Hypertext in the Open Air: A Systemless Approach to Spatial Hypertext

Jim Rosenberg
555 Davidson Road
Grindstone, Pa 15442
E-mail: jr@amanue.com

ABSTRACT

This paper presents a personal spatial hypertext authoring system called The Frame Stack Project, implemented as a lightweight set of classes in the generic object framework Morphic, available in the programming environment Squeak. Morphic provides a kind of off-the-shelf toolkit of objects and behaviors extremely relevant to spatial hypertext. In this project, run-time vs. authoring behavior is a state property of individual objects in a highly granular way. A key goal is the support of feral structure, in which objects can be created loose on the desktop, without assigning them any structural destination. This provides an implementation of an interactive version of the poet's notebook. The granular approach to object authoring supports "interactive writing" in the truest sense of the word.

INTRODUCTION

This paper presents an ongoing project called (loosely) The Frame Stack Project. The term 'frame stack' describes an interface concept which I have been using for a number of years, and have described previously [9]. It provides a user interface for overlaying word objects on top of one another, while still allowing them to be read legibly. In the past, "frame stack" has been more of a conceptual artistic framework than an actual implementation; until recently it would not have been possible to examine any of my finished works and uncover actual objects identifiable as frame stacks. This paper describes the implementation of the frame stack concept as actual working code. It has resulted in a kind of personal spatial hypertext authoring system, but of a somewhat different kind than usual. Rather than an "application", within which spatial hypertext development takes place, the Frame Stack Project consists of a lightweight set of classes that operates within a generic object desktop. This allows word objects — complete with interactive behavior — to be simply "loose on the desktop" — a concept discussed below as *feral structure*. Feral structure is closely related to the classical philosophy of spatial hypertext: that the user must have the ability to postpone creation of structure. The specific structural operation which is postponed in the case of a feral object is *parenting*. Accreting of "parentless" objects is a method of facilitating, in software, a time-honored central part of poetic practice: scrap collecting. Most poets keep some form of notebook, in which scraps are accumulated, often without any clear idea what the ultimate destination of the scrap will be at the time it is first written down. A scrap is thus an inherently parentless object.

The generic object framework within which the Frame Stack Project is realized is called Morphic [12] (discussed below), and is provided off the shelf in the programming environment Squeak [3]. Of course it is somewhat disingenuous to describe this approach as "systemless" — after all Squeak could certainly be considered a system. But can it be considered a "hypertext system"? Most researchers would probably agree this would be a stretch.

MORPHIC

Morphic is a user interface paradigm providing a wide variety of graphical facilities. A "morph" is an instance of the class Morph, or one of its subclasses. Morphs can contain other morphs; when a morph is moved on the desktop, its submorphs move with it. When a morph is selected by a mouse-up combined with a keyboard modifier (which depends on the operating system) — e.g. "command-click" on the Macintosh — a set of icons called a "halo" appears. (See Figure 1).

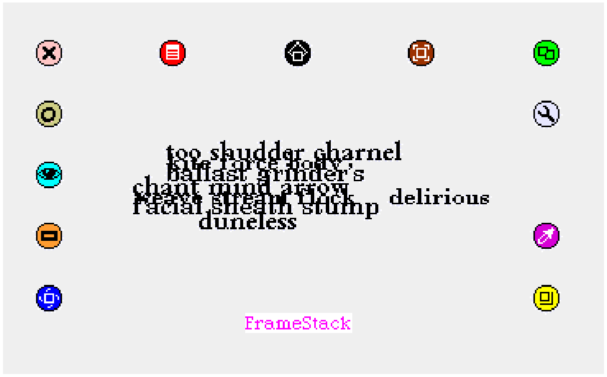

Figure 1

A set of icons called a "halo" is popped up surrounding a morph by clicking a morph with a keyboard modifier. These icons provide a user interface to standard morphic behavior for any morph.

The halo mechanism provides an interface to a set of generic behaviors of all morphs, which subclasses are free to override if necessary. Among these behaviors are: moving, pickup, resizing, menu, iconify, and delete.

