An Architecture for Pen-based Interaction on Electronic Whiteboards

Takeo Igarashi
University of Tokyo
7-3-1 Hongo Bunkyo-ku
Tokyo, 113-8656, JAPAN
+81 3 5841-7413

W. Keith Edwards, Anthony LaMarca
Xerox PARC
3333 Coyote Hill Rd.
Palo Alto, CA 94304, USA
+1 650-812-4405

Elizabeth D. Mynatt
GVU Center
Georgia Inst. of Technology
Atlanta, GA 30332-0280, USA
+1-404-894-7243

takeo@mtl.t.u-tokyo.ac.jp   kedwards,lamarca@parc.xerox.com   mynatt@cc.gatech.edu

ABSTRACT

This paper describes the software architecture for our pen-based electronic whiteboard system, called Flatland. The design goal of Flatland is to support various activities on personal office whiteboards, while maintaining the outstanding ease of use and informal appearance of conventional whiteboards. The GUI framework of existing window systems is too complicated and heavy-weight to achieve this goal, and so we designed a new architecture that works as a kind of window system for pen-based applications. Our architecture is characterized by its use of freeform strokes as the basic primitive for both input and output, flexible screen space segmentation, pluggable applications that can operate on each segment, and built-in history management mechanisms. This architecture is carefully designed to achieve simple, unified coding and high extensibility, which was essential to the iterative prototyping of the Flatland interface. While the current implementation is optimized for large office whiteboards, this architecture is useful for the implementation of a range of various pen-based systems.

Keywords

pen computing, whiteboard, architecture, implementation, GUI toolkit, Flatland.

1. INTRODUCTION

Office whiteboards are one of the most common tools in a personal working environment. People use whiteboards to take notes, organize to do lists, sketch paper outlines, and as a communication medium for discussions with office mates. In general, people use office white boards for informal, unstructured activities in contrast to well-organized activities on desktop computers [16].

Based on our observations, we are currently developing a computationally augmented office whiteboard, called Flatland (Figure 1). Our research goal is to provide computational support such as storage, calculation, networking, and diagram beautification, while preserving the physical whiteboard's lightweight interaction style and informal appearance. We envision that these enhanced whiteboards will support informal activities that are difficult on current desktop computers. Our current hardware configuration is a touch sensitive large board (SmartBoard™) and a LCD projector.

Our previous paper [17] introduced the Flatland system from the user's point of view. It described the features of the system's novel user interface, and discussed how we designed the interface based on our whiteboard usage study and iterative prototyping.

Figure 1: Flatland example

In this paper, we discuss the implementation of Flatland in detail from a programmer's point of view. We introduce our new software architecture to support various stroke-based operations, and describe how our example applications are implemented on the architecture. This architecture can be seen as a variant of Kramer's representation-based architecture [11]. While the current implementation is optimized for electronic whiteboards with large physical surfaces, our architecture is applicable to various pen-based systems such as small hand held PDAs, display-integrated tablet systems, and huge wall size interfaces.

The software architecture we present in this paper is analogous to a pen version of GUI-based window system for desktop computers. Both divide the screen into several regions (windows in standard window systems and *segments* in Flatland) to provide independent workplaces. Both have mechanisms to support task specific activities within the region (applications and *behaviors*). The difference is that our target is informal, pre-production activities while existing window systems are designed for well-structured, goal-oriented activities. To be specific, we observed the following design requirements, which led us to our unique architecture.

First, the user's input must be simple, and the system's output should be informal to encourage lightweight interaction. Standard window systems support a variety of operations such as typing, clicking, dragging, etc., but these are too complicated for informal interaction. Likewise, their rectilinear widgets and printed text discourage pre-production activities. In order to provide the appropriate look and feel of real whiteboards, Flatland uses a unified notion of "strokes" for input and output. The user input is always in the form of handwritten strokes, and system feedback is given as a set of "handwriting style" strokes.

Second, informal activities on a whiteboard are very dynamic: the structure of the drawings on the board can change over time, and each drawing can serve different purposes depending on the situation. This is quite different from well-organized, goal-oriented activities on desktop computers, and the traditional notion of static windows and applications turned out to be inappropriate. This observation led to two important design decisions: *dynamic segmenting* and *pluggable behaviors*.

Users do not have to decide on the organization of their board before they start working. They can simply pick up a pen and

begin writing. The system will use heuristics to try to group strokes into segments as needed, and users can flexibly override the system's behavior by joining and splitting segments as desired. Likewise, behaviors—code that supports the semantics of a particular domain or application—can be flexibly attached to or removed from the segment on the fly. So a user can write a "to do" list on the board and then later apply a behavior to cause the strokes to begin to "act like" a to do list. Other behaviors could be applied to the same strokes over the lifetime of the segment. This flexible relationship is quite different from static, persistent relationship between windows and applications in standard window systems.

Finally, drawings on whiteboards can persist for a very long time and can be continuously changing. Additionally, each "chunk" of information on a board can be very small compared with a document in a desktop environment. Traditional file based open-edit-close style document management causes too much overhead to maintain this fine grained, ever-changing information. So instead, Flatland is equipped with automatic backup mechanisms and allows the user to recover the drawing at any time in the past. This history maintenance mechanism actually records every event occurring on the board, and thus influenced the design of entire architecture.

The rest of the paper is organized as follows. After discussing related work, we briefly introduce how Flatland works from the user's point of view. Then we describe the overall architecture of the system in detail. Finally, we briefly note some implementation issues, and discuss limitations and implications of our architecture.

