ActiveText:
A Method for Creating Dynamic and Interactive Texts

*Jason E. Lewis**
*Alex Weyers**
Interval Research Corporation
1801 Page Mill Road, Building C
Palo Alto, CA 94304, USA
Tel: 1-650-842-6264
E-mail: {lewis, weyers}@interval.com

ABSTRACT
This paper describes ActiveText, a method for creating dynamic and interactive texts. ActiveText uses an object-based hierarchy to represent texts. This hierarchy makes it easy to work with the ASCII component and pixel component of the text at the same time. Static, dynamic and interactive properties of text can be easily intermixed and layered. The user can enter and edit text, adjust static and dynamic layout, apply dynamic and interactive behaviors, and adjust their parameters with a common set of tools and a common interface. Support for continuous editing allows the user to sketch dynamically. A prototype application called *It's Alive!* has been implemented to explore the ActiveText functionality. The documents produced by *It's Alive!* can be of use in a wide-range of areas, including chat-spaces, email, web-sites, fiction and poetry writing, and low-end film & video titling.

KEYWORDS: interactive text, typography, dynamic typography, continuous editing, dynamic sketching.

INTRODUCTION

The advent of the Internet has created a vast environment in which text is constantly on the move. The present landscape is filled with many different entities, from large corporations to design houses to individuals, who are interested in making web pages a more dynamic and interactive "read". When one takes into consideration the large traffic in email and chat-spaces, both of which are text-driven applications, the number of potential users for an easy-to-use tool for adding dynamic and interactive qualities to text is quite large.

* *Now at Arts Alliance AALab, 126 South Park, San Francisco, CA 94107. Email: lewis@thethoughtshop.com*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
UIST '99. Asheville, NC
© 1999 ACM 1-58113-075-9/99/11... $5.00

Most application currently used for creating text treat it as a fundamentally static medium. Dynamics and interactivity can be added to it, but only with considerable difficulty. The tools normally used are complex and expensive. Substantial training is required to use them well. These drawbacks could be ignored as long as the greatest consumers of such tools were based in high-cost industries like film and video production.

The current work attempts to address the much wider audience created by on-line communication. We assume that text is a dynamic medium. From that assumption we develop a method, ActiveText, that treats text as both character and image. Text can be given behaviors at various levels of granularity. Dynamic behaviors can be sketched out in real-time. The user can move fluidly from composing the text to adding dynamics and interactivity and back again. To illustrate the ActiveText functionality we have built *It's Alive! It's Alive!* is an application which supports rapid prototyping of static, dynamic and interactive text-driven documents. This paper describes the ActiveText architecture and the *It's Alive!* application.

MOTIVATION

The creation of a digital document that includes static, dynamic and interactive text is divided into several discrete stages. A user must compose the text, lay it out, manipulate the visual form at both gross and small levels of detail, apply dynamic behaviors and adjust the parameters of those behaviors, and specify any desired interactivity. We created the ActiveText architecture to make this process as fluid as possible.

Imagine that a user wants to make a short text-based piece which deals with how gossip can corrode the cohesiveness of a community. The user's idea is to have a single word, "gossip", affect a larger passage of text one word at a time. "Gossip" will move around the screen slowly, vibrating with excitement to convey the busy-body nature of gossips. When other words come into contact with "gossip", they will become "infected" with the same behavior, including the ability to spread their own behaviors further. Figure 1 shows several frames of what such a piece might look like.

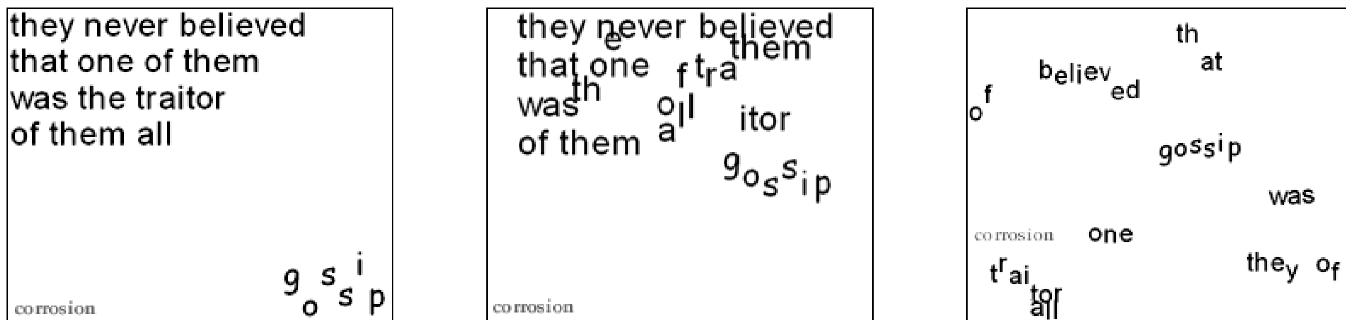

Figure 1: Screen-shots showing the progression of the "gossip" document. "Gossip" begins in the lower-left hand corner. It *cruises* randomly around that portion of the screen, its individual glyphs *vibrating*, until it makes contact with "all" in the larger body of text. At that point the *cruise* behavior—which resides on the word object—transfers to "all" and the *vibrate* behavior—which resides on the glyph objects which belong to the word object—transfer to the individual glyphs in "all". The *viral* behavior on "gossip" at the word-level gets transferred as well, with the result that "all" now moves on to infect other words just as "gossip" is doing. By the last frame all of the text in the upper left-hand block has been affected. The title, "corrosion", remains unaffected, as it possess the *immunity* behavior.

