An Interactive Multimedia Tutoring System For Digital Signal Processing

Sam Pointer

Computer Sciences Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

M. S. project report

March 1990

Table of Contents

1. Introduction 1

 1.1. The New Media 1

 1.2. Teaching DSP to Non-Specialists 2

2. The DISIPLE System 3

 2.1. Goals 3

 2.2. Implementation 4

  2.2.1. Platforms Used 4

  2.2.2. Sample User Session 9

   2.2.2.1. Description of Figure 9 11

   2.2.2.2. Description of Figure 10 11

   2.2.2.3. Description of Figure 11 11

  2.2.3. Code Examples 15

   2.2.3.1. Mathematica 15

   2.2.3.2. Communicating With Mathematica . . . 17

   2.2.3.3. Interface Builder and Objective-C . . . 20

   2.2.3.4. The Sound Kit 27

   2.2.3.5. The Music Kit 28

 2.3. Evaluation of DISIPLE System 29

3. Evaluation of Tools and Capabilities 32

4. Conclusions and Future Directions 35

Appendix A. Syllabus and Notes For DSP Course

Appendix B. The DISIPLE Mathematica Notebook (not included in tech. report)

Appendix C. Distribution and Execution Instructions

References

Acknowledgements

This material is based upon work supported under a National Science Foundation Graduate Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

Thanks to my advisors, John Wawrzynek and David Wessel, for their help and encouragement.

Thanks to the National Science Foundation for supporting me financially.

Thanks to Katy Linn for supporting me personally.

Thanks to Bobby Weinapple for putting up with a terminal in our living room.

Thanks to the Snakes in the Grass for sharing me with graduate school.

Thanks to the New Potato Caboose for rescuing me from graduate school.

Thanks to the coffee-producing nations for their major impact on this work.

Thanks to my parents for their major impact on my life.

An Interactive Multimedia Tutoring System For Digital Signal Processing

Sam Pointer†

University of California at Berkeley
Berkeley, CA 94720

ABSTRACT

We have developed an interactive multimedia software system to act as a tutor of digital audio signal processing. The DISIPLE system (DIgital SIgnal Processing Learning Environment) is intended for use by people without specialized engineering training, such as composers and psychoacousticians. We have implemented a prototype of the system on the NeXT computer to 1) explore the relationships among text, graphics, and sound in an interactive teaching environment, and 2) explore the tools available on that platform and the feasibility of developing the full systems. In this paper we concentrate on our use of tools to implement the interactive multimedia system rather than our specific approach to teaching individual topics. Our system is organized as a Mathematica notebook with external support from NeXT's Sound and Music Kits through Objective-C code. The environment employed is standard, making our system portable to any NeXT machine. This environment, while providing many programming conveniences, has presented some interesting implementation challenges. With the DISIPLE system we have illustrated some possibilities for developing interactive multimedia systems on the NeXT machine, and for teaching topics in audio technology using interactive multimedia. The experience gained in its development will be useful in developing related systems. Although we have concentrated on the domain of digital signal processing, these techniques can be used in other domains in music and music technology.

† This work was supported by NSF grant number MIPS-8958568.

single medium of expression such as text, sound, or animation, but instead utilizes two or more media. An "interactive" system is one in which the user can affect the behavior of the system, often by choosing among alternatives or choosing when s/he is ready to proceed to a later section. Researchers have explored educational uses of interactive multimedia in a variety of fields [Ambron 88, Freed 89].

Topics in audio technology are particularly suited to multimedia presentation, because they inherently involve sound and are normally taught using a combination of text and graphics. In addition, because they involve many phenomena which are non-stationary and evolve over time, the ability to use animation (non-stationary graphics) is useful as well. Understanding these topics often means the ability to leap freely between the realms of mathematics and sound; we hope to promote that understanding with an interactive presentation format allowing the user to change the representation in one realm and see the effect in another. For example, a user might design a filter graphically by dragging poles and zeroes in the z-plane, and in real time hear his/her own recorded voice though that filter and see the spectrum of that filtered output graphed.

Teaching DSP to Non-Specialists

Computer technology has permeated every aspect of music synthesis, composition, and psychoacoustics, yet many users of the new technology lack the knowledge to exploit it. It will never be possible for a user to achieve as much with a predefined set of capabilities (chosen by some equipment manufacturer) as with an understanding of the fundamentals behind them. The field at the heart of these capabilities is digital signal processing (DSP); thus, even people without engineering and computer science backgrounds— composers wishing to explore new sounds, psychologists wishing to develop psychoacoustic experiments, and architects wishing to model the acoustic properties of a proposed building—will need to understand the basics of DSP in order to harness the immense power of computers for their purposes. Unfortunately, DSP is a difficult subject. Many engineering students find digital signal processing one of the most difficult topics in their curriculum. The traditional way of teaching DSP provides little insight into practical

applications, and is nearly useless to those without an engineering background. Although these topics may seem difficult when presented in textbook form, we believe that they will become clear and intuitive with the use of examples using animated graphics and sound. To this end, we developed a syllabus for our interactive "course" in DSP (based on rather accessible articles on digital audio signal processing by Richard Moore [Moore 85] and Julius Smith [Smith 85]) and implemented several major sections.

The DISIPLE System

We will now present the DIgital SIgnal Processing Learning Environment. We will list our design goals for DISIPLE, discuss its implementation, and evaluate it against its design goals.

Design Goals

Our design goals for DISIPLE were:

- To structure it as a textbook with a syllabus. Although the user may revisit topics at any time, later chapters may depend on assimilation of earlier ones.
- To use sound, graphics, and animation where appropriate to illustrate topics.
- To allow the user to vary aspects of the demonstrations which help illuminate the topics being addressed.
- To allow the user to explore the interrelationship among topics via the interactive demonstration environment.
- To make it easy and convenient to interact with.
- To use primarily graphic interfaces in the interactive demos.
- To develop an expanding demonstration environment, adding a topic at a time, ending with an environment where the user can choose non-trivial combinations of options.
- To use standard DSP terminology and graphical conventions.

- To make it somewhat portable by encapsulating machine-dependent portions of the implementation.
- To make it fun to use, so as to hold the user's interest.

Implementation

We will describe the major tools and platforms used by DISIPLE, show snapshots of a sample user session, then discuss the implementation of the system. We will not go into detail on our reasons for choosing to present particular topics or for presenting them in the ways we did. See Appendix A, the DISIPLE syllabus, and Appendix B, the DISIPLE Mathematica notebook itself, for our presentation of topics.

Platforms Used

The major platforms used in the DISIPLE system are the NeXT machine, the Mathematica system, NeXT's NextStep development environment, NeXT's Sound and Music Kits, and Objective-C code.