Pickup

When a morph is picked up, e.g. by dragging the top middle icon fron the halo, this means not only moving it, but in addition when the mouse is released, dropping it into a target morph, so that the morph being picked up becomes a submorph of the target. However, morphs are free to reject drops. It is interesting to contrast this concept with the way that aggregation works in familiar spatial hypertext systems such as VIKI [8] or VKB [10]. VKB, for instance, provides a mechanism called Collections. In order to aggregate spatial objects, they are placed "into" a collection. A bit of text, by contrast, "isn't" a collection; in VKB a text object is a "terminal node" in the structure, and cannot have subobjects. Morphic, on the other hand, assumes that (1) *any* object may contain subobjects and (2) the decision whether or not an object should contain subobjects is made "on the fly". (To a poet this means: one can change one's mind about this!) I.e. for a morph to change state in either accepting or rejecting pickup, or to contain or not contain another morph as a submorph, is not a change of class. By contrast, if a phrase in VKB is treated as an "object" (terminal node) and the document author suddenly decides this phrase needs to have subobjects, "changing" the phrase so this is possible is a very heavy-weight activity. The user must (1) create a new collection; (2) copy the phrase into the title of the collection, (3) delete the original phrase. The Morphic concept of submorphs may be said to be closer to the spirit of spatial hypertext than a structural concept like collections, in that the decision whether an object is to "have" subobjects can be postponed. (As we speak repeatedly in spatial hypertext discussions about postponing the realization of structure, it must be emphasized that at the moment when structure procrastination ends, conversion to structure needs to be as light-weight an activity in the user interface as possible!)

Morphic makes a distinction between a morph accepting drops and a morph containing submorphs. Rejecting drops is considered a user interface property; however all morphs contain the mechanism needed to contain submorphs, and even if a morph rejects drops, submorphs can still be added programmatically by a Smalltalk method. It should be noted that Morphic does contain one serious user "hazard": if a morph is enabled for receiving drops (in order to "build it up" interactively), and then that morph becomes "finished", the user may forget to turn off acceptance of drops. If this morph becomes embedded in a larger morph, it may "attract" a drop that the user thought was going into the larger morph. I.e. in the version of Morphic used for this project there is poor visual identification of the target of a drop.

Event Handling

An instance of a subclass of Morph can "register" with the Morphic event handling system that it wishes to receive events, such as e.g. mouseEnter. In this case the object will be sent a mouseEnter message. This allows any morph to create its own user interface behavior. There is an important point here. Unlike complex paradigms such as the Model View Controller paradigm [5], a user interface designer employing Morphic need only subclass *at a single point* in the class hierarchy. Thus, to create the class FrameStack, it was only necessary to subclass RectangleMorph. An MVC approach would require subclassing at three places: for the model, view and controller. Smalltalk presents the novice with a gordian knot learning problem: where to subclass? Most programmers learn a new language by writing code. But in Smalltalk, one cannot simply "write a program"; all code must go into classes, and one cannot know where one's class should go in the hierarchy without learning the hierarchy, which one can't do without learning Smalltalk ... The genius of Morphic is that it provides a ready set of classes avail-

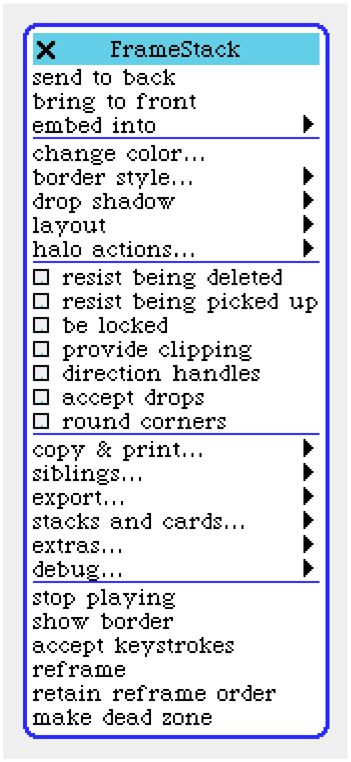

Figure 2

The standard morph menu for a frame stack. Note the "custom methods" at the bottom. Methods above "stop playing" are inherited from the superclass, and ultimately mostly from Morph.

able for subclassing whose function is completely intuitively clear. E.g. it was intuitively obvious that some of the important classes for this project needed to be subclasses of RectangleMorph.

Menus

A morph inherits a "standard menu", but the programmer creating a Morphic subclass can easily customize this. This provides an easy method to attach an interface to one's own classes. Figure 2 shows the menu for the class FrameStack. The entries at the bottom correspond to custom methods. E.g. reframe — the most complex piece of code in this project — tells a FrameStack to abandon its existing frames, make new ones for each of its submorphs, and resize itself appropriately.