Working With It's Alive!

If the user has an ActiveText-based application like the *It's Alive!* prototype, the process would be as follows. First the user composes, edits and lays out the words on the screen. The user then selects the word "gossip" and applies the *cruise* behavior to it. A palette of properties particular to that behavior pops up, allowing the user to adjust the speed and the degree of directional randomness with which the word will move about the screen.

Next the user selects all of the individual glyphs of the word "gossip" and applies the *vibrate* behavior to them. Another palette pops up with parameters to modify the frequency and degree of vibration. (We use the term "glyph" as a general term for any marking, symbol or character,) Finally, the user re-selects the entire word. The *viral* behavior is applied at this level, instructing "gossip" to transfer all of its behaviors to whatever other text elements it contacts.

The user saves the file and then switches *It's Alive!* into simulate mode. "Gossip" wanders around for a bit and then hits the large body of text. Its behaviors start spreading from word to word and glyph to glyph. The text block slowly breaks apart until all of its words are wandering around the screen, vibrating.

At this point, the user realizes that that there are some spelling errors in the larger block of text, and that there is no title to the piece. The user can choose to copyedit while the document is in action, or by turning the simulation off. The user chooses the latter, loading the saved file to return to the original composition. The user also adds the title—"corrosion"—at the bottom left of the screen and applies the *immunity* behavior to it. This will keep the title from being affected by the *viral* behavior, and thus it will remain in place.

Working With Conventional Tools

Creating such a document with conventional tools would be a significantly more involved process. Current tools enforce a degree of compartmentalization that reflects the many stages enumerated above. Creating a document like the "gossip" piece requires that the user work first with Microsoft Word [1], then move to Quark Xpress [2], then to Adobe Photoshop, then finally to AfterEffects [3] and/or Macromedia Director [4]. Making the process more onerous, the user can move text data through the tool chain in the direction described without too much difficulty. However, moving them in the other direction, from Photoshop to Xpress, for instance, is hard if not impossible.

Applying behaviors such as those in the "gossip" piece would be even more time-consuming. If the user wanted to make an animation, each individual glyph would have to be hand-animated. The complexity of interaction between text objects as they come into contact with and infect one another would be a challenging task for a seasoned professional.

If the user wanted to circumvent the involved animating process, she could use Director to script the desired behaviors. This approach suffers from several deficiencies. Coordinating the *vibrate* behavior at the glyph-level with the *cruise* behavior at the word-level would require a custom object model. Even after custom-crafting this object model, the user would find it limited. It would not support multiple inheritance or polymorphism, both of which make managing the behaviors of objects within objects more tractable. The interpreter would struggle to deliver satisfactory real-time performance as the complexity of the object model grew. At run-time Director converts display-text to pixel-only representations, removing the possibility of copy-editing while the document is in action.

The difficulty in applying complex behaviors, the multiplicity of tools and the asymmetry between them conspire to make rapid prototyping of textural appearance,

dynamic behavior and user interaction extremely difficult for all but the most experienced users. Such a process wastes time, stifles creativity, and discourages experimentation.

Surmounting the ASCII-pixel wall

The ActiveText architecture is designed to give the user a single, unified creative environment for manipulating text. The first step is to break down the ASCII-wall (Figure 2). Tools which treat text primarily as ASCII are on one side of this wall. Examples are common programs for word processing, such as Word, and for page-layout, such as Xpress. The user can edit the text at any time, inserting words and deleting passages, because the software treats the text as a collection of alphabetical characters. On the other side of the wall are tools which allow the user to modify the appearance, dynamics and interactivity of text. Examples include Photoshop, AfterEffects and Director, respectively. These programs handle the text as collections of pixels with certain color values, and neither know nor care that the text had a character aspect. The user can alter the visual aspect of the letterforms, but she cannot do much in the way of editing. The result is that when a user wants to move text from the ASCII world to the pixel world, she must give up all information about the text as language.

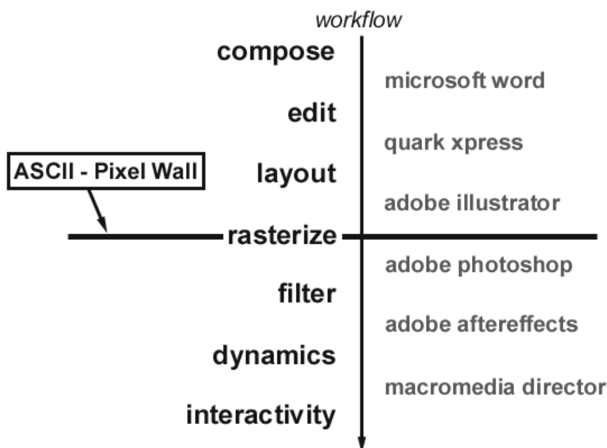

Figure 2: The location of the ACII-pixel wall in the flow of work required to make a dynamic and interactive text.

In the context where every byte of memory is precious, throwing away information in this manner is necessary to conserve storage space and maintain real-time responsiveness. The memory capacities and processor speeds of modern personal computers mitigate the need to be so conservative. If we retain such information not only can we move easily between ASCII and pixels, but we can also build upon it to offer other capabilities which are based on the syntactic structure of the text. We can cause all of the nouns in a sentence to vibrate, or all of the verbs to wriggle. We can perform dictionary and thesaurus look-ups and make substitutions on the fly. Or we can have all the articles in a sentence cluster into one corner of the screen as if shirking their yeoman's duties.