The NeXT machine was a natural hardware choice for our system. It offers a large screen, a graphic user-interface, software tools for building graphic interfaces to applications, software tools for use with sound, the Mathematica system as standard equipment, and digital signal processing hardware as standard equipment. The only other machine with a user-interface suitable for computer novices was the Macintosh, but Mathematica and DSP hardware are not standard on the Macintosh. That difference provided a strong case for the NeXT, since building the system in a standard environment would widen its potential audience. We also wanted to explore the multimedia capabilities of the NeXT, which are relatively untested compared to those of the Macintosh.

Much of DISIPLE's functionality, as well as its primary presentation format, comes from the Mathematica™ system from Wolfram Research [Wolfram 88]. Mathematica is a general system

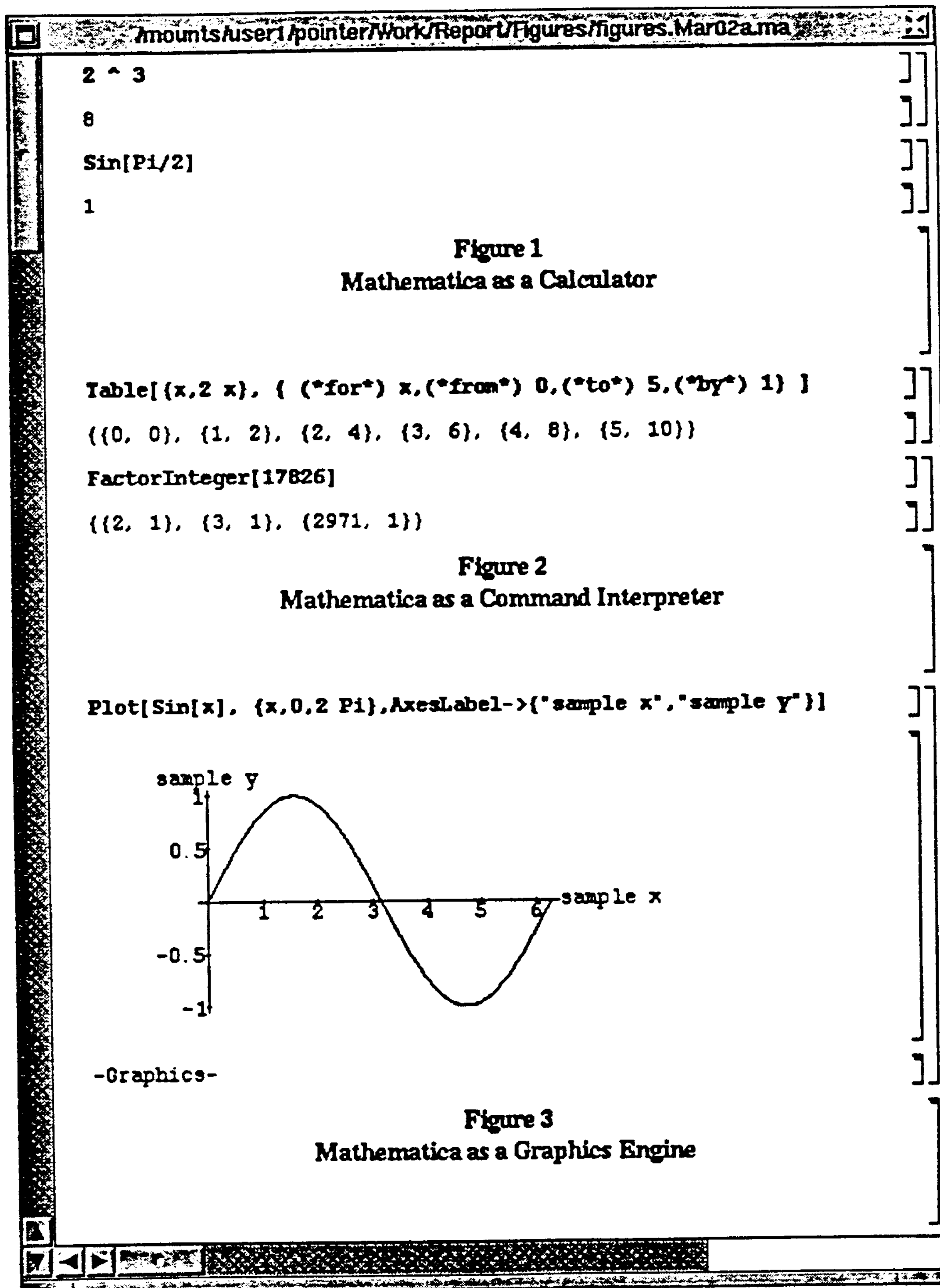
for doing mathematical computation which can be used as a glorified calculator, a engine for graphics and animation, a programming language, and a document processor. DISIPLE uses Mathematica for all of these things, and we will illustrate these capabilities briefly.

The Mathematica kernel is used as a calculator to evaluate mathematical expressions (Figure 1). It is more powerful than traditional calculators, though, with an extensive built-in vocabulary of commands (Figure 2). The kernel also can create graphics (Figure 3), with numerous options available for specifying visual aspects of the display.

