kinetext: Concrete-Programming Paradigm for Animated Typography

by Chloe Ming-shu Chao

B.A., Computer Science and Visual and Environmental Studies
Harvard University, 1996

Submitted to the Program in Media Arts and Sciences
in partial fulfillment of the requirements for the degree of
Master of Science in Media Arts and Sciences
at the Massachusetts Institute of Technology
June 1998

©1998 Massachusetts Institute of Technology. All rights reserved.

Signature of Author
Program in Media Arts and Sciences
May 8, 1998

Certified by
John Maeda
Assistant Professor of Design and Computation
MIT Media Laboratory
Thesis Advisor

Accepted by
Stephen A. Benton
Chairperson
Departmental Committee on Graduate Students
Program in Media Arts and Sciences

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUN 1 9 1998
LIBRARIES
Rotch

kinetext: Concrete-Programming Paradigm for Animated Typography

by Chloe Ming-shu Chao

B.A., Computer Science and Visual and Environmental Studies
Harvard University, 1996

Submitted to the Program in Media Arts and Sciences
on May 8, 1998 in partial fulfillment of the requirements for the degree of
Master of Science in Media Arts and Sciences
at the Massachusetts Institute of Technology

ABSTRACT

kinetext is a programming sketchbook environment for animated text design. Authoring any kind of animation involves a series of smaller decisions on movement, timing, and interplay of visual subject elements throughout the design process. While many authoring tools support this process, they fail to document the design process in such a way for others to easily discern the designer's decisions. Frequently the only artifacts are lengthy text programs or similarly oblique scores. *kinetext* presents an environment where authoring animation occurs via small visual programs. Each program illustrates each of the individual transformations responsible for the motion of each subject element in the animation. Viewed in its entirety, the environment becomes a sketchbook of the design process involved in bringing about the final animated piece. The visual nature of the system is inspired by a spatially structured type of poetry known as "concrete poetry," where the arrangement of words take on form. Hence one can refer to the system as a form of "concrete-programming."

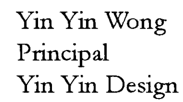
Thesis Supervisor: John Maeda
Title: Assistant Professor of Design and Computation

kinetext: Concrete-Programming Paradigm for Animated Typography

by Chloe M. Chao

The following people have served as readers for this thesis:

William J. Mitchell
Dean of the School of Architecture and Planning
Massachusetts Institute of Technology



Yin Yin Wong
Principal
Yin Yin Design

Acknowledgments

I wish to thank my advisor, John Maeda, for all the insight and wisdom he has instilled in me over these past two years.

I would also like to thank my readers, Yin Yin Wong and Bill Mitchell, for taking the time to review drafts and offer suggestions over the course of writing this thesis. I am especially grateful to Yin Yin for being an exceptionally devoted reviewer despite the fact we were on opposite coasts.

I am honored to have been a member of the first generation of the Aesthetics and Computation Group and will sincerely miss everyone. Special thanks go to dsmall, grenby, pcho, tom, and kram for making these past two years seem fun despite the hard work and late nights.

I wish to thank Sandy Pentland, for giving me a UROP position here at the lab three years ago and then becoming my co-advisor when I began my graduate studies the following year. I will sincerely miss the time I spent in Vismod and all my friends there.

Other people to thank include Tara Rosenberger, for being a wonderful office-mate and helping me keep perspective on life, and Ken Russell for kindly taking the time to proof-read a final draft of this thesis.

I would also like to thank the furry ones in my life for happily distracting me at times of great stress and worry during the past two years. I wish to thank Poopy the Hamster, now gone, but who forever will represent the softer side of the Lab. I would also like to thank Wulfie G., for his tireless rounds of wulfie-bowling.

As always, I wish to thank my family for being who and what they are. I want to thank my brothers, mom and dad for all the love and support they have given me and continue to give me in life.

To my grandparents who passed away this last year, I love you and miss you both.

TABLE OF CONTENTS

ABSTRACT 2

ACKNOWLEDGMENTS 4

1 INTRODUCTION 8

1.1 Motivation 8
1.2 Objectives 9
1.3 Research Issues 9
1.4 Accomplishments 10
1.5 Outline of the Thesis 10

2 RELATED WORK 11

2.1 Digital Typography and User Interfaces 11
2.2 Visual-Programming of Graphics 14
2.3 3D Environments for Programming 16

3 BACKGROUND 20

3.1 Why Typography Suits Computation 20
3.2 What is Concrete Poetry 23
3.3 The Concrete-Programming Paradigm 24

4 DESCRIPTION OF THE SYSTEM 26

4.1 Typographic Operators and Clusters 26
4.2 Defining New Typographic Operators 27
4.3 The Workspace and Time 28
4.4 The Crosshair 30
4.5 Implementation Notes 30
4.6 Program Architecture 30
4.7 The Output 31

5 VISUAL DESIGN OF THE SYSTEM 36

5.1 The Evolution of a Visual-Programming Environment 36
5.1.1 Visual Machines 36
5.1.2 Typographic Operators 37
5.2 Analyzing the Final Forms that Define kinetext 38
5.2.1 Bowed Planes 39
5.2.2 Staggering the Space 39
5.2.3 Webbed Clusters 40
5.2.4 Color and Contrast 41
5.2.5 Why Grids and Not Solid Planes 42
5.2.6 Animation: Why the Stage Border 43
5.2.7 Visual Appearance of Typographic Operators 43

6 AUTHORING TEXT ANIMATIONS WITH KINETEXT 46
6.1 Animation 1: The sentence 46
6.2 Animation 2: The poem 48
6.3 Animation 3: RSVP 51
6.4 Observations and Critique 52
6.4.1 Observations on Menus and Windows 52
6.4.2 Critique for Movable Clusters 53
6.4.3 Critique on Navigating the Space 53
6.4.4 Observations on Typographic Parameters 54

7 ANALYSIS OF METHOD AND PROCESS: A COMPARISON AMONG AUTHORING SYSTEMS 55
7.1 Macromedia Director 6.0 55
7.1.1 Observations on Director 6.0 56
7.1.2 Comparing Director 6.0 and kinetext 58
7.2 Side Effects Houdini 2.0 59
7.2.1 Observations on Houdini 2.0 60
7.2.2 Houdini's Visual-Programming Paradigm 61
7.2.3 Comparing Houdini 2.0 and kinetext 61
7.3 Results from the Authoring Experience 62

8 CONCLUDING REMARKS 64

8.1 Summary 64
8.2 Future Work 65
8.2.1 Legibility and Collaboration 65
8.2.2 Extending the Concrete-Programming Paradigm 66

APPENDIX A - USER MANUAL 68

APPENDIX B - AUTHORING A SIMPLE ANIMATION WITH KINETEXT 70

BIBLIOGRAPHY AND REFERENCES 73

READERS 75

CHAPTER 1

INTRODUCTION

kinetext introduces a visual-scripting system by embedding typographical elements with visual characteristics. In this way, the user can observe what visual effect would take place if they choose to apply the characteristic to other typographic elements. By removing the layer of abstraction that a separate scripting environment brings to the conventional scripting systems, it is apparent that animation scripting itself can become an expressive typographic piece literally illustrating the process to the final animation. In addition to authoring, editing the animation becomes much clearer as the user can observe individual text transformations by examining the members of a clustering structure, as well as the combined effect of all the members on whatever word the cluster is dragged over.

1.1 Motivation

The motivation behind the concrete-programming paradigm and the workspace in general, is a desire to create an authoring environment that can both coexist with the final finished piece, and also serve as a descriptive visual record of the design process.

kinetext is the result of research into the authoring process for dynamic typography in conjunction with computational design philosophy. Whereas the philosophy of conventional tools is to have the designer's intent, the tool, and the result exist as separate entities in the design process (Figure 1.1), computational design philosophy creates a place where the tool and the result can coexist in the design process (Figure 1.2). This coexistence model then fosters an environment where intent can emerge. In other words, an environment such as *kinetext,* enables an observer to see and understand the process from whence the result comes. In contrast, the conventional tool model separates the process from the result in such a way that an observer cannot necessarily see how the designer arrives at the result.

Figure 1.1: The conventional tool model.

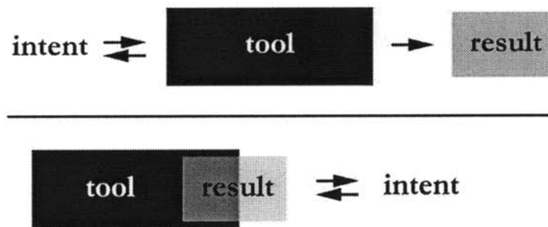


Figure 1.2: Computational medium supporting creative thought and observation.

1.2 Objectives

The goal of this system is to introduce a new paradigm to visual-programming that provides the designer with a typographic authoring environment that is able to illustrate the design process. Such an environment incorporates typographic operators with embedded visual characteristics and enables construction of programs that can visually convey their functionality.

1.3 Research Issues

The research addresses the design of a concrete-programming environment for authoring typographic animation. Hence the issues raised by the *kinetext* project include:

1. Concrete-programming as a visual paradigm for visual-programming
2. The construction of an animation system supported by concrete-programming.

Currently, visual paradigms for visual-programming are largely based on representing computational elements with geometric shapes or icons. This thesis explores replacing such abstract, and sometimes arbitrary, forms with the clear letter-forms provided by typography.

In addition, this thesis evaluates *kinetext's* viability as a concrete-programming based animation environment by assessing it alongside two current commercial animation systems. The evaluation addresses how evolution of such systems has come to affect the way designers construct and observe the process behind digital animation.

1.4 Accomplishments

Results from the *kinetext* project include:

1. Software implementation of a concrete-programming environment for authoring animated text.
2. Several sample animations created in the *kinetext* environment.
3. A secondary scripting language used for saving output from the *kinetext* environment and creating optional input.

In addition to these digital artifacts, the *kinetext* project also promotes the illustration and subsequent examination of the authoring process for animation. *kinetext* advances the concept of software providing a *computational medium* as opposed to isolated tools (Figure 1.2).

1.5 Outline of the Thesis

Chapter 2 reviews past and present related work. Chapter 3 reveals the reasoning behind the use of typography and concrete poetry for a visual-programming paradigm. Chapter 4 details the organization and implementation of *kinetext's* environment. Chapter 5 recounts the evolution of the visual display and use of space by the environment. Chapter 6 presents observations from composing three different types of animated pieces with *kinetext*, and a resulting critique of the system by the author. Chapter 7 examines two current animation systems, the authoring styles supported by each, how each illustrates the design process, and how they compare to *kinetext*. Chapter 8 concludes with future ideas and possibilities for the *kinetext* project.

CHAPTER 2

RELATED WORK

Much of the work related to the *kinetext* project lies within three areas: creation of animated typography, visual-programming paradigms for graphics, and 3D environments for programming.

2.1 Digital Typography and User Interfaces

If typography can be described as the visual treatment of written language [Wong, 1995], then it should follow that **digital typography** embodies the computer's representation of written language.

For a long time, digital typography largely remained static in its presentation. As with any new medium, people sought a way to familiarize the new with the old. Digital typographers drew on the familiarity of the letterpressed word and were naturally more concerned with perfecting the letterforms and systems of letter placement than exploring the temporal qualities the digital medium offered. Only with the advent of digital animation systems has digital typography become temporal. Today, we can find important paradigms in the field of digital typographic animation in typographic pieces authored by Small and Wong [Minsky Melodies, 1996] and Soo's scripting language for temporal typography [Soo, 1997].

Wong defines a framework for designers to consider when creating temporal typography [Wong, 1995]. Based on this framework, Soo developed a scripting system that allowed authors to create complex temporal typographic compositions. Together, Small and Wong used Soo's scripting language to generate the digital typographic animation presented in **Minsky Melodies**. Minsky Melodies is a typographic animation designed to accompany a segment of opera music. Words either directly display the lyrics of the opera or serve as a counterpoint to enhance the lyrics of the opera (e.g., having letters flying about like scattered bits while the lyrics are "pieces of brain"). According to Small the key to creating such a lengthy piece of digital typography (~6.5 minutes) without obvious repetition was being

able to reuse previous scripts by varying a few of the parameters or building upon them. However, Small and Wong also point out that once the scripts were written, it was difficult to go back and read through them to recall what visual effects they produced. *kinetext* seeks to address this problem by employing a visual-programming paradigm to heighten legibility of the programs responsible for visual effects.

Figure 2.1: Image of dynamic text from Minsky Melodies.

Other examples of noteworthy contemporary animated typographic work can be found in movie title sequences done by graphic designer **Kyle Cooper**. Cooper's work on opening title sequences in movies like *Seven (1995)*, with warped images and flickering text, revitalized the use of text in opening sequences to express the mood and symbolize the theme of the movie. Other titles Cooper has designed include *The Island of Dr. Moreau (1996), Twister (1996), Mimic (1997)* and *Lost in Space (1998)*.

Figure 2.2: An image from the opening title sequence of Seven.

Cooper's work is largely influenced by that of graphic designer, **Saul Bass**. In the 1950s, Bass pioneered the use of animation techniques and animated typography in movie title sequences. He revolutionized what had up until then been the conventional, straight text, opening credits by introducing broken text and bold dynamic shapes that summarized and symbolized themes for the movies they represented. Among the works best known for that style are *Vertigo (1958), Psycho (1960),* and *North by Northwest (1959)*.

Figure 2.3: Series of images from title sequence for Psycho.

Related work in the field of user interface design for type animation can be found in commercial tool packages like **Macromedia Director** [Macromedia Inc., 1996] and **Adobe After Effects** [Adobe Systems Incorporated, 1995]. Director and After Effects offer time-based authoring tools for creating animation. Their systems involve extensive menus and multiple editing windows: one for timing, one for holding all the subject elements, and one for placing subject elements for display in the animation. Through their popularity and widespread use, these software packages have set certain standards in user interface design for typographic animation applications that fit into the conventional tool philosophy. *kinetext* begins to explore interface possibilities outside of this standard model, with the goal of integrating all aspects of the authoring process in one space so the designer can leave both artifact and the description of the process behind the artifact for future viewers, and future reference.

In addition, the **Flying Letters** [Maeda, 1996] reactive book is an important work in addressing typographic user interface. Flying Letters contains a series of interactive typographic experiments whose interfaces are simple and intuitive such that no instructions are required. In one such experiment, the user simply moves the mouse over a clear, black screen leading a trail of changing white letters that fade out over time. Such seamlessness of interface and transparency of mapping input to output is ultimately the aim of any digital environment seeking to support a process as fluid as design.

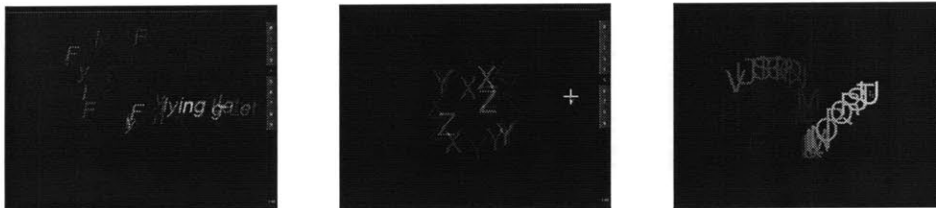


Figure 2.4: Flying Letters.

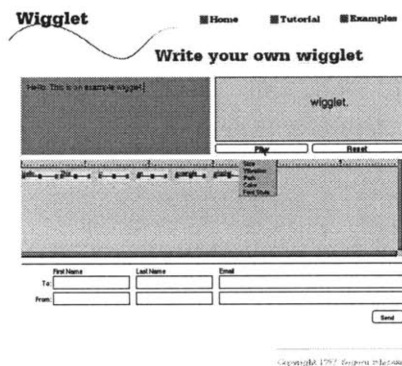


Figure 2.5: The Wigglet interface.

Among the most recent works in typographic user interfaces are **Wigglet** [Ishizaki, 1997], and **Cornix** [Tenax Software, 1997] rapid-serial-visual-presentation (RSVP) system. Cornix is a simple applet that allows the user to enter text that can be rapidly displayed one word at a time. The more extensive of the two systems, Ishizaki's Wigglet applet offers a way to author animated-2D-typographic e-mail messages. Ishizaki's system does employ some visual aids for authoring the text animation (mainly through the use of time bars and text placement), but for the most part Wigglet relies heavily on menus and windows. Our system differs from Wigglet in that *kinetext* addresses 3D authoring issues and employs a visual-programming paradigm without menus and fixed windows.

