This section introduces the graphical environment developed to create, manipulate, script, debug, and visualize my most recent artworks. Chronologically, it comes between The Music Creatures *and the dance works. It is presented in contrast to the predominant tools that digital artists use today.*

Chapter 8 — *Fluid*, an environment for digital art-making

In earlier sections of this document we have described an "authorship stance" to the critique and construction of artificial intelligent agents — a stance where the ease of construction of the agent is critical, a stance where the ability to conceptualize the creation of an artwork, while creating it, inside the agent framework, is vital. It is not enough to demonstrate the academic place and power of learning algorithms, action-selection techniques, and motor-system representations: these innovations must be framed in a relationship with a creative practice.

I have indicated that the authorability afforded by agent systems or AI systems in general receives little direct attention, but we have proceeded to construct a series of technologies that enable the assembly of the kinds of complex structures that AI agents seem to demand. And yet at the same time as the context-tree, the Diagram framework, the generic radial-basis channel and the use of historical databases of various kinds enrich the vocabulary for expressing agents, they in turn make their own complex potentials. In this section the authorship perspective on agent systems meets a more sustained critique of the authorship perspective afforded by "mapping" and its kin as we construct the toolset used for navigating our agent-based practices and the agent toolkit.

Further, in the introduction, *page 12*, we have set up the concepts surrounding the blanket term "mapping" as an opposing model to pitch an agent perspective, as both metaphor and implementation, against. I have already noted that this term is a pervasive one, but nowhere is its pervasiveness more manifest than in the tools that digital artists use.

1. A critique of existing environments

Some of my works, in particular the earlier agents — *Music Creatures* and *Loops* — exploit and motivate these AI techniques and metaphors, but they remain artworks that are created through the writing and testing and tuning of tens of thousands of lines of code. Some lines of course are part of the toolkit and they have been written and tested before; some of the new lines are refactored to become part of the next toolkit. However, a great many of them were specific to the particular installation. and the making of the art-work is the making and remaking of those lines of code until they are right.

Not many digital artworks are made like this today. Rather there is a burgeoning community forming around a growing set of software tools for making digital art, in particular interactive digital art without "recourse" to text-level programming. And since these tools are well entrenched in the community and form the basis of the courses and programs in schools, it is likely that very few digital artworks will be made like this tomorrow either.

With few exceptions, these popular graphical environments (they are controversially sometimes referred to as visual programming languages) are based on a common small reservoir of ideas: a few visual metaphors, a few structuring concepts. They each possess a surprisingly similar flavor and set of capabilities. So similar, in fact, that one might suspect that we are suffering from a digital art tools monoculture.

My recent works — *how long...* and *22* — were both created over a long, sustained period, their premieres scheduled two and a half years in advance, each given five, week-long workshops spread evenly out over that time, with a relatively stable set of hardware. Further, these works were been made in collaboration with other artists, both visual and in other media, some of whom programmed while others did not. This rare time-frame of sustained collaboration allowed a long and profitable look at the tools needed to *survive* these intensive workshop scenarios, effectively both allowing and necessitating a move away from a technique based completely on writing and testing and tuning those thousands of lines of code. Under ideal conditions one might argue that such a sincere look at not just "what should be done" but the inseparable "how it should be achieved" is part of the responsibility that artists have to their collaborators upon agreeing to work together. To construct and own one's tools as far as possible, marks nothing less than an openness to the potential of the collaboration. The resulting goal was to to find a different reservoir of ideas that could be drawn upon for the creation of a fresh programming environment for the making of interactive digital artworks — an environment for which I would be responsible for.

This section will begin with a brief survey of the common principles behind the graphical environments. This will not be a critique of their implementation details, their stability, their processing power, but rather of what it is they set out to do and the affordances they offer to artists who come to them.

I shall articulate three main weaknesses of these environments, before moving onto a more sustained, contrasting description of the graphical environment that emerged out of the needs and pressures of authoring the agents for *how long...* and *22* and the end of *The Music Creatures*. My discussion will summarily ignore the recent interest pure programming environments such as the notable Processing, Drawing By Numbers and Alice applications. The former compromises slightly Java's support for maintainably complex projects in exchange for a significant and admirable gain in pedagogical impact, but remains thoroughly and deliberately eclipsed by more fully fledged programming environments. From a community perspective Processing is extremely vibrant and interesting, but as a development platform it is only half-way toward something else. The other text base programming environments aim, in different ways, for even greater pedagogical impact and an even greater simplicity constraint.

Began by **Ben Fry** and **Casey Reas**: http://www.processing.org . Its tactical simplicity when compared to other Java environments was declared in a personal discussion between myself and Reas.

Alice: http://www.alice.org/

Drawing by Numbers is by John Maeda, **J. Maeda**, *Drawing By Numbers*, MIT Press, 2001.

The mainstream tools have been criticized before; however my purpose here is a little more focused. Nor is this discussion the place for a fruitless competition between the agent-based and extant tools. Rather, we are looking for environments that allow us not just to create complexity but to interact, navigate, manage and collaborate around the kind of complexities that the agent-based approach tends to create. While they may be sold (and even taught) on the basis of how rapidly they create potential what I will ask of these tools here is how they interact with, navigate and manage the potential of interactive media.

The most productive critical dialogue around Max— conducted by a number of major figures in computer must — happened a considerable time ago (to apparently little effect): It is collected in

P. Desain and **H. Honing**. *Letter to the editor: the mins of Max*. Computer Music Journal, 17(2). 1993.

Information about Max/MSP/Jitter can be found at http://www.cycling74.com

Other environments in this tradition:

Meso's *vvvv* —
http://vvvv.meso.net/

Infomus Lab's *eyes-web* —
http://www.infomus.dist.unige.it/eywindex.html

Trokia Ranch Dance Company's *Isadora* —
http://www.troikatronix.com/isadora.html

Miller Pukette's *pd* —
http://www-crca.ucsd.edu/~msp/software.html

IRCAM's *j-max* —
http://freesoftware.ircam.fr/

The graphical suite with longest pedigree is Max/Msp/Jitter — with Max being the name of the core and Msp and Jitter being progressively more recent extensions that allow the manipulation of sound and video respectively. More than anything else Max is the canonical data-flow programming environment for interactive digital art. The central metaphor is that the flow of data between processing modules will be represented as a visual circuit — this is a digital implementation of the wires of an analogue synthesizer. The computational strength of the environment is then measured solely in the number of available modules and perhaps the number of data-types that these wires can carry.

Circuits can be hidden inside custom modules and while a few modules present custom views and interface elements onto their inner workings, most, including embedded circuits, retain a rather generic appearance — a label and input and output terminals. This itself is not a particularly problematic design decision — an attempt perhaps to maintain a rather clean and minimal visual appearance to a complex circuit.

But visual programming is an idea that seems always to be sliced in two, and Max partitions the visual and the programming at a very particular place. What is visual is precisely that which is not programming and what is programming is, I argue, not made especially graphic. The actual layout, appearance, size and visual relationships between these modules are meaningless. This has the stated benefit that users are relatively free to reorganize the visual appearance of the circuit to create their own "interface" to the patch. In practice this flexibility is greatly curtained by the circuit's metaphorical use of wires which do act to constrain layout and worse: the primitive interface possibilities of the completely static layout of a circuit. Is any other complex software product content to display an interface that does not change structurally? Rather, the static panel of knobs and switches is again borrowed from the analogue synthesizer. But if one argues that Max is neither a interface language nor a programming language, it

There is one exception to the meaninglessness of the visual layout of a Max "patch" or circuit — that the top to bottom, left to right ordering of elements breaks ties in deciding the execution ordering of modules, but it is generally believed that if a patch depends on this subtle execution ordering the patch ought to be redesigned.

remains to be seen if there is a better way of slicing the problems presented by "visual programming".

Equally ambiguous but more important is Max's very assumption that a depiction of the flow of data through modules that process the data is a particularly good way of *capturing* what a program does, that the manipulation of the flow of data through modules is a particularly good way to *change* what a program does and that thinking about the flow of data is a good way to *think* about what programs do and should do.

It's hard to find any persuasive science either way on these questions — few care about the speed with which artists can assemble their programs and even fewer would try to measure the quality of the decision making under such constraints — although there are some researchers who measure the behavior of programmers in similar environments (the popular LabVIEW environment which is targeted at engineers, but has been used for interactive artworks). In any case, the literature is utterly inconclusive about the merits of data-flow versus opposite paradigms, most notably "control-flow", where visual elements elements represent the looping and gating constructs of imperative programming languages rather than the inputs and outputs of procedure calls. There are shades of this alternative presentation buried inside Macromedia's Director and applications that date from its era. It's telling that environments based on this depiction have been more popular in two areas: scripting languages and programming pedagogy than they have in interactive art per se. There seems to be something of relevance to the "temporal arts" that the pure data-flow path misses.

A survey of our incomplete knowledge concerning the efficacy of visual programming environments can be found in: **A. F. Blackwell, K. N. Whitley, J. Good, M. Petre,** *Cognitive Factors in Programming with Diagrams.* Artificial Intelligence Review 15: 95-114, 2001.

For a use of LabVIEW in interactive music: **T. Marrin-Nakra,** *Inside the Conductors Jacket: Analysis, Intepretation, and Musical Synthesis of Expressive Gesture.* PhD Thesis. MIT, 2000.

Macromedia — http://www.macromedia.com/

For control flow based visual "programming" take, for example, Apple's wrapper around the text based AppleScript — Automator: http://www.apple.com/automator.

The visual interface for the Alice environment is also control / object first: **S. Cooper, W. Dann, R. Pausch** *Teaching Objects-first in Introductory Computer Science*, Proceedings of SIGCSE 2003.

It is perhaps for this reason that PD, the next generation of Max-like environments, possess a nascent "data-type" system.

Data-*flow* also trumps data-*type* in these environments. Max's wires can move numbers, sound, video around in addition to nested lists of numbers and strings — in theory, a circuit can talk about any data "structure" that, say, LISP can. Yet at the same time the use and inspection of these non-uniform data-structures are utterly un-visual and un-composable. Nowhere is this more apparent than the handling of geometric scene data, which necessarily are complex hierarchical linked systems. For all of my earlier discussion of the controllability of geometry versus the blendablility of video texture, geometry in these applications — with its messy, heterogeneous, hierarchical, typed, non-flowing data-structures — is less controllable (and, of course much less blendable) than video. Geometry in these applications is fixed and solid, a container for texture, it is something that is imported and displayed rather than synthesized. This lack of interest either in variable data-structures or variable control-structures is clearly antithetical to the needs of my work — much of the technical contributions of this thesis has been given over to the task of making complex systems that change structurally while running — and I believe that a toolset and a methodology that draws one towards such "static" complexity actually draws one away from the potentials of interactivity — be this between artist and tool, dancer and stage or audience and screen.

That Max, and its progeny (including *PD*, a re-implementation by Max's main original author Miller Pluckett with different license restrictions, operating systems and, of course, modules; and *vvvv*, a re-implemention of the same ideas with, at the time, a different operating system and a greater emphasis on video), should focus on illustrating data-flow rather than control should come as no surprise. And we can use this as circumstantial evidence in the absence of any applicable visual programming language science. The Max module is the most succinct "visualization" of the mapping metaphor that one could imagine, short of our earlier "function" image. As far as it is visually concerned in the language of computer science the module is no more than a function call. Indeed if there

is a common computer science reference to data-flow programming environments it is not "object orientation" as has been claimed but rather a simple side-effect free functional programing language. Of course, Max cannot extend this principle too far, and ultimately compromises its functional "purity" — hidden, un-visualized side effects abound.

It need not be this way. Max's modules nest but never intersect; they are not views onto a complex system, but are, rather, the complex system itself. My point of departure is a slightly different place, for I require tools for manipulating the agent-toolkit, offering, that is, windows into systems rather than the material of the systems themselves.

In the language of user-interface design, however, this is not a matter subject to taste. Rather Max conflates the *model* (the data, here both the modules and the wires) with the *view* (the way of manipulating the data, here both the box and the lines) with the *controller* (the glue that binds the model to the view). Such a conflation is considered unforgivable by many a human user interface programmer. It couples the model so tightly to the view such that no other view can be offered onto the model. This visual monoculture is our first main criticism of Max and can be levied regardless of what one thinks of the power of the contents of its boxes and wires.

For a detailed definition of the Model-View-Controller pattern: **F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal,** *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.* John Wiley & Sons, 1996.

Forthe visual-programming literature does have consensus on one topic — the vast number of different visual metaphors available to choose from. There are hundreds of visual programming languages. Max offers one metaphor, but more critically, enforces this single view onto the "program". Indeed, its view onto the program is, as far as it is concerned, the program itself.

See, for a range, the two volume review of the field up to 1990: **E.P. Glinert,** *Visual Programming Environments: Applications and Issues & Paradigms and Systems.* IEEE Computer Society Press, 1990.

One can extend Max by adding a module type — either through a external, compiled textual programming language, or though the nesting mechanism —

but one cannot add a new way of looking at the "program" that Max has helped assemble either from outside or, critically, inside Max. This lack of self-reflexivity is the second of our main criticisms.

Of course, the exact same charge can be levied against a textual programming language — few source codes are open or reflexive in this sense. Although we'll note in passing a recent interest in doing just this — Xml based programming languages such as *Water*, the Inversion of Control Xml configuration files of several container systems and of course this less than recent aspect of Lisp, seek to blur program and data by programming with a structure designed for data. The goal here is to allow programs to view and remake programs, in much the same way as we asked previously if Max should not allow modules to make, move and delete modules, even if just for the sake of having dynamic interfaces. Some have gone so far as to predict the slow death of "single-text" programming languages as they become inherently multi-perspective. If textual programming is becoming self-reflective and in an odd way "multi-media" shouldn't there be a panoply of domain-specific, multi-media programming environments leading the charge?

Water, C.Fry and M. Plusch — http://www.waterlanguage.org/

This trend comes from plotting a line through the interest in extensible syntaxes for example, the Fortress languge — http://research.sun.com/projects/plrg/fortress0618.pdf

and **G. L. Steele, Jr.**, *Growing a Language*, Journal of Higher-Order and Symbolic Computation 12(3) 1999.

