Hierarchical Encapsulation and Connection in a Graphical User Interface: a Music Case Study

Gordon Paul Kurtenbach

Department of Computer Science
University of Toronto
October 1988

this thesis submitted in conformity with the requirements
of the degree of Master of Science

Abstract

Graphical representations consisting of nodes and arcs have proven useful in many interactive applications. However, as the number of nodes and arcs becomes large, viewing, editing and understanding the network becomes problematic and the value of the representation breaks down. We combat this problem by providing the user the ability to hierarchically structure the network by encapsulating related nodes into modules. Hierarchical structuring allows the user to collapse portions of the network into viewable and understandable sized chunks. This reduces the apparent complexity of the network in a manner similar to top-down structuring in programming; a top level view gives a concise representation of the network devoid of unnecessary details. As you descend the hierarchy, details about the internal structure of modules are revealed.

The intent of this thesis is to explore the use of hierarchical encapsulation by means of a case study. Our case study is a system for controlling the routing of audio and control signals among devices in an audio studio. Nodes in our network are audio devices and the arcs are the paths that interconnect them.

This thesis illustrates the value of user-control over structuring of the network, level and type of representation. The results have applicability in other applications and these are discussed along with general principles learned in the course of the case study.

Acknowledgements

I'd like to thank my supervisor, Bill Buxton, not only for his guidance and support, but for giving me the freedom to explore my own ideas and his assistance in transforming these ideas into concrete research. I would also like to thank Martin Snelgrove, my second reader, for his suggestions that led to a better thesis and John Kitamura whose system, the Katosizer, inspired of my work. Appreciation is extended to the members of the Dynamic Graphics Group at University of Toronto for creating a friendly, creative working environment. Most notably I'd to thank John Buchanan for his moral support and creative input, Ian Small for keeping the hardware and software running and Carson Schutze for his valuable comments on my thesis.

The research in this thesis involved the creation of software and acquisition of specialized hardware. I am most grateful to Dave Blythe and Steve Hutchings for creating critical hardware and software and to Apple Computers Inc, whose funding allowed us to purchase the music hardware.

Part of the valuable experience of graduate school is attending conferences. This would not have been possible without funding from the Dynamic Graphics Project, to whom I am grateful. Finally, I would like to thank the Natural Sciences and Engineering Research Council of Canada for financial support.

Contents

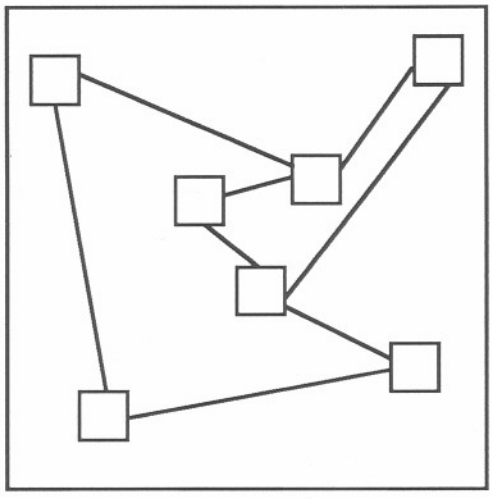
List of Figures

Chapter 1

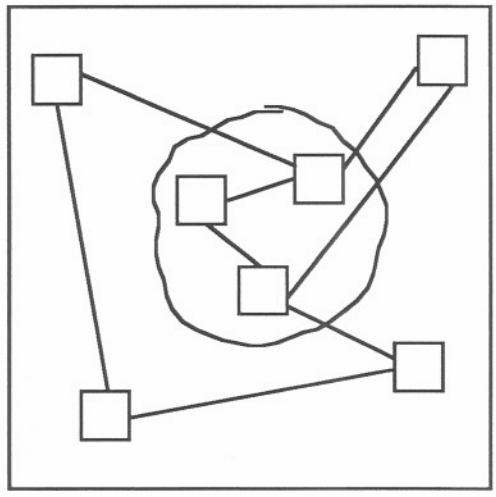
Introduction

The aim of this thesis is to investigate new forms of representation, and interaction that can improve human performance in certain classes of computer applications. The basis of the approach investigated involves the explicit hierarchical chunking of the problem, and the specification of the connection of those chunks. The general problem involves developing an interface to edit a set of objects and the relationship among them. An interface for editing a small set of objects and connections is relatively simple to construct, understand and use. However, as the number of objects and connections grow, a methodology must be applied to deal with the resulting complexity.

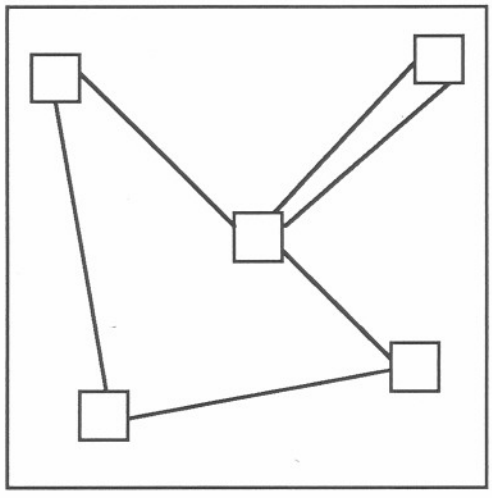
In this thesis we attempt to reduce the apparent complexity of such systems by providing a means whereby the user can encapsulate chunks of such networks and collapse them into a single node. The belief is that such hierarchical encapsulation will allow the user to bury complexities of the system. By enabling the user to navigate through the hierarchy, the means is provided whereby the complexity of the level of representation can be shaped to fit requirements of the task currently being performed. Figure 1.1 shows an example of how components of a network can be chunked into a node, thus distributing the complexity of the network over different levels of the hierarchy. This technique is investigated by means of a case study. In particular, we look at this technique in terms of the pragmatics of interaction, its impact on the semantics of the interaction and its effect upon the user's conceptual



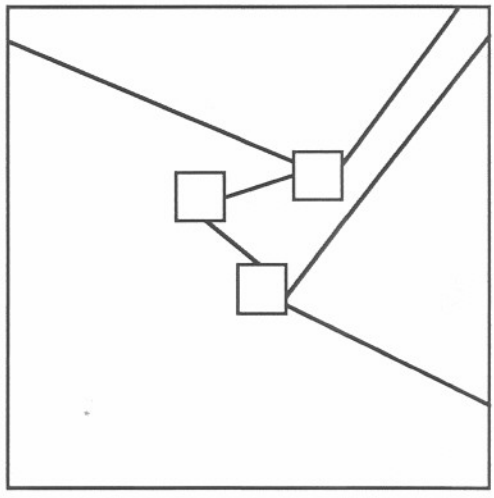
a) A generic network.



b) The user "chunks" a cluster of objects to be encapsulated into a module.



c) The system responds by creating a module.



d) The user has descended into the new module to reveal its internals.

Figure 1.1: Encapsulating components of a network.

model.

Hierarchical encapsulation is not a concept novel to this thesis. The value of this thesis lies in the fact that the methodology of hierarchical encapsulation is examined in terms of a graphical interface which involves novel techniques of interaction and graphical representation.

Hierarchical encapsulation is a means to handle complexity; the user encapsulates related pieces of information into manageable "chunks", hiding the details till they are needed. This is especially true in the area of computer science [34]. Techniques of information hiding such as modularization and top down structuring are proven to be helpful in understanding, designing, building and maintaining complex systems. It is on this premise that the importance of this thesis lies; surely such successful methodology, when applied to graphical interfaces, is worthy of investigation.

1.1 Meta Issues

1.1.1 In Defense of a Case Study

This thesis examines the issues of interest through the vehicle of a case study in computer music. Before proceeding with this study, it seems worthwhile to briefly discuss the merit of the case study approach in general, and the appropriateness of our specific application domain, music. There are, at least, two major approaches to investigating a particular concept in user interface design. One is to study the concept in the context of a specific application (the case study approach). The other is to work at a higher level of abstraction, in an application independent manner. Both approaches have merit and, in reality, both must coexist to a degree. In reality, differences are a matter of emphasis, often determined by the types of questions that one is investigating. In this thesis we lean towards "experimental programming", using the case study approach. The intention is to attempt to obtain, thereby, a sense of validity of ideas in the large context of a real system with real users. In

the following discussion we further motivate this decision.

The construction of a working system allows the general concepts of the interface to be tested by having real users performing real tasks. In addition, a real system is a vehicle in which the general properties of the interface can be demonstrated. Second, by examining the successful features of the interface and distilling them from the specific application of the case study, we can apply these tested features to other similar application areas. Third, there exist many similar applications to our case study. We will now expand on each of these claims.

First is the matter of demonstration and testing. An interactive user interface is just that: interactive. To attempt to demonstrate an interactive interface design by description or by static pictures cannot capture the essence of the interaction. One can describe a fine piece of art, but the display of the actual art piece is far more effective in demonstrating the properties. As for the matter of testing, user interface design is not a science and therefore it is difficult to predict the ramifications of design features "on paper". Designers have recognized the advantages of prototyping, and "quick and dirty" system construction as part of the design process. Not only does a real system give the designer a more concrete view of the system, but the construction of prototype systems allows the designer to test the interface on real users and redesign based on test results. In effect a real system allows us to prove, or disprove, the value of our design.

Furthermore, without a real application, reasoning about the interface cannot be done. For example how can we reason about an interface for hierarchical encapsulation and connection if we know nothing about the objects and connections? A real application provides application specific issues and details which reveal unpredicted problems and advantages in the broad issues. An example from civil engineering can be used to demonstrate this point: On paper the design of suspension bridges seemed perfectly acceptable. It wasn't until July 1 1940 when the Tacoma Narrows bridge in Puget Sound Washington began to flap in the breeze that a fundamental design problem was recognized.

Next we can examine the representativeness of our case study. In order to apply

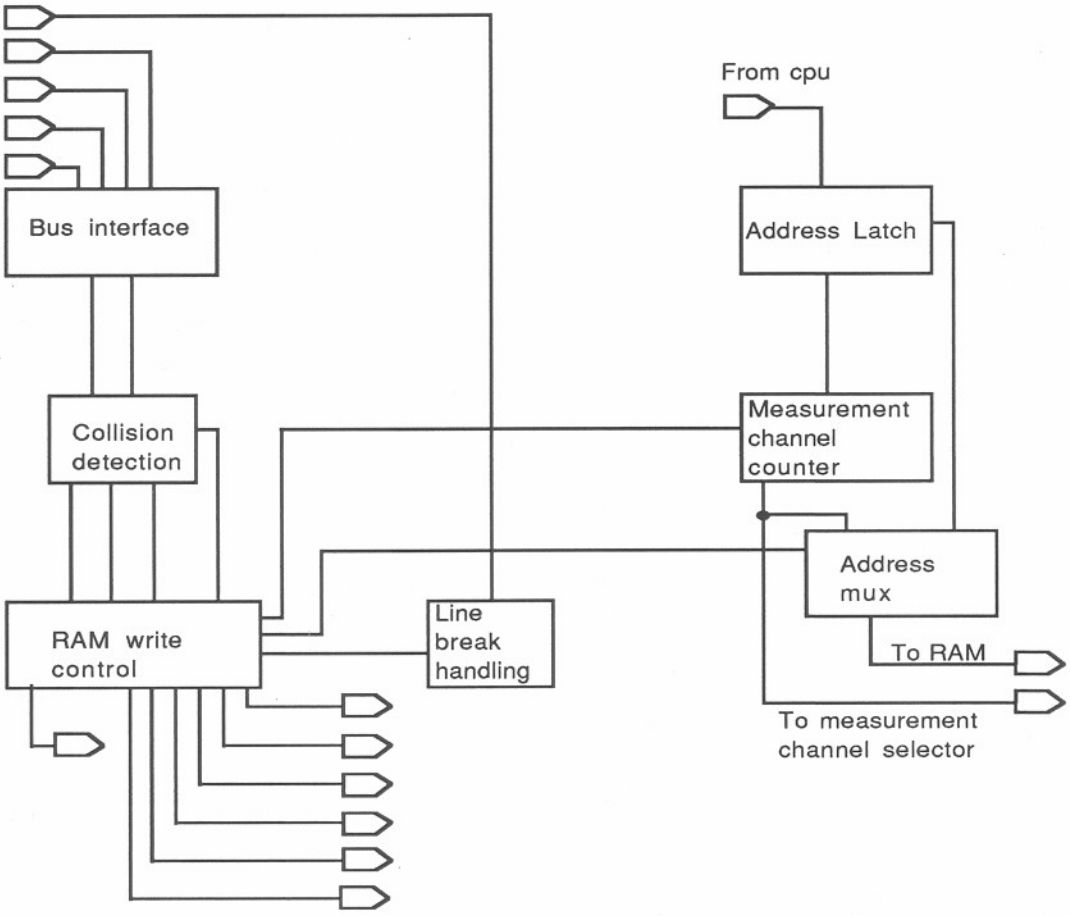
the results of a case study to another application area, a designer must be aware of the general characteristics of the case study and determine if the other application has the same general characteristics. If, in fact, the other application does share general characteristics with the case study, the designer can "borrow" successful design features from those developed in the case study.

With this in mind, we identify the general characteristics of our case study:

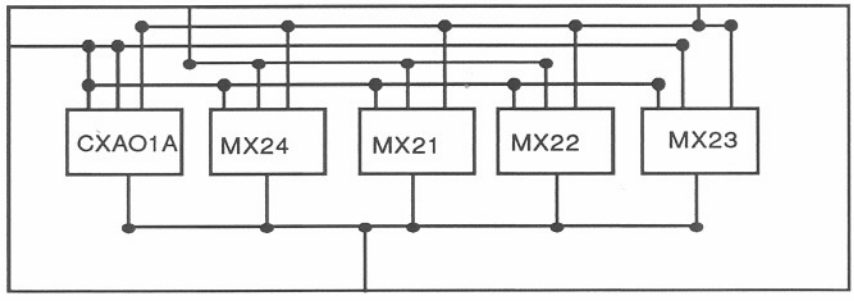
- Objects have data flow connections between them.
- The user must be able to specify extremely complex configurations of objects and connections
- More objects exist than can be viewed at once.
- Objects have functional relationships that lend themselves to logical grouping.
- Objects have parameters associated with them which the user must access.
- More parameters exist than can be viewed at once
- User access to parameters exhibit locality.
- There is a "design" and "operation" (or control) component, or phase, each requiring the display of different types of information.

Note that these general characteristics apply to a large class of applications. Essentially the case study deals with visually programming a system where objects are data sources and sinks, and connections represent data flow among objects. Any, if not most, of these general characteristics are demonstrated by many other applications such as VLSI design [31], Recursive Transition Networks [33], Data Flow Diagrams [12] and Hyper-Text [15].

In VLSI design, the objects being dealt with represent electronic components and the connections among them. Hierarchical structuring allows components to consist of other components and connections. Figure 1.2 is a simple example of Hierarchical structuring in VLSI design.

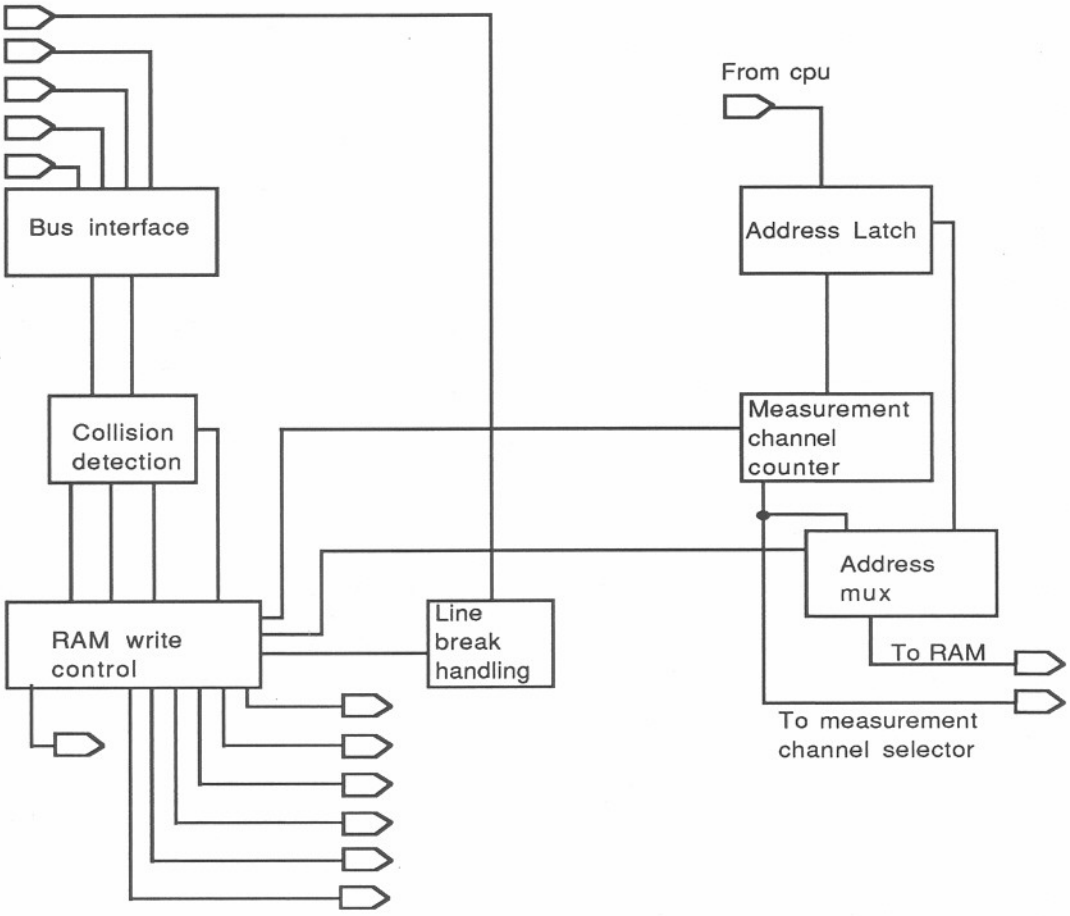


a) A portion of a block diagram of an analog to digital converter controller.

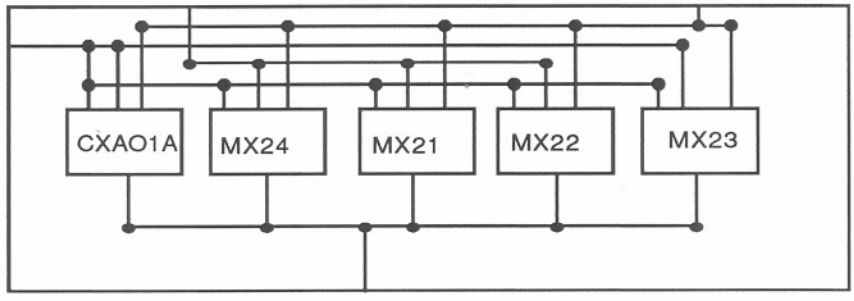


b) The internals of the Address mux block where blocks are basic components such as NAND gates, counters or multiplexers.

Figure 1.2: Hierarchical structuring in VLSI design.

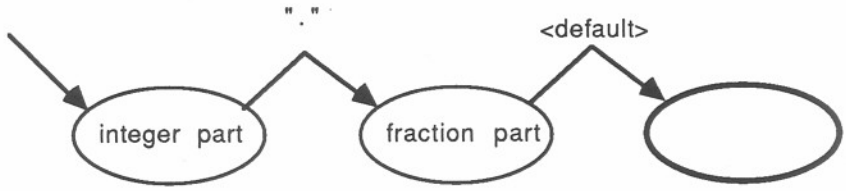


a) A portion of a block diagram of an analog to digital converter controller.

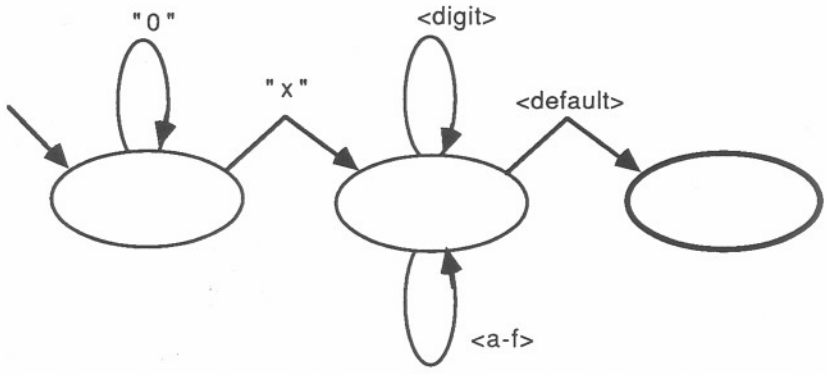


b) The internals of the Address mux block where blocks are basic components such as NAND gates, counters or multiplexers.

Figure 1.2: Hierarchical structuring in VLSI design.



a) The basic structure of the recognizer.



b) The internal structure of the node "integer part". Which recognizes the integer part of the number.

Figure 1.3: A simple recursive finite state recognizer for floating point hexidecimal numbers (for example: 0x34f.0g4).

Recursive transition networks (RTNs) consist of objects which represent a state of a finite state machine and connections which represent a conditional transition to another state. An object may have an internal recursive transition network that defines its behavior. RTNs are used in applications such as the design specification of interactive systems based on state transitions and lexical analyzers in language compilers. Figure 1.3 shows and example of the latter .

Hyper-Text systems allow hierarchical structuring and access to textual objects. Figure 1.4 shows a hypertext system called NoteCards [15]. NoteCards is intended to help people formulate, structure, compare and manage ideas by allowing the user to construct and access a "semantic network" of note cards. Note cards can be collected into special cards called "FileBoxes", thus imposing hierarchical structure.

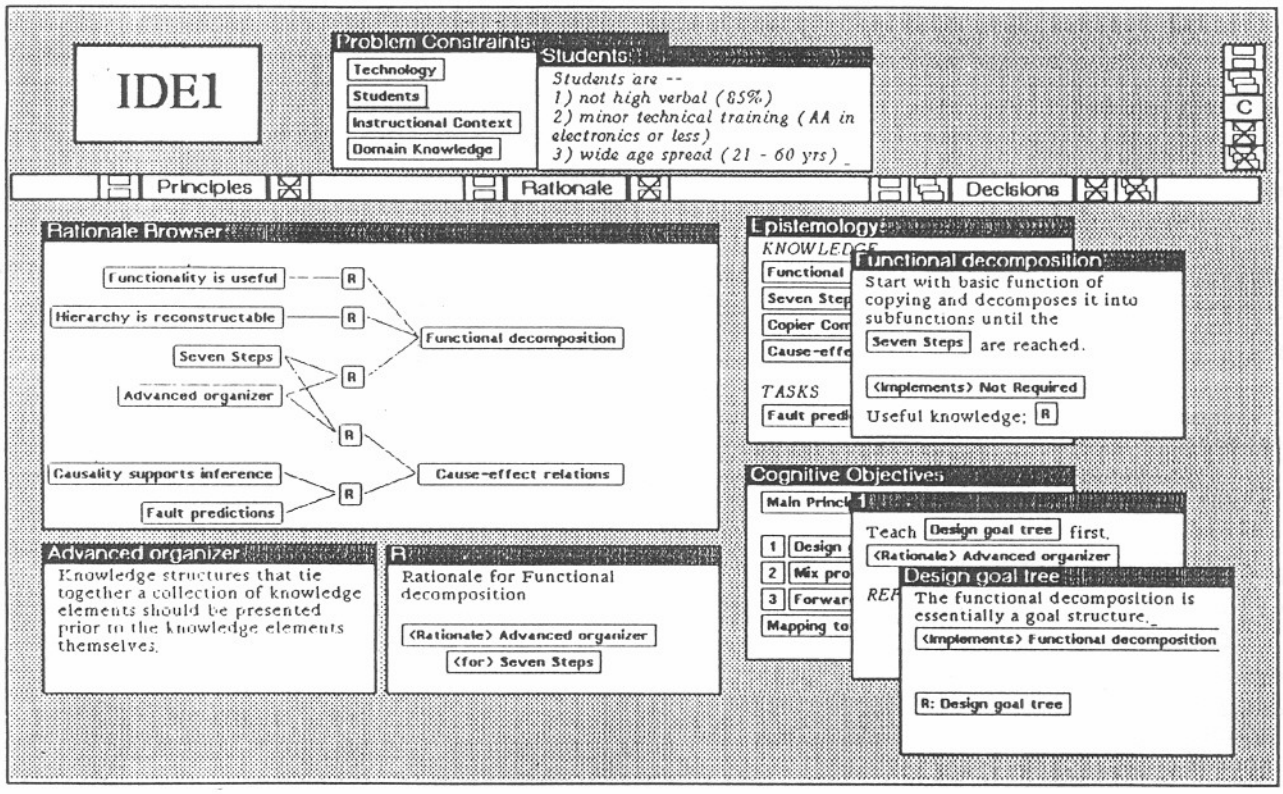


Figure 1.4: The hypertext system NoteCards. The Rationale Browser card displays a network of note cards. Nodes in the network expand into other cards, which in turn have keyworks which lead to other networks of cards. Figure from [15].

Data flow diagrams (DFDs), used for systems analysis and design, are also similar to our case study application. In DFDs, objects represent processes, data stores and system interface entities (for example, an employee is an interface to the system which supplies data). Connections represent information data flow in the system. An object's internal workings may be described by a data flow diagram, thus producing a hierarchy of data flow diagrams. Figure 1.5 shows a simple DFD with hierarchical structure.

Other application areas also exhibit the characteristic of having objects with numerous parameters and user access these to parameters exhibits locality. Typically these are computer interfaces to systems in which the user accesses many graphical control panels by time multiplexing a computer screen. Kantowitz and Sorkin in [18] describe computer simulated control panels for industrial and power generation processes which exhibit these characteristics.

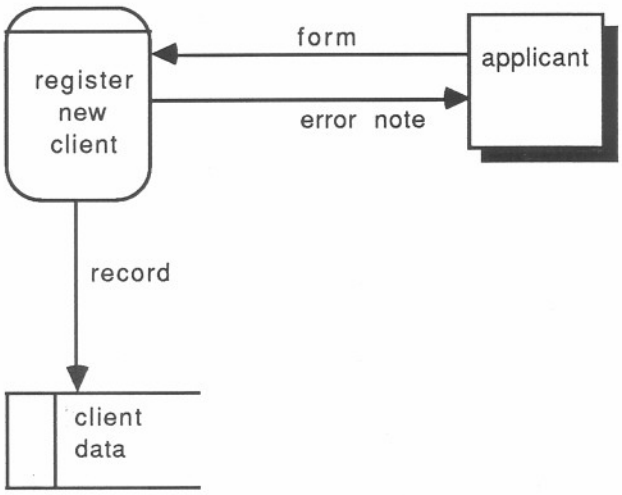
1.1.2 Meta Issues Concerning the Interface

The previous section has outlined the general characteristics of the case study interface. In this section meta issues independent of the application are discussed. To put this discussion in context, a description of the abstract characteristics of the interface is given. A discussion of meta issues then follows.

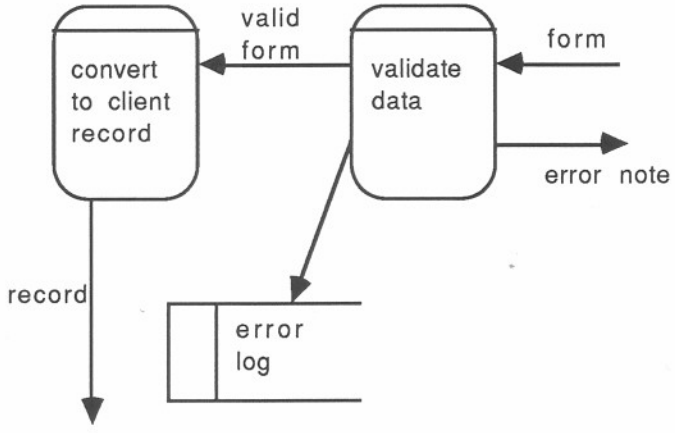
The system implemented consists of resources (referred to as objects) which can be selected, configured, and grouped into subsets. These subsets can be collapsed, thereby recursively creating new user defined objects.

The selection operation allows adding an object to a configuration and checking if an instance of the object is available. Figure 1.6 shows the graphical representation of a configuration. Objects in the system are represented by icons. Arcs between icons represent data flow connections between objects. The nature of the data flow (for example, control or signal) is application dependent.

The user may create new objects by grouping existing ones and their connections. The grouped objects and connections become the internal configuration of



a) The structure of a credit card application system.



b) The internals of the "register new client" process

Figure 1.5: Hierarchical structuring of data flow diagrams for system analysis.

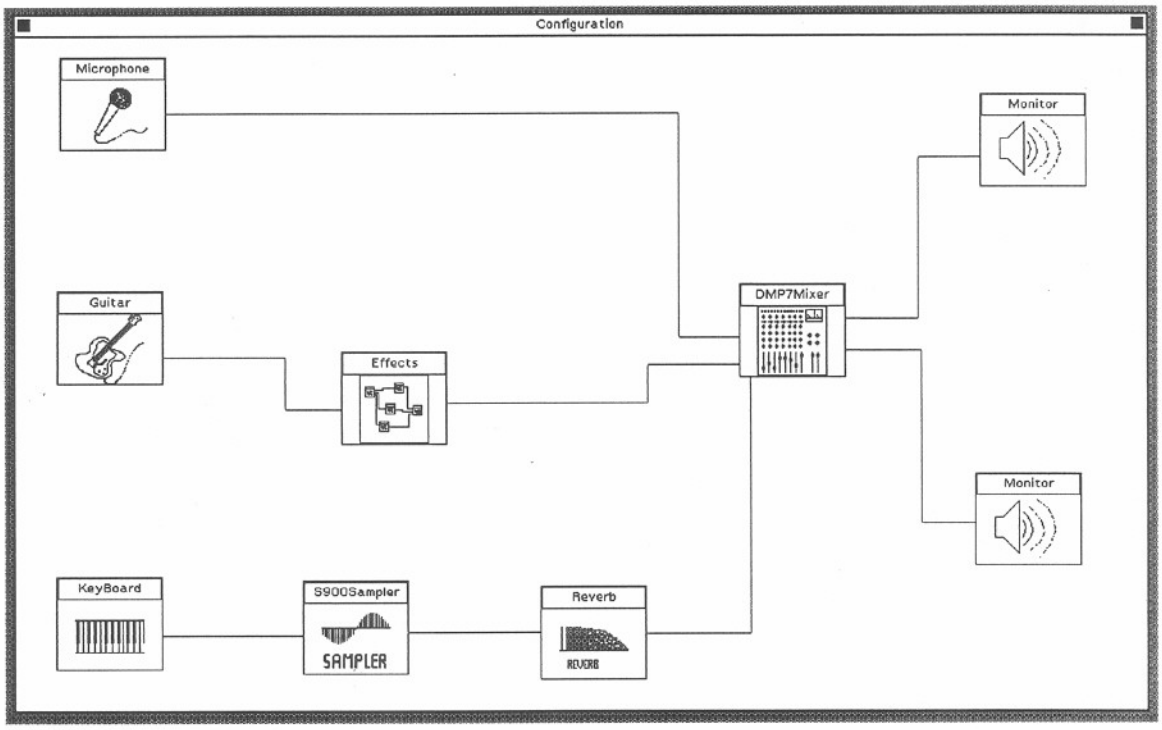


Figure 1.6: Graphical representation of a configuration

the module. This operation permits hierarchicly structured configurations. Conversely, the user can expand virtually any object to see its contents, or underlying semantics, in more detail.