2.2 Visual-Programming of Graphics

At present, there are a variety of visual-programming languages or paradigms for almost every kind of computation and so it should be no different when it comes to visual-programming of computer graphics. With the advent of 3D graphics libraries and programming software, it was inevitable that visual-programming interfaces arose to offer new ways of programming for 3D.

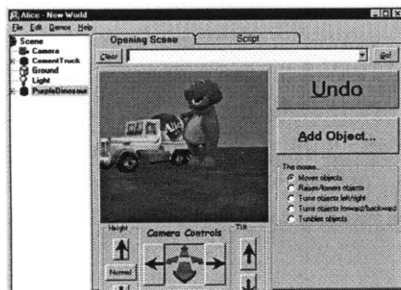
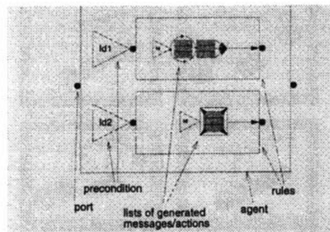
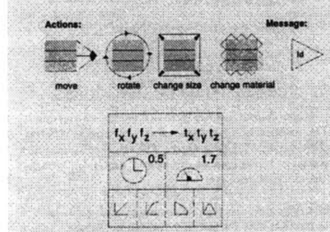


Figure 2.6: Screen-shot of Alice.

Alice [UVA User Interface Group, 1995] is a visual-programming environment aimed toward enabling novice programmers to rapidly create 3D interactive graphics. The scene graph appears as a series of nodes (as 2D icons) in a window that can be arranged and reconnected. The viewport allows for direct manipulation of camera and objects.

AAL-VL [Duecker, et. al., 1997] is a diagrammatic visual language for programming the Animated Agent Layer (AAL). The AAL enables animated objects to become intelligent objects, aware of their surroundings and able to autonomously determine their course of action in the environment. AAL-VL employs an icon-based representation of data and allows the user to create program/flow diagrams. Currently, *kinetext's* intelligence is largely based on the parsing and collision detection capabilities of the environment; the animated text clusters themselves have little to no sensing capabilities (Section 4.6). If *kinetext* were to adopt the idea behind AAL-VL, and have typographic operators and clustering structures become autonomous, the environment can become potentially much more flexible.





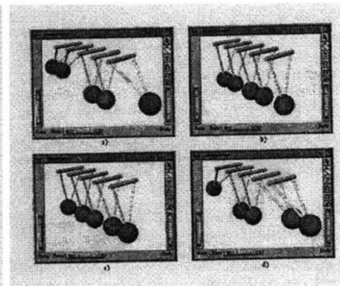
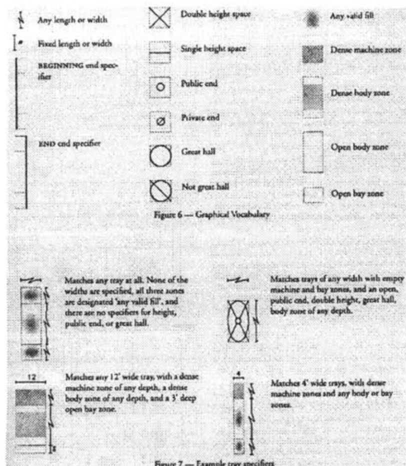


Figure 2.7: The left figure shows the icons used for visual programming of AAL. The figure above shows a 3D animation produced by AAL-VL.

The LaHave House Project [Rau-Chaplin and Smedley, 1997], is a visual-programming language composed of iconographic rules for architecture. The grammar of icons they have developed are based on a particular style of architecture and can be used to construct a large variety of different sequences to generate different models within the same style (Figure 2.8). This concept of having a grammar for a particular style of design is a definitive step in addressing the role of automation in design. The architect still plays a large role in the design process, but the computer now can offer extensive possibilities and combinations of the structure models for the architect to consider during the design process, instead of merely serving as an implementation tool.



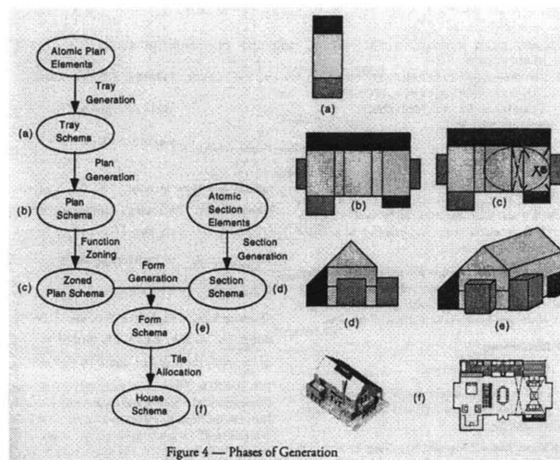


Figure 2.8: The figure on the left shows the icons used to describe the grammar developed for the architectural style of the LaHave House project, The diagrams on the right show a sample flow model of the phases of generation from a sequence constructed with the grammar and the resulting geometries produced.

The **Houdini** 3D animation system [Side Effects Software Inc., 1997] is one of the more powerful animation systems currently available. Although it employs what is easily recognizable as a visual-programming paradigm, Houdini represents a new way of authoring for much of the commercial animation community. It adopts a 2D procedural flowchart-like approach to animation and allows for non-linear authoring of 3D animation. Houdini successfully integrates a directed-graph paradigm where procedures and modeling objects alike are linked together and are directly editable. Further discussion of this system and how it relates to *kinetext* can be found in Chapter 7.

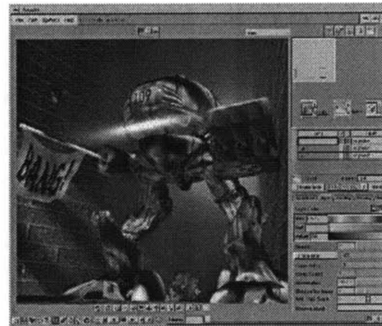


Figure 2.9: Screen-shot of Houdini.

2.3 3D Environments for Programming

While current research and development has produced many applications and environments supporting 3D graphics and animation, the great bulk of programming for such environments and applications is still done in 2D. More recently,

there has been increased interest in programming environments that explore a 3D interface for manipulating geometries and program components.

Inventor GraphViewer [Open Inventor, 1994] visualizes the nodes and corresponding hierarchies of a 3D scene-graph as spheres, cubes and other 3D shapes. The user can click on a node and have a window pop up allowing properties of that node to be changed. Animation is created through engine nodes that accept time arguments and in turn affect properties of attached nodes. Despite the 3D nature of node representation, the organization of the nodes (on the right side of the figure above) are in 2D and hence, the programming structures really only require a 2D representation. But even so, this initial mapping of programming structures to 3D introduced the possibilities of using 3D for programming and not merely viewing.

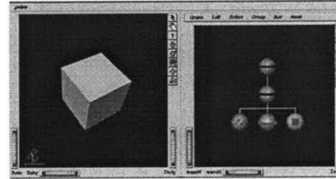


Figure 2.10: The right window shows the programming-nodes arranged in a tree-like format. The left window shows the resulting model in the viewport.

CAEL-3D, Computer Animation Environment Language [Van Reeth et al., 1995], is a menu-based, 3D graphical programming environment. All programming structures are given a visual representation within the 3D environment. The figure to the right displays CAEL-3D's representation of the Fibonacci function. CAEL-3D explores different aspects of 3D interaction for a 3D programming space. The choice to employ menus is based on the desire to take advantage of the tools readily available to any 3D modeler. Picking and navigation are performed in the 3D viewing window. In addition the user has the option of viewing the environment with a stereoscopic display to enhance the 3D experience. Despite such efforts, much like GraphViewer, CAEL-3D's programming structures and representations are largely 2D icons given some depth. There is no real use of depth in the programming space.

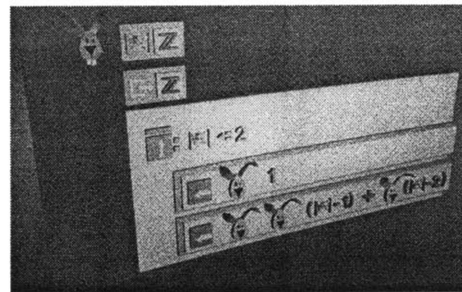


Figure 2.11: Fibonacci function in CAEL-3D.

SKETCH [Zeleznik *et. al.,* 1996], is an interface that employs gesture-based input to rapidly create and edit 3D scenes and geometries. It achieves a pencil-and-paper

sketching-like interaction that eliminates the overhead that coding such geometries would require. SKETCH begins to map an analog design process to a digital 3D canvas, and is successful in doing so. In addition, the crux of the interface paradigm makes a giant leap from the 2D-ish methods of previous 3D programming environments. SKETCH best describes one end of the visual-authoring spectrum that *kinetext* is striving to move towards. One hopes that the future of 3D environments for programming will continue to explore such seamlessness of interface.

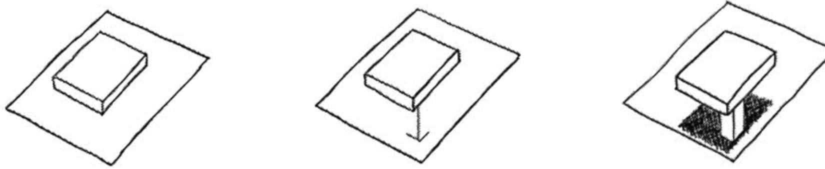


Figure 2.12: Creating a table leg in SKETCH. The user draws 3 defining lines (center picture) and the system creates the proper form with accompanying shadow so the initial shape looks elevated from the ground.

To summarize, in this chapter there was a review of three different areas of related work: animated typography, visual-programming, and 3D environments (Figure 2.13). As different as these fields seem, there is an undeniable crossover occurring in recent research and development. Any number of the works described here fall in more than one of these fields as visual-programming progresses to address animation and use of new interfaces. By addressing such work, the intent here is to cover work influential to the development of *kinetext*. The work in the *kinetext* project is primarily focused that of typographic interfaces and visual programming, with 3D playing an important visual role.

Animated Typography	Visual-Programming	3D Environments
Minsky Melodies	Alice	Inventor GraphViewer
Kyle Cooper	AAL-VL	CAEL-3D
Saul Bass	LaHave House Project	SKETCH
Macromedia Director	Side Effects Houdini	
Adobe After Effects		
Flying Letters		
Wigglet		
Cornix		

Figure 2.13: Table of related work.

CHAPTER 3

BACKGROUND

3.1 Why Typography Suits Computation

In terms of visual-programming, why have I stayed away from the usual pictograms or icons commonly employed? My decision to use words as the actual means of the visual-programming is based on a desire to avoid ambiguity. I sought to create an interface that would require as little instruction as possible. During my study of other visual-programming languages, I was always frustrated by the fact that although visual diagrams were supposed to facilitate my understanding of the computation they represented, I was still required to read through a "manual" to understand what exactly the forms in the visual diagrams were representing.

This clearly showed me that icons and other pictorial forms can be read in a variety of ways; one person's interpretation can differ from another's. Often when the author of such an iconic visual-programming language would try to illustrate a point on a slide or whiteboard of what was happening on the computer screen, they would merely compound the confusion of the audience who now had to map the hand-drawn form to a computer-drawn form to an abstract concept of computation (Figure 3.1). The conclusions I drew from such lessons lead me to explore the use of letters, which are different from almost all other forms. Letters are one of the few forms that actually transcend media.

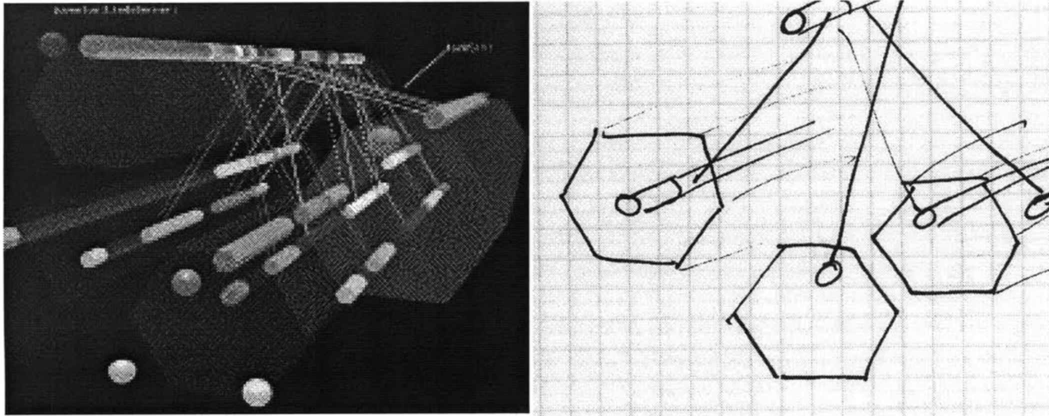


Figure 3.1: A screen-shot and a hand-drawn sketch of the visual-programming language for VisuaLinda, where spheres = processes, cylinders = process's behavior, hexagons = processors, lines = communication.

Even prior to the advent of computing, the idea that letters remained a fairly constant and unambiguous set of forms whether set down in stone or wax or on paper is evident throughout much of history. In *An Essay on Typography*, Eric Gill discusses this concept of the letter-form through the eyes of the craftsman:

> He did not say: Such & such a tool or material naturally makes or lends itself to the making of such and such forms. On the contrary, he said: Letters *are* such and such forms; therefore whatever tools & materials we have to use, we must make these forms as well as the tools and material will allow. This order of procedure has always been the one followed. The mind is the arbiter in letter forms, not the tool or the material. This is not to deny that tools and materials have had a very great influence on letter forms. But that influence has been secondary, and for the most part it has been exerted without the craftsman's conscious intention. [Gill, 1936]

That philosophy has carried over to the medium of digital letter-forms. Among the realm of spheres, cubes, and cones, we can still read letters for what they are.

As an experiment in testing the legibility of the letterform, I composed a vector-based font (line-based letterforms) with the objective of using the fewest lines I could to define a distinguishable letterform (Figure 3.2). As a further study, I went on to create an interactive application to generate animations that would gradually remove the heavy serifs from the letterforms to see at which point the forms would be illegible (Figure 3.3).

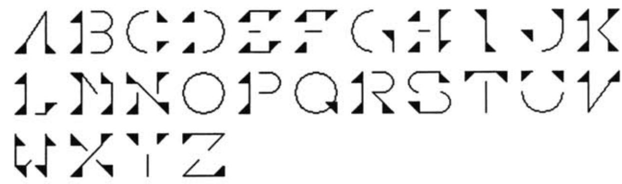


Figure 3.2: Vector-based set of minimalist letterforms.

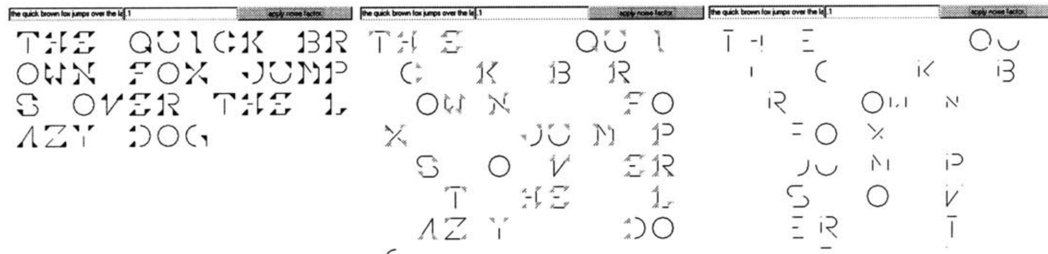
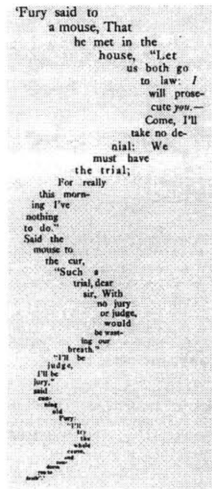


Figure 3.3: A series of images from testing the legibility boundaries of the minimalist letterforms.

