozilla Browser:
`http://optimoz.mozdev.org/gestures/`

[13] Home page of the image manipulation software “The Gimp:”
`http://www.gimp.org`

[14] Pie Menus by Don Hopkins.
`http://www.piemenus.com/piemenus-hopkins.html`, viewed on November 25, 2003.

[15] Source code of *Scribbling.* For academic use only, not usable standalone.
`http://www.cinderella.de/contrib/scribbling/`

[16] The Cinderella Website:
`http://www.cinderella.de`

[17] The Website of @Last company describing their SketchUp product:
`http://www.sketch3d.com/`

[18] The Website of Hitachi concerning the StarBoard product:
`http://www.hitachisoft-interactive.com/Templates/Hitachi_UK_StarBoard.asp?mID=Content&uID=267`

[19] The Website of mcr GmbH, German distributors of Numonics Whiteboards:
`http://www.mcr-gmbh.com/`

[20] The Website of Numonics, producers of interactive Whiteboards:
`http://www.numonics.com/ipdindex.html`

[21] Webpage by Opera software describing the Opera browser’s gestures:
`http://www.opera.com/features/mouse/index.dml?platform=linux`

[22] Website for Quicksketch of TU Ilmenau:
`http://tom.prakinf.tu-ilmenau.de/en/Research/Projects/QSketch`, viewed on November 22, 2003.