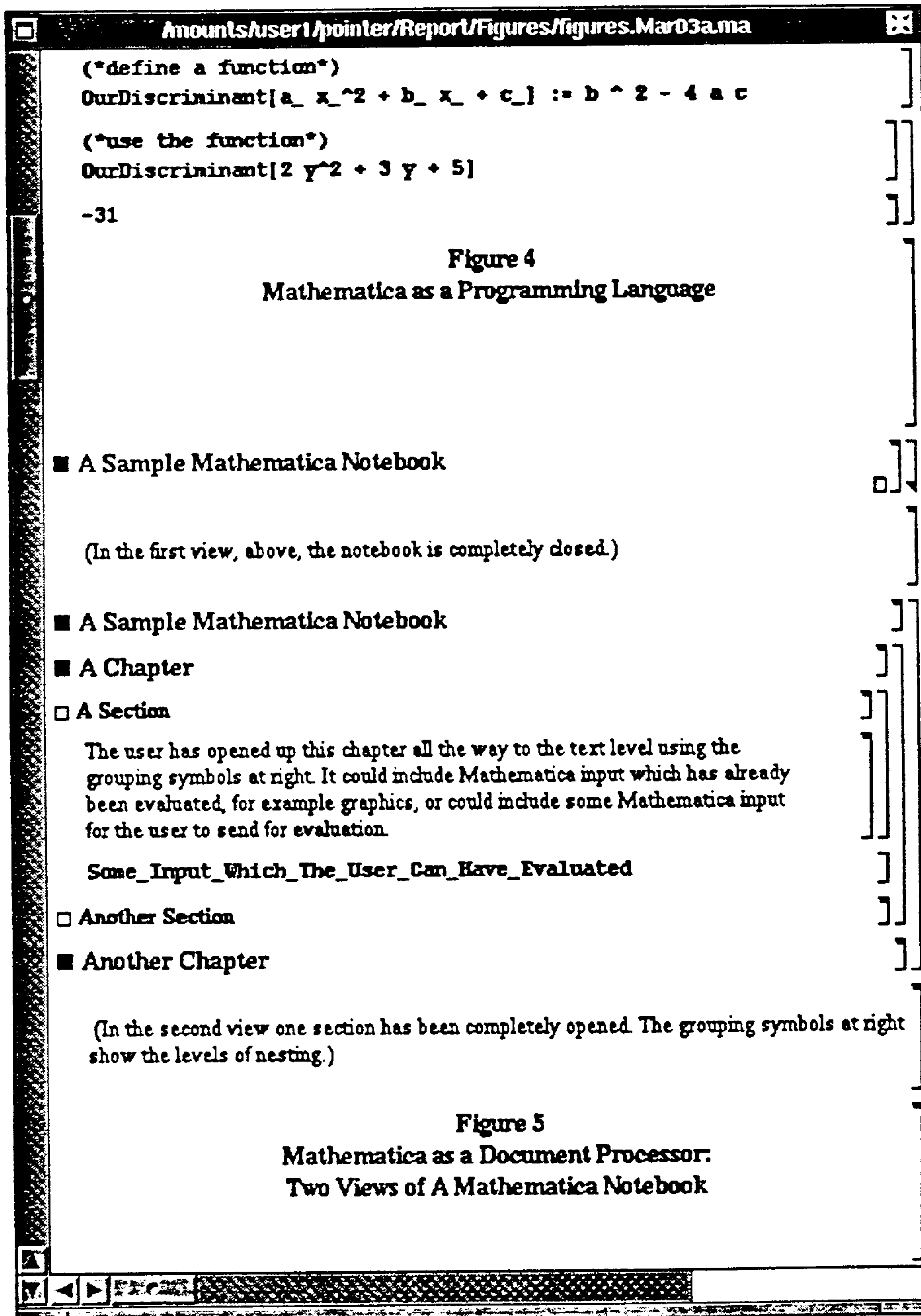
Mathematica can be used as a general-purpose programming language (Figure 4). It allows programming in imperative, functional, and declarative styles.

The Mathematica front end has two useful capabilities. First, it allows animation of graphics, which we will show in the examples. Second, it provides a way of organizing the presentation of information: the Mathematica notebook. The author of a Mathematica notebook can group his/her presentation (which may combine text, graphics, and Mathematica input) into chapters and sections which the user can open and close at will (Figure 5)). Information in a notebook is static, and is only treated as input to Mathematica when the user specifically asks for an expression to be evaluated. We included examples where the use would send a fixed expression to Mathematica (Figure 6), and some where the user could textually modify a Mathematica input string and send it for evaluation (Figure 7). However, in general we avoided requiring the user to remember or even understand the syntax of commands. We considered Mathematica functions tools for our use in implementing actions the user could choose graphically.

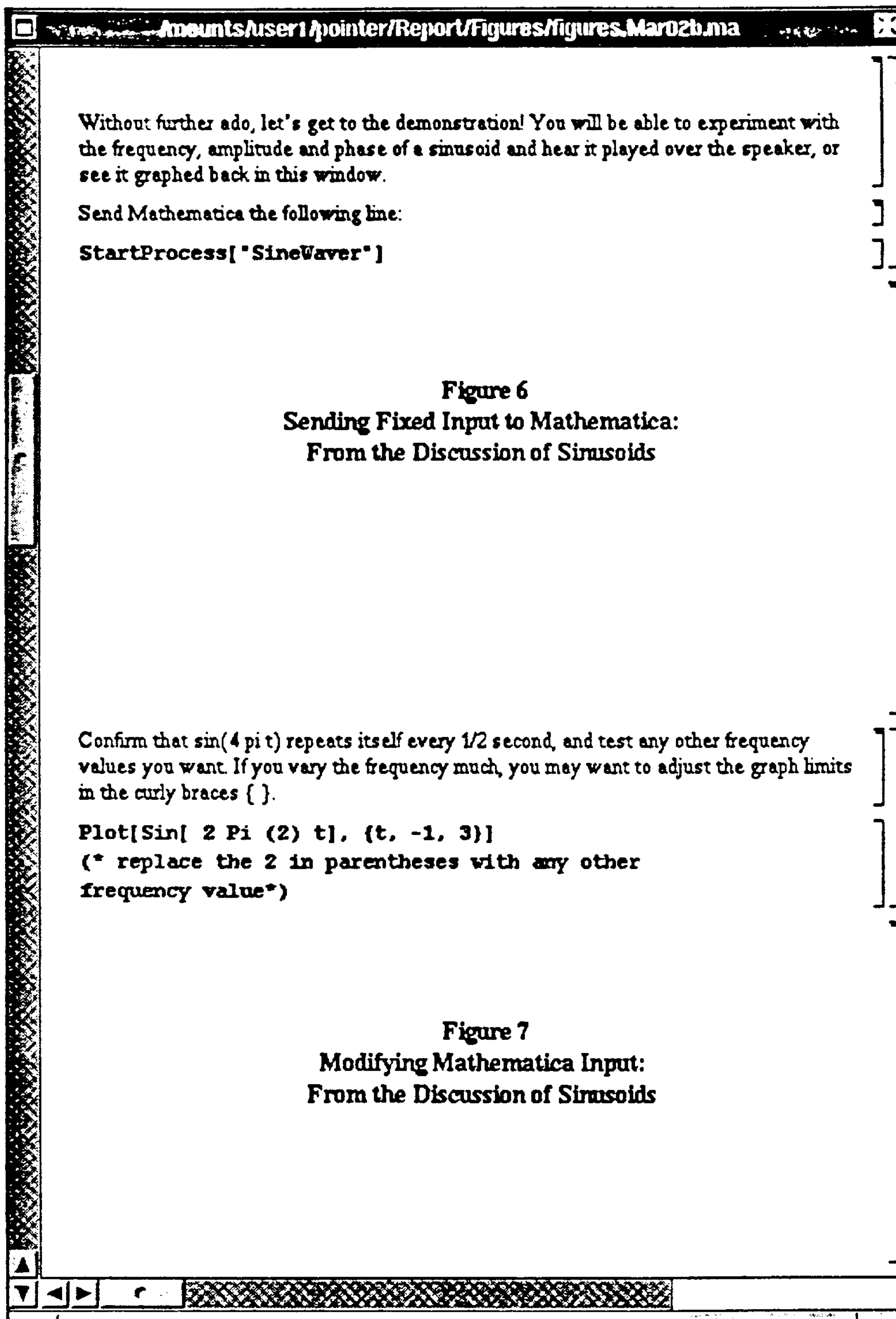
Since Mathematica input must be textual, there is no way to develop graphic user interfaces within Mathematica. For these, we turned to NeXT's application development environment, NextStep (which includes a window server, an Application Kit of pre-defined software objects, and Interface Builder, an application which aids user-interface development). With little more effort than writing a command-line-driven program, NextStep allows the development of an application environment suitable for the naive user. It also aids the programmer in developing Objective-C programs by automatically generating makefiles and code templates.