The hierarchical structure of a configuration has ramifications in terms of the interface. Such structuring solves problems associated with complex configurations. These are considered to be configurations where there exist more objects and connections than can be displayed concurrently on the screen and more objects and connections than the user can comprehend in a single display. Hierarchical structuring allows the user to reduce the amount of information about a configuration by hiding details of sub-configurations. Since configurations are created by the user, it allows them to create layers of complexity in the system which reflect their understanding of the configuration. The system can a be thought of as an onion with layers of complexity. As the layers are peeled away, a more detailed view is presented.

While hierarchical structuring solves problems, it also introduces them. The

user's access path to the objects involved in a configuration is dictated by the hierarchical structure (the "navigation" problem). In the case study examined in this thesis, it is demonstrated that once the user has completed specifying the configuration, hierarchical access paths to objects are tedious. This has been shown true in other systems [23]. Typically, the user specifies a configuration, then requires access to the parameters of various objects within the configuration. If the user alternates between accessing two objects which are widely separated in the hierarchical structure, access time and effort become unacceptable.

The navigation problem can be reduced if we recognize three types of reasons for wanting to access an object: (1) to connect it to some other object ("hook-up"), (2), to operate its controls ("control"), and (3), to view its function or the context it is being used in ("context/function"). This is analogous to setting up one's home stereo; initially you are concerned with connecting the various components ("hook-up"). Once you are satisfied with the connections, you "hide" the connections against the wall and only access the front panels ("control"). At some point in the future you may forget how a component is connected to the system and need to review its role in the system ("context/function").

While hierarchical access is effective when accessing objects for "hook-up" or "context/function", in the case of "control" it is not. For example, the user does not want to have to traverse the object hierarchy each time access to a control panel is required. The point is some mechanism is needed to support non-hierarchical access. Thus, a central issue is, given the hierarchical access paths to the objects, how can non-hierarchical access also be supported?

The home stereo analogy can be extended to reveal another issue. Suppose you purchase a CD player. Many issues arise: is it possible to connect it to the system? Can this be done without reconnecting the rest of the system? Is there enough room in the stereo stand for the player? These questions concern system extensibility and can be applied to the case study: Can new types of objects be added to the system? How are the various attributes of the object (its icon, connection points, control panel, etc) specified by the user?

A related issue to non-hierarchical access is the phenomena of locality of access initially defined in the literature by Denning [8], and applied to windowing interfaces by Card and Henderson [7]. Card and Henderson reported that windowing interfaces in office automation applications exhibit locality in terms of access to windows. Figure 1.7 shows how user access to windows is related to the task being performed. The case study in this thesis exhibits the same phenomena: access to object parameters exhibit locality. The user varies the working set of object parameters being accessed. For example, the user may work for 5 minutes on the mixing board and effects devices, then transfer to using the synthesizers. Furthermore, over long time intervals the user may vary the entire configuration of the system plus the object parameters he is working with. These facts imply that not only must the system support non-hierarchical access to objects and their parameters, but it must allow the user to custom build access schemes which can be dynamically changed to suit their needs. In addition, what kind of tools are effective in helping the user gracefully change the entire system configuration?

An issue related to customizability is selective views and alternate views of the configuration and object parameters. Hierarchical structuring, as described previously, has been used to selectively view objects and connections in a configuration. The user may also wish to selectively view based on type. For example, in the case study different types of connections exist. There are times when the user wishes to view a configuration in terms of a certain connection type. Questions arise as to how the user specifies what types should be masked. Similar questions can be asked concerning alternate views of data. What is meant by alternate views is allowing several different types of representations of same data. As an example, in the case study, a numerical parameter can be viewed as a number or alternately as a graphical slider. The central issue here is how alternate views are applied by user. In addition this thesis examines the constructs that must be present in the implementation to support selective viewing and alternate views.

It has been stated previously that a characteristic of the application is that the user must control numerous object parameters. This presents two problems. First,

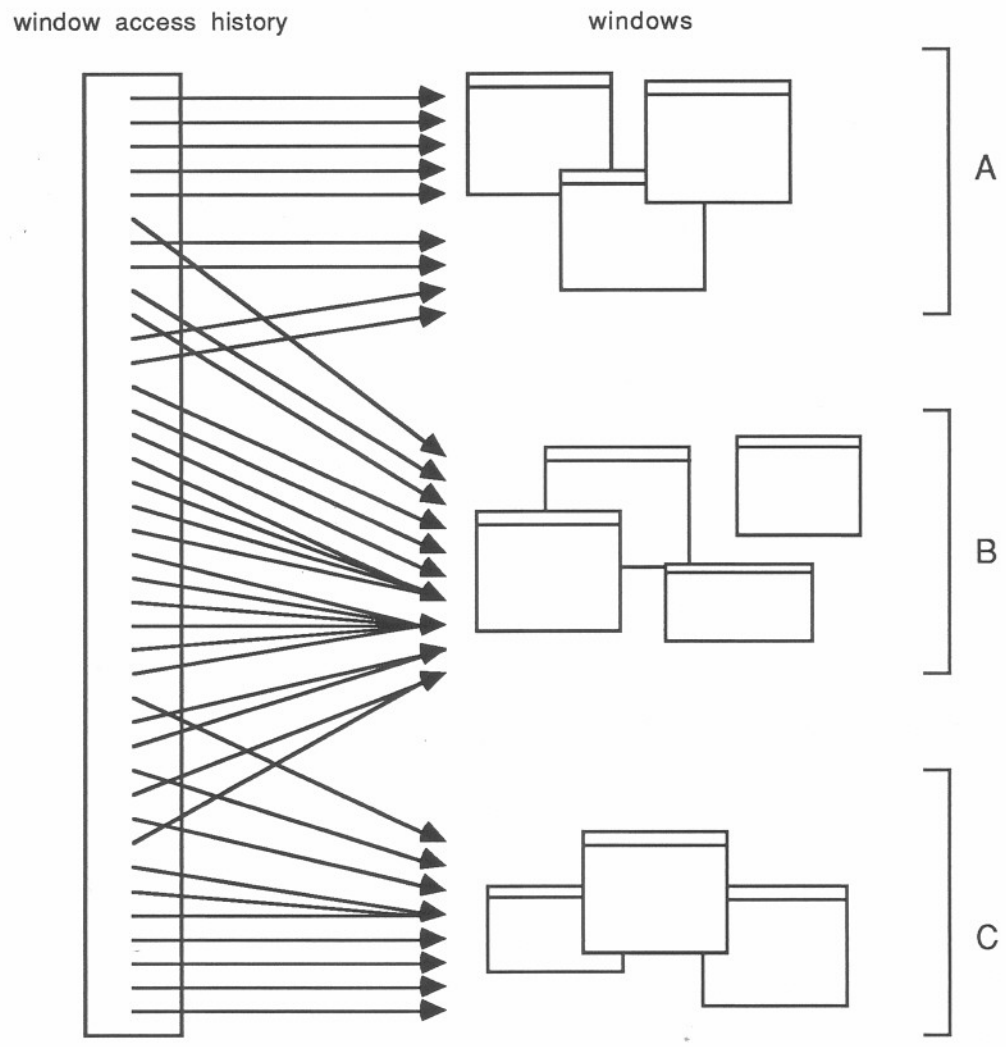


Figure 1.7: The phenomena of locality in windowing systems: In this example the user sequentially performs 3 tasks. As the user switches between tasks, the set of windows being accessed changes. Thus over times access to windows exhibit phases and transition between phases. In our case study, access to objects and their parameters exhibit the same phenomena.

given the fact that there exist many more parameters than input devices, what sort of scheme can be devised to facilitate linking an input device to a parameter? Second, even if an input device for each parameter is provided, it is physically impossible to manipulate all the input devices at once. In view of this, this thesis addresses schemes for extending control over parameters beyond simple one to one associations of parameters to input devices. Schemes to support one input device to many parameters associations, parameter control via functional dependencies and control through automation are investigated. The investigation is in terms of underlying software constructs that must be devised to support these schemes and the pragmatics involved in the user specifying associations.

Myers in [22] defines visual programming as referring to any system that allows the user to specify a program in a two (or more) dimensional fashion. Thus graphical programming languages are considered visual programming systems. The case study system can be viewed as a visual programming system: by "drawing" a configuration, the user programs the system. This viewpoint raises interesting issues. Typically, the user is concerned with prototyping configurations, that is, the user configures objects, checks to see if they operate in a satisfactory manner, then encapsulates them in module producing a new type of object. This is an experimental programming approach which is analogous to conventional programming languages where a block of code, once tested and proven to work, is placed in a parameterized subroutine. The issue is: Is this an effective programming technique?

A visual programming approach creates an interesting viewpoint of the interface pragmatics. Buxton in [4] defines the term pragmatics as the device and kinesthetic components of a user interface. Pragmatics, therefore, concern the human gestures of a dialogue and the transducers that capture them. Buxton also points out that the pragmatics of a user interface affect the learnability, the frequency of and nature of user errors, the retention of skills and the speed of task performance. If we adopt the view of the case study being a visual programming system, pragmatics can be considered the notation of our visual programming language. Notation, in any domain such as algebra, visual or symbolic programming, can be considered a tool

of thought which either helps or hinders the the performance of task. Thus we can ask the question: what notation best reflects and reinforces the way we think about the task? In other words: what gestures best suit encapsulation, connection and modification of object parameters? Furthermore, given the set of gestures used in the interface, how do these gestures interact with each other in terms of the user's model and the complexity of the implementation of the interface?

In summary, this chapter has outlined the basic issues surrounding this thesis. First the issue of investigating broad issues in terms of a case study was discussed. It was shown viable for three reasons: First, the construction of a working system is extremely valuable, second, results from the case study can be generalized to other problems and third, there exist many problems to which the results apply. Next, the meta-issues concerning the case study itself were outlined. The central issues consisted of the user interface ramifications of hierarchical structuring of access to entities, alternate and user customizable access schemes, selective and alternate views of entities, the effectiveness of experimental programming, pragmatics of interaction, and control over object parameters.

Chapter 2

Description of the Case Study Problem

This chapter gives a description of the case study problem. This description is intended to set the stage for a discussion of solutions to the meta issues described in the previous chapter and application specific issues presented in the current chapter.

The case study problem is to make an audio studio controllable from a central computer. The domain of the case study is a personal audio studio in which a single user operates the entire system. In recent years, personal audio studios, which are merely scaled down versions of professional audio studios, have become extremely popular due to reduced hardware costs and improvements in sound quality; it is now possible for a user to produce personal recordings of professional quality. The user, while using a personal audio, assumes the roles of performer, producer, engineer, instrument builder and composer.

This domain provides a potent testbed for studying the issues of interest discussed in the previous chapter. In everyday operation the user of a personal audio studio is faced with the tasks of configuring (or interconnecting) objects and controlling object parameters. Furthermore, the user may spend a great deal of time probing the the configuration of the system in order to understand or troubleshoot it. By constructing a system to deal with these problems we directly assault the

issues described in Chapter 1.

This domain was chosen for two reasons: it restricts the size of problem, thus making a solution more manageable (it was possible to assemble our own personal audio studio) and, because a personal studio is a scaled down version of professional studio, the same problems are encountered; thus ideally, our solutions can be scaled up and applied to larger professional systems.

The question now to be asked is: why take the computer control approach? In order to avoid prescribing a cure without a cause, the following sections describe problem areas that exist in the operation of personal and also professional audio studios. These problems will shed light on the reasoning and motivation behind our approach. In order to give context to the discussion of these problems, a more detailed description of the domain is first given.

2.1 User and Task Profile

Rubenstein and Hersh in [27] stress the importance of identifying and profiling potential users as part of system analysis. The typical users of personal audio studios are musicians. In terms of user profile we can address the following questions:

1. how experienced is the user with technology?
2. how experienced is the user with the application area?
3. what is the user's motivation for using the studio?
4. how intelligent is the user?
5. how critical of the system is the user?
6. what are the common tasks performed by the user?

In order to answer questions 1, 2 and 4 we need only look at the hardware that is commonly used in a personal audio studio. Generally, several types of instruments, such as keyboard synthesizers, guitars and percussion, serve as sound

sources. An audio mixing console is used for blending and processing sound from various sources. Multi-track tape recorders are used so a single musician can "play" several instruments on the same recording. Sound effects devices, such as reverberation simulators and harmonizers, allow the musician to simulate different types of acoustic spaces and enhance sounds. A micro-computer may be used to score pieces and drive synthesizers.

The point is that the personal music studio is a very complex environment and the user of this environment is highly experienced with the technology. While users may not be very aware of how a device in the personal studio works internally, they are very aware and experienced in what the devices are used for and how to use them.

The answer to question 3 is related to question 5. The user's motivation for using a personal studio is generally to produce recordings for personal use or for sale. This motivation makes the user very critical of the system: musicians generally expect high quality audio as an intrinsic feature of their work, and music produced for sale must meet professional quality standards. Furthermore, the user expects or hopes that the technology will help, not hinder, the creative process of creating music. These factors make the user very critical of the system and the ease of operating the system.

The final question is difficult to answer. The activities of a musician in the studio may vary greatly depending on hardware available, type of music being created and personal work habits. For example, if a multi-track tape recorder is available, some musicians construct a composition by playing and recording each instrument one at a time; thus building up tracks until the composition is orchestrated. In contrast, other musicians record a composition in one "take", using a computer sequencer which has been programmed to play all synthesized instruments at once. The point being made is: the user's tasks vary greatly and we can only isolate very general task types:

- instrument construction

- composition
- recording
- mix down

By instrument construction we are not referring to carving guitars out of trees, but to connecting audio devices together in such a manner that another type of instrument is, in essence, created. It is the design of the audio "chain" of sources, effects, and combinations, and the settings of each. As an example, the output of a guitar can be connected to an audio envelope generator device such that strumming the guitar controls the opening and closing of the envelope generator. The sound from a keyboard synthesizer may then be passed through the envelope generator. Thus, strumming the guitar causes the effect of the synthesizer sound cutting in and out and, in effect, a new strange instrument has been constructed.

Composition refers to the process of creating a "plan" which describes how a piece of music is to be played. Note that this may not necessarily be the traditional scribing of music onto paper. The "plan" may also be recorded by entering the composition into a computer sequencer or actually recording a performance on a tape recorder.

The task of recording takes place when the user has completed the composition. Essentially recording is the acoustic realization of the composition. Recording generally involves the playing and recording of several instruments onto a multi-track tape recorder as described earlier or the programming of a computer sequencer to play each instrument part.

The mix down process refers to blending of the various instruments of composition into a stereo recording. This generally involves the use of an audio mixer, sound effect processing devices and a stereo tape recorder.

The essential observations to be made about these tasks are that:

- devices serve different roles in different tasks and
- there is no concrete ordering of tasks.

The main point to be extracted from our task analysis is that the types of tasks and the ordering of tasks cannot be predicted accurately. Therefore, we must design a system in which there is provision for variability of task and user tailorability.

2.2 Motivation

Now that we have clearer idea of who the user is and what sort of activities take place, a discussion of our motivation behind the development of the case study system can be given. Our motivation falls into three categories:

1. Expertise and interest in the application.
2. There exists a gap in computer tools for personal audio systems.
3. We believe that computer tools could solve certain problems associated with personal audio studios.
4. The development of the case study system is critical to the success of future personal audio studios.

The following subsections discuss each of these categories.

2.2.1 Lack of Computer Tools

There exists a definite gap in computer tools for audio engineering. In recent years there has rapid growth in the use of computer tools for many areas of music production. This growth has followed a bottom up pattern. Initially computers were used in sound synthesis and processing. The next generation of music software consisted of composition tools aimed at making the composition process more productive. In the current generation of music software tools we are seeing composition tools in which the computer generates portions of the composition and tools to assist in audio engineering. Thus the pattern of growth can be seen going from the low level activities, such as sound generation, to higher level activities like composition and audio production.

Many effective computer tools for sound synthesis/processing and tools for composition have been produced and a excellent survey is given by Yavelow in [36]. The market for sound synthesizers offers numerous synthesizers using a variety of sound synthesis techniques. Tools, such as voice (or "timbre") editors, are available to assist the musician or audio engineer in synthesizer programming. Similarly, numerous compositional tools are available which run on a variety of microcomputers. The development of sound synthesis/processing tools and tools for composition has undergone several generations and many effective and efficient tools have evolved. In contrast, we are just seeing the first generation of tools to assist in audio engineering. Based on the success of computer tools in other areas of music, we are motivated to produce tools which will assist the audio engineer.

The real underlying issue is that while many good hardware and software modules exist, these modules do not function together as a consistent and integrated system —integration is the missing component. Ultimately, what is required is a "meta tool" which brings together all these modules by providing a unified, consistent user interface across modules. This "meta tool" should also be an extendible system where function can be added by "dropping in" new modules. Furthermore, communication between modules in terms of audio and control data should be standard.

2.2.2 Problem Areas

Task Switching

In the process of audio production many software tools may be used. In a typical modern recording studio you may find a variety of synthesizers each which support a different synthesis technique and a different style of user interface, several different types of microcomputers with different voicing programs used for programming the synthesizers, another microcomputer used for composition and sequencing, and yet another computer used for mixing console and tape recorder control. A system consisting of an integrated set of software tools does not exist. A state of the art

professional studio requires three operators, above and beyond the musicians, to man the various computers.

The lack of integration among software tools can severly hinder their effectiveness. This can be traced to two factors: high cognitive load and high cost task switching. Cognitive load is defined as the amount of knowledge required by the user to effectively use the tools. The user is already faced with a formidable task of remembering the functions of the various tools; the fact that the pragmatics of the interfaces may vary from tool to tool places an additional and unnecessary cognitive burden on the user. As an example, consider a voicing program and a sequencer. If the style of the user interfaces between the two varies (ie: one uses pop-up menus while the other uses pull down menus) the user must not only remember the difference in function of the tools, but also the different interaction technique.

The problem of integration also extends to hardware device interfaces. Typically an audio studio is a patchwork of numerous hardware devices such as mixing consoles, signal processors and tape recorders. Not all the devices are products of the same manufacturer and therefore the pragmatics of the front panel user interface are different from device to device (this may even be true for different devices from the same manufacturer). Differing pragmatics between device interfaces have two effects: when the user switches between adjusting devices, it may lead to a high frequency of user input error if the differing pragmatics interfere with each other, or it may distract the user from the higher level task at hand. The first effect is costly in terms of time to perform the adjustments. The second effect's cost is more subtle; a user's current train of thought may be broken and valuable ideas lost if distracted by a low level task. Austin in [1] refers to this phenomenon concerning music production. In music production where creativity has high value, such an effect can be costly.

Not only is the audio engineer burdened by the cognitive load of remembering the function and how all the devices in the studio operate, but there may be high cost in task switching. Devices may be connected together in different manners

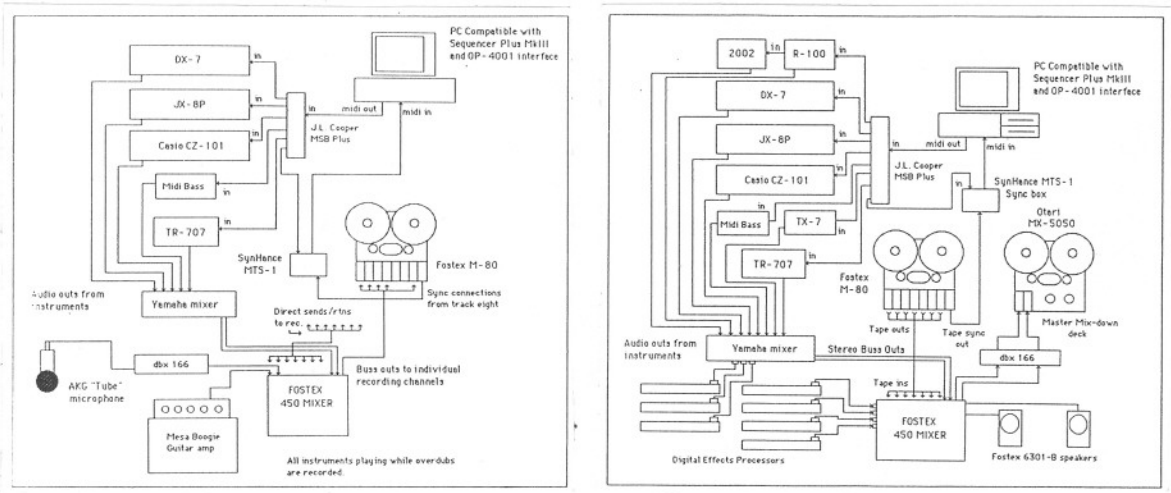


Figure 2.1: Two of many configurations of an audio studio. The first is the configuration used for recording vocals and guitar, the second for final mixdown. Figures from [24].

depending on the task. Figure 2.1 shows an example of the various configurations of the studio. The problem, as outlined in chapter 1, occurs when the user switches between tasks: what needs to be rewired? Heinbuch, in [16], describes this problem as "studio headache #1". In addition there is the problem of remembering what tools (be they software or hardware) were used in the task. Typically the user may want to temporarily change tasks then resume the original task or try a new configuration with the option of returning to the original. The cognitive and time costs of reconfiguring the studio can be staggering. Typically, 1/3 of studio time is consumed in this task, time in which musicians are idle, yet both musicians and studio rental are being paid. The user has to manually rewire the studio using a patch bay. A patch bay is a central switch of inputs and outputs for all the devices in the studio and, although functionally capable, it provides the user with a less than adequate tool for the job. Only small labels beside each input/output in a patch bay indicate the associated device input/output. Thus, it is difficult for a user to identify which input/output in the patch bay is associated with a particular device. This results in user confusion and further complicates repatching.

Once the studio is rewired, the tools needed for the task must be acquired.

For software tools this may mean restarting a program and opening a few files. Hardware tools may have to be set back to the settings used for the task and also brought closer if the user requires them to be close at hand.

Thus a possible solution to these problems is to have the computer remember the way things were set up and be able to restore the set up of the studio automatically. Ideally we desire a magic wand which allows the audio engineer to "change things back to the way they were" so the real task at hand, creating music, can be concentrated on.

Better Interfaces

In the preceding section we discussed the problem of hardware devices having different interfaces and how this can lead to poor user performance in the operation of these devices. A problem related to this stems from a recent trend in the design of hardware devices. The problem is simply that in order to reduce a device's cost, manufacturers produce devices with a minimal number of switches, knobs and buttons and very small displays. What few buttons remain on the front panel of a device are generally of the "soft" variety. That is, they are modal—serving different functions at different times. Figure 2.2 shows the control panel of a popular audio mixing processor. The device has 205 settable parameters while only 53 physical inputs exist. A 32 character display is used to display parameter values.

While "clean" front panels give the impression that a device is easy to use, the modal interface that results from time multiplexing a few buttons, sliders, knobs and the display may be frustrating to the user. Rubenstein and Hersh [27] report multiplexing of function (using the same input for more than one purpose) is usually poor in terms of human performance. Evidence of this can be seen in the prolific sales of more extensive "add on" control panels and micro-computer programs for controlling devices with minimal interfaces, and is reported in the literature [36].

Given the fact that a device's parameters can be controlled from a computer, we are capable of producing "soft" control panels. Since there is no cost involved in supplying graphical knobs, buttons, sliders and displays, we can avoid the problems

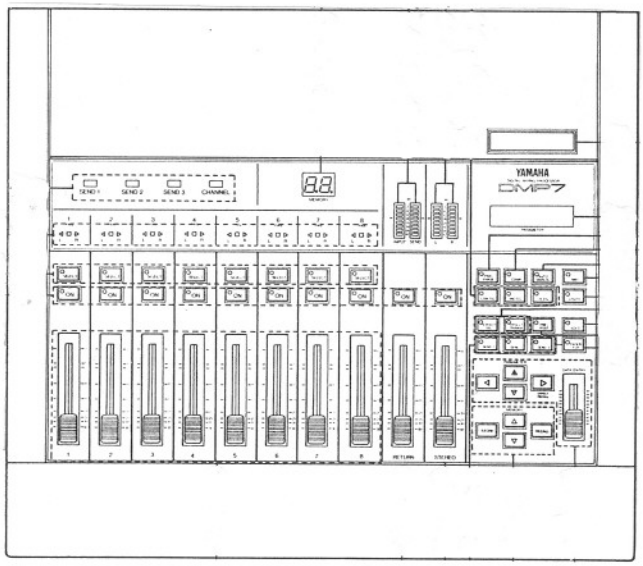


Figure 2.2: Control panel of the Yamaha DMP7 audio mixer.

involved in multiplexing of function.

Controlling a device from a computer screen control panel has another important advantage: parameter displays and parameter modification gadgets can be made to reflect the semantics of the parameter. An example best explains this. Figure 2.3 show the equalization section of a channel module of a sound mixing console. In this case, the physical knobs and buttons serve as a display of the equalization "setting" in addition to being a means to modify the "setting". Figure 2.4 shows another representation of equalization used on a computer screen control panel. We conjecture that Figure 2.4 better reflects the way the user "thinks" of equalization. In other words, Figure 2.4 is closer to the functional model the user has of equalization. The point to be made here is that the closer the representation is to the user's concept of function (that is, their mental model of function) the more effective we expect the user interface to be in terms of learnability and ease of use. For example, when the function of an equalizer is taught, generally decibels versus frequency graphs are used as opposed to knobs and buttons.

The previous point should be qualified. We are not claiming that a "correct"

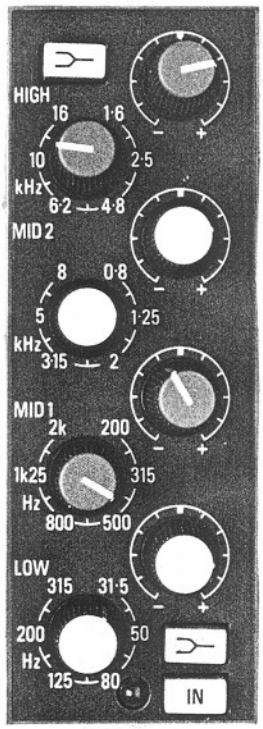


Figure 2.3: The equalization section of a channel module of a Neve V Series sound mixing console.

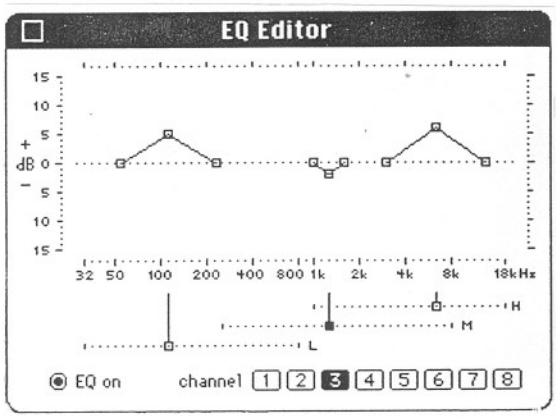


Figure 2.4: The EQ editor of DMP7-Pro.

representation of certain type of parameter exists; the "correct" representation depends on the user and the situation. The critical point is that soft control panels allow the option of alternative views of parameters. For example, the user may prefer the representation used in Figure 2.3 over the representation used in Figure 2.4 in certain situations. Furthermore, certain representations may be good for visual representation but poor in terms of input. For example, consider the representation of equalization using a graph where the user may modify the graph by drawing it. Although this input technique allows the user to quickly set the entire equalizer, users cannot "tweak" certain portions of the equalization as they can with traditional knob controlled equalizers.

More power to the user

Historically, when computers have been used in an application to make a task easier, they have also been effective in extending the power the user has. An example from computer aided composition can be used. Not only is it easier for the composer to get creative ideas recorded on paper, but the computer can perform complex material enabling the composer to "proof listen" to the material being composed or edited. Thus the composer's creative power is enhanced. In a similar manner we are motivated to develop computer aided audio production tools which extend user power. For example, the user may only be able to operate ten faders on the audio console at the same time. By having the computer control the faders we can perform operations that are not physically possible.

As described by Schwarts in [29]:

> The issue of physically interfacing a large and complex process with its human operator has plagued system engineers for over a century. In the past, a compromise between ease of operation, cost and physical constraints was inevitable, whether the control panel was for a petrochemical plant, nuclear reactor, aircraft or audio mixer. Even in the best of compromises, large systems still require more that one engineer for multiple simultaneous operations. In audio, the mixing desk/tape

> transport/signal display devices frequently occupy over 50 square feet of front panel with on the order of 40 controls and labels/displays per square foot. A single audio engineer simply cannot reach every control from one position. Thus we are concerned with developing schemes to overcome problems of this type.

Extension of the user's control falls into 2 broad categories: first, computer automation and second, control slaving schemes. In the latter category, the user specifies to the system that a single control will control more than one parameter in the system. In this way the user can "grow more hands". In the first category, the user indicates to the computer that it should "record" the manipulation of controls versus time or the user can "script" actions or configurations. The recording or script can be "played back" later, thus freeing the user's hands to adjust other controls.

