A Software System for Laptop Performance and Improvisation

Mark Zadel

Music Technology Area, Department of Theory
Schulich School of Music
McGill University
Montreal, Quebec, Canada

May 2006

A thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Master of Arts in Music Technology.

Abstract

Laptop performance—performance on a standard computer system without novel controllers, usually by a solo musician—is an increasingly common mode of live computer music. Its novelty is diminishing, however, and it is frequently described as being uninteresting. Laptop performance often lacks the sense of effort and active creation that we typically expect from live music, and exhibits little perceivable connection between the performer's actions and the resulting sound. Performance software designs tend to constrict the flow of control between the performer and the music, often leading artists to rely on prepared control sequences. Laptop performers become "pilots" of largely computerized processes, detracting from the live experience.

This thesis project presents an alternative software interface that aims to bring a sense of active creation to laptop performance. The system prevents the use of prepared control sequences and allows the live assembly of generative musical processes. The software resembles a freehand drawing interface where strokes are interactively assembled to create looping and cascading animated figures. The on-screen animation is mapped to sound, creating corresponding audio patterns. Networks of strokes are assembled and manipulated to perform music. The system's use of freehand input infuses the music with natural human variability, and its graphical interface establishes a visible connection between the user's actions and the resulting audio.

The thesis document explores the above issues in laptop performance, which motivated this research. Typical examples of performance software are presented, illustrating the interface design patterns that contribute to this situation. The thesis software project is presented, describing its goals, design and implementation. The properties of the interface are discussed in light of the project's initial objectives. It is concluded that the system is a solid step toward a novel approach to laptop performance software.

Sommaire

Les concerts de musique «pour laptop» (où un interprète solo se sert exclusivement d'un ordinateur pour faire de la musique) sont devenus monnaie courante sur la scène actuelle de musique assistée par ordinateur. Cependant, ce mode de performance est souvent critiqué par le public pour son manque d'intérêt visuel et le manque de cohérence entre les sons produits et les actions du musiciens. De plus, les logiciels dediés à ce type de créations ont la fâcheuse tendance de restreindre les possibilités de contrôle sur la musique et sont conçus de telle sorte que les artistes finissent par n'avoir qu'un rôle de «pilote» pour processus automatisés. Tout ces faits contribuent à amoindrir la qualité de ces concerts.

Ce projet de thèse présente un logiciel d'interprétation alternatif qui a pour but de donner aux concerts pour laptop cette spontanéité qui leur faisait jusqu'à présent défaut. En effet, le système empêche l'utilisation de séquences de contrôle toutes prêtes et permet l'assemblage, sur scène, de processus génératifs musicaux. L'interface ressemble à un outil de dessin interactif permettant d'assembler des traits pour créer des figures animées qui produiront les sons. On joue donc de la musique en assemblant et en manipulant des réseaux de traits. De plus, ces traits étant dessinés à main levée, le système donne une grande expressivitée à la musique résultante.

Le présent document approfondit les problématiques mentionnées ci dessus et fournit une justification à ce travail de recherche. Des exemples typiques de logiciels d'interprétation sont analysés afin de mettre en évidence les limitations de leurs interfaces qui entravent la creativité spontanée. Un nouveau logiciel est ensuite présenté. Ses objectifs, sa conception et son implémentation sont explicités. À la lumière des premiers résultats obtenus, il est conclu que le présent système constitue une étape intéressante vers le développement de nouveaux logiciels d'interprétation.

Acknowledgments

I am deeply indebted to a number of individuals who helped shape and support this project in many ways. My thesis advisor, Gary Scavone, provided significant guidance and feedback over the course of my research. His contribution to the development and organization of the underlying ideas greatly enhanced this work.

Thanks to my talented colleagues in the McGill Music Technology Area for providing a rich environment within which to pursue this research. The faculty and students served as a constant inspiration, and our various discussions were instrumental in shaping this project. Thanks to the user testers for their valuable feedback: Dave Benson, Andrey da Silva, Thor Magnusson, Elliot Sinyor and others.

Many thanks to my family—Veronica, Joe, Lydia and Andrew—who have been incredibly supportive, patient and understanding over the course of my degree. Finally, a special thanks goes to Penny Kaill-Vinish, without whose constant encouragement and perspective this project might never have been started, let alone finished.

Contents

List of Figures

Chapter 1

Introduction

Computer music is being transformed in the wake of powerful personal computers. It has never been easier to experiment with computer-generated sound thanks to readily available software tools. The computer can even be used as a performance instrument, generating audio in real-time.

Laptop performance is a common contemporary incarnation of live computer music. Laptop performance is taken here to mean live computer music played on a standard computer system without the use of novel hardware controllers, usually by a solo musician. This style of performance reflects the pre-eminence of software in current computer music production.

While offline studio techniques for computer music work well and are widely used, the performance of these pieces remains problematic. Laptop performance practice is still maturing, and it often lacks some of the elements that make acoustic musical performance compelling. This thesis project introduces a prototype software interface that responds to some of these patterns in laptop performance.

This introductory chapter contextualizes and motivates the thesis work. An overview of the project itself is presented, along with an outline of the rest of this document. Finally, the contributions of the research are summarized.

1.1 Context and Motivation

Laptop performance is becoming an increasingly popular mode of live computer music. Computer music producers now often work entirely in software, and turn to this same technology for performance.

While laptop performance is becoming more common, its novelty is starting to wane. These performances tend to feature little physical activity, but complex music. It is difficult for the audience to perceive exactly what is going on on-stage, and to reconcile what they see with the complex music that they hear (Tanaka, 2000). Laptop performances often lack the sense of effort, difficulty, and active creation that are felt to be inherent in live music. Laptop performance exhibits largely the same issues that exist in acousmatic performance (Cascone, 2003), which derive from those in live tape music.

The vast majority of software tools for making music are studio-centric, focusing on the non-real-time editing of sound and music. These techniques are mature and are well served by computer technology. Studio interface designs do not translate well to live use, however. Contemporary computers are able to generate audio and do digital signal processing in real-time, but this is not sufficient to engender engaging live performance.

Laptop performance pieces are often dense and complex. There are an overwhelming number of parameters that could be subject to live manipulation, and it is cognitively challenging for a solo performer to cope with this complexity. Musicians typically resort to using prepared control material to manage these pieces in a live setting. Samples, note sequences, effects presets, processes and layering are, to a large degree, prepared before the performance. While this is an effective strategy that makes these pieces more manageable, it comes at the cost of leaving the performer with too *little* control. Most of the music is automated, and the musician is left to drive fixed processes with a small set of controls (Collins, 2003).

Preparation is an important part of any kind of performance, but there is a difference between what we see here and preparation for acoustic performance. Jazz musicians, for example, prepare by practicing scales and patterns intensively for use in live improvisation. A computer is prepared by programming, and the resulting audio plays back exactly every time. When we see a well-prepared musician perform, it is interesting due to its inherent difficulty and natural variability. When we run computerized processes as performance, it is less interesting due to its ease and mechanical consistency.

Another factor is the design of the performance interfaces themselves. The software tools that are available for live use are often designed in the image of studio applications, borrowing their interface patterns and concepts. On-screen widgets—knobs, buttons, and faders, for example—are arranged into control panel layouts, allowing only one or two scalar parameters to be accessed at once with a mouse (Levin, 2000, p. 38). This interface arrangement constricts the flow of control from the musician to the computer, compromising live use.

Though the live acrobatics of laptop performers can be intricate and complex, it is often the case that musicians exercise limited control over a piece as described above. In these instances, performance is reduced to triggering events, playing prepared control sequences, calling up presets, and perturbing scalar parameters. Laptop musicians "pilot" their works, as we see in live acousmatic music (Cascone, 2003), and the computer itself manages much of the music production. The music varies little between performances, and improvisation is difficult. These factors are what essentially lead to uninteresting laptop performance.

This Master's research attempts to address some of these issues by offering an alternative performance software design. The system ultimately aims to bring a sense of active creation to laptop performance.

1.2 Project Overview

The focus of this research project is the creation of a software system for computer music performance. The system aims to be a flexible and expressive alternative to typical performance software that offers the artist increased agency in laptop performance. Its design eliminates the use of prepared control sequences and emphasizes the performer's on-stage musical control. It allows for improvisation, and induces a significant amount of variability in performance.

A specific goal for the software design is the live creation and control of multiple simultaneous voices. The approach is to allow musicians to create graphical, generative patterns that drive synthesis. We ultimately wish to be able to allow a performer to create layered pieces on the fly. We would like to shift users away from being "pilots" of their pieces and more toward being performers of them.

The interface is designed as a reactive, direct-manipulation simulation that resembles a freehand drawing application. The animated strokes interact, resulting in looping and cascading animated patterns that are subsequently mapped to sound. By drawing and combining groups of strokes, the user plays multi-voiced music.

1.3 Thesis Overview

The remainder of this thesis is organized as follows. Chapter 2 provides a survey of current software interfaces for computer music performance. Examples are given that show how an interface may encourage the use of prepared control mechanisms. Other interfaces are also presented that show creative system designs that take advantage of the unique affordances of software systems. Chapter 3 describes the design and implementation of the thesis project software. Design decisions, technical details, and lessons learned are presented. Chapter 4 presents a discussion and evaluation of the project. Chapter 5 details the thesis work's conclusions and suggests avenues for further research.

1.4 Contributions

The contributions of this thesis work are the exploration and articulation of issues present in typical laptop performance practice; the design and implementation of a prototype system that aims to improve on other performance interfaces; and this thesis document, which describes and explores the research in detail.

Chapter 2

Background

Live computer music is becoming increasingly feasible in the advent of practicable performance software solutions. This chapter surveys available software tools that allow complex, multi-layered computer music to be performed live. We will see that, to varying degrees, these interfaces emphasize the use of prepared material to reduce the complexity of the performance task. While effectively managing complexity, this also has the detrimental effect of compromising the performer's on-stage control and flexibility. These examples will serve as lessons for the design of improved computer music performance software. Finally, novel interface examples are presented that demonstrate the breadth of creative design possibilities for performance software.

2.1 Software Control of Computer Music

The trend in computer technology toward faster, smaller computers has allowed computer musicians to fit their entire studios into a single laptop. Computer power is much greater today than it was even ten years ago, and musicians' large hardware synthesizers and dedicated DSP units have been replaced by software applications running on general-purpose computers. Many of today's electronic musicians work entirely in software, and some have never made music with hardware tools. Software interfaces are popular for computer music performance because they are powerful, portable and accessible: software implementations are much more flexible than their hardware counterparts, and they inherit the power of today's computers; software takes up no physical space; and the applications usually only require standard computer equipment.

This shift in emphasis toward software brings its benefits to computer music performance. The computer can be played in unique ways that have no analogue in physical instruments. While hardware interfaces are bound by physical constraints, graphical user interfaces can be extremely malleable and dynamic. On-screen objects need not be passive; they can be reactive, exhibiting distinctive behaviours. These can be directly manipulated by the user, borrowing metaphors from our everyday experience and creating a rich interactive experience. Further, an interface's graphical, animated visualization can capitalize on human cognitive and perceptual properties, increasing its effectiveness.

Golan Levin describes three dominant interface metaphors used in audio creation software: scores (i.e., timelines or diagrammatic layouts), control panels and interactive widgets (2000, p. 34). Of these, score interfaces and control panels are most common in music systems. These interfaces are well-suited to compositional tools that focus on the offline editing of music. They allow the isolation and precise manipulation of any of the large number of variables that govern a piece: for example, volume curves, effects settings, and audio splicing. The bulk of music software is designed to be used in a compositional context, and these types of interfaces are largely mimicked in performance software. As we shall see, however, this focus on editing is not very conducive to live performance.

This software is often used to make types of music that are rooted in deejay and dance cultures. These pieces are typically structured as multiple layers of looping audio material. While adhering to this *modus operandi,* the more experimental of the non-academic electronic music producers have moved forward to explore new, uniquely digital phenomena: "glitches, bugs, application errors, system crashes, clipping, aliasing, distortion, quantization noise, and even the noise floor of the computer sound card" (Cascone, 2000, p. 13). This new music has been dubbed "post-digital" by Cascone, and is also referred to using the catch-all "laptop music." This emphasis on layering and looping is strongly apparent in the design of popular music software, and many applications are designed with this compositional style in mind.

2.2 Issues in Laptop Performance

While the shift toward software tools is appealing for computer music production, it leads to frequently unsatisfying laptop performance for the following reasons. First, complex, layered pieces are generally hard for a single performer to play. Performance compromises must be made to execute such pieces since our cognitive and physical capabilities limit the amount we can do and keep track of at once. Second, the tools that performers use to play their music live are not necessarily well suited to real-time use as they draw on a heritage of studio-centric tools.

Contemporary popular computer music is typically created by one or two producers in a studio environment. The many complex and subtle layers usually used add life and depth to a piece, but such complexity is obviously difficult for the same people to reproduce live in real-time.

Composition and editing are offline, non-real-time activities, and interfaces that cater to these tasks can hinder real-time performance. The live manipulation of large numbers of individual parameters through a control panel is quite difficult. Hardware controllers featuring knobs and faders that map to musical parameters are an improvement over using on-screen controls with a mouse, but there is still an upper bound on the number of simultaneously controllable parameters.

Musicians must make compromises to reduce the required amount of on-stage control bandwidth by letting the computer take care of some of the work. Audio loops, sequences, algorithms and presets are prepared in the studio beforehand; on-stage, these are triggered and mixed, and effects parameters are manually perturbed through control panel interfaces. This works well to reduce the amount of work one needs to do on-stage, but it has the unfortunate side effect of making for an uninteresting performance. The musician now has too *little* control of the music, and the performance itself becomes less engaging for the performer and for the audience.

Another reason behind the use of prepared material is that it can ensure a flawless performance. Artists generally strive to perfect their performances through practice. Acoustic musicians practice intensively to be able to render pieces with as few mistakes as humanly possible. Computer action is always perfect, and it is easy to ensure that this goal is attained. So, while the drive toward perfect performances is natural, the fact that the perfection is computerized makes it uninteresting.

The interface used in creation strongly colours the musical results as well. "We believe the interface…is an important factor in a musical performance. … Musicians working with acoustic instruments know very well how an instrument can have a unique personality which makes you play differently depending on its character" (Magnusson, 2005, p. 212). As well as hindering real-time performance, these interface designs also strongly impact the types of music that they can be used to make. An interface's assumptions about the compositional process are imposed on the final musical product.