Stream vs. Object-Based Hierarchy

A common means of internally representing text is by parsing the text into a one-dimensional array. The resulting "stream" of characters is manipulated via an offset into the stream that determines where characters are added or deleted, what font starts and stops where, what size starts and stops where, etc. The only text object the application must maintain is the stream. This method works well for a static paradigm in which the two most important pieces of information about the text are the ASCII-value of each of the characters in the stream and the font (including face, size, and style.)

The stream method is not as well-suited for a dynamic paradigm in which different chunks of text are subject to behaviors and in which the visual rendering of a character changes from moment to moment. Among other difficulties, it would require heavy modification so as to support storage for and real-time updating of the pixel representations for individual characters. It would also be a challenge to manage the interaction of various behaviors, as the stream would require constant reparsing to find the relevant chunks.

ActiveText addresses these issues by employing an object hierarchy (Figure 3.) The text is parsed into meaningful chunks such as glyph, word, passage, and text objects. Each object can be manipulated both autonomously and by inheritance from the larger structures above it in the hierarchy. Each glyph stores both its ASCII and pixel values. Behaviors mediate for control at the appropriate object level. Everything in the architecture, including objects and behaviors, is agnostic about how the glyphs are actually rendered. This encapsulation enables the development of fonts which are dynamic and interactive in and of themselves. We call these fonts "SoftType" fonts.

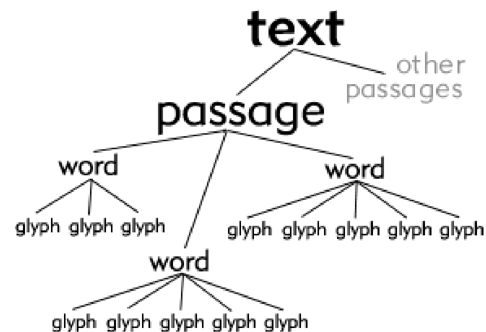

Figure 3: The hierarchy of text objects.

Continuous Editing

Current tools for doing dynamic and interactive text documents require the user to author in one mode and then switch to a separate mode to view the results. ActiveText supports continuous editing: all of the tools in *It's Alive!*, for instance, can be used for both static and dynamic composition.

RELATED WORK

Related work can be divided into two main groups,

experimental digital typography and *object-manipulation architecture.*

Experimental Digital Typography

MIT's Media Lab has conducted a large amount of research experimenting with alternative means of representing and manipulating on-screen text. Ishizaki's [5] dissertation on "typographical performance" anticipates the use of coordinated behaviors to create a rich interplay of text in an email system. Wong's [6] thesis on "temporal typography" uses Soo's [7] object-based, behavior-driven architecture to deftly marry dynamics and semantics. Chao [8] developed a system for specifying behaviors for the visual presentation of program code. Small's early work on text-based information landscapes [9] and later work such as *Stream* [10] experimented with different methods for representing text documents. These efforts all tackled different aspects of adding complex dynamics to text, often with compelling visual results.

On the level of innovative glyph representations one must turn to Maeda [11], who has a long history of exploring the dynamic aspects of letterforms. Rosenberger [12] developed a font that responds to human vocal prosody by modifying the visual aspect of individual glyphs, while Cho [13] created a series of malleable and animated fonts**.** Haeberli has created a tool for dynamically altering the shape and look of individual three-dimensional letterforms (personal communication.). All of this work, however, is sealed into its own specialized applications. Using any of it in an full-featured dynamic word processing environment would require extensive rewriting. The SoftType component of the present work seeks to make such radical glyph representations readily available and usable in a complete writing environment.

Lewis' [14] work on WordNozzle makes a analogy between stream representations of text and streams of paint applied, grafitti-like, to a canvas. This work highlighted some of the limitations of working within the stream paradigm, particularly when attempting to apply behaviors to a large body of text.

MotionPhone by Snibbe [15] removes the distinction between authoring and performance modes. The result is a fluid, accessible tool which is particularly well-suited to experimentation but proves quite difficult to use as a composition tool. This difficulty stems from the fact that the user is forced to constantly "catch up" to his own document. The continuous editing approach taken in the present work addresses this issue by giving the user the option to edit in both static and dynamic modes.

Several commercial products can be used for creating dynamic texts. At the high-end are film effects system such as Discrete Logic's Flame and related systems [16], and Quantel's Harry[17]. These systems run on specialized hardware and require extensive customization to use fluidly. The documents they produce are not interactive. Adobe AfterEffects, which dominates the middle-range, produces non-interactive video loops. At the low end are products like Microsoft's Liquid Motion and Macromedia's Flash, with very limited interactivity and weak scripting environments.

Object Manipulation Architecture

ActiveText's hierarchy of text objects is similar to any number of scene graph hierarchies used in graphical applications, starting with Lakin [18]. We specialized the ActiveText architecture to text in order to concentrate on fully exploring the possibilities such an approach offers to typographic and textual manipulation.

Similarly, the text objects are specialized "actors", in the sense meant by Reynolds [19]. They possess and process behaviors in a manner similar to Perlin and Goldberg [20], though the present implementation does not ahieve Perlin and Goldberg’s level of sophistication with regards to mediating conflicting behaviors. The text-objects have behaviors which can be layered, they pass messages back-and-forth, and respond to other text-objects and general environmental conditions. The present work attempts to specialize such techniques exclusively to text in an attempt to tap into heretofore unrealized methods for handling text-based documents.