2. RELATED WORK

This work is closely related to Kramer's seminal work on dynamic interpretations [10][11]. He introduced the idea of dynamic association between representation and internal data structure in the context of electronic whiteboards. He allowed the user to apply different interpretations (applications) to the same marks (freeform strokes on the screen). His goal was to capture the ambiguous nature of design activities.

We share basic ideas and research goals with him. The contribution of this paper is to extend and complement his work. While he established the framework for the representation-centered architecture, we address various implementation issues with more details and introduce a variety of example applications. To be specific, we discuss how a stroke-oriented architecture enables flexible screen real-estate control and efficient history management.

Pen based computing has become an active research area recently. In addition to research and commercial work on handwriting recognition, much work has been done on efficient text input methods [18] and gesture recognition [9]. Many systems use a pen-based sketching interface to encourage creative activities: SILK [14] uses it for GUI design, MusicPad [6] uses it for music composition, SKETCH [22] and Teddy [13] use it for 3D modeling. Pen-based techniques are commonly used on electronic board systems [8][19][20], with specialized interfaces designed for large boards. For example, a series of papers on the Tivoli system [15] proposes many interaction techniques to organize handwritten notes in meeting environment.

Although this previous work discusses the interaction techniques and specific applications for pen computers, relatively few papers discuss the software architecture to support pen based activities in general. Kramer's preceding papers and this paper are the attempts to design software architecture suitable for hosting these pen-based applications in a unified way. In a broader perspective, Flatland can be seen as one of a group of efforts (such as Pad++ [1] and Magic Lens [2]) that explore alternative software architectures *beyond* existing GUIs.

3. FLATLAND USER INTERFACES

This section briefly illustrates how the Flatland system works from the user's point of view. Some minor features are abbreviated because of space limitations. Detailed discussion on the user interface design is found in [17].

3.1 Inking and Segmenting

As the very first level approximation, Flatland works just like a physical office whiteboard. The user can draw any handwritten stroke anywhere in the screen just by dragging the stylus on the surface (called *stroking*). Erasing is done by drawing a scribbling stroke with the stylus's modifier button down (called *metastroking*).

Unlike a physical whiteboard, painted strokes are automatically grouped together into clusters, which we call *segments*. Each segment is explicitly presented to the user by a boundary surrounding its strokes. When the user draws a stroke on some open space, a new segment is created for the stroke. If a stroke is drawn within or close to an existing segment, the stroke joins to the segment. If necessary, the user can also manually split or join segments (Figure 2).

To ensure visibility, segments are not allowed to overlap. The user can drag a segment by grabbing its boundary, but if the segment collides with another segment, the collided segment is pushed away. If no more space is available, the collided segment starts to shrink to give more space (Figure3). When the user starts working on a shrunken segment, it restores its original size.
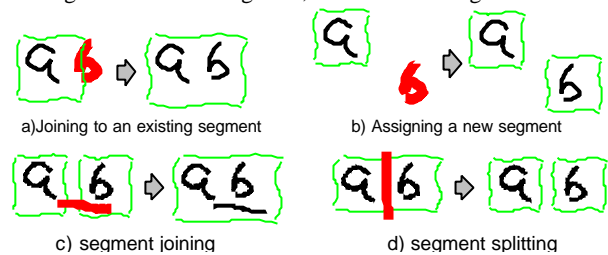

a)Joining to an existing segment   b) Assigning a new segment

c) segment joining   d) segment splitting

Figure 2: Segmenting

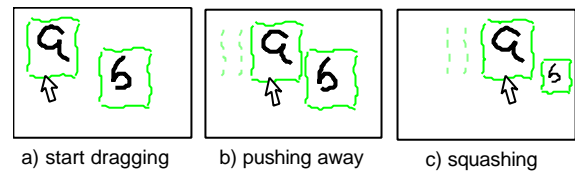a) start dragging   b) pushing away   c) squashing

Figure 3: Moving and squashing

3.2 Application Behaviors

In addition to functioning as a simple whiteboard, Flatland supports specific activities by allowing the user to attach *application behaviors* to segments. An application behavior interprets the user's freeform strokes, and gives appropriate feedback in "handwriting" style to preserve informal appearance. An active behavior is indicated as an animal figure in the corner of the segment. The following is the list of currently available application behaviors.

To do list: maintains a vertical list of handwritten items with check boxes. A new item is created when the user draws a short stroke (tap). A vertical stroke starting at a check box reorders the item, and a horizontal stroke deletes it (Figure 4).
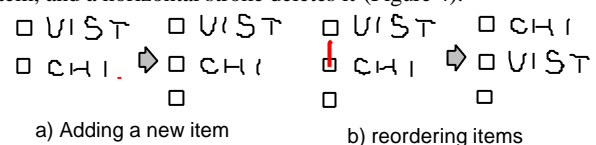

a) Adding a new item   b) reordering items

Figure 4: To Do behavior

Map drawing: turns strokes into a double line representing a

street. Intersections are handled appropriately for incoming stroke and erasing operations (Figure 5).

a) Adding a new street   b) Erasing a street

Figure 5: Map Drawing behavior

2D geometric drawing: automatically beautifies freeform strokes considering possible geometric relations. The system generates multiple candidates as pink line segments, and the user can select a desired one by tapping on it. This behavior also predicts the next drawings based on the spatial relation among the new line segment and existing line segments [12]. The predicted line segments are displayed as pink line segments as well (Figure 6).
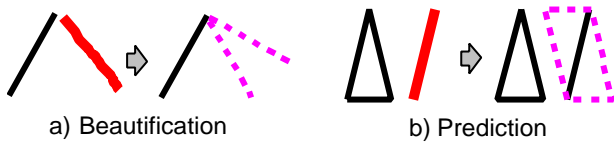

a) Beautification   b) Prediction

Figure 6: 2D Geometric Drawing behavior

3D drawing: automatically constructs a 3D model based on the 2D freeform stroke input, and displays the result in pen-and-ink rendering style [13]. The user can rotate the model by metastroking. It also supports several editing operations such as cutting and extrusion (Figure 7).