Figures 1-3
Mathematica as a Calculator, Command Interpreter, and Graphics Engine



Figures 4-5
Mathematica as a Programming Language and Document Processor



Figures 6-7
Mathematica Input

Objective-C is a C-based object-oriented programming language [Cox 87], the programming interface to NextStep and consequently the implementation language for the demonstrations in DISIPLE. In the object-oriented paradigm, the developer defines *classes* (blueprints for self-contained software entities which have variables called *instance variables* and can perform certain actions called *methods*), and then builds an application using instances of these classes, or *objects*. A class which is a *subclass* of another class inherits its methods and instance variables and may also extend or augment the behavior of the parent class. Objects communicate by sending and receiving *messages* to execute methods.

Interface Builder allows much of this object-oriented development to be done graphically, via the *target-action* paradigm. The programmer can graphically connect a user-interface object to another object, its *target*, and choose which of the target's methods should be executed (the *action*) whenever the user-interface object is triggered (for example, when a button is pushed). By defining a custom object to have more than one *outlet* (a special type of instance variable), this new object can be connected graphically to multiple other objects.

While NextStep and Interface Builder allow much of the interface design and the connecting of objects to be done graphically, one must still write the code for the specialized behavior desired from the objects under development. Most of our custom code in Objective-C served either to build up strings to send to Mathematica for evaluation, or to implement sound examples using NeXT's Objective-C class libraries, the Sound and Music Kits.

The Sound Kit provides classes for recording, playing, and editing sampled sounds. The Music Kit provides classes for creating and modifying computer-generated sounds via the DSP chip. We avoided programming directly in DSP code, although the NeXT offers that capability. The classes provided by the Sound and Music Kits are documented in [NeXT 89].

Sample User Session

A user invokes DISIPLE by double-clicking on its icon from NeXT's graphic menu (as with any other NeXT file). S/he is presented with the Mathematica notebook as in Figure 8, and opens

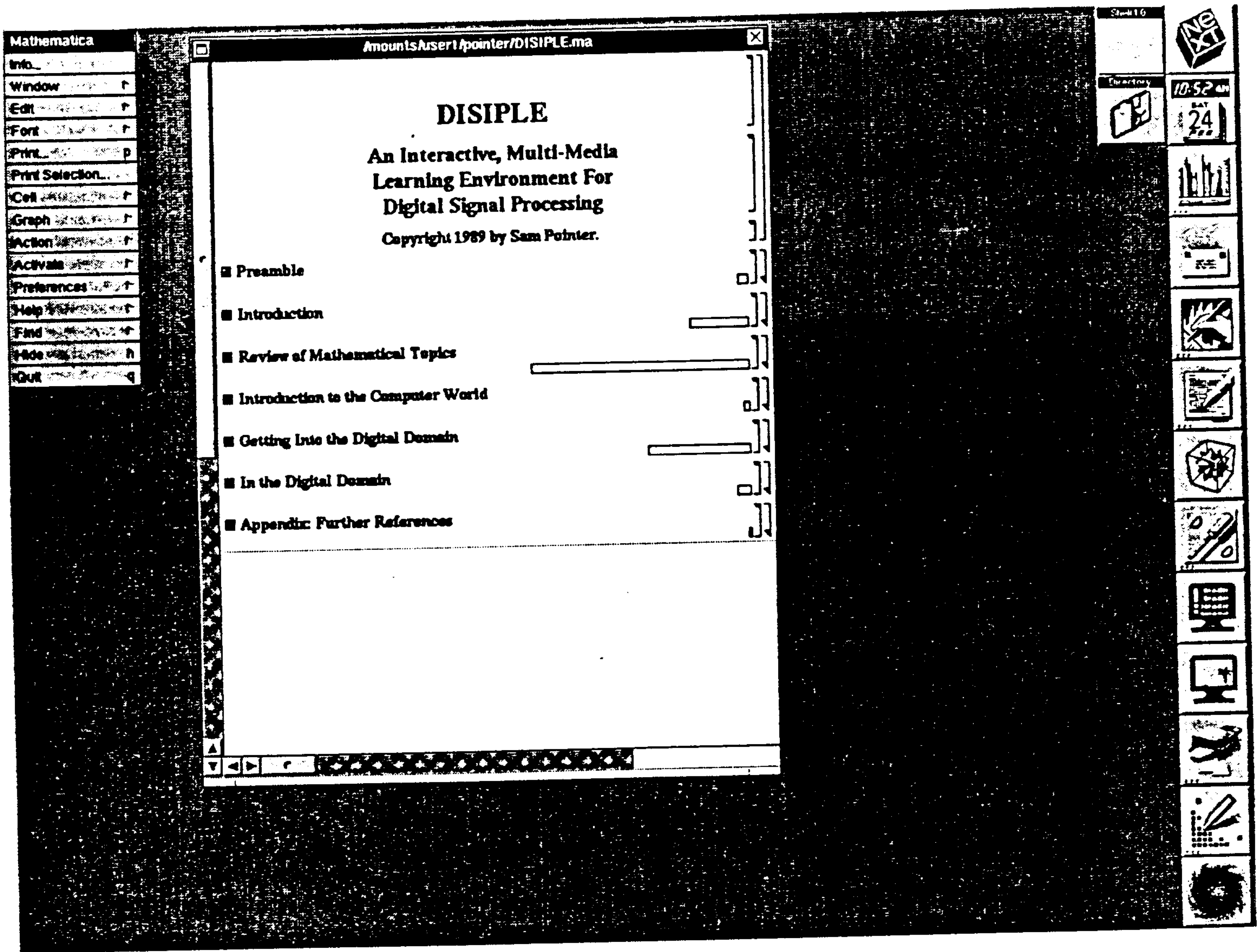


Figure 8
Sample User Session: DISIPLE At Invocation

the notebook to any desired chapter. The user may wander through the notebook at will, although the course is designed to be used sequentially. Topics are generally presented as static text and graphics, as in a textbook, with the addition of interactive demonstrations such as the following.

Description of Figure 9. The user has finished reading an introduction to sinusoids, presented through text and static graphics in the Mathematica notebook at left. S/he now highlights and sends a command to start an interactive demonstration. A graphic interface window appears at right. The user chooses parameters of the sinusoid using the sliders at the top of the interface window. S/he clicks on the "Play" button to hear the sinusoid through the speaker, or on the "Graph" button to see it graphed back in the Mathematica window. Two sample graphs are shown.

