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

Jim Rosenberg
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.

1. Introduction

This paper [1] 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 (Rosenberg 2001). For the sake of completeness, the frame stack concept will be reviewed briefly here. A frame stack is a user interface for overlaying word objects on top of one another, while still allowing them to be read legibly. In its starting, or "closed" state, all of the word objects simply appear one on top of the other, with no particular graphical indication of any interface components. The bounds of the frame stack constitute an active mouse-over hot-spot. When the mouse enters the bounding box of a closed frame stack, a stack of frames is displayed, one for each element of the overlay. These frames serve as mouse-over hot-spots to allow navigation of each member of the overlay; note that the frame stack explicitly embeds the peer structure of the overlay. An open frame stack is illustrated in Figure 1. (The same frame stack is shown closed in Figure 2, surrounded by a "morphic halo" (see below). The frame stack concept works well with nested collections, since it allows a frame stack to be opened inside an outer frame stack. One drawback to the frame stack concept is that it is quite expensive in terms of screen real estate. Assuming all frames in one stack are the same size, this must be large enough to accommodate the largest element — with sufficient room for a bit of margin.
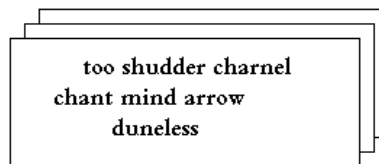

Figure 1: An "open" frame stack

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 at the time it is first written down what the ultimate destination of the scrap will be. A scrap is thus an inherently parentless object.

The generic object framework within which the Frame Stack Project is realized is called Morphic (Smith et al, 1995) (discussed below), and is provided off the shelf in the programming environment Squeak (Ingalls et al, 1997). 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.

2. 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-click 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 2). 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.
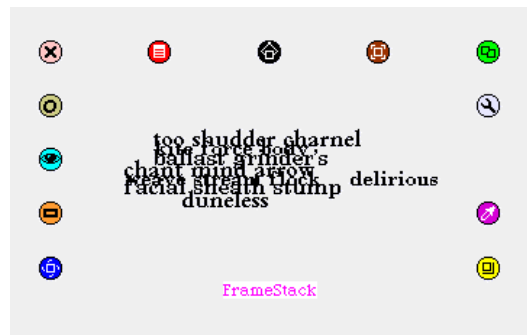
Figure 2: A closed frame stack surrounded by a "morphic halo"

2.1 Pickup

A morph can be *picked up*, by dragging the top middle icon from the halo. This means not only moving its position; when the mouse is released at the end of the drag, if the mouse cursor has entered the bounding box of another morph, the morph being picked up is dropped *into* the target morph, becoming a *submorph* of the target. The receiving morph may have been set to reject drops; in this case the picked-up morph will "fly back" to its starting position. It is interesting to contrast this concept with the way that aggregation works in familiar spatial hypertext systems such as VIKI or VKB. 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, transforming 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 contain 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 a "state property" — i.e. it's determined by the current state of an instance variable, which might change. 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. Thus the state of rejecting drops may be considered an interface property, since the rejection behavior is enforced by the interface and can be overridden. 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.

2.2 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 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 (Krasner and Pope, 1988), 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 available 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.

2.3 Menus

A morph inherits a "standard menu" listing generic Morphic behaviors, but the programmer creating a Morph subclass can easily customize this. This provides an easy method to attach an interface to one's own classes. Figure 3 shows the "morph menu" for an instance of 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.
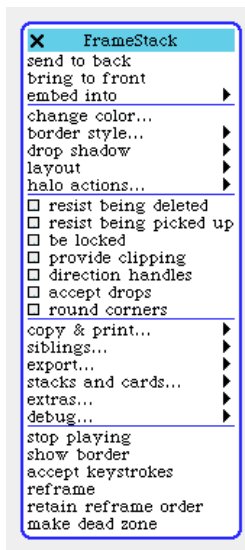
Figure 3: The "Morph Menu" for a frame stack

It is also worth commenting in more detail on the menu choice seen in Figure 3 "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 enabling the run-time behavior of the frame stack object. The point here is that unlike environments like Flash, 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 property 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.

3. 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.

3.1 "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 one 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 that each card is opaque to any cards behind it, this interface can be implemented in Morphic by having each card be a rectangular morph of the same size and position 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. (This same effect can be seen on some web pages, where simply moving the mouse — e.g. from a javascript button to the menu bar — can inadvertently trigger a rollover.) Mitigating this contention can require forgoing behaviors. A frame stack which is occluding some interface object which one wishes to be at the front can simply be told to stop playing; since the interface objects are likely to only be present during authoring of some nearby object, the frame stack can be told to resume playing later, with only minor inconvenience. In other cases resolving contention for the front can be more painful. The Frame Stack Project utilizes a package for Squeak called Connectors (Konz) which allows creation of connected diagrams. In this package a "live connector" also makes claims on the front. As of this writing, to resolve this conflict has meant restricting Connectors objects to their own space, forgoing "live connection".