a) Creation   b) Cutting

Figure 7: 3D Drawing behavior

Calculation: recognizes handwritten formulas in a segment and returns the result of calculation. The user draws a desired formula using hand drawn numbers, and the system displays the result in handwriting style when the user draws a long horizontal line below the formula (Figure 8).
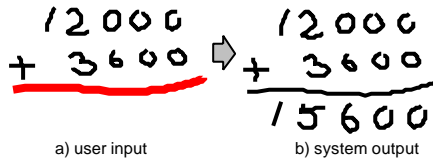

a) user input   b) system output

Figure 8: Calculation behavior

Unlike application programs of standard window systems, these application behaviors can be flexibly applied to and removed from the segment, and different behaviors can be used in combination over time. For example, in order to draw a map, the user draws streets using the map behavior, draws buildings using the 2D geometric drawing behavior, and writes comments without any application behaviors.

3.3 History Management and Context-based Search

Another feature of Flatland is its automatic history maintenance mechanism. Every event on the surface is continuously recorded, and can be retrieved later. This mechanism frees users from explicit save operations, which are not suitable for informal activities on whiteboards.

The current implementation provides three interfaces for accessing automatically stored strokes and segments. The first is infinite undo and redo. Using undo and redo, the user can access any past state of the segment. Next is the time slider. Using the

slider, the user can specify the time point directly, or use jump buttons to get to discrete "interesting" time points. Third is context-based search, which is implemented as a behavior. The search behavior allows the user to retrieve previous strokes and segments based on context information such as time, segment location, segment size, ink colors, etc. Search results are shown as a set of thumbnails on the screen, and the user can work on the stored segment by clicking on a thumbnail.
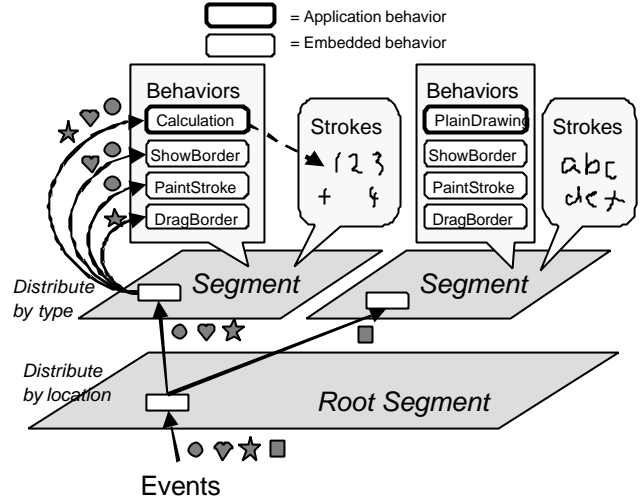

Figure 9: Overview of the Flatland architecture

4. FLATLAND ARCHITECTURE OVERVIEW

This section gives an overview of the entire Flatland architecture. Following sections describe each feature in detail.

The most basic primitive in the Flatland system is a *stroke*. The system receives user input as a stroke, and stores information as a set of strokes. All information processing in the system can be seen as manipulation of the stroke set on the screen. This simplifies the implementation, and matches the user's perceptual model of the physical whiteboard.

Strokes on the screen are grouped together based on spatial proximity, and are maintained by a *segment*. A segment allows the user to manipulate multiple strokes within the region as a group, and provides a workspace to accomplish specific tasks. Segments are different from standard windows in that they can be flexibly joined or split. Every segment is a part of the *root segment*, which handles the events that influence the entire whiteboard.

A segment delegates actual computations to *behaviors* attached to it. Behaviors respond to various events occurring on the segment, and modify the segment's stroke set or perform specific operations (such as painting). At any given time, a segment can have one "application" behavior and several "embedded" behaviors. Application behaviors provide task-specific functions and are explicitly attached to the segment by the user. Embedded behaviors provide basic services—such as inking and event storage—to the segment, and are not visible to the user. In contrast to applications of standard window systems, multiple behaviors can be attached to a segment, and a behavior can be attached or detached on the fly.

Figure 9 shows how the system maintains a set of segments, each of which holds a set of strokes and a set of behaviors[1]. When an event occurs, the root segment distributes it to a child segment,

---

[1] In [11], strokes are called *marks* or *inks*, segments are called *patches*, and behaviors are called *interpretations*. *Properties* in [11] are handled as behavior specific internal structures in our framework.

which dispatches the event to its behaviors. Then, a behavior can modify the segment's stroke set or perform other specific operations. We will see how this architecture efficiently supports each feature of the Flatland system in the following sections.

5. STROKES AS UNIVERSAL INPUT AND OUTPUT

Flatland is characterized by its use of strokes as a universal primitive for both input and output. The user's input always comes into the system in form of a freeform stroke (called an *input stroke*), and the system's feedback is presented as a collection of handwriting style strokes (called *painted strokes*). Since both output and input are in the form of strokes, the system is capable of using its own output as later input—we will see later some examples of behaviors that exploit this feature. In this section, we describe how input strokes are processed and how painted strokes are maintained in the Flatland architecture

5.1 Processing an Input Stroke

When a user draws an input stroke on a screen, the root segment first decides which segment to send it to. If the input stroke is within or close enough to an existing segment, the root segment sends the input stroke to it. If no segment is found, the system creates a new segment, and sends the input stroke to it.