We see that it is difficult to perform complex, multi-layered music to begin with, and that musicians have to make considerable performance compromises when using tools that are better suited to offline editing tasks. The remainder of this chapter examines popular computer music performance systems and their control possibilities, discussing how they are used in performance and how we can improve on them, as well as novel software interfaces that move away from these design patterns.

2.3 Loop-oriented Performance Software

In this section, we examine performance software that emphasizes the use of audio and MIDI loops. These systems are some of the most widely used for computer music production. Their control panel interfaces and loop-oriented design put them squarely in the category of offering limited on-stage control.

Ableton Live (2006) is the most widely used performance software available. While its interface is innovative in comparison to its peers and its design focuses on live use, it still fundamentally de-emphasizes on-stage flexibility and requires the use of prepared material.

Live's performance mode is called its "session view," pictured in Figure 2.1. Sample loops are arranged in a grid; columns correspond to mixer tracks, and rows ("scenes") correspond to preset groups of loops across all tracks. Each cell and row features stop and start buttons for triggering a particular sample clip or group of clips. Generally, each row corresponds to a different part of the song, and the performer clicks on the play buttons on each row as she moves through its different parts. This nonlinear organization of audio clips allows a performer to mix, match and navigate through prepared audio material. Clip playback can be configured in different ways (quantization, play-

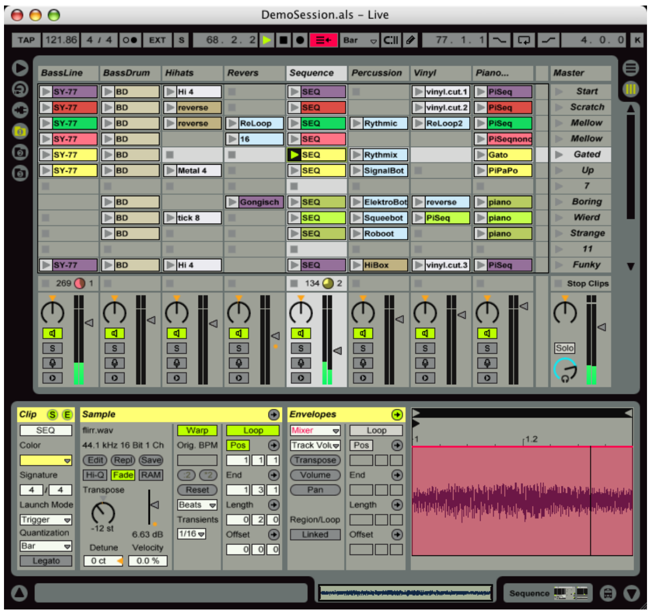


Figure 2.1 Ableton Live's session view

back style), and audio effects parameters can be subject to various kinds of automation. The system is designed to support the live manipulation of its controls, and can be used with the mouse or with external MIDI hardware.

The interface is aesthetically pleasing and well designed, and the tool is rightly very popular among computer music producers. However, its use remains in the "prepare and pilot" paradigm. The interface largely follows a control panel design. Ultimately, control is limited to triggering events (starting and stopping tracks, launching audio clips) and modifying individual parameters (mixer settings, effects controls). Creating tracks from scratch cannot be done easily in performance, and the session view grid needs to be prepared before going on-stage. Clips are usually set up to come in on the beat so the tracks stay synchronized. While it is possible to use the system more abstractly, the interface is very strongly biased toward using layers of synchronized loops.

Propellerhead Software's *Reason* is another very popular application for electronic music production and performance. Its interface is modelled after older analogue hardware interfaces, and its main window appears as a rack of mixers, synthesizers and samplers, as seen in Figure 2.2. Units are connected via patch cords that are accessible through the "back" of the rack. These units can be patched into other synthesizers for

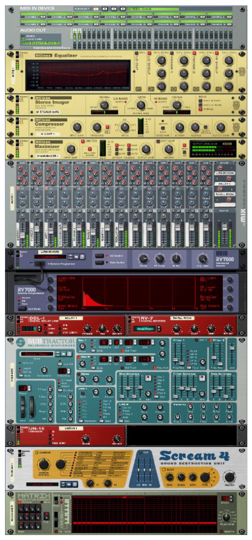


Figure 2.2 A Reason rack

different effects, creating an analogue-studio style of production. Reason's interface also follows a control panel design in keeping with the hardware it mimics.

The main way Reason is used in performance is through the sequencing capabilities of its step sequencer and drum machine modules. The units can store a number of bar-long preset sequences. During playback, the user can switch between different sequences, effecting changes in the music. It is not practical to edit these sequences on the fly, and so they must be prepared beforehand as presets. Like Live, performance with Reason is more or less reduced to switching between different preset sequences and individually perturbing effects parameters.

Reason's Combinator unit is specifically designed for switching between prepared performance patches. It acts as a container for a whole rack of synthesis and sequencing units. A performer can switch between these patches to move between different songs in a set. Like Live, however, the synthesis patches are extensively prepared, and the sequencing is assembled beforehand.

2.4 Algorithm-oriented Performance Software

Performance interfaces do exist that are more flexible and make fewer assumptions about the music the user is trying to create. These emphasize general-purpose com-

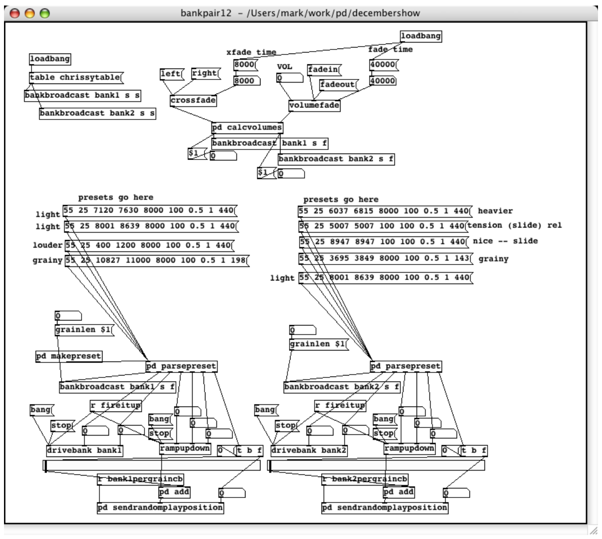


Figure 2.3 A Pure Data patch

putation and programming, allowing the user to create music-making machinery as she sees fit. These tools are regarded as more academic and involved since they require a certain amount of effort and understanding on the part of the user. It takes a significant amount of work to get these systems prepared for live performance, but the rewards are potentially greater. We will see, however, that even though these systems offer increased flexibility, they still require that the algorithms and musical material be prepared to varying degrees.

2.4.1 Dataflow Interfaces

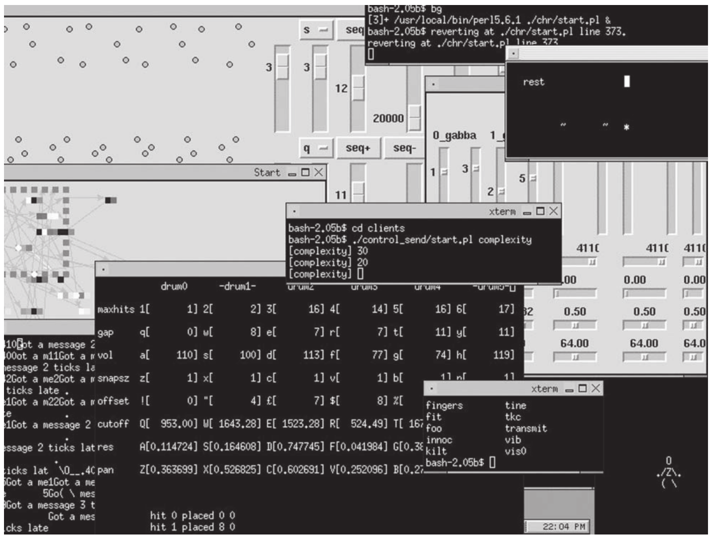
The most popular style of algorithm-oriented system is the *dataflow interface*. The seminal examples in audio software are *Max/MSP* and *Pure Data* (Puckette, 2002). They allow the graphical specification of DSP and control algorithms. Graphically, the interfaces resemble a flow chart or directed graph, as shown in Figure 2.3. Simple operators are connected together on-screen to define the algorithms. Control and audio signals propagate through the graph and are transformed at the nodes. In this way, the user can assemble procedures for sculpting the audio and for creating algorithmic control patterns. The graphs (also known as "patches") can be assembled in real-time as audio is computed, allowing an iterative, interactive development process.

These dataflow languages are frequently used for performance. Graphical control widgets can be embedded directly in the patch for real-time manipulation by the user. Buttons, sliders, grids and other widgets are available. Essentially, a performer creates a customized control panel for her custom music processing system. Performances thus consist of control panel-style manipulation of the musical algorithms. We see that again this is a "prepare and pilot" system, where the vast majority of the work in the piece is preparatory, and relatively little control and effort is exerted on-stage.

This is offset somewhat by the potential complexity of the underlying patch structure—it need not conform to the loop-oriented paradigm of the interfaces in the previous section. Also, it is conceivable to patch together operators on the fly in performance (Collins et al., 2003). However, the operators available are relatively low-level, making it potentially difficult to assemble an interesting patch live. Libraries of medium-level objects could be used to reduce the amount of work required in performance, but a balance must be struck between what is live and what is prepared. Further, the general-purpose programming nature of these languages does allow for the creation of more complex control structures beyond control panels.

Reaktor (Native Instruments, 2006) is a similar dataflow application, allowing users to assemble complicated DSP algorithms graphically from simple components. On-screen control panels resembling analogue hardware are assembled to control the various parameters, and the patches can be performed in the same way as with Max and Pure Data. Due to these similarities, Reaktor can exhibit the same performance problems as other dataflow languages.

Dataflow interfaces still feature some of the issues we would like to address: patches must largely be prepared in advance, and are most often played through predominantly automated means. However, we see an improvement on the rigid software systems presented in the previous section. Fewer assumptions about the music and compositional structure are made than in the previously explored interfaces. The notion of algorithmic music and the potential of interactively building one's musical machinery on-stage is a perfect example of how computer performance can be unlike any other kind of musical performance known to date.



```
bash-2.05b$ bg
[3]+ /usr/local/bin/perl5.6.1 ./chr/start.pl &
bash-2.05b$ reverting at ./chr/start.pl line 373.
reverting at ./chr/start.pl line 373
```

```
rest

    ~     ~  *
```

```
bash-2.05b$ cd clients
bash-2.05b$ ./control_send/start.pl complexity
[complexity] 30
[complexity] 20
[complexity] 
```

```
             drum0        -drum1-

maxhits 1[         1] 2[         2] 3[        16] 4[        14] 5[        16] 6[        17]

gap     q[         0] w[         8] e[         7] r[         7] t[        11] y[        11]

vol     a[       110] s[       100] d[       113] f[        77] g[        74] h[       119]

snapsz  z[         1] x[         1] c[         1] v[         1] b[

offset  ![         0] "[         4] £[         7] $[         8] %[

cutoff  Q[    953.00] W[   1643.28] E[   1523.28] R[    524.49] T[ 167

res     A[0.114724] S[0.164608] D[0.747745] F[0.041984] G[0.38

pan     Z[0.363699] X[0.526825] C[0.602691] V[0.252096] B[0.27


         hit 0 placed 0 0
         hit 1 placed 8 0
```

```
10Got a message 2
00ot a m11Got a m
essage 2 ticks la
2Got a me2Got a m
ticks late .
1Got a m22Got a m
e
1Got a message 2

ssage 2 ticks lat

icks lat `\0__.40
Got a me1Got a me
       5Go( \ mes
Got a message 3 t
       Got a mes
cks late
```

```
fingers         tine
fit             tkc
foo             transmit
innoc           vib
kilt            vis0
bash-2.05b$ 
```

```
  0
./Z\.
 ( \
```

Figure 2.4 A live coding desktop, from (Collins et al., 2003)

2.4.2 Live Coding

Live coding (Collins, 2003; Collins et al., 2003) is an excellent example of a uniquely computer-based musical performance technique. Its practitioners "work with programming languages, building their own custom software, tweaking or writing the programs themselves as they perform" (Collins et al., 2003, p. 321). The interaction is text-based, requiring that the performers work within a text editor and perhaps a command shell, manipulating running algorithms as one would operating system processes. Figure 2.4 shows an example live coding desktop. Performers sometimes employ utility libraries designed for live use, or may program as much from scratch as they can. Most often, these performers use interpreted real-time audio languages, but can also use more generic programming languages and tools coerced into musical uses. Two popular examples of such real-time audio languages are *Supercollider* (McCartney, 1998) and *ChucK* (Wang and Cook, 2004).

Supercollider is a real-time programming language for audio. Its syntax is based on Smalltalk. The application interprets commands typed into its main window, much like a Lisp listener. The resulting sound is audible immediately, and the sound can be changed interactively by modifying and re-sending textual expressions. Supercollider is popular for live coding since the language supports all standard programming constructs and allows powerful audio manipulation. Graphical control widgets can be created as well, further augmenting the control possibilities. The latest version of Supercollider separates the code interpreter from the audio renderer, allowing the two to communicate via a network connection. This architecture could allow for interesting cooperation between multiple live coders.