And trends towards direct manipulation of abstract syntax trees, including James Gosling's Jackpot project: http://today.java.net/jag/page15.html

Isadora —
http://www.troikatronix.com

If Max has its technical roots in the analogue synthesizer and its conceptual roots in mapping, few environments can be seen to widen these bases a little. *Isadora* is an interesting environment for the purposes of this thesis because of its development in a interactive dance context — it is the work of the Troika Ranch Dance company, artist and engineer Mark Coniglio. Its more accessible revisitation of the visual design of Max is noble, but not an issue for this discussion any more than its processing speed or range of modules.

More interesting are the two concepts that Isadora, depending on your view, either adds to Max or pulls out of Max and names: that of the specific recoverable graph configuration or "scene", and that of a separate control surface onto a circuit or "control panel". The "scenes" are the most interesting innovation — they offer specific support for a clumsily created control structure implemented in a master "graph" in Max that switches between activating various subgraphs.

An Isadora document can have any number of Scenes, each of which is a collection of actors (modules) that manipulates one or more streams of digital media. Isadora scenes are

Isadora (v1.1, pre-release)
manual, p. 68 —
http://www.troikatronix.com

> like scenes in a play: each one may have a different set, different lighting, etc. [...] Because you can jump almost instantly from one scene to another [...] it is possible to move from one interactive setup to another as you move through sections of a performance.

In a sense they are environment support for episodic pieces. The metaphor given is one of lighting or stage cues but while this is useful for understanding what they are, it is just as useful for discovering what they are not. For in lighting and stage cues there are a well-defined set of resources for a scene change to act on. This enables the idea of a transition to be defined as resources — light levels, ropes etc — are moved from one state to another. Not so in digital media progressing networks. A scene change, as given by Isadora or by Max's limited control flow, a necessarily dramatic event — it involves the initialization and

configuration of one circuit and the termination of processing in another. No amount of support, which Isadora has, external to the patch, for fading in or out the video output of a circuit really meets the challenge of a live multi-media "transition". A cross-fade of end-products does not allow the outgoing modules that control that video output to negotiate their relationship with the modules that will soon replace them. Even the simplest L-cut of film-editing — where a cut in one medium precedes a cut in another — violates the constraints of this clean cross-fade.

This technique depends on the superimposability, and we might rather say, texturality, of the predominantly video-like media that flows through those circuits during this switch over or "cross-fade". This is in contrast to a tool that would acknowledge the geometricality of the processing graphs that are being juggled by this scene switch.

The ultimate inadequacy of Isadora's "scene" helps us discover our third main criticism with the whole data-flow paradigm, while Max and others may organize the flow of data around efficiently and somewhat visually, and their control structures, while questionable, are clearly serviceable, their relationship with *time* is particularly weak. To my taste, to be able to create and manipulate complex temporal flows is more central and harder problem than the creation of complex flows of data. Superiority in the domain of creating complex data-manipulations are something that visual environments such as Max can battle over with programming languages like Java, but the layering and negotiation of temporal structures is something that both Java and Max do unquestionably poorly.

The perception / action / motor system decomposition of an AI agent is as much about the layering and negotiation of time than it is about anything else, and we have already seen (*page 226* and *page140*) a number of additions to our

core programming languages that extend their vocabulary in this regard over traditional imperative languages. Perhaps these victories for a text-based environment can be secured if they are placed in a visual framework that starts by depicting the flow of time rather than the flow of data.

2. *Fluid*, an overview

Fluid, while it might compete in the same arena, has to solve a different problem. While it shares with Max and Isadora the goal of being a working environment for artist — the tool that they use live *in* rehearsal, the schetchpad that they use on the plane *to* the rehearsal and the environment that they develop ideas and materials in long *before* the rehearsal — unlike Max it doesn't have to take responsibility for creating all of the complexity of the piece. Rather, Fluid's job is to make the agent toolkit approachable, improvisatory, extensible and developable, to cull from its potential pieces of an artwork. Thus, it is necessarily hybrid, sharing the development space with the environment used to make the toolkit itself. This can be both a feature and a drawback: it is a feature because the more conventional textual programming environments have had far more development time spent on them than any tool in the arts probably ever will, and a drawback because the Fluid system will never have a complete view of the development of work.

A summary of some design principles behind Fluid will help locate Fluid and define its relationship to both traditional pure"code practice" and the traditional use of environments like Max.

Visible, editable code is a ubiquitous glue in every visual element — thus, every visual *element* on a Fluid *sheet* contains code be it a simple box, a time marker, a graph and this code that inspectable and changeable and, of course, executable. This means that while Fluid is most certainly a visual

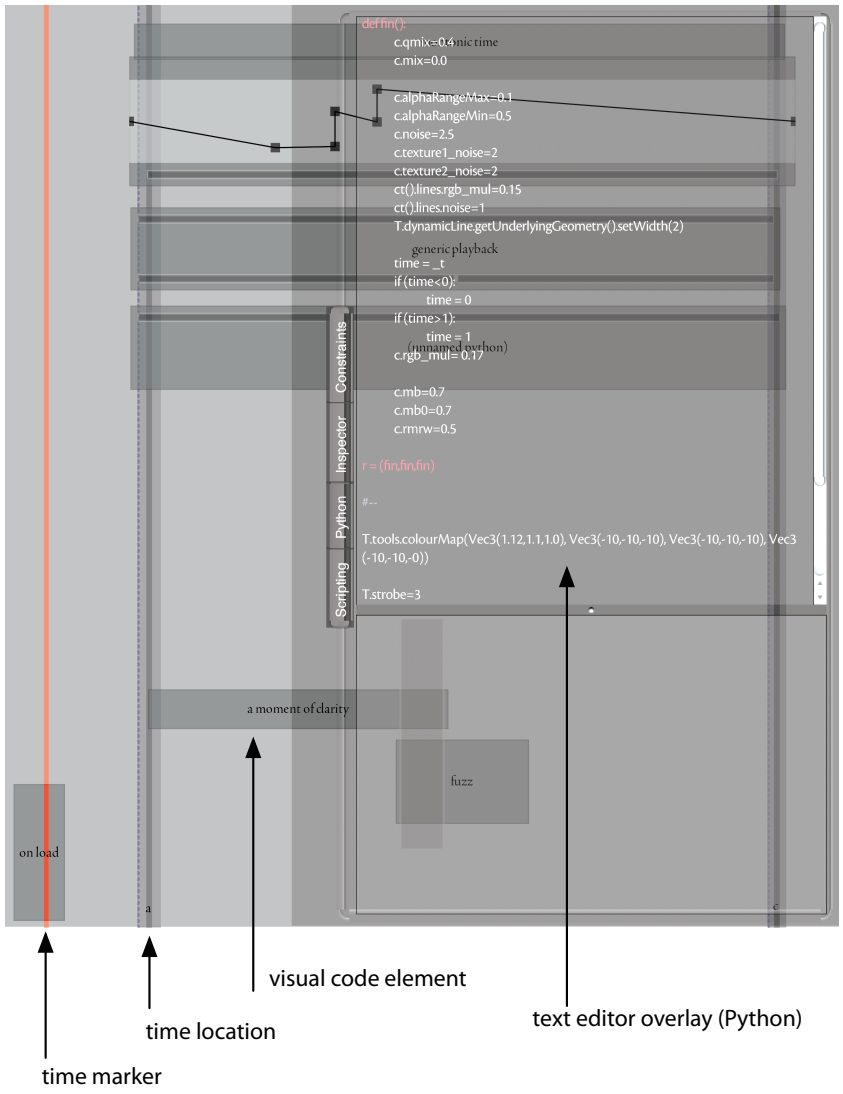

figure 134. The main window for a sheet — visible here are some code elements of various kinds, and the text editor and the output panel.

programming *environment*, it is most certainly not a visual programming *language*. The atoms of programming are readily available and editable but they are not necessarily graphic.

Every visualization is "executable" — a Fluid sheet is a place where one makes fragments of code that call upon the whole agent toolkit; however, it's also a place where the whole agent toolkit can deposit visualizations of its state. But, here, the same principles apply — editable code surrounds and connects the visualized elements to the sheet and to each other.

Visual presentation matters (somehow) — the visual arrangement of sheet elements is typically interpreted by other sheet elements and is always meaningful for some process. Therefore Fluid invests a considerable amount of code towards making general-purpose spatial manipulations available: multiple groupings of objects are possible, two constraint based layout systems are available, code can talk about spatial filtering. But, the visual presentation's meaning is open to definition and redefinition — it might be that there is a flow of scripting type from left to right, or it might be that child elements above other elements are responsible for the children's life-cycles. The visual presentation's interpretation is not set, but there are enough tools for controlling the layout of elements that it can be made usefully important.

One visual element may be in a number of "places" — since visual position is meaningful in a variety of ways, visual position is no longer completely available to the programmer to act as a "secondary notation" for organizing thought and storing memory. To restore some of this flexibility we allow and expect objects to be able to exist on a number of sheets simultaneously, including sheets that may not be currently loaded. Often multiple sheets are stacked in layers showing visualizations of lower layer's contents and workings or showing the relationships between sheets.

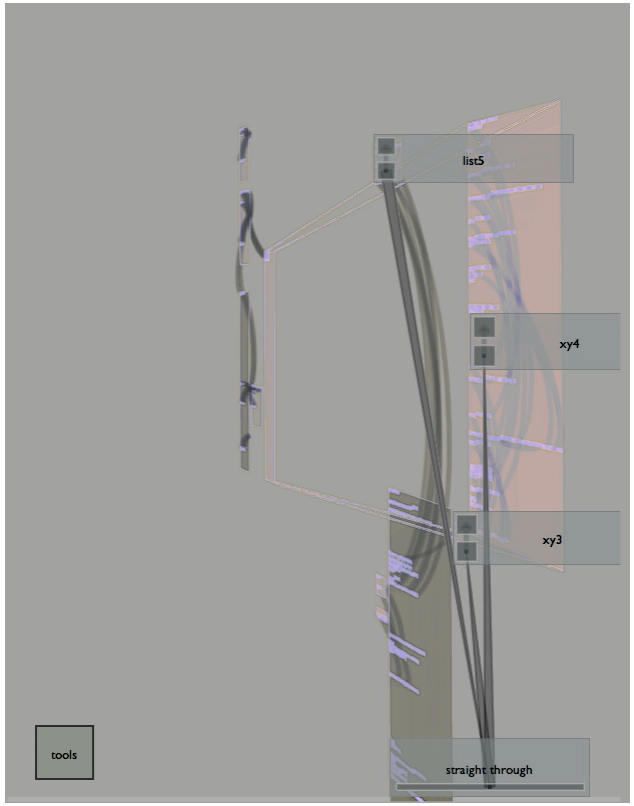

figure 135. The boundary between finished artwork and development tool is blurred. This image is of the Diagram visualizer, which can underlay the *Fluid* framework. The visualizer itself showed along side *Loops & Loops Score*.

The history of using the tool is in the tool itself — programmers have typically surrounded their editors with extra versioning systems that keep track of how textual code is changing from day to day. These tools are almost always domain-agnostic, handling text files with no knowledge of their contents. However, in collaborative art-making the history of the collaboration is part of the collaboration, and environments should make the history of their use directly part of the environment. We have seen this need in our analysis of *alphaWolf, page 84*, and I have seen it in my own work, *pages 98, 127* and *209*. This necessitates the creation of domain-specific versioning systems and domain-specific loggings and analyses of how the tool is being used.

The environment can refer to itself — Code in Fluid is open to manipulation by code (in Fluid). Following on from the previous principle, the code inside each sheet element can talk about, manipulate and script the appearance of itself and connect to an interface that allows the manipulation of other components in the sheet. Additionally, the format for storing Fluid sheets is a both human- and machine-readable xml.

Fluid sheets can be instantiated with or without visual display — A sheet can be instantiated without creating any visual component, in the complete absence of the underlying windowing system. This is an apparent feature of almost any graphical programming environment, however it is harder to maintain in the presence of embedded code that might end up manipulating the "appearance" of a non-graphically instantiated sheet element. To fully exploit the power of *ad hoc* visual layouts for all kinds of tasks both big and small, we should expect hundreds or thousands of sheets to be instantiated during the life cycle of the work. Any hint of the underlying windowing system in such a process prevents this use case from being either practical or stable.

The boundary between finished artwork and environment is blurred — no clean separation between what is "Fluid", what is "toolkit", and what is finished artwork has been maintained in my practice. The movement of ideas has not always gone from toolkit to environment and the flow of control isn't always from environment to the toolkit. Some examples: the layout constraint system become the motivation for some of the advanced generic radial-basis channel combinations; Fluid can overlay the main graphics canvas and track objects on the screen; there is a pose-graph motor system representation for fluid visual element positioning — one can think of Fluid "agents"; parts of Fluid have even been exhibited alongside *Loops*. The visual element structure can quickly become just the visualization of a Diagram marker channel and vice versa.

Together, these principles imply a visual programming environment that is a radical break from the Max tradition, indeed any art-tool relationship widespread today.

This environment was tailored for a specific domain and a specific working style: the creation of collaborative interactive artworks through intensive workshops, with generous but expensive time in theaters, through improvisation and through the condensation of improvisations. It dodges completely several problems that others confront and, I believe, fail to convincingly solve. Unlike the Max / Isadora / vvvv / EyesWeb tradition it does not attempt to be a visual programming language — it limits its use of graphical display and make extensive use of editable code. This places it firmly toward the language tradition Processing / Drawing by Numbers / Alice. Yet at the same time, this allows it to be in some cases more visual — the visual layout of elements can be made meaningful to and visible to the program itself. However, unlike these extant "art languages" Fluid sheds any pedagogical claims in favor of offering an environment that retains the power of "full" programming languages to scale to large,

multi-hundred thousand line code-bases. Yet it retains and exploits a commonly available programming language that, while it is extended, does not break its connection to all of the literature for the language or research behind the language.

Without extensive and difficult to control user studies, Fluid can make few claims for ease or power of use. This makes it no worse off than the dominant "products" in the marketplace today. which operate without a firm predictive or explanatory theory of their use. At the very least Fluid stands as a unique set of wholly implemented, and demonstrated hypotheses about what features art-making environments need to possess to survive long collaborations around open-ended and complex systems.

The principles above form the backbone of my description of Fluid.