The segment does not add the input stroke to its painted stroke set directly when it receives an input stroke. Instead, the segment sends the input stroke to its application behavior by calling the addInputStroke method of the behavior. It is the behavior, and not the segment itself, that adds or modifies the segment's painted strokes. This allows an application programmer to build custom application behaviors by just defining the addInputStroke method which receives input stroke from the segment, without worrying about low level events (stylus down, stylus move, etc.).

The application behavior analyzes the input stroke, and modifies the segment's painted stroke set. For example, the Calculation behavior adds multiple painted strokes showing the result of calculation when the user draws a horizontal line. The Map behavior adds two painted strokes based on an input stroke, and it also modifies the existing painted strokes to represent intersections appropriately.

When the user hasn't attached a specific application behavior, a default application behavior called *Plain Drawing* behavior is installed. This behavior simply adds each incoming input stroke as a painted stroke to the segment's stroke set, mirroring the behavior of a physical whiteboard.

An application behavior adds a new painted stroke to the segment by calling the segment's addPaintedStroke method with the painted stroke as an argument. Behaviors can also remove an existing painted stroke by calling the removePaintedStroke method. These methods actually update the segment's stroke set, and perform some low level processing to adjust the segment size and to push away surrounding segments if necessary. Figure 10 illustrates this event processing flow.

User input based on simple pen down (stroking) is always handled as a single input stroke in this way. However, user input with the pen's modifier button down (called metastroking) is handled in the conventional button down-move-up event model, and processed variously depending on its location. Metastrokes are used to start pie/marking menus, drag/split a segment, erase a painted stroke, etc. We do not have enough space to discuss each of the metastroke operations in detail, but generally, metastrokes are processed in a similar manner to strokes: an metastroking event starts from the root segment, goes to the target segment, and is distributed to the appropriate behaviors.
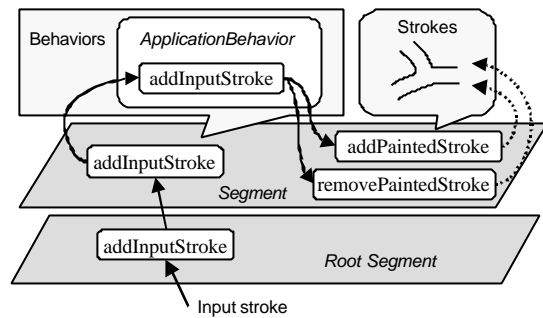

Figure 10: Input stroke processing.

5.2 Strokes as Universal Output

Flatland uses a stroke as the basic primitive for displaying information. In addition to directly showing the user's hand drawn strokes, system feedback is also presented in the form of freeform strokes. This decision primarily comes from aesthetic reasons to give informal appearance, but also helps simplify the entire architecture.

When an application behavior wants to give feedback to the user, it has to create an appropriate new stroke and add it to the segment's stroke set. Application behaviors are not allowed to directly paint on the screen using primitive operations such as drawText, drawLine, drawImage, etc. For example, when the 2D geometric drawing behavior shows the result of beautification or prediction, it adds corresponding line segments in form of strokes to the segment's stroke set, instead of directly drawing lines on the screen. And the calculation behavior displays the result of calculation by adding a set of strokes representing numbers instead of drawing printed text directly on the screen.

This design has two benefits from the implementation's point of view. First, application programmers do not have to worry about low level painting operations, and they gain the appropriate informal appearance for free. Second, and more importantly, the system can recover the appearance of the board just by recording the segment's stroke set at each time point. If each behavior paints arbitrary things directly on the screen, the recovery of the screen snapshot would have to involve the behavior and could be very complicated. We will discuss this in detail in the History Management section.

6. DYNAMIC SEGMENTATION

The structure of drawings on a physical whiteboard is very volatile and flexible. Our dynamic segmenting mechanism is designed to capture this property. Dynamic segmenting frees the user from defining the structure of the board beforehand, and allows him or her to organize the board on the fly. Flatland segments are different from windows in a number of ways.

First, a segment is created automatically in response to the user's input stroke, while a window has to be explicitly constructed before stating interaction. Second, segments are not allowed to overlap. This results in "pushing away and squashing" effects, which allows more information to be presented while preserving visibility. Finally, segments can be dynamically merged or split.

6.1 Distribution of an Input Stroke

When the user draws a freeform stroke, the root segment calculates the distance between the stroke and existing segments. If the stroke overlaps or is close enough to a segment, the stroke will be send to the segment. If the stroke overlaps multiple segments, the system merges the corresponding segments, and sends the stroke to the resulting segment. If no such segment is found, the root segment generates a new segment, and sends the stroke to it.

6.2 Moving a Segment

The user can move a segment by making a metastroke starting at the segment's border. An embedded behavior called *Drag Border* responds to the event, and moves the segment according to the pen movement. It generates a surfaceMoved event to the application behaviors to update their internal structures. This surfaceMoved event also occurs when the segment is pushed away by another segment.

6.3 Pushing and Squashing a Segment.

When the Drag Border behavior tries to move a segment, the segment asks the root segment for space. If any segment occupies the space, the root segment pushes it away to make space. The pushed segment then requests space for itself, and this continues until a segment is pushed against the screen boundary.