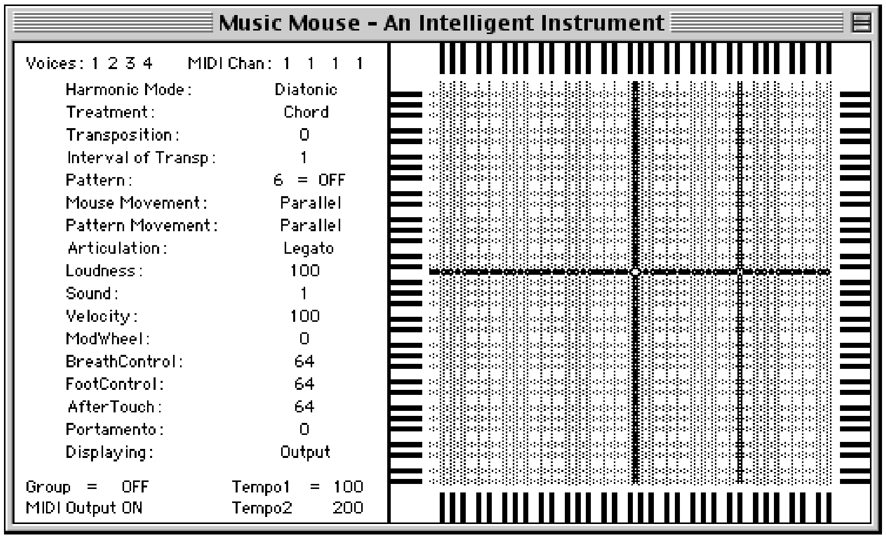
ChucK is also a real-time programming language for audio whose syntax resembles C/C++. Its most distinctive feature is the way it handles timing. Timing interrupts can occur anywhere in a user's algorithm, allowing user-defined control rates of any granularity, even down to the sample level. The basic ChucK installation does not feature an interactive, graphical interface. Instead, all manipulations are done through standard text files and the command line. This brings the live coding experience even closer to traditional computer programming.

We see in live coding a very different way of creating computer music, moving away from visual representations and toward textual encodings of algorithms. The low-level, general-purpose programming emphasis of these techniques makes them extremely flexible. Live coding is unique to computer music and serves as an inspiring example of the novel performance techniques that computer technology makes possible.

However, the paradigm is not without its drawbacks. Live coding in performance appeals only to a niche market of artist-programmers who possess a relatively significant amount of programming experience. This barrier makes the techniques inaccessible to less technically inclined artists. Live coding makes laptop performance even less accessible for the audience as well: even if they were to see what was going on on-screen, most members would probably not understand it (Collins, 2003). Further, the textual interaction required for these systems is cumbersome and could hinder expressivity. A live coding performance might require non-stop typing from the performer, and it might take a relatively long time for a particular musical idea to be realized. Also, depending on the way the code is used, an implicit control panel-style interaction could be a consequence of this scheme—individual parameters might need to be changed one at a time through textual commands.

2.5 Novel Graphical Performance Software

Graphical performance interfaces exist that move away from the designs we explored in the previous sections. These interfaces also capitalize on the unique affordances soft-



```
Voices: 1 2 3 4     MIDI Chan: 1  1  1  1
     Harmonic Mode:          Diatonic
     Treatment:               Chord
     Transposition:             0
     Interval of Transp:        1
     Pattern:               6  = OFF
     Mouse Movement:          Parallel
     Pattern Movement:        Parallel
     Articulation:            Legato
     Loudness:                 100
     Sound:                     1
     Velocity:                 100
     ModWheel:                  0
     BreathControl:            64
     FootControl:              64
     AfterTouch:               64
     Portamento:                0
     Displaying:              Output

Group  =   OFF          Tempo1   =  100
MIDI Output ON          Tempo2      200
```

Figure 2.5 Music Mouse

ware interfaces provide, featuring novel feedback and control mechanisms. They are more experimental and are used by fewer performers than ones cited above, but they represent an interesting middle ground between designing one's own algorithmic mechanisms and accepting conventional interface designs.

2.5.1 Music Mouse

Music Mouse (Spiegel, 2004) is one of the earliest real-time software performance instruments, developed by American composer and software developer Laurie Spiegel. It was originally created in 1985 for the Macintosh, Amiga and Atari platforms. The software was intended to be usable by non-expert musicians, allowing the music to be explored intuitively in real-time. The program allows very high-level control of algorithmic music using the mouse and keyboard commands. MIDI notes are generated procedurally based on this input and are sent out to external synthesis units. Mouse movements control the output pitch by displacing a set of vertical and horizontal bars in the interface, seen in Figure 2.5. These bars intersect on-screen piano keyboards, selecting pitches.

The sequencing algorithms of Music Mouse are fixed; they were developed by Spiegel to automatically produce sensible music based on her personal tastes. It is billed as an "intelligent instrument" since the program makes its own musical judgments. As such, Spiegel considers the program at least a partial collaborator in the compositional process.

The system was designed from a compositional perspective, and the available parameters reflect this. These include harmony (chromatic, diatonic, pentatonic, middle eastern, octatonic, quartal), transposition, voicing, and rhythmic treatments (chord, arpeggio, line, "improvisational"), among others.

Music Mouse is intended to control all voices in an entire piece of music at once through a single interface. Spiegel "want[s] to work on the level of playing a full symphony orchestra plus chorus as a single live instrument" (Gagne, 1993, p. 313). This is precisely the goal we wish to achieve in this thesis work, and is much like what today's electronic musicians aim to do in performance.

2.5.2 FMOL

Sergi Jordà's *FMOL* interface (2002) allows a unique interaction with its sound producing mechanisms. It attempts to break new ground in musical interaction instead of simply mimicking physical instruments or audio hardware. The program was designed to appeal to a wide audience, interesting to both novices and experienced music-makers, and to allow collaboration over the Internet. These factors constrained the software design to work with standard computer hardware, focusing on a pure software interface for music making.

The on-screen GUI elements, shown in Figure 2.6, are used for input as well as feedback, and there is meant to be a direct correlation between the sound produced and the state of the interface. The synthesizer parameters are reflected in "symbolic and non-technical" ways in the interface (Jordà, 2002, p. 31). The audio produced by each voice appears on-screen via an oscilloscope-style display that is directly manipulated by the user to affect it. These appear in the interface as six independent "strings," arranged vertically to resemble a guitar. Each string corresponds to a synthesis chain, consisting of a generator and a stack of three sound processing modules. Each generator and processing unit is controlled by two input variables, and all these variables are accessible through the same FMOL interface (called "Bamboo"). The software has preset facilities for quickly moving to a particular audio state, but performance consists of direct real-time manipulation of the interface. Jordà uses the interface for performance as a member of the FMOL trio.

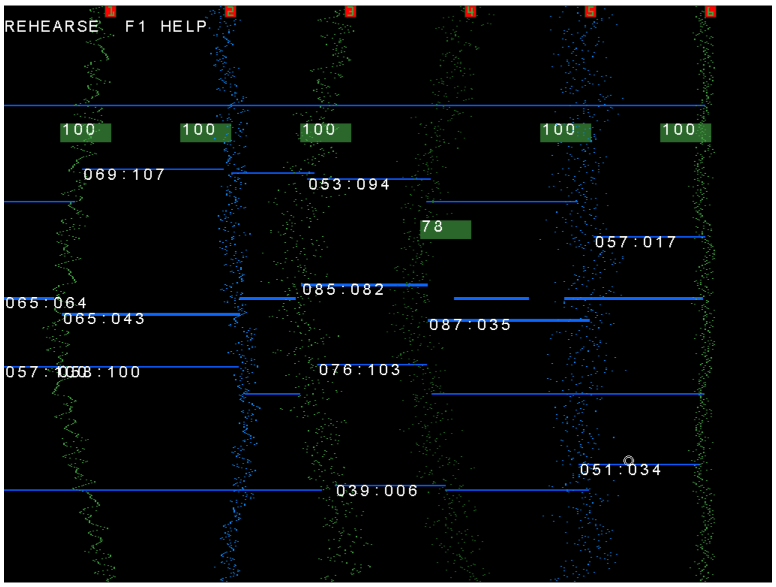


Figure 2.6 FMOL

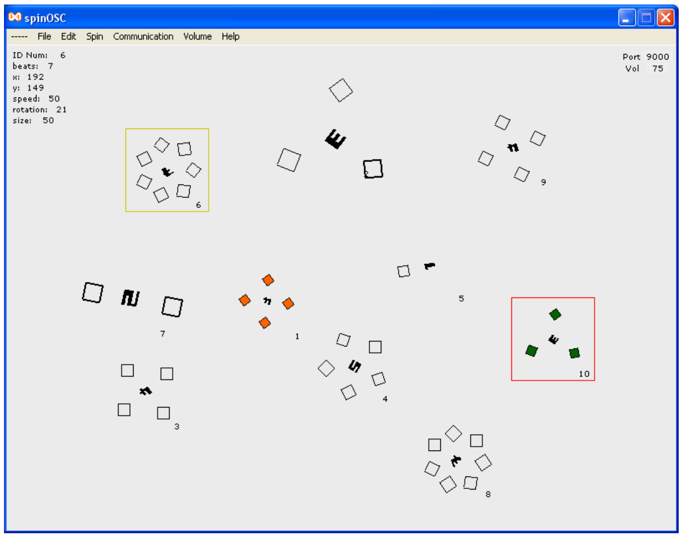
2.5.3 ixi software

The *ixi software* design group is responsible for a number of creative software instruments that have no analogue in other domains. They believe that:

> [C]omputer music software and the way their interfaces are built need not necessarily be limited to copying the acoustic musical instruments and studio technology that we already have, but additionally we can create unique languages and work processes for the virtual world. The computer is a vast creative space with specific qualities that can and should be explored. (Magnusson, 2005, p. 212)

ixi is interested in how the interface affects the music we make through the ideas and limitations it imposes on the creative process. They concentrate on the design of the user interface. The sound generation is handled by third-party tools like Pure Data, and the two communicate over a network socket using the OpenSound Control (OSC) protocol (Wright et al., 2003). They have a number of interface designs, all featuring this basic architecture.

One example is their *SpinDrum* interface, seen in Figure 2.7. The interface appears as spinning circles made of one to ten boxes. A sample is associated with each wheel, and



ID Num: 6
beats: 7
x: 192
y: 149
speed: 50
rotation: 21
size: 50

Port 9000
Vol 75

Figure 2.7 ixi's SpinDrum interface

sequences can be created by activating and deactivating individual boxes in the wheel. When an activated box reaches the twelve o'clock position, the sample sounds. Further control of pitch and panning are possible by changing the position of the entire wheel in the x/y plane. The interface results in polyrhythmic sequences, and fosters a sense of interactivity and exploration.

Another interesting example of ixi's work is their *Picker* interface. The program's main window displays an image or a video feed, and a set of four positions in the window are chosen by the user via crosshairs. The RGB values at each position are continuously sent out via OSC. These message streams can be sent to a synthesis program and used to control audio processes. The crosshairs can move randomly around the window, or the user can supply a movement trajectory. Repetitive video can be used to create rhythmic content in the audio. Further, if the video input is a webcam, a performer can use bodily movements and other visual phenomena to drive the music.

2.5.4 Sketching Interfaces

Drawing and sketching gestures are very natural for humans because we use them on a daily basis for communication. These same gestures are therefore useful for musical control. They are highly communicative, and are related to other physical gestures we make in everyday life. Sketching is precise and of very high bandwidth, simultaneously communicating not just (x, y) position, but pressure, angle, speed and acceleration. All of these variables can be used for control. Further, our physical sketching movements are naturally sensed through proprioception. In this section, we examine work that exploits sketching gestures for musical control.

While strictly a non-performance composition tool, Iannis Xenakis's *UPIC* system is an important example of the use of a graphic tablet in a musical creation system (Lohner, 1986; Marino et al., 1993). It was first realized in the mid-1970s on mainframe hardware. It has since undergone a number of reimplementations, culminating in a version for PC systems. In designing the system, Xenakis wanted to free himself from the relatively rigid encoding of notes on a staff and move toward continuous relationships between pitch and time. This is reflected in the notion that "the system should not impose predefined sounds, predefined compositional process, predefined structures, and so on" (Marino et al., 1993, p. 260).

The system's central concept is the (x, y) plot, which is edited via the graphic tablet. These plots are used for many different purposes at different time scales: they are used for working at the "microcompositional" level (audio-rate waveforms) as well as at the "macrocompositional" level (control-rate pitch versus time curves) (Lohner, 1986). A composer first creates audio waveforms by drawing, iteratively refining them until the desired sound is achieved. Then these sounds are sequenced on a separate page by drawing sets of pitch versus time trajectories and assigning a waveform to each one. This could be understood as a kind of continuous piano-roll sequencer, making smooth effects like glissandi easy to achieve. Pieces of audio from previous computations can be assembled and mixed together, allowing one to iteratively build a piece from these individual fragments. Additionally, sine tones, FM synthesis and sampled sounds can be used in place of the hand-drawn waveforms. There is also another layer of indirection for pitch and time: vertical position can be mapped to pitch through a "frequency table," and timing can be adjusted through the "sequence," essentially a read-position versus time curve.

This is an important example of musical ways to use two dimensional spaces via a graphic tablet. A significant point to note is that the plots used in UPIC are fundamentally diagrammatic: an external definition of what the vertical and horizontal axes mean is necessary to make sense of the user's marks.

Wright et al. (1997) explored the use of the digitizing tablet for musical control. They created a high-level model for using the tablet for musical applications that featured a generic set of recognized events that were then mapped onto particular musical models. The tablet was divided into regions, and a number of discrete events and continuous parameters were detected (pointer enters/leaves a region, pointer touches tablet, pointer button press, (x, y) position, (x, y) tilt, pen pressure). Both the tablet's stylus and puck were used, allowing two-handed input. These events were used to drive models of stringed instruments, as well as models for timbre space navigation. Events were transmitted to the models via MIDI and OpenSound Control. Since the tablet sensed absolute position, a physical, printed map of the regions could be laid onto its surface to guide the performer. This obviated the need for graphical feedback via a computer monitor.

A set of very inspiring examples comes from Golan Levin's Master's work (2000). His research focused on the creation of software interfaces for the simultaneous creation of synchronous abstract audio and visuals. All of his thesis pieces use the mouse to capture gestural sketching input from the user. He created a number of prototypes over the course of his research that explore these ideas. Some of his work uses the input gesture trajectories as animation data, an idea that was very influential for the present thesis work. The three interfaces presented here are all creative and aesthetically compelling, which positively impacts the experience for both the performer and the audience.

Levin's *Yellowtail* interface features animated versions of the user's input strokes. The animated strokes move across the screen, cyclically repeating their drawing gesture. They are sonified in real-time using an inverse spectrogram method. The screen is considered to be a spectral plot, and the positions of the animated strokes dictate which frequencies are played at each given instant. The input to the inverse spectrogram is a horizontal line that sweeps periodically through the interface's animated image space. The strokes move across the screen like live creatures, and the audio is synchronous with their motion. The screen acts as a continuously changing spatial score for the audio output.