It is also worth commenting in more detail on the menu choice seen in Figure 2 "start playing". When a frame stack is "playing", as the mouse enters its boundary, the frames for the submorphs are made visible, and one of the submorphs is selected at random to be on top. The frames are opaque white, and are active mouseEnter regions. As the mouse enters one of the frames, it and its designated morph are brought to the front, making them visible. When the mouse leaves the boundary of the frame stack, all of the frames are made invisible. When the frame stack is not playing, it is impervious to mouseEnter. In effect, "playing" constitutes *the run-time behavior* of the frame stack object. The point here is that unlike environments like Flash [6], where authoring and run-time behavior are so completely separate that they take place in separate applications (with widely divergent kinds of licensing!) in Morphic a distinction (if needed) between authoring and run-time behavior can take place in a highly dynamic granular fashion as a state of *individual objects*. Run-time behavior for an object can be "left on" until it "gets in the way"; at that point it can be turned off for that individual object. This subject will be discussed further below.

STRUCTURE VS. PRESENTATION

It is customary in hypertext that structure and presentation should be separated. Following a link is a structural operation. There may be a wide variety of ways of presenting this operation to the user, even though the structural operation is in each case the same. In spatial hypertext, however, we take a different point of view. Presentation is what "replaces" structure at a point where the user is not willing to commit to structure. Or, to put it somewhat differently, "spatial structure" and presentation are inseparable. Thus for instance while "nearby" is a presentation property, it is of essential importance in replacing a structural concept in spatial hypertext. Morphic provides a ready stock of presentation abstractions which can be used by spatial hypertext objects as a kind of off-the-shelf toolkit. In this section we review some of them.

"The Front"

Morphic provides a "layering order"; it knows which objects are in front of other objects, and can render them appropriately. A morph can be brought to the front or sent to the back. This operation does the right thing regarding the submorph hierarchy. E.g. sending a morph to "the back" sends it behind all other submorphs of the same parent, but does not send it behind its owner. The presentation concept of bringing a morph to the front can substitute for the structural operation of "navigating to" the morph. For instance, consider a "card interface": to the user it appears that she is navigating among a set of cards, which contain various objects. Some of these act as buttons, which take the user to other cards. Provided the "bottom layer" of each card is arranged to be opaque, this interface can be implemented in Morphic by having each card be a rectangular morph of the same size and same origin in the coordinate system and then "navigating to" a card by simply bringing it to the front. Technically, the user is "at" all of the cards (at once) but from the point of view of user experience, the only card that is visible is the one on top (at the front, in Morphic terminology).

Using the ability to bring objects to the front as a substitute for structural navigation can have some tricky consequences. Just as an interface can suffer from contention over screen real estate, there can be contention for the front. For instance, when bringing up a halo on a morph, an inconveniently placed frame stack which is playing may bring a frame in front of a halo.

Visibility

Hiding an object or making it visible is another way that a presentation operation may substitute for a structural navigation. Again, Morphic provides the ability to hide or show any morph. (It is surprising that hiding of objects is not an operation commonly supported by spatial hypertext systems.)

IMPLEMENTATION SPECIFICS

The hierarchy of classes created for the Frame Stack Project is shown in Figure 3. The amount of code in these classes is so small that it is an almost infinitesimal fraction of the corpus of Squeak / Morphic. I offer this not as an apology, but rather as testimony to what can be accomplished by an individual cybertext author using the "open-air subclassing" approach on top of a rich generic object framework like Morphic. The Frame Stack interface is particular to my own artistic practice; other writers will have drastically different needs. It is unlikely that very many cybertext authors will find the Frame Stack Project code directly useful; I am offering it more as a kind of living example of what can be accomplished using this method.

```
(Array)
        FrameStackRectArray

(Form)
        FrameStackGlyphs

(PasteUpMorph)
        FrameStackCard

(RectangleMorph)
        FrameStack
        FrameStackFrame
        FrameStackRectangle

(SketchMorph)
        FrameStackSketch
                FrameStackScope
        FrameStackThumbnail

(TextMorph)
        FrameStackText
```

Figure 3

Class hierarchy created for the Frame Stack Project. Classes shown in parentheses are off-the-shelf classes provided with Squeak.