When this happens, the segment at the boundary starts to shrink to give space. When a segment shrinks, the actual coordinates of its strokes remain unchanged. Instead, the segment maintains a "scale" field, and the Paint Stroke embedded behavior renders the scaled strokes on the fly. In other words, the shrinking effect occurs only superficially. This frees application programmer from taking care of the scaling effect.

6.4 Merging and Splitting Segments

In order to merge segments, the root segment constructs a new segment, and calls its addPaintedStroke method using all strokes in the original segments as argument. After that, the system deletes the original segments. In order to prevent confusion, the current implementation does not allow the user to merge segments with application behaviors.

A segment is split when the user draws a splitting stroke (i.e. long vertical or horizontal line that cross the segment). This event is handled by the root segment instead of the segment that is being split. The root segment constructs a new segment, and transfer strokes one by one by calling the deletePaintedStroke method of the original segment and the addPaintedStroke method of the new segment. Again, the current implementation does not allow the user to split a segment with an application behavior.

7. PLUGGABLE BEHAVIORS

Behaviors provide a way to associate domain-specific computational processing with a particular segment. While behaviors are superficially similar to traditional applications running within windows, there are some fundamental differences.

First, a segment can have multiple composed behaviors active at a time, while a window cannot belong to multiple applications. Second, a behavior can be attached to and removed from a segment on the fly, even after the segment has been created (so users can "create first, process later"). Third, visual representation is maintained as a set of strokes by the segment, and behaviors do not directly render onto the screen (except for the Paint Stroke and Show Border behaviors).

7.1 Event Processing

A segment distributes a variety of events to its behaviors for them to perform appropriate action. This process is implemented based on the event listener model of the Java language. When a segment detects an event, it distributes the event to the behaviors equipped with the corresponding event listener such as SurfaceListener, StrokeListener, MetastrokeListener, etc.

SurfaceListeners handle events related to the segment configuration. They react to changes in segment location, size, activation, and inactivation. They also react to requests for painting of the segment. Most embedded behaviors are instances of this event listener. For example, the Paint Stroke and Show Border embedded behaviors respond to requests for surface painting. Some application behaviors respond to this event to modify their internal structure.

StrokeListeners handle the incoming strokes drawn by the user. This event listener is used by application behavior to detect input strokes. MetastrokeListener handles the events related to metastrokes. The Drag Border behavior responds to this event, and some application behaviors use this event to handle specific gestures such as erasing.

7.2 Embedded Behaviors

Embedded behaviors are implicitly attached to the segment, and work as a part of underlying system service. It would have been possible to implement these services as a part of a segment, but we chose to implement them as separate entities to make the entire system highly extensible. For example, it is possible to give the system a completely different look and feel just by changing the embedded behaviors without modifying the segment itself. It is also easy to add new features as new embedded behaviors. Actually, our "moving a segment" feature came later in the development process; the Drag Border behavior was added without requiring much rewriting of the segment code.

7.3 Application Behaviors

This section describes the implementation of some application behaviors in detail. The Flatland infrastructure provides an API which programmers can use to build their own application behaviors, without worrying about low level implementation details of the entire system.

Basically, an application behavior receives an input stroke from the host segment, and modifies the set of painted strokes maintained by the segment. An application behavior can also respond to any other events to maintain some task specific semantics. For example, most application behaviors respond to metastroke events to delete the closest painted stroke.

Figure 11: Behavior specific internal structures.

An application behavior does not maintain the stroke set—this is done by the segment—but it may require additional internal information about the strokes of the host segment. For example, the To Do behavior has a list of to do items, and each item has a pointer to its corresponding check box and strokes. The Map Drawing behavior has a network of streets and crosses, and each street has a pointer to two strokes (Figure 11). This internal information disappears when the behavior is detached from the segment, and is reconstructed when the behavior is re-attached, as described below in the section, Reapplication of Application Behaviors

Plain Drawing Behavior

This behavior is the default application behavior and works as a prototype for other application behaviors. The code for this behavior is quite simple. It adds a new input stroke to the segment's stroke set directly, and removes a painted stroke when it detects erasing gesture. This behavior does not cause any side effects on other painted strokes, and it maintains no behavior specific internal structure.

Map Drawing Behavior

This behavior maintains a graph representation of streets and intersections internally. Each street has pointers to the two painted strokes representing the street, and each intersection has pointers to the streets connected to it.

When an input stroke comes in, this behavior first examines whether the stroke overlaps some existing streets. If no overlap is found, the behavior creates two painted strokes at the both sides

of the input stroke, and adds them to the segment's stroke set. In addition, the behavior adds the new street to its street set. If the stroke overlaps some existing street, the behavior divides the street and the input stroke at the section, and reconstructs the appropriate graph topology. The behavior deletes the painted strokes associated with the modified street, and adds a set of new painted strokes. When the user tries to erase a painted stroke, the behavior erases the corresponding street. Then it reconfigures the internal graph representation and edits the segment's stroke set.

Calculation Behavior

This behavior works just as a plain drawing behavior until the user draws a long horizontal stroke requesting calculation. When this happens, the behavior searches for the set of strokes above the horizontal stroke, and hands them to a handwriting recognizer. The recognizer returns the symbolic representation of a formula. The behavior calculates it, and adds a set of painted strokes that represent result digits to the segment's stroke set. This behavior maintains no internal structure, and scans the entire segment each time. As a result, this behavior can accepts any painted stroke as input, including the painted strokes created by the behavior itself or those painted before the behavior is applied to the segment.