Control slaving schemes can be more general. Traditionally, control slaving schemes have involved simply slaving sets of controls, which the user wishes to manipulate at the same time, to a single control, which the user actually manipulates (hierarchical control). A more general scheme is to allow arbitrary slaving of parameters to other parameters. As an example of the usefulness of a general scheme of this sort, we can examine a typical "trick" used in the recording of popular music. The trick is to apply an echo to a recording and make the time delay before the echo is heard some multiple of the tempo of the piece. This trick has the psychoacoustic effect of giving the recording livelyness and bounce. Typically the engineer determines the tempo of the piece, calculates the amount of time delay needed and sets the time delay parameter on the echo device accordingly. This technique works very well when the tempo of the piece is constant; when the tempo of piece varies, the effect of this trick makes the piece sound out of time. Ideally, what the engineer wants is a relation that specifies that the delay of the echo is at all times some fraction of the tempo of the piece. Thus, when the tempo of the piece varies, the delay shall vary with it and the intended effect will be maintained.

Thus we would like some method of relating controllable parameters in the

system such that we could "set and forget them" or in other words, apply constraint relations among parameters. The system would then be in charge of updating the parameters according to the relationships between them. Hence the user can turn his attention towards manipulating other parts of the system.

2.2.3 User Interfaces for Digital Audio Workstations

In recent years we have seen much development in computer systems strictly devoted to audio processing . This class of systems, referred to as Digital Audio Workstations (DAWs), process all audio signals in digital format. Digital processing has several fundamental advantages: there is none of the quality loss typically experienced with analogue processing and more sophisticated processing of audio signals can be performed. The essential idea, or inspiration, behind DAWs is that they will provide integrated audio signal processing. A DAW will emulate all the traditional devices of the personal recording studio plus give the user more power to manipulate these devices than the traditional analogue systems. Furthermore the user will be able to control all this power from one single central computer workstation.

DAW systems generally consist of programmable networks of digital signal processors in which all processing and routing of audio signals takes place in the digital domain. Sets of audio devices are emulated by asynchronously running programs which input digitized audio signals and perform processing on them. Device connections are emulated by interprogram communication of digitized audio signals and control signals. A central computer, which pervades over device programs, is used for control and user interface. Control panels for emulated devices appear on a high resolution graphics screen, where the user may directly manipulate device settings. A common feature of these systems is the ability of the system to be reprogrammed to perform different audio processing tasks. For example, by loading different programs into SoundDroid, a DAW developed by The Droid Works [21], it can be an audio mixer, a music synthesizer, an audio recorder or an outboard signal processor such as a reverb, flanger or pitch shifter. This feature of "virtual devices" is extremely powerful. It allows the user to have different types of devices available

for use, while paying only for the hardware cost of one device. Furthermore, since devices are software programs, replacing obsolete devices with new ones is extremely cost effective. These features make systems of this type extremely attractive to the user-owner and give reason to the belief that these types of systems will be the personal and professional audio studios of the future.

The complete development of a DAW has not come about yet. While much research has been done on DAWs, this research has been focused on low-level functionality. Many systems (SoundDroid [21], Compusonics DSP-2000 [29], WaveFrame [25], IRCAM's 4X [13] and the Katosizer [19] [3]) have concentrated on low level problems and advantages of processing digitized audio signals and the type of hardware architecture needed to support a DAW. Little research has been done on the user interface requirements of a DAW. It is our belief that research in this area is critical to the success of DAWs; while quality, powerful function and virtualness attract musicians, the ultimate success of these systems will depend on the interface that the musician sees. The interface must be able to present the system's powerful function in such a manner that the musician can understand and control this power. Thus the case study focuses on the development of a user interface for systems of this type.

This motive is not in conflict with the motives previously described; the problems encountered in a traditional personal audio studio also occur in the realm of DAWs. In order to be effective, the software tools presented by the DAW must a be an integrated set. Switching between tasks on a DAW still requires "rewiring" of the connections between devices and reacquisition of the tools for the task. We must still be concerned about producing high quality interfaces for "soft" devices because traditional front panel interfaces do not exist. Finally, the potential for enhancing the user's power is equal, or greater than the potential for power enhancement in traditional personal audio studios.

Essentially, designing a user interface to a DAW presents a golden opportunity to develop a system that solves the problems associated with traditional personal audio studios, and exploits the inherent advantages of a DAW.

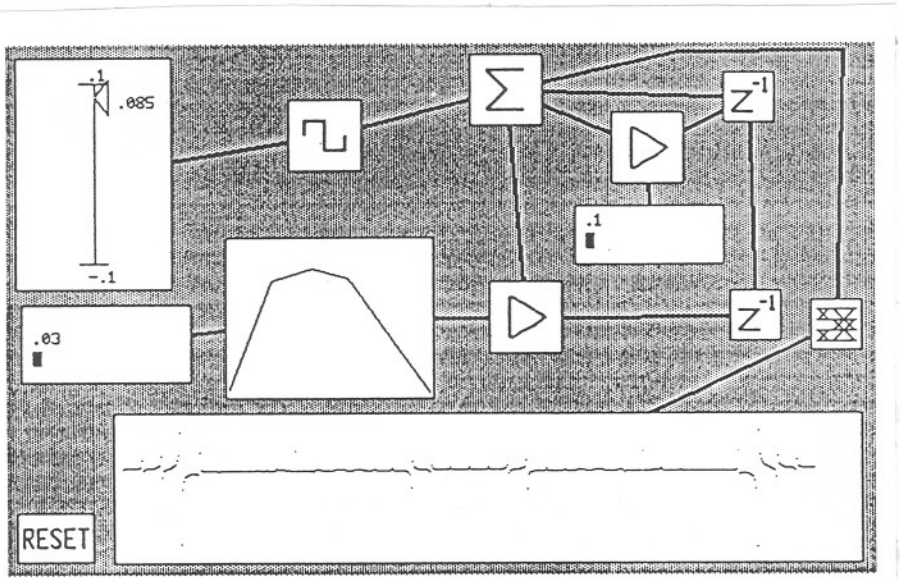


Figure 2.5: A graphical representation of a second order filter. The filter has been swept by a hand drawn waveform and its output is displayed in a graph. The icons represent primitive devices, each carries out a simple operation on the audio signal. The combined operation of the primitive devices results in the input audio signal being filtered on output. Note that control signals are also represented by connections. Figure from [11].

We have outlined that a major advantage of DAWs is their programmability. Blythe and Kitamura [3] have showed the tremendous advantage in the ability of the user to acquire and interconnect devices interactively. In a sense, the user visually programs the system to emulate a certain set of audio devices with a certain set of connections among the devices. Galloway, et al. in [11] show how an interactive graphical language can be used to specify a device's function. Figure 2.5 shows an digital filter constructed using interactive graphics. By combining these two techniques, a user could not only specify what devices are to be emulated on the DAW, but also the internal function of the devices.

This ability has potential far beyond the traditional personal audio studio. In the traditional case, the function of a device is "locked" inside the device's physical construction. That is, the function of a device is determined by the designer of the device, generally the engineer who designed the device for the manufacturer. But in the DAW case, the user can be the engineer. The advantage is that the user can custom build devices to suit their personal needs and, in doing so, the user acquires

a clear understanding of the function of a device and, ultimately, the function of the system. There are also some inherent problems: device construction is a complex task; can we produce an interface which allows the user to accomplish it?

As outlined in chapter 1, the approach we take is to allow the user to construct devices using hierarchical encapsulation. This approach allows the user to bury the complexity of the system in "chunks". These "chunks" correspond to user built devices and consist of a collection of devices and connections. At the lowest level of the device hierarchy, there are primitive devices in the system which represent small pieces of micro-code. The micro-code of a primitive device, when executed, carries out a small audio signal processing task. For example, a primitive device may read an audio signal input, amplify it by multipling by a scalar and output the result.

Figure 2.6 shows the embedded nature of our approach. At the top level, devices are interconnected to perform a task. These devices may have an internal structure defining their function. This structure consists of smaller, primitive devices and other devices plus the connections among them. At the lowest level primitive devices perform simple functions such as amplifying, delaying or mixing audio signals. Also note that devices inherit their controls on their control panel from their lower level devices.

It is important to note that our approach does not view DAW systems and MIDI module based systems as separate entities. It would be naive to expect a DAW which cannot interface to existing MIDI modules to be popular; many excellent MIDI modules exist which audio engineers and musicians would be hard pressed to give up. We see the evolution of the hardware base of our system beginning with strictly MIDI modules, evolving to a combination of a DAW interfaced to MIDI hardware, and finally a DAW emulating all modules. The point is that while the hardware base is changing our interface will not. The fact that a device is a MIDI module or is emulated by a DAW program should be transparent to the user. Moreover, it is our vision that MIDI modules should be programmed using the same technique as DAW modules composed of micro-code. This has two advantages. First, it allows

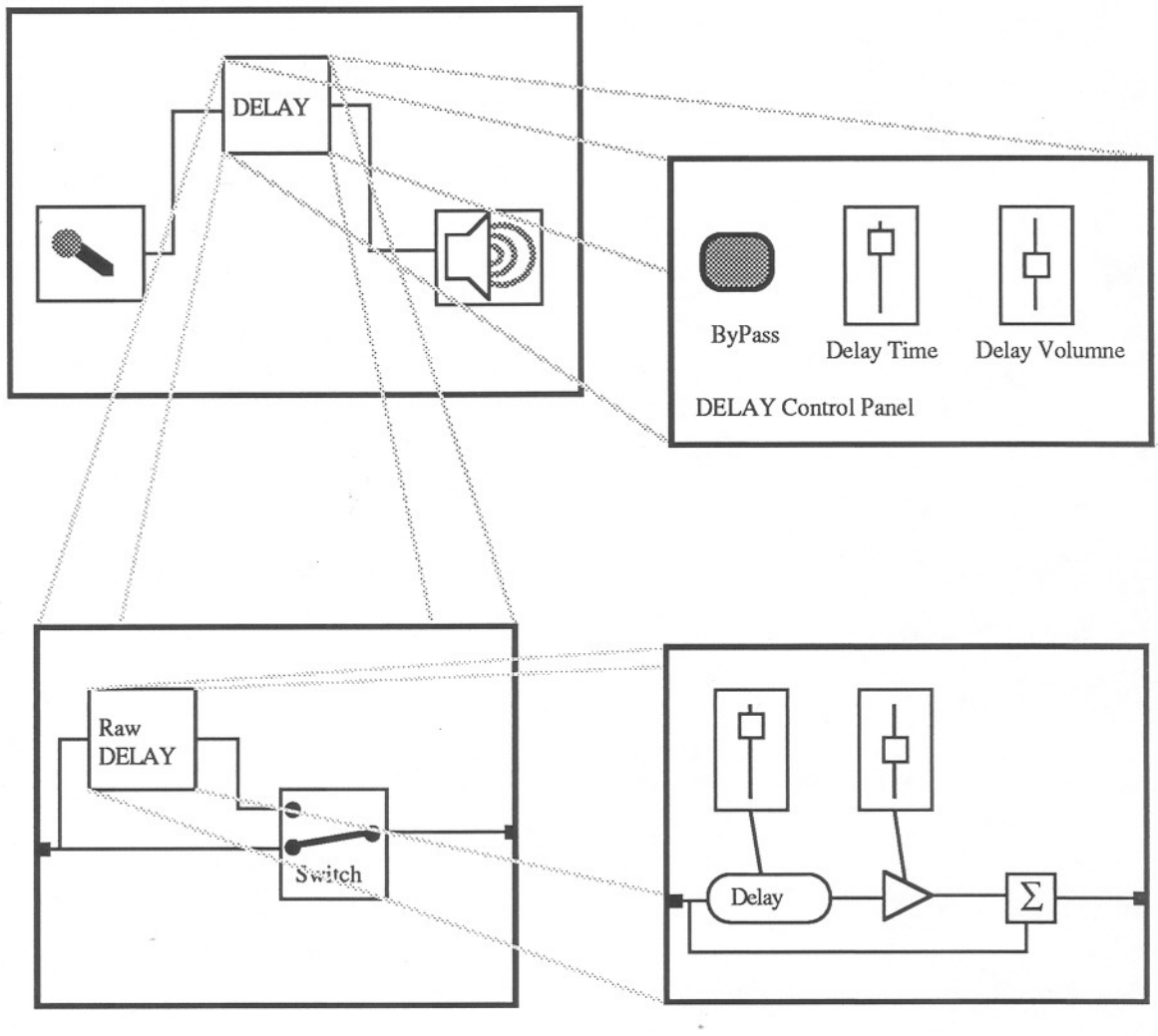


Figure 2.6: How devices are embedded in the hierarchy: at the top level a microphone outputs to a DELAY device which outputs to a monitor speaker. The next level down shows the DELAY device consists of a Raw DELAY device connected to a bypass switch. The next level down shows a Raw DELAY is composed of a primitive delay, amplifier and mixer, plus 2 control devices. The DELAY Control Panel shows how controls are inherited from lower level devices.

MIDI based system users an easy transition to a DAW; programming a DAW will require the same skills a programming a MIDI system. The second advantage is more concerned with interface design; by making our approach to designing an interface to a DAW similar to our approach to MIDI based systems, the techniques used to solve problems in the latter can be applied to problems in the former.

2.3 The Case Study Environment: Virtual Studio

The hardware arrangement used for the case study audio studio is shown in Figure 2.7. The audio devices used included a Yamaha DMP7 mixing console, an Akai S900 sampler, a Yamaha TX802 Tone generator, an Akai AX73 keyboard synthesizer, a Digital Music Systems MIDI mixer/switcher and an Akai DP3200 Audio Patch Bay. All these devices are controlled from a central computer via MIDI communication lines. All audio inputs and outputs are connected to the Akai DP3200 audio patch. Under computer control, the audio patch bay allows routing of any input to any output and visa versa. Similarly, all device MIDI inputs and output are connected to the Digital Music Systems MIDI mixer/switcher, thus allowing computer control over the routing of MIDI control signals.

The central controlling computer is a Sun 3/50 workstation. The computer communicates through its RS232 output port to a RS232 to MIDI converter, which in turn is connected to the MIDI mixer/switcher. The interface, which allows the user to control the configuration of the system and access "soft" control panels for each device, is named Virtual Studio. The physical input devices to the interface consist of the workstation's mouse and keyboard (Chapter 4 discusses using more and different types of input devices). The display screen is monochrome with a resolution of 1152 * 900 pixels.

Virtual Studio is implemented in the Smalltalk programming environment. Figure 2.8 shows a snapshot of a typical screen that appears in the virtual studio. In this situation the user is accessing a tool to configure the audio connections in the

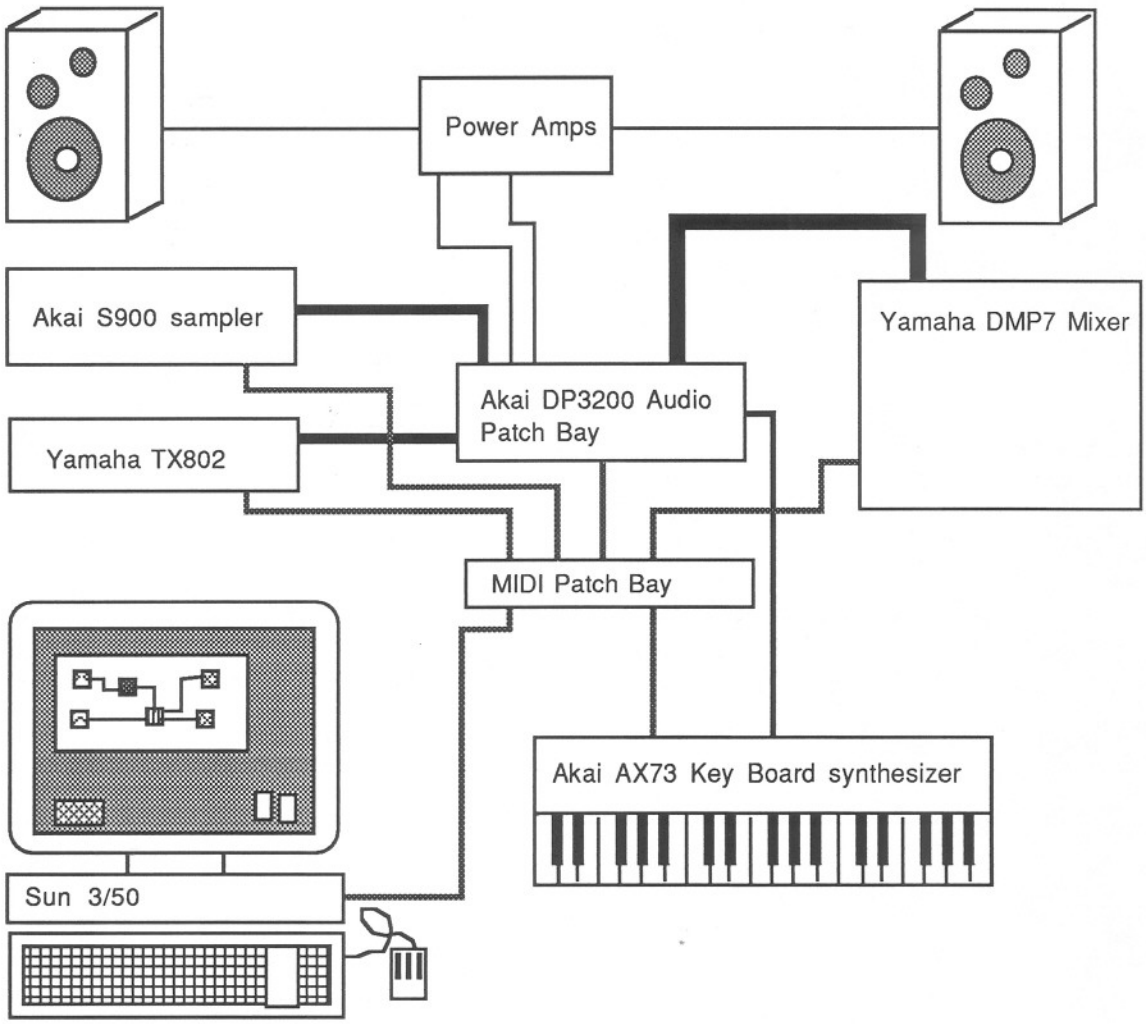


Figure 2.7: The hardware used in the Virtual Studio. Gray lines represent MIDI in and out connections. Solid lines represent 1 or more audio connections.

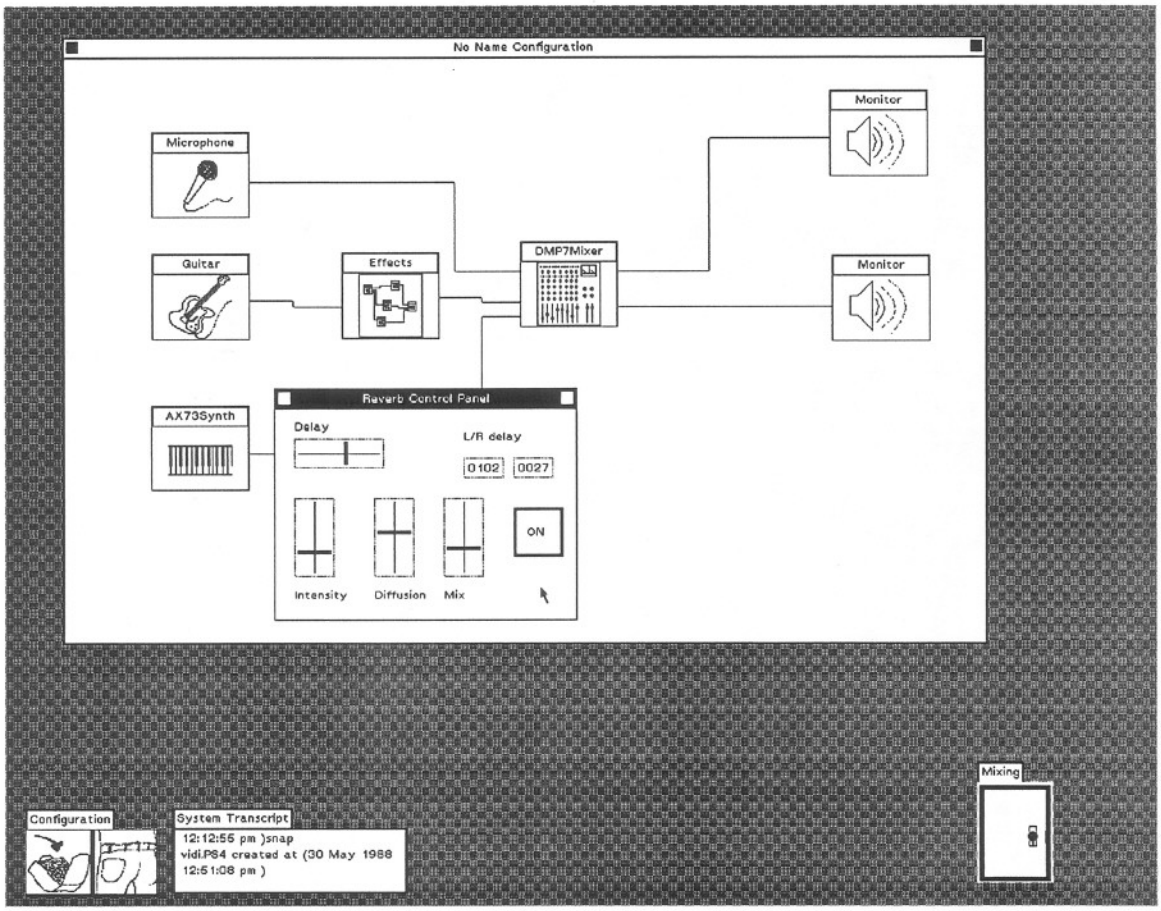


Figure 2.8: A typical screen in Virtual Studio.

system.

2.4 Summary

In this chapter a description of the case study environment and the associated problems were given. The case study environment was identified as the personal audio studio. The description consisted of a profile of the user and the type of tasks performed in the personal audio studio, the problems hoped to be solved by developing the case study system, and a description of the hardware used in the development of Virtual Studio, the case study system.

Chapter 3

Survey of Related Systems

In this chapter we examine systems that are related to the case study problem. More specifically, we survey audio systems which have a central controlling computer and concentrate on the interface presented by these systems. The systems surveyed here are not comparable to one another due to the fact that each system was designed with a different function in mind. Fortunately, we are not concerned with comparing the systems, but in examining the features of each system which are related to problems being addressed in this thesis.

The area of computer user interfaces for audio studios is relatively unexplored. Most research concerning computers and audio has centered around low-level functionality and hardware architecture [21]. The literature reflects this focus; for some systems very little description or documentation of the interface has been produced.

3.1 The Katosizer

The Katosizer is a digital audio workstation (DAW) developed at the University of Toronto by John Kitamura [19] [3]. The hardware consists of micro programmable digital signal processors (DSPs) and utilizes a recently designed architecture which supports multiprocessing. The Katosizer is modular in design, so that an inexpensive minimum configuration can be expanded into a large and powerful machine. Micro-code modules loaded into the DSPs emulate the function of conventional

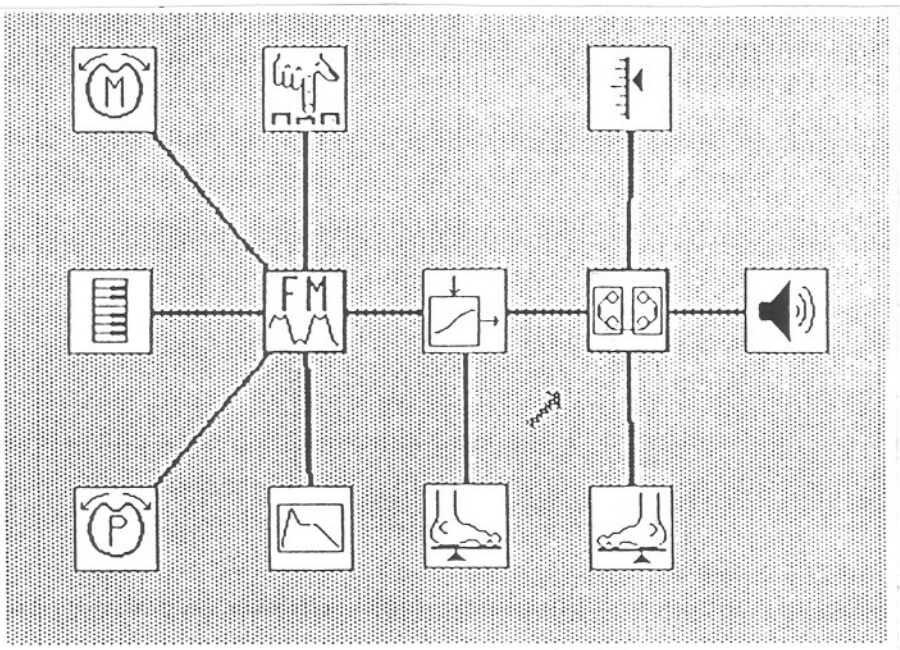


Figure 3.1: The user's view of the Katosizer.

audio signal processing and synthesis modules. By adding more processors, more audio devices can be emulated.

The user interface developed for the Katosizer is called Virtual Patch Cords (VPC) [3]. Figure 3.1 shows a screen dump of the user's view of the Katosizer. It shows a "patch" (patch is slang for a particular set of connections and devices) in which icons representing the function of micro-code modules and physical objects have been connected by "virtual patchcords". A MIDI keyboard is shown driving an FM synthesizer (actually consisting of Katosizer software). Its carrier envelope may be changed with an envelope editor. Preset envelopes and other parameters may be selected by using a MIDI program-select button. The keyboard's pitch bend and modulation wheel are also connected to the synthesizer. The synthesizer output is fed into a distortion box controlled by a foot pedal. This in turn enters a "tape loop" whose loop gain is controlled by a slider, and whose input gain is set by another pedal. The result goes to the speaker.

Icons can be interrogated to gain access their internal settings or "control panels". For example, if the user double clicks on the distortion box icon, the control

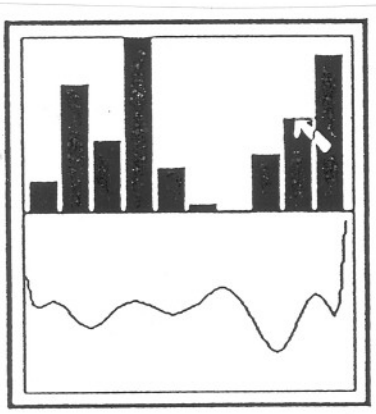


Figure 3.2: The control panel for a distortion box in the Katosizer.

panel for the distortion box appears in a window as shown in Figure 3.2. Using this technique the user can adjust the parameters associated with each device.

Connections (or "patch cords") between devices can be either audio signals or control signals (which could be MIDI). Furthermore, "patch cords" may exist between "soft" devices and physical devices. As an example, in Figure 3.1, a "soft" device, the emulated distortion box, is connected to a *physical* input device, the foot pedal.

The Katosizer was a research effort and therefore is not completely developed. Many more objects need to be added to the system. Many of these are unrelated to synthesis or processing, such as more sophisticated recorders, sequencers, sampled sound editing tools, etc. Nevertheless, the Katosizer project developed concepts and revealed issues that are relevant to this thesis. First, the concept of a user interface controlling the interconnection of devices was proven to be very effective. The fact that the implementation of a connection is transparent to a user is intrinsic to the success of the Katosizer; the user may create a connection between devices without being aware, or even caring, whether it is a hardware or software connection. Second, the way in which both graphical input devices and physical input devices are are used in the system helps in presenting a consistent, easy to use interface to the user. Finally, from experience it was observed that hierarchical encapsulation, albeit restricted to two levels, could also be used to collect a set of interconnected objects and represent them by a single icon. This provided the user with a way

to unclutter the screen; a major problem with the Katosizer patch cord diagrams was that their complexity was limited by screen space and the ability of the user to understand complex diagrams.

While the Katosizer can serve as personal audio studio, the interface does not address several critical problems outlined in chapter 2:

- No interface to facilitate task switching was developed.
- Interfaces to devices with numerous controls were not explored.
- Complex control slaving schemes were not explored.
- The complexity of device networks was limited by screen space.
- Information about which ports a cord connected was not available.

3.2 SoundDroid

The SoundDroid [21] [30] [2], a development of The Droid Works, is a DAW similar to the Katosizer, but oriented towards professional audio post production. A block diagram of the SoundDroid system is shown in Figure 3.3 . The basic SoundDroid station consists of two computers, one linked to a touch screen and other controls and another which controls a high speed Audio Signal Processor or ASP. The ASP controls several hard disk drives and one or more DSP boards which process digitized sound.

As described in [21], the SoundDroid, like the Katosizer, is capable of emulating many types of audio devices and configurations of audio devices. Programs can be loaded into the processors to transform SoundDroid into systems to perform a variety of tasks such as multitrack recording and mixing, editing of recordings, sound designing and effects production. Essentially the system emulates an 8 channel recording console and recorder, a stereo recorder and several effects devices.

The designers of SoundDroid also concentrated on developing the system to take advantage of storing digitized sound on hard drives. In a traditional multi-track

to unclutter the screen; a major problem with the Katosizer patch cord diagrams was that their complexity was limited by screen space and the ability of the user to understand complex diagrams.

While the Katosizer can serve as personal audio studio, the interface does not address several critical problems outlined in chapter 2:

- No interface to facilitate task switching was developed.
- Interfaces to devices with numerous controls were not explored.
- Complex control slaving schemes were not explored.
- The complexity of device networks was limited by screen space.
- Information about which ports a cord connected was not available.

3.2 SoundDroid

The SoundDroid [21] [30] [2], a development of The Droid Works, is a DAW similar to the Katosizer, but oriented towards professional audio post production. A block diagram of the SoundDroid system is shown in Figure 3.3 . The basic SoundDroid station consists of two computers, one linked to a touch screen and other controls and another which controls a high speed Audio Signal Processor or ASP. The ASP controls several hard disk drives and one or more DSP boards which process digitized sound.