3.2 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.

4. Implementation Specifics

The hierarchy of classes created for the Frame Stack Project is shown in Figure 4. 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

(Object)

       FrameStackExport

(PasteUpMorph)

       FrameStackCard

       FrameStackRelation

(Rectangle)

       FrameStackAttachment

(RectangleMorph)

       FrameStack

       FrameStackCardButton

       FrameStackFrame

       FrameStackNode

       FrameStackProjectButton

           FrameStackQuitButton

       FrameStackRectangle

(SketchMorph)

       FrameStackSketch

       FrameStackScope

       FrameStackThumbnail

(TextMorph)

       FrameStackText

Figure 4: 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.

4.1 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 — called "exhibits" — for which the frame stack acts as an interface are not "specially" designated in any way; a frame stack identifies its exhibits as any submorph which is not a FrameStackFrame. Thus a new submorph can be added using any means supported by Morphic (such as Pickup, described above) 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. Any graphical object can be an exhibit, as long as it is a morph.

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 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 of the exhibits are visible and appear overlaid.) If one of the exhibits needs to be edited, having the frame stack continue to play will interfere, but the "layer" with the given exhibit needs to be "open" so the submorph is easily accessible for editing. Because the mouse is already used to "navigate" among the exhibits 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 — as implemented by the off-the-shelf Morphic class TextMorph.) This is consistent with a deeply held philosophy of this project, that the distinction between authoring vs. run-time behavior should be a state property of individual objects, not of "the system" or "environment" as a whole.

4.2 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. Note that this type of grouping is entirely "author-convenience" grouping: the group allows its submorphs to be moved as a unit, including all their behaviors, but the group is not visible to the reader. While such a group is arguably "a structure", it may be described as a purely *private* structure, much like the corner braces for a painting's stretcher which are invisible to the viewer. Where groups are nested, there may be many different ways such a group may be constructed; e.g. elements a, b, c, d may be grouped as {a, b, c, d} or {{a, b}, {c, d}} or {{a, b, c}, d}; since group substructure is not visible at reading time, the choice among these different substructures *may not matter* as long as ultimately all the right elements are "together". Note that this position on structure is in total contrast with the point of view of Structural Computing (Nürnberg et al, 1997) where structure is considered primary and structures are first-class objects.

Issues pertaining to grouping in the context of spatial hypertext are discussed in more detail in (Rosenberg, 2004).

4.3 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 (Goodman, 1997). 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 all have 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.

4.4 FrameStackRelation, FrameStackNode

Relations as a built-in structural concept were first seen in hypertext systems in Aquanet (Marshall et al. 1991) but have seldom been implemented since. In addition to the kind of direct juxtapositions realized as frame stacks, my artistic work for many years has utilized a diagram notation which has an explicitly relational model. Fragments of this notation can be seen in Figure 5. In Aquanet, a relation is primarily a structural concept, and secondarily a relation may have a graphic content by which the relation is rendered visually. In the Frame Stack Project this relationship is reversed: the relation is primarily a graphic object, that secondarily has structural properties. The objects that are "tied together" by the relation are pointed to by FrameStackNode objects as an attachment of the frame stack node. FrameStackNode has an attachToNearest method which attempts to locate the nearest suitable target; FrameStackRelation has an attachAll method that executes attachToNearest on all nodes. This is the closest thing there is in the Frame Stack Project to a spatial parser.

At this writing, the frame stack relation concept may be described as "weak structure". The original implementation of the Frame Stack Project (in which several artistic works were completed) did not have FrameStackRelation implemented. Instead, in these works, relational structure was denoted by entirely visual means. (These works were later revised so that all of the PasteUpMorphs constituting "visual relations" were replaced by actual FrameStackRelations.) Even now, the structural concept is still weak in that (1) visual structure and node / attachment structure might be inconsistent; (2) the node / attachment structure has no consequence for the reader. Errors in the node-attachment structure do have consequences for the ability to export and to "walk" the structure (see below).