The *Loom* interface, seen in Figure 2.8, builds on the ideas set out in Yellowtail. However, it does away with the diagrammatic nature of the space imposed by the inverse spectrogram sonification. The user's strokes are considered spines along which a timeline is wrapped. The timelines record the temporal evolution of secondary parameters along the stroke's length, and are used in turn to drive FM synthesis. The temporal dynamics of the stroke are recorded, and it is played back repeatedly, creating a periodic

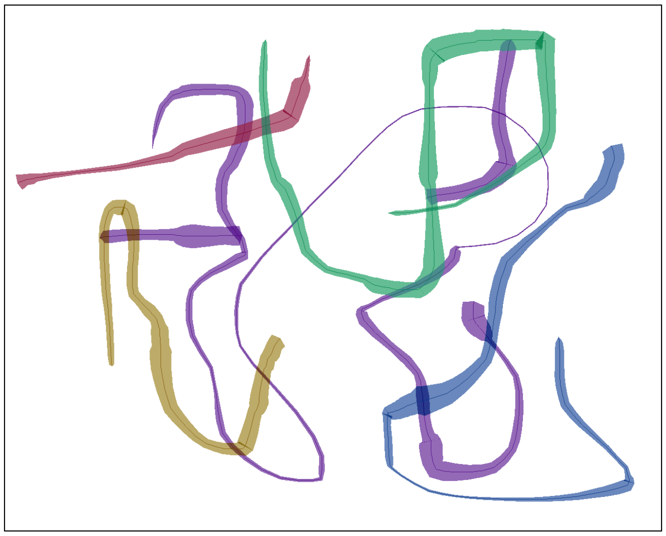


Figure 2.8 Loom, from (Levin, 2000)

animation. The timeline information includes position, velocity, pen pressure, and local curvature over the stroke's length; these variables are mapped onto the FM synthesizer, making periodic audio in sync with the video.

Levin's *Warbo* interface is the last of those discussed in his Master's thesis that use stroke trajectories as their main focus. The user deposits coloured, animating polygons and circles on the canvas. These are surrounded by colour fields that are brightest at their centre, and that overlap each other. The cursor is passed through the canvas, and a chord sounds based on the intensity of the intersected fields. Each source is associated with a sine tone, so this sonification amounts to additive synthesis. The spots are "played" in this way, creating sound output. A pen and graphic tablet are used for input, and the performer uses the puck in her other hand to control the timbre. The shape of the non-dominant hand input controls a nonlinear waveshaping effect that controls the brightness and spectral complexity of the resulting sound. This contributes to a more interesting audio output than would be exhibited by mixing simple sine tones.

The final example presented here is Amit Pitaru's *Sonic Wire Sculptor* (Pitaru, 2003). The software creates rotating, three-dimensional "sculptures" from freehand input trajectories. As the user draws on the screen, the drawing plane is rotated through space, resulting in a cluster of meandering, curvilinear filaments. A planar slice of the rotating sculpture is taken at a given location and mapped to sound; the spatial positions of these intersections thus control audio voices. This scheme allows a performer to interactively create looping, multi-voiced music that builds as the sculpture is assembled.

2.6 Summary

We have seen that software performance interfaces offer tremendous potential for live computer music, making it possible to play complex, multi-layered pieces. However, the most popular loop-oriented interfaces reduce the control complexity by fixing aspects of the music, thus limiting on-stage flexibility. Some alternative interfaces improve on this situation by taking a more general-purpose approach, allowing the user to create her own performance systems, but these still feature relatively low on-stage control bandwidth. Finally, some interfaces take novel approaches, leveraging the computer's dynamic graphical and processing capabilities to allow richer control possibilities. The next chapter presents the core of this thesis work, a graphical performance interface that aims to improve on these interfaces and allow the real-time simultaneous control of multiple voices.

Chapter 3

Design and Implementation

This chapter describes the thesis work itself. The project focused on the implementation of a software system for computer music performance and improvisation. The underlying goals and design ideas are presented to illustrate the initial seeds of the project. The project's user interface and technical designs are described, presenting its workings on both high and low levels. Early experiments and the resulting lessons are given. Finally, some of the challenges encountered over the course of the project are detailed.

3.1 Goals

The main goal of the thesis work was to create a software performance system that brings a sense of active creation to laptop performance. As we saw in the previous chapter, conventional performance software interfaces can compromise live control and can encourage a reliance on heavily automated control mechanisms. This project aims to eliminate this trend by disallowing the use of prepared material. A more specific subgoal was to allow for the live creation and modification of multiple, simultaneous, generative control patterns. The project targets dance, "glitch," and other related computer music styles that tend to compose pieces as layers of looping audio material. The live creation of generative patterns would allow these performers to play pieces by defining and layering loops on the fly.

Another goal of the project was to explore the notion of using a spatio-temporal, direct-manipulation interface for controlling these generative processes. The use of animated on-screen objects that embody the logic and action of parallel control structures could help a user to visualize and understand them. Directly manipulating these objects could make for a more intuitive user experience, "physicalizing" the processes and providing a visual, symbolic, non-numeric method of controlling the system. A perceptual fusion between user action, animation and sound could further reduce the user's cognitive load. This style of interface could take advantage of the potential dynamism of screen-based software interfaces.

An ancillary goal was to create a usable piece of software that could be distributed for use by computer music performers. The software design should allow for extension and expansion. It should provide a clean framework within which to experiment with interface design ideas.

3.2 Interface Design Ideas, Heuristics

In addition to the above goals, the interface design process followed a set of loose heuristic principles. These were meant to influence the character of the interface design. Each are treated in turn below.

The system concepts should be minimal, but not shallow. The workings of the system should be easy to understand, but that fact should not detract from its expressivity and depth. This idea is embodied in various examples in games and mathematics, where very simple systems can give rise to very complex interaction and behaviour. For example, chess has a fairly simple set of rules, but it is incredibly deep and subtle. This is an attractive idea that should be integrated into our interface design. Instead of providing high-level functions that have obvious musical uses, we would prefer to provide more minimal atoms and encourage users to exploit them in oblique, creative ways. This could contribute to the overall richness and potential of the system. Note that this idea could also impact novices' experiences with the system: effective use might require in-depth experience with it.

Object interaction should be based on their relative spatio-temporal relationships. While meaningful x and y axes can be very useful for control, it can also be interesting to refrain from assigning them any particular function. This idea comes from real space: there are no defined axes in open space, and relationships are based on physical contact and proximity. Interesting systems that feature emergent behaviour often use adjacency for defining relationships between actors. This reinforces the notion of parallelism as well, since every point in the space is equal, and every actor in the system operates only from its own point of view. A good example of this is John Conway's Game of Life (Dewdney, 1988, p. 136); every cell acts from its own point of view, and only knows about the other cells immediately adjacent to it. Golan Levin's research features a similar requirement in its desire for a "non-diagrammatic image space" (2000, p. 98), reflecting the absence of axes from the artist's page.

There should be a perceptual connection between user gesture, animation and sound. The simulation should be as believable as possible to the user. A tight relationship between the user's gestures, the on-screen graphics and the resultant sound helps maintain the illusion that the user is interacting with a set of "physical" objects that inhabit a "real" space. Golan Levin touches on this in his thesis (2000, p. 100), showing that this is necessary to enhance the interface's usability.

The system should use pointer-style input. An early decision in the design process was to use a graphic tablet for input. As discussed in Chapter 2, pen gestures are natural for humans and convey a great deal of information. Shape, speed, direction and pressure are all conveyed continuously along the length of a stroke. An important property of freehand input is that it is subject to natural physical variability. Computer music often suffers from being over-consistent and unnatural, and the use of input pen gestures could help infuse the user's control with an organic quality. Further, this variability would ensure that pieces would not be precisely repeatable, but rather would vary between performances. A drawn stroke is also intimately linked with the physical movement used in its production, which could help reinforce the connection between the on-screen representation and the user's input. Further, it leaves open the possibility of gesture recognition as another means of communicating with the system. However, in order to keep the system accessible to users without specialized hardware, it should also be usable with a generic mouse.

The system should impose limits. Part of what makes acoustic performance interesting is that musicians push against the inherent limitations of the activity: physical human limitations (speed, endurance, precision) and the limitations of the instrument (control mechanisms, producible effects). The presence of limitations helps drive artists to use their tools in interesting, creative ways. This element should be present in the interface as well. Limitations help foster creativity and they give a system its flavour (Tanaka, 2000; Bridgett, 2005). Computer technology tends to emphasize precise control over all parameters, focusing on minute details instead of efficient creative output. The hope is that users will find creative ways to achieve their musical goals when presented with a more limited interface.

These goals and ideas helped colour the final system design. The following section describes this design in detail.

3.3 System Description

This section describes the system design. The application is a simulation resembling a freehand drawing program. The user's strokes create looping and cascading animated figures that are mapped to audio to play music.

3.3.1 Overview

The core idea of the simulation is that the user is defining pathways in the plane. The main program window, shown in Figure 3.1, opens as a blank canvas, and the user draws these paths using the pen. Each pen stroke defines one path, which can be drawn arbitrarily in the canvas.

"Particles"—small singular points—travel along these pathways. The particles are displayed as white points with fading tails trailing behind them. These particles only move along the defined pathways; that is, only along the drawn strokes.

The particles have a special action at the points where two paths intersect. When a particle reaches an intersection, it makes a copy of itself and deposits the copy on the other stroke. Each of the two particles then continues to travel as normal. The copy is deposited at the location of the intersection. What we see in the interface, then, is a single particle approaching a crossing point, and *two* particles emanating from it. This concept is illustrated in Figure 3.2.

These simple ideas—particles travelling along user-defined paths and replicating at intersections—form the entire basis of the system. Use of the system could thus be understood as the creation of path networks for particles to follow. These networks can feature branching, tree-like topologies, where particles move outward from the trunk and replicate along the branches. The networks can also feature cyclical topologies,

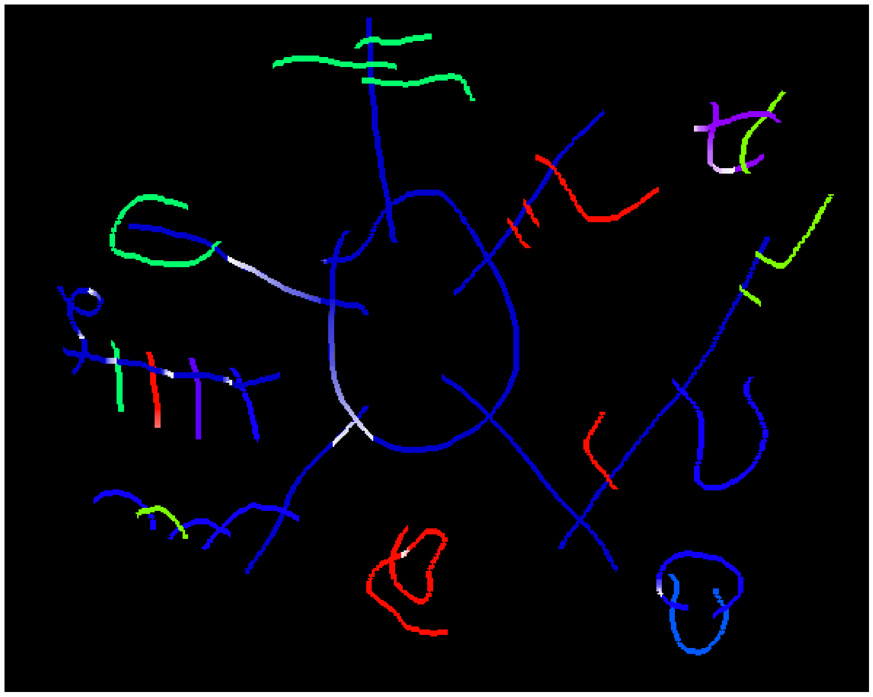


Figure 3.1 A screenshot of the thesis software

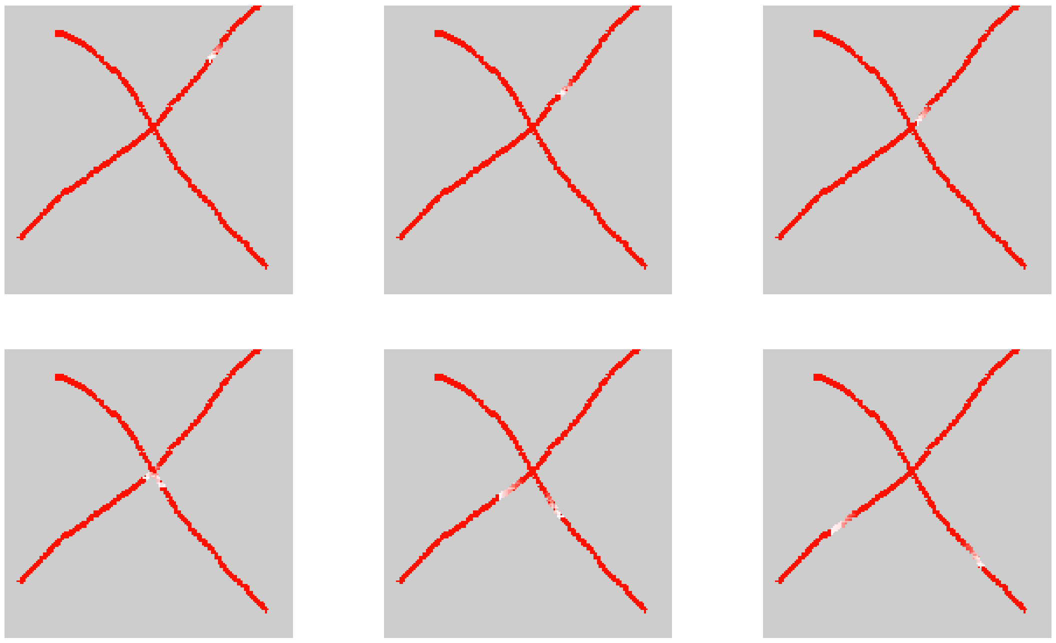


Figure 3.2 An illustration of particle behaviour at stroke intersections

where the particles spin perpetually through a loop. The particles flow through these networks as they are being built, so the visual experience that the system engenders is that of a person creating colourful line drawings as white particles move over and cycle through them.

All of the user's action consists of defining these freehand paths. The user does nothing to control the motion of individual particles; once the pathways are set, the particles move on their own. Where the particles come from and precisely how they move along the strokes are explained in the following section.