THE ACTIVETEXT ARCHITECTURE

ActiveText is an architecture for creating dynamic and interactive texts. It consists of a set of C++ libraries written on top of the Microsoft Foundation Classes [21]. The architecture parses text into an object-hierarchy based on textual granularity. Character information is stored as both ASCII- and pixel-values. Behaviors for transforming the text objects utilize a message subscription model to coordinate between and mediate among themselves. Object encapsulation enables continuous editing.

The architecture meets several challenges. In order to be usable in a wide range of areas, it has to support dynamic and interactive behaviors of all forms. The text objects on which these behaviors are operating must interact with a particular behavior at the level or levels which is/are appropriate. For instance, if the user places the *move to mouse* behavior on individual glyphs, the interaction between glyph objects and behavior must reflect this by moving the glyphs independently towards the mouse. If the user places *move to mouse* on a word objects, the appropriate result is to have the word move as a unit towards the mouse.

Text objects must have a means of notifying a behavior if it is not a candidate for the behavior's particular function. An example of such a case is the *synonym shift* behavior, which uses the WordNet [22] lexicon to cycle a word through its different synonyms. A synonym is a word-level phenomenon; it makes no sense to apply the concept to a glyph. Yet it does make sense to apply it to a passage if the user wants to have all of the words in a passage cycle through their synonyms. The architecture must support such nuance.

In order to support complex, layered behaviors, the architecture must provide a means to negotiate control of a particular object. It must also support the activation and deactivation of behaviors by other behaviors. Similarly, support for SoftType fonts requires that the architecture decouple how behaviors operate on text elements from how the actual glyphs themselves are drawn.

Finally, the continuous editing requirement means that all of this coordination must be possible while the document is in motion and the user is interacting with it. If the user deletes an object, wherever it is in the hierarchy, the architecture must remove it cleanly and notify behaviors which are acting upon it of this removal. If objects are added, such as adding a word object to a passage object, the behaviors must be able to incorporate the new object into their actions on the fly. Further complicating matters is the fact that not only can the user add elements and behaviors and adjust their properties, but other behaviors can do this as well.

Hierarchy of Text Objects

The basic element in the ActiveText data structure is the book object. The book functions as both an encapsulation of the text objects and an engine for driving behaviors. Every new ActiveText document creates a single book, and it is the book that is stored and retrieved when a file is saved or loaded.

The Text object contained within the book functions as the root node for the entire hierarchy of text objects. (We will use a capital "T" to differentiate between this particular root node and "text objects" as a general term for glyph, word and passage objects.) The Text object is composed of passage objects, which are in turn composed of word objects, which are in turn composed of glyph objects (Figure 3). As a user enters text into an ActiveText document, each new character is stored as a glyph object. Every collection of glyphs separated by spaces or other punctuation is considered a word and linked to its own word object. Finally, all words are joined into a passage object. New passages are created by placing the cursor outside of the existing text.

Text, passage, word and glyph objects are all derived from the same base object class. This class maintains pointers to the other members in the object hierarchy. Each object inherits a set of basic properties that includes font, color, and position. The positions of all objects are relative to that object's parent. Consequently, a glyph stores its location as a position within the coordinate system of its parent word. The position of a word is an offset from its parent passage. This arrangement facilitates the sensible positioning of objects at different levels within the hierarchy, either by the user or by behaviors that need to naively operate on objects regardless of their type.

The encapsulation of the objects in the hierarchy allows ActiveText to support continuous editing, i.e., the composition, layout, and creation of complex behaviors while the document is in action.

Behaviors and Messages

ActiveText behaviors are objects that alter the properties of text objects over time or in response to user interactions. Such properties include font, color and position. Each behavior has a list of subjects (i.e., text objects) upon which the behavior acts (Figure 4.) Both user interaction and the activity of other behaviors can modify the subject list. Behaviors and objects are coordinated using a message subscription model [23].

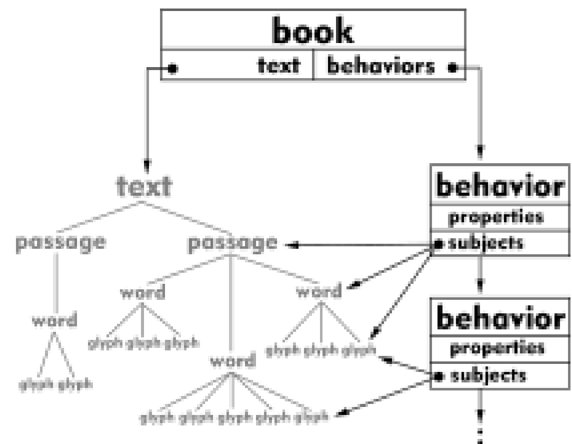

Figure 4: The interaction between the hierarchy of text objects and the list of behaviors. Text objects are subjects of a particular behavior. The properties of the behavior determine how it modifies the properties of the text objects which subscribe to it.

When a behavior is instantiated by either the user or by another behavior, the instance is added to the book's list of behavior objects. Initially, behaviors are inactive. At the point of activation they subscribe to the appropriate messages. The book will then notify that behavior instance when an event of interest occurs. Similarly when a behavior is deactivated, its instance is unsubscribed from all of the messages with which it had registered.

Primitive behaviors such as *move to mouse* and *fade* respond to messages by altering the state of properties which belong to all members of its subject list. For instance, the *move to mouse* behavior subscribes to the system :movemouse message. When an instance of *move to mouse* receives a :mousemove message it responds by altering the position values of its subjects to decrease the distance between the subjects and the mouse.

Behaviors are capable of creating new properties on members of their subject list. These properties can be accessed by behaviors other than the one which created them. Behaviors which operate on the same properties do so successively (Figure 5.)