As described in [21], the SoundDroid, like the Katosizer, is capable of emulating many types of audio devices and configurations of audio devices. Programs can be loaded into the processors to transform SoundDroid into systems to perform a variety of tasks such as multitrack recording and mixing, editing of recordings, sound designing and effects production. Essentially the system emulates an 8 channel recording console and recorder, a stereo recorder and several effects devices.

The designers of SoundDroid also concentrated on developing the system to take advantage of storing digitized sound on hard drives. In a traditional multi-track

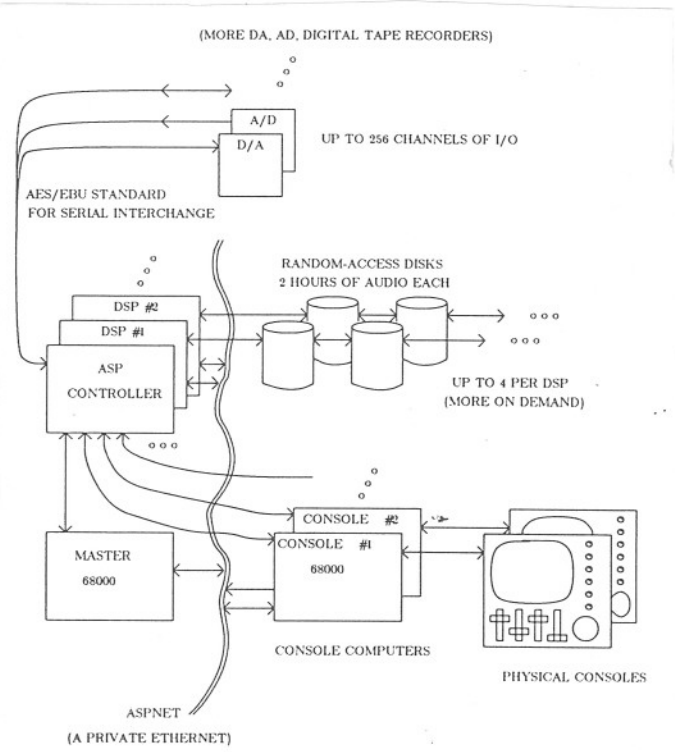


Figure 3.3: Block diagram of the SoundDroid. Figure from [21].

tape recorder, tracks are recorded in a linear fashion thus making changes to the synchronization of one track with respect to another quite difficult and random access difficult, if not impossible, because winding the tape to right spot takes time. The advantage in storing tracks of sound on hard drives is that fast random access of sounds can be supported and tracks, or portions of tracks, can be "slipped" in time to allow synchronization corrections or special effects.

The user interface to SoundDroid, shown in Figure 3.4, utilizes a high resolution monochrome touch sensitive monitor, 8 input sliders and buttons and a continuous knob. Depending on the program which SoundDroid is running, parameters in the system are controlled by the physical input devices and graphic controls which appear on the screen. Thus different control panels appear on the screen depending on what program the user is currently running.

What makes SoundDroid different from Virtual Studio and the Katosizer is that the designers of SoundDroid make no provision in the interface to allow the user to program the system. While Moorer claims in [21] that SoundDroid has "several

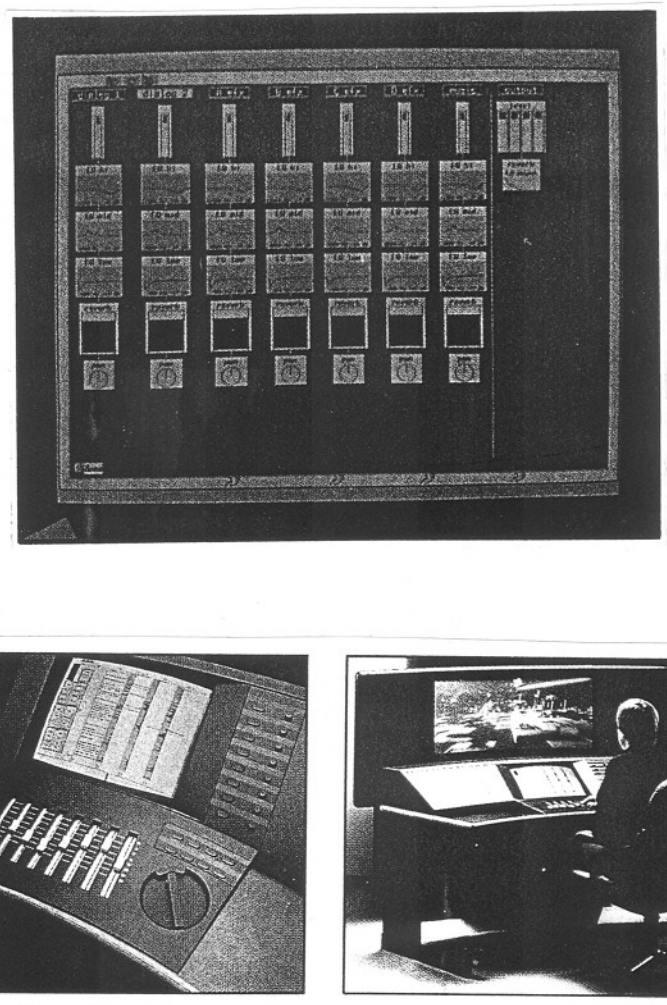


Figure 3.4: The interface to SoundDroid. Top: a sample of SoundDroid's "Meter Screen". Bottom: the console with input devices and video viewing screen. Figures from [2].

hundred different processing programs that can be freely intermixed" from which the user can choose, the user cannot customize the controls, or combine parts of what is supplied into new "consoles". Devices are pre-programmed by the system designers. Configurations of the system are in fact large programs which load into the system at the beginning of the task; there is no concept of a set of tools in which the user can quickly change from one tool to another or construct new tools. Graphical control panels and associations between audio device parameters and physical input devices are determined by the system designer not the user. Thus SoundDroid is lacking in task tailorability and user customizability.

3.3 CompuSonics DSP-2000

The CompuSonics DSP-2000 [29], a research effort of the Compusonics Corporation, is a multi-processor audio computer configured as a console for digital recording and mixing of live music. Figure 3.5 shows the hardware configuration used in the system. Similar to SoundDroid, the DSP-2000 has an array of DSP boards, corresponding DAC, ADC converters, a central controlling computer, hard disks for sound storage, a RGB monitor for display of control panels, a printer, and a keyboard and track ball array used for input.

The function of the DSP-2000 is similar to SoundDroid. What is different, and interesting to this thesis, is the interface. Figure 3.6 shows a schematic of the user interface for DSP-2000.

The essential idea is to have a "virtual console", rather than a traditional physical console to the control the system. The "virtual console" is piece of software that describes the entire mixing console and associated device control panels as the audio engineer wishes them to be, without any physical constraints in terms of size and accessibility. The number of channels, style of faders, color on EQ knobs, type and size of VU meters, layout of tape transport controls, etc are unrestricted.

The actual hardware that the user manipulates is a physical console consisting of an array of trackballs, a keyboard and a color monitor. Figure 3.6 shows a schematic

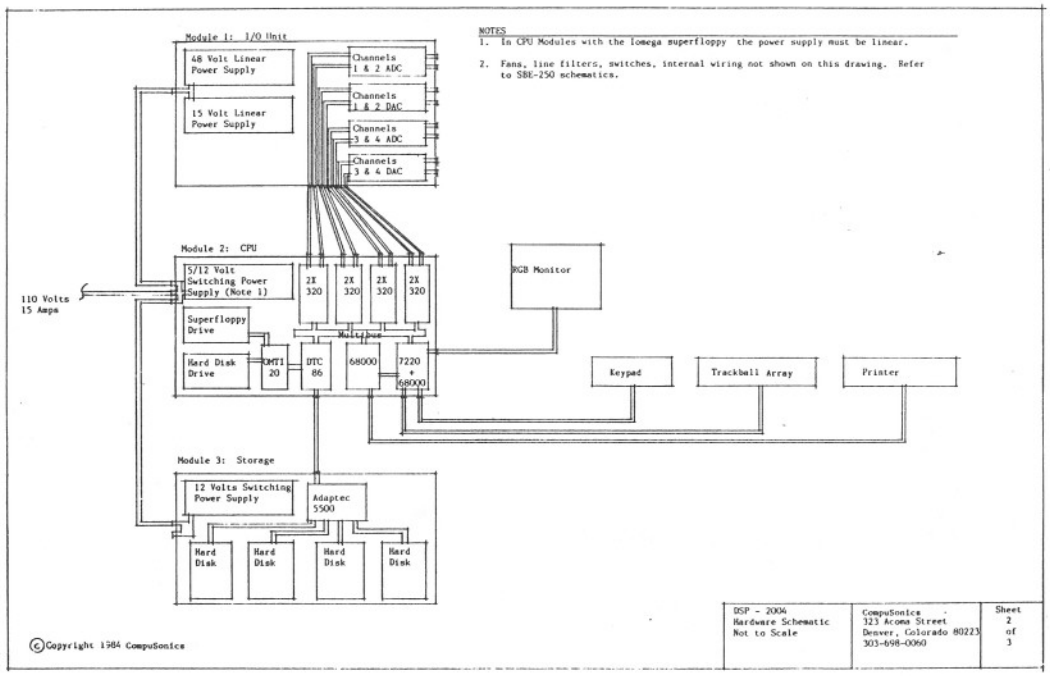


Figure 3.5: The hardware configuration of the Compusonics DSP-2000. Figure from [29].

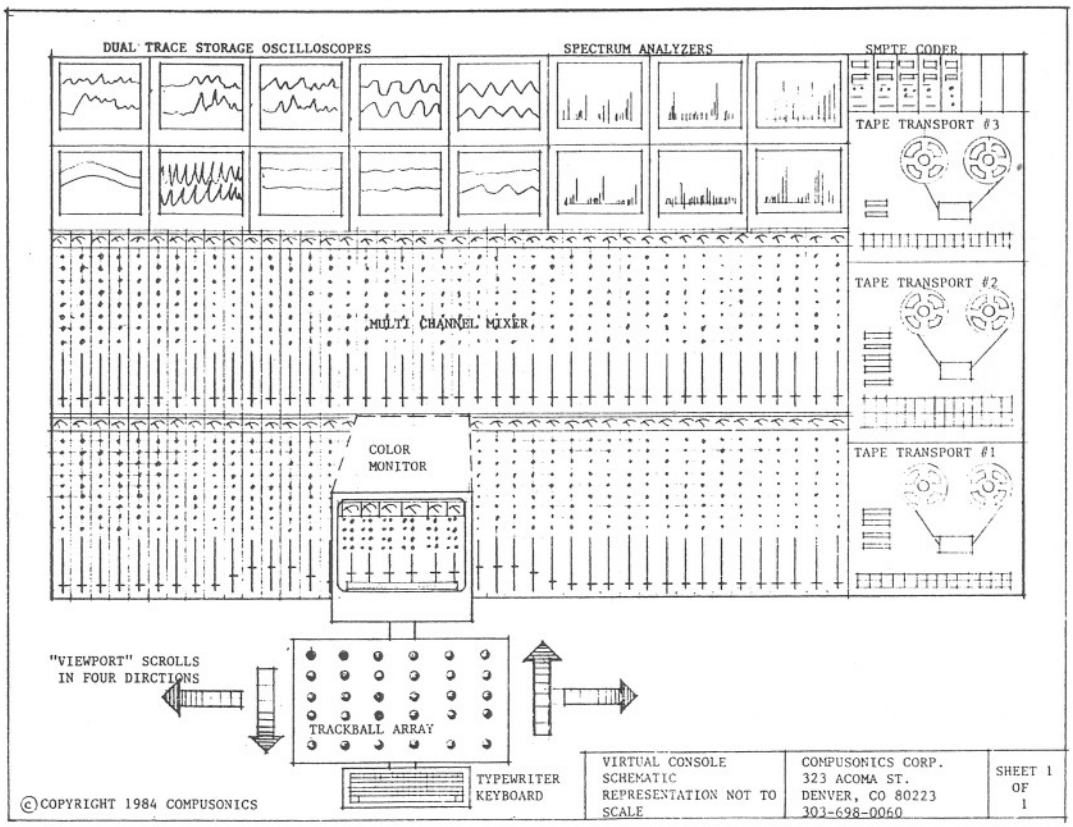


Figure 3.6: Schematics for Compusonics DSP-2000 user interface. Figure from [29].

of the interface. The color monitor is a "window" onto the virtual console. This window can be scrolled vertically and horizontally till the engineer views the desired portion of the virtual console. The graphic display is automatically coupled to the trackballs so that when a trackball is moved, the corresponding control is adjusted on the screen. The keyboard is used for textual input such as names of recordings, and for programming the system. The method of programming the system is by traditional construction of "C" programs.

The interesting aspects of the DSP-2000, relative to this thesis, is that Compusonics recognized (1) the value of graphical control panels over traditional audio console control panels and (2) the concept of virtual control panels, that is, being able to navigate from control panel to control panel rather that having a static display of controls.

The major disadvantage in this approach is the two dimensional navigation technique and lack of flexibility in the mapping between physical input devices and graphical controls. For example, suppose a user wishes to view two faders on the virtual console which are widely separated. This entails scrolling the "window" on the virtual console back and forth between the two faders. This interaction may be disturbing and unacceptable to the user who wishes to view both faders at once. Similar problems occur when the user wishes to simultaneously adjust these same two faders. Since the tracks balls are remapped to the controls displayed in the window, it is impossible to adjust these two faders at once.

Another limitation of the DSP-2000 is that, like SoundDroid, the system does not allow the user to customize the environment to suit the task. The only way to redesign control panels and the system configuration is by rewriting the "C" programs which implement the system.

3.4 DMP7-PRO

DMP7-PRO is a program for the Macintosh computer developed by Digital Music Services [10]. The components of this system consist of the DMP7-PRO program,

a Macintosh and a Yamaha DMP7 digital mixing console. The DMP7 is an eight channel to two channel mixing console where all audio signals are digitized on input and converted to analogue signals on output. Because all processing is done in digital format, the audio quality is of high standard and it was possible for the system engineers to design the mixer such that it could be controlled from an external computer. In addition, the mixer has two multi-effect processors. Thus, using MIDI, the computer can communicate with a DMP7, changing its controls remotely. Note that the DMP7 cannot be reprogrammed to perform any other audio task besides mixing. As described in chapter 2, the DMP7 front panel presents a minimal user interface. It takes a time-multiplexed modal approach where the same controls have different mode-dependent effects. The idea behind the program, DMP7-PRO, is to offer the user of a DMP7 an easy to use interface and extend the amount of control the user has.

Figure 3.7 shows the main screen display used by DMP7-PRO. Emulating a traditional mixing console, the software provides an uni-modal space multiplexing interface for the mixer. In other words, it represents the DMP7 with all its controls displayed simultaneously. The user can manipulate controls by direct manipulation using the mouse, but only one at a time. Of course, the tradeoff is that you now have to time-multiplex the mouse to exercise control over different faders, whereas on the DMP7 itself, each mode is "space" multiplexed with a number of controls.

There are several aspects of DMP7-PRO that are of interest to this thesis. For several types of controls there are representations available other than the "front panel" view. Figure 3.8 shows an alternate representation of a channel equalization control. DMP7-PRO also supports user customizability features.

Figure 3.9 shows the various tools that can be placed anywhere on the screen for the user's convenience. Thus, in a limited sense, the user can tailor DMP7-PRO to suit the task being performed.

DMP7-PRO also provides the user with more extensive control over the DMP7. While most mixing and automation systems provide grouping of input faders, DMP7-PRO allows any set of parameters in the DMP7 to be grouped together

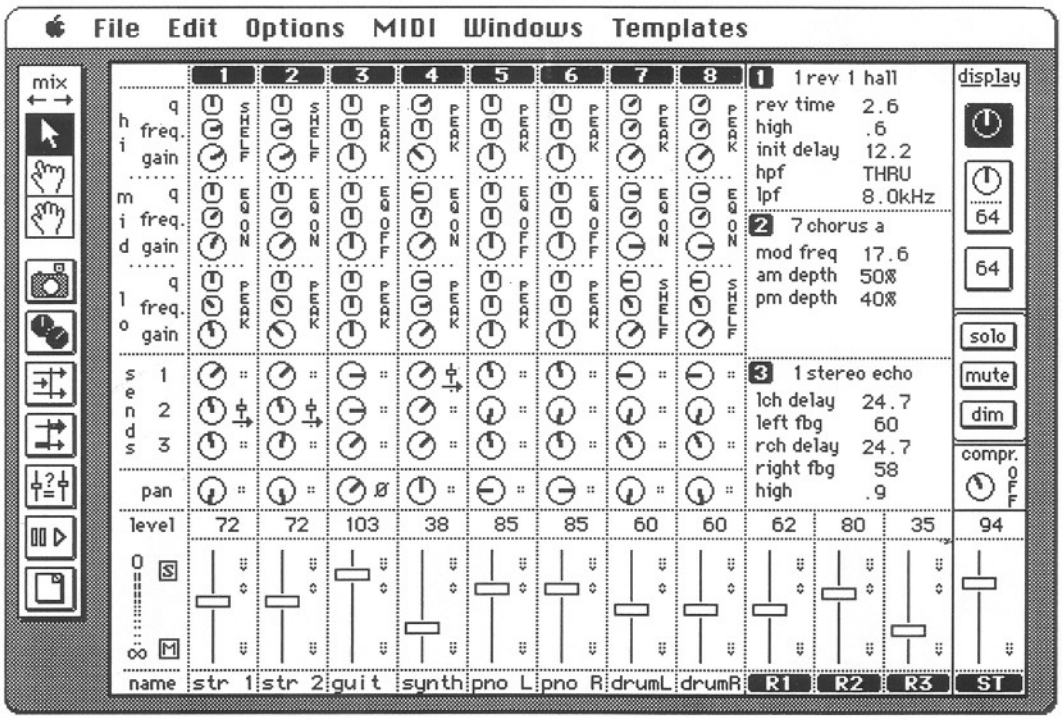


Figure 3.7: The main screen display used in DMP7-PRO.

and slaved, temporarily, to the mouse. For example, a single mouse movement can cause several channels to fade while others are panning, while others are changing their effects level sends, while others are changing their EQ settings. The groupings can be saved and recalled in an instant, thus greatly extending the amount of control the user has when compared to the real front panel.

The DMP7-PRO gives us a good example of the effectiveness of some of the motivating concepts described in chapter 2. Alternate representations, user customizability, and extended control are features present in DMP7-PRO because they were deemed valuable by users and designers. The difference between Virtual Studio and DMP7-PRO is that DMP7-PRO is designed for a single specific device; Virtual Studio is a much more general approach. We wish to develop a system in which interfaces for other devices, besides the DMP7, can be constructed by the user and used simultaneously. Furthermore, the configuration of these devices can be specified down to the level that if it was micro-programmable, one could use a consistent technique right down to the level of designing the functionality of some-

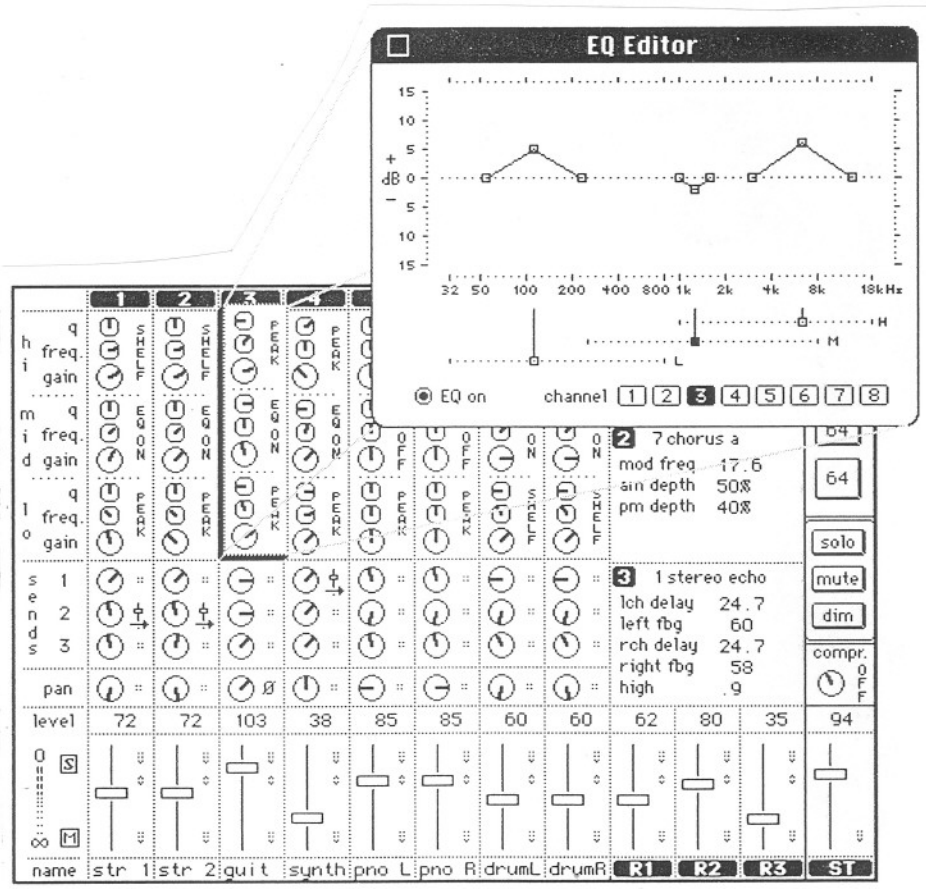


Figure 3.8: DMP7-PRO: by double clicking on an EQ control a more detailed editing window appears.

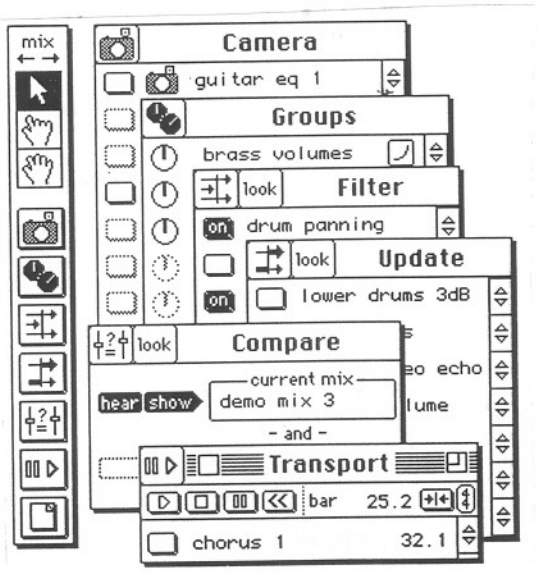


Figure 3.9: Various tools in DMP7-PRO that can be placed by the user anywhere on the screen for their convenience.

thing like the DMP7 itself, rather than just its controls. Kitamura, Galloway, etc [11] [19], have demonstrated how this technique can be extended to the design of digital filters. Our objective is to pursue this direction.

3.5 Q-Sheet

Q-Sheet, a program for the Macintosh, is a MIDI event sequencer and automation system [26]. Q-Sheet is used in conjunction with any MIDI controllable device to automate control over it. By using Q-Sheet, the Macintosh can, by generating MIDI events, "play" synthesizers and samplers, and control sound processing devices. MIDI events can be recorded and stored in "tracks" in a manner similar to multi-track tape recorder. Events can be added and removed from tracks and have the time of their occurence changed. In effect, Q-Sheet allows the user to assemble a carefully edited sequence of MIDI events and execute this sequence. Q-Sheet is intended as a tool for audio post-production and sounds effects assembly.

Figure 3.10 shows the interface supported by Q-Sheet. The window called

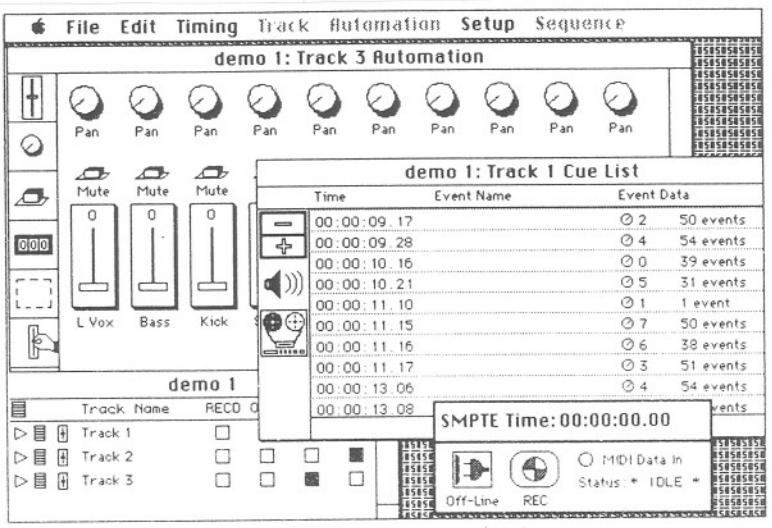


Figure 3.10: A typical screen display in Q-Sheet.

"SMPTE Time" displays information about the system's master clock using standard time code, which synchronizes all events. In the lower left hand corner is a window which displays the current modes of the tracks (record, play, muted, etc). The window called "demo1: Track 1 Cue List" is a type of window that can be displayed for each track and displays the events recorded in that track. The large window with the faders and knobs is a window, associated with each track, in which the graphical devices are used to trigger MIDI events. These events are then recorded in the corresponding track. This window type is called an "automation window".

Essentially, Q-Sheet allows the user to build control panels in automation windows. The menu on the left of the automation window allows the user to create and drag new instances of faders, knobs and numerical controls onto an automation window. Once the graphical device has been positioned, the user can double click on the device and a dialogue box is popped up. Within this dialogue box the user can specify the semantics of the device. That is, the user can specify what MIDI events are generated when the device is adjusted. In effect, the user can construct custom controls panels to suit their own needs.

It is this feature that makes Q-Sheet relevant to this thesis. Chapter 2 described

our motivation in giving the user the capability to customize their environment. User construction of control panels is one method by which the user can tailor the system to their needs. Q-Sheet is a working example of this concept. Unfortunately the type of control panels that can be constructed by Q-Sheet are not very sophisticated. Only a few types of graphical controls are available and no group slaving scheme is supported. Furthermore, as control panels grow large (for example: a 16 channel mixer's control panel), the only methods for dealing with numerous controls are (1) use a control panel from another track or (2) layout a control panel which is larger than the automation window and scroll the window. Solution (1) is inconsistent with the user's conceptual model of the system: a control panel is associated with a single track. If solution (2) is taken, the same problems with windows on large virtual control panels, as described concerning the Compusonics DSP-2000, occur.

3.6 Summary

This chapter has presented several systems which are either similar to or have features relevant to Virtual Studio. For each of these systems the concepts that contributed to this thesis were identified and relevant problems and features were discussed. While many useful concepts appear in these systems, no one system implements all them, thus providing further motivation for the research in this thesis.

Chapter 4

The Case Study System: Virtual Studio

In this chapter we describe and discuss the system developed to explore the problems presented in the previous chapters. First a general description of the design of Virtual Studio and our approach to interaction pragmatics is presented. Next, our description focuses on the configuration process, user navigation and studio customization features, and how Virtual Studio aids the user in controlling devices. Finally, we take a step back from these descriptions to discuss what we have observed and learned from the case study.

4.1 General Approach

Virtual Studio is based on the interface developed for the Katosizer [3]. Figure 4.1 shows a typical screen. The large window is a type of window referred to as a configuration window. A configuration window displays a diagram of the data flow in the system. We have adopted the Katosizer technique of having devices represented by icons and the connections between these devices represented by arcs between them. Similar to the Katosizer, when a user clicks on an icon, a control panel window for the device appears. Figure 4.2 shows the control panel for an equalizer object.

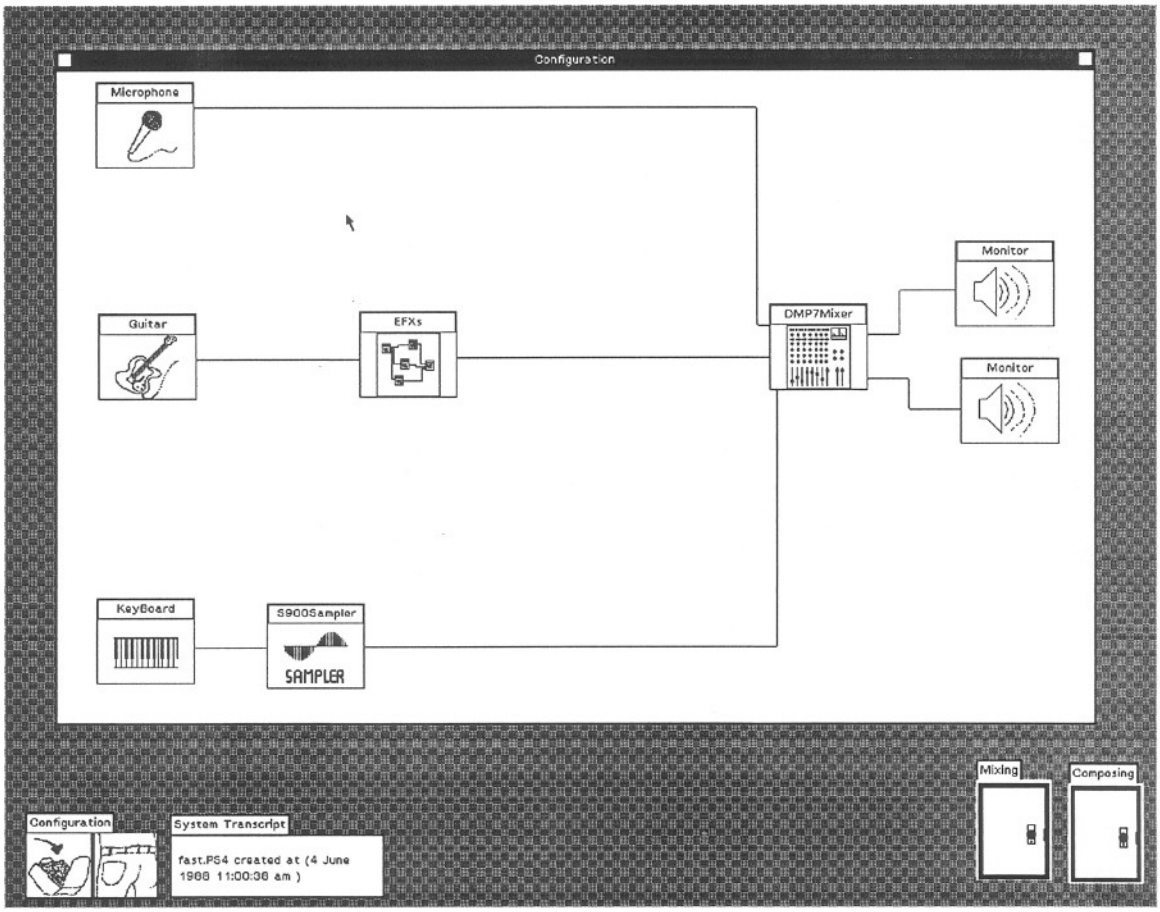


Figure 4.1: A typical screen in Virtual Studio.