One simple, user-interface observation from these short, interactive experiments was that with the use of typography comes the fact that the user actually has to type in commands. Granted it is a more time-consuming input method, once entered, there can be no ambiguity for another observer. The first instinct a literate observer has when seeing letter-forms is to read them. While abstract shapes may be more time-efficient as input, the respective abstract output may baffle the average observer, for there are far fewer universal picture-icons in our visual vocabulary than our verbal vocabulary.

3.2 What is Concrete Poetry

Figure 3.4: Some examples of picture poetry: on the left, a mouse's tail from Alice in Wonderland, and on the right, Easter Wings by George Herbert.



Lord, who createdst man in wealth and store,
Though foolishly he lost the same,
Decaying more and more,
Till he became
Most poore:
With thee
O let me rise
As larks, harmoniously,
And sing this day thy victories:
Then shall the fall further the flight in me.
My tender age in sorrow did beginne:
And still with sicknesses and shame
Thou didst so punish sinne,
That I became
Most thinne.
With thee
Let me combine,
And feel this day thy victorie:
For, if I imp my wing on thine,
Affliction shall advance the flight in me.

Concrete poetry began as a literary form in the early 1950s, an outgrowth from the work of concrete painters in the 1940s. The thought was that poetry could use placement of words in the same way painters use the placement of representational forms to convey meaning.

What concrete poetry does for typography, is introduce a new level of form that interacts with the already present letter-forms. There is an interplay of the unambiguous letter-forms being arranged into forms that can either enhance or contrast with or the communication of the letter-forms. On one end of the spectrum of what has been considered concrete poetry there is an emphasis on form-imagery. By observing the form of picture poems like Herbert's Easter Wings, or Lewis Carroll's Mouse's Tail, we are given some direct context for interpreting the words that comprise the form. At the other end, however, one of the initial definers of concrete poetry, Eugene Gomringer, speaks of the use of less pictorial forms known as *constellations*.

> The constellation, the word-group, replaces the verse. Instead of syntax it is sufficient to allow two, three or more words to achieve their full effect. They seem on the surface without interrelation and sprinkled at random by careless hand, but looked at more closely, they become the center of a field of force and define a certain scope. In finding, selecting and putting down these words (the poet) creates "thought-objects" and leave the task of association to the reader, who becomes a collaborator and, in a sense, the completer of the poem. [Gomringer, 1951]

In a way, Gomringer's constellations refer back to the celestial definition of finding associations between the stars in the night sky. Concrete poets provide the

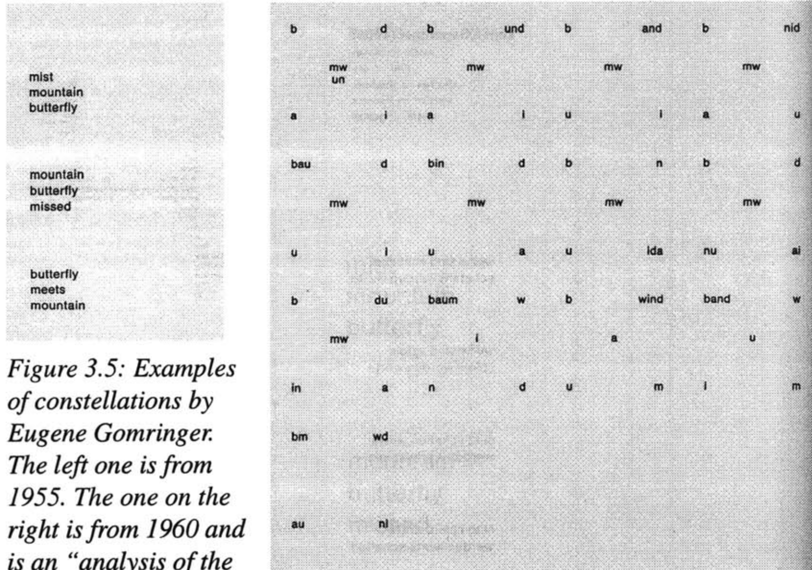


Figure 3.5: Examples of constellations by Eugene Gomringer. The left one is from 1955. The one on the right is from 1960 and is an "analysis of the words baum *(tree) and* wind *[which] yields a field of sixty-five one-, two-, three- and four-letter groups, which in turn yield many other words and associations. [Williams,*

placement and choice of words, but it is left to the reader to connect these words into forms they can readily accept, much like astronomy's constellations.

3.3 The Concrete-Programming Paradigm

The decision to pursue a concrete poetry approach toward visual-programming was made with the hopes that by arranging function words (typographic operators) into forms akin to picture poems or constellations, these forms could further communicate the function of the programs the words described.

When first designing the environment for *kinetext* it was decided that the visual representation of the programs created within should follow the concrete model of form helping to convey meaning. However, as the system developed, it was found that legibility increased when *kinetext* also affected the forms of the words in addition to arranging the words into forms. A simple example of this is comparing "scale = 2" to scale = 2. By changing the actual form of the instruction, one can imbue the operator with visual information in addition to its contextual information. In addition to visual information, one can also add temporal information. For example, when the user is setting time duration for sprites, it is difficult to

get a feel for how the timing is working until they actually play out the final animation. *kinetext* affects the form of the words in the time operator by fading out the words over the time period they are describing while the user is still within the workspace. Such visual and temporal cues are simple enough that their actions are fairly unambiguous when seen in conjunction with the function words.

While it is eminently useful to explore these different uses of form in conjunction with typography, one must also keep in mind never to emphasize form at the expense of content. One example of such can be found in "ASCII art." In these pictures composed of letters and punctuation, there is often no lexical meaning to the letter combinations and the letterforms themselves becomes little more than a pixels for the form they depict.

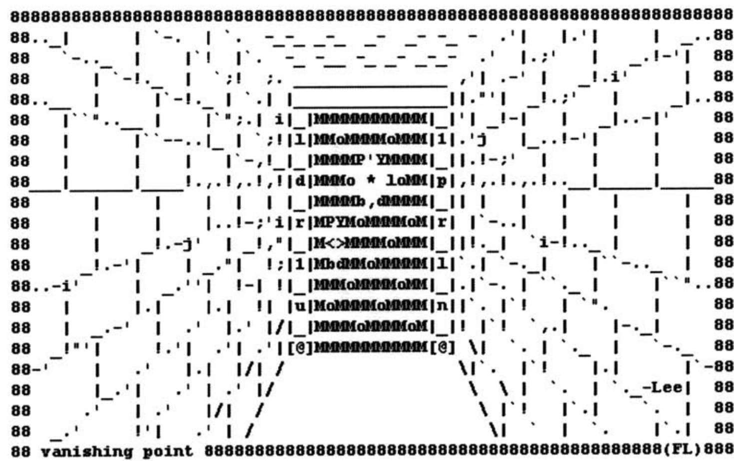


Figure 3.6: An example of form overwhelming content. Vanishing Point, by Felix Lee. ASCII art.

In summary, the decision to use typography as a basis for a visual-programming language is largely based on a desire for the clarity of form letters represent. The progression to pursue a concrete poetry paradigm for a visual-programming environment is based on the objective to use word arrangement to convey added meaning beyond the letterforms.

CHAPTER 4

DESCRIPTION OF THE SYSTEM

4.1 Typographic Operators and Clusters

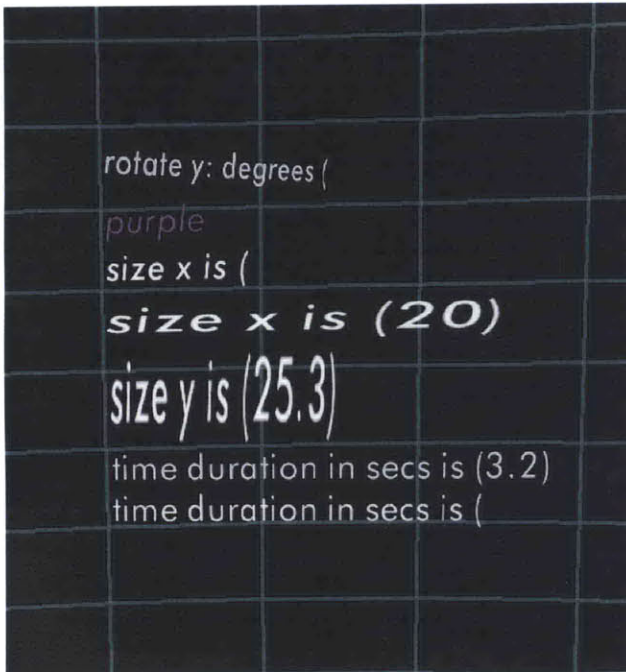


Figure 4.1: Examples of kinetext's typographic operators.

Typographic operators serve as the basic building blocks for the system. They convey their function by changing their individual forms. For example, when the user types the color word "purple," the text immediately changes its material property to reflect the color purple. Similar visual cues are embedded for function words such as "rotate" or "size" or "time" where the user is prompted for numerical arguments.

These typographic operators then are used to create programming clusters. The cluster serves as one type of form words can assume through the concrete-programming paradigm. It arranges the words into visually webbed-like form that assumes its members' characteristics, or in the case of multiple operators controlling the same characteristic (e.g., color), averages its members' characteristics (Figure 4.3). Through these clusters, the user authors different animated effects for the subject text of the final animation.

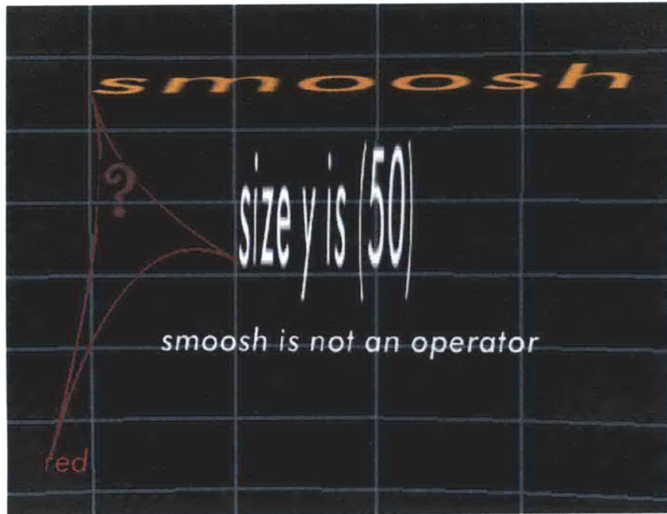


Figure 4.2: An example of a piece of subject text, 'smoosh' becoming a typographic operator for a new cluster.

kinetext also allows for an interesting duality to take place between subject text and typographic operator. Once the user has created a cluster and applied it to a piece of subject text, that newly animated subject text can then become a typographic operator for another cluster. This duality serves to expedite the creation of new, higher-level typographic operators, or macros.

4.2 Defining New Typographic Operators

If a particular effect will be appearing frequently in a final animation, the user can choose to formally define a new typographic operator such that when the string is next typed, it will automatically assume its assigned characteristics. The way macros are formally defined is by surrounding the word to be defined with braces. For example, to define a new typographic operator "bigIndigo" the user would type "{bigIndigo}" and create a cluster of characteristics to drag over the word. The act of dragging the cluster over the word transfers the combined characteristics of the cluster over to the word. From then on when the user next types "bigIndigo" the word will immediately assume the characteristics now bound to it (Figure 4.3).

In this way, users can define their own libraries of higher-level typographic operators. The libraries can be reused in future animations since the definitions are saved out in script form and can be reloaded into new animations. The importance of this arises when designers create effects they wish to reuse in later animations. This feature allows the designers to personalize *kinetext* with their own keywords.

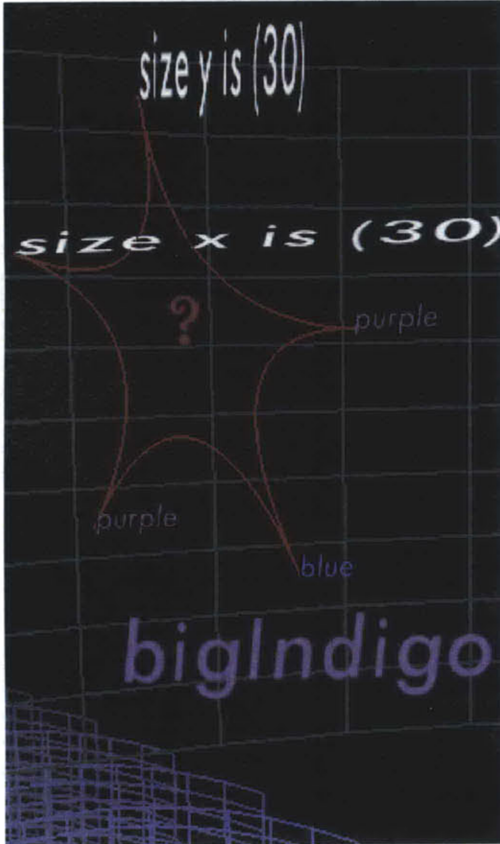


Figure 4.3: Every word that is not a typographic element begins with default characteristics. Hence an empty cluster is:

? =
scale{ 10, 10, 1 }
color{ 1.0, 1.0, 1.0 }

How the characteristics of the cluster change as each child node is added:

ADD: size y is (30) = scale{ 10, 30, 1 }
? =
scale{ 10, 30, 1 }

ADD: purple = color{ 0.3, 0.1, 0.3 }
? =
color{ 0.3, 0.1, 0.3 }

ADD: blue = color{ 0.3, 0.3, 0.8 }
? =
color{ 0.3, 0.2, 0.55 }

ADD: purple = color{ 0.3, 0.1, 0.3 }
? =
color{ 0.3, 0.17, 0.47 }

ADD: size x is (30) = scale{ 30, 10, 1 }
? =
scale{ 30, 30, 1 }

4.3 The Workspace and Time

The workspace of the 3D environment is composed of a series of workplanes where each plane represents a different point in time, much like a key frame. The default setting of the system is to create workplanes at every half-second interval, so there is a key frame every 15th frame when the animation is to run at 30 frames per second.

When the user first drags a cluster with a time characteristic over another word, copies of the word will appear on all the workplanes it traverses over time. For example, if a cluster containing "time in seconds is (2.1)" is dragged over the word "hello" at workplane "0.0 secs," copies of "hello" will appear on workplanes at

0.5, 1.0, 1.5, and 2.0 seconds. In addition, the workplane on which that word now ends (2.0 secs) will rise up to the same height as the current workplane in order to give the user a sense of the time distance the word covers (For another example, see the word "head" in Figure 4.4).

If the user changes any characteristics of the word (color, squash, stretch, rotation, or translation) on the last plane the word resides on, the system interpolates between workplanes and visually reflects the changes (See the word "sleepy" in (Figure 4.4)).

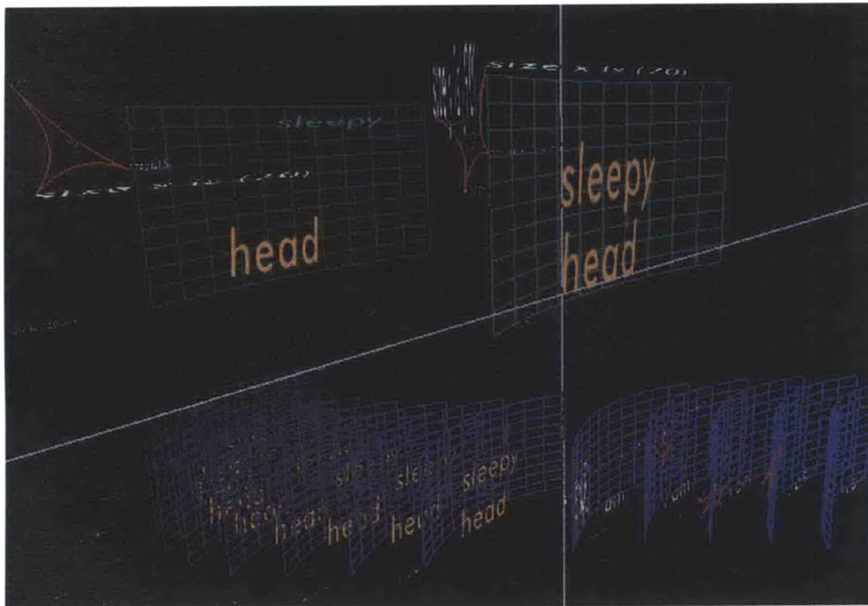


Figure 4.4: The workspace from kinetext.