The canonical definition of the Java language is **J. Gosling, B. Joy, G. L. Steele Jr., G. Bracha,** *The Java Language Specification, 3rd edition*, Addison-Wesley, 2005.

for Python, see http://www.python.org

for Jython, see http://www.jython.org

The code that is ubiquitous in Fluid is a"dynamic language" called Python, specifically the python implementation written in Java known as "Jython". In contrast to the main language of the agent toolkit (Java) Jython is an interpreted, late-binding language, the semantics of which are extremely malleable, containing a full and rather usable meta-class programming framework. It has been designed with the purpose of being hosted from within a larger system in mind, and Jython has been designed to be hosted within a Java runtime. Finally, as a language that has a compact and open source implementation, the interpreter is not a closed black box, but is rather an ideal site for further introspection and augmentation. The basic strategy is as follows: by using language such as Java for the large and intersecting agent toolkit, and a dynamic language such as Python for the assembly, glue and interface to the toolkit, the strengths of both programming languages can be combined.

Because of the modifiable semantics of the language we can build carefully prepared classes and environments — for small amounts of Python code to use — and glue visual elements that contain these codes together, giving them purpose on the sheet. Obtaining an editor for these pieces of code hidden inside elements is easy — Fluid presents a typical, and rather complete, code editor that happens to support rich text format files and runtime automatic line completion. Objects in python can be inspected live, and code executed per-line, per-selection or per-element with a single key-press. Quite a bit of time, especially in the early stages of development or testing, is spent purely inside the editor window executing code and inspecting objects. Useful code is then propagated outside the editor into separate visual elements for execution or even just documentation of ideas, examples and tests. At this point we have a system that can support simple spatial mnemonics — the equivalent of a "Desktop" metaphor for small snippets of code.

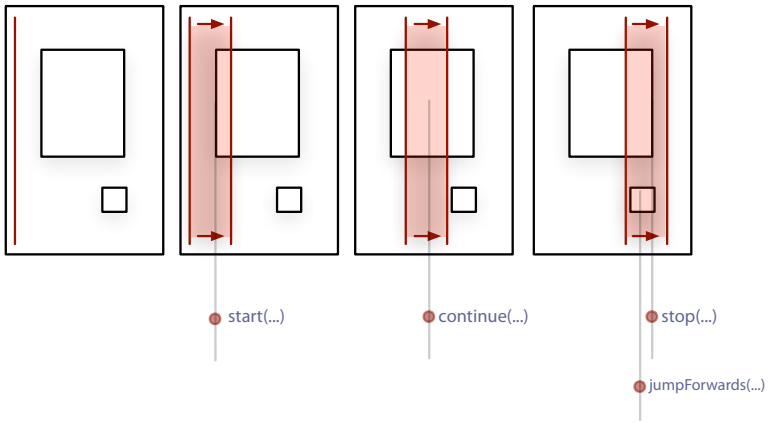

figure 136. The intersection of a moving time-marker and visual elements causes a number of messages to be sent to the elements.

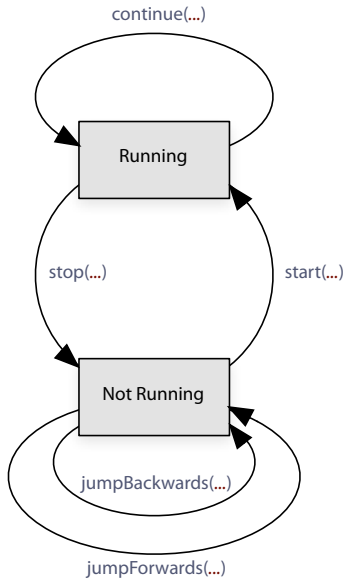

figure 137. The full state and transition diagram for a visual element.

This is one route into a visually "extended" programming, but to go further, the positions of these elements must begin to mean something. The simplest example of this is a Fluid timeline. This is a commonly used way of configuring a sheet and is a good starting place from which to build an improvisation environment.

Inside a time-line there is a series of executable elements and one or more time markers. Broadly put, a time marker executes elements that it crosses. Of course, this means that there is a *life-cycle* for the executable elements — at some point in time they transition from lying dormant to being "executed" each cycle (at various times) and then later from executing to being "stopped'. We know from the rest of this work that it is important to state the contract between the executor and the executee in such life-cycles, the life-cycle for these elements is detailed, open and strongly enforced.

Runners and execution

Helping the time-marker is a Runner class, that maintains this contract, actually executes the code inside each visual element, and interprets that code's "return values" or effects on the codes local environment. Although time-lines are ubiquitous, especially inside computer music systems (one could think of any sequencing package or audio editor in the last 20 years), the presence of executable code rather than musical notes or sounds inside the elements adds a complicating dimension.

The possible state diagram for visual elements executed by a moving line is more complex than it first appears. Since the timeline runner executes code by intersecting visual elements' bounding boxes with the rectangle formed by the sweeping time marker over one execution cycle, we should look at how these rectangles can intersect.

The traveling time-marker can cut across the start or the end of a visual element but it might also wholly consume the visual element (effectively starting it and stopping it in one cycle); additionally it may do this while moving backwards. It is important to allow the code inside the element to respond to each of these events differently if needs be. Many visual elements run only once (on startup), some do the same thing at start, continuation and ending, some are "unmissable" (the proper execution of subsequent elements depends on this element having started) and execute at least their start and end on being jumped over, others are not worth starting unless they are going to continue for a while.

It is the Runner's responsibility to take the text of the code that the visual element contains and map it onto this finite state structure. In Fluid this is done by executing the entire code box once and looking for the value of a "return variable", ***r***. By interpreting the value of *r* — which may be any one of a number of Python objects — the runner interprets what part of this transition diagram gets populated. The following summarizes the return values that have been found to have use inside the current Fluid system (over *how long..., 22, Imagery for Jeux Deux*):

nothing, *r* remains unset after execution. In this case, the visual element wants no further execution. This doesn't mean that it never executes, in fact it has already been executed once to see what *r* was going to be. Such visual elements, therefore are executed only on "start", "jumpOverForwards" and "jumpOverBackwards".

r = **an-executable** (any of a class containing a Python function, method or generator, or a Java or Python instance implementing Updateable). In this case the visual element has offered up an object that should be evaluated or executed for each execution cycle that this visual element is "running". Generators are called only until they no longer return values. In this case,

nothing additional is executed in the case of "jumpOverForwards" and "jumpOverBackwards".

***r* = a 3-tuple**; *r*=(start-executable, continue-executable, stop-executable). In this case the visual element has offered up something for each of the"start", "running" and"stop" stages of the visual element life-cycle. This case is by far the most commonly used case throughout Fluid.

***r* = a dictionary**; *r* = {*start*:start-executable, *go*:continue-executable, *stop:* stop-executable, *jumpF:* jump-executable, *jumpB*:jump-b-executable}. The completely, and rather more verbosely, supplied dictionary of things that might be executed. Any of these can be omitted without error.

In addition to reading this "return value" the runner ensures that certain variables are configured before execution (and before the execution of the returned components of r in the future). These are purely for convenience and readability, all the information is available from the Fluid interface with a little indirection.

_t — the normalized position of the time marker through the visual element. This can, in the case of start and stop parts of the life-cycle, be greater than one or less than zero.

_dt — the normalized instantaneous velocity of the time marker.

_attributes — the persistent attributes dictionary for the visual element. This is a window onto the visual element from the outside world, and a place for the visual element to store things that will survive across executions and even across application launches. This is also how visual elements customize some of their user interface — Python functions that are stored in this dictionary become menu items for the visual element, numbers become interactive sliders and strings become editable text boxes.

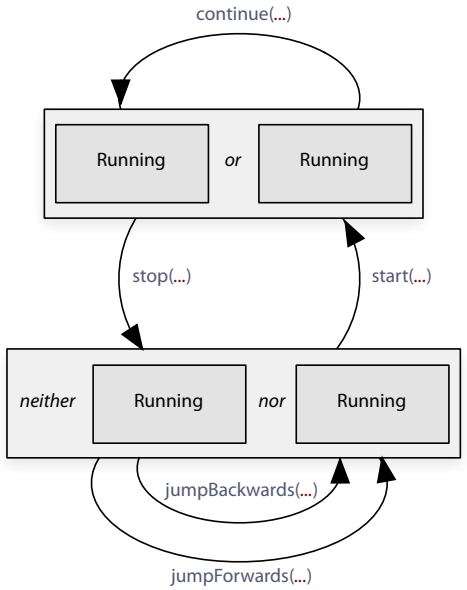

figure 138. The full state and transition diagram for an element on a sheet with two time markers.

In the case of multiple time markers the **_t** in the execution environment is modified to become an instance which masquerades as a number, but contains all the **_t** information from each of the time-markers should the visual element require access. At present the scalar version of this is the **_t** corresponding to the most recently created time marker.

Finally, we note that there can be multiple time-markers at work on one sheet — we'll see below how this can become increasingly useful in more complex sheets. Indeed, there is, in addition to any time-marker on a sheet, another Runner, one corresponding to explicit mouse-clicks on the visual elements. Fluid elements can be executed by option-clicking on them, which spawns a time marker local only to that element for the duration of the mouse movement.

This means that runners must organize themselves such that the life-cycle transition diagrams for individual markers can be effectively shared. This is achieved through a context-based parenting mechanism — specifically all the children of a runner share the activations of the parent. Although there are a great many ways of taking the product of two of these state-diagrams, in practice only one based on the logical "or" of whether a diagram is executing has been of use. Specifically, if any runner claims a visual element as running then it remains or becomes running. One could imagine forming "and" and even "exclusive-or" intersections between runners and their time-lines, but so far no project has needed them. This multiple, distributed-access state diagram is supported using the deferred dispatch and channel rewriting capabilities of Diagram, *page 256*.

Clearly, even with our time-line example there is quite a lot going on — we should begin to look at how these elements are coupled together, and how the sheet assemblage is designed and perhaps even more importantly stored over time. We have spent some time analyzing the conditions under which tight coupling between systems occurs in the agent framework and building techniques such as the context tree that prevent relationships between apparently independent code fusing solid.

Firstly we'd like to be able to reuse visual elements inside different contexts, different sheets, and specifically, according to our design principles above we'd like to be able to use them non-visually. This means that they must communicate with the visual presentation system but not couple to it. Fluid makes extensive use of two techniques — a tree delegation system and an external extension mechanism. The first technique is similar to the delegation chains used for event handling in many windowing frameworks. But I extend this idea to allow arbitrary method calls to be propagated up a branching container chain in a breadth first fashion: from element, to containing elements, to the sheet and eventually to an interface to the containing agent. All of the event handling, execution and visual presentation is handled in this open, over-ridable way.

the Cocoa application framework that is used to implement Fluid also uses delegation chains in part to deliver events from input devices to visual elements, and to delegate method calls; Fluid extends this technique to include the storage of attributes.

Data storage acts in a similar way to the delegation chain, external to the visual element itself. The component of "plug-in" architecture of Fluid is fundamental, rather than just an extra layer of extensibility. Python code execution is a plug-in, time-markers, constraint systems are plug-ins, the very visual position and size of the "visual" element is maintained by a plug-in. The actual information maintained by the visual element itself consists of nothing more than a unique ID. Plug-ins are added to the sheet and as a result offer the ability to set, get, store and delete properties with respect to this tree of containers. This allows

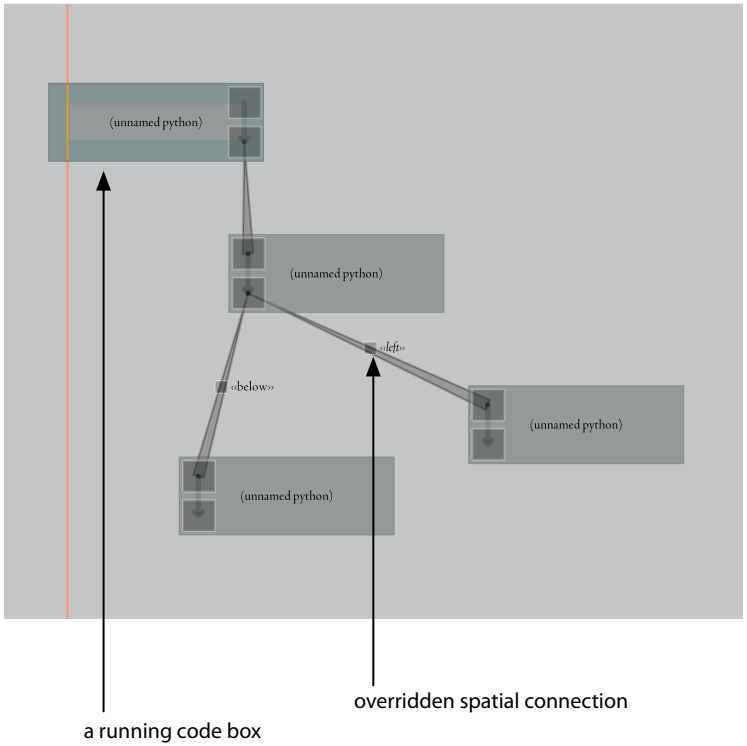

figure 139. Code elements linked together, either by hand or by code.

plug-ins to overlay services into the sheet: by manipulating the container chain, plug-ins can affect the default behavior of some or all of the visual elements. This allows extensions to the Fluid system that are "multiplicative" rather than "additive", extensions that alter the ways that sets of visual elements can execute, combine and can be manipulated, what the visual elements actually present visually, and how they act visually. This is in stark contrast to systems such as Max where "externals" simply add to the numerical quantity of the modules available.

Secondly, although the quantity of code stored by any one sheet is much smaller than that of the framework supporting it, the situation is just as important from a storage perspective. For in order commit to an environment one has to trust that it will always be able to recover one's work even after a several month hiatus, during which time the agent framework might have changed, but more importantly Fluid itself might have undergone revision. The file structures of Fluid were explicitly designed to allow the environment to grow without loosing the ability to load previously saved files. This, in itself, isn't a particularly hard problem, and can be achieved by storing versioning information in the files (for similar techniques, see the long-term learning database, *page 127*). However, it's also important that the environment can shrink or that sheets can be loaded into and saved from completely different environments. So, for the purposes of long term storage, data travels with both the plug-in and with visual element. Unknown data is both carefully ignored, carefully propagated, and in most cases still accessible even in the absence of an actually executing plug-in. Plug-ins are defensively coded to verify the relationship between their internal structures and what remains in the individual visual elements upon load. In practice, two-year old sheets are still loadable today.