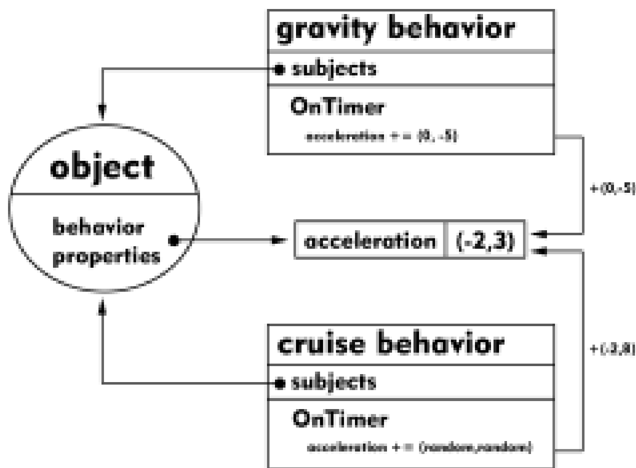

Figure 5: The result of two behaviors, *gravity* and *acceleration*, acting on the same text object.

By creating, deleting or reordering objects, behaviors can manipulate the gross structure of the text. They can also create or destroy other instances of behaviors.

Complex Behaviors

The creation of complex behavioral interactions is accomplished through the use of a particular derived class of behaviors called trigger behaviors. These trigger behaviors contain lists of behaviors to activate and deactivate in response to the trigger. A trigger can take the form of a wide range of possible stimuli including time-offsets, movement of the mouse, or some chosen state of a text object's properties. Trigger behaviors thus become the building blocks for creating complex interactions without the requirement that the user learn a scripting language. Simple time-based animation is accomplished by creating a set of time-based triggers for each moment of relevant change.

The SoftType Fonts

ActiveText supports standard TrueType fonts as well as dynamic and interactive variations on TrueType outlines. These variations are called SoftType fonts. A glyph in a SoftType font will change shape in response to user action, behavior intervention or timer messages (Figure 6.)

In order to safely mix standard TrueType fonts and SoftType fonts, the rendering of glyph outlines is performed by an ElementRenderer object (Figure 7.) The ElementRenderer determines how and when the object will draw itself on-screen. An object without an ElementRenderer will not draw itself. The first time a text is entered and parsed into objects, each glyph is given an ElementRenderer set to the default or standard text output. Words and passages do not have an ElementRenderer unless the user or another behavior explicitly assigns one.

Like behaviors, ElementRenderers are capable of subscribing to messages. They can be applied at any level in the hierarchy. Unlike behaviors, an object has at most one ElementRenderer. This restriction is to prevent the needless activity that would occur if multiple ElementRenderers attempted to draw the same object.

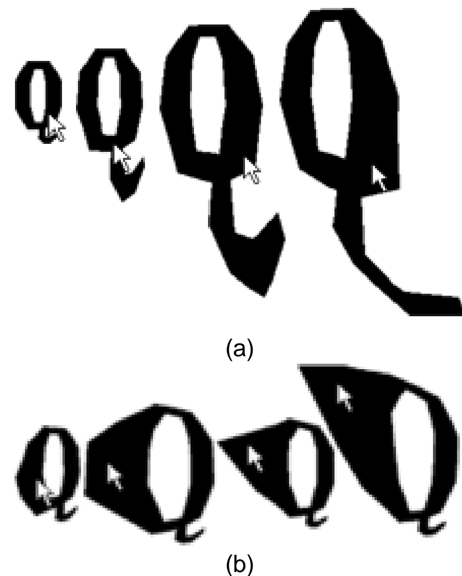

Figure 6: Screenshots of a "Q" distorting in response to mouse movement over it. 6a shows successive frames as the mouse moves around the lower-right portion of the letter. 6b shows successive frames as the mouse moves around the left and upper-left of the letter.

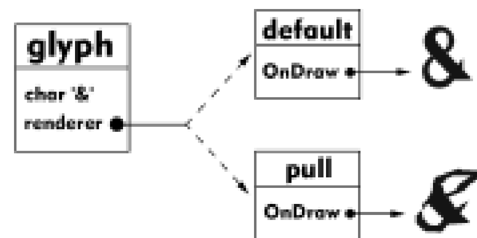

Figure 7: The ElementRenderer renders glyphs as either unmodified TrueType outlines (top) or as SoftType (bottom).

THE *IT'S ALIVE!* PROTOTYPE

It's Alive! is a prototype application which uses the ActiveText libraries to create an interactive- and dynamic-text processor. *It's Alive!* is written in C++ for the WindowsNT/95/98 platform.

Description

It's Alive! is a single-window environment (Figure 8.) Within this window is a canvas. Text is entered by locating a cursor on the canvas with the mouse and typing. Basic editing functionality such as adding and deleting text is accomplished in this direct manner.