One of the advantages of this workspace model that a 3D work area offers many viewpoints for observation of 3D structures. For although an application like Director [Macromedia Inc., 1996] provides for a similar time-organized authoring process, the process is absolutely flat in that the user can only ever see one frame at a time and all animation is restricted to that plane of that one frame. In contrast, *kinetext* allows the user to actually view and manipulate an animation in 3D, as well as the ability to view more than one frame at a time. The user can even flatten planes together to be able to see exactly where objects in one frame are in position with respect to another frame. This is exceptionally useful when the user wishes to align certain elements over time (Figure 6.1) or align sprites appearing at different times in the animation.

Viewed in its entirety, the workspace becomes a sketchbook of the process involved in bringing about the final animated piece. The clusters responsible for each transformation can be viewed at the corresponding workplanes. Complex transformations are discerned from simpler ones by merely viewing the clusters from afar. The environment becomes one large space of concrete poetry giving insight to the design of the final animation.

4.4 The Crosshair

Given the complexity of interface 3D introduces, there arises a need to provide the author with a constant reference point at all times. To address this, there is a 2D crosshair that constantly lies parallel to the XY plane. See Figure 4.4. The mouse controls the crosshair when *kinetext* is in editing mode. When the author toggles the mouse to move the camera around the workspace, the crosshair remains where the author last left it during editing, to offer a reference point when the author wishes to return to editing mode.

4.5 Implementation Notes

kinetext is written in C++ using the Inventor 3D-graphics library and runs in Irix 6.1. The typographic forms are derived from the texture-mapped, anti-aliased font library developed at the MIT Media Laboratory.

4.6 Program Architecture

All typographic elements are of the class ccElement. ccElement holds information for the geometries, location, orientation, and appearance of the letters. ccElements are arranged in a series of lists according to their function. *kinetext* keeps track of one such list, which is the dictionary of macros. All other lists are maintained by the individual workplanes.

All workplanes are of the class zPlane. Each zPlane holds a two lists. One is the list of the sprite ccElements and the other is the list of the clusters living on the workplane. Each cluster contains a list of operator ccElements (see Figure 4.5). The only thing differentiating whether a ccElement is a sprite or an operator is which list it is living in. All ccElements begin as sprites until attached to clusters. This flexibility allows the typography in *kinetext* to have the adaptable duality

described in Section 4.1.

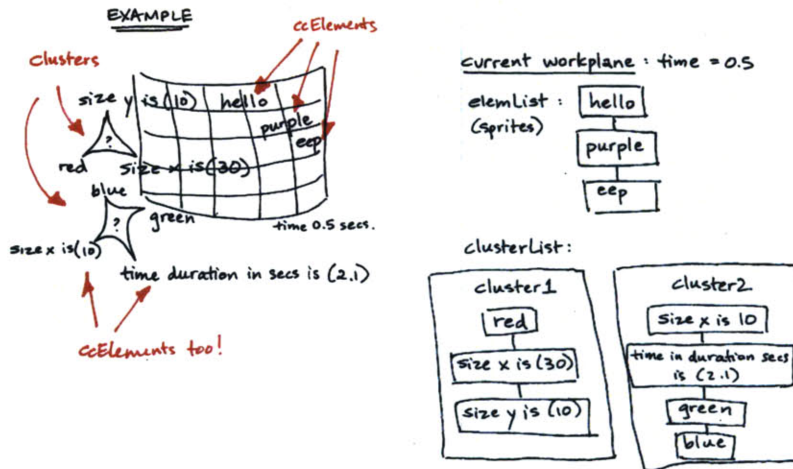


Figure 4.5: Diagram describing the software structure of a sample workplane.

ccElements themselves have no sensing capabilities of other elements around them, hence the system itself handles all collision-detection and characteristics-transferal between ccElements.

When the user toggles to play the animation, the system methodically goes through each zPlane at every half-second and accesses the list of sprite ccElements, executing the characteristic transitions as specified by each ccElement at the rate of 30 frames per second. The clusters are ignored during animation play.

4.7 The Output

Currently, animations produced in kinetext can only be viewed within the system. However, *kinetext* does have the ability to output a text file that serves as a script of all the ccElement structures and workplanes. Since it is not meant for the user to edit the scripts in a text editor, the organization of the output script is done in such a way for the system to quickly load up the structures when given the file as input.

The output script is organized according to workplane. Therefore it follows that if a sprite lives on more than one workplane, it has copies that live on other work-

planes. The script details all the interpolation values for each of the copies since each ccElement is largely unaware of its siblings (Figure 4.6).

Figure 4.6: This is an excerpt from an output script produced by kinetext.

```
time 0.000

<word>
string words
time_label time_650_start
total_duration 650
color 1.000 0.500 0.000
translation 145.550 99.190 0.000
scale 70.000 70.000 1.000
size_orig_interpolation_vals 70.000 70.000 1.000
size_start_interpolation_vals 70.000 70.000 1.000
size_end_interpolation_vals 70.000 70.000 1.000
size_term_interpolation_vals 70.000 70.000 1.000
size_delta_interpolation_vals 0.000 0.000 0.000
transl_orig_interpolation_vals 145.550 99.190 0.000
transl_start_interpolation_vals 145.550 99.190 0.000
transl_end_interpolation_vals 145.550 99.190 0.000
transl_term_interpolation_vals 145.550 99.190 0.000
transl_delta_interpolation_vals 0.000 0.000 0.000
colr_orig_interpolation_vals 1.000 0.500 0.000
colr_start_interpolation_vals 1.000 0.500 0.000
colr_end_interpolation_vals 1.000 0.500 0.000
colr_term_interpolation_vals 1.000 0.500 0.000
colr_delta_interpolation_vals 0.000 0.000 0.000

<clusterword>
string size x is (70)
time_label time_650_start_compressed
color 1.000 1.000 1.000
translation -13.742 131.671 0.000
scale 70.000 10.000 1.000

<clusterword>
string orange
time_label time_650_start_compressed
color 1.000 0.500 0.000
translation -74.037 89.985 0.000
scale 10.000 10.000 1.000

<cluster>
<clusterword>
```

If a user wishes to edit a script or write a short script by hand, the syntax is very simple:

```
time 0.0                    /* first workplane */

<word>                      /* sprite word */
string hello world          /* sprite reads 'hello world' */

<clusterword>               /* cluster word */
string myRed                /* cluster word reads 'myRed' */
color 1.0 0.0 0.0           /* RGB values for color */

time 0.5                    /* empty workplane */

time 1.0                    /* empty workplane */

time 0.0                    /* the last workplane referenced will
                               be the active plane on start-up */
```

Why such incongruity between the two scripts? *kinetext* has default values for all objects in the system, so the user can set as little or as many characteristics they wish. For the sake of completeness, the system will list all characteristic values when called upon to generate a script for output. A complete listing of possible characteristics generated for the script can be found described on the table in Figure 4.7 on the next page.

Figure 4.7: Table describing the scripting language for kinetext's output/input.

```
time /* creates a new workplane at the appropriate time in seconds */
<word> /* creates new word sprite */
<cluster> /* creates a new cluster; all workplanes start with one empty
                                          cluster already made */
<clusterword> /* creates new word to be attached to current cluster */
```

Characteristics for <word>:

```
string
time_label
total_duration /* in milliseconds, how long this sprite will be playing */
color
translation
scale
```

For the next series of instructions, there are five types of interpolation values:

orig = beginning value of the original sprite
start = value at the start of the workplane
end = value at the end of the workplane
term = ending value of the last copy of the sprite
delta = increment value per frame if 30fm/s

```
size_orig_interpolation_vals
size_start_interpolation_vals
size_end_interpolation_vals
size_term_interpolation_vals
size_delta_interpolation_vals
transl_orig_interpolation_vals ...
colr_orig_interpolation_vals ...
rot_orig_interpolation_vals ...
```

Characteristics for <clusterword>:

```
string
time_label
color
translation
scale
```

The time_label argument has several different attachments:

_start = this is the original sprite
_middle = this is a copy of a sprite
_start_compressed = this is a sprite that should have copies made

To summarize, this chapter covered the basic organization of the *kinetext* system, describing the various components used for authoring. In addition, a cursory description of the underlying software organization was discussed, concluding with a demonstration of the scripting language *kinetext* uses for output and subsequent input.

CHAPTER 5

VISUAL DESIGN OF THE SYSTEM

Beyond the mechanical workings of the environment, it is also important to discuss the visual layout of the system. This is how *kinetext* uses the 3D environment, this is what affects how users perceive interacting with the system. In the beginning, *kinetext* was an experiment in 2D visual-programming, with the intent of being able to compose a field of computation that could stay in perpetual motion. The move to 3D came with a desire to have both more space for a programming-environment, and a perceptible goal for the computation: typographic animation.

5.1 The Evolution of a Visual-Programming Environment

5.1.1 Visual Machines

Initially, when I began with the idea of creating a visual-programming language based on concrete-poetry, I sought to create an environment that offered a graphical representation of algorithms and other computation the user would create in the space. Function words would be arranged in box-like forms surrounding smaller box-like forms of internal computation (Figure 5.1). The idea was to create visual-machines built of words

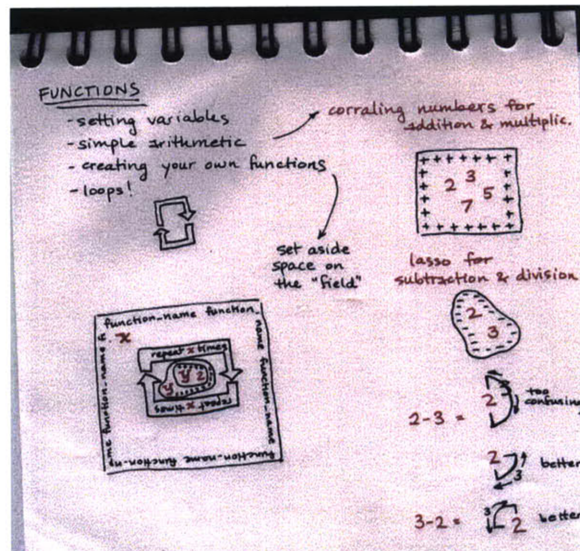


Figure 5.1: Initial sketched ideas for a visual-programming environment.

that also showed the flow of data (more numbers and words) through its internals.

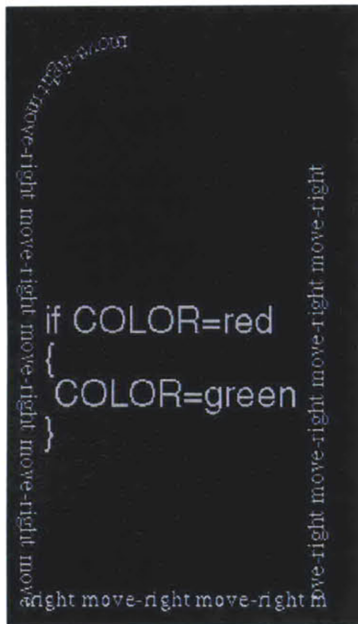


Figure 5.2: In this early prototype, inner words describe a simple program, while outer words define the motion of that program across the screen. Data, in the form of other words and numbers would be swept into the open top of the visual machine, be affected by the inner program and be pushed out of the machine.

5.1.2 Typographic Operators

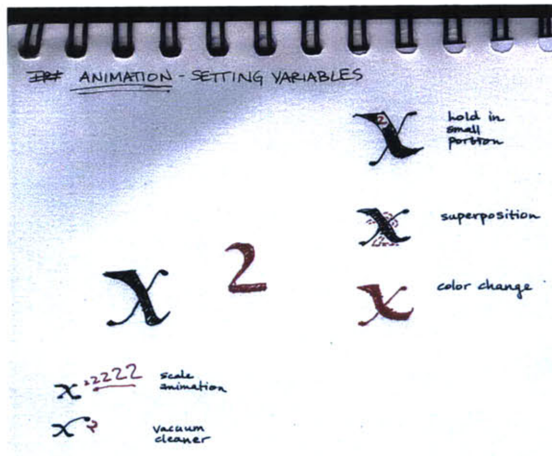


Figure 5.3: Initial sketched ideas for behaviors of typographic operators.

The first typographic operators I sketched out were all mathematical. Sometimes proximity would dictate when operators would affect data. As I progressed I also began incorporating operators that adjusted alignment and size of appearance of text. These operators could then be dispersed alongside the visual machines. The intention was to use these operators and machines to generate a field of computation constantly in motion where operators and machines alike would output data which would then become new input for the other operators. The user served as both initiator and mediator of data and operation.

Figure 5.4: In this early prototype of a program cluster, an amorphous blob containing instructions passes over some text and the number 8. The end results are aligned text and the numbers 3, 13, and 40, corresponding to the numbers one would obtain from subtraction, addition, and multiplication accordingly.

5.2 Analyzing the Final Forms that Define kinetext

After spending a few months experimenting with various ideas for a compelling concrete-programming environment, I decided to shift the focus of the typographic output. Rather than having a user fill an arbitrary space with computational structures, set these in motion and observe the intriguing, yet seemingly unpredictable output (much like cellular automata), I chose to explore the challenge of integrating design process with typographic output such that the two were interchangeable. In essence, I now sought to make concrete-programming be both result and record of the design process.

Naturally, it followed that this endeavor would be treading very closely to current commercial animation packages and would undoubtedly be measured against such systems. One effort to place a distancing factor between those systems and *kinetext*, was to pursue this experiment in the realm of 3D.

Figure 5.5: An early vision for the kinetext environment.

5.2.1 Bowed Planes

When I initially chose to address the time dimension, I knew I wanted to use depth as a scale of time. I began with square grids lined up much like a filing-cabinet metaphor, but almost immediately I saw how both flat and confusing this looked. The ambiguity arose in distinguishing one plane from another. When planes are lined up one directly after another, there is no opacity to show which are the front-most.

The next idea was the treat the time labels for the planes as distinguishing tabs, but this alone was not enough. The space still looked flat. So the next idea was to actually bow the planes to give a sense that each keyframe was almost like a box holding the geometries that were moving in each frame.

5.2.2 Staggering the Space

So now *kinetext* had bowed planes, but there was still the occlusion confusion to address. The obvious solution would be to stagger the planes in a staircase configuration, but this is inherently an inefficient use of space requiring panning of the space in order to see all the workplanes (see Figure 5.6). Instead, I decided to arrange the planes in a horizontal sine wave configuration (see Figure 5.7). This works well in that the environment can have the differentiating factor

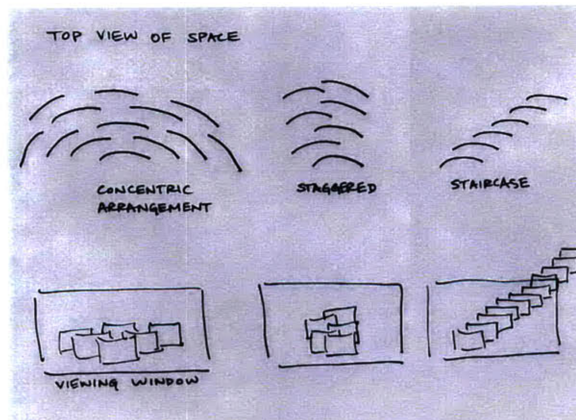


Figure 5.6: Sketches of different possible workspace

of staggering the planes within a relatively small window of horizontal space.

Figure 5.7: Top and front views of the stagger configuration (horizontal sine wave) used in the final configuration of kinetext.

5.2.3 Webbed Clusters

Originally, the clusters appeared as a static spoked structure with a center connecting all the typographic operators together. This worked well for showing connectedness, but wrongly encouraged the impression of a tree-like organization. Since the members of the cluster did not follow any kind of hierarchy, I sought to arrange them in a more amorphous form to show a loose grouping. The result was a webbed-like structure which connected all the members, and perpetually undulated in such a way so that it never seemed that any one member carried more importance than another.