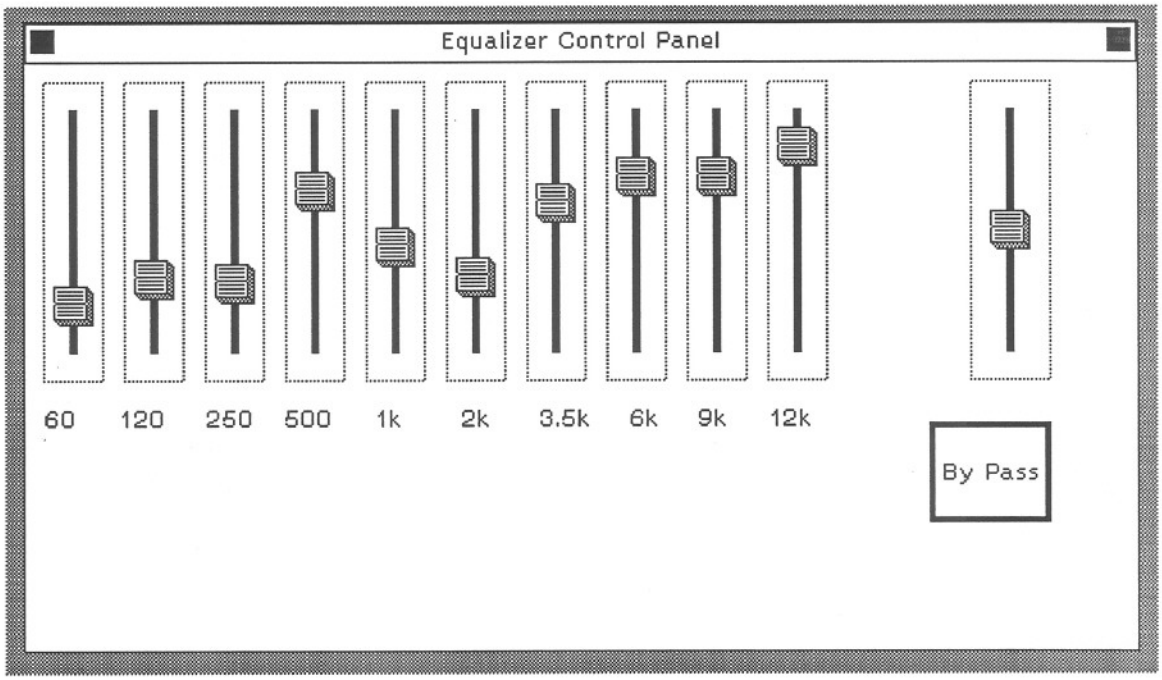


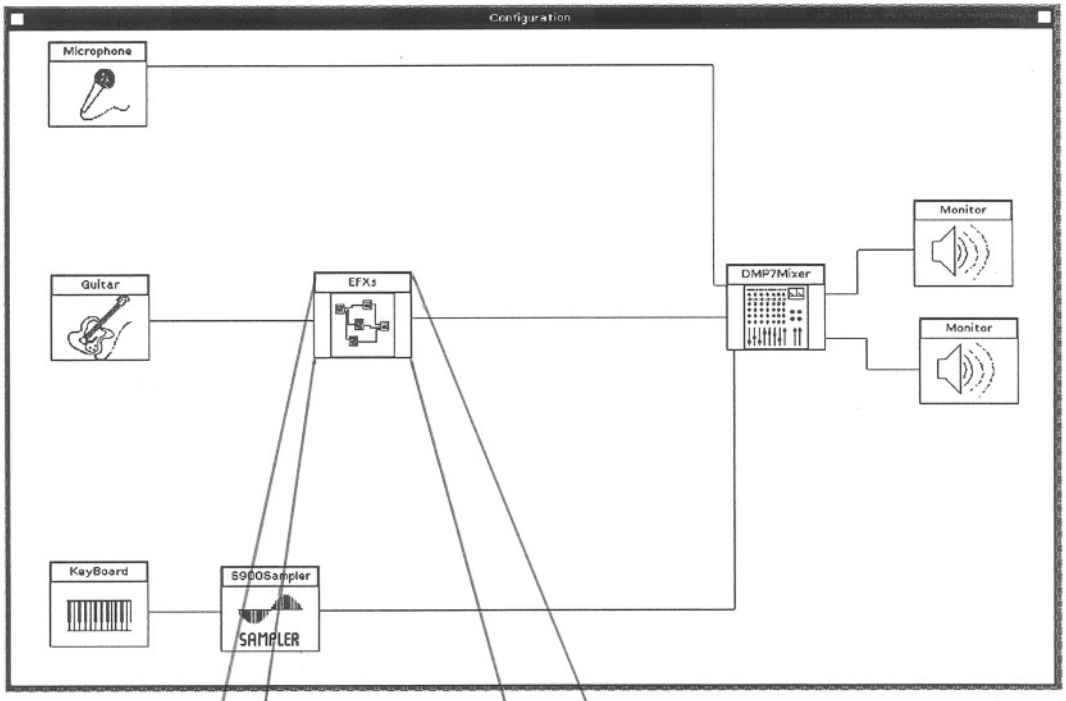
Figure 4.2: A graphical control panel for an equalizer.

The major difference between the Katosizer data flow diagrams and Virtual Studio's data flow diagrams is that Virtual Studio diagrams have hierarchical structure. For example *double* clicking on a device icon causes a configuration window to be displayed. This configuration window contains a data flow diagram revealing more about the internal workings of the device. Figure 4.3 shows the result of double clicking on the device called EFXs. The new window reveals what devices are used in the internal function of EFX.

The user may interactively edit an object's configuration window. Devices are added to a configuration window (hence to the total configuration) by requesting an instance of a device type from a resource manager. If an instance of the device is available, for example, the physical or logical resources are available, the resource manager allocates it for use. Deleting a device from configuration window causes it to be returned to the resource manager.

Devices have input/output ports or "jacks" into which connections or "cables" can be plugged. When a cable is plugged between two jacks, two constraints must be satisfied before the resource manager creates the connection: (1) is the connection

Before



After

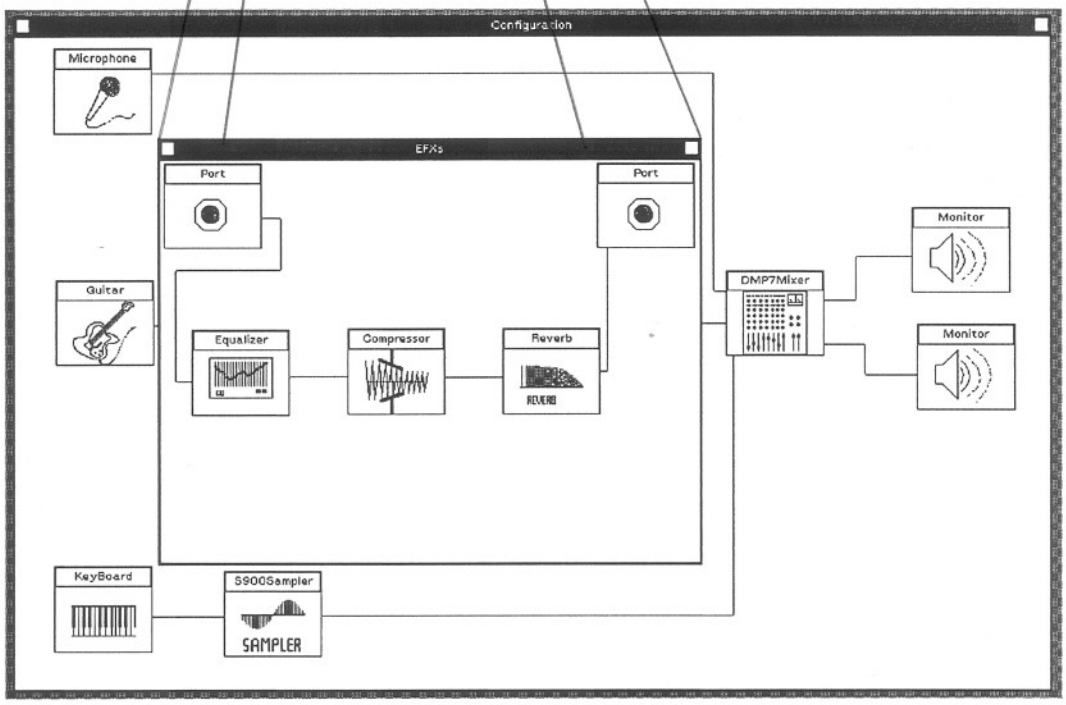


Figure 4.3: Clicking on the EFXs module causes a window to be displayed which reveals the internals of the EFXs module.

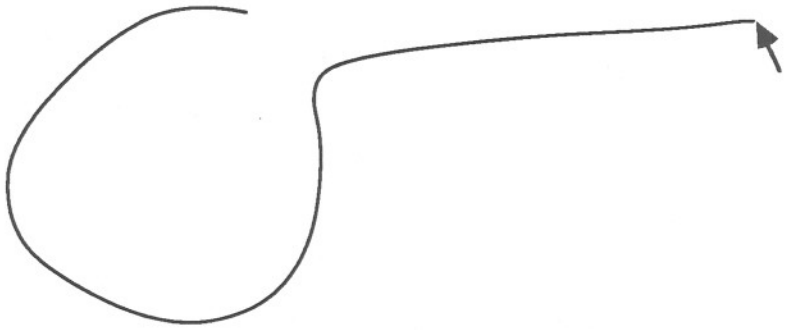


Figure 4.4: The pen and paper metaphor: when the user depresses the mouse button and moves the mouse, an ink trail is left by the cursor.

physically possible? and (2) does the connection make sense versus application constraints. As an example of the type of constraint specified by (2), a connection from an input to an input may be physically possible, but application-wise makes no sense.

4.2 Pragmatics of Interaction

Virtual Studio utilizes gestures as a means of interaction. The style in which gestures are used is metaphorical to pen and paper. An example is given in Figure 4.4. The mouse is used as the pencil; moving the mouse, when a mouse button is pressed, leaves an ink trail which represents the user's gesture. Releasing the mouse button terminates the gesture.

Other systems have used gestures as a means of interaction, but the gesture technique used in Virtual Studio is unique in several aspects and represents a refinement of the gesture interaction technique.

Unlike the system developed by Hosaka and Kimura [17], gestures in Virtual Studio are recognized interactively. The command associated with a gesture is carried out immediately after the termination of the gesture. In contrast, the Hosaka and Kimura system allows the user hand sketch an engineering diagram, then input this hand sketch a into batch program which performs recognition on the hand sketch and outputs a cleaned up and properly annotated diagram. Figure 4.5 shows

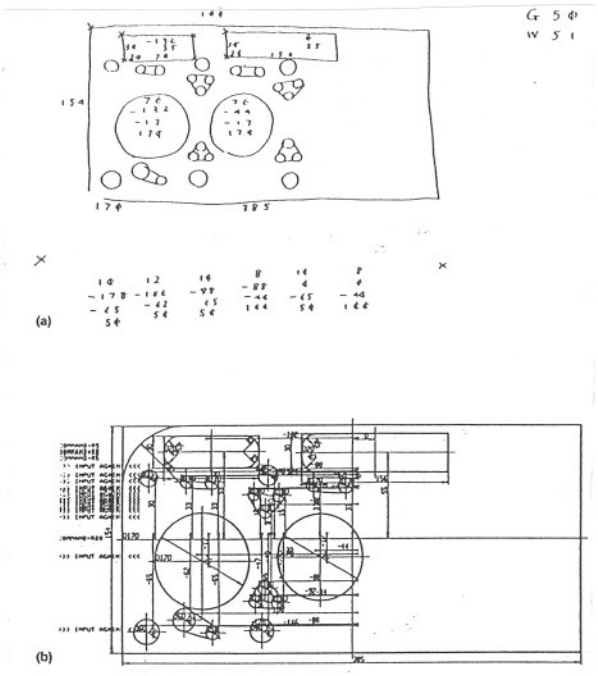


Figure 4.5: Use of hand written drawings for the definition of objects in CAD: (a) handwritten drawing is input to the system to define an object. (b) the system "cleans up" the handwritten drawing and redraws it. Figure from [17].

an example of this process.

Gestures in Virtual Studio are not used to roughly draw icons. In a system developed by Hornbuckle, an interactive flow chart drawing program, the user adds flow chart icons by roughly drawing an outline of the icon using a stylus and tablet. The outline of the icon is then recognized by the system and a "real" icon replaces the outline. We have avoided this approach for three reasons: (1) the icons in our diagrams are not of unique shape (they are all square), therefore a set of gesture symbols based on icon outlines was not possible, (2) we allow the user to define new devices, therefore new device icons. Thus, if the gesture symbols were used for adding new devices, the user would have to specify new gesture symbols for new device icons. Since designing a set of non-ambiguous gestures symbols is even difficult for designers of systems that use gesture recognition [32], it was felt this was beyond the ability and patience of the average user and (3) our input device is a

mouse and therefore drawing small outlines of icons would be tedious and difficult.

What makes the gesture technique used in Virtual Studio unique from these other systems is the application of the notion of "tension". This notion is defined and the advantages of its application are described by Buxton in [5]. Buxton observes that a gesture is preceded by a state of muscular neutrality, followed by the performance of the gesture and upon completion or closure is followed by a neutral state. The performance of the gesture involves a short period of muscular movement or tension and there is evidence that such periods of tension are accompanied by a heightened state of attentiveness and improved performance. Buxton conjectures that tension can be used as a "glue" to hold together the sub-tasks which must be performed in the specification of a command. Gluing sub-tasks together results in reduced user error in command syntax and therefore better performance.

In Virtual Studio, we have adopted the use of tension to improve user performance. Certain commands, which are composed of several sub-tasks, can be invoked by a single gesture. This approach differs from approaches used in other gesture based systems. In a gesture driven system described by Konneker in [20], a gesture is used to invoke a command without explicit initiation or closure. In other words, gestures are drawn in a continuous trace without the use of buttons; the system is constantly scanning for a gesture corresponding to a command. Rhyne and Wolf report [28] on a prototype spread-sheet program which uses gestures for editing. In this system, a single gesture may not always correspond to a complete command (for example, a "+" gesture, used to add two columns of numbers, actually consists of two small gestures). The inherent disadvantage in both these approaches is the ambiguity in determining when the gesture-command is complete. In the Konneker approach, the user can never be entirely sure as to when the gesture/command has begun or ended. In contrast, the Rhyne and Wolf system, in certain cases, can never be sure when the command is complete; in attempt to overcome this problem, the system waits for pause in user input to determine if the command complete. This of course produces a processing delay for all commands and forces the user to pause between two gestures which may possibly be interpreted as a single command.

In Virtual Studio, by using the tension involved in a single gesture (the depressing of a mouse button, the movement of the mouse, and the release of the button), the initiation and closure of commands is obvious to both the user and the system.

It is also important to note that gestures in Virtual Studio are integrated with other interaction techniques such as direct manipulation, menus and textual input. We have not attempted to have a gesture for every function of the system; we have attempted to use gestures where they are well suited to the function. Section 4.6 discusses the integration of gestures with other interaction techniques in further detail.

4.3 Managing Configurations

As outlined in section 4.1, the user configures the system by editing icons and the connections among them in a configuration window. Essentially the user "draws" a diagram of the desired configuration. As each portion of the drawing is created, the hardware is automatically programmed. We now describe how this "drawing" is created and edited.

4.3.1 Adding and Deleting Devices

The interaction involved in adding a device to a configuration is menu based. It can be initiated by pressing the middle or right mouse button over an empty spot in a configuration window. A menu of all devices that can be added to the configuration window is displayed. Once a selection is made, an icon for the device appears at the location the cursor was at when the command was initiated. Thus the add command has two arguments. The first is the location of the new icon and the second is its type.

The menu displayed when adding an object is actually a list of available device resources constructed by a software entity called the Resource Manager. Resources are subject to physical limits of the system and the Resource Manager is responsible for keeping track of the number of devices of a given type that are available or

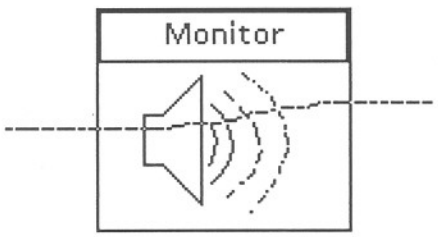


Figure 4.6: Deleting a device from a configuration by drawing a "scratch" through it.

allocated for use. The menu displayed in adding a device is actually a list of device types that are available for use in the current configuration. A selection from the menu causes the Resource Manager to allocate an instance of the selected type of device for use in the current configuration.

A device may be deleted from a diagram by drawing through it a horizontal "scratch it" gesture. Figure 4.6 shows a monitor device being deleted. In this case, the icon and any incident connections are automatically deleted from the diagram. In terms of the underlying semantics, deletion of a device from a configuration returns the allocated device to the Resource Manager and causes the hardware connections to the device to be broken.

Devices may also be deleted in groups by a gesture similar to the international "not allowed" gesture. Figure 4.7 shows an example of this gesture.

4.3.2 Adding and Deleting Connections

A connection between two devices can be made by simply "drawing" the connection using a gesture. The left and right sides of a device icon are designed to be hot spots; the left hot spot corresponds to the input jacks for a device and the right hot spot corresponds to its output jacks. To initiate drawing a connection, the user presses the mouse button over a hot spot, then gestures the path they would like the connection to follow and the other device's hot spot they wish to connect to. Figure 4.8 shows this process. If the connection is valid (for example, a connection

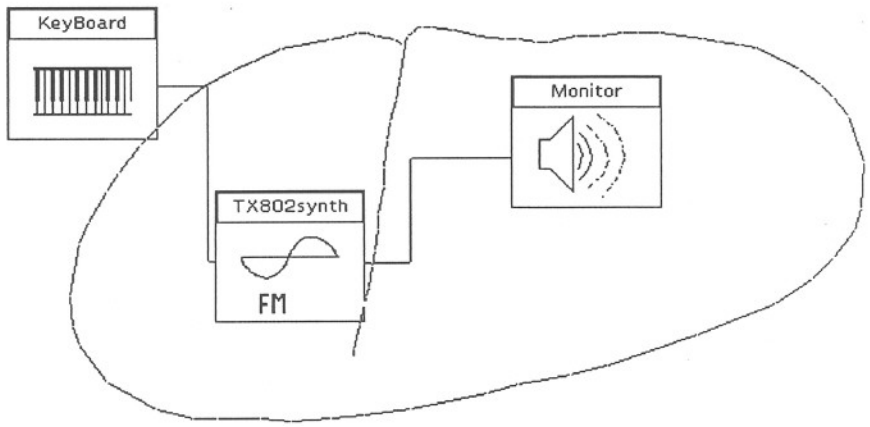
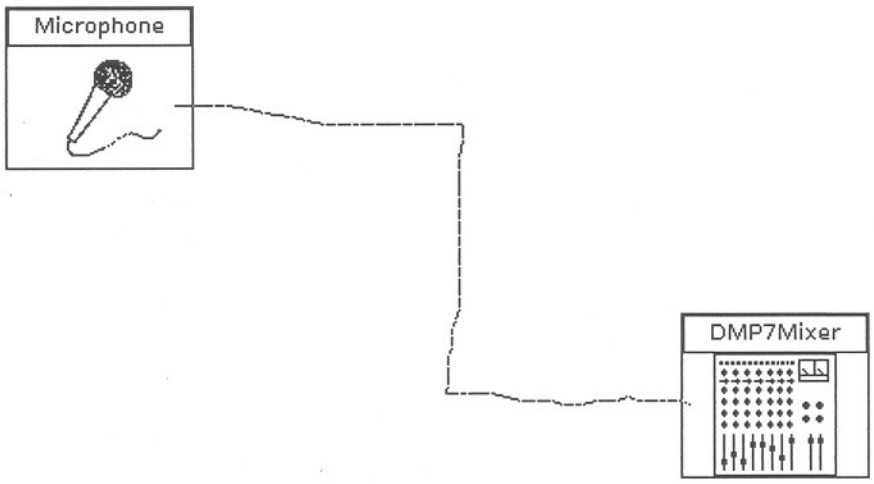


Figure 4.7: Deleting several devices at once: the user circles the devices and finishes the circle with a stroke through it. The gesture is metaphorical to the international "not allowed" symbol.

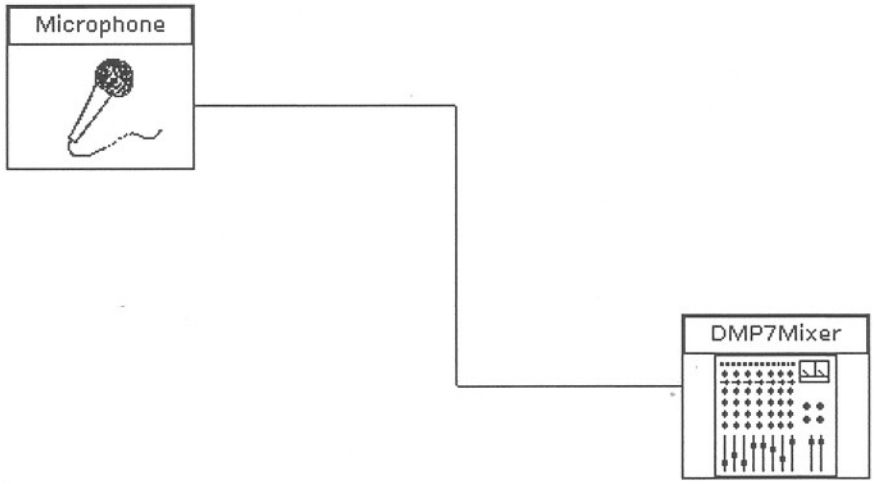
drawn from an output hotspot to another output hotspot implies an output jack to output jack connection and is therefore not valid), the system then erases the hand drawn connection and replaces it with a Manhattan version of the gesture (the Manhattan process will be described in detail later).

Note that the exact pair of jacks to be connected has not been specified by the user. In effort to answer this question, the system displays a table of possible connections that the gesture implies. The user can then select the exact jacks to be connected. Figure 4.9 shows this process. If there is only one possible connection (for example, the source device has only one output jack and the sink device has only one input jack), no table is displayed and the connection is immediately made by the system.

The table displayed during the connection interaction is a mechanism which allows the user to specify which jacks the connection runs between and also provides a display which describes the state of connections between two devices. A table is constructed as follows: the jacks represented by the hot spot picked first correspond to the rows of the table. The jacks represented by the hot spot picked last correspond to the columns of the table. The i,j element in the table corresponds to a possible



a)



b)

Figure 4.8: Making a connection: in a) the user "draws the cable"; in b) the system replaces the hand drawn cable with a cleaned up version.

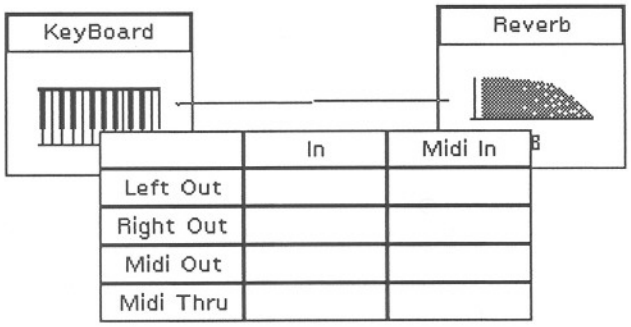


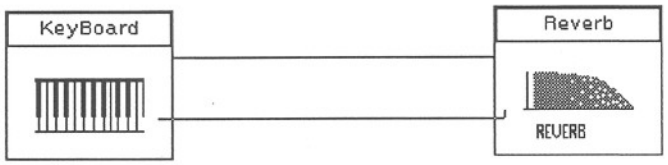
Figure 4.9: Specifying the details of a connection: a table is displayed showing all possible connections between the two devices given the endpoints of the cable. The user can select the exact jacks to be connected.

connection between row i's jack and column j's jack. If an connection does exist, the word "Connected" appears in the element. If no connection currently exists, the element is blank. Figure 4.10 shows a table in which a connection exists.

When used to specify a connection, the user interacts with a table in a manner similar to a traditional pop-up menu. While a mouse button is pressed the user may move the cursor from element to element. Similar to a menu, the elements are displayed in inverse video when the cursor is within them. Releasing the mouse button in an element selects the element and indicates to the system the connection just drawn represents a connection between the two jacks associated with the row and column of the element. Releasing the mouse button outside of any element causes the connection transaction to be aborted.

When the user gestures a connection, they are also indicating graphical information about the connection. That is, the user gestures the path of connection in the configuration diagram. In order to improve the appearance of the gestured connection it is replaced by a Manhattanized version. The Manhattanizing process maps the gesture to a 2D grid where only vertical or horizontal lines are allowed. Figure 4.8 gives an example of a Manhattanized connection.

Connections may be deleted in the same manner as icons are deleted. The user



	In	Midi In
Left Out		
Right Out		
Midi Out		
Midi Thru		Connected

Figure 4.10: An example of a connection table which indicates an connection between the KeyBoard's Midi Thru and the Reverb's Midi In jacks already exists.

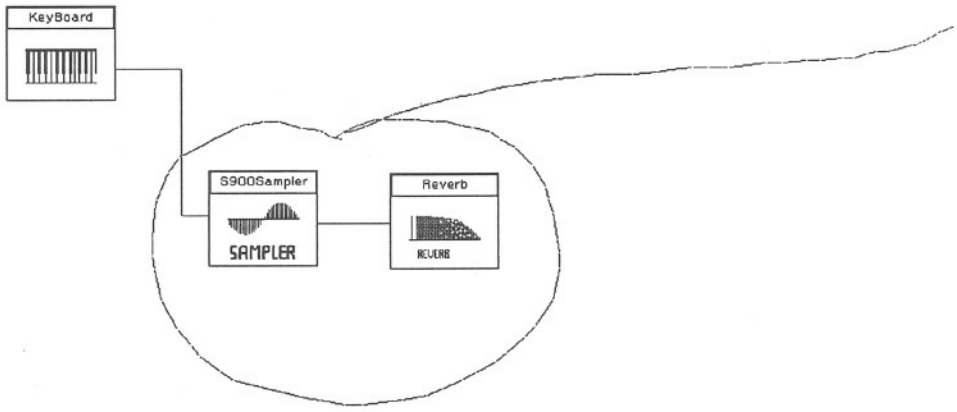
can make a horizontal "scratch out" gesture through some portion of the connection to delete it or the user may circle a group of connections using the "not allowed" gesture in order to delete several connections at once.

The connection interaction is discussed further in Section 4.7.1.

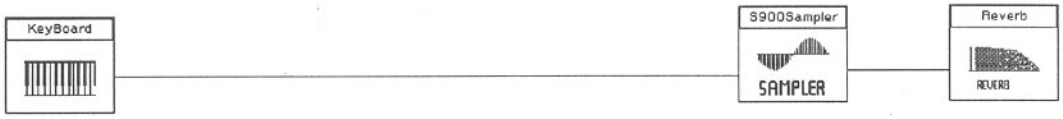
4.3.3 Layout Commands

Layout commands operate only on the graphical layout of a configuration diagram and do not affect the hardware configuration. The only layout command supported in Virtual Studio is the icon move command which comes in two flavors: move one icon at a time or move a group of icons. Icons can be moved one at time by pressing a mouse button down over the label bar of an icon and dragging it to a new location. The interaction is similar to the icon dragging technique utilized in the Apple Macintosh with the exception being that icons in Virtual Studio may have connections. Connections are not "dragged" along while an icon is being moved, but are "moved" after the icon is dragged to its final location.

Icons can also be moved in groups. A gesture, as shown in Figure 4.11, can be used to chunk icons into a group and indicate a new location for the group. Unlike moving single icons, groups of icons are not dragged, but moved in one jump to the new location. Once the icons are moved, their connections are automatically



a)



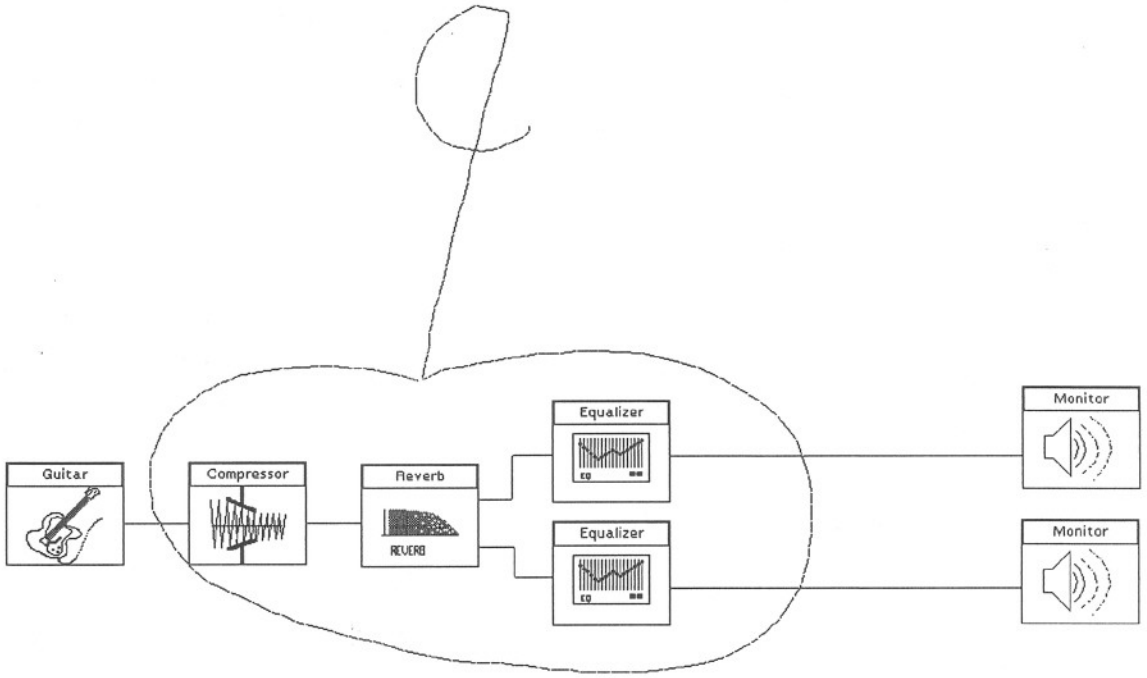
b)

Figure 4.11: Moving groups of devices: in a) the user gestures the devices to be move and where to move them; in b) the system responds by moving them.