In the following section, some of the key classes implemented in this project will be described.

FrameStack, FrameStackFrame

FrameStack is the "signature class" from which this project takes its name. A frame stack is an object with a rectangular boundary and a collection of submorphs for which the frame stack acts as interface. The goal is to provide an intuitive interface by which transparent word objects can be overlaid in the same space — which would normally render them illegible — and allow individual objects to be read by a set of opaque "frames" that are controlled by mouseEnter hot-spots; each frame corresponds to one of the word objects. These frames are implemented by a class called FrameStackFrame. The submorphs for which the frame stack acts as an interface are not "specially" designated in any way; a frame stack identifies these as any submorph which is not a FrameStackFrame. Thus a new submorph can be added using any means supported by Morphic, without requiring any special code in FrameStack. I.e. "authoring" a frame stack is as simple as creating a new empty FrameStack (using the Squeak desktop "new morph" menu entry), turning on "accept drops" in the FrameStack, and then dropping morphs into it.

In the current implementation, a frame stack is "reformatted" for a change to its submorph population by an explicit reframe method. (Future versions should probably do a reframe automatically in response to various relevant events.) The reframe method, which was easily the most complex in the whole project, discards any existing FrameStackFrame submorphs and then recreates them, sizing them to their designated submorph; the boundary of the frame stack itself is also resized.

In addition to controlling whether a frame stack is "playing" or not (discussed above), another behavior implemented by FrameStack is a "freeze". Normally when a frame stack is told to stop playing, it will be in a "closed" state. (All frame stack frames are invisible, so that all the other submorphs are visible and appear overlaid.) If one of the submorphs needs to be edited, having the frame stack continue to play will interfere, but the "layer" with the given submorph needs to be "open" so the submorph is easily accessible for editing. Because the mouse is already used to "navigate" among the submorphs of a frame stack, the keyboard is used to register a freeze. When initially created, a frame stack does not accept keystrokes, but it can be told to do so. Once accepting keystrokes, when a frame stack is sent the 'f' key from the keyboard, it freezes in its current state. This allows the submorph for the layer showing to be edited *in place*. (Typically editing occurs using off-the-shelf behavior of Morphic; e.g. if the object in question is text, it may be edited using customary text editing mouse moves and keystrokes.) This is consistent with a deeply held philosophy of this project, that authoring vs. run-time behavior should be a state property of *individual objects*, not of "the system" or "environment" as a whole.

FrameStackRectangle

This class is used to implement grouping. An instance of this class is a transparent rectangular area with various submorphs; it is needed as a specific class mainly to allow grouping in such a way that the mouse events are properly passed through to any frame stack submorphs.

FrameStackCard

The actual cybertexts so far realized in this environment have used an "outer interface" extremely similar to the original card interface provided by HyperCard [2]. This interface assumes a non-scrolling fixed "portal" which does not move on screen; as the reader moves through the piece the content of this portal is changed. In the Frame Stack Project there is no formal concept of portal. Rather, its appearance is created by the cybertext author creating a set of frame stack cards which are all of the same size and position on the screen. This class implements a parent-child relationship among frame stack cards, using methods seekParent and acceptChild. When a frame stack card receives acceptChild, a button is created that will bring the child to the front when clicked; the button is a thumbnail image of the child. (At the child there is a method that will set the magnification scale for creating this thumbnail.) An "up-button" is created on the child, that when clicked will bring the parent to the front.

Consistent with another major philosophy of this project, a frame stack card may have no parent. (There is also a method of FrameStackCard called unparent, which will delete the parent relationship and render a card parentless.)

FrameStackCard is a subclass of an important Morphic class called PasteUpMorph, also known as a playfield. This class is the basic form of Morphic "canvas", and provides many facilities for graphical editing. The Squeak desktop (known as a "world") is in fact a playfield.