4.5 FrameStackExport

FrameStackExport's main function is to be a "state holder" to support export of information about a project to an external file format. Currently the only format implemented is text; although all structural information and text content is exported, spatial information, relation geometry, and text metadata (e.g. font) are not yet exported. Because the relational structure may contain loops, the export algorithm, though not difficult, requires more than one pass. This class includes a "descent" method whose argument is a Smalltalk block, allowing it to serve as a kind of open framework for code that needs to operate on all objects in a project. "Ad hoc" use of this method to walk all objects in a project proved very useful lately getting around an obstacle in porting the Frame

Stack Project from Squeak 3.6 to Squeak 3.8. In general, such a method is enormously valuable for cybertext preservation, and provides a kind of generalized "extensibility hook". It is unfortunately true that our record in spatial hypertext of providing extensibility is not good.

4.6 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. The FrameStackGlyphs class was created to allow importing an entire character set at once. As the Squeak font system evolves, the fonts used will probably change.

5. Feral Structure

[2]. 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 (Smith et al, 2002) — 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 run-time, raise profound difficulties for collecting cybertextual scraps. Figure 5 shows a screen dump of the actual live Squeak desktop for my work in progress as of July, 2003. 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.
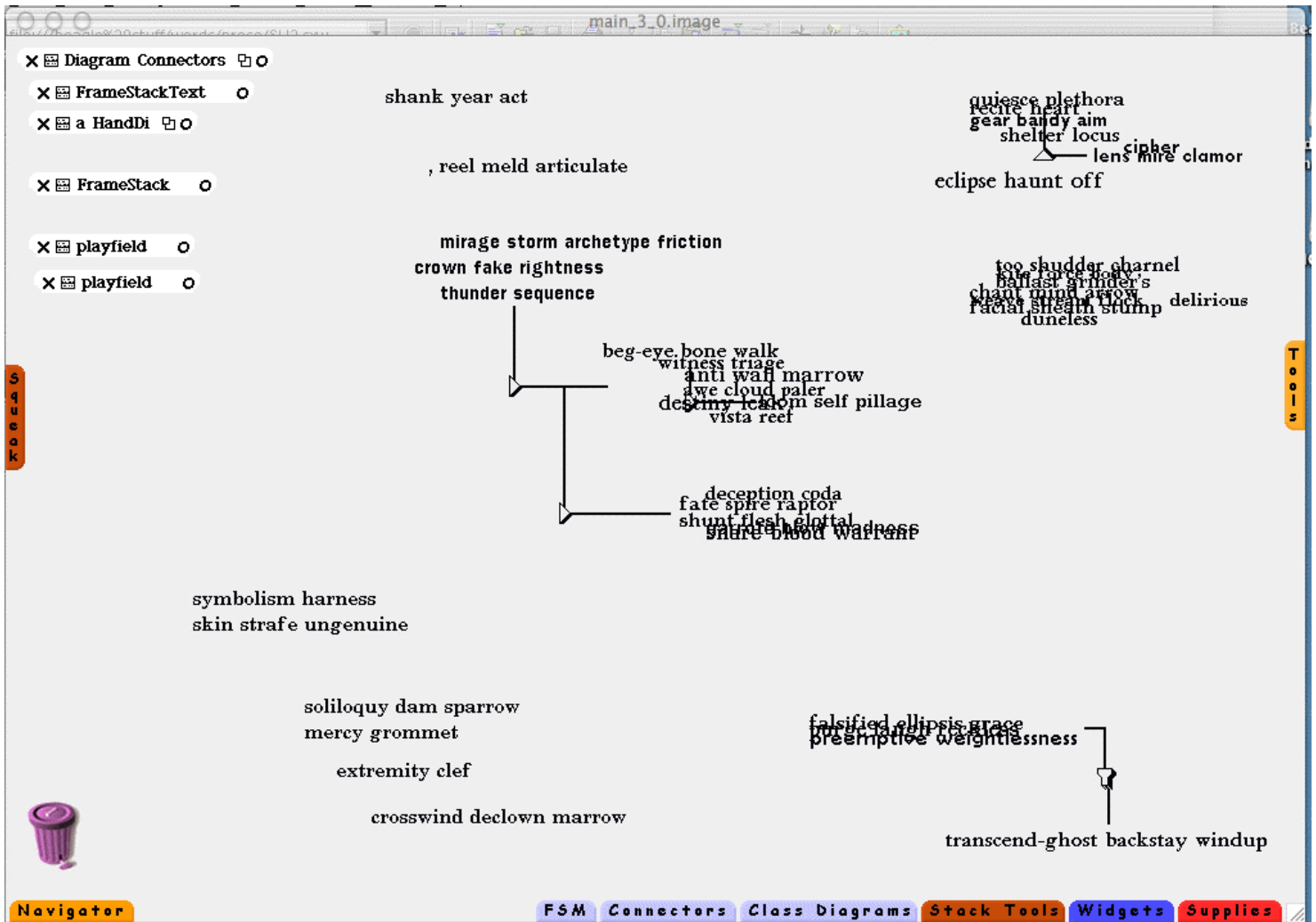
Figure 5: Screen shot of live work in progress, July 2003, showing objects "loose on the desktop"

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 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.