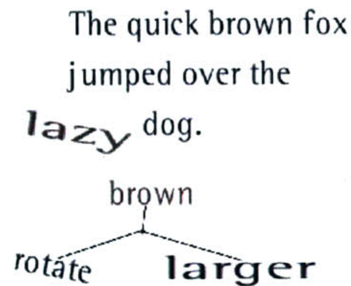


Figure 5.8: An early cluster structure.

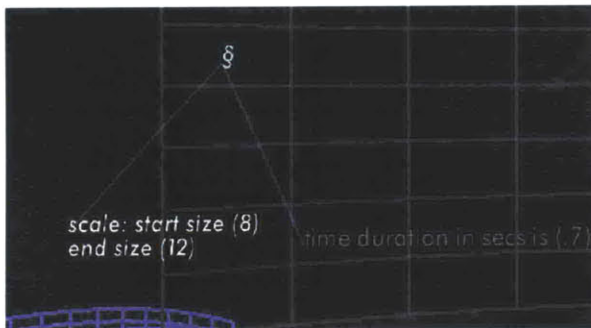
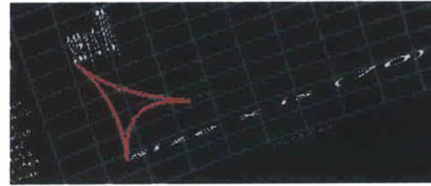


Figure 5.9: Another early cluster structure.

Figure 5.10: The final webbed look for the clustering structures.



5.2.4 Color and Contrast

For the background of the space, I chose to use black as it lent the greatest sense of depth to the space and encouraged the illusion of 3D. I experimented with using white, but found it to flatten the space somewhat, and in the case of animating text, encouraged a disconcerting comparison to printing the text on paper. However, interestingly enough, when it came to actually printing out illustrations of the space, the white background carries the opposite effect (Figure 5.12). This contrast is likely due to the different ways color is processed on paper versus the screen. On the screen, color is an *additive* process such that mixing the three primaries (red, green, blue) results in white, whereas on paper, color is a *subtractive* process where mixing the three primaries results in black.

I wanted the colors in the space to be subtle and not distract from the work being done within and so chose to use cooler colors (blues, greens) for representing the workplanes.

I added the illusion of depth to the inactive workplanes by altering the transparency levels of the planes such that planes closer to the front were brighter and successively darkened the further back they were. The brightness value of each workplane was determined by dividing how close to 0.0 seconds the plane was by the total number of workplanes.

For the active workplane, it was decided that the color should be different instead of the same color to better differentiate from the inactive workplanes. In addition, since the active plane also gets larger, it was thought that it would be less confusing if a different color was used. But the different color used had to be not too contrasting to the other workplanes, or it would distract or confuse the environment. In terms for the brightness of the color for the active workplane, it had to be visible but not too bright as to again distract the user as he or she is busy composing and

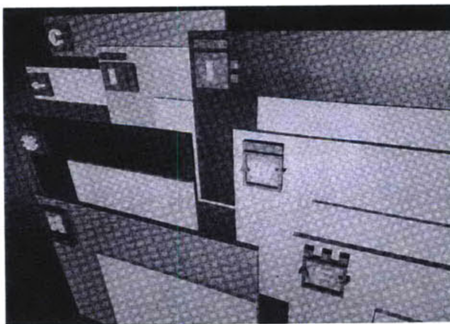
arranging typographic elements to be animated. Hence a green, close in saturation to the blue that is used for the inactive workplanes, was chosen.

In contrast, for the programming clusters, I sought to make it contrast with the cooler colors of the workplanes by making it reddish, but not a bright red which would seem to "pop" and distract from the sprites being animated (Figure 5.12). Initially the clusters had been the same color as the workplanes (Figure 5.9) but I found they blended in too well with the planes.

5.2.5 Why Grids and Not Solid Planes

Why is the workplane a finite grid if the space is really infinite? The purpose of the grids are to make the user feel comfortably grounded and not "lost in space." It offers a reference point because when you actually do play the animation you will notice the faint gray frame showing the borders the workplane would have defined, but as the user you are still free to go outside these bounds in your animation. In addition, the freedom you gain from having an "official" space and an unofficial one is that you have the sketching area where you can test out animation ideas that don't necessarily have to appear in the final animation. These can be bits and pieces that you use eventually for future animations but for the time being have them stored inside the workspace where you have created this animation.

Figure 5.11: An example of opaque planes in 3D. An empty set of CAEL's 3D control structures.



What about the grid itself? Why not use solid or translucent planes? The problem with solid or translucent planes is that more often than not, the effect one gets is that of having windows, much like a 2D desktop environment (see Figure 5.6). As absurd as that may seem, one has to realize that this phenomenon does not occur through any fault of 3D itself. Instead, the reason for this perception is due to the fact that almost every windowing system (e.g., Win95, MacOS, XWindows) available today "fakes" 3D with subtle drop-shadows and color changes for their windows that create the illusion of depth. One might add that the illusion is so effective as to be in some cases, more visually-pleasing than the real 3D-plane geometries. When forced to make this comparison, I realized that simple wire-grids were all the refer-

ence the user would need. The motion of traversing the wire-grids was enough to communicate a sense that each represented a separate plane. The motion also helped to clarify what sprites were anchored to which planes.

Seeing through keyframes also allowed for better observation of sprite-interpolation occurring on the in-between keyframes.

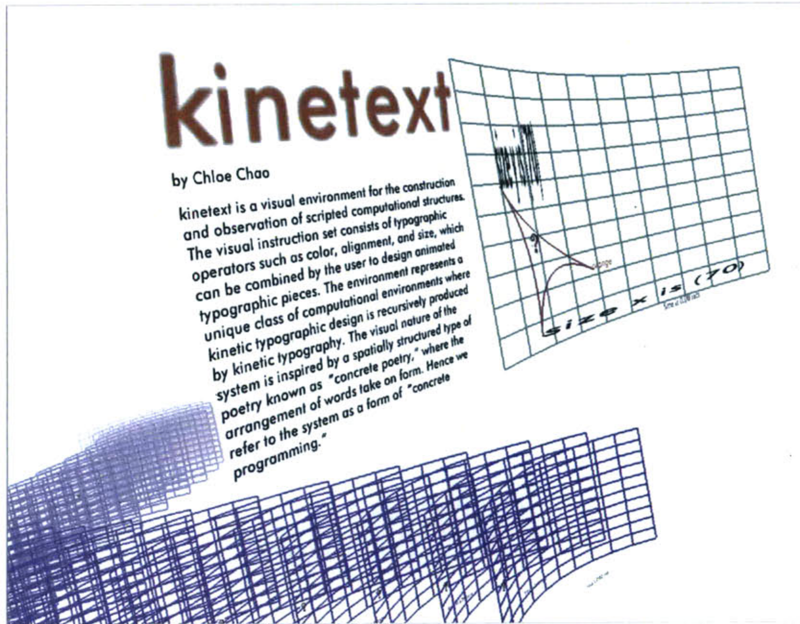
5.2.6 Animation: Why the Stage Border

During animation mode, there is a light gray border to offer the designer a reference point. Since kinetext allows the user to move the camera around in 3D, the border helps to define when the user has the camera pointed perpendicular to the animation (the border will be squared up as opposed to distorted by perspective). In addition, the border shows a 300:400 (worldspace coordinates) ratio for designers to have a concept of how large their animation is spatially. This is especially important, since the power of zooming quickly distorts one sense of scale in a 3D space. The border does not cut off any parts of the animation if sprites go beyond the lines defining the border.

5.2.7 Visual Appearance of Typographic Operators

The emphasis was to avoid ambiguity with the typographic operators. Initially, the typographic operators would try to show start-to-end transformations (See the scale operator in (Figure 5.9)). Having a rapid motion continually loop distracted the user, so eventually it was decided that motion should be avoided in the operators. However, if the operators and clusters were absolutely static, then they started to be confused with the sprites. It was clear that there should be some ambient motion for the clusters to differentiate from the background, but not be distracting for the user. Eventually I settled on a slow, fluid rotating motion for the clusters. The x and y values each correspond to different circle paths that are out of sync with one another. The resulting effect is a web that looks like each one of its tips is slowly growing then receding in turn, giving an organic, almost ambulatory motion to the program clusters.

Figure 5.12: A white background and then black background to offer contrast.



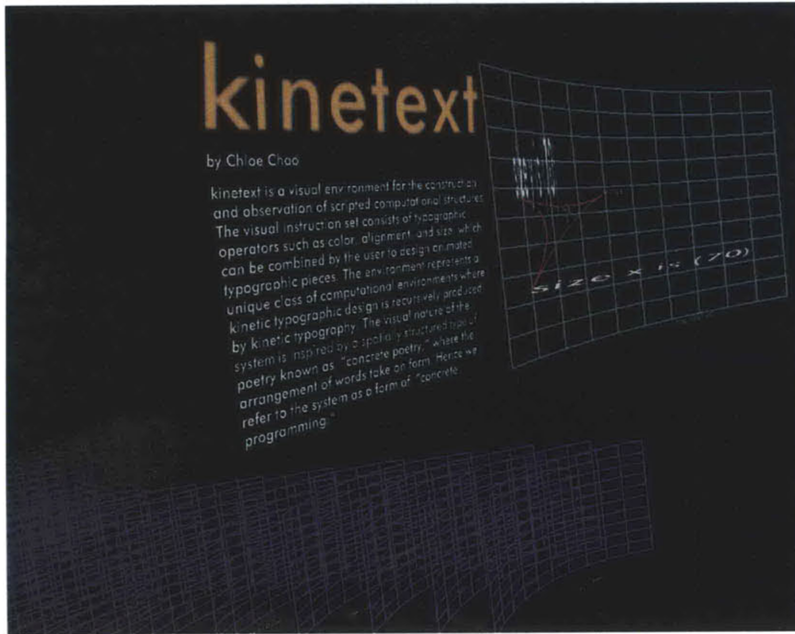
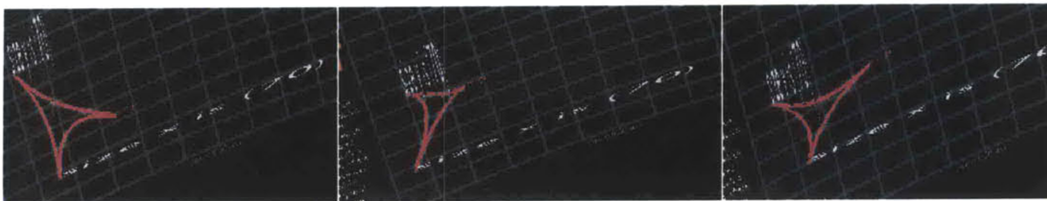


Figure 5.13: Code segment for the cluster motion in kinetext.

```
void clustermotion( double &locx, double &locy, int breakcycle ) {
  locx+=(RADIUS*cos(breakcycle+(frame*(PI/90.0))));
  locy+=(RADIUS*sin(breakcycle+(frame*(PI/180.0))));
}// clustermotion
```

Figure 5.14: Series of images showing the changing form of a 3-member cluster due to its ambient motion.



In summary, the visual components of the *kinetext* system were all carefully designed with respect to one another and the interaction and visual presentation of the overall environment. If I made changes to one component, I would have to go back to examine how those changes would affect all the other components surrounding that one component. In a way, the visual design of the system begins to have as much, if not greater, importance than the underlying software design. The purpose of any visual design is to enhance the communication of what it depicts. In the case of *kinetext*, the visual design is meant to enhance the communication of the relations between the different software elements and the computation involved in animating type. If the visual design failed, it would be highly difficult to illustrate the quality and depth of the software. One can even generalize to say that user studies are truly experiments examining how successful the visual design of an application is in communicating the purpose of its software.

Even now, it is evident that further visual design is needed for the definition of macros in *kinetext*. Because of time constraints, a visual dictionary in which to reference all the macros was never designed. As a result, the full potential of the macros still lies hidden in the software, lacking a human interface.

CHAPTER 6

AUTHORING TEXT ANIMATIONS WITH KINETEXT

To best explore how *kinetext* affects the authoring process, a series of three different animations were created:

1. a short animation of one sentence
2. a longer animation of a poem
3. a short rapid-serial-visual-presentation (RSVP) animation

For a step-by-step process of authoring an animation in *kinetext*, please see Appendix C. This chapter assumes a rudimentary familiarity with using the system and will focus on overall observations of the experience. The concluding sections of this chapter present reflections on lessons learned from creating and using the system for *kinetext*.

6.1 Animation 1: The sentence

This was the first and simplest animation of the series. Given the short sentence, "I am tired." the goal here was to animate this one idea. To achieve the expression, I decided to make the sentence appear to drowsily slump over time and have the word 'tired' change from a bright white to a muted blue.

For the short animation of one sentence, there was no need to create macros so I just made clusters for each of the different effects. Having the ability to compress the planes was useful in that I could position the end state to move vertically down, to maintain illusion of the word "tired" stretching and getting sluggish. In addition, I also added a series of "yawns" that get progressively larger and brighter as the sentence completes its slump.

Figure 6.2: The column on the left shows a series of images from Animation 1. The column on the right is a simple, yet dramatic animation of two words sized proportionately to one another.

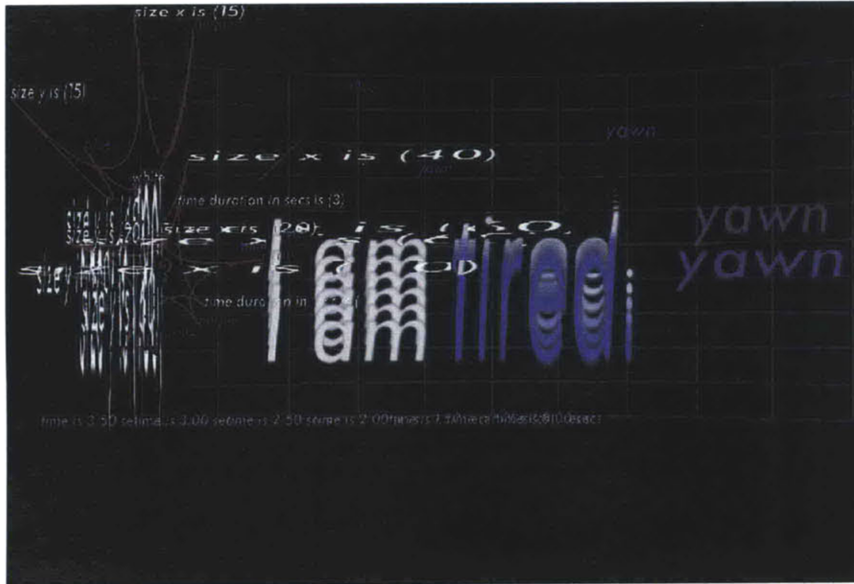
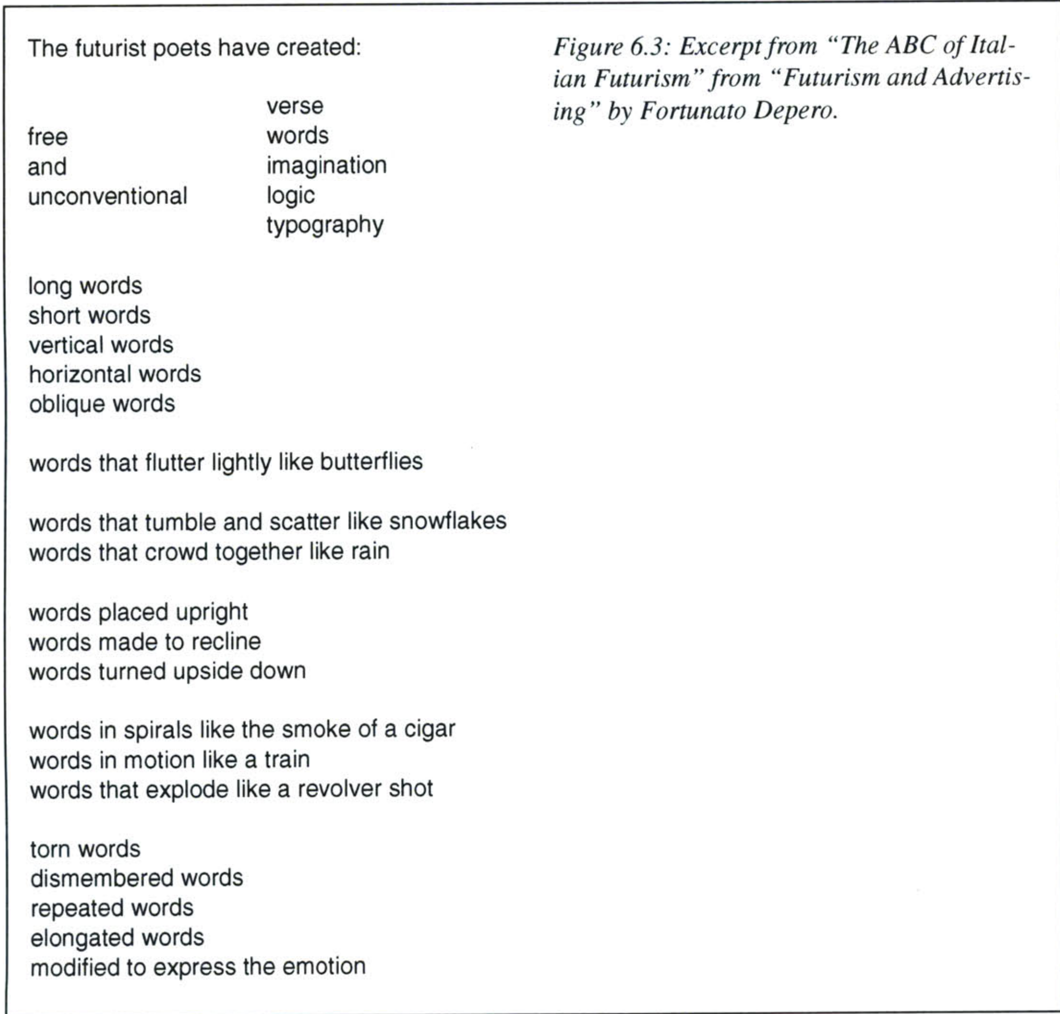


Figure 6.1: A view of Animation 1 with all the workplanes compressed. This allows the user to ensure the vertical descent of the sentence with no horizontal shift.