Fonts

Fonts are an extremely tricky issue in any discussion of cybertext authoring systems. It is customary among hypertext system designers to assume that fonts are *someone else's problem;* e.g. the native operating system windowing system is presumed to provide fonts, the user may have fonts of her own, etc. Scalable outline fonts, such as TrueType or Postscript Type 1 fonts, are a form of intellectual property subject to their own system of rights. A cybertext author wanting to control the exact appearance of the text is thus confronted with a difficult dilemma: embedding fonts in a cybertext may create unpleasant rights problems for distributing the cybertext. Technologies like Flash seem to allow distribution of cybertexts with embedded fonts in ways that have apparently avoided this problem, but at the cost of a heavy-weight distinction between the authoring environment and the run-time environment. An important goal of the Frame Stack Project was to be able to support creation of cybertexts with embedded fonts that the cybertext author can *edit*. As of the time this project began, the native font system of Squeak is bitmapped. The decision of whether to use bitmapped rather than antialiased fonts was one of the more aesthetically difficult decisions made during this project. In the end, a set of fonts was created based on outline fonts believed to be unencumbered; from these, screen renderings were imported into a Squeak font editor to create bitmapped fonts with a close aesthetic resemblance to the effect of antialiased fonts on screen. As the Squeak font system evolves, the fonts used will probably change.

FERAL STRUCTURE

There is a great deal of research involving integration of hypertext systems with a larger computing environment, particularly in the OHS community. Hypertext has certainly had a wider perspective than just "the confines" of hypertext applications for quite a long time. Still, it is most common for hypertext objects to be found inside hypertext systems. While the Squeak desktop is not the native operating system desktop — though it could become the native OS desktop; see [11] — it is certainly a "generic object desktop" in which the user could spend the entirety of her time and which is not especially devoted to hypertext. The desktop is the cyberspace equivalent of the open air. A desktop such as the Squeak World allows objects to be simply "loose" in the open air, much as a physical desktop allows physical objects to be loose on its surface, without being placed in a drawer. The appeal of such freedom is similar to the attractiveness of spatial hypertext itself. Among the features offered by feral structure are:

- Objects near at hand are presumably prioritized.
- A disposition of the object can be postponed.
- A persistent desktop allows work to be resumed in exactly the state it was left in a previous session.

It is particularly important to note that feral structure is ideally suited to collecting cybertextual scraps where the destination of the scrap is not known at the time it was collected. As mentioned above, there is a deep historical affinity for poets in particular to write by a method that in part involves accumulating materials in notebooks. Systems such as Flash, with their extremely heavy-weight distinction between authoring and runtime, raise profound difficulties for collecting cybertextual scraps. Figure 4 shows a screen dump of the actual live Squeak desktop for my current work in progress. Note there are several objects placed on the desktop wherever I found it convenient to work with them: some are frame stacks or frame stack rectangles, some are frame stack texts. Note the objects in the top left corner. These are iconified morphs. The ones marked "playfield" are frame stack cards which are more or less finished, but have not yet been integrated into any higher level of structure.

Long time users of (say) VKB may wonder why there is any difference between the concept of feral structure as articulated here and the VKB "root collection". After all, in VKB no one is obliged to make collections; one may place all of one's objects in the root collection. I.e. VKB allows a structure which is "flat". What is the difference between a flat structure and feral structure? Perhaps one could argue that this distinction is simple hair-splitting, but the major difference is that an application like VKB is not a generic object system, in which any kind of object (with any kind of behavior!) can be placed. A VKB collection can only contain the kinds of objects that have been specifically implemented in VKB. It is not "the open air", but rather a very special atmosphere in which only a severely limited variety of creatures can breathe. While it would be easy to imagine the Squeak desktop as the native OS desktop, this would not be possible with