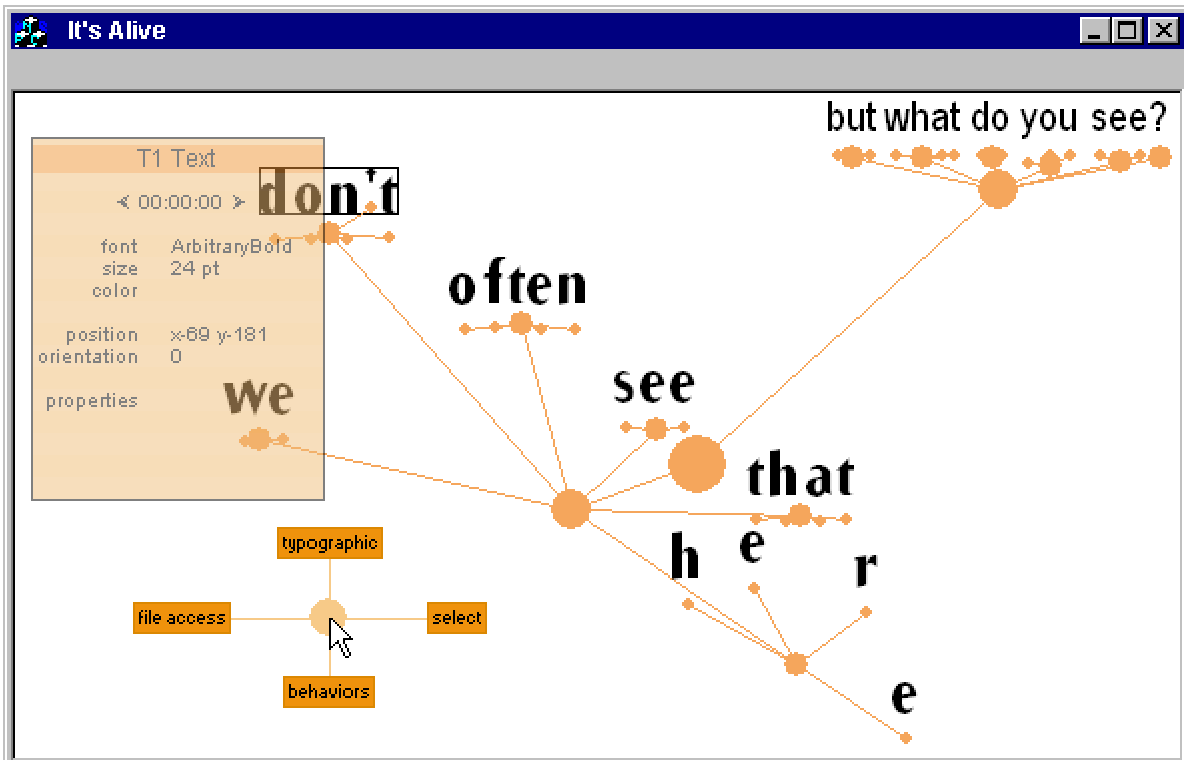

Figure 8: The *It's Alive* application with transparent palette upper-left, pie menu lower-left and "spider" display connecting the components of the text hierarchy.

Two types of pop-up windows, or palettes, are used. The properties palette displays the properties of a selected object. The behavior palette displays a specific behavior and provides access to the parameters of that behavior. Following Bier, et al, [24] the palettes have variable transparency. This allows the user to make adjustments while maintaining view of the full canvas.

In order to keep track of what text objects are related to each other and how, *It's Alive!* uses a node-and-spoke display. An example can be seen in the middle of Figure 8. Small circles are placed under each glyph in a word. Spokes connect each glyph to a medium circle placed under the center of the word. This circle is in turn connected to a large circle which is at the center of the passage to which it belongs. All of the passage are connected by spokes to the largest circle, which is placed at the center of the entire text. This display is called a "spider" display within the document. With it the user can see which glyphs are related to which words, which words to which passages, etc. This is particularly helpful for dynamic sketching, or while editing complex documents.

Environment, property and behavior functionality is accessed through a pie menu system derived from Hopkins [25]**.** The pie menu is activated by the right mouse button (see lower-left of Figure 8.). It has four axes: *typographic, selection, behaviors,* and *file.* Each of these axes opens out into a series of submenus:

Behaviors The user uses this menu to apply or remove

behaviors to and from selected text objects (Figure 9.). This is also where the user can choose to turn the dynamics on or off by selecting "simulation".

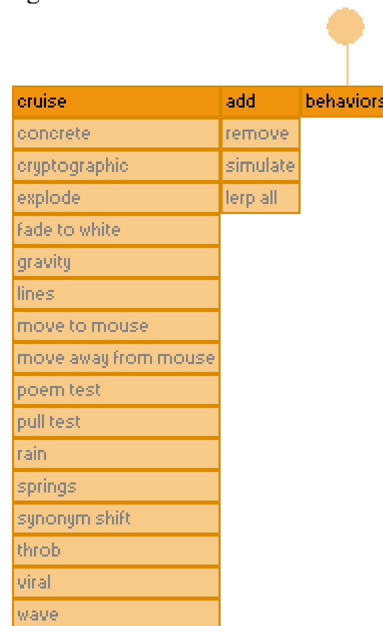

Figure 9: The behavior menu.

File Operations such as creating a new document, loading a file, QuickTime output and other exporting functionality are accessed here.

Typographic Controls the appearance of glyphs in the text (Figure 10.) The user can change the font, size, color and style from this menu. All of these submenus provide instant feedback, e.g., as the user moves the mouse through the list of fonts the font of the selected text changes appropriately.

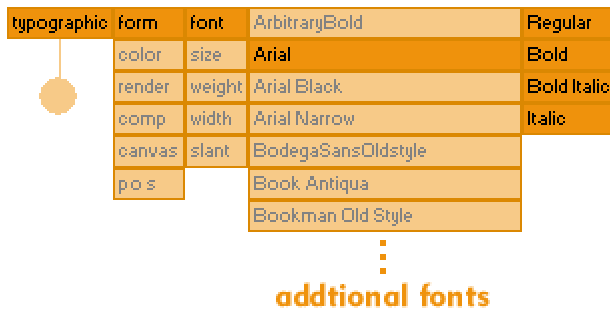

Figure 10: The typographic menu selected out to 4 levels.