Overall, this animation was quickly implemented and made good use of the clustering structures and the capability of manipulating the workplanes.

6.2 Animation 2: The poem

For the animation of the poem, I ran into a few problems. Among other considerations, *kinetext* forces a very linear authoring process by virtue of the workspace being organized by time. Add this element to the fact that kinetext currently does not support "drag and dropping" of different segments of an animation and a large problem arises. For lengthy animations one has to know almost exactly how they want the animation to proceed before laying it out in *kinetext*. Ironically, I found myself sketching out how the timing of my sprites should be on paper before sitting down in front of the computer.



The futurist poets have created:

verse
free words
and imagination
unconventional logic
typography

long words
short words
vertical words
horizontal words
oblique words

words that flutter lightly like butterflies

words that tumble and scatter like snowflakes
words that crowd together like rain

words placed upright
words made to recline
words turned upside down

words in spirals like the smoke of a cigar
words in motion like a train
words that explode like a revolver shot

torn words
dismembered words
repeated words
elongated words
modified to express the emotion

Figure 6.3: Excerpt from "The ABC of Italian Futurism" from "Futurism and Advertising" by Fortunato Depero.

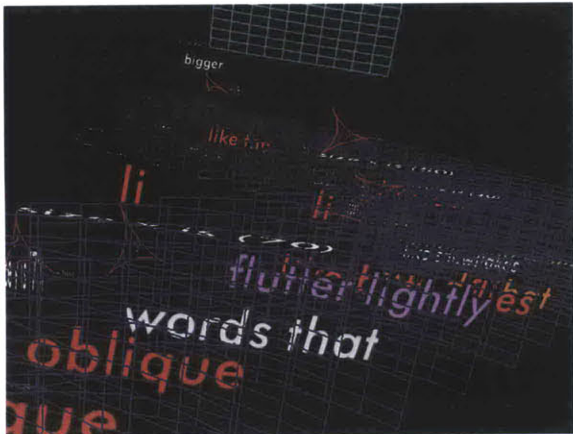


Figure 6.4: A view of the workspace for Animation 2.

Because the animation grew so large, the ability to compress the planes became useless as having all the planes flattened on one another presents something illegible. Having the ability to create macros was useful, but not helpful enough because I still had to orchestrate all sprite movement by hand as there was no feature to automate this.

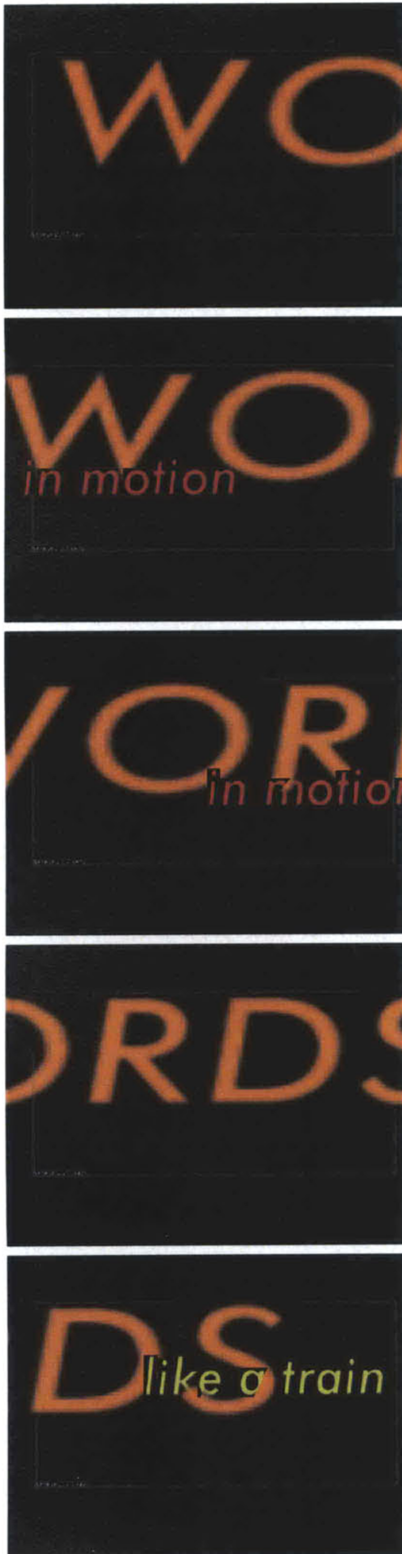


Figure 6.5: A series of frames taken from Animation 2. For this line of the poem, I chose to make "WORDS" move across the screen like a train, seeming to speed up when "in motion" and "like a train" move in the opposite direction like passing trains.

On the whole, this second animation presented an excellent challenge to *kinetext*. Authoring an animation longer than a few seconds introduced many scalability issues. In addition, complex animation also raised questions of whether greater automation will aid the expediency of authoring at the price of legibility.

6.3 Animation 3: RSVP

This animation does not explore the cluster features, but instead experiments with different views of the animation in 3D. The premise of this animation was that given a short paragraph of text, animate it legibly in under 10 seconds. Using *kinetext*, I decided to try for the effect one gets when driving over words painted on the street.

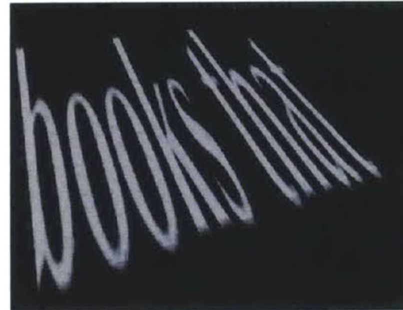


Figure 6.6: A view of Animation 3.

The workplanes were made to appear at every 0.3 seconds as opposed to 0.5 seconds in order to increase the rate at which words could appear. For the sake of legibility of the words rapidly flashing by, the same visual effect was used for all the words, so that once the reader's eyes were trained on the pattern, they could read faster. Using only one effect streamlined the authoring process greatly. After establishing a macro for the effect, all I did was enter words two or three at a time per workplane, and then went into play mode to adjust the camera viewing angle.

Figure 6.7: A different view of Animation 3.

I think we ought to read only the kind of books that wound and stab us . . . We need the books that affect us like a disaster, that grieve us deeply, like the death of someone we loved more than ourselves, like being banished into forests far from everyone, like a suicide. A book must be the axe for the frozen sea inside us.

Figure 6.8: The text animated for RSVP. A Letter to Oskar Pollak, by Franz Kafka.

This experiment in RSVP differed from both of the previous animations in that it

took a simple approach to the authoring process and much of what affected what the final animation looked like was really just taking advantage of the 3D space the system lives in. Given the exact same composition, a small change to the camera viewing angle could give the finished piece an entirely different quality (Figure 6.6)(Figure 6.7). I had always considered the use of 3D camera movement as an excellent way to view the workspace, but had never really considered how it would affect the viewing of the final animated piece. Hence, it was an unexpected boon to find that the 3D camera movement could also add its own design element to the animated pieces.

6.4 Observations and Critique

Through these series of animations, I drew several conclusions and observations about *kinetext* and the animation authoring process. Because *kinetext* is not a commercial authoring system, it lacks the flexibility of the average commercial authoring systems. *kinetext* is best suited for the design of short animated text pieces or simple, repetitive text pieces. The reason for this is because of the nature of the authoring process. For every animated effect, the user is required to type in the appropriate effect, be it a predefined macro or a new program cluster. This can become very time-consuming, but if one recalls the goals of the concrete-programming paradigm, the authoring process fostered by *kinetext* can be viewed as being analogous to writing well-commented program code. The focus here is to create animation pieces that have routines that can be easily be reused for future animations since the "documentation," in the form of visual representation, lives within the routines themselves.

6.4.1 Observations on Menus and Windows

kinetext employs no menus or windowing, although arguably the workplanes can be viewed as windows of sorts. The experiment here is to see how successful an interface can be without the ubiquitous menus and windows. So without menus, among the other recourses is to map functions to key presses. This becomes an issue because it requires remembering what certain keystrokes mean, but at least the environment space is not cluttered.

One problem with working in 3 dimensions is that a good fraction of screen space must be used to create the illusion of 3D. With 2D, every pixel of the screen can be filled with information, whereas with 3D, some fraction of the 2D screen must be

utilized to give the illusion of depth, whether it be with shadows or the tops and sides of shapes, e.g., square vs. cube.

The inverse argument to this problem of screen real estate is that 3D provides relatively infinite space with the power of zooming and panning. Instead of having a single workspace, the user can have multiple workareas that can be viewed simultaneously and also share macros across the environment. Level of detail in a 3D digital environment can be near infinite with the power of zoom and pan. So why has the popularity of 3D environments not yet overtaken 2D environments? I think the real hurdle to the commercial 3D environment is physical interface tools. The mouse is simply not an intuitive navigation device when it comes to 3D space. As a result, menus and windows will continue to dominate until new interface devices and paradigms can present a new standard to challenge the desktop mouse.

6.4.2 Critique for Movable Clusters

Even though I could create macros to use on other workplanes, sometimes I found myself wanting to merely be able to move clusters from one plane to the next to avoid having to re-type commands. But the point of anchoring clusters to workplanes is to keep a record of the designer's design decisions. One compromise I considered is being able to make copies of clusters which you could then paste on another workplane. This could ease the tedium of re-typing while still maintaining a record.

6.4.3 Critique on Navigating the Space

One of the difficulties I discovered while working on longer animations was that in order to say go from time 0.0 to time 10.0, one had to traverse *all* the keyframes in between. Ideally, I should have built a picking mechanism whereby one could jump from one workplane to another instead of having to scroll through.

kinetext uses Inventor's built-in 3D navigation controls. As with many of the 2D controls mapped to navigate through 3D space, the interface is not always clear. As a result kinetext's workspace is not the easiest to zoom or pan through. This becomes an issue because the nature of how *kinetext* utilizes space requires that the user zoom and pan. For instance, suppose one is working at time 0.0 and then moves to time 3.5. First of all, time 3.5 is further back in space, requiring the user to zoom in order to get the same level of detail they had while working on time 0.0.

In addition, time 3.5 will reside shifted to the left of where time 0.0 is due to the horizontal sine-wave staggering of the workplanes. If the user wishes to have the time 3.5 workplane centered in their window after working at time 0.0, they will have to pan a little over to the left. Ideally, to correct this, I should automatically move the camera to center on the active workplane after the user has stopped scrolling through the planes.

6.4.4 Observations on Typographic Parameters

One seeming anomaly in kinetext's development is the fact that the system does not have operators to change fonts or spacing of letterforms. The reason for this is because when *kinetext* was first designed, its primary focus was on providing a illustrative visual-programming environment using typography. The emphasis was not on building this system so I could compose a specific typographic animation with it. Hence, because the actual compositions produced by the system were considered secondary to the system itself, certain features I never gave much focus to during the design of the system were suddenly brought to the forefront once I actually sat down with my ideas for animated pieces to be implemented. By losing sight of the eventuality of authoring with the system and getting absorbed in trying to fully-realize the concrete-programming paradigm, I think a certain imbalance occurred and certain basic principles of typography were temporarily dismissed. It is my hope that in future work the balance can be restored.

In summary, the three animations offered me an excellent opportunity to test my system. The concrete-programming aspect of the system was all I could hope for, but I quickly realized I had not considered possible problems of scalability when I first designed the system. In retrospect, I should have designed with both the concrete-programming paradigm and a vision of a specific animated piece I wished to produce in mind. I believe that if each bore equal-weight in determining some of the system design decisions I made, *kinetext* would have been more flexible in accommodating variable length, complex animations.

CHAPTER 7

ANALYSIS OF METHOD AND PROCESS: A COMPARISON AMONG AUTHORING SYSTEMS

Granted, the engineer of one animation system is not the most objective judge of other systems, it is still an interesting and useful exercise to compare the authoring experiences in each. In this chapter, two systems (Macromedia Director 6.0 and Side Effects Houdini 2.0) will be examined and compared to *kinetext*. The organization of each experiment is as follows:

1. Give a general overview of the purpose and primary toolset of the system.
2. List observations on the system and discuss its approach to the authoring process.
3. Compare how the system relates to *kinetext* based on the preceding observations.

7.1 Macromedia Director 6.0

Macromedia Director 6.0 is an application for creating 2D animations and interactive presentations. Director's origins hail back to an early multimedia application called VideoWorks that was written for the Apple Macintosh. VideoWorks was aimed at the business presentations market as a more powerful tool than Powerpoint in allowing incorporation of sound and animations in presentations [Phillips, 1994]. Director today has kept some of the initial elements that made VideoWorks successful. Those elements were the ability to manipulate pieces on a "stage" by means of a "score," offering one of the first direct visual representations of the constructed piece and the construction process together. In addition, Director supports programming via an object-oriented scripting language called Lingo. Today, when viewed among competing animation products like Adobe Premiere and

After Effects, Director is seen as a heavily time-based application where the final piece is viewable and editable on a frame-by-frame basis. The feature that distinguishes it from the Adobe products is its support of interactive presentations. This makes it a widely used tool for creating many of the interactive CD-ROM titles available today.

The authoring process here is focused around 3 windows: the **Cast**, the **Score**, and the **Presentation** window. The Cast window holds all the elements to be animated, otherwise known as *sprites*, the Score window displays all the frames for the entire animation, and the Presentation window (originally the Stage) displays whichever frame is currently selected in the score.