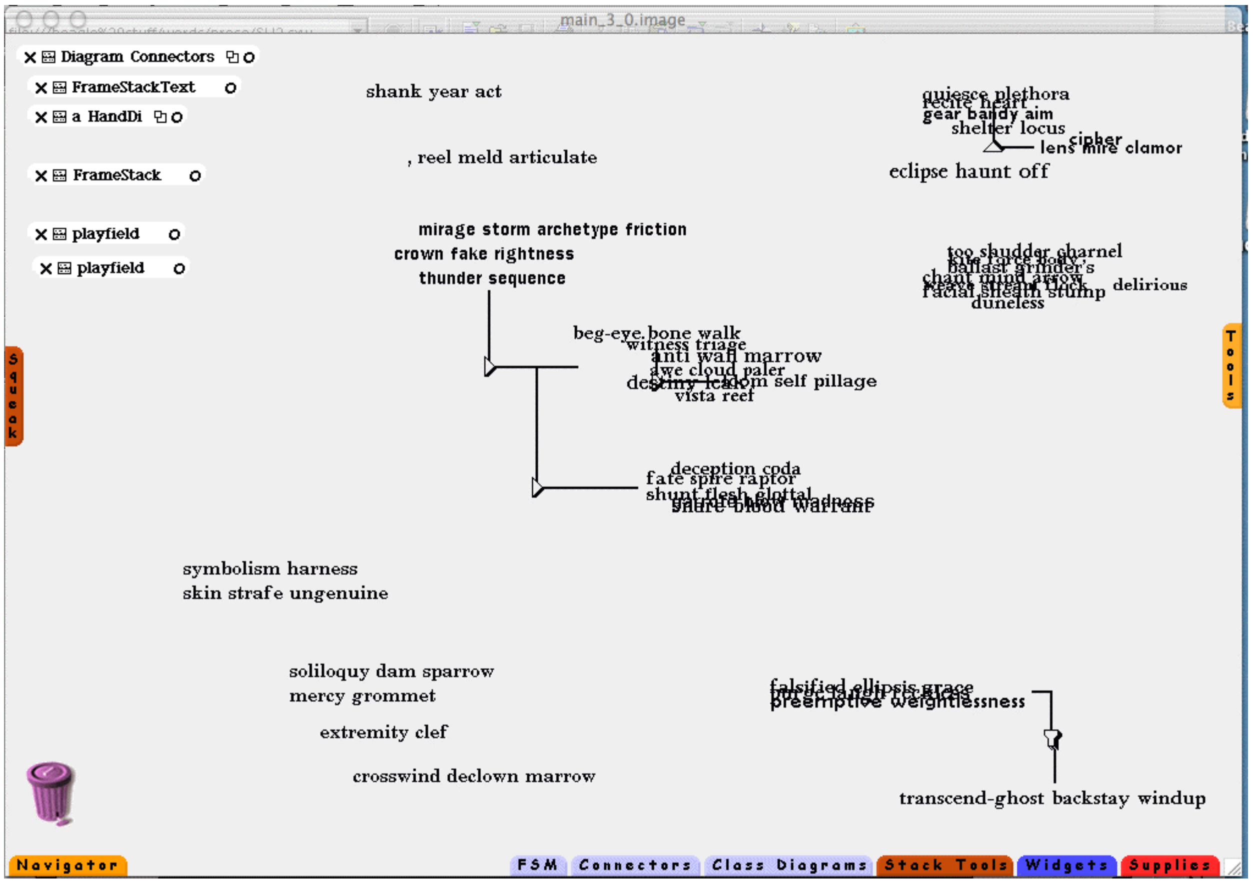

Figure 4

Screen dump of current work, showing the live Squeak desktop for a poem in progress. Many of these objects are not yet finished, and the ones that are finished have not yet been given a final “destination”. The objects are “feral” because they have not yet been captured into structure. (Though some have internal structure.)

VKB without a very significant amount of work.

Of course a necessary component of support for feral structure must be the ability to “capture the feral animals”: parentless objects which are simply loose on the desktop must be easy to move into a more defined structural place, once that place has been determined. The pickup mechanism of Morphic makes this simple enough that it does not intrude on the “aesthetic stream” of making poetry.

Is the Native OS Desktop a Spatial Hypertext?

It can be argued that a desktop is not truly the computer equivalent of the open air unless that desktop is the ultimate “native” operating system desktop. That raises an interesting question: should we consider native OS desktops as “already” spatial hypertext systems? Many users certainly place a great deal of information on their desktops, and some users become completely lost if a desktop icon goes missing: they navigate not through the file system, but spatially on the desktop.

There are no commercial operating systems that have a desktop with the object power of anything like even a fragment of Morphic. Perhaps we can look forward to this in the future.

USABILITY

The concept of usability takes on an odd cast in the context of a personal authoring system. How should the author of a personal authoring system carry out an unbiased usability study? This is clearly impossible.

It will have to suffice for me to simply offer anecdotal evidence. Based on a few months of creating finished works in the Frame Stack Project, I can say that total elapsed time to complete such a work is cut by a factor of about 3 from my previous methods. More importantly, (and even more anecdotally, alas) the feeling of composing in this environment is substantially different than it was using tools like HyperCard. When writing in the Frame Stack Project, the word object is a true object, and can easily become a finished “interactive scrap” during a single session. By contrast, using previous methods the objecthood of what appears on the screen as a word object is a mere facade; inside the work there is no real object, and it might have taken weeks after all aesthetic decisions were made before there was any interactivity present at all. Writing by such methods relies on a completely non-interactive document which Bootz [1] calls the *texte auteur*, which provides a kind of implementation specification for how a cybertext is to be assembled. While opinions can differ concerning what the term "interactive writing" might mean, it is hard to call a writing process interactive if interactivity appears only at the end of a long process, taking weeks or months in which there are no interactive objects present.