Description of Figure 10. The user has been learning about quantization. The interface panel at top right allows him/her to bring up an existing sound file, or record his/her voice to create a new one, then play it by pressing the "Play" button. S/he uses the slider to choose the number of bits of quantization, in this case three, then clicks on the "Quantize" button to have the sound quantized. Now playing the sound will play a three-bit version. Both versions are displayed graphically in the windows at bottom.

Description of Figure 11. The user has been learning about the discrete Fourier transform. This example allows the user to choose the sampling and quantization of an input sinusoid and see the digitized signal or its DFT graphed in Mathematica.

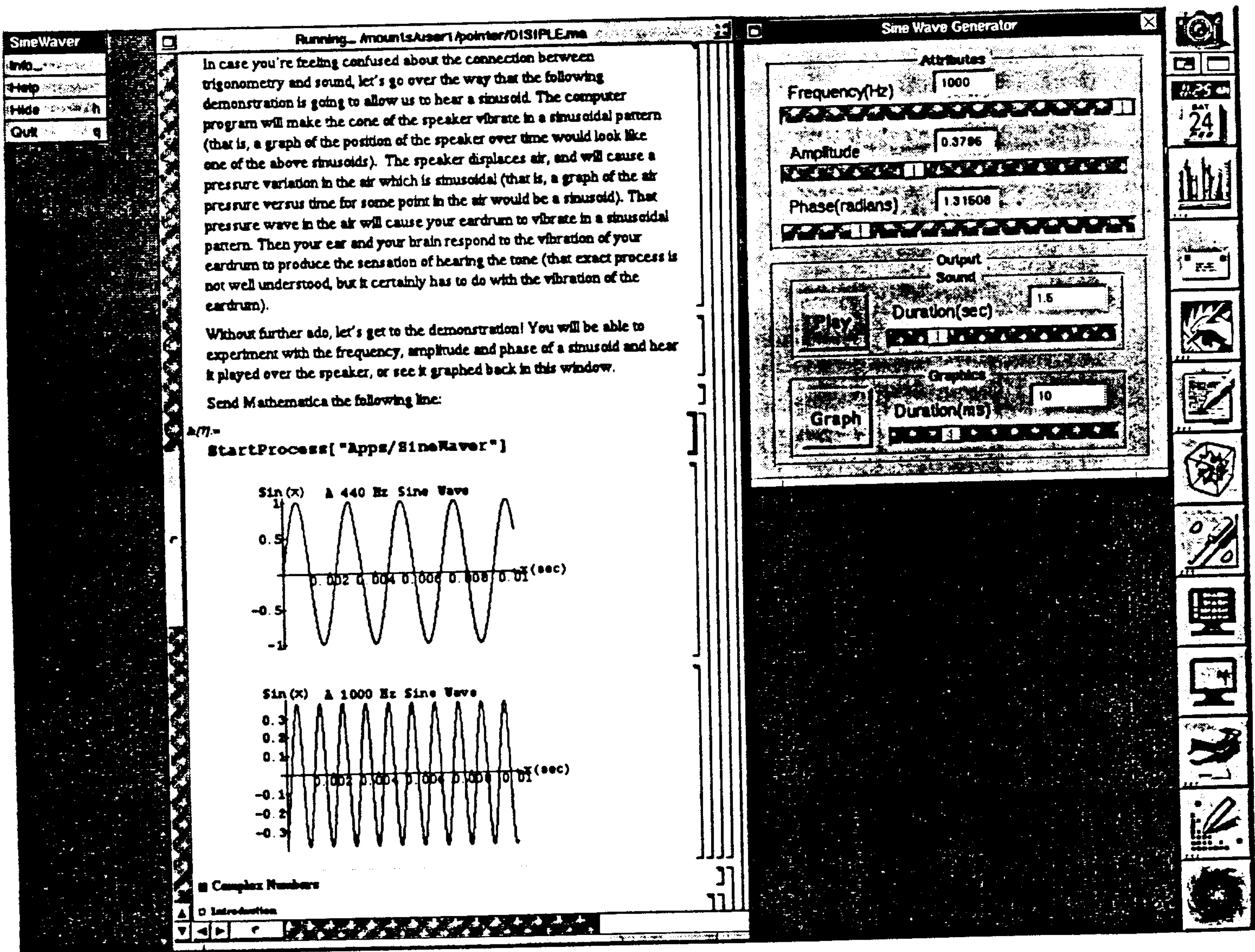


Figure 9
Sample User Session: Sine Wave Generator

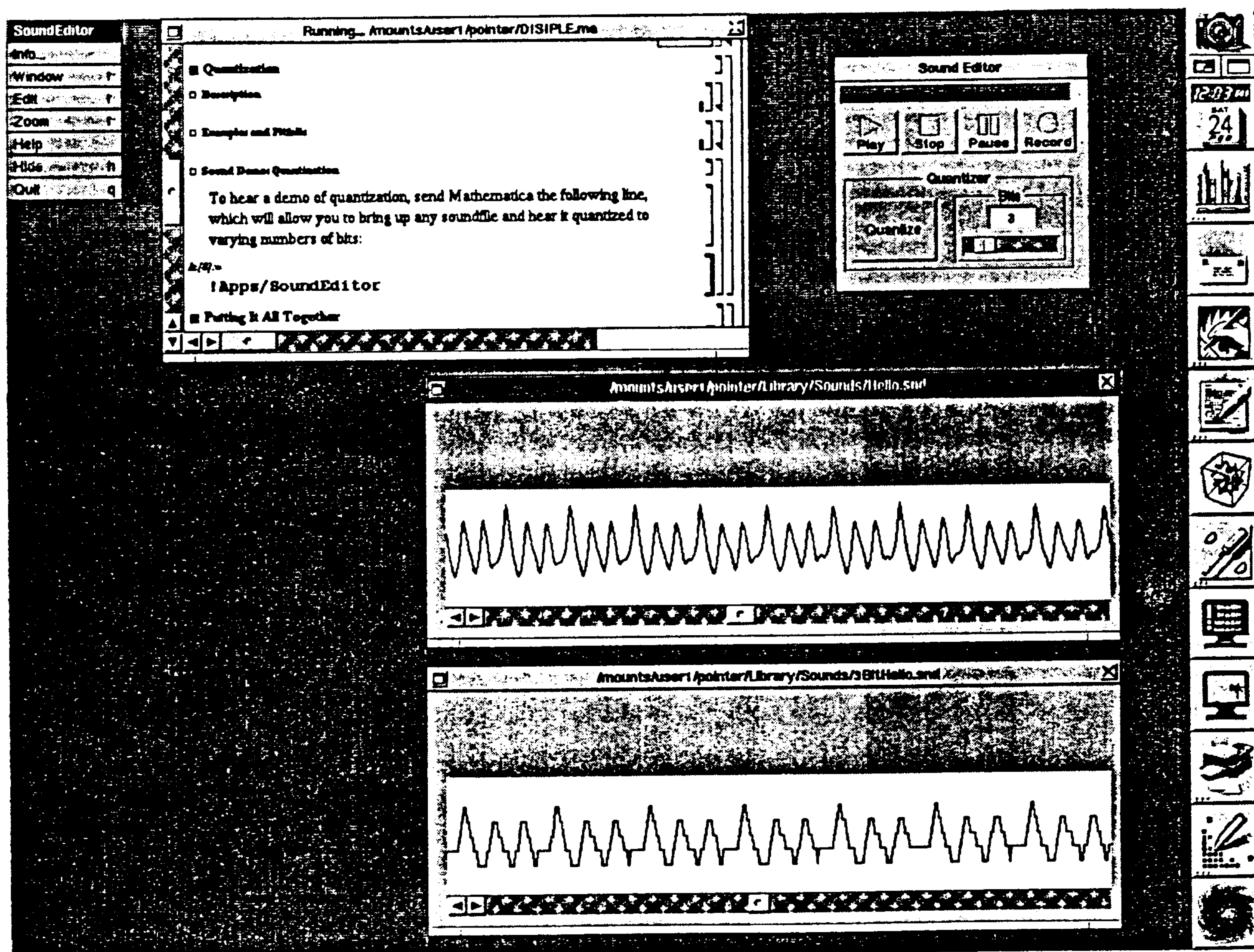


Figure 10
Sample User Session: Quantizing Sound Editor

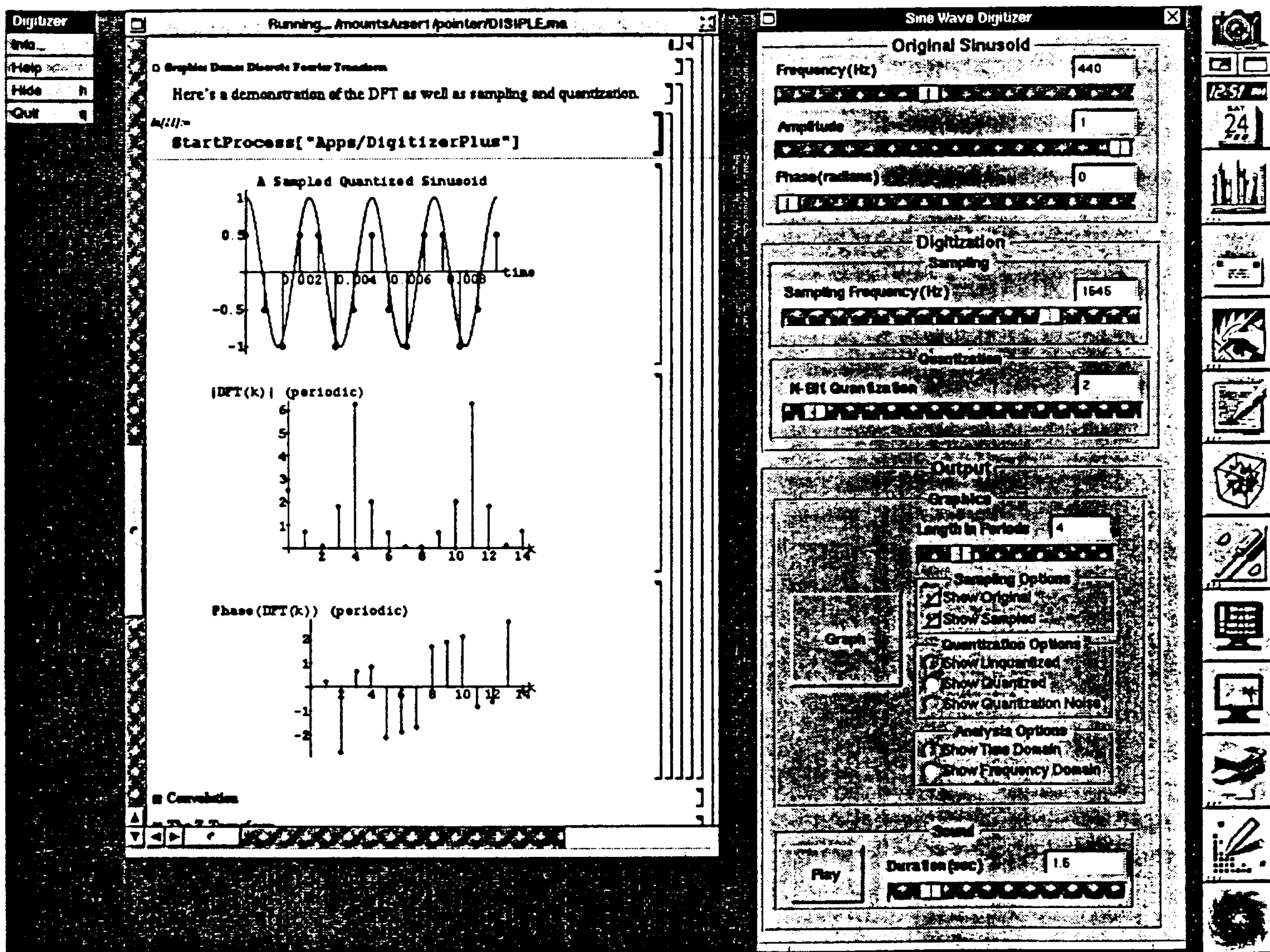


Figure 11
Sample User Session: Digitizing and the DFT

Code Examples

We will now present examples of how each of the above software platforms were used in implementing demonstrations such as the above examples.

Mathematica. Most of the functions which implement topics in digital signal processing or create special graphics are written in the Mathematica language. Of these Mathematica functions, most are written in a procedural style, for example the code to quantize input to an