Selection Allows the user to alter the granularity of a selection (Figure 11.) For instance, if the user has selected an entire passage, choosing *words* will select each of the individual words in that passage. Conversely, if the user has selected a word, choosing *passage* will select the entire passage of which that word is part.

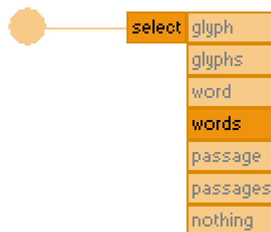

Figure 11: The selection menu showing the various ways in which the user can get a handle on the text.

User Experience Study

We conducted a user study on a highly restricted set of the ActiveText functionality embodied in *It's Alive!* The subjects' exposure to the full capabilities of the prototype was limited in order to focus on how they reacted to behaviors, and, in particular, if they thought it possible that they would make use of such behaviors in their daily text-based communication. The *It's Alive!* interface in all its complexity was not tested. For most task, users button-clicked through different examples of simple documents made with *It's Alive!* In one task they were themselves able to add, remove and layer behaviors.

The subject pool was composed of eight dyads of experienced personal computer users. All were from a nearby undergraduate and graduate university. Subject ages ranged from 20 to 24 years old, with an equal number of males and females.

Preliminary results from the test suggest that users found the behaviors interesting and novel, but it is unclear how much they might use them in every day communication. Users showed the most interested when layered behaviors produced unexpected results, such as adding a *wave* motion to an *explode* behavior, or by combining any of the SoftType fonts with anything else. The subjects clearly thought it necessary that the application of behaviors require little time or overhead, particularly for communication media such as chat and email. We look forward to testing the full *It's Alive!* interface to see if it is lightweight enough to address their concerns.

The subjects' responses were not the only valuable outcome of the study. No crashes happened over a week of rigorous testing, suggesting that the ActiveText libraries underlying *It's Alive!* possess a fair degree of robustness.

Continuing Work

The *It's Alive!* prototype is only in its first phase. The current phase of implementation includes numerous extensions to the current functionality.

A simple timeline allows the users to specify when objects appear on screen, when behaviors turn on or off, and get an overview of longer documents. An environment-level behavior called ActivePaper has been defined. ActivePapers are regions of the screen which interact in a particular way with whatever text passes through them. The incorporation of a Postscript module for exporting individual glyphs makes *It's Alive!* available for use as a tool for experimenting with typography.

As it is currently written, ActiveText anticipates run-time registration of behaviors and SoftType fonts. Except for text objects, all the critical objects in the system possesses a unique identifier. These identifiers are declared statically in the current version of *It's Alive!*. In the next version these identifiers will be generated at run-time as part of registration process. Run-time registration will allow third-parties to create plug-in behaviors and SoftType fonts.

Negotiation between conflicting behaviors is presently quite rudimentary, following a simple precedence model. We anticipate a more sophisticated solution growing out of further research into architectures such as Perlin's IMPROV system.

DISCUSSION

The ActiveText architecture improves the process of creating a dynamic and interactive text. It releases the user from having to worry about crossing over the ASCII-pixel wall and not being able to return. It permits the user to both compose statically and sketch dynamically. The architecture supports increasingly complex applications of behaviors, and provides a robust means for incorporating radical glyph representations such as those represented by the SoftType fonts.

Arriving at the present implementation involved significant

experimentation and learning. From that process, we have selected a few points that may be of particular interest to anyone interested in utilizing the existing libraries or implementing a similar architecture.

The object model complicates basic editing functions. In moving away from a token stream text representation we gain greater flexibility but also create new problems. Some of the most basic editing tasks became rather complex. For instance, a stream model is not concerned with position on the screen but with position within a one-dimensional array. Actually placing the text in the proper position, ensuring that it wraps in the correct place and so forth is a task given to the interface. In the ActiveText architecture, position is stored as a two-dimensional value. Every text object calculates and stores its position as an offset into the coordinate system of its parent object. The position property of every object requires constant maintenance while the user composes, edits and otherwise modifies the text.

Designing a tractable hierarchy is not the same as designing a meaningful one. The composition of the object hierarchy above the level of the word and below the level of the text, i.e., at the level we presently call the "passage", is questionable. We considered several other chunks at this level, including sentences and phrases. "Passage" proved to be general enough to encompass both terms. Yet it denotes a vagueness which is unsatisfying, and which does not reflect the way in which people talk about language. We anticipate user studies will assist us on resolving this matter.

Complex interactions among complex behaviors frustrates predictability. Once behaviors reach a sufficient degree of complexity, it is difficult to predict the outcome of their interaction. Seen one way, this provides a constant stream of pleasant surprises on the order of "that's interesting; how did that happen?" and is a fruitful way of exploring unexpected compositions. Seen another way, this lack of predictability leads to frustration as the text behaves in undesirable ways. One solution to the predictability issue is to rigorously divide behaviors into a set of primitive behaviors and a set of compound behaviors, with the former built upon the latter. The primitive behaviors would be simple enough that the user could reliable predict what certain combinations of them will be like. Another solution may lie at the interface, in presenting behaviors to the user in functional groupings and in providing a quick preview mechanism.

Managing complex interactions among complex behaviors is an unending tuning process. It took us some time to fine-tune the behavioral structure such that it is manageable from a programming standpoint. The subscription model helps manage the complexity, yet it does not remove the necessity for ongoing modifications to the way behaviors interact.