Consider the following illustration to reiterate the simple, core ideas behind the system's workings. The user's strokes can be seen as "streams" along which small "balls" float. When a ball reaches an intersection, it quickly makes a copy of itself and sends the copy on its way along the other branch, then continues travelling. This encapsulates the central ideas of the system. The rest of the details are simple additions to these concepts.

In summary, the interface allows a user to create topologies through which particles travel. The particles travel only along the user's strokes, and they divide at the points of intersection. Performance with the system is simply the creation of these networks of paths. We play music with the interface by sonifying the motions of the particles according to a particular mapping, described in Section 3.3.3.

3.3.2 Particle Motion

As the user draws a stroke, its spatial and temporal evolution are recorded by the software. This means that alongside the incremental pointer movements, the speed and timing of the pointer's trajectory are also recorded. This information is used to drive particle motion.

As a particle travels along a stroke, its motion always replays precisely that stroke's drawing gesture. This is a key notion that governs all motions of all particles in the system. If a stroke is drawn quickly, particles moving along that stroke will travel quickly. If a stroke is drawn slowly, its particles will move slowly. Any nuance in the timing of the drawing gesture is reproduced in each particle's movements.

Since the particles play back the drawing gesture, they always travel in the same direction along a given stroke—the stroke paths are unidirectional for particle movement. This means that the networks of paths the user creates act like directed graphs, specifying both direction and path connectivity.

An unlimited number of particles may move along a given stroke simultaneously. Since the particle motions along a given stroke are identical, particles will never pass each other while travelling along a stroke.

Particles are introduced into the system as the user draws. A particle always follows the user's cursor when drawing, causing new particles to be spawned immediately when intersections are first created.

So, in addition to defining paths along which particles may travel, the user's gestures dictate *how* the particles will move along those paths. This fact enriches the particles' patterns of motion, and allows a user to control the timing of the animated figures. This will eventually drive the timing of musical patterns.

3.3.3 Mapping to Sound

The animated motions of the particles through the stroke structures are mapped to sound synthesis. This is the last step before the final musical result. Note that the interface itself imposes no particular mapping; the following is the designer's choice for this incarnation of the project.

The idea behind the mapping is quite simple. We "stretch" an audio sample along the length of a stroke, so that points along the stroke correspond to points along the audio sample. Particle motion drives audio playback via this correspondence.

We can imagine a stroke to be a one-dimensional parametric curve embedded in the plane. The curve has a beginning and an end, and a coordinate value can be associated with each point along its length. An audio sample is also one-dimensional, has a beginning and an end, and features a coordinate system. We define a linear bijection (one-to-one mapping) between the audio sample coordinates and the stroke coordinates, matching the endpoints appropriately. Using this mapping, we can compute a position in the audio sample from a position along the stroke, and vice versa. The beginning of the sample maps to the beginning of the stroke, and the end of the sample maps to the end of the stroke. The stroke is parameterized by arclength, with respect to Euclidean distance and Cartesian coordinates in the plane. The mapping is illustrated in Figure 3.3.

As a particle moves along a stroke, we imagine it to be a playhead reading through the sample data. We convert the moving particle position to a moving point in the audio

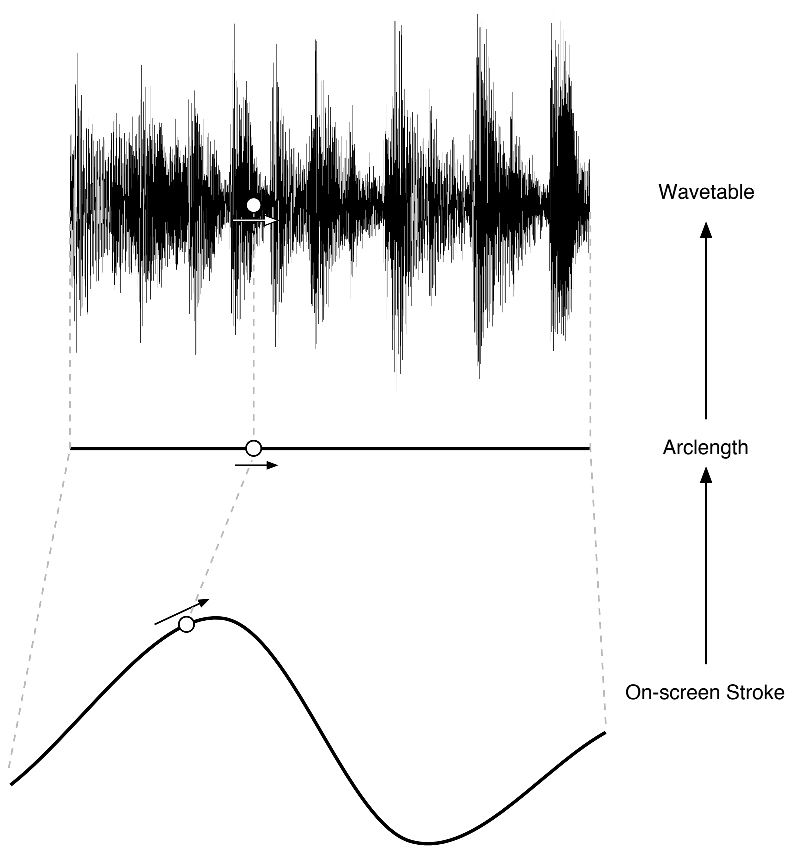


Figure 3.3 An illustration of the mapping from stroke to wavetable

file via our mapping, and read sample values from there. This is the basic mapping to audio.

There are some interesting consequences in combining this mapping with the simulation system. Since the drawing speed drives the particle motion, it also affects the speed of playback. If part of a stroke is drawn quickly, particles will also move quickly; this fast particle motion will drive high-speed playback, producing correspondingly high-pitched audio. Slow drawing gestures behave in kind, producing low-pitched audio. If the particle is introduced to the stroke halfway along its arclength, playback starts from halfway through the corresponding wavetable.

Since the drawing motion can never be executed at an exactly constant speed, this scheme leads to interesting playback effects—there is always continuous pitch variation in the audio output. The natural irregularity in the physical action leads to engaging variation in the corresponding sound. This is a deliberate feature of the system that prevents exactly reproducible audio output. Further, a musician can consciously vary the drawing speed to produce specific playback effects.

We allow one synthesis voice per stroke. A detail to consider is the fact that multiple particles can be present on a given stroke, but only one may control playback at any one time. This is resolved by defining the most recently added particle as the active playhead. If a new particle enters the stroke, it immediately becomes the new playhead and the sound jumps to the new position in the wavetable. If the currently active play particle leaves the stroke, the playback stops until a new one is introduced.

A bank of wavetables is made available by the system. Key commands are used to select which wavetable to associate with subsequently drawn strokes. The user hits the "1" key to select the first sample, for example, and subsequently drawn strokes will be associated with that wavetable. Each wavetable is automatically assigned a colour when the program starts, and the on-screen strokes are coloured according to their corresponding wavetable.

Also, not all strokes are necessarily associated with a wavetable. Some strokes remain silent but otherwise behave in the same manner as the other strokes. These can be used for control operations and for propagating particles where the performer does not need any sound output.

3.3.4 Performance Figures

A number of natural figures arise from this system design. The simplest is a stroke that loops once on itself, as shown in Figure 3.4(a). This self-intersection acts as a fixed orbit for a particle, and a new particle is emitted along one of the ends of the stroke on each revolution. This structure can be used as a periodic trigger. The speed of the loop depends on how the stroke was drawn: quick stroke trajectories will produce quick loops, and slower trajectories will produce slower loops. If the looped stroke is associated with an audio sample, the circling particle will repeatedly play back a section of the sample. This is an easy way to make a simple audio loop.

Looping structures can also be made from multiple strokes. For example, if three or four strokes overlap at their endpoints, they will transfer a moving particle repeatedly between them. This can be used in a manner similar to the one-stroke loop, but the

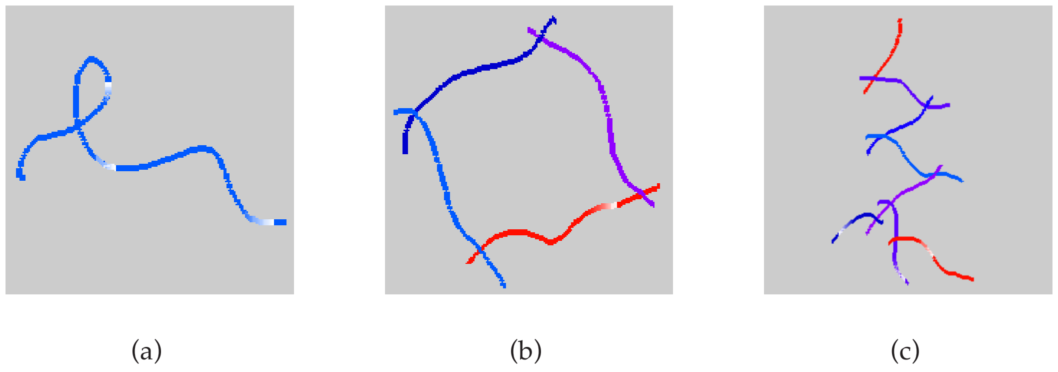


Figure 3.4 Typical performance figures: (a) looping with a single stroke; (b) looping using multiple strokes; (c) a non-looping cascade of strokes.

individual strokes could be associated with multiple samples to create different audio effects. This structure is shown in Figure 3.4(b).

The stroke topologies need not feature loops. A single stroke that does not intersect any others can be used to play back a sample once. It can be triggered by the user with a silent stroke; a short mark that intersects the sample stroke at its beginning starts the playback. Cascades of strokes are possible as well, where multiple strokes trigger one another in turn. This results in a domino-like pattern, as shown in Figure 3.4(c).

A single stroke that features sharp corners will have speed close to zero at those corners. This can be used to make a sequence of audio events from a single sample.

3.3.5 Summary

The salient features of the above system definition are reiterated here. The system resembles a freehand drawing interface where the user draws strokes in the plane. The networks of strokes define topologies through which small particles move and cycle. This on-screen animation is mapped to sound, and the particles' repeating patterns of motion drive corresponding patterns in the resulting audio.

The strokes define the paths along which the particles move. Particles replicate at the points of intersection between two strokes; this association between crossed strokes allows the user to define the above topologies. The particles' motion mimics the speed and timing of the corresponding stroke's original drawing gesture.

The motion of the particles is mapped to sound by associating a wavetable with

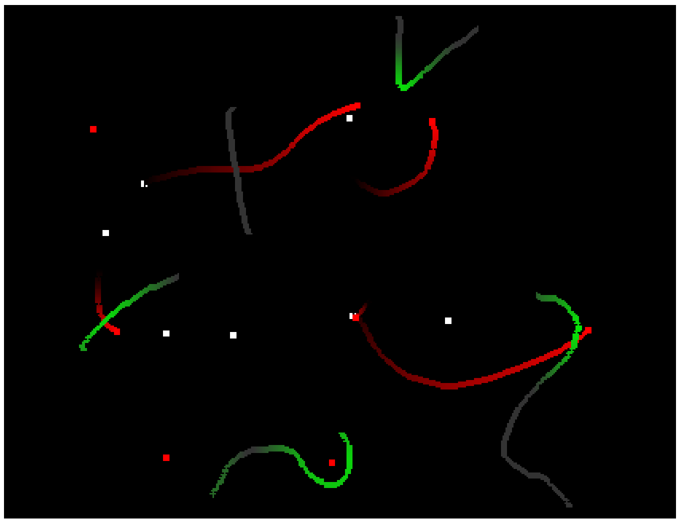


Figure 3.5 A screenshot from the first interface experiment

given strokes. A one-to-one correspondence is defined between stroke positions and wavetable positions, and the motion of the most recently added particle is used to drive sample playback.

The early design process leading to this final prototype saw the implementation of some preliminary interface designs, aspects of which survive in the current version. This early work is detailed in the following section.

3.4 Early Experiments

Some experimental systems were implemented early in the design process while working toward the final prototype. These led to important lessons that influenced the system design.

In the first experimental design, shown in Figure 3.5, the user was able to make two kinds of marks on the canvas: synthesis strokes, which generated sound, and control points, which controlled this sound generation.

The control points followed the user's cursor as she drew. When a control point was near a synthesis stroke, it "lit up" the portion of the stroke near the point, activating it. These active stroke segments were graphically indicated by a change in colour. When the control point moved away from the stroke, the segment ceased to be active and returned to its normal colour. For aesthetic reasons, moving control points were followed by a red, fading tail.

When part of a synthesis stroke was activated, it produced sound using frequency modulation. The local shape of this activated portion of the stroke was mapped to synthesis parameters to generate sound. The synthesis strokes were "played" by the moving control points in this way, and the stroke shapes defined the timbres that were produced. Each synthesis stroke was allocated one voice.

The user selected between control points and synthesis strokes using predefined keystrokes. When in control point mode, each user mark allocated a new control point. Each mark also placed a trigger on the canvas, associated with that control point. The triggers were coloured white, and were stationary. Trigger points were deposited at the first location of the user's mark. When a control point came close to a trigger, the motion of that trigger's associated control point was played back, and the control point retraced its motion.

Thus, the movements of control points could trigger movement in other control points. This could be used to create animated patterns of control point motion on the canvas. As these points moved past synthesis strokes, they generated sound. This scheme allowed the creation of cascading and looping patterns, and drove corresponding sound synthesis.

A second experiment was implemented in an attempt to integrate sample playback into the above design. It featured control points and synthesis strokes, as in the first experiment, but the synthesis strokes generated sound by wavetable playback. When a moving control point came near to a synthesis stroke, it triggered that stroke's associated sample. There was no meaning attached to the shape of the synthesis stroke, or which part of the stroke was activated.

This scheme tended to play back many samples simultaneously, and easily became very noisy. Volume envelopes were implemented to control the sample volume. The proximity of a control point controlled this volume envelope, as well as initially triggering the sample playback. When a moving control point was near a stroke, that stroke's playback was audible; when the control point moved away from the stroke, the playback volume was attenuated to zero. The volume of sample playback was controlled in this way. This design worked well for controlling the audio output, and eliminated the design's initial noise problem.