arbitrary number of bits (Figure 12). The specialized graphic functions are written procedurally using primitive graphic operations. Although the syntax can be confusing, and the execution speed slower, Mathematica allows most of the programming constructs of other procedural languages.

```
Quantize[list1:{{_,_}..},numBits_] :=
(*  Quantize takes a list of input pairs of the form
(sampleNumber, sampleValue) and returns a similar pair with the
sampleValue quantized to numBits bits.
*)
    Block[ {i,x,y,dy,yMin,yMax} ,
(*dy,yMin, and yMax are respectively the smallest increment we
can distinguish between with numBits bits,
the smallest value we can represent, and
the largest value we can represent*)
        dy =  2 ^ (1 - numBits);
        yMin = -1;
        yMax = 1 -dy;
        (*We will create a table to return.*)
        Table[
        (*We're iterating over i.  Tell it to view the ith input
value as the list {x,y} *)
              {x,y} = list1[[i]] ;
(*
Return the list {x, quantized_y}, where y is quantized as
follows:
If y is outside the range (-1,1), the quantized value is yMin or
yMax.  If it's within that range, subtract the part that is
truncated.  Use numerical value N[] so a value will be returned
if the input is symbolic.
 *)
```

```
{x,
     Which[
     y <= -1,
          yMin,
     y >= 1,
          yMax,
     True,
          y - Mod[ N[y], dy]
     ]
},
(*Define the iterator i to go over the length of the
input list.*)
     {i, Length[list1]}
] ]
```