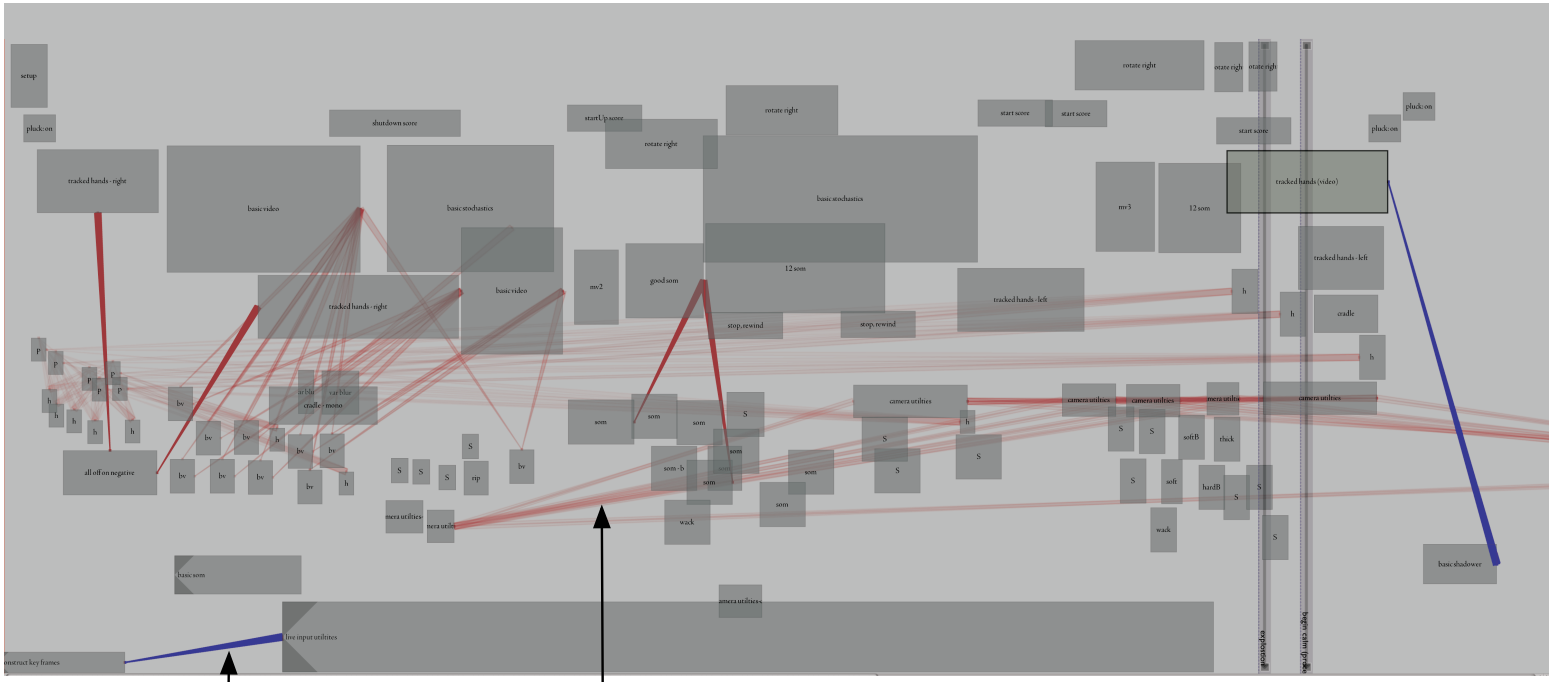

figure 140. The history of ephemeral, implicit connections between modules can be visualized as a separate "layer" to the environment. Such connections are gathered automatically monitoring the Python interpreter.

Connectivity

In fashion similar to that of data-flow environments, we can add to our elements inputs and outputs and begin to draw connections between boxes. The values at the connections can be push from outputs to input from within the Python environment:

```
_output[0] = 5
```

— but these connections are not necessarily for data-flow. These connections manifest themselves as set variables inside the Python environment:

```
print _input[0]
```

however they do not necessarily "represent" the flow of data. They might represent the aliasing and thus sharing of variables between otherwise local python namespaces.

From the output module:

```
makeAliasOutput(0, "a")
a = 5
```

declares that the variable *a* will alias the zero-th output, and from the input visual element we might have:

```
makeAliasInput(0, "a")
print a
```

By going further, and propagating not values, but small objects that reference the visual elements from which they come, visual elements can conspire to appear to implement a style of data-flow programming.

From the output module, it looks the same:

```
makeLiveOutput(0, "a")

a = 5
```

declares that the variable *a* will alias the zero-th output and sets things up to allow this visual element to be executed if needed to evaluate *a*. From the input visual element we might have:

```
makeLiveInput(0, "a")

print a
```

What are the contents of these small objects? Since anything can, with the assistance of a Runner, ask for the execution of another element, these objects ask for just that — to ensure that it has the latest value at its inputs. These executions are safe and timely: aware of that runner's life-cycle state diagram and thus the element that is asking for the computation is guaranteed to both maintain the correct life-cycle contract of the element and only cause one execution per execution cycle. This hybridizes a pull-based data-flow style with a more orderly time-marker style, proving that data-flow, variable execution and alternative visual metaphors can coexist on a single interactive surface.

In data-flow environments, one connects boxes and these boxes remain visually connected — as a "visualization" of the history of interaction, and as a "user interface" that allows the connection to be broken, and restored. Since Fluid visual elements contain code, Fluid offers many other ways of "connecting" elements together by writing code rather than by interacting with the sheet using a mouse: code can look up an element by name, by regular expression; code can find the visual element to the left of it, all of the visual elements underneath it. That these global or spatial lookups can be written in code rather than made by mouse-clicks is a fundamental result of the principle that code can "see" the vis-

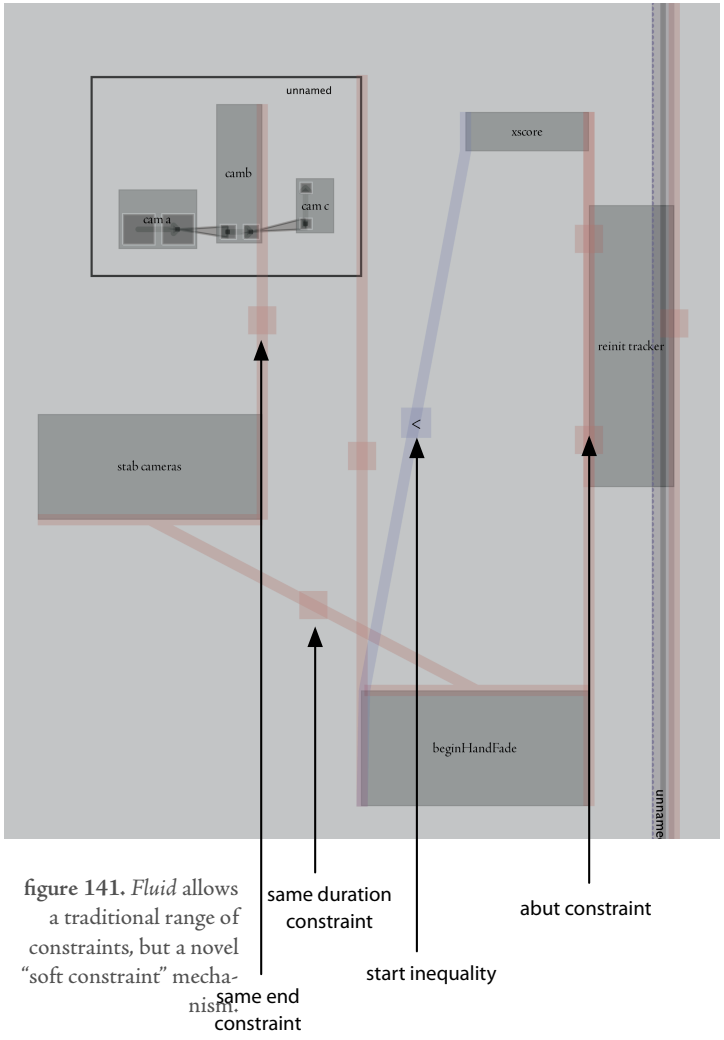

figure 141. *Fluid* allows a traditional range of constraints, but a novel "soft constraint" mechanism.

ual layout of the sheet. However, at the same time, there is much to be said for allowing the environment to provide a "visualization" of what that code is doing and a "user interface" for allowing these connections to be rearranged.

The solution is to construct a visualization layer that, by collecting information from the python environment, as it is executing code, annotates the sheet with the connections that the code makes and offers the opportunity for these connections — which might be one visual element connecting to an element to the "left" of it — to be frozen or reconnected. These layers are translucent, optional and overlay the sheet — they are coupled to the sheet through the plug-in architecture.

Inside the Python these connections look like:

```
target = leftOf()
```

or

```
target = find("fade out *")[0]
```

This provides enough stability and flexibility that the layer can edit the environment of the visual element to ensure that connections overridden by direct interaction continue to be overridden. The connections fade over time as they remain unmentioned by the executing code.

leftOf(), find(...) etc. return Python objects that masquerade as references to other visual elements, and exploit the *_attributes* dictionary to ensure that they remember whether or not they have been overridden in the sheet.

There are additional reasons why we would like to be able to trap and interpose all references to external visual elements by the code inside a particular visual element. We'll see the importance of being able to draw a circle around the context accessed by an element when we look at recording the history of interaction with Fluid, *page 390*.

This highlights two of the design principles: that code and visual elements should coexist on the same sheet, and that the history of using the environment

should be reincorporated into the environment. This interplay between making the results of code visual and turning visualizations back into code is what constitutes one axis of "fluidity" inside Fluid. But before picking up the thread of incorporating use history into tools in a more focused fashion, I will survey some of the other kinds of layers implemented in the current Fluid system and how they are used.

Multiplicative extensions — Alternative layers

I. Sutherland, *Sketchpad : a Man-Machine graphical communication syste*m,Annual ACM IEEE Design Automation Conference, 1964.

Of the most important layers available in the current version of Fluid are the constraint systems. Constraints have a long history in visual layout tools — in particular they form the very basis of one of the very first visual layout tools Ivan Sutherland's seminal Sketchpad system. But, despite Sketchpad's hybrid programming / drawing approach, visual layout constraints are completely absent from the history of visual programming environments for digital art — present neither as a tool in the Max/Isadora/vvvv series , for visual layout is unimportant in these applications. Paradoxically, each of these graphical systems offer less support for fast visual layout than most drawing or painting applications.

However, when the visual layout of the sheet means something — to the code contained as well as to the user — and when the environment is the platform for a certain amount of improvisation, it is important to allow the specification of more complex and quick manipulations of visual element layout.

Two constraint systems are implemented inside Fluid, both have the same interface and appearance and allow a conventional set of constraints to be specified on the layout of the visual elements. Same, Before (and by inverting the parameters After) relative constraints on both the start and the end of elements; Same, Bigger (and thus Smaller) on the duration of elements. These constraints

can be applied to visual elements that group visual elements (and distort their contained children equally when needed). Finally, elements positions can be pinned to a particular spot — this allows all of the previous, relative, constraints to have an absolute aspect, since visual elements to represent absolute positions and sizes can be created with ease.

The perennial problem with constraints, however, is that it is extremely easy to construct over-constrained systems, and extremely hard to build fast but stable solvers for these systems. Although much work was done in Sketchpad and afterwards to address these issues, Fluid dodges the problem creating two constraint implementations, neither of which has any claim to optimality, but rather a focus on stability and speed of execution. In the future, a more complex linear-programming-based constraint system could be implemented.

The first constraint system is a rather typical damped iterative solver that tries the best that it can, with a decaying amount of effort, to maintain each of the constraints in turn. Should a constraint be broken (due to over constraining) it is brightly indicated on the sheet. This ad hoc solution works well for under-constrained problems and tends to break stably in over-constrained situations. Never has a sheet "exploded" during all of the improvisatory use of Fluid in developing *how long..., 22* or *Imagery for Jeux Deux*.

The second constraint system is a little different. It is based on the generic radial-basis channel formulation for competing processes. In under-constrained domains it acts the same as first constraint system — it is a damped, iterative solution to the problem. However, rather than trying to find a nearby, stable solution to an over-constrained problem, the competitive channel representation gives each constraint a certain amount of time to apply. This "solver" actively explores the over-constrained partial-solution space, generating not an attempt at a single solution, but an ongoing animation. To date I have used this solely as

an exploratory technique (for the generation of rhythmic cells that are perturbed in different ways), or a visualization technique for the similar Diagram based processes of *Loops Score, page 241*. It is expected that in the future that this technique will autonomously create rich rhythmic patterns in the domain of motor systems on its own.

Regardless, the constraint system is perfect for making visual and maintaining the ordering constraints of the visual elements code — that one thing should take place before another, or that this visual element cannot end before another — that allow sheets to be quickly reorganized during rehearsal before calling upon a time-marker to"scrub" with.

Another available layer is related to the constraints system — the layout snapshot. This is a duplication of a sheet (in the sense given below, *page 390*) that saves the positions and sizes of the all the elements. This layout can then be blended with the current layout. New visual elements that are not present in the saved copy can remain stationary or, more usefully, can get pulled around by the nearby saved elements movement. This reuses the same techniques as found in the Diagram channel system, *page 232*. A pose-graph-based view of these snapshots exists, and in the future we might see an agent acting upon a Fluid sheet itself.