3D Drawing Behavior

This behavior has a 3D polygonal model internally, and renders the model by adding painted strokes representing visible silhouette edges to the segment's stroke set. When the user rotates the model, the behavior removes all previous strokes, and adds new strokes. Unlike other application behaviors, it directly responds to the low level metastroke events to implement rotation.

Search Behavior

This behavior is a part of the system infrastructure, and is very different from other application behaviors. While other application behaviors provide feedback by editing the segment's stroke set and letting the PaintStroke embedded behavior paint them, the search behavior paints buttons and search results to the screen by itself. This prevents the search result to be recorded as new strokes, and gives a distinctive look to the segment.

7.4 Reapplication of Application Behaviors

As we have mentioned already, behavior specific internal structure disappears when the behavior is removed from the segment, and the structure is recovered when the behavior is applied to the segment again. This section discusses the implementation of this reapplication process in detail. A naïve implementation may be to save the behavior specific structure *in the segment*, but this strategy fails because of our dynamic segmenting feature. Segments can be merged or split, which means that a segment is too fragile an entity to store these structures safely.

As an alternative strategy, we store the behavior specific structure *in the painted strokes*. Each stroke remembers the associated partial internal structure, and a re-applied behavior uses these partial structures to recover the entire structure. This allows segments to be split and joined appropriately.

For example, the To Do behavior gives each painted stroke a pointer to a corresponding to do item object. When the To Do behavior is reapplied to a segment, it examines all the painted strokes, and groups them based on the associated to do item objects. Each to do item can originate from different To Do behaviors. Then, the To Do behavior constructs a list of to do items, and organizes the strokes appropriately (Figure 12).
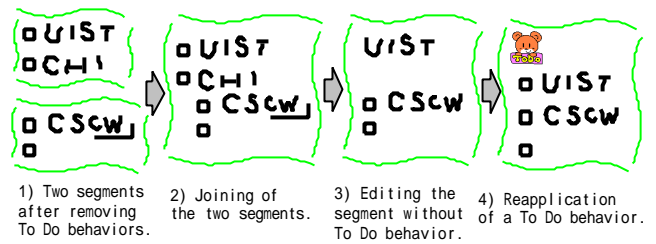

1) Two segments after removing To Do behaviors. 2) Joining of the two segments. 3) Editing the segment without To Do behavior. 4) Reapplication of a To Do behavior.

Figure 12: Re-application of a To Do behavior.

1) Two segments after removing Map behaviors. 2) Joining of the two segments. 3) Editing the segment without Map behavior. 4) Reapplication of a Map behavior.
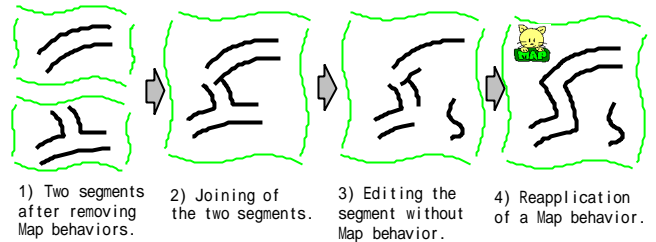
Figure 13: Re-application of a Map behavior.

The Map Drawing behavior embeds a pointer to the corresponding street object in a painted stroke. When a map drawing behavior is reapplied to the segment, it extracts the set of street objects embedded in the strokes, and constructs a complete street-intersection graph. Again, each street object can be generated by different Map Drawing behaviors (Figure 13). Strokes generated by other behaviors remain unchanged.

The 3D drawing behavior embeds a pointer to the 3D model in each stroke. When the behavior is reapplied to a segment, it extracts the 3D geometry from the stroke. If more than one 3D model is found in the stroke set, the 3D drawing behavior ignores the rest of them.

An application programmer has to write code to store and recover the internal structures when he or she uses internal structures. Currently, this part of coding is too complicated and difficult. It is our future work to find a more unified way to handle behavior reapplication.

8 HISTORY MANAGEMENT

One of the goals of Flatland was to create a "change safe" whiteboard. What this means is that users should be able to safely change any content on the board, knowing that they can recover it later if need be. To satisfy this requirement, Flatland must be able to reconstruct the contents of any given segment as requested by the user.

To implement this ability, Flatland uses the combination of two different mechanisms. One is a command object model, which maintains short-term history and supports infinite undo/redo and the *time slider*. Time slider allows the user to view segment status in the past [21]. The other is a persistent document management system based on associative memory, which maintains long-term history and supports context based search.

8.1 Undo/Redo Model

Our infinite undo/redo is based on the "command object" idiom [7]. Command objects are objects—in the object-oriented sense of the word—that encapsulate an operation that can be performed in an application. Each type of action that the user can take on the whiteboard is represented as a discrete class of command object. Instances of commands are "invoked" by calling a well-known method on them that causes them to perform their operation, updating the state of the board.

In our model, commands can be invoked and they can be reversed. That is, each command supports the ability to both "do" and "undo" its operation. Once this ability is added to the base command object pattern, command objects can be connected together in graphs to form complex histories that represent all of the possible states in which the application has existed. By

traversing the graph, sequential sets of operations can be done or undone. This model of graphs of command objects as a means to represent time has been used by Timewarp [5] and other systems. (Although, unlike the generalized time model supported by Timewarp, Flatland doesn't allow divergent or revergent histories—in Flatland, the history graph is strictly linear, and thus avoids issues with conflicts [4].).

The time slider is implemented based on this infinite undo/redo model. When the user moves the slider forward, the system invokes redo methods of the command objects sequentially, and vice versa. Semantic jumping is implemented by putting markers in the command object sequence. If the user presses the jump button, the system searches for the next marker and jumps to the time point.