Figure 12
Mathematica Function in Procedural Style

Some of the functions are defined in more of a functional style, for example the function of Figure 13 which demonstrates aliasing of analog frequencies when sampled. In several of the cases, the function does not solve the problem all at once, but instead poses the problem in a simpler way and makes a recursive call on itself.

```
ApparentFrequency[fIn_,fS_] :=
(*ApparentFrequency[fIn,fS] takes two frequencies (in Hertz) and
returns the frequency (in Hertz) that fIn would appear to be when
sampled at fS.  This function is specialized for audio because it
converts all negative frequencies to positive ones.  It is useful
for studying aliasing.*)
    Which[
         (*Which resembles a LISP cond statement*)
         fS < 0, (*case 1*)
              ApparentFrequency[fIn,-fS] ,
         fIn < 0, (*case 2*)
              ApparentFrequency[- fIn,fS] ,
         0 <= fIn <= fS/2, (*case 3*)
              fIn ,
         fS/2 < fIn < fS , (*case 4*)
              fS - fIn,
```

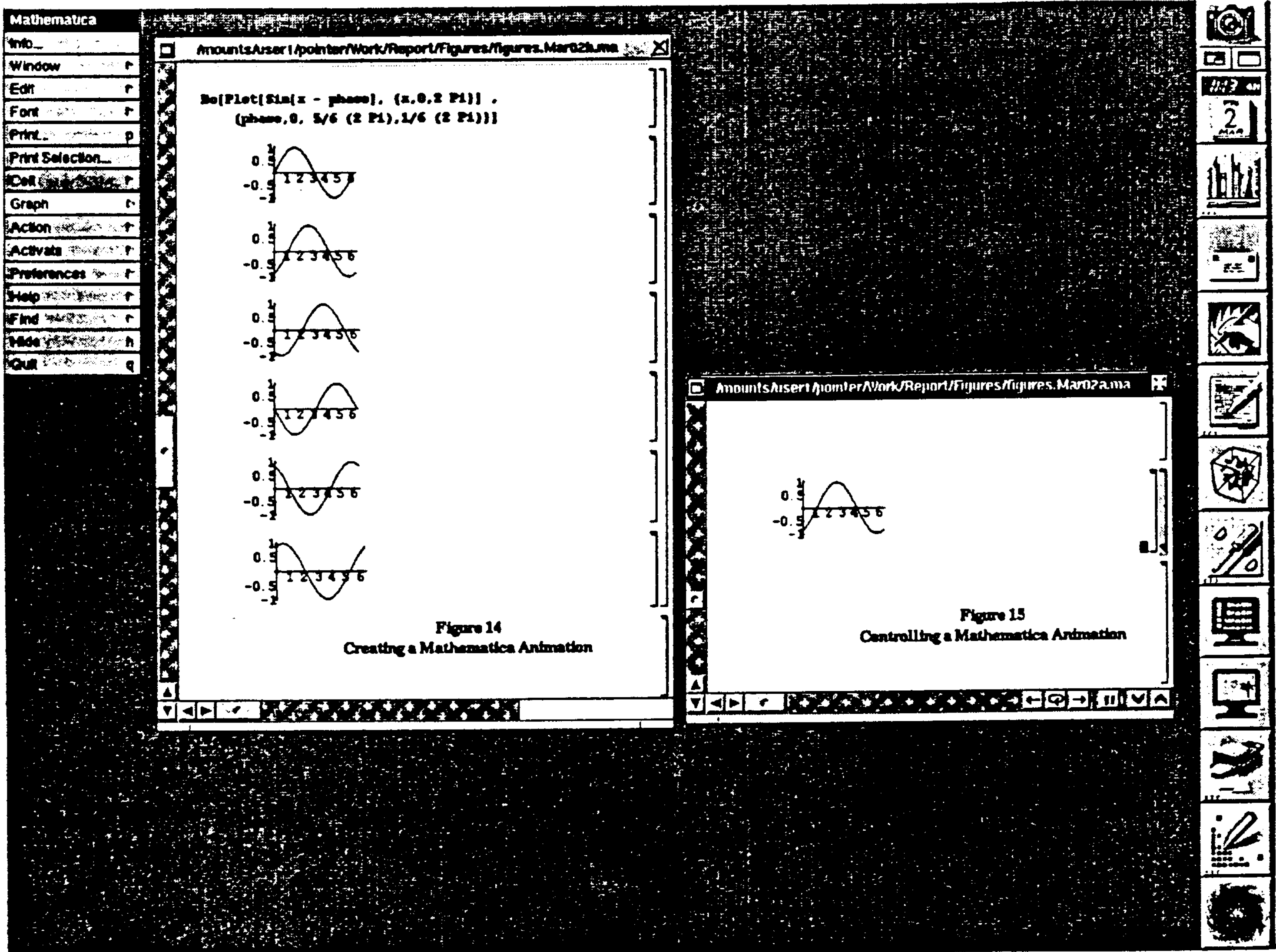
```
fS <= fIn, (*case 5*)
      ApparentFrequency[Mod[fIn, fS],fS] ,
True, (*default case*)
      error_found_in_ApparentFrequency_function]
```

Figure 13
Mathematica Function in Functional Style

The final capability of Mathematica used in DISIPLE is its ability to animate graphics. Creating an animation requires two steps. The first is to create (through normal Mathematica graphics) each still frame of the animation. This is most readily accomplished with iteration, because for our purposes each demonstration was a set of mathematically related plots (Figure 14). The second step is, unfortunately, completely the responsibility of the user: controlling the animation, which is done interactively at run-time. A command-key stroke starts the animation and a series of buttons pop up to control the speed and direction of the animation (Figure 15).

Communicating With Mathematica. In order to use the graphic interface capabilities of the NeXT while maintaining the above capabilities of Mathematica, it was necessary to set up communication between Mathematica and external programs. Mathematica offers several means for doing this, including the ability to install external C functions as Mathematica functions and the ability to call an external function and get back its return value. A addition made after DISIPLE was under development is the MathTalk protocol for communication with external programs. Finally, support is promised in the future for Mathematica as an object in the Application Kit.