Fast visualization for the agent toolkit

During the use of the agent toolkit many programmers — inside and outside the Synthetic Characters Group, myself included — have produced carefully crafted visualizers and debugging tools for various systems. At any given time one could expect to find a motor system visualizer or two, three or four for the context tree and so on at various stages in the development of the agent toolkit. With modern, graphical tools, building and maintaining these tools isn't hard,

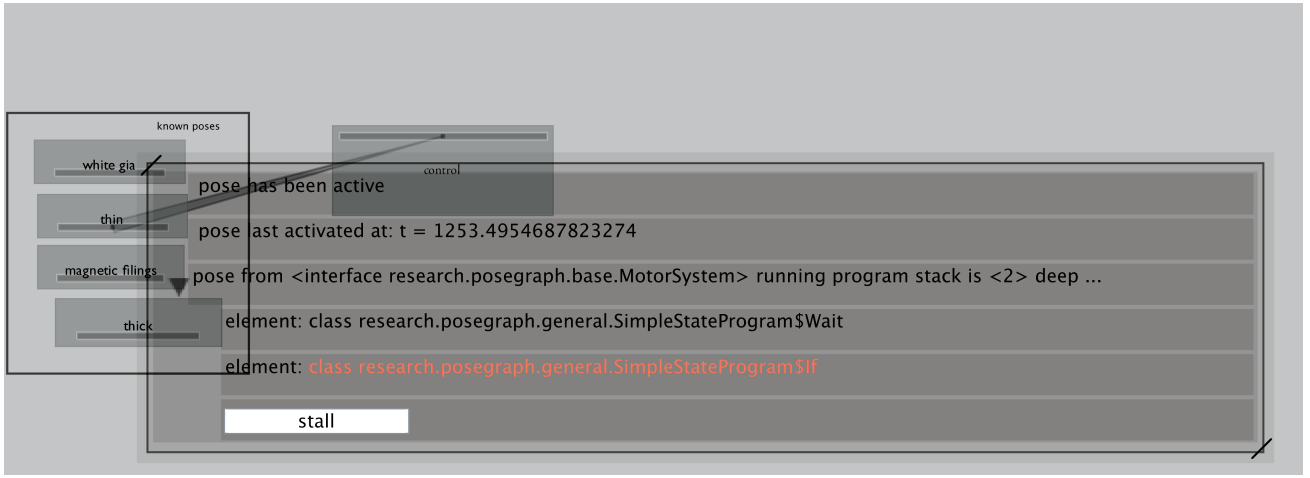

figure 142.
A dynamically created, *ad hoc* debug display and interface

nor is it as time consuming as it used to be. But it does require a constant effort parallel to the development of the system being visualized. And there is a constant tension between creating a well-designed interface (code) for the interface (visual) and creating one quickly. As a result, these carefully crafted visualizers are often out of date at any particular point in time — if, that is, they get created at all.

Fluid potentially offers much more than either a hand-crafted visualizer or a traditional debugger since it integrates graphical user interface construction tools, code execution and domain specific storage in one place. In order to bring the toolkit closer to the Fluid sheet, it is important that the toolkit can offer objects *to* the sheet on an equal level to the visual elements; that agent toolkit objects can be visual elements — that one can connect to the motor system or a pose in the pose-graph, visually and spatially. This is important even in the simplest, and least"creative" use of Fluid — having a sheet be a place where visualizations of a running system can take place.

Clearly, it isn't hard to have the toolkit load a sheet and procedurally create visual elements inside it, but some caution is needed — if these "offered" visual elements are to fully participate in the Fluid framework they need to participate in the long term storage of the sheet. This implies that offered elements, which are free to change in number and nature from invocation, to invocation must be matched up with visual elements that are free to be edited, moved around and otherwise adorned as they are loaded from the persistent store.

Three sets of parameters must be considered in this merge — the new creation parameters offered by the toolkit, the creation parameters previously offered by the toolkit (at the last occasion that the sheet was saved) and the parameters now specified by the sheet itself. The differences between the first two are applied to the third unless there are corresponding differences between the last two — this is the classic, three-difference merge algorithm applied across a set of attributes, and the visual element position. Later, we will see another application of this algorithm to the textural contents of the visual elements, *page 390*.

For an overview of the three-way merge algorithm — **T. Mens.** A *state-of-the-art survey on software merging*. IEEE Transactions on Software Engineering, 28(5), 2002.

Once offered, these visual elements are now a bridge between the agent-toolkit and Fluid. However, both parties ought to be able to create simple layouts and interfaces that are more complicated than a simple box. Fluid, of course, allows one to surround these offered elements with code and other visual elements.

Yet at the same we should realize that most "debugging code" exists inside the agent toolkit as textual output not user-interface construction code — how should these pieces of text describe user-interface layout? The quickest and simplest debugging output statement from deep inside the agent toolkit looks like the following:

```
stream.println(" motor system value :"+amount);
```

These statements are ubiquitous throughout programming — stream could be an interface to a complex logging interface or simply an interface to the system log. The sheer number of statements like the above make the use of such code seem almost inevitable. The prevalence of these lines inside the agent-toolkit seemed impervious to the increasing flexibility and availability of visual user interface design tools prior to Fluid. All collaborations (and all programming collaborators), from *alphaWolf* to *how long...*, include these lines.

It's not hard to see why they might be more maintainable than a hand constructed interface — they are programmatically described, compiled with the system that is being investigated and require no interface for that system to be created and maintained simply to get at the misbehaving number. If, in comparison to contemporary data-flow tools we are to conceptually embed complex systems *inside* our visual elements, rather than construct complex systems *with* visual elements, there ought to be a way for these opaque complex systems to talk back to the visual elements.

So we start here, with the kind of talking that seems so prevalent, and construct a incrementally more complex "stream visual interface" to the visual element. Offered visual elements provide an object, "stream", that can be written to as above. We augment the traditional stream output with the following features which augment the visual layout of the stream and provide a lightweight bridge to the Fluid's visual elements:

Text "lines" become rows in **an outline view** — rows are collected only for a fixed number of update cycles and such cycles group the output; the elements "[[name" and "]]" bracket sub branches of the tree. This allows the debugging output to be presented in a hierarchical, multi-resolution fashion.

html processing tags are acceptable — since we are free from the assumption of plain-text output there is no reason not to allow a subset of the rich-text format to be displayed.

Stable user-interface objects are possible — the tag "<button name='name'> label </button>" writes "label" not in an hierarchical outline-view text row, but rather a button in the row; the tag "<slider name='name'>label</slider>" makes a labeled slider.

The values for these two interface elements are written as attributes to the visual element and can be read by the agent toolkit as:

stream.getAttribute(*name*)

From the Python interface these attributes are read and written simply as the value "*_attributes*.name". The hierarchy of debugging output can be parsed (in plain text) through an object called *_debugStream*. For example:

_debugStream[2] is equal to "motor system value :5.0"

and

_debugStream.motorSystem[0] is equal to "at SIT"

These two accesses to the debugging information mean that the visual elements that surround the offered element, and the code inside the offered element can access everything about the *ad hoc* debugging interface's output. This text-based graphical visualization completely avoids the overhead and complexities of creating visual interfaces and code interfaces that they connect to, which is a particularly error-prone area of programming. One must take special care to maintain the same behavior of a system regardless of whether anyone is looking or not. This often requires the caching away of transient data and, depending on the windowing toolkit used, may even have thread-safety issues. The near inevi-

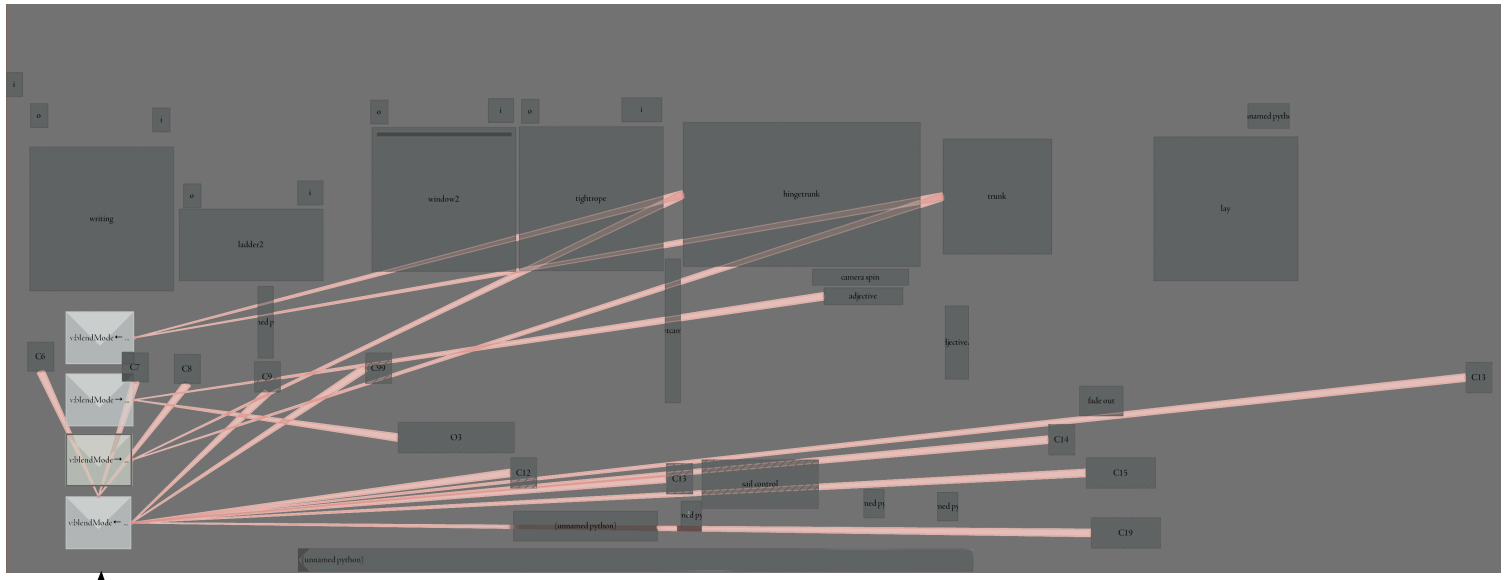

figure 143. This figure shows a, live, layered display indicating read and write access to a sheet-local variable

tability of text-based debug output, and the error-prone nature of the mix of user-interface code inside the agent-toolkit stands as one of the lessons learned from the complex collaborative endeavors from *alphaWolf* to *how long*.... This push-based debugging, although offering a more generic, more rudimentary visual presentation, meets the complex code-base where it stands — the rest of the Fluid environment can be used to customize the presentation of information. Fluid becomes a site of interactive visualization and investigation that meets the agent-toolkit on the toolkit's own terms.

Expressing history

During the course of creating the piece *how long*... the master sheet that controlled the piece was loaded and modified 206 times; secondary sheets, for testing elements and working on specific sections were loaded and modified 3223 times. With the exception of some 45 unexpected fatal crashes which may have resulted in data-loss, a detailed history of the creation of the piece, and all other

works created in Fluid since May 2004 (when the database came online), was stored. But what is a "detailed history" of working inside Fluid? and, equally important, how should it be made available to the tool itself?

Programmers have long surrounded even solo work with versioning systems that allow them to consciously checkpoint their work — storing it a central database format. Concurrent versioning systems, designed for more than one programmer to work on a set of resources at the same time, are also the backbone of both the open-source movement and almost all large closed-source development models. This is very much prehistoric computer science — the core formats and algorithms for storing and generating annotated views of the changes that resources undergo in these systems have been found and fixed for decades.

For a history of one of the oldest version control systems that is still in use today: http://www.gnu.org/software/rcs/rcs.html

But the importance of making the history of the development present in the tool itself was brought to my attention in at least three ways. Firstly, the presence of the "commented history" throughout the text of *alphaWolf, figure [x], page [x]*. If this history was so important as to be preserved in the files itself, despite universal struggles with the bulk and complexity of those files perhaps this is an indication not only of the important of developmental history but the inadequacies of conventional version control tools (which were also, of course, very much in use during *alphaWolf*). Secondly the realization that I repeatedly required access to the history of many of my persistent stores — the long-term learning databases of *The Music Creatures, page 127*, and the bundles of parameters in *Loops, page 98*. This history was not always in a form that conventional version control systems found easy to use.

Finally, watching my own work patterns inside Fluid I identified a number of cases where having history at my disposal would prove useful. The simplest case is execution history. In a fully through-out sheet execution is often under the

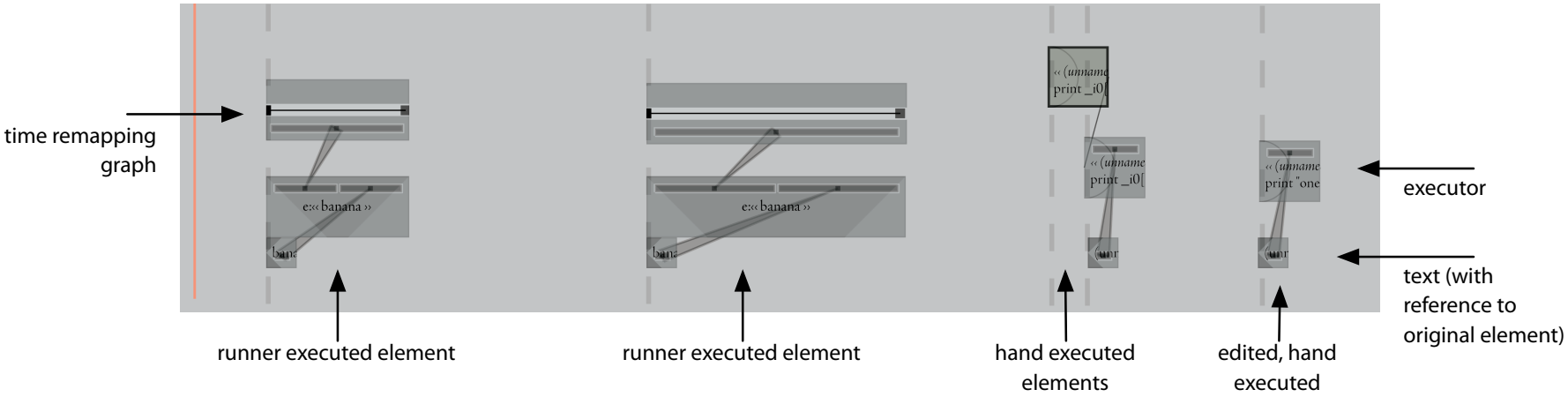

figure 144. This sheet was automatically created from the interaction history with another. This "unrolling" of a marker sweep and piecemeal execution of code inside visual elements is itself executable and remains linked to the original sheet.

control of the visual elements themselves — intersecting time-markers, moving visual elements, running scripts and so on. However, early in the development of something, or when some specific case is being explored, execution is often much more piecemeal — one has found a case that doesn't quite work right, or one is beginning to test a new component — through highlight parts of code and executing them, using a Fluid sheet as a sketchbook to sample from rather than as a place to put ideas down. Further, in improvisations one is often moving too fast to remember what one is doing.