The intensely granular individual object nature of the distinction between authoring and run-time achieved in the Frame Stack Project simply gives a different feeling to the act of writing. It allows interactive writing in the true sense of the word.

FUTURE WORK

In addition to the frame stacks, my work also includes a formal structuring via a diagram notation which is fundamentally relational. The result is structures not unlike those achieved in Aquanet [7]. Currently the Frame Stack Project does not support relations by means of any explicit classes. The relations are simply drawn, graphically, using an off-the-shelf Squeak project called Connectors [4]. There needs to be an explicit interface so that relations in the Frame Stack Project are real objects.

Currently there is no method of exporting the text in all FrameStackText objects of a project. (This should be quite simple to implement using available code for Squeak.)

As of this writing, I've completed three finished poetic works in the Frame Stack Project. The word 'finished' is used here in aesthetic terms; there are still some open issues regarding how such works are to be published. Also not yet investigated are issues of how to "harden" Morphic for publication purposes..

ACKNOWLEDGMENTS

This work would not have been possible without the immense contributions of many other people, specifically: Allan Kay et al for Squeak, John Maloney for bringing Morphic to Squeak, Ned Konz for Connectors, Boris Gaertner for his font editor, and of course Cathy Marshall, Frank Shipman, and many others for the concept of spatial hypertext in the first place.

REFERENCES

1. Bootz, Philippe. "Le point de vue fonctionnel: point de vue tragique et programme pilote". *alire 10 / ~~DOC(K)S~~*, MOTS-VOIR, Villeneuve d'Ascq, 1997, pp. 28-47.

2. Goodman, Danny, *The Complete HyperCard Handbook*, Bantam Books, New York, 1987.

3. Ingalls, Dan, Kaehler, Ted, Maloney, John, Wallace, Scott, and Kay, Alan. "Back to the Future: the Story of Squeak, a practical Smalltalk written in itself", *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, ACM, New York, 1997, pp. 318-326.

4. Konz, Ned, Connectors, http://bike-nomad.com/squeak/index.html

5. Krasner, G. E., and Pope, S. T., "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80", *Journal of Object Oriented Programming*, August/September, 1988, 26-49.

6. Macromedia, Inc., *Flash.* San Francisco, http://www.macromedia.com/support/flash/documentation.html., 1995.

7. Marshall, Catherine C., Halasz, Frank G., Rogers, Russell A. and Janssen, William C. Jr., "Aquanet: a hypertext tool to hold your knowledge in place", *Proceedings of Hypertext '91*, ACM, New York, 1991, pp. 261-275.

8. Marshall, Catherine C., Shipman, Frank M. III, and Coombs, James H., "VIKI: Spatial Hypertext Supporting Emergent Structure", *European Conference on Hypermedia Technology 1994 Proceedings*, ACM, New York, 1994, pp. 13-23.

9. Rosenberg, Jim, "User Interface Behaviors for Spatially Overlaid Implicit Structures" First Workshop on Spatial Hypertext, Århus, 2001, http://www.csdl.tamu.edu/~shipman/SpatialHypertext/SH1/rosenberg.pdf.

10. Shipman, Frank M. III, Hsieh, Haowei, Maloor, Preetam, and Moore, J. Michael, "The Visual Knowledge Builder: A Second Generation Spatial Hypertext", *Hypertext '01: Proceedings of the 2001 ACM Conference on Hypertext*, ACM, New York, 2001, pp. 113-122.

11. Smith, David A., Raab, Andreas, Reed, David, and Kay, Alan, *Croquet The User Manual*, Viewpoints Research Institute, Glendale, 2002, http://glab.cs.uni-magdeburg.de/~croquet/downloads/Croquet0.1.pdf.

12. Smith, Randall B., Maloney, John, and Ungar, David, "The Self-4.0 User Interface: Manifesting a System-wide Vision of Concreteness, Uniformity, and

Flexibility", *Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, ACM, New York, 1995, pp.: 47 - 60