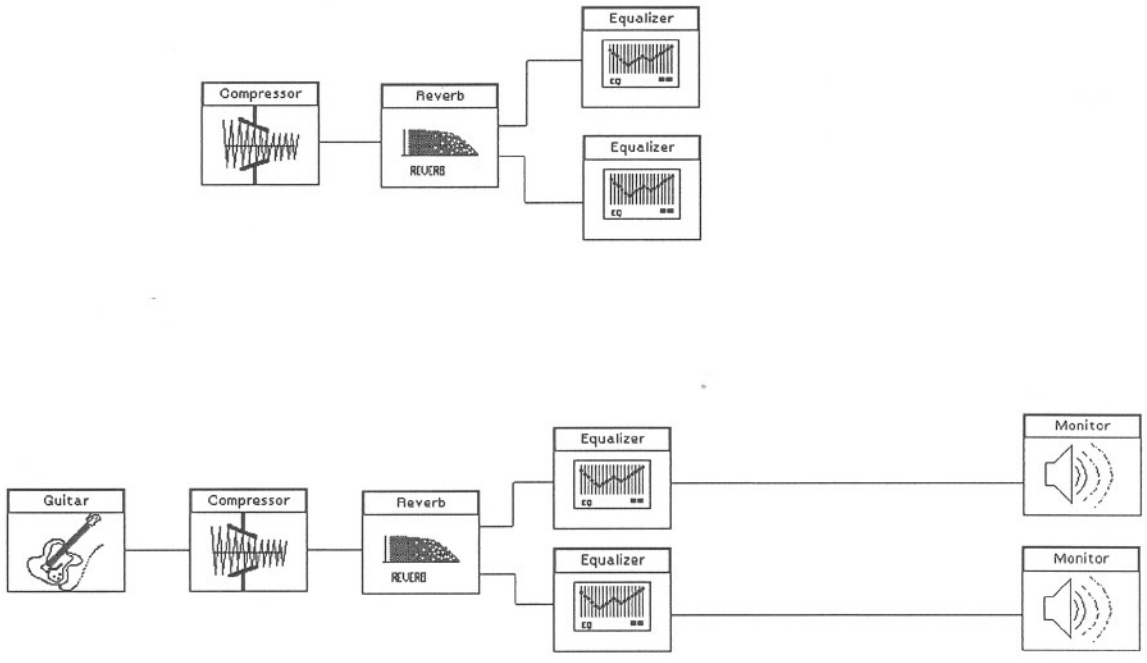
rerouted.

4.3.4 Copying

Using the copy command, chunks of configuration can be duplicated. The interaction required to invoke the command is identical to the group move command except the "copy" gesture terminates with a "C". An example of the copy gesture is shown in Figure 4.12. Using the circled chunk of the configuration as an example, the system allocates, connects and lays out another instance of this chunk. Note that inherent in the semantics of the copy command is the rule that connections to

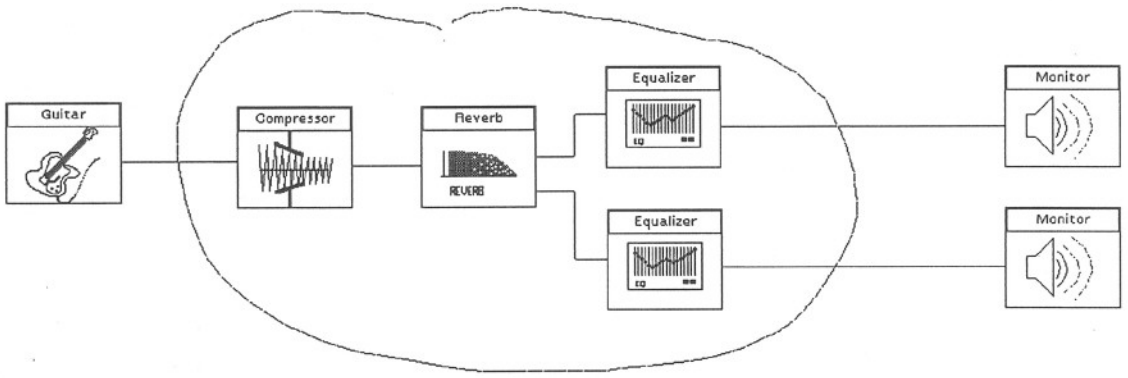


a)

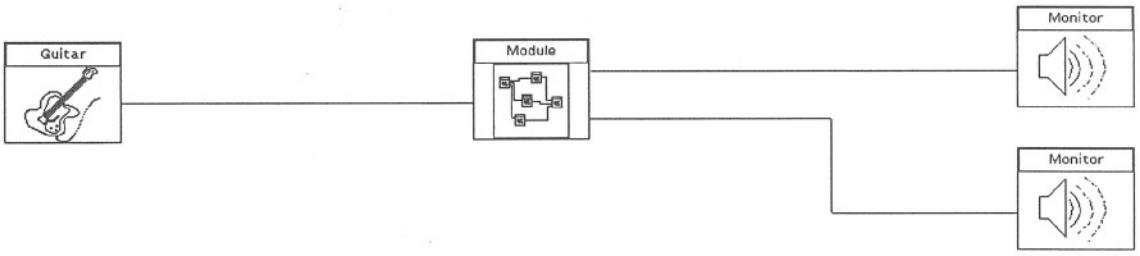


b)

Figure 4.12: Copying groups of devices: in a) the user gestures the devices to be copied and where to copy them; in b) the system responds by copying them.



a)



b)

Figure 4.13: Encapsulation: in a) the user gestures the devices to be be encapsulated; in b) the system responds by shrinking them into a module.

devices external to the chunk are not copied.

4.3.5 Encapsulating

Chunks of a configuration may be encapsulated into module. As shown in Figure 4.13, by circling a group of icons the user indicates to the system that a chunk of the configuration should be collapsed into a module. The chunk of icons and connections are replaced by a module icon and the connections from the chunk to devices outside the chunk are replaced by connections from the module to the external devices. The system automatically creates a module with input and output

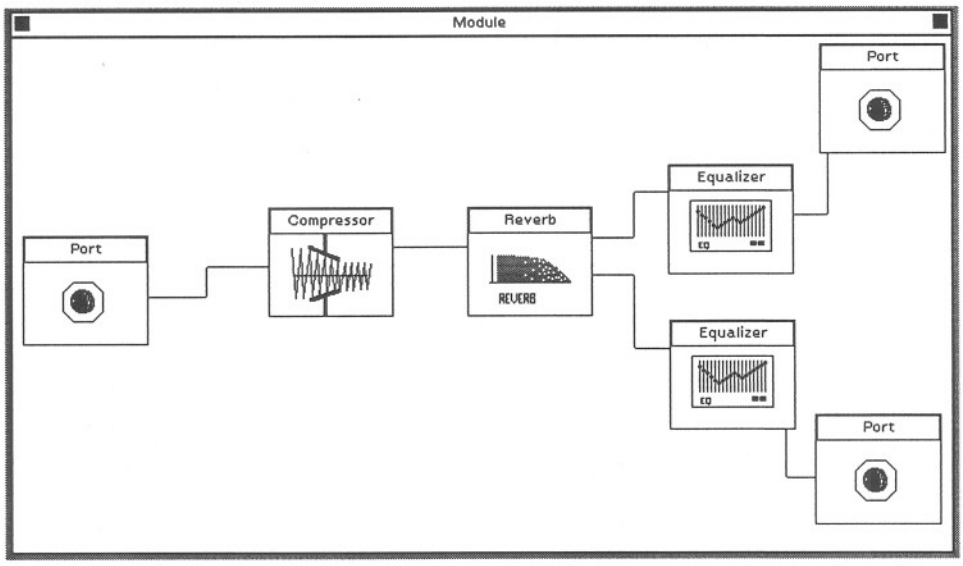


Figure 4.14: The internal configuration of the module produced by encapsulation in Figure 4.13.

jacks according to the external connections of the chunk.

The chunk of icons and connections become the internal configuration of the module icon. By double clicking on a module icon the internal configuration is revealed in a configuration window. Figure 4.14 shows the internal configuration of the module created in Figure 4.13. The internal configuration of the module has the same layout and devices as the chunk, with exception of external ports. The external port icons represent the input/output jacks of the module.

4.3.6 Saving and Restoring

Virtual Studio can save and restore configurations in a manner similar to the way a text editor handles files. By moving over a title bar of a configuration window a menu can be popped up which contains the "save" and "restore" commands. Selection of the save command causes Virtual Studio to make a copy of the state of the configuration and store it in a global list of configurations. The configuration is stored under the name given in the title bar of the configuration window. Selection of the restore command causes the system to be reconfigured to the state it was in at the time of the most recent save command. The configuration window is also

updated to reflect the changes in the configuration.

Since we have a method for saving configurations in a list, a tool must be available for the user to gain access to this list and browse through the configurations. A configuration browser is such a tool. The browser is a window which contains a list of names of the different configurations in systems. When a name is selected, the browser displays the root configuration window of a configuration. Since the user is only browsing configurations, the displayed configuration is not applied to the hardware. Only if the user selects the restore command from title bar of the configuration window will the configuration be applied to the hardware. Thus, using the browser, the user can search for a desired configuration, preview it without altering the current configuration of the system, and if they desire, automatically reconfigure the entire system by restoring the previewed configuration to the hardware.

4.4 Information Hiding in Configurations

In Chapters 1 and 2 we presented the advantages of using hierarchical structure to hide information associated with a configuration. The application of this technique involves the user deliberately hiding information about certain chunks of the configuration in modules. In Virtual Studio, we are also concerned with another type of information hiding which involves connections between objects. Anyone who has dealt with complex audio systems generally practices, although not consciously, a methodology in connecting components; first the AC power connections are made, next, perhaps the MIDI cables are plugged in and, finally, the audio connections are made. Although different people may follow different methodologies, the point is that in effort to reduce the complexity of the task, the user likes to group the connection of cables into tasks based on the type of connection. Furthermore, in trouble-shooting or trying to understand a configuration, the user once again applies this methodology. For example, typically when some component is not functioning correctly, the user will first check the AC connections, then the audio connections and finally the MIDI connections.

In Virtual Studio we have attempted to apply this methodology and have designed configuration windows such that the display of objects can be *filtered.* For example, if the user is viewing a configuration and has the "MIDI" filter on, only the MIDI connections are shown. Alternatively, the user may have the "audio" filter on and see only audio connections.

The concept of a filtered view can be generalized such that the user has the ability to filter his view of the configuration using arbitrary constraints. For example, the user may want to view the configuration without viewing the mixer icon and all its connections, or view only the devices connected directly to another device. In order to design a system which features this type of constraint based display, two problems must be dealt with: (1) what sort of underlying software structures are needed and (2) what sort of user interface can be designed so the user can easily express constraints. In terms of problem (2), Virtual Studio supports the expression of popular constraints by having a menu item which represents "show all devices and MIDI connections" (the MIDI filter) and "show all devices and audio connections" (the audio filter). Currently there is no way of expressing arbitrary constraints and future work could examine this problem.

Although the interface currently does not support user expression of arbitrary display constraints, we have designed mechanisms in the underlying data structures for configurations and configuration windows such that arbitrary display constraints can be supported. Furthermore, because of the generality of these mechanisms, they are used to implement the MIDI and audio filters. Figure 4.15 shows the architecture of the software design to implement constrainted based display. A configuration window is actually a display of a data structure called a **ConfigFilter** which consists of a list of objects or object types that are allowed to be displayed and two other data structures called **Config** and **ConfigLayOut**. **Config** is a list of devices and connections in the configuration and completely describes how the hardware is configured. **ConfigLayOut** describes the graphical presentation of the devices and connections in **Config** when displayed in a configuration window. When a configuration window is called upon to display all or a portion of a configuration

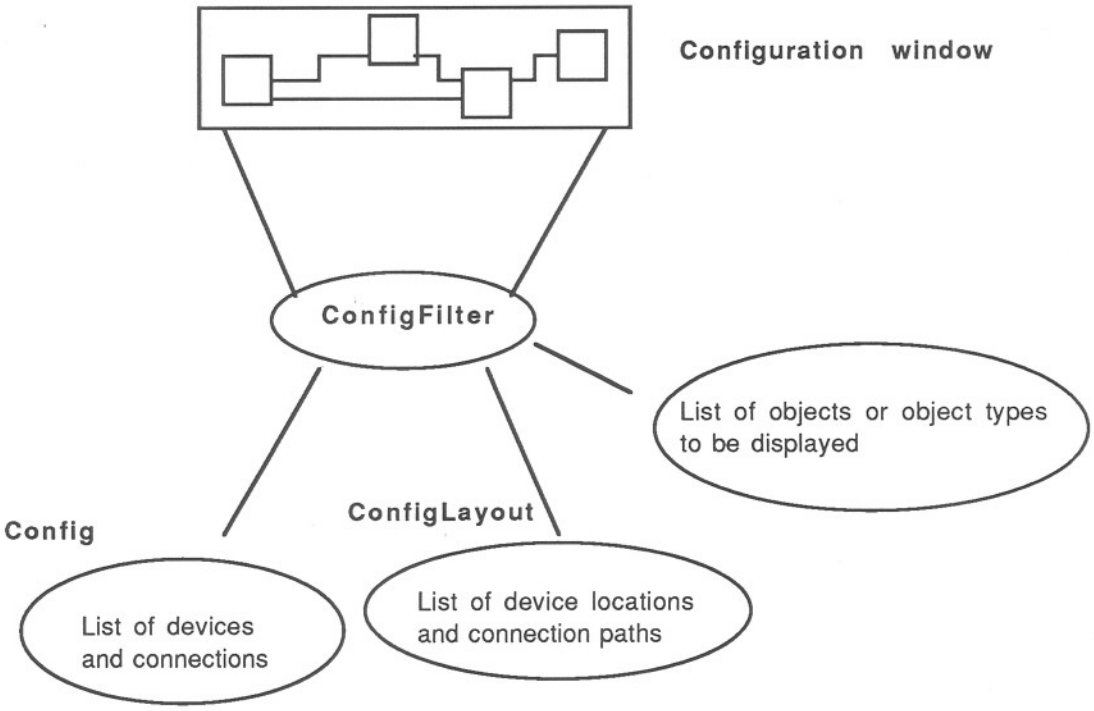


Figure 4.15: Software architecture of constraint based display. A configuration window is a display of a data type called **ConfigFilter**. A **ConfigFilter** consists of data types **Config** and **ConfigLayOut**, and a list of objects or object types that are allowed to be displayed.

it consults the **ConfigFilter** data structure. Each object to be displayed is tested versus the list of displayable objects and object types. If the object passes "the tests", it is displayed.

As an example of the filter mechanism in action, we can examine the method by which the MIDI filter is implemented. The **ConfigFilter** has two objects in it in this case: the class **Device** and the class **MidiConnection**. Before each object in a configuration is displayed, the system checks to see if the object passes through the filter. In this case the system recognizes both objects in the filter as classes and therefore checks the class of the object about to be displayed. If the object is neither a type of **Device** or a type of **MidiConnection**, it is not displayed.

Another advantage in our choice of architecture stems from the separation of the description of the configuration from the description of the layout of the configuration. In doing this, it allows us the option of having different layouts for same configuration. Thus the user can use different layouts for the same configuration to emphasize certain properties of a configuration and improve their understanding of the system. The implementation of the interface to allow alternate layout schemes is considered future work.

4.5 Navigation and Alternate Access

In this section we examine how the user in Virtual Studio may navigate through the hierarchy of devices and device control panels and how alternate access schemes can be constructed by the user.

4.5.1 Traversing the Hierarchy

The most simple method a user has of navigating through the hierarchy of devices is by following the hierarchical structure of a configuration. As described earlier, double clicking on a module icon causes a window with the module's internal configuration to be displayed. In this manner a user may access the devices which are internal to a module. The configuration window of a module's internal configuration

is displayed "on top" of the window containing the module's icon, thus maintaining a "child of the hierarchy on top" relation among over lapping configuration windows (although the windowing system does allow any window to be brought to the "top").

A user is not constrainted to close a configuration window before another can be opened and, therefore, the persistence of windows can a be utilized by the user to construct alternate access schemes. In other words, the user can traverse the hierarchy of device configuration windows, by double clicking on modules, till they have opened the desired window, then close all the windows opened by the traversal which are not needed. The desired window can then be "set aside" on the screen. The user can repeat this process till they have collected all the windows they desire for performing a task. Essentially this process allows the user to override the hierarchical access scheme by allowing a random access scheme, restricted to a set of user selected windows, to be constructed.

4.5.2 Rooms

While the two access schemes described in the previous section provide a well defined and customizable access scheme, they both have drawbacks. Hierarchical access, as described in chapter 1, can become extremely awkward if the user must toggle between using two windows widely separated in the hierarchy. The random access scheme, described in the previous section, is limited by the size of screen; the screen can only concurrently display a few windows at once, thus restricting the number of windows the user can access quickly.

In the application being studied, the ability to provide quick access to numerous control panel windows is critical to success of the system. In effort to satisfy this need we have adapted a windowing system technique from an office automation technology called Rooms. This technique, described in detail in [7], was developed by Card and Henderson and is designed to help users switch among tasks on which they are currently working. Rooms accomplishes this by providing the user with a number of screen-sized workspaces called Rooms. Figure 4.16 shows two typical

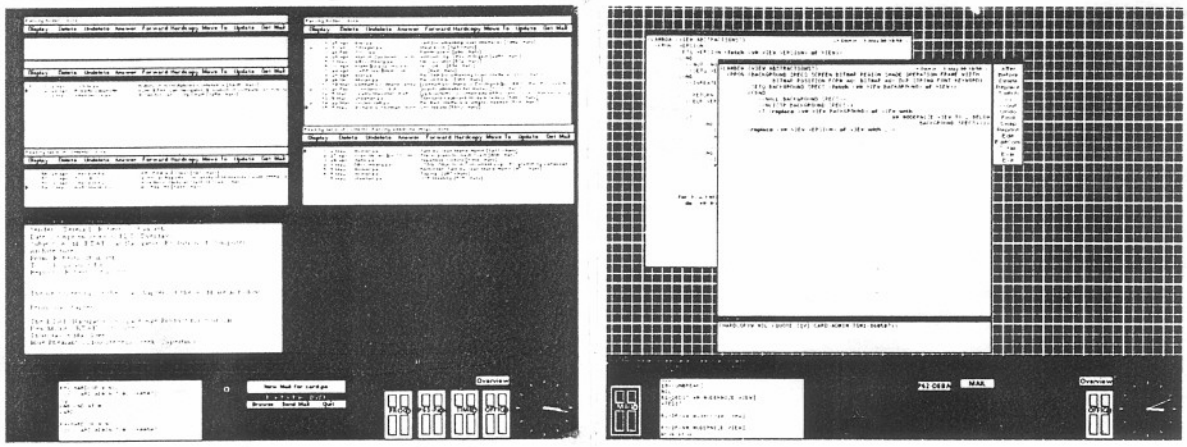


Figure 4.16: Two examples of Rooms from [7]. On the left, a Room used for reading mail. On the right, a Room used for programming. Note that there are windows common to both Rooms.

Rooms.

In each Room, there are a number of small icons called Doors. When a Door is selected with the mouse, the user has the illusion of transferring to a new Room containing other windows. By collecting the windows needed to perform a certain tasks in different Rooms, the user can easily switch between tasks by switching between Rooms. When a user enters a Room, he finds it in the same state it was when he left it and, therefore, no time or effort is wasted reacquiring windows.

Figure 4.17 shows a typical Room in Virtual Studio. The little window in the lower left corner is tool associated with Rooms itself and its label shows the current room the user is in. In this case the user is in the "Mixing Room". Note in the lower right hand corner are door icons which lead to other Rooms.

Windows can be added to a Room in Virtual Studio by three methods. First, the window can be invoked, or displayed in the Room. For example, a configuration window of a module may added to a Room by clicking on a module icon in another configuration window, or a window may be displayed by means of textual command given by the user in a command window. Second, a window may be explicitly "carried" from one Room to another. The carrying operation is supported by a Room's mechanism called Baggage. By selecting the Baggage icon, (shown in the lower left of the screen of Figure 4.17), the user is placed in a special mode in which

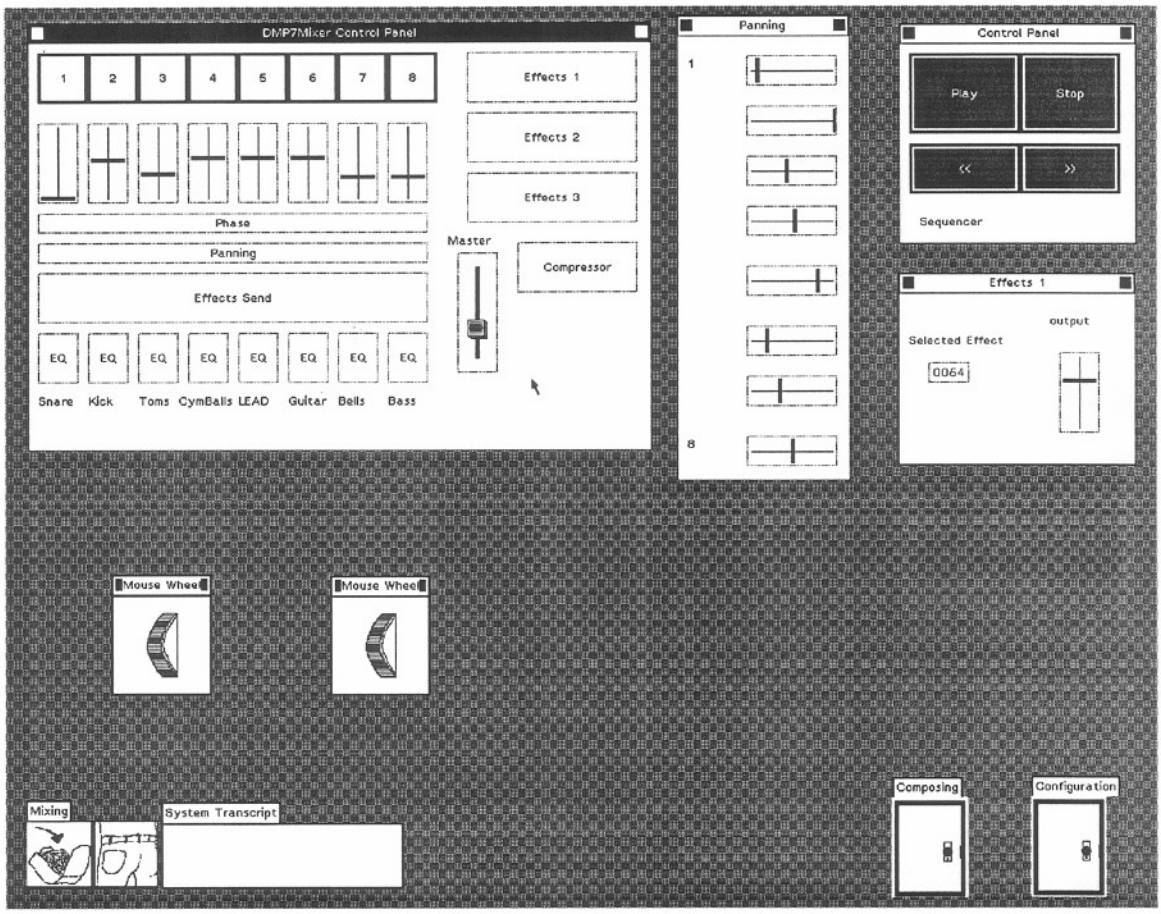


Figure 4.17: A "mixing" Room in Virtual Studio. In this Room the user has acquired and organized device control panels needed for the task of mixing.

they can select windows to be placed in their Baggage. When the user transfers to another Room, his Baggage is automatically unpacked. That is, the windows in Baggage are automatically added to and displayed in the new Room (the windows in Baggage also remain in the old Room). The user can also have a constant piece of Baggage called a Pocket. The icon in the lower left of the screen operates in the same manner as the Baggage icon. Whichever windows are placed in the user's Pocket are automatically displayed in each and every Room the user visits. Thus, the Pocket provides the user with global set of windows which appear in all rooms at the same location. For example, the window which contains the Baggage and Pocket icons is a window which is in the Pocket.

Windows which are shared between Rooms can have independent placements. In other words, a window may simultaneously be displayed with a different location and shape in different Rooms.

The user is also capable of constructing their own Rooms and Doors. By pressing the mouse button over the background of a Room a menu is displayed featuring commands to "add a Room", "add a Door", "teleport to a Room" and "delete a Room". When adding a Room, the user is prompted for name of the new Room. When adding a Door , the user selects which Room the Door is to from a menu. The command "teleport to a Room" allows the user to move to Room without constructing a Door to it. In both commands, "teleport to a Room" and "delete a Room", the user selects the desired Room name from a menu. Once a new Room is created, it can be populated by adding windows as described above. In the future, we hope to add the ability to save and restore Rooms.

4.6 Control

In this section we describe how the user can control devices within Virtual Studio. Since we have elected to represent device control panels graphically, the question is: what sorts of techniques can be used to manipulate the graphic controls? We are also faced with the input device bottleneck; while graphical control panels have numerous

knobs, sliders, etc. how can we, with just the keyboard and mouse, facilitate in some reasonable manner the manipulation of these gadgets? The answer to this question lies in a control slaving schemes. One control slaving scheme might be to allow the user to slave sliders to the mouse, as demonstrated in the DMP7-PRO system described in chapter 3. Another method is to allow graphical controls to be slaved to other graphical controls, then directly manipulate the master control with the mouse. Each of these schemes is a solution to a specific case. What we focused our attention on in the development of Virtual Studio was not specific slaving schemes, but a general and extensible scheme that could solve most or all the input control problems and an interaction technique to make slaving schemes easy to specify.

4.6.1 Direct Manipulation

Most obvious way a user can manipulate controls in Virtual Studio is by direct manipulation. Direct manipulation in Virtual Studio follows the common approach shown in numerous other graphical applications. For example, a user manipulates a graphical slider by selecting the wiper with the mouse then moving the mouse up or down. Figure 4.18 shows the types of graphical controls that are available for the construction of control panels in Virtual Studio.

Among the "standard" types of direct manipulation graphical gadgets we have special gadgets designed to solve some of the problems encountered in Virtual Studio. For example, a gadget called the "numerical slider" (developed by Buxton [6]), and the "virtual slider" are designed to give the user effective feedback and control over the value of a parameter while consuming little display space. Since display space is limited and our application has the property of having numerous parameters to be displayed, having gadgets with small display "footprints" is critical. Another advantage of these types of gadgets is their large "hit" area. That is, the region that has to be selected before the control can be manipulated is large. In both the "numerical slider" and the "virtual slider" the hit area is the entire footprint. In contrast, in a typical graphical slider which models a real slider, the hit area is only the wiper and, therefore even if this type of graphical slider is twice

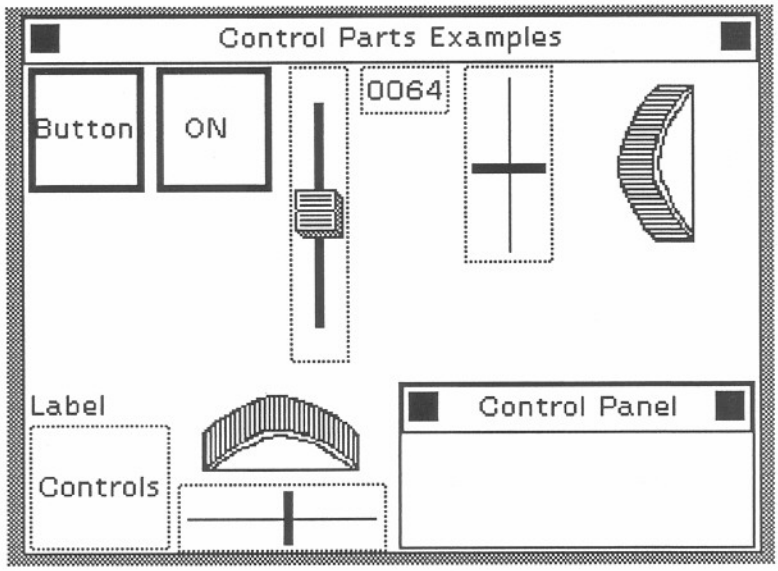


Figure 4.18: Parts used in constructing control panels. Top row: 2 types of ON/OFF buttons, a slider, a numerical slider, a virtual slider, and vertical continuous wheel. Bottom row: a user settable label, a control panel icon (used to reveal sub-control panels), a horizontal continuous wheel, a horizontal virtual slider, and a blank control panel.

the size of a numerical slider or virtual slider, its hit spot is still smaller.

Both the virtual slider and the numerical slider operate such that once the user selects the gadget, by pressing a mouse button over the gadget, vertical movement of the mouse increases/decreases the value of the parameter. While the mouse button is pressed the gadget's value is linked to the mouse even if the cursor moves outside the display box of the gadget. This feature gives the user the feeling he is adjusting a long throw high resolution slider; unlike a typical graphical slider, he does not have to worry about slipping off the wiper or moving past the end of the wipers run (an example of this irritating feature is the implementation of scroll bars in the Suntool programming environment). In effect, the user can have the feel of a long throw high resolution slider without the consumption of display space.