What each of these cases really needs is the execution history of a sheet and its attached textual editors — what code, in what elements, when. Early in the development, when executing samples from a sheet, or samples from a long unstructured piece of textual code, one ends up trying the same pattern of execution repeatedly — to get back to the place where a problem occurs or where the horizon of knowledge lies. In improvisational contexts one needs to go back and look at what was done in the heat of the moment. The solution is to begin to look at ways of turning the execution of a sheet into a new sheet.

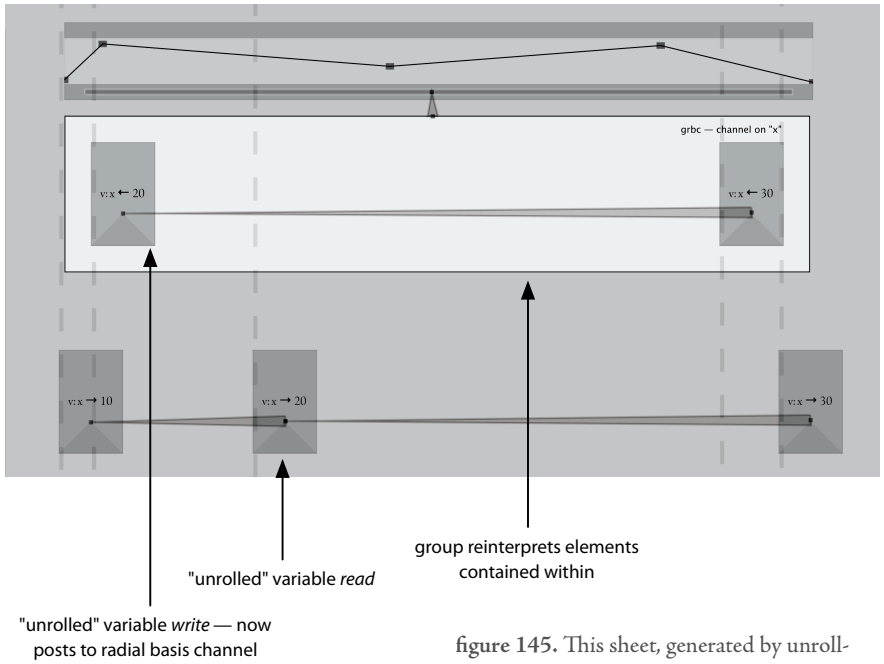

figure 145. This sheet, generated by unrolling another, has had some variable accesses grouped together inside a subgroup that "reinterprets" the code that writes to those variables. Now, rather than directly setting values, postings are sent to a generic radial-basis channel.

As a naïve start, we can take each executed element and copy them to a new sheet, a time-line sheet, where execution time runs monotonically and evenly from left to right. This, for example, "unrolls" or flattens-out any temporal manipulation that was happening to an underlying time-line sheet or "scores" an improvisation that sampled from various parts of a sheet in an *ad hoc* fashion. Since highlighted snippets of code can be executed in the textual editor, these need to have visual elements created for them. This is the (visual) equivalent of converting a marker generator in the Diagram framework to a channel representation.

This unrolling must carefully propagate a snapshot of the local environment of the visual element at the time that the element or snippet was executed to the newly created elements — otherwise the new sheet will not perform in the same way when executed. There are up to three places that this "local state" can be put in relation to the newly created visual elements: back inside the local context of the visual element, as an explicit addition to the code stored in the element and as a separate, but (visually) linked element on the sheet.

The "local context" to a piece of code executing inside a Fluid visual element is a rather complex affair — but in all cases we can trap it by a placing a few hooks into the Python interpreter. The complete context caught by the unrolling history functionality is as follows:

Python-level local variable access — this needs to be recorded in the unrolled sheet, if it is read by the visual element or script before being written to. It can be a separate element — injecting a value into the new element's local variable space — or additional code in the new element's textual description.

context-tree variable access — very similar to local variable access; writes and reads are typically annotated on the unrolled "score" as separate visual

elements. Making these global accesses explicit is a useful visualization understanding. Below we shall see more advanced uses of this score-like style.

visual element persistent attributes — visual elements have a stored (across loading and saving sheets) set of attributes that can be read or written by code, or by inspectors. Reading or writing these requires duplication in the new visual element. This is always performed by making new stored attributes.

sheet-level access — what should operations which result in obtaining references to other visual elements return in the duplicated sheet? Should a visual element A find a visual element B that also ends up duplicated in the unrolled sheet then we can transport the reference, making a new reference from the duplicate A' to the duplicate B'. Should B not already be duplicated, then we either have to try to copy *B* or make a new reference to the original element B. Currently this second operation seems well defined and Fluid makes a cross-sheet reference A'->B and allows this reference (using the same techniques as we use for overriding spatial references, *page 379*) to automatically load the missing sheet, if needs be, on access.

Since these "visualizations" of the interaction history are executable, we can sweep time across the sheet and play back what was done before. We are free to take these sheets and begin to edit them — changing the order of operations, splicing them with alternative takes, etc. We are also free to re-express their contents in a different way. One highly useful modification of a sheet that is typically produced by this unrolling is to take variable access and replace it with generic radial-basis channel postings.

The new group, surrounding the variable reads and writes re-interprets the contents of the code below — wrapping the execution environment of the visual

elements in a structure that maps variable access to generic radial-basis access. One channel (sharing a time-base with the sheet) per variable is created as needed by monitoring the underlying Python interpreter for global variables, and temporarily overriding the object that is used for context-tree access. Writes become postings (with window parameters set by the duration of the visual element) and all reads return python objects that masquerade as numbers, but access the corresponding channel.

Even without their modifiable executability, these unrolled sheets or score-like diagrams have been extremely useful in both remembering and showing what happened in an improvisation that takes place in a theater under time constraints dictated by dancers and musicians. But this kind of use is a short-term use: logging information taken in the moment is there to be looked at soon after and understood, perhaps played and replayed with a little more, but strictly from the point of view of understanding what took place.

These sheets, as described thus far, cannot offer a longer term record of what is taking place, because they go out of date. They loose their connection to the sheets whose execution they annotated when those sheets change. They are a tool for recording only so far as they become separate from what it is that they record. Is there a deeper way to link the record of an improvisation around a sheet to the sheet while, at the same time, maintaining this connection through potentially separate evolutions of these sheets? A history of interaction with a tool that isn't a frozen record but a new view onto the material interacted with?

A network of text: copy & paste as a version control system

At the root of this issue is the problem of duplicated textual code. Currently copy and paste is a ubiquitous part of any textual interface — there is hardly any text box or textual widget that does not support this on any contemporary

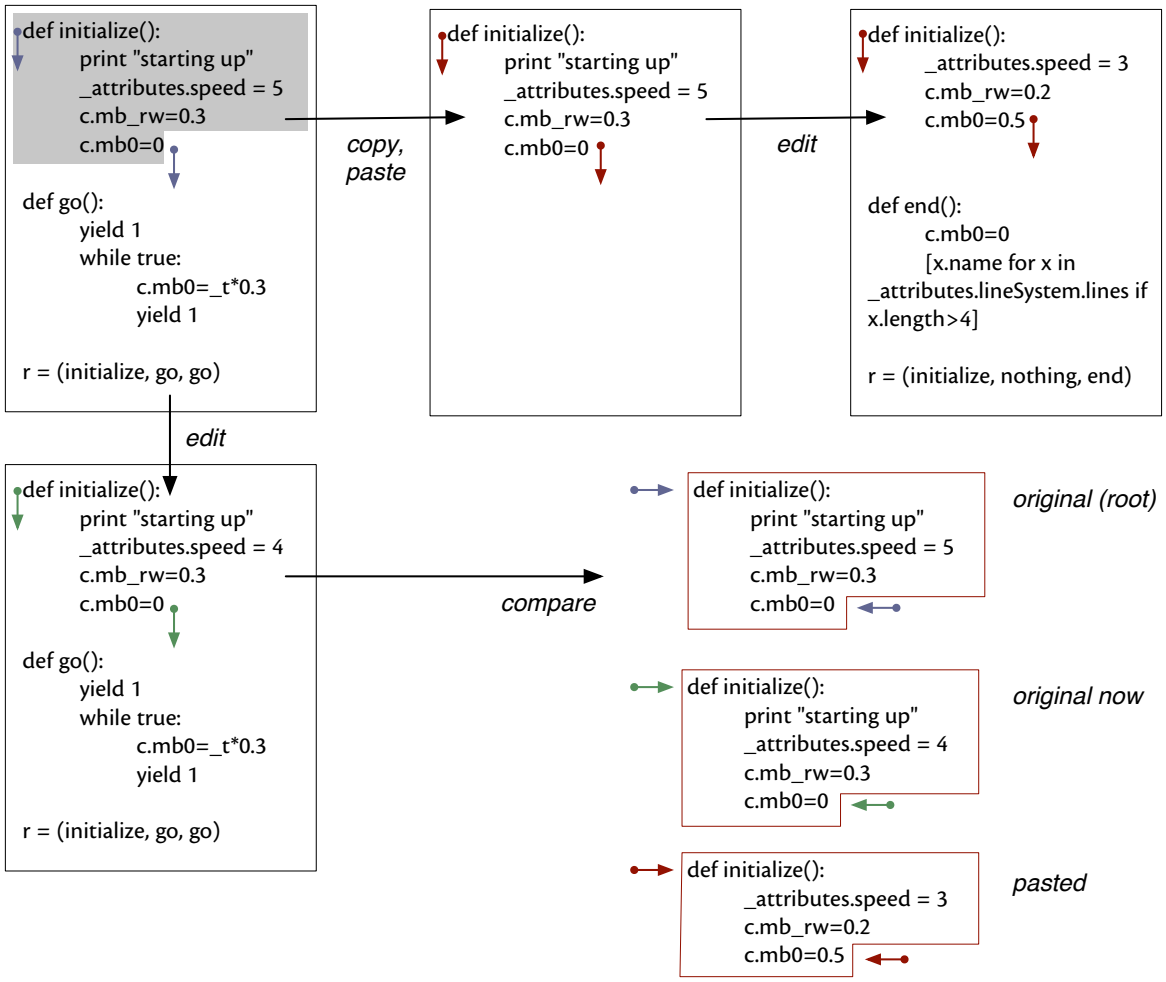

figure 146. Copy, paste and edit creates a versioning "problem" and an opportunity to use the three-way difference algorithm to inspect the history of snippets of code that are transfered around and across sheets.

windowing system. However, a copy and paste operation leaves no process trace. Further, the nature of the process, this duplication of code, is invisible to extant programmers' version control systems. These systems realize that code has been added somewhere, but do not retain the connection between the copied and the copy — for that connection is lost a long time before the versioning system operates on the file. Typically this is not a significant problem in versioning control — versioning systems are so old now, that had it been a problem we might have seen a good few solutions, especially as such systems are being integrated into programming environments. Copy and paste, after all, is often considered a symptom of a poor programming infrastructure, and some of the classic design patterns and indeed some of the motivations for object oriented-programming itself are to reduce the amount of code that has to be copied and pasted in order to program.

However, the theoretical goals of a programming language and its use in practice often diverge dramatically — despite the inability to accurately reconstruct the copy and paste history of, for example, the source code files of *alphaWolf* one can feel its presence throughout. The simple solidity of taking one element that is known to be tested and working and duplicating it (rather than refac-

toring it in such a way that it can be multiply instantiated in doing so potentially break what is known to be working) is a powerful temptation even for the best programmers when under pressure.

In any case, in Fluid, we are are operating in a completely different domain from where these arguments for careful object-oriented design typically take place. Rather than large, re-factorable, and pre-thought out complex code-bases, Fluid's visual elements are an *ad hoc*, often improvised arrangement of very small parts. That copying code is simply the fastest and easiest thing to do (rather than re-factoring the design of material inside a visual element) is much less avoidable in this domain. Rather than reinventing the theory of re-factoring to cope with the kind of fragmentary, poorly planned, spontaneous code that Fluid encourages and circumstances dictate, we do the opposite — shape the tool around the use, and open up the history of duplication to the versioning systems.

Thus, in Fluid, copy and paste operations leave persistent process traces. Fluid exploits the commonly used *rich-text format* for the storage and the presentation of code to the user. By embedding custom tags inside the text structure of a modern text formating system we can annotate the relationship between the copied and the copy in a persistent fashion. We can then recreate the common versioning system operations not in terms of files of code, but of chains of copied and pasted elements.

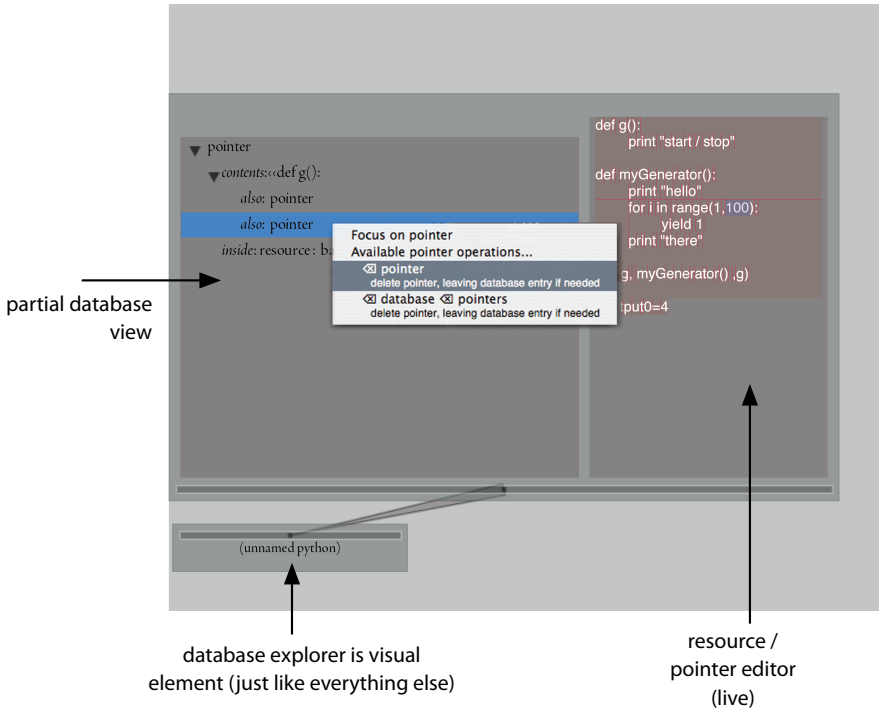

figure 147. The text database explorer interface can be manipulated from within the fluid environment — it is constructed from visual elements.