8.2 Supporting Undo/Redo with Extensible Behaviors

Flatland faced some unique problems in representing its state as a linear graph of command objects. In "traditional" uses of the command object idiom, each command is atomic—that is, it can reliably and completely do or undo its operation, and has no side effects that aren't represented by the state in the command object itself. As an example, when a command object in a drawing program is rolled forward, it must take care to store all information needed to completely reset the state of the application if it is rolled back. If performing the operation causes some change to be made to the graphics context of the application, the creator of the command must be aware of this side effect, and must account for it when performing the corresponding undo.

This situation is in contrast to the basic architecture of Flatland, where the use of extensible, pluggable behaviors means that essentially *every* interesting update to the state of the application *does* occur as a side effect to user input. The set of operations that can occur when a user draws a stroke on the board is dependent on the set of behaviors installed, and the current state of each of those behaviors. This leads to some problems in applying the command object idiom in the face of extensible behaviors.

One naïve approach would be to represent only the original user input in the command history. So if a user made a stroke, and the map behavior then drew two parallel strokes to represent a street, only the original stroke (which doesn't even appear on the screen after the map behavior is finished with it) would be present in the history. The problem here is that the history no longer represents the complete state of the application. Jumping to a different node in the history graph involves "replaying" the user input to the behaviors, causing them to perform all of the same operations they would in response to "fresh" user input. The computations done by behaviors can be arbitrarily complex, which means that jumping to distant states can be arbitrarily expensive.

Flatland uses an alternative approach, where any behavior expresses its updates in terms of new command objects. So in the example of the map behavior, the history would contain a behavior-specific command object indicating that a new street is present, followed by two painted strokes (added by the map behavior). This approach has a big advantage: changes based on user input are "pre-computed" by the behaviors, and only their final outputs are represented in the history. The "side effects" of the input are turned into "foreground effects" and represented as first-class citizens in the history.

8.3 A Transaction Model for State Changes

This second approach does have a drawback: since behaviors write their operations into the command history, simple atomic roll-forward/roll-back is now inappropriate.

For example, with the Map behavior, suppose that a user has drawn a stroke that corresponds to a new street, and then needs to roll time back. The original stroke command is not represented in

this history—it has been replaced by a set of commands representing the effects of the stroke. Clearly, rolling back atomically is probably not what the user wants to see: such a roll back would reveal the individual operations of the map behavior, rather than the semantic "chunk" of the whole set of operations.

To solve this problem, we adopted a transaction model for the commands in our histories. Each original user-level input begins a new transaction. As the Flatland event dispatch code runs and behaviors perform their operations, their effects are grouped into this new transaction. Figure 14 shows an example of a transaction. From this model, causality relationships are clearly indicated, as all operations in a transaction are effects of the same cause. Transactions are represented explicitly in the history as commands, and the history roll-forward/roll-back machinery is augmented to process entire transactions atomically.

```
121 OpenTransaction
122 BehaviorSpecificCommand
        (Map, addstreet, street#12a)
123 AddPaintedStrokeCommand(stroke#23a1)
124 AddPaintedStrokeCommand(stroke#23a2)
125 CloseTransaction
```

Figure 14: An example of transaction.

8.4 Local versus Global Timeline Management

One final timeline management issue we had to deal with was the distinction between the "local" timelines of individual segments and the "global" timeline of the entire board. We wanted the ability for users to interact with the timelines of individual segments without affecting others: the entire history of a segment should appear continuous, even though in "real" time, operations on other segments may be interspersed with it. But we also wanted the ability to roll forwards and backwards in global (whole-board) time. Global undo and redo means that the histories of all individual segments are "packed" into a single timeline ordered by "real" time, rather than "segment logical" time.
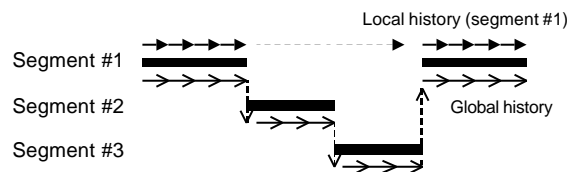

Figure 15: Local versus Global Timeline

In the implementation, each segment maintains its own local history, and Flatland creates the illusion of a global history timeline by composing individual segment histories together. Because users can visit and leave segments as often as needed, segment histories can be arbitrarily interleaved in "global" time—so the global history is represented as a list of "chunks" of history from individual segments, stitched together (Figure 15).

8.5 Persistence

Since Flatland whiteboards are designed for long-term use—much as physical whiteboards are—we needed a way to ensure that all data on the board is saved persistently. We want to ensure that *everything* on the board is saved and recoverable, for the entire duration of the board's use, without requiring users to have to explicitly save or name files that correspond to segments. Clearly this would violate the informal nature of the system and, in many cases, the work required to explicitly save a persistent data file would outweigh the benefit of using the board! We needed a much lighter-weight approach.

Flatland is built atop the Presto document management system [3]. Presto provides a loosely-structured "information soup" into

which arbitrary content can be stored. Presto presents an associative memory programming model to its users—chunks of information can be tagged with arbitrary key/value pairs—which maps nicely into the Java implementation of Flatland (Presto tuples can be arbitrary serialized Java objects).

Flatland saves every "dirty" segment periodically to stable storage via Presto. Each segment is represented as a discrete Presto "document," with Flatland-specific objects tagged onto it as key/value data. The contents of each document are the serialized command objects that constitute the segment's history. The system maintains a "segment cache," which reflects all of the "live" segments currently on the board. If an old segment needs to be retrieved (either because the user searched for it, or an undo or time slider operation causes the segment to become live again), it is "faulted in" from persistent storage. Only the storage layer in Flatland needs to know about persistence—from the perspective of behavior writers, all segments are always "live" and in core at all times. From the user perspective, users never have to explicitly save at any time, and they never name the data that is saved.