CONCLUSION AND FUTURE WORK

We have argued that current tools for creating dynamic and interactive texts limit the ability of the user to conceive of, experiment with and produce such texts. The proposed ActiveText method supports a more expressive and fluid method for creating such texts. The object-based hierarchy, combined with dual ASCII-pixel representations, a mechanism for sensibly applying and coordinating behaviors, and rigid data encapsulation allow the user to easily modify the text in interesting way in both static and dynamic modes. We have described a prototype, called *It's Alive!*, which is built on top of ActiveText libraries. Finally, we have reported cautiously positive results from a user-experience study in which subjects were exposed to a limited portion of the *It's Alive!* application.

We anticipate that the next major phase of research will address issues such as:

Integration into on-line applications. We would like to investigate ways of making the ActiveText functionality accessible to browsers, email applications and chat. This will most likely be done by constructing the appropriate plug-ins, but may also be accomplished within a Java or modified Flash framework.

Performance version. We also wish to develop a version suitable to live performance of spoken word, perhaps incorporating functionality similar to Rosenberger's Prosodic Font.

Further iterations on the user interface. We would like to further refine the user interface, and test it to see if it is indeed an improvement over the standard toolbar interface found in conventional text-tools.

Integrating computational linguistics. The trajectory we are most excited about following is to firmly tie the text objects into techniques for performing linguistic computation. The text's existence as a fully parsed structure provides easy access to words and passages/phrases/sentences. The addition of a dynamic parts-of-speech tagger would enable users to specify behavior based upon whether a word is a noun or a verb, etc. Combined with lexical information similar to that provided by WordNet, a wealth of potentially interesting behaviors operating simultaneously on semantics and aesthetics should become possible.

ACKNOWLEDGEMENTS

Scott Snibbe and Douglas Soo assisted in developing SoftType, and conversations with Sha Xin Wei led to the ActivePaper concept. Jenny Dana and Tom Ngo helped with the intricacies of mediating behaviors. Warren Sack provided sound advice on computational linguistics, and Gavin Miller lent optimization assistance. Debby Hindus provided invaluable commentary on several drafts of this paper.

REFERENCES

1. Microsoft Corporation. Product specifications. On-line at http://www.microsoft.com/office/95/word.htm.

2. Quark Corporation. Product specifications. On-line at http://www.quark.com/

3. Adobe Corporation. Product specifications. On-line at http://www.adobe.com/.

4. Macromedia Corporation. Product Specifications. On-line at http://www.macromedia.com.

5. Ishizaki, Suguru. Multiagent Model of Dynamic Design: Visualization as an Emergent Behavior of Active Design Agent. In *Proceedings of CHI '96 Human Factors in Computing Systems,* pp. 347 - 354, Vancouver, BC, April 1996.

6. Wong, Yin Yin. Temporal Typography: Characterization of time-varying typographic forms. Masters thesis, Massachusetts Institute of Technology, 1995.

7. Soo, Douglas. Implementation of a temporal typography system. Masters thesis, Massachusetts Institute of Technology, 1997.

8. Chao, Chloe. *Kinetext:* A concrete programming paradigm for kinetic typography. Masters thesis, Massachusetts Institute of Technology, 1998.

9. Small, David. Navigating large bodies of text. IBM Systems Journal, Vol. 35, No. 3&4, 1996.

10. Small, David and Tom White. An Interactive Poetic Garden. In *Proceedings of CHI '98 Human Factors in Computing Systems,* pp. 303-304, LA, CA, April 18-23, 1998.

11. Maeda, John. Flying Letters. Reactive Book Series, Number Two. Digitalogue, Tokyo, 1996.

12. Rosenberger, Tara. Prosodic Font: The space between the spoken and the written. Masters thesis, Massachusetts Institute for Technology, 1998.

13. Cho, Peter. Pliant Type: Development and temporal manipulation of expressive, malleable typography. Masters thesis, Massachusetts Institute for Technology, 1997.

14. Lewis, Jason. WordNozzle: Painting with Words. Sketches section of the Special Interest Group Graphics (SIGGRAPH) Annual Conference, Los Angeles, August 1997.

15. Snibbe, Scott. MotionPhone. Interactive communities section of the Special Interest Group Graphics (SIGGRAPH) Annual Conference, Los Angeles, August 1995.

16. Discrete Logic. On-line at http://www.discrete.com.

17. Quantel Corporation. On-line at http://www.quantel.com.

18. Lakin, F.H. A structure from manipulation for text-graphic Objects. Computer Graphics (SIGGRAPH '82 Proceedings), 14 (3), pp. 100-107, July 1982.

19. Reynolds, C.W. Computer animation with scripts and actors, Computer Graphics (SIGGRAPH '82 Proceedings), 16 (3), pp. 289-296, July 1982.

20. Perlin, Ken and Athomas Goldberg. IMPROV: A System for scripting interactive actors in virutal worlds, Proceedings of SIGGRAPH '96, pp. 205-216, August 1996.

21. Microsoft Visual Studio 6.0. Microsoft Corporation, Redmond, WA, 1998.

22. Fellbaum, Christiane (ed.) WordNet: An electronic lexical database. MIT Press, Cambridge, MA, 1998.

23. Gamma, Erich, Richard Helm, Ralph Johnson and John Vlissides. Design Patterns: Elements of reusable objected oriented software. Addison-Wesley, Reading, MA, 1995.

24. Bier, Eric A., Maureen C. Stone, Ken Pier, William Buxton, and Tony DeRose. Toolglass and Magic Lenses: the see-through interface.

25. Hopkins, Don. The design and implementation of pie Menus. *Dr. Dobb's Journal*, December, 1991, 16-26.