A number of lessons were learned over the course of these experiments. First, using proximity for activation did not work well. Sound was produced when an activating point came close to a target stroke. When the strokes are distributed densely throughout the canvas, this scheme becomes problematic as many strokes are activated unintentionally.

Second, the visual effects in the user interface improved the engagement of the user. As the control stroke tips played back, they trailed fading "tails" behind them. This visual detail is not necessary to the functioning of the system—the effect is simply for making the graphics more interesting—but it enhanced the user's experience. The tails made it easier to see the moving points, and were simply interesting to look at. The lesson here is that although graphical effects might not be useful to the system's mechanics, they increase the interface's effectiveness by appealing to users.

Third, editing operations are very important. Neither of these experiments were able to remove or silence strokes, and the audio output quickly became noisy and cluttered. Since one voice is associated to each stroke, and many strokes are created over the course of a session, many voices sound simultaneously. Further, a method of attenuating or removing sounds is a necessary feature. This is a manifestation of a more general observation: there needs to be a way of taking energy out of the system, either by deleting voices or removing active particles.

Fourth, target acquisition does not work very well for our purposes. White points acted as triggers for the control points' playback. When the user is trying to perform rhythmic motions with the pen, the ballistic gesture and timing are most important. Trying to hit specific trigger points on the screen compromises those motions. This fact led directly to the use of intersections to interrelate strokes in the final prototype.

3.5 Implementation Details

The interface design process is closely linked to the process of implementation. The latter evolved along with the interface design, and the realities of programming coloured the behaviour of the interface objects. This section describes the various subsystems that make up the program's inner workings, and discusses how they work together to create the interface behaviours. The interface's core framework is detailed, as well as the software that implements the specific behaviours of the strokes and particles. Challenges encountered as part of the implementation process are also discussed.

The application is implemented in C++ and is strongly object-oriented. This made the design and implementation process more involved, but led to several benefits: the code is easier to maintain and extend, it is more efficient than higher-level interpreted code, and it can be distributed as a single binary file without external dependencies. The system is designed for extensibility and modification. It is divided into a number of loosely coupled subsystems, easing testing and recombination. The application uses OpenGL (2005) for graphics via GLUT (OpenGL.org, 2006), and uses STK (Cook and Scavone, 1999) and RtAudio (Scavone, 2002; Scavone and Cook, 2005) for audio processing. Design patterns (Gamma et al., 1995) are used extensively throughout the program code, as well as the C++ standard template library (Stroustrup, 1997).

3.5.1 Framework Overview

The framework of the application supports the specialized stroke and particle behaviours. It provides services for audio, graphics, user input, and simulation. Specific behaviours and interface designs all build on this lowest level of functionality.

The application is divided into three main subsystems: the simulation core, the graphics subsystem, and the audio subsystem. The simulation core handles the strokes and their interactions in space. It is independent of the graphics and audio code and can be run on its own. The graphics subsystem handles input from the keyboard and pointing device, manages the application window, and handles graphics output. The audio subsystem handles synthesis and the audio hardware. The dependencies between these subsystems are shown in Figure 3.6.

3.5.2 Framework: Simulation Core

The simulation core dictates how the on-screen objects behave and interact. It consists of a single *workspace* object that contains multiple *stroke* objects, each of which corresponds to a stroke on the screen. All communication with the simulation happens through the workspace's C++ class interface. Three basic methods are provided: beginning a stroke, appending a point to a stroke, and ending a stroke. This corresponds to the act of drawing a stroke with the pen: lowering the pen, moving it through a trajectory, and lifting it. The workspace can only add points to the most recent stroke, so only one stroke

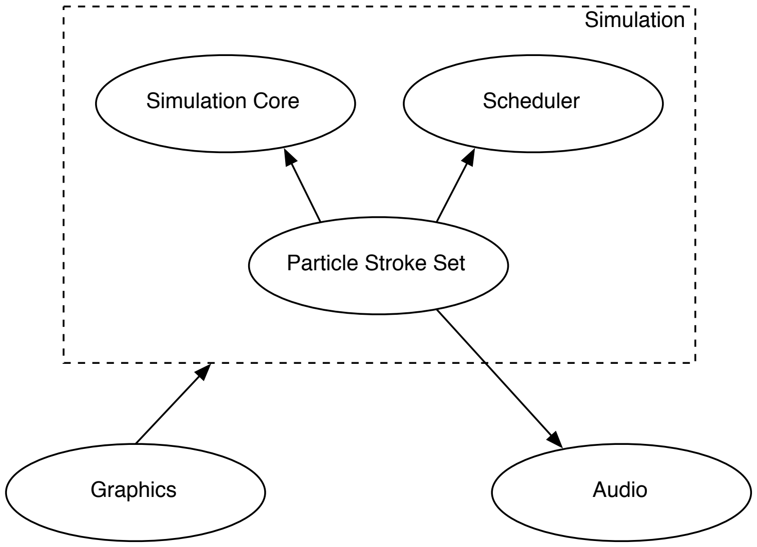


Figure 3.6 An illustration of the dependencies between the software subsystems. The arrows are read as "depends on."

may be drawn at a time. This basic foundation of workspaces and strokes is meant to be extended through subclassing to create particular behaviours.

A scheduler class is used to manage sporadic simulation events and to dispatch each at the appropriate time. It holds the current logical time value and a sorted list of pending events. Events are scheduled to occur at a given time and are dispatched when the scheduler's logical time is rolled forward. An event corresponds to some function call with a given set of arguments; when an event is dispatched, the appropriate function is called. Note that the scheduler's current logical time need not match the current *real* time. This gives the option of running the simulation offline while maintaining a notion of time. The basic workspace and stroke classes have no knowledge of the scheduler, and can be run independent of it.

3.5.3 Framework: Graphics Subsystem

The graphics subsystem handles input from the user and output to the screen. Input is received through callbacks registered with GLUT. When the mouse is moved or a key is pressed, the appropriate callback is called to handle the event. When it is time to draw the screen, the graphics subsystem visits each of the workspace strokes with a draw visitor. The visitor draws each stroke using OpenGL according to the stroke's type. Drawing is based on the current state of each stroke, and stroke properties that affect its visual appearance must be externally accessible through its class interface.

GLUT manages the main program loop, and our system registers a method that is called on each iteration to update the application. When this method is called, the simulation state is brought up to date and the screen is drawn. To update the simulation state, the scheduler's logical time is rolled forward to the actual current time, which dispatches pending events as applicable. When the redraw is complete, GLUT's main loop goes back to process more input. This sequence is inspired by the internal architecture of Pure Data (Puckette, 1996), which works in a similar way to interleave message and DSP processing.

3.5.4 Framework: Audio Subsystem

The audio subsystem is responsible for audio generation and output. It initializes the audio hardware and registers the audio output callback. Only one top-level audio manager object is instantiated over the course of the program's execution. It manages a list of synthesizers and acts as the application's main audio bus. To compute an output buffer of samples, it simply mixes the contribution of each synthesizer with unity gain.

The audio functionality is implemented by subclassing non-audio classes. All of the objects in the system that have associated sound have an audio subclass. The simulation core accesses both non-audio and audio versions of a given object polymorphically, through a pointer to the non-audio base class. Thus, audio subclasses are not permitted to change the external interface of the class in order to remain interchangeable with the non-audio versions.

The audio subclasses contain a synthesizer for generating samples. This synthesizer is registered with the main audio bus when the subclass is instantiated. Class methods for changing the object's state are overridden in audio subclasses to add hooks to control the sound generation, as shown in Figure 3.7. When one of these methods is called to update the object's state, the overridden version effects changes to the synthesizer as appropriate. In this way, the generated sound is kept in sync with the state of the corresponding on-screen object.

Note that this implies that synthesis parameters are updated when simulation events are dispatched. Since the scheduler is only run at each frame, synthesis parameter up-

```
class DemoStroke : public Stroke
{
  public:
    DemoStroke();
    virtual ~DemoStroke();
    virtual void updateState( void );

  private:
    float mystate_;
};

class AudioDemoStroke : public DemoStroke
{
  public:
    AudioDemoStroke();
    virtual ~AudioDemoStroke();
    virtual void updateState( void );

  private:
   FMSynth *fmsynth_;
};
```

```
void DemoStroke::updateState( void )
{
   // update the state
   mystate_ = someFunction( somearguments );
}

void AudioDemoStroke::updateState( void )
{
   // override the base class updateState() method to
   // add a hook for updating the synthesizer when the
   // state changes

   // call the base class update method to update the
   // stroke's state
   DemoStroke::updateState();

   // change synth parameters based on the newly
   // updated stroke state
   float fm_modindex = someFunctionOfTheBaseClassState();
   fmsynth_->modindex( fm_modindex );
}
```

Figure 3.7 Pseudocode that illustrates how audio subclasses override state-changing methods, adding code to update synthesis parameters when the stroke's state changes.

dates occur at the animation frame rate.

3.5.5 The Particle Stroke Set

Each design experiment is referred to as a *stroke set*, encapsulating a specific set of user interface features and behaviours. The final design of this thesis work is called the "particle stroke set" since it simulates the motions of particles along the on-screen strokes. Stroke sets are implemented as application subsystems that derive from the core workspace and stroke classes. These exist in the context of the framework, which provides all of the basic system services.

The particle stroke set manages two parallel data structures: the strokes themselves and a collection of one-dimensional spaces ("onespaces") in which the particles are embedded. There is a one-to-one pairing between strokes and onespaces. The onespaces are connected to each other at points that correspond to stroke intersections in the workspace.

The majority of the activity happens with the onespaces and their embedded particles. Each onespace represents a number line, extending infinitely in both the positive and negative directions. They contain a list of particles located at specific positions along their length. Particles may be added to a onespace at any point or removed. Particles

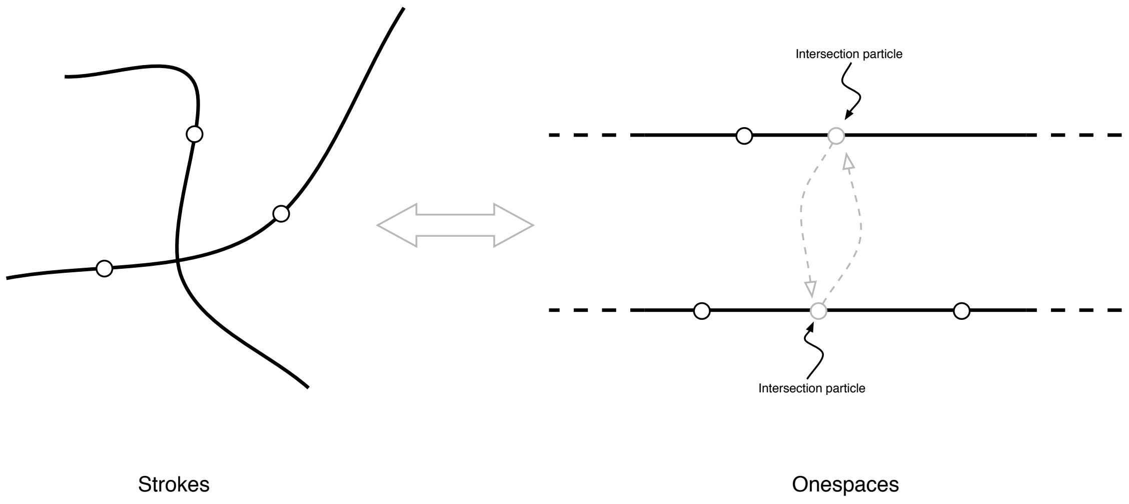


Figure 3.8 The correspondence between strokes and onespaces. Intersections are represented as a pair of particles, one in each onespace. The intersection particles are not drawn in the graphic interface.

may also be moved from their current position to any other position on the onespace. When a particle is moved past another one, the two particles interact.

The particles move along a stroke's onespace according to how the stroke was drawn. Each particle has access to the animation data defined by the drawing gesture. The particle's propagation through the onespace is evaluated as a series of discrete incremental movements. At each step, the particle determines how far its next displacement should be and then schedules a move event at the appropriate time. These movement events happen at floating-point logical times and can occur between graphics frames. This allows particles to move properly regardless of the animation frame rate.

Intersections occur between pairs of onespaces, and are represented as a pair of special particles, each embedded in a onespace. The intersection particle contains a pointer to the corresponding intersection particle in the other onespace, as illustrated in Figure 3.8. These particles are not drawn, but rather exist to be acted upon by other moving particles. When a moving particle passes an intersection particle, the intersection tells its other half to spawn a new particle in the other stroke. This is how the particle behaviour at intersections is implemented.

The intersections are computed when a new point is added to a stroke. The intersection algorithm is quite naive: it takes the newly added stroke segment and intersects it with all of the other segments in all of the other strokes in the workspace. This algorithm is linear in the number of segments, but it is made more efficient by considering the bounding box of each stroke and segment to eliminate unnecessary computation.

Each stroke has a sample embedded along its length that is played back by the moving particles. An audio subclass of the onespace class is used for controlling this sample playback. Each onespace (and thus each stroke) corresponds to one synthesis voice. When the onespace is created, it instantiates a wavetable playback synthesizer and registers it with the main audio bus. The active play head particle is determined as described above, and the wavetable play head position and speed is matched to the moving particle.

3.5.6 Challenges

A number of challenges were overcome during the design and implementation phases of the thesis work. These highlight some of the project's special characteristics.

The most fundamental challenge was to devise the interface design in the first place. Early on in the work, there was no clear design direction to follow. The basic elements were present—a spatial simulation for performance, the use of a pen and freehand gestures—but the specific details about how the system should work were not obvious. This unconstrained design space was both an exciting and challenging aspect of the project.

The implementation itself was fairly involved. The design of the class hierarchy for extensibility and loose coupling took a substantial amount of planning and effort. C++ is a relatively low-level programming language, requiring that more features needed to be implemented from scratch than might have been required in other languages. The implementation might have been faster in a high-level dynamic language like Python (Lutz, 2001) or Ruby (Thomas et al., 2004).

The implementation was involved, but the up-front programming investment promises a number of benefits. As the C++ code is lower-level, it is more computationally efficient. C++ is portable to all major platforms. All dependencies on external libraries (GLUT, STK) can be supplied at compile time, making it easy to distribute the application as a single binary with no external dependencies. The investment in the design of the class hierarchy and application framework will make the program easier to extend and maintain. The implementation is highly object-oriented, taking advantage of C++'s integration of high- and low-level programming facilities. Once the framework was in place, the implementation of the individual stroke behaviours was not any more complex than it would have been in a different language.