The rich text format — http://www.microsoft.com/downloads/details.aspx?FamilyID=ac57de32-17f0-4b46-9e4e-467ef9bc5540&displaylang=en

The use of it here depends on the application framework's handling of alien RTF tags, which seems implied by the specification. These tags are, in the current implementation in Cocoa under Macintosh OS X persistent across all applications that deal with text, not just internal to Fluid — they mark blocks of text as having references to a database through the use of unique IDs.

Given a snippet of text we can perform the following three operations on it with respect to viewing its history:

show resource — when a copy / paste relationship is first created a resource is created with it. This is the central representation and marks that this particular piece of code is important and should be tracked. Everything else, the copied and the copy, has a relationship with this resource. By looking at the resource we can then see everywhere this text ended up, or where it came from. By looking at these resources, we can compare how they have changed, or how they are being used. We can begin to examine the ramifications of changing something that this code depends on, and begin to repair it when we do change it.

force changes to (resource, later, earlier, all) — this forces an overwrite of the contents of the text snippet-to the resource or to a subset of children of the resource. The labels "later", "earlier" and "all" refer to child snippets that were created either later or earlier than this particular snippet of text.

merge changes to (later, earlier, all) — this performs a three-way difference merge with this text, its resource and each of the places linked to this text that appeared after or before this relationship was created. This difference / merge algorithm is standard as part of a concurrent version system — here, however, it isn't the work of different programmers at different times that are being considered as happening "concurrently", it is the work of one programmer in a number of places. Unlike version-control systems, the "files" (fragments of text spread across visual elements and sheets) that need to be considered are being automatically inferred. Collisions (incompatible, "simultaneous" changes of text that cannot be reconciled) are flagged for special handling.

loose snippet (resource, all) — breaks the connection that this piece of text has with the database with respect to a single database resource or with respect to all resources associated with this snippet.

The interface shows an local, hierarchical view of the database — resources point to snippets (ordered in time) as children; snippets have a resource and a visual element associated with them; visual elements contain multiple snippets. The textual contents of all these elements can be browsed and edited without loading the associated sheets. The database of resources and concurrent snippet versions is maintained in parallel with the text storage of the individual sheets. This provides safety in redundancy (since Fluid remains an experimental and evolving system); at any time we could delete the entire database, loosing the text level history information, but maintain the present state of the sheets together with their version history (maintained in a more traditional version control system).

Finally, while I have described this system as one fore keeping track of where copy and pasted code ends up — maintaining the relationship between copied and the copy on behalf of the programmer — it also serves to maintain a connection between the unrolled sheets and the sheets that were unrolled — on behalf of the Fluid system itself.

The flow of time — more controllable time markers

The central idea behind the time-marker on a sheet is to allow a visual environment to start with what I believe is the one of the most central parts of the problem of digital art — the patterning of time. And, in starting here, we start by analogy to the representation most present in the temporal arts — the linear score.

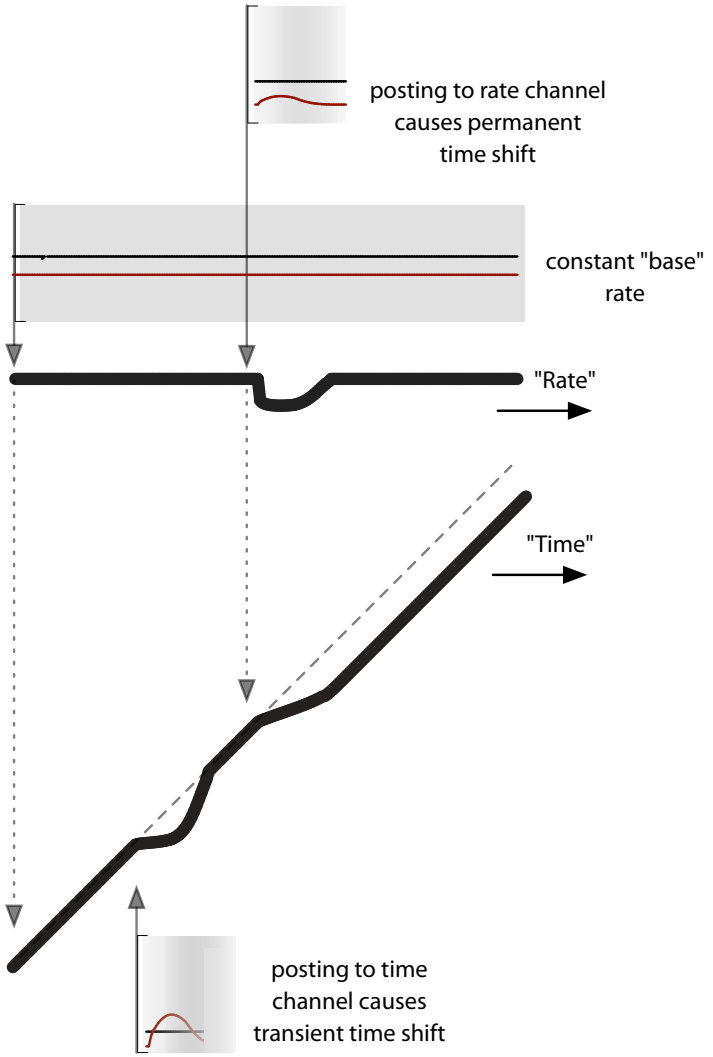

figure 148. Two generic radial-basis channels control the movement of a time marker allowing both changes to the position that are both transient (in the sense that the duration of the work does not change) and permanent.

While I have argued that the agents constructed for the work I've presented do an excellent job of patterning the time that they occupy, and an equally satisfactory job of provoking ways of thinking about how that time of interaction could be filled, in many works there has been a layer either above or below the agents that has had a strongly score-like flavor to it. In *Loops* I constructed a colony of creatures capped by a score and noted that this began to look like a motor system of another super-agent, *page 101*; in *how long...* I deploy agents throughout the work but organizing their sequence and overlap in a fairly linear fashion to align with the performance, *page 353*; *Lifelike* is performed in a similar way, with a more complicated overlapping of less complex agents; in *22* we are in effect seizing control over the material that the motor system of the agent there uses, *page 269*; at the detailed levels of parts of *how long...* we are constructing movement out of overlapping linear sequences, *page 341*; in *Loops Score*, there is again, a score not of notes but of opportunities for action, a "perceptual score", *page 242*. In each of these examples we are not so much scripting the actions that the agents will take, depriving the metaphor of its idea of autonomy; rather we are either scripting the manipulation of part of the perceptual world that the agents are in or providing scripts for the agents to in turn manipulate. In both cases the lines of these linearities cut right through hundreds of files of code and we are forced to tools such as Fluid to make these broad strokes or detailed manipulations.

But clearly, this linear, or perhaps more accurately, the monotonic, score is the point of departure not destination. Thus we should begin to break down and complicate this scripting environment to bring it closer to the agent toolkit that it intersects with.

First, I shall look at a few mechanisms for controlling the time-marker as it moves across the sheet. Then, as these mechanisms get more sophisticated they will lead to multiple time-markers — time-markers that are concurrent, and

markers that are under the control of some other organizing principle.

Improvisation on a time-marker sheet often takes the form of a combination of hand-executing visual elements, sweeping the time-maker, playing back previously made time-marker "scrubs" and of course, sometimes executing individual pieces of code straight from the text editor.

For example, consider a time-marker that is under procedural control, moving from time A to time B over a certain duration. How can a visual element that this marker strikes change the flow of movement? Perhaps it might try to slow down or pause until an event occurs, perhaps it might skip ahead to catch up with some other process, perhaps it might loop backwards to the beginning of some sequence while a condition has not occurred. Thus, we might consider two kinds of alterations: alterations of the speed of onward time flow, and unconnected jumps. However, these two categories obscure two other, perhaps more useful, categories: time modifications that are temporary and time modifications that are permanent.

It is hard to overestimate the need to both calculate and fix durations, to both give and receive durations, during a collaboration around a time-based work. Sheets of durations have always occurred as common language throughout all stages of development of *how long..., 22, imagery for Jeux Deux*, and even (rather illicitly since it was for a Cunningham stage work), in *Lifelike*. Even works that appear far removed from the problems of occupying a period of time have such "ballistically scored" elements: the development of *alphaWolf* would have benefitted noticeably for the addition of such a representation for handling both the large scale life cycle of the piece (a 5 minute "growing up period" of the wolves) and the small contrivances of the scene setting introduction (falling asleep and being woken up by the participants). Such "scripts", be them interactively modifiable, cut across whole action systems and are hard to bring about in piecemeal,

distributed architectures. it is appropriate to construct tools and passages of time that have more global and graphical views over the agent and its environment.

Of course, in remaining open to the interactive possibilities of the environment, such scores or scripts often don't remain ballistic for long. The importance of maintaining interactivity under a fixed duration constraint has ramifications for any process that wants to change how time flows through a work. In an area that ought to have a fixed duration its simple control of the rate of our time-marker is meaningless at best, and dangerous at worse. Rather, it is much more important to be able to, for example, slow down the apparent movement of time in such a way that it will speed up later to exactly compensate. Such non-permanent changes are vital if we are to be serious about moving away from the fixed linear score. Of course some changes are permanent — the duration of *how long...* varies by 10 seconds in performance and perhaps even 90 seconds during rehearsal because of permanent changes in the positions and durations of sections.

One time-marker control system that I have had recourse to in both *how long...* and *22* stacks two generic radial-basis channels: one controls the rate of increase of time which feeds into a permanent posting in a second channel that controls the time itself. This posting integrates these instantaneous rates to come up with a current time. Temporary pauses, speed-ups and even loops are placed as postings in this second channel — and placed with considerable weight in order to have an effect — permanent changes are expressed in terms of changing the value of the rate channel by introducing a (temporary) posting.

This approach works well for both *how long...* and *22*. In the former, there is a very minimal description of what agents get created and destroyed that is played out by a time marker. This high level sheet contains visual elements that them-

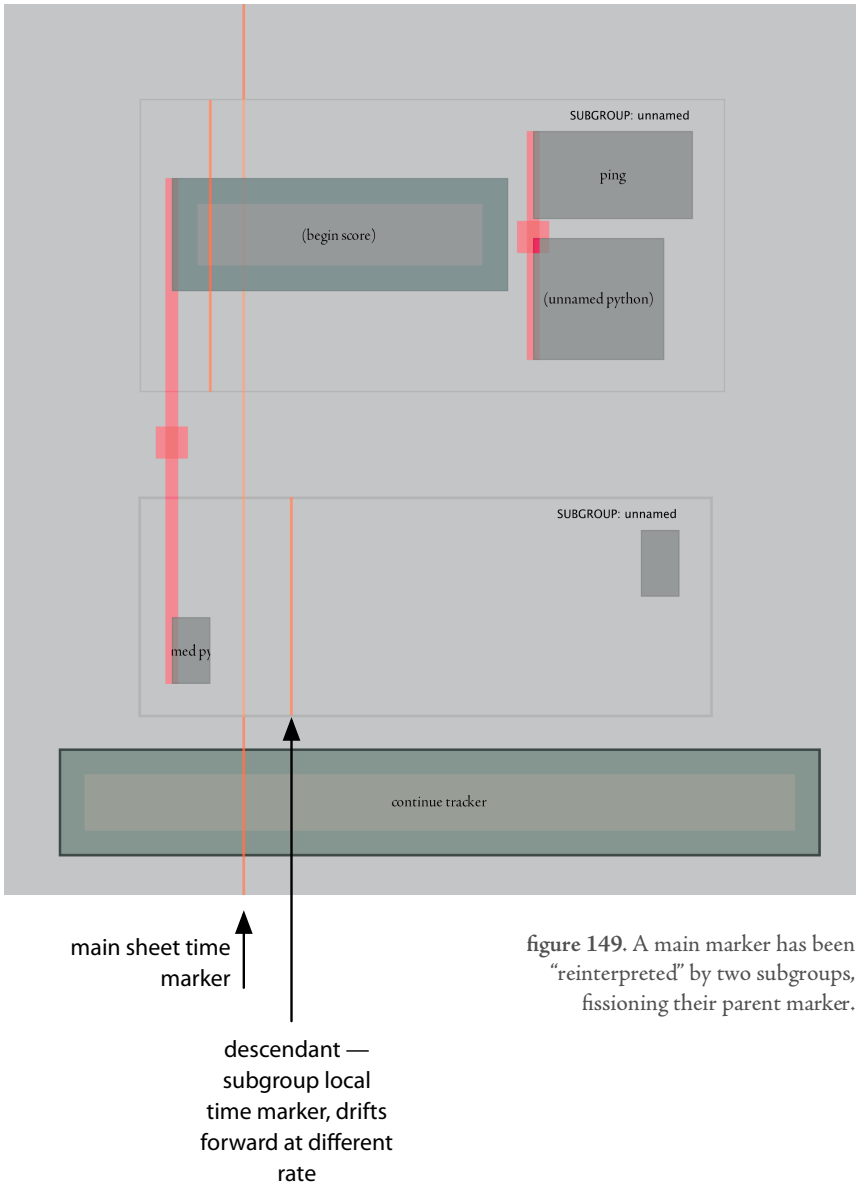

figure 149. A main marker has been "reinterpreted" by two subgroups, fissioning their parent marker.

selves contain and run sheets. In *how long...* these sub-sheets are also reasonably simple (perhaps ten elements) but at the top level, there is a single time marker controlled by this two level generic radial-basis channel. This time marker is set to move through the piece, over thirty minutes, but at several places time is temporarily paused, waiting for cues from the global choreographic tracking system; in three places it is permanently paused effectively waiting for permission to continue, and in two of these places there is a "hand cue" (one corresponding to a particularly difficult to capture rapid entrance, and one corresponding to the very end of the piece). In 22 the situation is rather more complex, because the world to be scripted is rather more complex — the piece incorporates, in addition to the main manipulation of video and geometry, textual elements, linear graphical elements and a rather complex system of cuing signals sent to the music. In this work the usefulness of a single time-marker begins to break down.