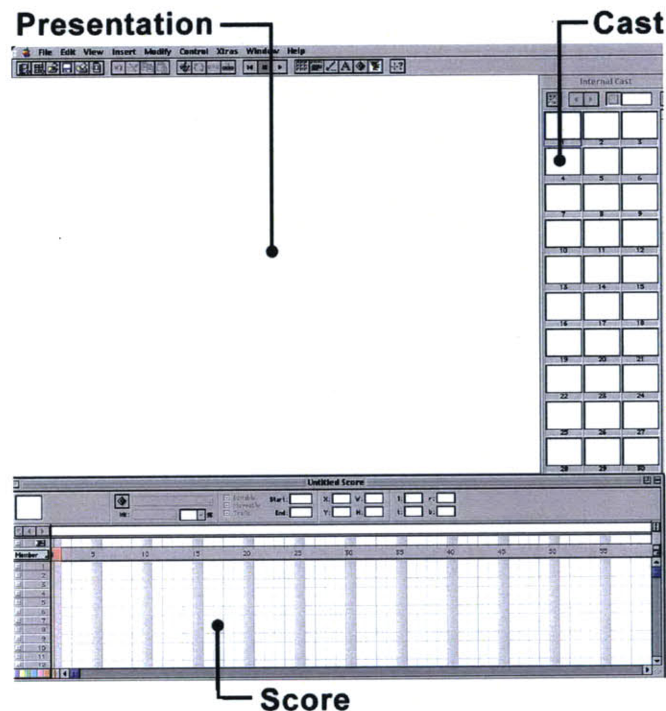


Figure 7.1: The layout of Director 6.0.

7.1.1 Observations on Director 6.0

The areas which I found myself spending the most time during the authoring process were almost equally divided between the Presentation and Score.

During the authoring process the user must continually scrolls back and forth across the score to properly time and layer sprites. While this affords the user a great deal of control, it becomes inconvenient when putting together an animation to run at 30 frames per second. Given the screen space, the window can never show more than two seconds (60 frames) at a time.

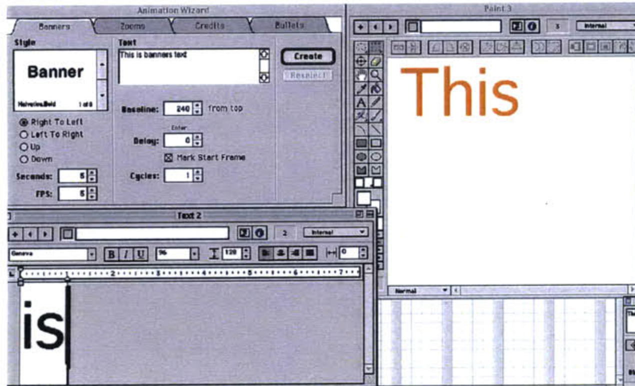


Figure 7.2: The three different editors which can create text sprites in Director 6.0 - the Animation Wizard, the Text Editor and the Paint Editor.

To resize text or perform other sprite appearance effects, one cannot directly manipulate these characteristics in the presentation window, but instead must pop up a separate editor window, be it either the text editor window or the paint window. As a result, there are two ways to create text sprites. The method available in the active toolbox only allows for black word-processor text. In order to create colored text of any sort, one has to create text in the paint window instead. Ideally, Director should have integrated the two such that there is only a single editor for the sprite to avoid confusion.

What the user can do directly in the Presentation window is create paths for the sprites to follow via dragging. This is convenient, and one can easily compare the path of multiple sprites in the same frame. Director also allows for direct editing of the text contents within the Presentation window.

What is inconvenient for text animations is how the background of the text is not transparent by default. This makes for awkward layering. However, Director has an "Animation Wizard" which consists solely of text effects. But again, it is a separate window and yet another editor in which to create text sprites, and the menu description of "Animation Wizard" would do better to be called "Text Effects Wizard" instead.

One of the clever features of Director is having a "details" box pop up when the sprite is selected. In addition, when you go to reposition the sprite, the details box does not interfere as it becomes semi-transparent when passed over other sprites.

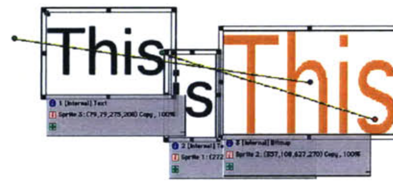


Figure 7.3: The Presentation window of Director. The paths of two sprites are shown ('is' and orange 'This'). In addition, the Details boxes of the sprites are visible and partially transparent.

Overall, while Director is an excellent tool for putting together interactive graphic presentations quickly, its strength is not in integrating its tools together well. It uses familiar conventions of word processors and paint programs but fails to merge the separate tool sets very well, offering multiple ways to author sprites instead of unifying them into one tool.

7.1.2 Comparing Director 6.0 and kinetext

Director is similar to *kinetext* in that there is an inherent emphasis on working with time during the authoring process. The Score is the equivalent of *kinetext's* keyframe workplanes, and the Presentation window is the equivalent of a single workplane. Whereas Director maintains separate areas to work in, *kinetext* integrates all areas. However, Director enjoys more scalability in terms of project sizes because by maintaining separate window areas, the user can focus on the Score to quickly layout a rough version of an animation and then at a later time go back to the Presentation window to tweak. By integrating score and presentation, *kinetext* loses the abstraction layer needed to simplify the authoring process for large animations.

Inherently, there is a trade-off in what abstraction can bring. Generalization adds a layer of abstraction that enables more complicated things to be simplified and grasped more easily. However, generalization essentially masks details and specifics, and if there is not an easy way for the user to map back to this detailed information, this becomes an issue. With the case of Director, the generalization of having a score and being able to quickly organize cast objects simplifies the authoring process, but obscures later legibility of the composition of the animation. The user is required to browse through multiple windows in order to unravel the composition, and even then can only really view around two seconds at a time. Alternately, with the case of *kinetext,* there is no generalization such that the composition contains every bit of information the author put into the animation. Unfor-

tunately, this makes the process very labor intensive for the author the lengthier an animation grows.

7.2 Side Effects Houdini 2.0

Side Effects Houdini 2.0 is a system for creating sophisticated 3D animations. Houdini's origins lie in Side Effects PRISMS, a 3D animation system that uses a procedural authoring process. The emphasis of PRISMS was to give the user expert control in by providing many options, parameters, controls, an extensive scripting language, etc., sometimes at the expense of efficiency. As its successor, Houdini both streamlined PRISMS with tighter integration of features and placed more emphasis on making the user interface easy to use so that complex animated effects could be achieved by users relatively new to the environment.

The authoring process here is based primarily on a set of 3 areas: the **Layout Area**, the **Viewport**, and the **Parameter Area**.

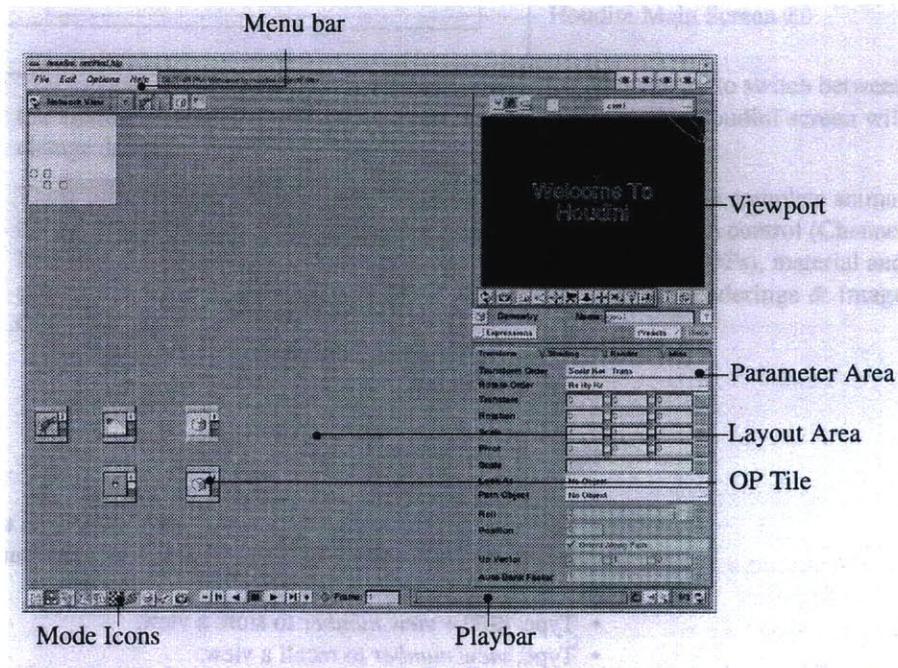


Figure 7.4: The layout of Houdini 2.0.

7.2.1 Observations on Houdini 2.0

The areas where I found myself spending the most time during the authoring process were the Layout Area and the Viewport.

There is not much emphasis on "seeing time" in Houdini. Time consists of a playbar with a thumb. The thumb indicates what frame in the animation is currently being viewed. The user drags the thumb along the playbar to move across frames through time. To animate, one goes to a frame, changes the scene, verifies the changes, and the system the interpolates between positions. In addition, the user is able to set the kinds of interpolation whether linear or periodic with growth or decay.

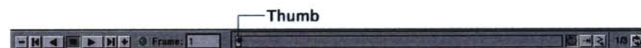


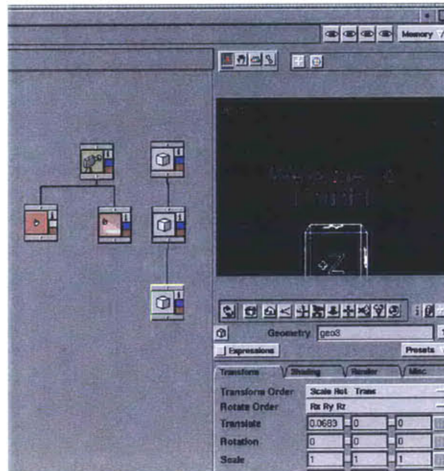
Figure 7.5: The playbar of Houdini

Because Houdini supports such a large host of tools for modeling, animation, and rendering, there is an increased start-up time where many more controls have to be learned before the user can get started. In addition, the user also has to become accustomed to the visual-programming paradigm for authoring. This may sometimes prove difficult, especially if a person is acclimated to time-organized animation authoring. But once the user learns the primary set of controls, Houdini becomes a lot easier to use in that all the tools are integrated in such a way that the designer can easily use the other toolsets without having to learn a new set of controls. After going through the tutorial for performing a simple animation via object tiles and manipulation of objects in the 3D viewport, it is relatively easy to extrapolate these techniques when using the other capabilities of the system. Of course, the user still has to rely on manuals to implement more complicated effects and learn how to access the other capabilities of the system, but the basic structure and organization of the authoring process remains fairly uniform, even when scaling up the complexity of the animation.

7.2.2 Houdini's Visual-Programming Paradigm

Houdini introduces "OPerator tiles" which are the geometric components, actions, and procedures that are used to compose scene graphs. These tiles can then be arranged in the Layout Area to form graphic representations of the dataflow between operators.

Figure 7.6: A view of Houdini's Layout Area, Viewport, and Parameter Area. Note the connected object tiles present in the Layout Area.



This paradigm is useful because the user can work in general terms and set up an entire scene graph before tweaking parameters. In addition, with the workflow model, there is a concept of multiple, non-sequential "undo" since all the operations are tiles that can easily be connected and disconnected, and re-attached later.

Text animation is treated much like animating any other 3D geometric form. In effect, any visual effect a user creates for a shape can easily be applied to a piece of text, be it a bouncing ball or fluttering leaves. In the case of a bouncing ball, the user literally exchanges the Sphere OPerator (SOP) Tile with a Font OPerator (FOP) Tile in the Layout and a piece of text is now bouncing.

7.2.3 Comparing Houdini 2.0 and kinetext

kinetext is different in that its layout area is integrated with the animation display area. Houdini keeps the layout area distinctly separate from the animation area. But this paradigm works well for Houdini as it provides a way to generalize the complex scene graphs being constructed. *kinetext* is not targeting animation of large, complex 3D geometries as Houdini is, and instead chose to study the possibilities of doing all authoring in a *single* space.

Houdini's approach to visual-programming is very different from the paradigm that *kinetext* follows. Houdini's visual OPerators are more focused on directing the flow of operation than offering direct visualization of sprite components. Again, it is an issue of abstraction layers and generalized complexity. Houdini's separate authoring areas allow the user to work simultaneously with different levels of

detail visible at once. For example, the Layout Area packages all the information of different scene objects into small, square tiles, and the information that pertains to the actual scene object receives a kind of magnifying glass effect when the user clicks on a tile and looks over into the Parameter Area. On the whole, Houdini is an excellent example of how a different kind of visual-programming paradigm can be employed in a complex animation application.

7.3 Results from the Authoring Experience

Both Director and Houdini use 2D paradigms for authoring animations. There are multiple windows, extensive menus, and plenty of boxes for tweaking numbers for position and appearance. Houdini does have a 3D Viewport which conveniently allows for direct manipulation of scene objects in 3D, but the bulk of the authoring takes place in the 2D Layout Area.

Upon examination, *kinetext's* workspace can be viewed as a hybrid of Director's Score and Houdini's Layout Area. The workspace is organized by time, like the Score, but also supports the construction of small visual programs, like the Layout Area. At the time of *kinetext's* development, Houdini had not been available for study, otherwise I believe *kinetext* could have used Houdini's programming paradigm as a quality benchmark.

In terms of legibility of the authoring process, Director presents a somewhat oblique set of data. In order to understand the Score, the user has to cross-reference ID numbers of the sprites in the Cast. Houdini fares better with the use of Object Tiles, allowing the user to have both a general idea of the layout of the system and a detailed view by clicking on any of the Object Tiles. Houdini is well on its way towards proving that a visual-programming paradigm offers an excellent record of the authoring process.

Houdini does generalization well, as opposed to Director. Critics have even said that Houdini streamlines the authoring process and avoids the bottlenecks of more traditional computer animation tools. Such bottlenecks usually arise from poor integration of the multiple toolsets required for animation.

Director and Houdini represent two different ends of the animation authoring para-

digm. One carries traditional 2D animation into the digital arena, and built its tools according the what the physical 2D authoring process demanded. As a result, Director is highly time-based and sprite-based. On the other end of the spectrum there is Houdini, which moves outside of current authoring paradigms and instead introduces a new archetype. This new visual-programming model streamlines the authoring process and is flexible enough to apply to all aspects of digital 3D animation. *kinetext* lies somewhere in between these two systems, combining some of the old paradigms (time-based organization) with some of the new (visual-programming of sprite behavior).

	Authoring Paradigm	Workspace Layout
Director:	Time-based authoring	A Score depicting all the frames of the animation, a Presentation window showing the current frame of the animation, a Cast cataloging all sprites.
Houdini:	OPerator-tile visual-programming	A Viewport showing the scene, a Layout Area for arranging OPerator tiles, a Parameter Area for viewing details of the OP tiles.
kinetext:	Concrete-programming	A single workarea with keyframes organized by time.

Figure 7.7: Table depicting the three systems.

To summarize, the preceding examinations of Macromedia Director 6.0 and Side Effects Houdini 2.0 coupled with the preceding chapter on *kinetext* animations offer a framework of reference in which to compare *kinetext* to current animation systems capable of typographic animation. It is apparent that each system has different goals for what kind of paradigm the authoring process should follow. Director favors an emphasis on time, Houdini favors an emphasis on a flow organization of sprites and operations, and *kinetext* favors a legibility of design decisions over time. Each system pursues its own paradigm sometimes at the expense of other aspects of their system. Director loses legibility, Houdini loses ease of use in requiring a steeper initial learning curve of authors, and *kinetext* loses scalability.

CHAPTER 8

CONCLUDING REMARKS

8.1 Summary

kinetext introduces a user-interface environment that allows for the authoring of text animation through the construction of visual programs. These programs are composed of words arranged in forms designed to reflect their function, in this case it is simple loose averaging in the form of clusters. This emphasis on words arranged in forms defines the concrete-programming paradigm.

As an interface, *kinetext* is not meant to be a novice's tool, but nevertheless it explores certain visual cues that are more intuitive than their 2D counterparts. For instance, time becomes a simple matter of depth. If the workplanes seem to be disappearing over the horizon, it indicates a fairly long animation. In 2D systems, time frequently is represented via a score that can only be seen in small chunks, giving no real indication to overall length.

The authoring process fostered by kinetext can be viewed as being analogous to writing well-commented program code. The focus here is to create animation pieces that have routines that can be easily be reused for future animations since the "documentation," in the form of visual representation, lives within the routines themselves.