A specific implementation problem came from the mapping of particle motion to wavetable playback. The first iteration of the mapping algorithm produced undesirably noisy audio. The speed of the particle is computed based on the stroke points and the delta times between them. This speed is computed with respect to the stroke arclength, and this drives the wavetable playback rate. The computation was initially done from the raw stroke information, that is, the sequence of all mouse movements received for the stroke. This data is of quite a high resolution, giving about 100 data points per second. The mapping was thus modulating the wavetable playback rate at around 100Hz, which was responsible for the noise.

A downsampling approach was used to address this problem. A particle position versus time curve is maintained for each stroke that tracks the raw pointer movement information. This curve works very well for driving the graphics updates. From this curve, a second, lower-frequency curve is derived through subsampling, and this second curve is used to drive the playback rate. The subsampling rate is 5Hz. This technique eliminates the unwanted noise while maintaining the perceptual fusion between the sound and image.

Chapter 4

Results and Discussion

This chapter presents a discussion of the thesis software design and its consequences for performance. The system's characteristics are explored, first generally and then with respect to particular features of its design. User comments from a set of informal tests are presented. Some comparisons are made between the thesis software and other related research projects. Finally, the system is evaluated with respect to the project's initial goals.

4.1 General Interface Properties

The system illustrates the potential of software-based control of computer music. This style of performance—the live manipulation of generative processes—could be difficult to achieve in a hardware-only interface solution. The on-screen objects are active and need not respect any physical constraints. Arbitrary relationships can relate gesture, sound and visuals, allowing considerable design flexibility. Software solutions offer interesting prospects for music that warrant exploration (Magnusson, 2005).

The interface design deliberately avoids a control panel layout. As discussed in Chapter 1, control panel designs permit a user to manipulate at most one or two scalar parameters at a time, compromising musical control. The thesis software requires that the user set multiple parameters (sample speed, playback timing, control sequencing) at once with a single pen gesture. This could allow a performer increased control flow versus control panel interface designs.

The system allows the live assembly of generative control mechanisms. This strategy aims to allow a performer to create complex, layered pieces on the fly in performance. We see here an attempt at balance between live performance and computerized control: the generative structures feature the strong influence of live user gestures, but are ultimately run autonomously by the computer. We aim to allow the performer to multiplex her effort and attention between different parts of the music, leaving the computer to manage those parts that are not being manipulated. In this way, increased musical complexity is possible for a solo performer. The approach used here is similar to the "prepare and pilot" scheme that we discussed at the outset in that we take advantage of the computer's capacity for automation. The approach differs, however, in that the definition of the automated material happens live. This aims to bring a sense of effort and skill to laptop performance.

This strategy resembles the live coding approach discussed in Chapter 2. The interface trades live coding's low-level precision for efficient, high-level control, while trying to retain the same spirit.

The interface's freehand design allows a user to quickly specify sound-producing structures. A few short strokes with the pen can make a repeating audio pattern, and structures can be silenced by rapid removal. Layers of repeating patterns are also easy to make by creating separate figures in different parts of the canvas.

An important feature of the software design is that it lacks functionality for saving screen layouts or particular configurations. This is in keeping with the system's philosophical underpinnings, where live effort is prioritized over pre-performance programming. This lack of saved data will affect a musician's approach to the system. We return here to the acoustic musician's notion of preparation: performance through practice, not programming.

The system is very interactive and encourages experimentation. It is interesting to try different stroke configurations and to enjoy the resulting audio. Even though the interface's primitives are quite simple, there is a depth of possibilities to be explored. The sonic potential is extensive, even with a fixed set of input samples.

There is a certain learning curve in getting used to the interface. It is possible to make sound quickly, but controlling the strokes properly to obtain particular effects takes practice. Some experimentation is also required to appreciate the different control possibilities and structures that the system makes possible. Note that there is no attempt here to necessarily accommodate novices. Making the interface too easy to use could hinder its expressive potential, and the system design aims toward a balanced approach that leaves open the possibility of expert use.

The interface's graphical display allows a user to correlate the audio output with the movement of the particles. The active, on-screen structures behave like physical "machines." This "physicalization" of the computer's automated processes makes them easier to visualize and understand. However, when there are many strokes and many moving particles, it can become difficult to track what is going on.

It should be stressed that the system's current design is the result of the designer's personal preferences and ideas. The particle behaviours, mappings to sound, and other details were all chosen from an infinite set of possibilities. While the interface could be an interesting and compelling tool for performing computer music, it has not been shown empirically to be the best possible solution to the laptop performance problem. That said, its design is not completely arbitrary: it follows from the core ideas defined at the project's outset and from the early interface experiments, which helped to refine specific features.

4.2 Stroke Set Design

The current intersection behaviour tends to create many particles. In general, the intersection mechanism can only add new particles to the system. Douglas Hofstadter notes that "there really is not much deep interest in formal systems with lengthening rules only; it is the interplay of lengthening and shortening rules that gives formal systems a certain fascination" (1979, p. 48). Though the stroke set is not designed as a formal system, having mechanisms for both adding and removing particles could deepen its capabilities. In future design experiments, there could be a second intersection behaviour that would remove particles from the system, possibly leading to more interesting patterns and interplay between the strokes.

A feedback issue is present in the current system design for stroke topologies that feature two or more loops. The first loop continuously produces particles and feeds them to the second loop, which traps a continually increasing number of them. When the number of particles on the stroke grows large enough, the interface becomes overloaded and unresponsive. This can happen unintentionally, and makes the system slightly pre-

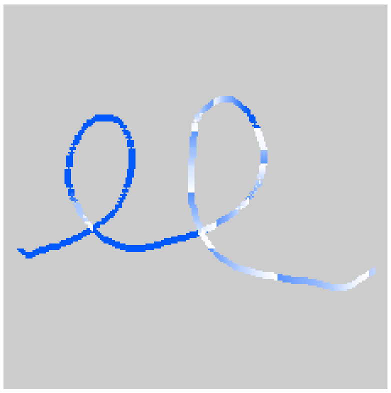


Figure 4.1 A figure featuring feedback

carious for performance. An example which demonstrates this behaviour is shown in Figure 4.1. Some measures will be needed to address this problem in future revisions of the software.

The current system design makes it relatively difficult to synchronize between different layers in the composition. While this may not be serious for certain abstract musics and polyrhythmic processes, musicians might want to have the option of dictating timing relationships between otherwise independent patterns.

The system's design couples elements of the music that might be manipulated independently in other applications. This gives the system a unique character that could prove interesting and unconventional for musicians. For example, sample triggering is determined by the shape and speed of the strokes, which also determines their timbral manipulation. A single pen stroke is used to control many different aspects of the sound sequencing and effects. This coupling forces the user to specify many different aspects of the music at once. It serves as a limitation for the musician, potentially helping to drive the creative process. However, it can be a problem if a particular musician wants to work with the related parameters separately. This coupling gives the system a unique feel that colours the musical output, which may or may not be desirable depending on the individual.

4.3 Freehand Input

The use of freehand input is a key feature of the system. It serves as a natural, high-bandwidth mechanism for live control. The physical variability in the pen gestures helps to give the control an inexact, organic, human quality that is often missing from computer music. When taken with the fact that one cannot save performance material, freehand input prevents pieces from being precisely repeatable from performance to performance. These factors infuse the system with a compelling humanness that will hopefully impact the music it is used to make.

Surprisingly interesting transformations of the sounds are possible through the simple mapping that the system currently provides. Sample playback is generally easy to control by varying the drawing speed, but the natural variation in the drawing gestures keep the audio surprising and engaging. This also makes it hard to drive a given sound at a constant speed, however.

The software is meant to be used with any type of pointing device, be it a graphic tablet or a standard mouse, in order to keep the system accessible to those without specialized hardware. However, just as it can be hard to draw a picture with a mouse, it is somewhat difficult to use a mouse to draw strokes in the system. A graphic tablet provides a much more natural experience and is more useful in this particular context.

4.4 Mapping

The mapping between wavetables and strokes has some consequences to the way the program is used. In the current scheme, the length of the wavetable is mapped along the arclength of the stroke. The duration of the samples relative to the average stroke length affects the typical pitch transformations applied. If all of the samples used are quite long compared to a typical stroke, their playback will be generally high-pitched. If they are short compared to a typical stroke, the output audio will be low-pitched. While a musician can certainly use this for particular musical effects, the relationship between the strokes and the sample lengths must be understood.

This mapping may be problematic for certain samples. For example, if a user wants to use a long sample, and wants its playback to be at roughly proper speed, the stroke required could have to be quite long, snaking around the canvas and taking up space.

This might end up being an obstacle for future strokes on other parts of the canvas.

As mentioned above, the system's current design can make it challenging to drive a sample at a constant speed. All sample playback is based on the drawing speed of the user's strokes, which is naturally variable. While this a deliberate design feature, some musicians may want to play sounds at their canonical speeds.

A consequence of this mapping scheme is that certain samples could be made to play very quickly, potentially leading to aliasing in the audio output. If a given sample is quite long and is mapped to a very quick stroke, its playback will be driven at a very high speed. Users should appreciate this fact and avoid any potential aliasing if that is deemed necessary.

A further repercussion of the interface mapping is that performers must be aware of each sample's approximate energy envelope. Playback begins from an arbitrary position after the beginning of the sample, and often continues until its end. If the majority of the sample's energy is at its beginning (as in a single drum hit, for example), the performer will have to trigger it at an appropriate location near the start of the stroke.

Finally, there is an asymmetry between the mapping and the simulation; only one voice is allocated per stroke, but we allow many potential playheads to travel along it. This issue is resolved by specifying that the most recently added particle is the active playhead. This design produces interesting audio cutting effects as new particles are added to a stroke and the wavetable playhead jumps around quickly. It might be clearer for users, however, if the mapping were more consistent with the simulation. We could preserve the current voice allocation and allow only one particle per stroke. In this case, newly added particles would cause the stroke's current particle to disappear. This approach would eliminate the simulation's feedback problem. A different strategy would be to allocate a synthesis voice to each particle and continue to allow multiple particles per stroke. This would potentially introduce interesting flanging effects in the audio output.

4.5 Other Issues

This section discusses some miscellaneous issues present in the system design. These are principally implementation details that will be relatively easy to address through further development of the software.

The editing capabilities currently present in the system are quite limited. This seems to be an obstacle for users, and expanded editing functionality will be a necessary addition in the future. Richer manipulation of the on-screen strokes—moving, rotating, or grouping them, for example—could be implemented as well as sub-stroke manipulations, such as cutting strokes, extending them, or adjusting their timing. These changes would allow for fine-grained tuning of running musical patterns.

As every newly drawn stroke has a particle following the drawing tip, it is impossible to add a stroke to an existing configuration without introducing a new particle. This is not always desirable, and it should be possible to draw strokes without adding a particle as well. It should also be possible to add a particle to the system manually, without drawing a new stroke.

The on-screen graphics are not a highly developed part of the system design. Much more information could be conveyed to the user that would be helpful during performance. For example, there is no on-screen indication as to which part of the sample corresponds to which part of a given stroke. A performer may wish to single out a particular part of a wavetable, but it is nearly impossible to do so consistently with the current setup. When many particles are present on a sounding stroke, there is no indication as to which one is currently active. More generally, there is an overall lack of feedback in the system. The current drawing mode should be indicated, for example.

In the current interface scheme, there is also no way to control the playback volume of each sample. This could make the output audio quite loud if many voices were playing at once. Some mechanism should be introduced to allow volume control for individual voices.

The current audio mapping produces sound by jumping around wavetables, playing small parts of them at varying speeds. Depending on the wavetables, this can produce audible clicks in the output. While some musicians will certainly make use of this effect, it should be possible to generate audio without these artifacts. Further, the wavetable playback head can stall completely at a particular position in the sample, producing DC audio output.

Another basic function that has not yet been implemented is the ability to specify the samples at run time. This is certainly possible, and will need to be added to make the system more convenient for the end user.

4.6 User Comments

A number of informal user tests were performed in which individuals experimented with the interface. This section presents their impressions and suggestions for improving the system. These comments will be taken into account during further development of the application.

The first thing the users noticed was that the interface was engaging. The new users were eager to try it out for themselves, and all said it was enjoyable to experiment with. The system remained interesting even after extended sessions, and there is a certain learning curve involved in trying to understand its workings. Sound can be made right away, but practice is required to properly take advantage of the interface.

The users commented that there was a general lack of visual feedback in the interface. Some users expressed that feedback might be beneficial for targeting specific parts of the wavetables and for understanding what is happening in the audio output. A graphical representation of sample data could be plotted along the length of a stroke, or in a separate pane at the top of the canvas window. More visual feedback would be good in general—displaying the current drawing mode or other information would help keep the user aware of the internal state of the system.

Some users isolated the lack of editing functionality as a drawback. The current system only supports stroke deletion and does not allow a stroke to be modified once it has been drawn. Editing functionality should be provided at stroke and sub-stroke levels. For example, users wanted to be able to adjust the timing of a running figure, slowing it down or speeding it up. It should be possible to extend or shorten strokes without having to entirely delete and redraw them. Users wanted to have better control over the speed of the moving particles; at the moment it is very difficult to play a sample back at its canonical speed. The current interface has no way of stopping moving particles, and so the samples tend to be played to their end. Users wanted to be able to optionally prevent this behaviour with other kinds of strokes or extra options.

Some users suggested potential improvements to the design of the system itself. The most obvious one was to prevent the triggering of feedback loops. This consequence of the current design makes the system somewhat unstable. Also, the users suggested that it be possible to omit the particle that follows the drawing cursor. This behaviour forces performers to add particles to loops when they may not want to. It would be useful to be able to disable this behaviour temporarily. Other users wanted to have more control over the particles themselves. Currently, there is no way to manipulate or remove individual particles.

Some other suggestions concerned changing the keyboard mappings to particular mouse button combinations, and changing the default samples that are loaded into the system. Many possibilities exist for integrating gestural input into the interface as well, and users offered a few suggestions to this effect. Users wanted to see OSC or MIDI support for driving external programs such as Supercollider or Pure Data. They suggested many different variations and extensions of the current design model that integrated notions of energy and new mapping parameters.