To see how Fluid might manage a move to multiple time-markers, we should first look at how visual elements control the time-marker of the sheet that they are on. There are two levels of control — a direct-drive "scripting" interface and an indirect or "deferral" specification.

The direct-drive interface is extremely direct — instantaneous Python statements have permanent or temporary effects on the time marker through two time objects — *time* and *tempTime*. The following examples should convey the directness and the usefulness of the lines of code that can be constructed using these objects.

***time*.now** = 40 — sets the time to be 40, this is a "permanent" change and there is no compensating speedup or slowdown. Behind the scenes, a instantaneous change to the rate channel causes the jump. Hence, *time*.now = startOf("beginning") — goes back to the start of the visual element called "beginning"; *time*.now = *time*.now -40 — jumps back 40 units.

***tempTime*.now** = 40 — sets the time to be 40, this is a "temporary" change, made by adding a posting to the time position marker that lasts, by default around 10 seconds with weight 1. For finer control: *tempTime*.now = {*to*: 40, *duration*:5, *weight*:100, *bias*:1} — sets the time to 40, for around 5 seconds, with weight 100, with a window strongly biased towards the start, giving a percussive jump to the beginning and a long "ease-out". Hence, *tempTime*.now = {*to*:*time*.now-40, *weight*: lambda *t* : *t**100} — jumps back 40 with a channel window function that gets stronger as time goes on.

***time*.line** = {*to*:40, *over*:10} — animates time from wherever the marker currently happens to be, to 40 over 10 seconds. This is, again, a permanent change and is actually performed by modifying the rate channel. Hence: *time*.now=0; *time*.line = {*to*:40, *over*:10, *weight*:0.5} — jumps to the very edge of the sheet and begins to move forward to 40. *tempTime*.line can perform a similar end by writing to the position channel.

***time*.rate** = *time*.rate/2 — a (temporary) slowdown in the rate of time propagation that results in a permeant modification of duration. Finer control similar to *tempTime*: for example, *time*.rate = {*to*: *time*.rate/2, *duration*:5, *weight*:10, *bias*:1}. This is achieved by posting to the rate channel. Likewise, *tempTime*.rate.

Both *time* and *tempTime* are ideal for scrambling around in an improvisation, helping one construct and blend loops of time by executing one line statements by hand out of a pre-organized visual element's text editor. And, of course, there is nothing to stop visual elements from incorporating these statements inside their offered executable functions — in fact, this is predominantly how the score-follower and the hand triggers from the performer are integrated with the flow of time through *imagery for Jeux Deux*. However, we can combine the above primitives to offer a more goal-directed manipulation of the flow of time through the sheet based on discrete triggering events.

This simplest, general-purpose interface to an event supported directly by Fluid looks like:

```
interface DeferSpecification {

        DeferSpecification begin(double time);
        double passed(double time);
        void end(double time);
}
```

begin(...) informs the specification that the are about to start listening; passed(...) queries whether (1) or not (<1) the event has taken place, and is monotonic — that is, once passed(...) a specification is never not passed(...); and end(...) informs the object that the caller is no longer willing to wait. Additionally, begin(...) has the opportunity to return the specification that will be used for subsequent passed(...) and end(...) calls. Consider the case of a *next_floorwork* event, that is passed(...) when all of the dancers simultaneously go to the ground. This may happen more than once in a work — hence we register our interest in the next such event using begin(...). This returns the object that we will query for passed(...) — allowing us to pass DeferSpecifications around globally, without passing around the means to create them afresh should we want some event's passing to remain local.

DeferSpecifications are composable using (fuzzy) boolean operators — specifications are commonly composed with an "or" operator with a time-out which limits how long a piece of code will wait. Inside the co-routine / resource framework we can bridge DeferSpecifications with standard co-routines by converting an increasing passed(...) to *progress*, a constant passed(...) to *continue*, a passed(...) = 1 to *stop* and a violation of monotonicity as *failure*, *page 140*. This bridge, of course, can be traversed in both directions.

DeferSpecifications allow the creation of higher-level time-manipulation primi-

For example, the basic set of fuzzy operations defined in : **K.Tanaka,** *An Introduction to Fuzzy Logic for Practical Applications,* Springer, 1996.

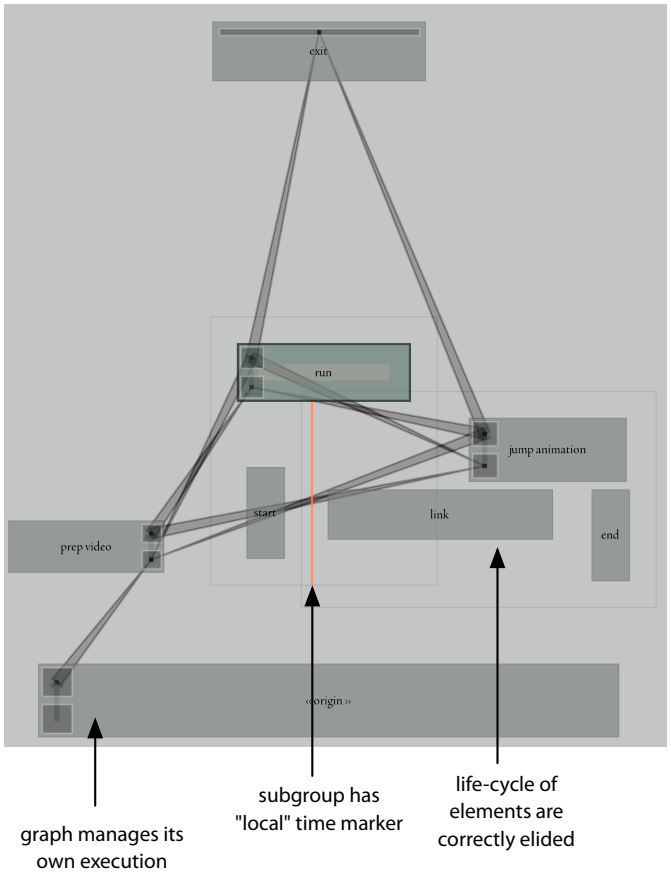

figure 150. A "graph selection" sheet. Time-markers can be local to particular visual elements — here a marker is local to the area underneath a visual element. Deactivations and subsequent reactivations between markers are elided. Note the different, but still meaningful, interpretation of the visual layout in this sheet.

tives. Most simply, we might pause at the start of a visual element until a specification has passed. It is most convenient to revisit the return-value of a visual element with respect to a Runner and write:

r = (*start_executable, continue_executable, end_executable*)

r = defer_until(*r, specification*)

The defer_* family of return-value decorators also understand 5-tuple as well as 3-tuple r values:

r = (*start_executable, while_waiting_executable, transition_executable, continue_executable, end_executable*)

where the first is executed immediately, the second while the specification has not passed and the third as the transition from not-passed to passed.

More commonly used is a softer-pause:

r = defer_until(*r, specification , fraction, smoothness, permanence*)

this works the generic radial-basis channel structure a little harder, slowing down to arrive at a time that is "fraction" through this visual element when the specification becomes passed, but in any case never going beyond this fraction; the remaining parameters control the fading in and fading out of the window functions on the postings that achieve this, and how the control between permanent and temporary channels is partitioned out. Finally, we have:

r = defer_forever(*r, specification*)

that runs the underlying visual element should "specification" pass regardless of whether this visual element is still executing or not (in which case, it runs the "start-executable", "continue-executable" and "end-executable" in three execution cycles).

Each of these mechanisms — the defer_* method"decorators" and the *time* and *tempTime* direct objects are manipulating the time-marker for the whole sheet — the shared generic radial-basis channels ensure that competing ideas as to what this time should be are blended and faded in and out. However, the impact of changing the time is shared throughout the sheet — there is only one time-marker.

To allow two or more threads of action to drift in and out of sync in response to events would require the use of multiple sheets — which seems a less than perfect solution for it potentially deletes the spatial relationship (the "sync") which grounds both visual layouts. However, within in the Diagram framework a better solution takes almost no effort to implement — we make the generic radial-basis channels for rate and time-position have context-local storage (for postings) where the context is given by the sheet grouping. Now we have an *ad hoc* but hierarchical structure in which to place postings, and whenever we manipulate the rate and position channels we get to choose at which level to place the posting — local to the visual element, local to any group that the visual element is in, or "local" to the sheet (i.e. global).

At present the grouping visual elements make this choice for their children, by interposing a text preamble and post-amble to the code of their children that causes their access to the time and rate channels to be at the group level. All other access is, by default, at the sheet level. This is the fundamental work that allows the reschedulable notations of the *parachute* / *accumulation* agents of *how long...*, the to-ing and fro-ing of *forest fire* and *stage machine* and the rhythmic traps of *22*'s scrubbing through video.

Now that we have the techniques required to fission and fuse time markers back together again, we can go further and build from time markers alternative ways of"executing" a sheet. The most developed are the graph-based structures.

Inspired by the general usefulness of the pose-graph motor system and the task of creating a visualizer that allowed the visual assembly of pose-graphs from animation materials, we can create an executable graph structure using Fluid.

Of course, creating a directed, cyclic graph of connected visual elements is already supported in Fluid, *page 379*, so there is little visual programming work that needs to be done. However, we can form a graph Runner, by extending the time-marker Runner. This Runner moves through a graph structure and executes the contents of the visual elements that it encounters. The visual element return value *r* structure remains similar — scalars, lists or dictionaries of functions, methods or Updateables — but is extended with a graph return value *g* which is a dictionary of attributes that informs the graph runner that is executing the code how to act. At present there are three keys in this dictionary:

duration — how long should this visual element execute for (in seconds); This value may be changed during the execution, but only when we get to the"end" of the visual element do the other keys become significant. *duration* controls the rate radial-basis channel in a two-level time-control similar to those used in time markers.

next — this refers to the visual element that we should go to next: it can be a visual element itself, the a name of an element or a regular expression over the name of the elements. It can also be the special elements *_nowhere* (to stop the runner completely), *_top* (refers to the spatially highest visual element connected to this element), *_bottom* (likewise). next may also be a dictionary that maps any of these elements to floating point values, in which case it is used to create an un-normalized probability distribution which is sampled from to create the next visual element.

fork — an optional list of alternative "*nexts*" that causes the graph runner to fork a copy of itself. By default these copies do (unlike time markers) re-

execute the visual elements that they encounter — that is, they do not share a common parent (see *page 376*).

Even without the forking paths and the probabilistic *next* selection this structure is enough to visually create a pose-graph motor system from coarse-grained animations. To more fully exploit the visual potential of Fluid, in particular in pose-graph like domains, we visualize the sweep of time across the graph visual element using a time-marker with a conventional runner, separate from the graph runner, local to the area beneath the visual element and sharing a common parent with all of the other local time markers.

This allows the visual creation not just of a pose-graph motor system, but of procedural processes on top of the pose-graph structure — that overlap, for example, with the beginnings and endings of animations. It is work for the future to try these ideas on a representational character, but it is my suspicion that Fluid will compare favorably with the recent interest in graph-based animation editing engines.

All three of the main modeling and animation packages now have some kind of transition graph based character animation editor:

AliasWavefront's, Maya — http://www.aw.com . Autodesk / Discreet, Character Studio — http://www.discreet.com. SoftImage's XSI — http://www.softimage.com/products/behavior/v2/default.asp

While these are innovations in their product domains, it is clear that the user interface is still catching up with the expressive power of even computer game industries, and little of the substantially more advanced motion editing algorithms of the last 5 years worth of Siggraph have made it into the products yet, nor have the interfaces for motion editing reaches the level of programmability taken for granted in other software domains.

With the forking runners and probabilistic *next* selection these extras it becomes possible to visually create, manipulate and improvise with the stochastic rhythmic cell generators of *Loops Score* and *how long....*

closing remarks

Fluid is a tool that has been made available to others. Indeed, if it were not for the urge to generalize before specifying that bore it, it would be a completely specific, personal tool. Nor does it, as a work of engineering, independent of the agent-toolkit (and its lower-level graphics rendering, sound systems and network resources) offer anything sellable to the "non-programming" digital artist. Nor does it offer a segmentation of its structure into "modules" that are easily

shared as currency between members of a "community". It seems to have few of the attributes of Max's social success.

However, while it does not compete with these tools, yet, I believe that part of its contribution is in that arena. That Fluid's database-like handling of its use-history, its self reflexive monitoring capabilities, the way in which it enables the structuring of an environment to peer into the workings of a complex system, even the length to which it can stretch a dynamic language's syntax toward terse domain specific tasks all have something to offer this domain, even in the absence of any widespread acceptance of the expressive need for text-based programming. It seems that should Fluid be augmented with a more extensive, Python-based library fitting a problem domain shaped a little more traditionally it would be a short and productive matter to turn this environment into something that could be productively used by other people. It does share some of the hallmarks of a truly learnable environment — it is non-modal, it reveals its flexibility gradually, it is itself open to inspection; and it is certainly adaptable to areas smaller than manipulating a full blown agent toolkit. At the same time however there are some aspects that would need to be reconstructed or strengthened: the development of additional layers is something that ought to be achievable in Fluid directly, without recourse to the underlying toolkit; the version control system should be expanded to handle multiple simultaneous authorship on independent computers. In general Fluid could benefit from being re-architected in such a way that it can truly turned upon itself and made even more independent of the underlying toolkit. Yet in general, turning Fluid into a widely used tool is at least a less complex task than turning the agent toolkit into a widely used architecture and perhaps this will offer an intermediate point in the distribution of the engineering contributions of this thesis.

But part of Fluid's contribution might be to question whether this arena — the single available art tool — should continue to be structured as it has been. Fluid

would surely add more to the debate around the technical and conceptual bases for digital art making if there were in fact a debate raging.

With its acceptance of the expressive power of text-based programming, Fluid keeps company with what might be an emerging counter-trend toward the text-based — as evidenced by the languages Processing / Drawing By Numbers / Supercollider. Yet at the same time as these environments reject the problems and half-solutions of visual programming, they also reject the entire visual interface and run the danger of finding themselves ignorant of an emerging counter-trend in textual programming languages — one towards multiple, semi-textual views onto a program. These art-making environments are all in danger as they grow of falling between the "professional" environments for large text-based programming projects — which have more support and manpower behind them — and the"easy-to-learn" visual environments. Fluid points toward a new class of hybrid environment and a path out of this rapidly shrinking space.