Thus, using these types of innovative gadgets, we can pack more displays of parameters into a fixed area than we can with traditional graphical gadgets, without making selection and manipulation of the gadgets difficult or reducing the amount of feedback concerning the value of parameters.

4.6.2 Slaving Software Architecture

Our goal in Virtual Studio was to develop and implement a general slaving scheme which allows arbitrary controls to be slaved to one another. In other words, we desired a general sort of system where graphical controls can be slaves to physical input devices, or slaved to other graphical controls, or both. We wanted to be able to construct arbitrary networks of slaves; for example, a master graphical slider may, in turn, be slaved to an physical input device and graphical knob. Essentially, we desired the ability to create master-slave relations between any "parameter" in the system. A parameter, as defined in Virtual Studio, is a variable. For example, a slider is really a graphical display of some parameter's value, or the y location of the cursor is a parameter. A master-slave relation between two parameters is really a dependency relation. That is, the value of the slave parameter depends on the value of the master parameter.

The relationship between a master and slave parameter is more complex than simply "the slave follows the master's value". There are an infinite number of relationships that could exist: the slave's value could be some fraction of the master's; it could be the log of master's value; it could be the master's value plus some constant, etc. What was desired was a method in which different functions, describing the relationship of the slave's value to the master, could be "plugged in".

The master slaving scheme in Virtual Studio is based on the above reasoning. We have developed a scheme in which any two parameters in the system can be cast in a master-slave relationship with an arbitrary function defining the relationship. A class of objects called Parameter play a central role in this scheme. The formal definition of the class Parameter, using a Smalltalk like language, is:

Object subclass: Parameter

instanceVariableNames: 'value name changeAction

masterSlaveRelations dependents'

The instance variable **value** is an object representing the value of the **Parameter**. In general the value is some sort of number, although it can be any type of object (for example, the string 'OFF'). **name** allows a **Parameter** to be given a

textual name. **changeAction** is a block of executable code which is executed each time the value of a **Parameter** changes. **masterSlaveRelations** is a list of all the relations in which the **Parameter** is a slave. **dependents** is list of all other objects in the system that depend on the **Parameter**. Note that the objects in this list may not all be **Parameters**; for example, a graphical display of a **Parameter** also depends on the value of the **Parameter**.

The object class used to represent relations is called **MasterSlaveRelation** and is defined as:

Object subclass: MasterSlaveRelation

instanceVariableNames: 'master slave function'

master and **slave** are the two **Parameters** involved in the relation. **function** is a block of code which, when evaluated, produces a value for the **slave** and thus serves as a description of the functional relation between the **master** and **slave**. **function** uses the **master Parameter's value** and the **slave Parameter's value** as arguments.

In order to understand the procedures that maintain a master slave relation, we can describe flow of control in the life cycle of a master slave relation. Initially the user indicates somehow that a **Parameter** should be slaved to another **Parameter**. This causes the system to invoke a procedure called **enslave**. This procedure adds the slave to the master's list of dependents and creates a **MasterSlaveRelation** describing the relationship and adds it to the **masterSlaveRelations** list of the slave.

Due to external events, the master **Parameter** may have its value changed. In order to change the value of a **Parameter**, a procedure called **value** must be called with the new value as an argument. After the instance variable **value** is set to the new value, the update procedure for each dependent is called. If the dependent is a **Parameter** and therefore a slave, it obtains the proper **MasterSlaveRelation** from its **masterSlaveRelations** list, and evaluates it, assigning this value to its own **value** instance variable. Thus the master slave relation is maintained.

Eventually the user may indicate that a master slave relationship should be

broken. In this case the procedure **free** is called which removes the slave from the dependents list of the master **Parameter** and removes the corresponding **MasterSlaveRelation** from the slave's **masterSlaveRelations** list.

Clearly this scheme is very general. There is no restriction on which **Parameters** can be involved in a master slave relation. Thus networks of master slave relations can be implemented. Furthermore, as a result of the functional relation between two Parameters being described by a replaceable block of code, different functions can be easily implemented. For example, the function in which the slave follows the master value is implemented by setting **function**, in the **MasterSlaveRelation** for the two **Parameters**, to:

function $\Leftarrow$ **[master value]**

Thus when the slave evaluates **function**, it returns the master's **value**.

Similarly a relative relationship is expressed by:

function $\Leftarrow$ **[slave value + master value]**

or a 2:1 ratio is expressed:

function $\Leftarrow$ **[master value / 2]**

Ultimately, function can be any Smalltalk expression. For example:

function $\Leftarrow$ **[clock time inhours < 12 ifTrue: [slave value] ifFalse: [master value]]**

creates a master slave relation where the slave only follows the master in the afternoon and evenings.

4.6.3 Interaction Pragmatics

Although the previous section has presented an elegant and general scheme for slaving parameters, if the user is given an awkward or non-existent method of specifying master slave relationships, our elegant scheme is of little or no use to the user. Thus, in this section we examine the interaction techniques used in Virtual Studio which allow the user to specify master slave relations.

The basic interaction used in expressing a master slave relationship is very simple: the user indicates the slaves followed by the master. This approach models the

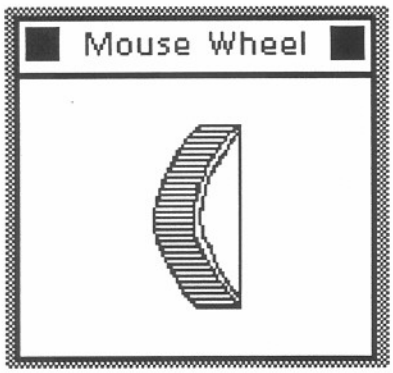


Figure 4.19: A graphical gadget which emulates a continuous wheel. The user can "move the wheel" by selecting it with the mouse and moving the mouse up or down.

way users think about slaving: "these controls, slave them to this control".

Borrowing a technique from recording studio console design, graphical controls exist in Virtual Studio to act purely as master controls. We extend this technique by allowing the user to create their own master controls. Furthermore, Virtual Studio master controls are relative controllers and therefore nulling problems, which plague recording console technology, are avoided. Figure 4.19 shows a graphical master control which emulates a continuous wheel. The user "moves" the wheel by selecting it with the mouse and moving the mouse up or down.

In Virtual Studio, the user specifies master slave relations using gestures. Figure 4.20 shows the basic "slaving" gesture. The user circles the controls to be slaved and then points to the control which should be master. Figure 4.21 shows the freeing gesture. The user circles the controls to be freed and indicates they have no master.

Virtual Studio can be extended to incorporate other physical input devices besides the mouse and slaving to these devices is specified in the same manner as slaving to graphical controls. Icons are used to represent alternate physical input devices. In order to slave controls to a physical input device, the user circles the controls to be slaved and points to the icon of the physical input device. For example, Figure 4.22 shows the icon used to represent a two slider four button input device in Virtual Studio. Controls are slaved to a particular button or slider on this device by circling the controls to be slaved and pointing to the appropriate button

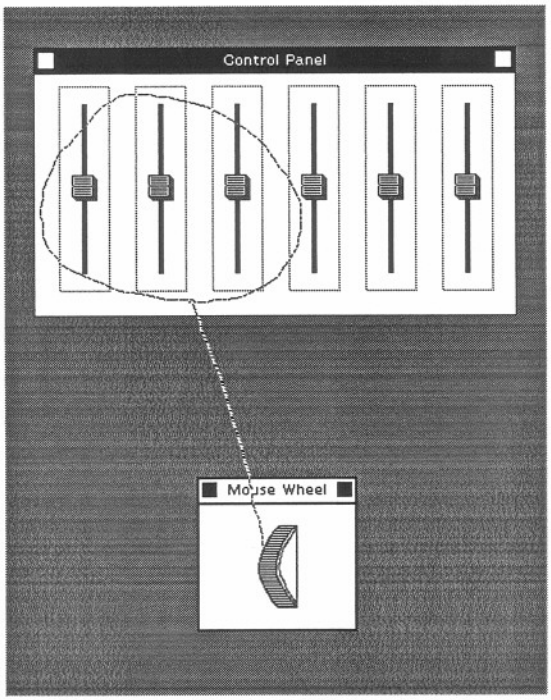


Figure 4.20: An example of the slaving gesture.

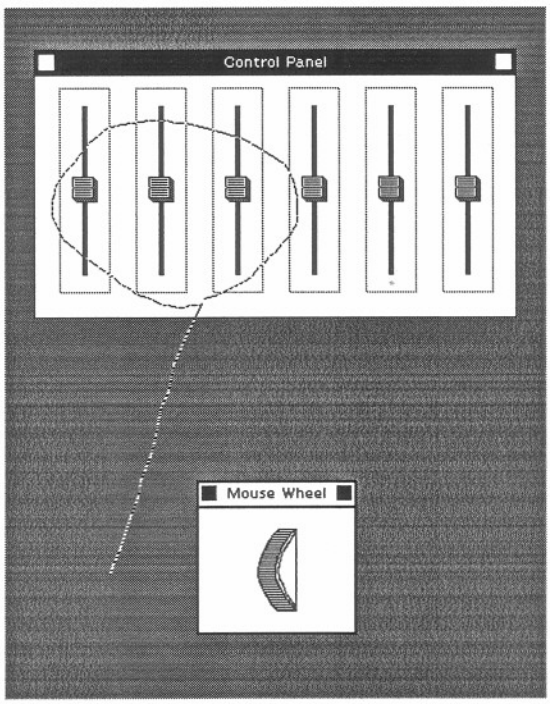


Figure 4.21: An example of the freeing gesture.

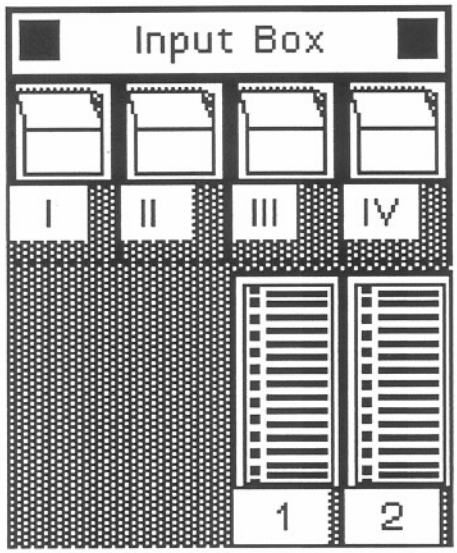


Figure 4.22: The icon used to represent a real four button, two slider input device.

or slider on the icon.

By repeated use of the slaving gesture, networks of master slaves can be setup. For example, graphical master controls can be slaved to other controls, thus creating master controls which are also slaves, or a single control may be slaved to several masters.

Currently no interaction is available for the user to specify the functional relationship between master and a slave (for example: "the slave follows the master at a 3 to 1 ratio"). A functional relationship is selected from a set of default functions by the system depending on the types of the two parameters involved. The default functions are designed to provide such things as relative/absolute conversion and reasonable control:display (C:D) ratios between different types of parameters. The burden of specifying the functional relationship for each master slave relation is alleviated from the user by providing "smart" default relationships. In the future, we hope to give the user the power to interrogate master controls, access a list the functional relations to slaves and the ability to modify these functional relations.

A feature missing in our current implementation of Virtual Studio is the ability to display information about what master slaving relations are in effect. Future work could be done on developing diagrams which display the network of masters

and slaves. Another effective tool would be one which answers queries such as:

- Who is slaved to this control?
- Who controls this parameter?

This problem is discussed further in Chapter 5.

4.7 Discussion

The previous sections of this chapter have given a description of the system which is the object of the case study. In this section we take a step back to discuss and justify the design decisions made in the development of Virtual Studio and highlight issues revealed by the case study.

4.7.1 Interaction Pragmatics

Central in the development of Virtual Studio is the use of gestures as a means of interaction. We made a decision to avoid the approach of "use a gesture for every command" based on the fact that gestures, like most interaction techniques, are more suitable for certain types of tasks than others. This, of course, gave rise to the issue of the side effects brought about from the integration of gestures with other interaction techniques such as direct manipulation and menus. The major side effect revealed by the case study was the interference of direct manipulation commands on the initiation of gesture commands. This interference is best described by an example.

Suppose the user wishes to delete a device from a configuration window. The interaction involved begins by pressing down the mouse button and then drawing a horizontal line through the device's icon. But suppose the user begins the gesture by pressing down the mouse button over a device icon label; this action corresponds to the initiation of the direct manipulation move command and, thus, the wrong command is invoked and the user ends up moving the icon horizontally according to their gesture. We conjecture that this type of error results from a belief that

users have about gestural interfaces. This belief can be described informally from the user's view as: "Gestures are like using a pencil and paper. You don't directly manipulate objects by picking them with the mouse, you indicate to the system the manipulation and it carries it out, just like a typist carries out a proof reader's penciled in corrections". For example, when a person points with a pen to a character on a piece of paper and then makes a horizontal stroke, they do not expect the character to "move" along with the pen. The problem is that giving the user gesture-commands which are based on this kind of pencil and paper metaphor produces errors when the metaphor breaks down. The metaphor, in the case of Virtual Studio, breaks down due to the inclusion of direct manipulation as an interaction technique.

However, this problem is not fatal to the success of the integration of gestures and direct manipulation. After the user has made this type of error several times they learn quickly that there are such things as direct manipulation "hot spots" and what they thought was the initiation of a gesture command was, in fact, clearly a direct manipulation command. Based on this, we feel that a training tool could be implemented which would allow first time users to enter an interactive training session where they can learn the interaction "vocabulary" of the system through exploration.

It is interesting to note that the interference of gesture commands and direct manipulation commands can be avoided by the use of a double button mouse: one button can be used for direct manipulation and the other for gestures. When the "gesture button" is pushed the system only interprets gesture-commands and similarly for the "direct manipulation button". Unfortunately the user may still make errors by confusing which button does what. In Virtual Studio we used a single mouse button approach for two reasons: (1) we wanted to avoid the problems associated with confusion caused by "which button does what" and (2) we felt a single button approach would be a good acid test for the integration of gestures and other interaction techniques (which was an underlying interest in the study).

The case study has also revealed that even though both gesture commands

and direct manipulation commands are used in Virtual Studio, the kinesthetics of the interface are consistent. This consistency is due to the use of the concept of "tension" in the design of command interactions. As we have described earlier, gesture commands are invoked by a short uninterrupted period of tension. All direct manipulation commands and menu based commands behave in a similar manner. For example, the adjustment of a numerical slider requires the tension of keeping the mouse button pressed while the adjustment is being made.

Another major issue concerning gesture-commands used the case study revolves around gestures as "tools of thought". As described in chapter 1, interaction pragmatics are essentially the notation of our visual programming language and the choice of notation can either help or hinder the user in expressing himself. The case study shows how gestures and the consistent way they are used to invoke commands reflect (and therefore reinforce) the semantics of the commands. For example, the grouping gesture is used in many commands such as device encapsulation, slaving/freeing groups of controls, moving, copying and deleting devices. While each one of these commands carries out a different operation, the way the user specifies the scope of the operation is identical. Furthermore, the grouping gesture reflects the way the user thinks about scoping objects. Rhyne and Wolf and also Gould and Salaum have shown, in a studies presented in [35] and [14], that a popular free hand scoping gesture is the "circling" gesture as used in Virtual Studio. Thus, by using gestures in Virtual Studio, commands are invoked in a manner which reflects the way the user "thinks" about the command. This approach is not only apparent in the way scoping is done, but also in other commands. For example, the deletion of objects involves the "scratch out" gesture or the addition of a connection involves "drawing" the connection.

The case study has shown how simple gestures can be used to relate a large amount of information to the system. A chronic problem, in applications which utilize diagrams with icons and data flow connections among them, is in how the user specifies a connection. A "make a connection command" may require many arguments: the two icons involved, the type of the connection and the path the con-

nection should follow. Generally the interaction involved in invoking the command is a series of menu selections and mouse points. Because of the fragmented nature of the interaction, the possibility of user error is high. In contrast, the interaction to add a connection in Virtual Studio involves one continuous gesture which specifies the objects involved, the type of the connection and the path of the connection. There is no fragmentation in the interaction so the possibly of error is reduced.

Continuity of the interaction also leads to better time-motion performance. The connection operation essentially consists of many simple subtasks: selection of output jack, selection of input jack, indication that a connection should be made, and specification of the path of the connection. If we perform a button-stroke analysis of our connection interaction, we find it consists of a button down, a drag, a button up, then a selection from the connection table (button down, up). In contrast, the connection interaction in the Katosizer system requires selection of the input (button down, up), selection of the output (button down, up), then specification of the connection path (button down, drag, button up). Thus our connection interaction requires one less button up and button down. Furthermore, we also display information about other connections that exist between the two objects thus aiding the user in making the correct connection and masking out information not related to the connection.

Many popular systems use accelerators to provide expert users with "short cuts" to invoke commands. Unfortunately, accelerators may involve the use of awkward control key sequences or non-intuitive interactions and may appear to user as afterthought retrofits to the interface which are difficult to learn and hard to remember. Essentially, hard to use accelerators make the transition from a novice to expert user difficult. This problem has two solutions: (1) provide the user with an easy to learn set of accelerators or (2) provide the user with commands that require no "acceleration". Approach (2) is ideal, since it eliminates the transition from novice to expert user, although difficult to implement. The case study has shown how gestures fit nicely into this approach.

Generally, accelerators allow the user to bypass menu selections or "object picks"

Figure 4.23: The "upside down T" gesture, left, results in the creation of a master controller, right.

by replacing the entire interaction by one single gesture (for example, a control key sequence). Gestures in Virtual Studio allow you to do just that. For example, issuing the encapsulate command does not involve individually picking each device then selecting the command from a menu; it involves one single gesture. Thus the novice benefits from the fact that not only are gesture commands easy to learn and remember, but they are also intrinsically efficient and therefore no special accelerators are needed.

Related to the issue of gestures as accelerators, is the ability in Virtual Studio to "chunk" several gestures commands into one "macro" command. For example, Figure 4.23 shows the "upside down T" gesture used to create a master controller.

This gesture can be combined with the slave gesture into a single gesture which allows the user to group a set of controls controls to be slaved, and create and specify the location of the master control. Figure 4.24 and Figure 4.25 show an example of this "macro gesture".

Once again, gestures demonstrate the ability to relate a large amount of information to the system with very little user effort. This example also points out how gestures can be used to minimize the amount of effort required from the user to issue a command. Using this fact, we make use of gestures in Virtual Studio for commands which are performed with high frequency. The success of Virtual Studio was dependent on the how easily slaving could be performed. Thus gestures were developed which allow the user to quickly perform slaving commands.

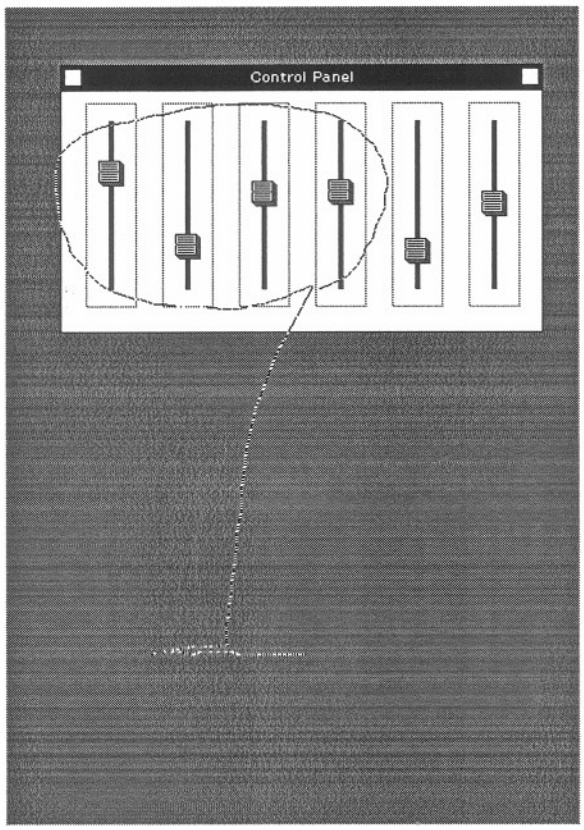


Figure 4.24: The combination of the "upside down T" gesture and the slaving gesture.

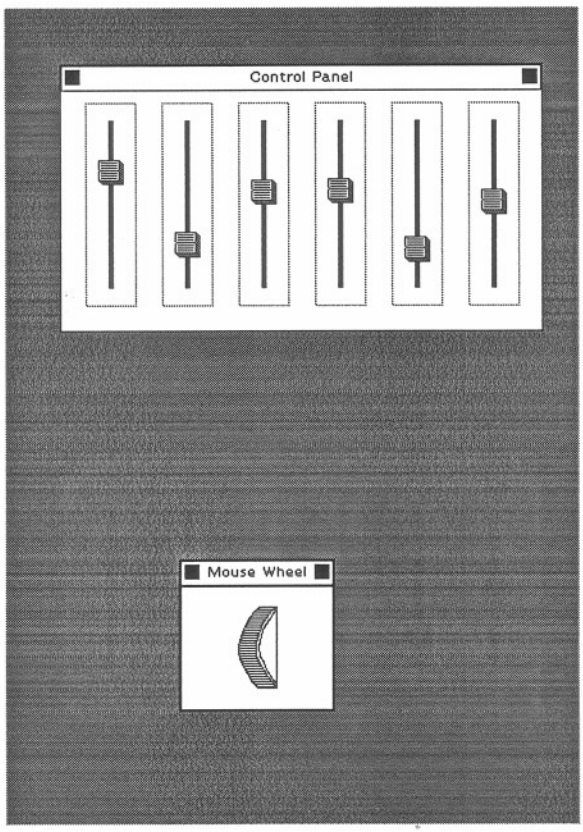


Figure 4.25: The result of the gesture in Figure 4.24: the system responds by creating a master controller and slaving the circled controls to it.

4.7.2 Rooms

The adoption of the Rooms model to Virtual Studio solves several problems. Essentially, Rooms is required for the operation of the Virtual Studio: while hierarchical access is effective in setting up and understanding the configuration of the studio, once the studio is configured, non-hierarchical access is better suited to the operation of the studio.

Why is Rooms effective? There are several answers to this question which we will now discuss.

The Rooms model increases the amount of virtual display space available to the user. Thus the user can have more windows virtually displayed over all Rooms than can be displayed on a single screen. This greatly increases the number of windows the user can quickly access without having to traverse the configuration hierarchy. Since Virtual Studio is populated by numerous control panels windows, a method which increases display space greatly aids the user.

Grouping of windows (most likely device control panel windows), into Rooms has several advantages. First, windows can be grouped into Rooms according to the task they are associated with. Thus, when a user enters a Room to perform a certain task, all the tools needed for the task will be on hand. For example, the control panels for the mixer, associated effects units and the tape recorder can be placed in a "Recording Room". Associated to this, is the property that when a user reenters a Room, his "tools" appear just as he left them. Thus the time for the user to refamiliarize himself with the Room is very small. In effect, Rooms models a traditional studio. The set of control panels that is within arms reach in a traditional studio correspond to the set of control panel windows displayed in a Room and in the traditional studio, when a user moves back to a "spot", he finds things as he left them.

While Rooms models a traditional studio, it also goes beyond it. Ideally, in a traditional studio, the user collects around him all the devices needed to accomplish some task. Unfortunately, it is not always physically possible to bring the control panels for every device needed within arms reach. In contrast, in Virtual Studio,

every control panel is either directly accessible on the screen or only a few mouse clicks away. Thus, all control panels are within arms reach.

On the other hand, even though all control panels are "at arms reach", the user in not inundated by the display of every control panel at once. The use of Rooms allows the user to keep all control panels close at hand, yet hide the control panels which are not associated with the task. If the user's wishes to temporarily suspend their current task and access a hidden a control panel, the process of suspension and resumption ("knocking on a door") is painless.

The feature of sharing windows between Rooms enhances the user's power. In a traditional studio there are many situations where a control panel is needed in two places. For example, the transport control panel for tape deck may be needed when the user is sitting at the console and also when at a synthesizer keyboard. Generally, only one physical control panel exists, so the user must make some sort of compromise, such as moving the synthesizer close to the console. In Virtual Studio, windows (and therefore control panels) can be shared among Rooms, thus giving the user an infinite number of convenient "remote control panels".

Inherent in the success of the adoption of the Rooms model into the Virtual Studio, is the fact that Rooms can be easily customized by the user. Setting up new Rooms allows the user to construct their own workspaces specialized to a task. Allowing the user to construct their own doors permits them to set up navigation paths through Rooms which are suitable to their work habits. Furthermore, we can think of a group of Rooms and Doors like a house; the person that builds their own house, clearly understands it and can find their way around it.

Finally a word should be said about the effectiveness of the concepts of Baggage and Pocket. Both Baggage and Pocket are effective mechanisms in aiding the user in the distribution of windows into Rooms where they are needed. The Pocket mechanism implements the "this window in every Room" notion. This fits in very well with the notion of having certain control panels in the studio, such as a tape transport controls, always accessible. The Baggage mechanism implements "put this window in this Room as well" notion and is used to selectively place windows

in Rooms such that a Room can be customized to suit a task. Furthermore, Baggage can also be used to temporarily bring a window into a Room in a sort of mini task interruption; no matter how organized the user is in setting up his Rooms according to task, inevitably there will be times when the user thinks: "I just need this control panel temporarily". Because of the simplicity of the Baggage mechanism, the user can easily move into a Room where the control panel needed is, "bag it" and return to Room where they are working. The point is that Baggage is another feature of Rooms which aids the user in the face of unpredicted task switching.

4.7.3 Control

In terms of the software architecture, the beauty of Virtual Studio's master-slaving scheme is that problems such as conflicting C:D ratios between input devices and parameters, complex relations between parameters, and alternate views are solved by the generality of the approach. Conflicting C:D ratios between input device and parameters are solved by specifying a master-slave relation between the input device parameter and the slave parameter such that the converting function produces a reasonable C:D ratio. As we have shown earlier, a relation function may be a complex Smalltalk expression and therefore complex relations between master and slave parameters can be supported. Finally, Virtual Studio's master slaving scheme can also be used produce alternate views of parameter values. As a simple example, a numerical slider can be made slave to vertical slider. Now the numerical slider is in fact an alternate way of displaying the value of the parameter associated with the vertical slider. We feel slaving as it relates to alternate views has great potential. For example, in the future, we imagine things like envelope display gadgets being slaved to a set of numerical sliders to produce a graphical view of several parameters.

A minor drawback in our master slaving scheme is that elements of the system, which could potentially be a master or slave, must be declared as Parameters. In other words, the implementor of a piece of software must predict what elements of software the user might want to use as a master or slave. For example, suppose an implementer constructs a sequencer for the Virtual Studio. What elements of the

sequencer does he declare as parameters? One obvious one is tempo and perhaps the sequencer transport controls. The point is that elements which are not implemented as Parameters can never involved in master slave relations. Thus, the implementor is forced to "predict the future" when deciding what elements the user will desire as slaves or masters.

The use of a single gesture to express master slave relations has many advantages. First, the interaction for slaving/freeing one control is that same as slaving/freeing many controls, thus related commands have consistent syntax. Second, and most importantly, this technique is extremely time efficient; our approach to solving our control bottleneck (that is, having many graphical controls and just few input devices to manipulate them) has been to slave sets of controls to input devices. During a typical session using Virtual Studio the user may invoke the slaving/freeing commands numerous times and, thus, the efficiency of these commands are critical.

We believe our approach gives the user the same amount of control as they would have in a traditional studio and, in some aspects, Virtual Studio provides more control. This can be proven by examining the amount of control a user has in a traditional recording studio. Our proof has 3 cases. Case (1): the user wishes to manipulate a single control. In this case the user simply reaches out and adjusts the control. In Virtual Studio the interaction is equivalent; the user just points to the control and manipulates it with the mouse. Case (2): the user wishes to manipulate eight or fewer controls at once. In this case an the user can push each control with a finger. In order to adjust this many controls at once, the controls generally have to be adjacent faders. In effect, two sets of four faders each are slaved to each hand and the faders in a set all increase and decrease the same amount. In Virtual Studio, in order to imitate this type of manipulation, the user could slave 4 faders to a graphical master control, and 4 faders to another physical input device. Now using two hands the user can adjust the faders. In this case, setting up the two slave groups requires two gestures, after which the effort required to adjust them is similar to the traditional studio. Case (3): the user wishes to manipulate more that 8 controls at once. In the traditional studio this generally requires slaving faders

to master faders. In Virtual Studio the requirements are similar: the user must use gestures to group controls together to be slaved to master controls.

We have shown that Virtual Studio can roughly handle all the types of control interactions encountered in the traditional studio but it also extends the amount of control the user has. Generally, in a traditional studio, the user can only slave recording console input faders. In Virtual Studio, the user can slave any Parameter in the system. Although, in the traditional studio has many knobs, sliders and buttons serving as input devices, the user only has two hands and, therefore, can only manipulate a few at a time. By allowing the user to slave any parameter, Virtual Studio effectively provides the user with extra hands.

Finally, a comment should be made regarding our experience using graphical master controls. First, it was critical that master controls were relative controllers. Not only does the use of absolute controllers introduce typical nulling problems, but the limited range of an absolute controller introduces problems. Suppose a fader is slaved to an absolute controller at a ratio of 1:2. In this case, the user cannot manipulate the slave control over its entire range because the range of the master is limited. Making master controllers relative eliminates this problem; a relative controller has no limit on its range.

4.8 Summary

The first six sections of this chapter described the system developed as a case study. After this description the remaining sections discussed what was observed and learned from the case study. Central to this discussion was the use of gestures in Virtual Studio, the adoption of the Rooms model to facilitate operation of the studio, and the master-slaving scheme developed to aid the user in device control.