This thesis has covered the many aspects of kinetext, beginning with the reasons behind the initial drive to use typography for visual-programming and describing the different iterations involved in the evolution of the visual design of the system. The system itself was described, along with a series of animations constructed with the system, allowing the author to critique aspects of the system with regard to the authoring process and legibility of design. Finally, an analysis of two other current authoring systems was performed in order to gain a sense of where the *kinetext* system stands with regards to peers in related work.

8.2 Future Work

kinetext represents the beginning of a whole new generation of elegant visual-programming environments, moving beyond the traditional 2D flow-chart styles to adopt more expressive and legible forms.

8.2.1 Legibility and Collaboration

The success of *kinetext* as a readable 3D environment is clear in that casual observers immediately grasped how the space is being used and could easily envision mapping the environment to different controls. Having staggered planes in space and the ability to compress these planes easily can translate to represent different information.

Future work can capitalize on this readability. For instance, a similar model can be used to show a representation of one plane per sprite instead of time. Each plane could illustrate the full animation of one sprite and the user could select which planes to compress and see the resulting animation of a few select sprites or all active sprites.

One can also imagine *kinetext* being able to support collaborative animated pieces, since this idea of documenting the process through visual programs is immensely useful for cooperative work between multiple authors. Designers could either confer simultaneously sharing a single workspace, or merely have the system keep track of each designer's additions to a shared piece. The additions would be differentiated by displaying multiple copies of the same workplane sequence, where each designer would be responsible for one of the series, and the collaborators could then examine each others' workplane sketches as they work on their own. Designers could leave the equivalent of post-it notes to one another by simply typing on whichever workplane they wish to refer to, and the system could then assign a blinking characteristic to the note so it can draw the other designers' attention when they next come to visit the environment.

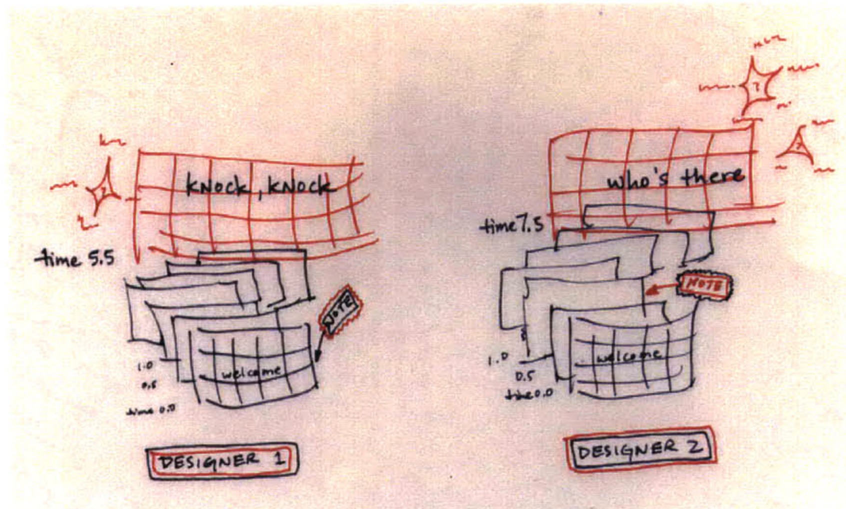


Figure 8.1: Sketch of idea for kinetext to support collaboration. Both designers have identical series of workplanes describing the same animation. Designer1 can work at 5.5 secs while Designer2 works at 7.5 secs. Changes made in one series will propagate to the other, except for post-it notes. Post-it notes signallers match the visual appearance of the respective designer's color scheme.

8.2.2 Extending the Concrete-Programming Paradigm

Another venue to pursue in future research is to further build on the concrete-programming paradigm by introducing additional forms words can assume besides the cluster. One such idea is to have a scaffolding structure that allows for more complex animation transformations by allowing individual transformations for each letter of a word via a chain of clusters (Figure 8.2).

Figure 8.2: An idea for chaining clusters to create more complex animation behaviors.

Another idea is to introduce the concept of having a visual dictionary living within the environment that depicts the visual definitions of all the macros the user has created or imported.

Other ideas for future development can stem from the results of the animation experiments in *kinetext*. One key problem was the need for additional features to allow for expedient authoring of lengthier animations (lasting beyond 15 seconds).

Among the other features that seem needed are ways to allow motion of sprites to be programmed into clusters. Other features regarding better navigation of the workspace could also be designed to make the system more efficient.

Another venue to explore would be to adapt successful paradigms from Houdini into the concrete-programming framework. The ability to have procedural flow diagrams appears to be a very powerful way to author animation. Perhaps the scaffolding-structure could be based on that analogy of flow.

The Aesthetics and Computation Group at the MIT Media Laboratory is currently in the process of developing a 3D environment for prototyping computational design ideas. The project is informally called acWindows and shares some of *kinetext's* principles regarding the computational medium. The aim of this windowing system is to create a platform that provides a general framework for rapidly generating a variety 3D applications. By having this common foundation, 3D applications that would otherwise be stand-alone units, would instead now be able to communicate and pass messages between one another through the environment. By way of such a system, one can imagine the concrete-programming paradigm of *kinetext* eventually branching out to be used by other systems for purposes beyond typographic animation.

APPENDIX A - USER MANUAL

- The executable is
  `/mas/acg/u/cchao/demos/kineText`

- When you run it, after the window comes up, hit `Esc` to get the crosshair.

- Type away. Colors and keywords are:

```
red
yellow
green
blue
purple
orange
brown
white
size x
size y
time
rotate x
rotate y
rotate z
```

keywords should then prompt you for a numeric value. Type it in and close parentheses.

- To attach words to clusters, select the word with the left mouse button, and then click and drag the right mouse button.

- Clusters combine word properties so when you drag clusters over other words, the effects should occur.

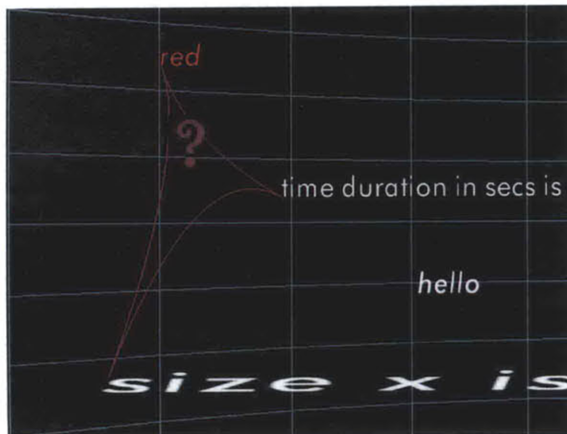
```
Up arrow and down arrow scroll through the time planes.
F1 - creates a new cluster
F2 - compresses all the planes together
F3 - expands plane out again
F5 - animation play mode
F6 - return to workspace mode
F11 - return camera to home position
F12 - save workspace out to file output.kT
Esc - toggles between typing mode and camera movement mode
```

- To upload a workspace, you can copy `output.kT` to something like `input.kT` and type: `kineText input.kT`

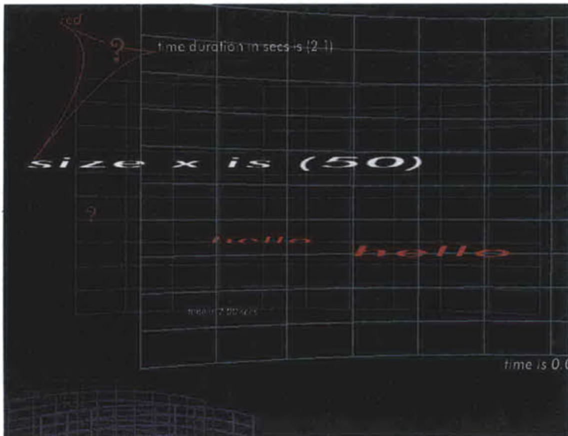
- To create a macro called `dog` type: `{dog}`
- The { } should vanish and the word should look like: `dog`
- Create a cluster with `brown` and `size x is (30)`
- Drag cluster over `dog`
- Now the next time `dog` is typed it will immediately assume the given characteristics.

APPENDIX B - AUTHORING A SIMPLE ANIMATION WITH KINETEXT

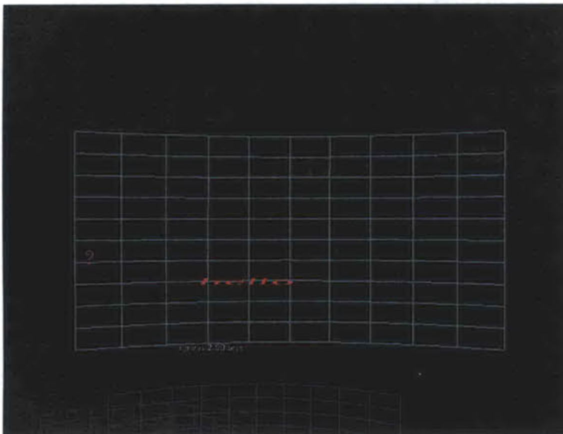
Type hello.



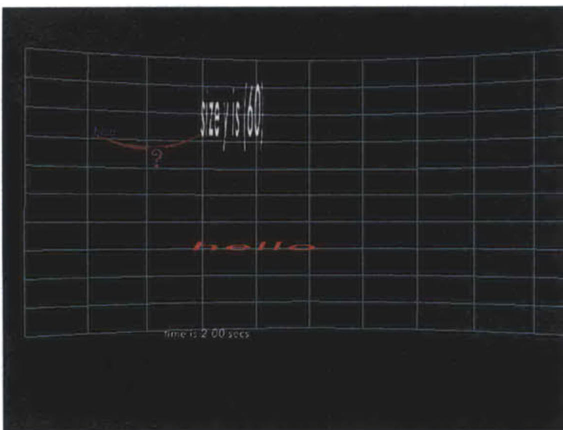
Create a cluster with red and size x is (50) and time in seconds is (2.1).



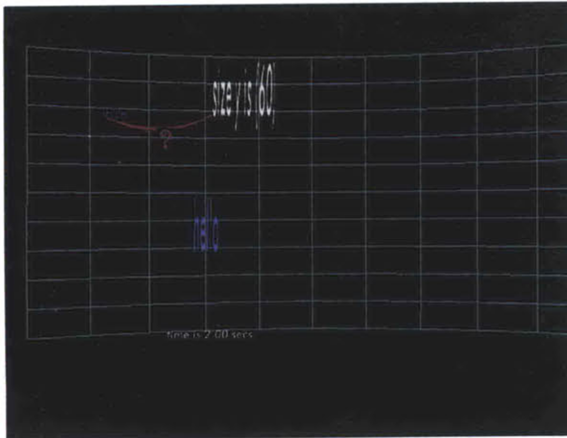
Drag cluster over hello. Another workplane ~2 secs further back will pop up.



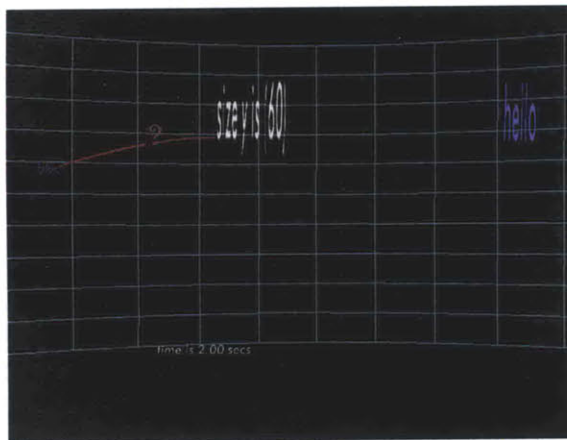
Arrow up to the popped-up work-plane.



Create a cluster with blue and size y is (60).

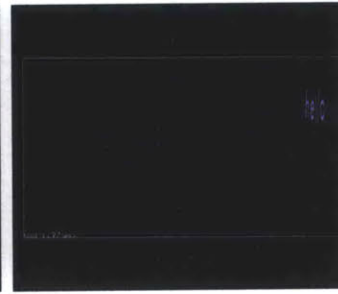


Drag cluster over hello.



Move hello over to a different spot in the workplane.

Play the animation.



BIBLIOGRAPHY AND REFERENCES

Adobe Systems Incorporated. *Adobe After Effects*, 1995.

Eric Gill. *An Essay on Typography*, Godine, Boston, 1936.

Berjouhi Bowler. *The word as image*, Studio Vista, London, 1970.

Lewis Carroll. *Alice's Adventures in Wonderland and Through the Looking-Glass*, Oxford University Press, London, page 28, 1971.

Chloe Chao and John Maeda. Concrete Programming Paradigm for Kinetic Typography. *1997 IEEE Symposium on Visual Languages Proceedings*, pages 450-451, 1997.

Marita Duecker, et. al. Visual-Textual Prototyping of 4D Scenes. *1997 IEEE Symposium on Visual Languages Proceedings*, pages 332-339, 1997.

Eugene Gomringer. *konstellationen*, Spiral Press, Berne, 1953.

Suguru Ishizaki. Wigglet. http://www.wigglet.com, 1997.

Hideki Koike, Tetsuji Takada, and Toshiyuki Masui. VisuaLinda: A Framework for Visualizing Parallel Linda Programs. *1997 IEEE Symposium on Visual Languages Proceedings*, pages 176-182, 1997.

Richard Kostelanetz. *Imaged Words & Worded Images*, Outerbridge & Dienstfrey, New York, 1970.

Macromedia Inc. *Macromedia Director*, 1996.

John Maeda. *Flying Letters*, Digitalogue, Japan, 1996.

Mihai Nadin. Design in the Age of a Knowledge Society. *formdiskurs: Journal of Design and Design Theory*, pages 41-59, 1997.

Ian Phillips. A comparative review of HyperCard and Director as tools for time-based expressive work, Technical Report, Coventry University, 1994.

Andrew Rau-Chaplin and Trevor J. Smedley. A Graphical Language for Generating Architectural Forms. *1997 IEEE Symposium on Visual Languages Proceedings*, pages 264-271, 1997.

Side Effects Software Inc. *Houdini*, 1997.

David Small. Expressive Typography: High Quality Dynamic and Responsive Typography in the Electronic Environment, Masters Thesis, MIT, 1987.

David Small and Yin Yin Wong. Minsky Melodies, http://www.media.mit.edu/~dsmall/brainop, 1996.

Douglas Soo. Implementation of a temporal typography system, Masters Thesis, MIT, 1997.

Tenax Software. *Cornix*, 1997.

UVA User Interface Group. Rapid prototyping for virtual reality. *VR Blackboard, IEEE Computer Graphics and Applications*, 1995.

Josie Wernecke. *The Inventor Mentor*, Addison-Wesley, 1994.

Yin Yin Wong. Temporal Typography: Characterization of time-varying typographic forms, Masters Thesis, MIT, 1995.

R. Zeleznik, K. Herndon, and J. Hughes. SKETCH: An Interface for Sketching 3D Scenes. *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 163-169, 1996.

Frank Van Reeth, Karin Coninx, Sam De Backer and Eddy Flerackers. Realizing 3D Visual Programming Environments within a Virtual Environment. *EUROGRAPHICS '95*, pages 361-370, 1995.

Eugene Wildman. *Anthology of Concretism*, Swallow Press, Chicago, 1967.

Emmett Williams. *An Anthology of Concrete Poetry*, Something Else Press, Inc., New York, 1967.

READERS

John Maeda is Interval Assistant Professor of Design and Computation at the MIT Media Laboratory, where he also directs the Aesthetics & Computation Group (ACG). His mission at MIT is to foster the development of individuals who can find the natural intersection between the disciplines of computer science and graphic design.

William J. Mitchell is Professor of Architecture and Media Arts and Sciences and Dean of the School of Architecture and Planning at the Massachusetts Institute of Technology. He teaches courses and conducts research in design theory, computer applications in architecture and urban design, and imaging and image synthesis. He consults extensively in the field of computer-aided design and was the co-founder of a California software company.

Yin Yin Wong is an interaction design consultant working on projects in the area of kinetic typography and tools for designers. She holds a MS in Media Arts and Sciences from MIT, and a BFA in Graphic Design from Carnegie Mellon University. She has worked as a print designer and as a user interface researcher with Apple Computer's Advanced Technology Group. Yin Yin's work has been exhibited at SFMOMA and appeared in publications including ID Magazine, the Computer Human Interaction Proceedings, and the Atlantic Monthly.