8.6 Search

Our search behavior retrieves past segment states using this document management system. The system tries to intuit information about the context of a segment's use, and its content, and uses this information to satisfy queries. For example, users can search based on content attributes such as segment stroke density (using ambiguous terms like "dense" or "sparse" or "medium") and color ('mostly blue"). Context-based searches can use information about what behaviors were attached to the segment ("my map" or "my to do list"), and time of last use.

The search result is displayed as a set of thumbnails representing past states of the segments. The construction of this thumbnail is done by rolling the addPaintedStroke and removePaintedStroke command objects forward starting from a blank segment, ignoring any behavior specific command objects in the segment history. This allows the system to reconstruct the segment appearance quickly. If the user tries to interact with the retrieved segment, the system reconstructs the behavior specific internal structures by rolling behavior specific command objects forward.

9. IMPLEMENTATION NOTES

Flatland is implemented in Java, and is approximately 42,000 lines of code. Handwriting recognition (used by the Calculator behavior) is done by the Calligrapher online recognizer from Paragraph Corporation.

We did not pay much attention to performance tuning, but the overall speed is satisfactory on a standard PC as a proof of concept prototype. Some operations such as the display of search results cause delay, and require improvement.

Implementation of history management is not yet complete. Especially, our long-term persistent history causes a sort of time travel paradox. Multiple restorations of an old segment and time traversal over merged or split segments badly confuse the timeline management, and more research is required to address these problems.

10. SUMMARY AND FUTURE WORK

This paper has introduced our efforts to build a software platform for a variety of pen based applications. Our design goal was to support informal, pre-production activities on a whiteboard, in contrast to the well-organized activities supported on desktop computers. To achieve this goal, we have introduced the ideas of strokes as a basic primitive for both input and output, dynamic segmentation of the screen space, pluggable behaviors working on a segment, and persistent history management mechanism.

Our next step is to deploy the Flatland system in real office environment to observe its usage. However, this is difficult with our current hardware set up, and its requirement for front projection. We expect that a large plasma display with touch sensitive screen will be a good solution. We also plan to implement additional application behaviors that support common activities on a white board such as paper outlining, communications, calendars, and so on.

Another interesting research direction is the application of our architecture to other pen computing environments. We believe that our architecture can provide a uniform framework for a variety of pen based devices to work in corporation.

11. REFERENCES

[1] Bederson, B.B., Hollan, J.F., Pad++: A Zooming graphical interface for exploring alternate interface physics, *UIST'94.*

[2] Bier, E.A., Stone, M.C., Pier, K., Buxton, W., DeRose, T., Toolglass and magic lenses: The see-through interface, *SIGGRAPH'93.*

[3] Dourish, P., Edwards, W.K., LaMarca, A., Salisbury, M., Using Properties for Uniform Interaction in the Presto Document System, *UIST'99.*

[4] Edwards, W.K., Flexible Conflict Detection and Management in Collaborative Applications. *UIST'97.*

[5] Edwards, W.K., Mynatt, E.D., Timewarp: Techniques for Autonomous Collaboration. *CHI'97.*

[6] Forsberg, A., Dieterich, M., Zeleznik, R.C, The Music Notepad, UIST '98

[7] Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing, 1995. Reading, Mass.

[8] Geissler, J., Shuffle, throw or take it! Working efficiently with an interactive wall, *CHI'98.*

[9] Gross, M.D., Do, E.Y., Ambiguous intentions: a paper-like interface for creative design, *UIST'96.*

[10] Kramer, A., Translucent Patches–dissolving windows, *UIST'94.*

[11] Kramer, A., Dynamic Interpretations in Translucent Patches –Representation-Based Applications-, *AVI' 96.*

[12] Igarashi,T., Matsuoka, S., Kawachiya, S., Tanaka,H., Pegasus: A Drawing System for Rapid Geometric Design, *CHI'98* summary, pp.24-25.

[13] Igarashi,T., Matsuoka,S., Tanaka,H., Teddy: A Sketching Interface for 3D Freeform Design, *SIGGRAPH 99.*

[14] Landay, J.A., Myers, B.A., Interactive sketching for the early stage of interface design, *CHI'95.*

[15] Moran, T.P., Chu, P., van Melle, W., Kurtenbach, G., Implicit structures for pen-based systems within a freeform interaction paradigm, *CHI'95.*

[16] Mynatt, E.D., The writing on the wall, INTERACT'99.

[17] Mynatt,E.D., Igarashi,T., Edwards, W.K., LaMarca, A., Flatland: New Dimensions in Office Whiteboards, *CHI'99.*

[18] Perlin, K., Quikwriting: Continuous Stylus-based Text Entry, *UIST'98.*

[19] Prderson,E., McCall,K., Moran,T.P., Halasz,F., Tivoli: An electronic whiteboard for informal workgroup meetings, *INTERCHI'93*, pp,391-399.

[20] Rekimoto,J., A Multiple Device Approach for Supporting Whiteboard-based Interactions, *CHI'98.*

[21] Rekimoto,J., Time-Machine Computing: A Time-centric Approach for the Information Environment, *UIST'99.*

[22] Zeleznik, R.C., Herndon, K.P., Hughes, J.F., SKETCH: An interface for sketching 3D scenes. *SIGGRAPH '96.*