4.7 Comparison with Other Systems

The thesis work draws on existing computer music research. In particular, it shares characteristics with some other systems that use freehand gestures for musical control. This section compares this research to these related works.

The software system bears some similarity to Golan Levin's Master's work, especially his *Loom* interface, described in Chapter 2. *Loom* features the use of animated freehand stroke trajectories that are mapped to sound for performance, similar to the design presented here. While Levin's work provided inspiration for this thesis research, the overall goals of the two projects differ.

The idea behind this work was to create a "spatial language" for creating and modifying generative control patterns in performance. The aim was to develop a graphical system with atoms, rules and grammar that could be used for creating music. The main focus is on mechanical control structures, and on quickly specifying multiple simultaneous sequencing voices. The relationships between the strokes are meaningful, and are in fact necessary for making sound. The emphasis here is on the interface design issues and the effectiveness of the software for laptop performance.

Levin's focus was on the simultaneous creation of abstract visuals and sound. His research explored the aesthetic of the interaction from the performer's perspective, as well as the character of the resulting images and audio. The interfaces can be understood as "live paintings," and his research explored their qualities as artistic media in a performance context.

This Master's research also resembles the UPIC system, also explored in Chapter 2, in certain respects. Again, however, the goals of the two systems differ. UPIC is designed as a general-purpose compositional tool for computer music. It uses two-dimensional (x, y) plots for defining all aspects of the music, from sound synthesis to sequencing and arrangement. UPIC is strictly an offline system, and is used iteratively as a composition is built.

The work presented here is designed as a real-time performance interface and is not intended for general compositional purposes. There is no notion of axes as in UPIC. The strokes are meant to be understood as active objects unto themselves, and not as graph plots.

4.8 Goals

A set of goals for the project was established in Chapter 3. These were to create a software performance system that brings a sense of active creation to laptop performance; to allow the creation and manipulation of multiple simultaneous, generative control patterns; to explore the notion of using a spatio-temporal simulation for musical control; and to develop a distributable software package. In addition to these goals, a set of heuristics was also established to guide the design process. This section discusses the thesis software in light of these goals and design heuristics.

The software forces the performer to create all of the musical material on-stage. Predefined control sequences or processes cannot be employed, and the musician is necessarily active in performance. This is a reaction to overly automated laptop performance, and attempts to address the first goal. Another problem that was identified in laptop performance was the lack of visible causality. If made visible to the audience, the system's graphics can help correlate the user's gesture, the computerized action and the resulting audio. While it has not been established empirically that the interface brings a sense of active creation to laptop performance, the system serves as a promising step in that direction.

The interface allows the creation of multiple concurrent, generative voices in a piece. This aspect of the software attempts to strike a balance between the user and computer contributions to the music, and to allow a solo performer to execute the complex, layered pieces typically seen in laptop performance.

The interface is designed as a spatio-temporal simulation. The hope was that this design would allow better control of the large amount of parallel action often exhibited by computer music. While this aspect of the interface is engaging for the user and the audience, it is still unclear whether it is demonstrably better for the performance task. Formal user studies will be required to assess this with any certainty.

The final goal was to create a distributable software package that can be downloaded and used by computer musicians at large. The software has few external dependencies and relies on portable C++ code, so it can be easily distributed in the future.

The interface design also reflects the design heuristics established earlier. The system design is conceptually minimal, but still allows an interesting breadth of behaviours. The strokes have relationships, and must be combined to define generative patterns. The user's gesture, the particle motion, and the audio playback are all perceptually related to one another, helping the user internalize the simulation's workings. The system uses a pen or a mouse for input, capitalizing on the natural properties of freehand input. Finally, the system imposes some limits on the performer, forcing her to find creative solutions to performance problems.

Chapter 5

Conclusions and Future Work

This chapter summarizes the explorations of this thesis work, presents some conclusions, and details avenues for further research.

5.1 Summary and Conclusions

This thesis began by exploring some common problems in laptop performance. Laptop performances typically do not foster a sense of active creation, nor provide a visible, causal link between the performer and the music. These performers often resort to heavy automation to cope with the complexity of multi-layered pieces. This cedes too much to the computer and compromises live control.

The thesis project was motivated by this situation in laptop performance. The software aimed to improve on other performance tools by focusing on high-level live control and by preventing the use of prepared material. The system was designed to allow the efficient creation and control of simultaneous, independent generative processes. This design targets styles of computer music that typically appear in laptop performance and feature layers of looping audio patterns.

Examples of existing musical performance software were presented to illustrate common designs. These applications tended to encourage the use of prepared sequences and processes, and featured control panel interface layouts. Other examples were presented that illustrated the potential of software systems for live musical control.

The final software prototype took the form of a freehand drawing program. Networks of user strokes serve as topologies through which on-screen particles move and replicate. The particles travel according to each stroke's drawing gesture, and replicate at stroke intersections. This design allows the user to quickly create looping and cascading patterns. A mapping from these motions to audio generates music corresponding to the visual activity. Performing with the system is accomplished by drawing and manipulating strokes live to create music.

The system was implemented as a stand-alone software application. It was designed as a set of distinct subsystems that work in concert to implement the program's features. The application's stroke behaviours are implemented via one-dimensional parameter spaces that are associated with each of the strokes. These spaces are logically connected to one another where the on-screen strokes intersect. Particles move through these spaces and transfer between them at the connections. A wavetable is associated with each one-dimensional space, and particle motion through the space drives corresponding wavetable playheads. Sample playback is thus governed by the movements of particles along the strokes.

The project largely met the goals established at its outset. The system serves as a novel performance interface that emphasizes the live creation of computer music and that allows the manipulation of independent generative patterns. These patterns can be quickly created and modified on the fly, making the system useful for performance and improvisation. The application has an enjoyable, unusual character that influences the user's creative experience, as well as the final musical output it is used to create. Though it remains to be seen if the software actually brings a sense of active creation to laptop performance, it is a solid step in the right direction.

The application is, however, a first prototype, and a number of issues in its design and implementation need to be resolved to support its regular use in real-world performance. Specific issues in the design of the stroke behaviours need to be improved; in particular, the problem of feedback loops in the stroke structures should be addressed. The general usability of the program needs to be improved to make it more suitable for end users.

In this thesis project, a promising software system for laptop performance has been designed and implemented. It provides an interesting, encouraging foundation on which to base further work.

5.2 Future Work

This section details a number of specific extensions of and improvements to the system that will be pursued in future development efforts. Some of these address problems described in Chapter 4, and others describe new features that would improve the system's usability.

First, the overall stroke interactions need to be developed. As discussed in the previous chapter, the current intersection behaviour tends to create many particles. Some mechanism for taking particles out of the system would help control this proliferation. Such a feature should lead to more interesting patterns and interplay between the strokes.

A specific design idea to this effect is to introduce a second type of stroke into the system. Intersections would be categorized depending on whether they were created by two strokes of the same type, or two strokes of different types. At heterogeneous intersections, incoming particles would be divided, with one travelling along each outgoing branch. (This is the intersection behaviour as it exists now in the current system design.) At homogeneous sections, however, incoming particles would simply be transferred to the other stroke without being split. This scheme would allow the user to take particles out of a system by diverting them. Some of the challenges inherent in this proposal are the need to devise graphical cues for differentiating between the two types of strokes, and to determine how the user should indicate that one or the other stroke type should be used.

Other types of stroke behaviours could be useful as well. For example, certain strokes could introduce conditional behaviour, or even randomness. These behaviours could make for richer musical patterns. The specific behaviour designs that would give rise to these effects still need to be researched.

A notion of particle energy could be introduced into the system design for controlling volume. When a particle divided at an intersection, its energy would be distributed between the two output particles. This would lead to interesting exponential decays in stroke loops, and might mitigate the feedback problem.

As mentioned in the previous chapter, more drawing options should be added to the system. Every newly drawn stroke presently has a particle following the drawing tip, and options should be available for suppressing this behaviour. Individual particles could also be manually added to existing configurations without introducing a new stroke on the canvas. A particle could just follow the user's pointer without the accompanying stroke, and it would be added to existing stroke structures as it crossed them. In the interface, these could be provided as per-stroke options. A modifier key could be used to draw a stroke without a new particle, and another could disable the stroke itself, just leaving the moving particle.

Also identified in the previous chapter is the problem of transients and DC components present in the audio output. The current naive mechanism for scrubbing through wavetables makes no provision for these special cases. Improving the synthesis functions to remove these artifacts would be an important refinement. Adding envelopes at the audio level would eliminate these transients. A DC-blocking filter could be added to the main audio output bus to correct for DC components. Perhaps these features could be enabled at the user's discretion, so that those musicians who would like to exploit those artifacts creatively would still be able to do so.

The connection to the built-in audio synthesis engine is arbitrary, and the state of the simulation could be used to drive an external audio synthesis engine. Changes in the simulation state could be encoded as messages sent over a socket; a proprietary message format could be used, or a standard like OpenSound Control. This particular feature would be of interest to musicians already using audio synthesis programs that accept external input from a network connection (e.g., Max/MSP, Pure Data, Supercollider). The system could even be used to produce MIDI messages, which could be used to drive other external processes. The details involved in these mappings have yet to be investigated; it remains to be decided exactly what information from the simulation would be useful to output.

The simulation canvas has no set scale units or orientation. This is done deliberately to avoid imposing a defined set of axes on the canvas space. It should be possible for a user to take advantage of this fact by rotating, panning, or zooming the canvas. This could make it easier to draw particular figures; it could be analogous to an artist using her non-dominant hand to orient a piece of paper she is drawing on. These viewpoint manipulations could be triggered by particular input gestures, making them efficient and natural for a user to execute.

References

Ableton AG. Homepage, 2006. <`http://www.ableton.com/`>. 18 February 2006.

Rob Bridgett. Music with limits. *Computer Music Magazine*, 91:72–77, October 2005.

Kim Cascone. The aesthetics of failure: "post-digital" tendencies in contemporary computer music. *Computer Music Journal*, 24(4):12–18, 2000.

Kim Cascone. Grain, sequence, system: Three levels of reception in the performance of laptop music. *Contemporary Music Review*, 22(4):101–104, 2003.

Nick Collins. Generative music and laptop performance. *Contemporary Music Review*, 22 (4):67–79, 2003.

Nick Collins, Alex McLean, Julian Rohrhuber, and Adrian Ward. Live coding in laptop performance. *Organised Sound*, 8(3):321–330, 2003.

Perry Cook and Gary Scavone. The synthesis toolkit (STK). In *Proceedings of the International Computer Music Conference*, pp. 164–166, Beijing, China, 1999.

A. K. Dewdney. *The Armchair Universe*. W. H. Freeman, New York, 1988.

Cole Gagne. *Soundpieces 2: interviews with American composers*. Scarecrow Press, Metuchen, NJ, 1993.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

Douglas Hofstadter. *Gödel, Escher, Bach*. Basic Books, New York, 1979. Reprinted 1989, Vintage Books, New York.

Sergi Jordà. FMOL: Toward user-friendly, sophisticated new musical instruments. *Computer Music Journal*, 26(3):23–39, 2002.

Golan Levin. Painterly interfaces for audiovisual performance. Master's thesis, Massachusetts Institute of Technology, 2000.

Henning Lohner. The UPIC system: A user's report. *Computer Music Journal*, 10(4): 42–49, 1986.

Mark Lutz. *Programming Python*. O'Reilly & Associates, Sebastopol, CA, 2001.

Thor Magnusson. ixi software: The interface as instrument. In *Proceedings of the Conference on New Interfaces for Musical Expression*, pp. 212–215, Vancouver, Canada, 2005.

Gérard Marino, Marie-Hélène Serra, and Jean-Michel Raczinski. The UPIC system: Origins and innovations. *Perspectives of New Music*, 31(1):258–269, 1993.

James McCartney. Continued evolution of the SuperCollider real time synthesis environment. In *Proceedings of the International Computer Music Conference*, pp. 133–136, Ann Arbor, MI, 1998.

Native Instruments. Reaktor product page, 2006. <`http://www.native-instruments.com/index.php?id=reaktor5_us`>. 18 February 2006.

OpenGL Architecture Review Board, Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *The OpenGL Programming Guide*. Addison-Wesley, Upper Saddle River, NJ, 2005.

OpenGL.org. GLUT – the OpenGL utility toolkit, 2006. <`http://www.opengl.org/resources/libraries/glut/`>. 4 May 2006.

Amit Pitaru. Sonic wire sculptor, 2003. <`http://www.pitaru.com/sws/`>. 8 May 2006.

Miller Puckette. Pure data. In *Proceedings of the International Computer Music Conference*, pp. 269–272, Hong Kong, 1996.

Miller Puckette. Max at seventeen. *Computer Music Journal*, 26(4):31–43, 2002.

Gary Scavone. RtAudio: A cross-platform C++ class for realtime audio input/output. In *Proceedings of the International Computer Music Conference*, pp. 196–199, Göteborg, Sweden, 2002.

Gary Scavone and Perry Cook. RtMidi, RtAudio, and a synthesis toolkit (STK) update. In *Proceedings of the International Computer Music Conference*, pp. 327–330, Barcelona, Spain, 2005.

Laurie Spiegel. Computer software by Laurie Spiegel, 2004. <`http://retiary.org/ls/programs.html`>. 14 November 2005.

Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley, Reading, MA, 1997.

Atau Tanaka. Music performance practice on sensor-based instruments. In Marcelo Wanderley and Marc Battier, editors, *Trends in Gestural Control of Music*, pp. 389–405. IRCAM – Centre Pompidou, 2000.

Dave Thomas, Andy Hunt, and Chad Fowler. *Programming Ruby: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, Raleigh, NC, 2004.

Ge Wang and Perry R. Cook. On-the-fly programming: Using code as an expressive musical instrument. In *Proceedings of the Conference on New Interfaces for Musical Expression*, pp. 138–143, Hamamatsu, Japan, 2004.

Matthew Wright, Adrian Freed, and Ali Momeni. OpenSound Control: State of the art 2003. In *Proceedings of the Conference on New Interfaces for Musical Expression*, pp. 153–159, Montreal, Canada, 2003.

Matthew Wright, David Wessel, and Adrian Freed. New musical control structures from standard gestural controllers. In *Proceedings of the International Computer Music Conference*, pp. 387–389, Thessaloniki, Greece, 1997.