Chapter 5

Summary

The general approach of this thesis has been to investigate how new forms of representation and interaction can enhance user performance for a broad class of computer applications. The basic problem faced in this class of application is editing a network of objects and the relationships among them. As the network becomes denser, the user must have a methodology for dealing with the resulting complexity or face information overload. We have focused our attention on a representation which uses hierarchical chunking of the network in attempt to reduce its apparent complexity. With this representation in mind, we took the approach that the interactions used in editing the representation were just as important as the representation itself: interaction should help, not hinder, the user in editing and understanding the representation. Using this concept as our mandate, we attempted to create interactions in which the pragmatics reinforced the semantics of the representation and the interaction itself. To facilitate our investigation into representation and interaction, we elected to use a particular application as a case study: a computer interface to a personal audio studio. Although this thesis deals with particular application area, we felt our choice of application was representative of a large class of applications and therefore our results can be generalized and applied to many other areas.

Chapter 1 attempts to more clearly identify the class of application being addressed in this thesis and the associated problems. Initially, the general charac-

teristics of the case study problem were presented: networks of objects with data flow connections, these networks are generally large and complex, and hierarchical chunking of the network is a meaningful method of reducing complexity. In addition, we observed that objects in our case study had parameters associated with them and that the characteristics concerning parameters were: more parameters exist than can be viewed on the screen at once and user access to parameters exhibits locality. We then proceeded to present examples of other applications which exhibit these same characteristics in order to demonstrate that our case study was representative of many other applications. The examples used included VLSI design, Recursive Transition Networks, Data Flow Diagrams and Hyper-Text. Next, the general issues surrounding the case study were identified. First, hierarchical access was seen as a disadvantage in some cases. The question was what sort of scheme could be developed to support both hierarchical access and user customizable access? The phenomena of locality of reference in terms of user access to parameters was seen as a key concept in the solution of this problem. Next, the value of selective and alternate views of the network using other information hiding techniques in addition to hierarchical structuring was presented. The central issues were the type of data structures needed to support alternate views, and how alternate views could be applied by the user. We then proceeded to the issue of providing the user with the means to control numerous object parameters. It was noted that some sort of scheme which permitted slaving parameters to input devices and to other parameters was a possible solution. Finally, we focused on viewing our case study system as a visual programming system. Given this viewpoint, the pragmatics of interaction could be considered the notation of the language. The question then became what form of notation best suited our representation?

A description of the case study problem was presented in chapter 2. The case study problem was identified as making a personal audio studio controllable from a central computer. In following sections a description was provided by first presenting a system analysis of problem in terms of user and tasks performed, and then by identifying the problems encountered by users in the performance of their tasks.

The major problems concerned lack of integration of software tools, inefficient task switching, manual reconfiguration of the studio, poor control panel interfaces to devices and limitations in the number of devices that can be controlled by a single user. These problems provided our motivation in "computerizing" the audio studio; we felt by migrating all device control panels and the configuration process to a computer these problems could be solved and, thereby, improving user performance and the state of the art in user interfaces to personal audio studios. Our final motivating factor was than our work could be elegantly extended to systems of future, such as DAWs.

With chapter 1 and 2 providing a description of the types of problems being addressed by the case study, chapter 3 examined other systems which faced some of these problems. For each system we described how it was related to Virtual Studio: what problems it did solve and what problems it failed to solve. Furthermore, we identified the features of other systems that influenced the design of Virtual Studio. Among the systems described, the Katosizer was by far the most influential. Finally it was concluded that no system solved all our problems and therefore the development of Virtual Studio was justified.

Chapter 4 describes the system developed in an attempt to solve the problem and then discusses what has been observed and learned from the case study. The key design features of Virtual Studio were the adoption of configuration diagrams to control the configuration of the hardware in the studio, the Rooms model and the development of a general master slaving scheme. The discussion of what was observed and learned from the case study was broken into three categories: gestures, Rooms and control. Our main observations concerning gestures were: (1) gestures can be successfully integrated with other interaction techniques (2) in this class of application, gesture interactions can be developed that reflect the semantics of the representation and the interaction itself and (3) this, plus the efficiency of the pragmatics of gesture commands, may improve user performance. In terms of Rooms, it was observed that it was key in solving many of our problems concerning non-hierarchical access and user customizability. In terms of control the combina-

tion of gestures as a means to specify enslavement/freedom of controls, plus the general nature of slaving scheme not only provides the user with the same amount of control as they have in the traditional studio but extends their control.

5.1 Conclusions and Contributions

The results of our case study can be divided into two groups: results that can be generalized to other applications and results that contribute to solving application specific problems. While generalizable results are important because they can be applied to solve problems in other application areas, we feel our application specific results are equally important because our case study application area exists in the real world and exhibits problems which affect real users.

The generalizable results of this thesis fall into two categories: representation and interaction. In terms of representation, this thesis demonstrates the effectiveness of a representation which uses hierarchical encapsulation as a means to reduce the apparent complexity of networks of objects and relations. This representation technique not only allows the user to bury the complexities of a system, but it provides the user with a hierarchical access scheme to objects in the system. This hierarchical access scheme can then be used to the user's advantage to permit navigation through the system and control over the level of complexity being viewed.

In terms of interaction, we have developed and demonstrated interaction techniques which not only permit the construction of these hierarchical representations, but, by virtual of their pragmatics, are easy to learn and perform. Furthermore, and most importantly, our interaction techniques reinforce the semantics of the interaction and the representation. We believe that this cooperation between representation and interaction can improve user performance.

As spin off from the development of these interaction techniques, the case study has also shown that gestures can be successfully integrated with other interaction techniques. By application of the notion of "tension", a unified feel to the kinesthetics of all types of interaction techniques was accomplished. Not only is tension used

as the "glue" to bind the subtasks of the command together, but it serves as the common feature between gesture, direct manipulation and menu based commands. Thus interactions with the system have a consistent feel.

While our case study has demonstrated the effectiveness of hierarchical encapsulation as a means to reduce complexity and provide structure to the problem, it has also addressed the problem that, in actual operation, hierarchical access to objects is unsuitable. Once again we can return to our stereo metaphor: "when the user is done configuring the system, they no longer care about the connections and only want to deal with control panels". In other words, hierarchical structuring is, in fact, the wrong representation to use when the user is done configuring the system, and wishes to deal only with the control panels of objects. The solution to this problem, demonstrated by the case study, is the adoption of the Rooms model which allows the user to construct their own access schemes based on personal needs and tastes.

A critical observation made in developing Virtual Studio was the realization that access to control panels in traditional studios demonstrates the phenomenon of locality of reference. When we first began our systems analysis of personal audio studios we were faced with several serious questions: How can we present all the physical controls available in a studio on one screen? Can we help reduce the information overload the user encounters when using numerous controls? In the endeavor to answer these questions, we realized that user access to device control panels is based on the task being performed and therefore, given the fact that users switch between many tasks, we reasoned that the locality of reference concept could be applied to control panel access. The realization of this concept's applicability to our case study problem and the fact that the Rooms model exploited this concept in order to support task switching led to the adoption of the Rooms model. We feel that the adoption of the Rooms model is key to the success of Virtual Studio.

In terms of application dependent results, the heart of this thesis is that we have developed and demonstrated the value of: (1) the framework of an integrated environment to serve as the interface to a personal audio studio and (2) a general

master-slaving scheme to enhance the amount control the user has over devices in the studio. Our framework is designed such that the user can "hang" all his tools in it. For example, not only does the interface supply tools for configuring devices and controlling them, but other tools such as sequencers, text editors, voicing programs, etc. can be introduced. Thus, all tools exist under the same "roof". The integration of all device control panels and software tools into a single interface solves the problems described in chapter 2. First, all the tools the user needs are on hand plus the computer can assist the user in switching between tasks and tools and second, consistent interfaces between different device controls panels and other tools can be supported. Furthermore, since the design of controls is not limited by physical constraints, more meaningful displays and controls can be presented to the user. Finally, once the connections among devices are under computer control, we can use the power of the computer to remember and restore complex configurations.

Our generalized master-slaving scheme was developed to solve two problems. First, by virtue of our "graphical control panel" approach, we were faced with an input device bottle neck: How could the user with just a few input devices manipulate numerous graphical controls? Second, we saw Virtual Studio as a prime opportunity to enhance the user's control over devices. The question was: what sort of scheme was needed to provide this extended control? We feel our generalized master-slaving scheme solves both these problems. The first problem is solved by allowing the user to slave graphic controls to physical devices. We also developed an intuitive and efficient technique which allows the user to specify master-slave relations. The second problem was solved by allowing any parameter in the system to be "slavable" to any other parameter. Using this mechanism, the user can exercise control over parameters in the system that is not possible in the traditional studio.

5.2 Future work

Virtual Studio is a vision of the future. While we have done much of the ground work there still exists many areas for invention and enhancement.

We feel the Rooms model can be further adapted to our application. We would like to provide the user with an "Overview" Room which shows the user a diagram of how the Rooms and Doors are laid out. The Overview Room is intended to help users get their bearings and assist them in navigation through and in the layout of Rooms. We would also like to adapt the "suite of rooms" concept. The "suite of rooms" concept corresponds nicely to the project concept of a studio. Different projects have different objectives and tasks, and therefore each require their own setups. Giving the user the ability to painlessly switch between projects would be a god-send. A Suite is a set Rooms which can be saved and restored at any time. In addition, the user can also edit Suites by copying new Rooms from other Suites. Thus, by saving and restoring Suites, the user could quickly switch between projects.

The virtualness of the system could be further exploited. In the current version of Virtual Studio, when a user switches Rooms, the configuration of the hardware remains the same. We would like to incorporate the ability of the system to re-configure itself upon entry into a Room. For example, suppose the user has the hardware configured for performing the mixing down task. Suddenly, the user has an inspiration for a new song and needs to capture the idea by playing his synthesizer. Unfortunately, the system isn't configured such that the synthesizer can be played. The ideal situation would be for the user to transfer into a practice Room, where he knows the synthesizer is hooked up to play. Upon entry to the practice Room, the system reconfigures the hardware to its practice Room configuration. Once the user's creative burst is over, he returns to the mixing Room where upon entry the mixing configuration is restored.

Currently we have developed the fundamentals of a parameter control scheme. The real power of control over parameters lies not in user control, but in a combination of computer and user control. Using the master slaving architecture of Virtual Studio, tools could be constructed which would allow the computer to record and playback the control manipulations of the user. In effect, the computer could act as the user assistant, operating a set of controls while the user manipulates an-

other set. The success of automated mix down in professional recording studios is testimony to the effectiveness of the computer as an audio engineers assistant; what makes computer automation so attractive in Virtual Studio is that while only input faders are under computer control in a professional studio, every parameter in Virtual Studio can be controlled by computer.

We would also like to incorporate the notion of a "snapshot" of the system. By taking a snapshot, the user could record the state of the entire system at any moment and always have the option of returning to that state. We feel this would promote user experimentation; many times users are so relieved to finally get the system configured to a workable state, that they are afraid to experiment with different configurations. Thus, snapshots would allow the user to always go back to a safe state and hence promote experimentation. Currently, Virtual Studio can save the state of its configuration and its Rooms, but it cannot save the state of its devices. We would like to develop a scheme which allows the state of all objects, be they hardware or software, to be saved and restored.

A major hole in the interface of Virtual Studio exists concerning the display master slave relations and connections between devices. Concerning master slave relations, there is no visual method of determining who is slaved to whom. Similarly with connections between devices, a MIDI connection appears the same as an audio connection. We feel that the use of color would be an elegant solution to each of these problems. For the latter problem, different connection types could be displayed in different colors. For the master slaving problem, we feel a scheme where a slave inherits the color of its master would be an elegant solution. Since the prototype version of Virtual Studio was developed on a monochrome display, these schemes could not be explored.

Another problem area in Virtual Studio concerns integrating new types of MIDI controllable devices into the system. The problem is: many MIDI controllable devices have hundreds of parameters; when a new type of MIDI device is added to the system, how can the user specify to the system the MIDI message which should be sent to the device when a knob, slider, etc. on the device's control

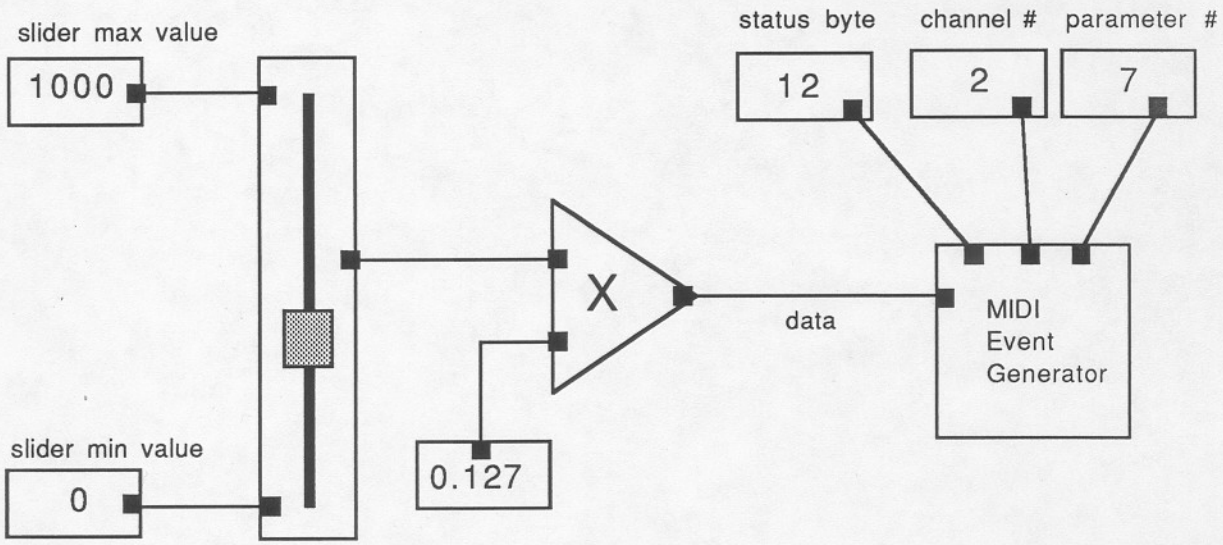


Figure 5.1: A MIDI "circuit": a graphical slider is being used to generate MIDI events. The slider outputs a value between 0 and 1000. This value passes through a multiplier which scales it to the range 0 to 127. The MIDI event generator requires a status byte, a channel number, a parameter number and data to compose a "parameter value" event.

panel changes? Currently only the implementer of Virtual Studio is capable of "installing" new devices, due to the complexity of the specification of the required MIDI protocol. This task is even difficult for the implementer. We feel there is great potential in developing a graphical language to specify how control panels operate. A graphical MIDI event specification language would integrate elegantly into our hierarchical structure scheme; double clicking on a control panel would reveal its internal configuration diagram which would be a MIDI event generator diagram. Figure 5.1 shows a simple example of this type of diagram. The MIDI "circuit" generates controller number 7 values on MIDI channel 2 when the slider is adjusted.

Finally, we feel that Virtual Studio has laid the user interface ground work for a DAW which emulates a "studio in box". The only difference, from a user interface

perspective, between Virtual Studio based on MIDI devices and a version based on a DAW, is that in the DAW case, when a user double clicks on a control panel, rather than revealing a MIDI event generator diagram, a diagram which shows how the device is emulated in software is shown. Thus, we feel the Virtual Studio interface could be ported and extended to provide an interface which reflects the quality of functionality offered by DAWs.

5.3 Final Remarks

It is hoped that the research in this thesis will assist others in developing systems based in the same class of application as our case study or in pushing further the state of the art concerning audio studio user interfaces. Historically, quality of functionality has preceded quality of user interface in the development of systems. We hope that our research will assist in eliminating this time lag. It is clear that personal audio studios of the future, such as DAWs, will have powerful functionality; we hope our research, by contributing to state of the art in user interfaces, will ultimately assist musicians, composers and audio engineers in harnessing this power.

Bibliography

[1] Austin K. (1987). The Zipless Track, *Electronic Musician*, 3(19), 60-65

[2] Bergren P. (1986). Sound Effects: New Tools and Techniques, *Mix*, 10(4), 90-97

[3] Blythe D., Kitamura J., Galloway D. and Snelgrove M. (1986). Virtual Patch Cords for the Katosizer, *International Computer Music Conference 1986*, 359-363

[4] Buxton W. (1983). Lexical and Pragmatic Considerations of Input Structure, *Computer Graphics*, 17(1), 31-37

[5] Buxton W. (1986). Chunking and Phrasing and the Design of Human Computer Dialogues, *Proceedings of the IFIP World Computer Congress, Dublin, Ireland*, Sept. 1986, 1-6

[6] Buxton W., Reeves W., Fedorkow G., Smith K.C., and Baecker R. (1980) A Microcomputer based Conducting System, *Computer Music Journal*, 4(1), 8-21

[7] Card S.K. and Henderson Jr. A. (1987). A Multiple, Virtual-Workspace Interface to Support User Task Switching, *CHI + GI '87 Conference Proceedings*, 3-59

[8] Denning P.J. (1980). Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6, 66-84

[9] *DMP7-Pro Users Manual*, Digital Music Services 23010 Lake Forest Drive, Suite D334, Laguna Hills, California 92653

[10] DMP7 User's Manual, Yamaha Music Corporation, P.O. Box 6600, Buena Park, CA 90622

[11] Galloway D., Blythe D. and Snelgrove M. (1987) Graphical CAD of Digital Filters, *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, June 1987

[12] Gane C. and Sarson T. (1979). *Structured Systems Analysis*, Prentice-Hall, Englewood Cliffs, New Jersey

[13] Gordon J.W. (1985) System Architectures for Computer Music, *ACM Computing Surveys: Special Issue on Computer Music*, 17(2), 191-233

[14] Gould J.D. and Salaun J. (1987). Behavioral Experiments on Handmarkings, *CHI + GI '87 Conference Proceedings*, 175-181

[15] Halasz F.G., Moran T.P. and Trigg R.H. (1987). NoteCards in a Nutshell, *CHI + GI '87 Conference Proceedings*, 45-52

[16] Heinbuch D. (1988) Audio Aspirin for Studio Headache #1, *Electronic Musician*, 4(6), 29-34

[17] Hosaka M. and Kimura F. (1982). Using Handwriting Action to Construct Models of Engineering Objects, *Computer*, Nov. 1982, 35-47

[18] Kantowitz B.H. and Sorkin R.D. (1983). *Human Factors: Understanding People-System Relationships*, John Wiley and Sons, New York, ISBN 0-471-09594X

[19] Kitamura J., Buxton W., Snelgrove M. and Smith K.C. (1985). Music Synthesis by Simulation of using a General Purpose Signal Processing System, *Proceedings of the International Computer Music Conference, Vancouver*

[20] Konneker L. (1984). A Graphical Interaction Technique which uses Gestures, *Proceedings of the IEEE First International Conference on Office Automation*, 51-55

[21] Moorer J., Curtis A., Nye P., Borish J. and Snell J. (1986). The Digital Audio Processing Station: A New Concept in Audio Postproduction *Journal of Audio Engineering Society*, 34(6), 454-463

[22] Myers B.A. (1986). Visual Programming, Programming by Example, and Program Visualization; A Taxonomy, *Proceeding of SIGCHI'86: Human Facters in Computing Systems*, 13-17

[23] Newell A., McCraken D.L., Robertson G.G. and Akscyn R.M. (1984). ZOG and the USS CARL VINSON, *1980/81 CMU Computer Science Research Review*, Pittsburgh PA, Carnigie Mellon University, 95-118

[24] Osteen G. (1988). Synchronized Recording with Virtual MIDI Tracks, *Electronic Musician*, 4(2), 62-69

[25] Product of Interest: WaveFrame Audio Frame, *Computer Music Journal*, 12(2), 87-88

[26] Q-Sheet User's Manual, Digidesign, 1360 Willow Road, Suite 101, Menlo Park, CA 94025

[27] Rubinstein R. and Hersh H. (1984). *The Human Factor*, Digital Press, Digital Equipment Corporation, Massachusetts, ISBN 0-932376-44-4

[28] Rhyne J. and Wolf C. (1986) Gestural Interfaces for Information Processing Applications, *IBM Research Report RC-12179*, T. J. Watson Research Center, IBM Corporation

[29] Schwartz D. (1984). Specifications and Implementation of a Computer Audio Console for Digital Mixing and Recording, *76th AES Conference*, preprint 21139 (A-3)

[30] Snell J. (1982) The Lucasfilm Real-Time Console for Recording Studios and Performance of Computer Music, *Computer Music Journal*, 6(3), 33-45

[31] VanCleemput W.M. (1980) Hierarchical VLSI Design, *IEEE CompCon80, VLSI: New Architectural Horizons*, 83-87

[32] Ward J. and Blesser B. (1985) Interactive Recognition of Handprinted Characters for Computer Input, *IEEE Computer Graphics and Algorithms*, Sept. 1985, 24-37

[33] Wasserman A.I. and Shewmake D.T. (1982) Rapid Prototyping of Interactive Information Systems, *AMC Software Engineering Notes* 7(5), 171-180

[34] Wirth, N. (1971) Program development by stepwise refinement, *CACM* 7(5) 14(4), 221-227

[35] Wolf C. (1986) Can People Use Gesture Commands? *ACM SIGCHI Bulletin*, 18(2), 73-74

[36] Yavelow C. (1987) Personal Computers and Music: The State of the Art, *Journal of Audio Engineering Society*, 35(3), 160-188

Appendix A

Gesture Recognition

This appendix describes the gesture recognition algorithms used in Virtual Studio. As described in Chapter 4, gestures are input by depressing a mouse button, moving the mouse and then releasing the mouse button. In our system, we consider a gesture to be the motion that occurs between the mouse button down and button up. There are no commands which require more than one gesture to invoke. Thus, once mouse button up is detected, the system analyzes the gesture, and if it corresponds to a command, carries out that command.

A.1 Gathering Input

When a mouse button is pressed in Virtual Studio, the system checks if the cursor is over a direct manipulation hot spot. If it is, then the routine to handle direct manipulation for that hot spot is called. Otherwise, the gesture handler is called. The gesture handler begins sampling the x,y position of the mouse cursor at approximately 100Hz. If the location of the cursor changes between samples, the new location is recorded in a list of x,y positions called **RawPoints** and a black line is drawn on the screen from the previous location of the cursor to the new location. Thus giving the effect that the cursor is leaving an "ink trail". Once the user releases the mouse button, the gesture handler stops sampling, recording and "ink trailing" the cursor location, and passes the list of points, **RawPoints**, to gesture

the preprocessor.

A.2 Preprocessing the Gesture

The gesture preprocessor is responsible for:

- Filtering out "noise" in the gesture.
- Reducing the amount of data to be analyzed.
- Creating from the data a list of line segments.

Filtering the gesture involves resampling **RawPoints** according to distance between points. A new point is added to list called **FilteredPoints** if it is at least 16 pixels from previous point added to **FilteredPoints**. The filtering process is started by placing the first point in **RawPoints** into **FilteredPoints**. This process effectively removes noise in the data caused by small "jitters" in the cursor's movement and reduces the number of data points.

The next step is to create a list of line segments. Each pair of points in **FilteredPoints** is considered as a line segment: its slope is calculated and stored in a list of called **Slopes**, and its approximate direction is recorded in a list called **Direction**. The direction of the line segment is quantized to 6 values: right, left, upRight, upLeft, downRight and downLeft.

Finally the gesture preprocessor records information about the gesture such as its starting and ending points, and its extent. Figure A.1 shows an example of preprocessing.

A.3 Analyzing the Gesture

The essential process used to recognize gestures is scanning the line segments for articulation points based on direction and position. A gesture is scanned from the first point sampled. The analyzer algorithm first checks if the gesture begins with

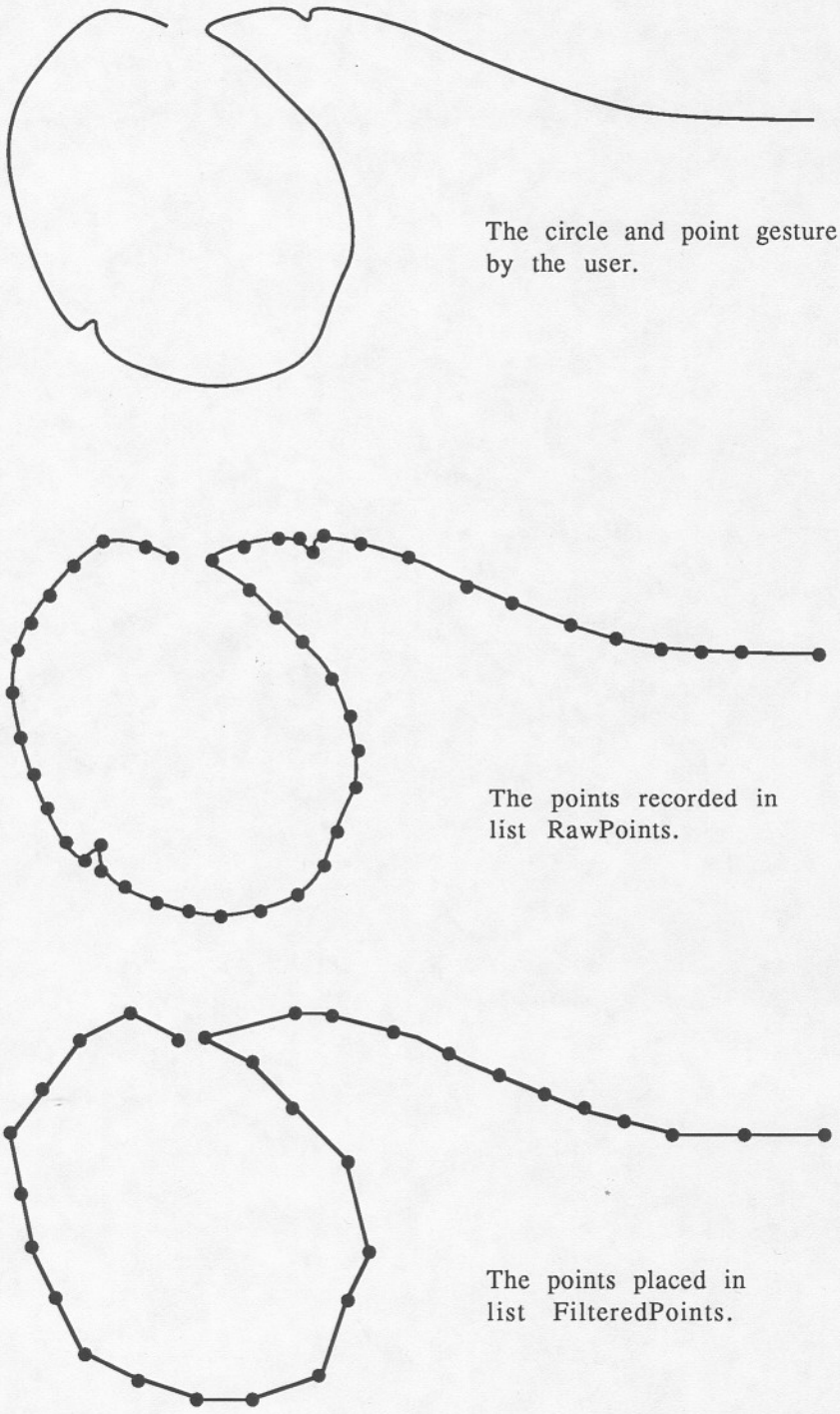


Figure A.1: Preprocessing a gesture before analysis. Preprocessing removes jitters in the user's movement and reduces the amount of data. The slope and approximate direction of each line segment is then computed and recorded for later use by the gesture analyzer.

scoping circle. In reality, we begin scanning the gesture for a polygon of some minimum size. The algorithm is:

- Starting at the first point, scan forward till we are the minimum polygon size away from the first point.
- Continue scanning until a point very close to the first point is found.
- Extract this part of the gesture as the scoping polygon. Extract the remaining part of the gesture as the "tail".

If a polygon is indeed found, the "tail" part is analyzed:

- If there is no tail, then the gesture is marked as a "circling gesture".
- Otherwise, if there is a tail:
  - If the tail terminates within the scoping polygon, then the gesture is marked as a "circle tail in" gesture.
  - Otherwise, if the tail is shaped like the letter C at its end, it is marked as a "circle with C tail" gesture.
  - Otherwise, if the tail has an inverted letter T at its end, the gesture is marked as a "circle with inverted T tail" gesture.
  - Otherwise, we do not expect any other special tails, so we mark the gesture as the "circle with tail" gesture.

If no polygon is found, then it is possible the gesture is a simple symbol such as a delete or "inverted T" gesture. In this case the gesture is checked once for each symbol recognized. The general approach of the recognition process is to first analyze the size of the gesture (for example, symbols are generally not terribly large, or small), then the shape of the gesture. For example, when testing for the "inverted T" gesture, the list of line segment directions, **Direction**, is scanned as follows:

- Scan **Direction** while the line segments are going down left or down right: If the line segment slope's absolute value is less than 0.5, this can't be a "inverted T" gesture —exit.

- Scan **Direction** while the line segments are going left, down left or up left: If the line segment slope's absolute value is greater than 0.5, this can't be a "inverted T" gesture —exit.

- Scan **Direction** while line segments are going right, down right or up right: If the line segment slope's absolute value is greater than 0.5, this can't be a "inverted T" gesture —exit.

- Mark as "inverted T" gesture.

Finally, if the gesture is not recognized as a symbol or a circling gesture, it is marked as "unknown". If it is recognized, it is marked according to its symbol type and passed to the gesture interpreter

A.4 Interpreting Gestures

The interpretation of a gesture depends on the context of the gesture. For example, the "circle with tail" gesture may imply different commands depending on what is circled and pointed to. If the user circles a group of device icons in configuration window, this gesture then implies a "move" command is required. Alternatively, if the user circles a group of controls and points to another control, this implies a slave command is required.

The general algorithm for interpreting gestures is as follows: First, gestures which are independent of context by definition have there corresponding command carried out directly. Otherwise, depending on the type of the gesture, a gesture's scope and arguments are determined. The combination of the gesture's type, scope and arguments are used to determine the command called. Essentially, a case statement which checks for all possible command types is used. For example, the check the for "slaving" command type is:

- If the gesture is a "circle with tail".

- And there are controls in the circle.

- And the tail ends in another control.
- Then call the slave command with the circled controls and the pointed to control as arguments.