5.1 Structure Objects vs. Activity Results

In speaking about feral structure, it is useful to consider what the objects are "feral from". Adherents of structural computing tend to see structure as primary; from this vantage point what one sees in spatial hypertext is often described as "spatial structure". While this approach is interesting and can give some fruitful results, it also misses the essence of spatial hypertext. Consider the word objects in the top right of Figure 5. Why is this cluster in the top right? Is its being in the top right really a case of "spatial structure"? In this case the author is available to testify. I often tend to put a cluster in the top right of my workspace when (1) it is available to use as material in a larger structure; (2) I don't want to forget about it; (3) I *have moved it out of the way* to clear space nearer the center of the workspace for the objects I am actively composing at the moment. I.e. the location of the cluster in the top right is the result of an *activity*. Perhaps its spatial relationships in its out of the way location can be fruitfully analyzed in structural terms, but what mattered at the time the object was moved was: getting it out of the way. Any concern about structure was at best secondary. Structure is important, to be sure; but even more important is the workspace as an arena for word object *activities*. Certainly in this arena structure happens. But also structure "unhappens". Word object activities occur that may or may not give structural results.

What open-air word objects may in fact be "feral from" is exactly: *structure!* A generic desktop which allows interactive word objects to be loose on the desktop allows such objects to simply *be present*. This allows the author to look at them and say: "Hmm. I don't know ... ". The structural relationship of such an object to those already present in the space may be far less important than the simple fact of its arrival.

5.2 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.

6. Reflections on usage

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 approximately three years 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 ([Bootz 1997](#)) 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. The act of writing involves countless acts of reading by an author of pieces of the work in progress; the writing process intimately involves intense feedback from reading material already written. For writing interactive works, having fragments of objects already composed "loose" and at hand greatly facilitates this feedback. Many effects of reading can easily *persist* into the writing process; such persistence is inhibited by the need to transition back and forth from separate environments for authoring and run-time. (See ([Rosenberg 2004](#)) for further discussion of persistence effects.)

7. Future work

Because the Frame Stack Project takes place on the substrate of an open spatial object framework, the way is wide open to add new kinds of objects. My current poetic work includes some new object types for spatialized linear word objects. There is much more work to be done in export functionality; ideally enough should be exportable to allow complete reconstruction of an artistic work. Some form of XML as an export format would clearly be highly desirable. Another useful export format would be Smalltalk code; using such an export, a project could be reconstructed by the simple expedient of filing it in.

The FrameStackRelation concept currently has no built in support for "off-card" nodes. In fact the relation model so far implemented has no associated behavior whatsoever. It should be easy to associate any objects in a project into a relation; there is much work to be done in this project to support "distant" relations.

8. Acknowledgements

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.

Notes

1. This paper is a revised version of ([Rosenberg 2003](#)).

2. Walker ([Walker 2005](#)) presents a somewhat different concept of "feral"; in her concept objects are feral in the Internet at large. As she cites in her paper, the discussion here — as originally found in in ([Rosenberg 2003](#)) — predates her discussion.

References

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.

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

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.

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

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.

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

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.

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

Nürnberg, Peter J., Legget, John J., and Schneider, Erich R.. "As We Should Have Thought", *Hypertext 97*, ACM, New York, 1997, pp. 96-101.

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.

Rosenberg, Jim. "Hypertext in the Open Air: A Systemless Approach to Spatial Hypertext", Third Workshop on Spatial Hypertext, Nottingham, 2003, http://www.csdl.tamu.edu/~shipman/SpatialHypertext/SH3/rosenberg.pdf.

Rosenberg, Jim. "Reflections on Spatial Writing in Place", Fourth Workshop on Spatial Hypertext, Santa Cruz, 2004, http://www.csdl.tamu.edu/~shipman/SpatialHypertext/SH4/rosenberg.pdf.

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.

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.

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.

Walker, Jill. "Feral Hypertext: When Hypertext Literature Escapes Control", *Proceedings of the HT 2005 - Sixteenth ACM Conference on Hypertext and Hypermedia*, ACM, New York, 2005, pp. 46 - 53.