Of the communications methods available when we began development, the only one which allowed us to create an interactive demonstration environment was to call external programs which can send Mathematica text strings to evaluate (via interface functions supplied with Mathematica). We would have needed to choose a different method if we had wanted the external program to have access to values returned from Mathematica, but this method allowed us to control Mathematica graphics from an external program with a minimum of effort. Once we had



Figures 14-15
Creating and Controlling a Mathematica Animation

received values from our user for a desired graph, we would simply incorporate them into a string to send to Mathematica, as in the C code fragment of Figure 16. These strings generally invoked one of the specialized graphic functions we had developed in Mathematica.

```
void sinegraph(float amplitude, int frequency,float phase,
    int samplingFrequency,
    int periods, int bits, BOOL showOriginal, BOOL showDigitized,
    int quantizationPlotOption,int analysisPlotOption)
{
(* Here's where we'll store the string.*)
msg[200];
(* This shows how we build a command to send Mathematica
for one particular type of graph.  In the production code, the
Mathematica function invoked depends on the values of the last
four arguments.*)
sprintf(msg,
    (*Here's what we want to send to Mathematica.
    Same format for inserting values as printf.*)
    "PlotSampledAndOriginalCosFragment
    [%d, %d, Phase -> %f, Periods ->%d,
    Bits ->%d,Quantization ->Signal ,Amplitude -> %f] ",
    frequency,samplingFrequency, phase, periods,
    bits,amplitude);
    (*Now send it to Mathematica.  In order for this to work, we
must have called MathInit() at some earlier time.*)
    MathExec(msg);
}
```

Figure 16
C Function For Communicating With Mathematica

Interface Builder and Objective-C. Since Interface Builder and Objective-C provide an integrated environment for application development on the NeXT, we will consider them together in showing how to develop a small application which allows the user to set a frequency value via a graphic interface. This example is necessarily somewhat detailed, and it is important to refer to each figure when it is mentioned. The casual reader could skip to the next section, **The Sound Kit**.

After invoking Interface Builder for a new application, we first built our interface (Figure 17); that is, we resized the interface window, dragged the desired interface objects into it (a slider, a textView, and a title) from the Palettes window at top right, and gave the window a title (*FreqSetter*) using the Inspector Window at right. Next we defined some things about the new application we want to create (Figure 18): we defined a new class in the Classes window at left center, calling it *FreqSetter* and making it a subclass of Application, then used the Inspector window at right to give it the method *changeFreq:* andthe outlets *freqSlider* and *freqViewer*. We needed a *FreqSetter* object in order to proceed, since classes are just blueprints, so (not shown) we made the application we're developing (known as *File's Owner* in the window at bottom left) an instance of *FreqSetter*. Now we made the desired connections to our *FreqSetter* object (Figure 19): we dragged a connection between the *FreqSetter* and the textView in our interface window, then used the Inspector window at right to connect the textView as the *freqViewer* outlet of the *FreqSetter* object. The slider was similarly connected as the *freqSlider* outlet. The last connections we made graphically were to connect the user-interface objects with targets and actions (Figure 20): we dragged a connection between the slider in our interface window and its new target, the *FreqSetter* object, then chose the action *changeFreq:* from the Inspector window.

Saving these files from Interface Builder produced code templates, for which we just needed to fill in the details of the *changeFreq:* method (Figures 21-22).

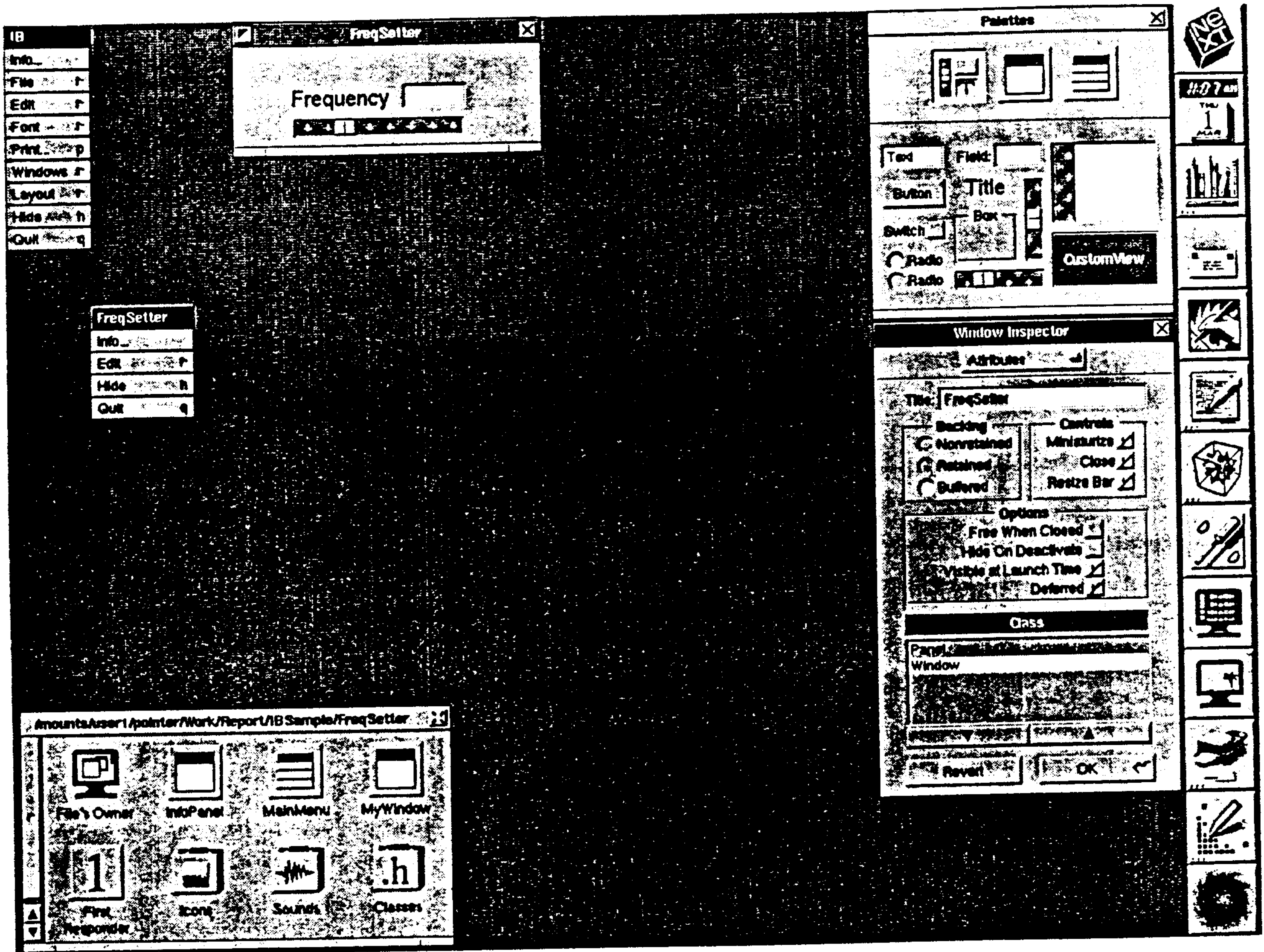


Figure 17
IB Example: Building an Interface

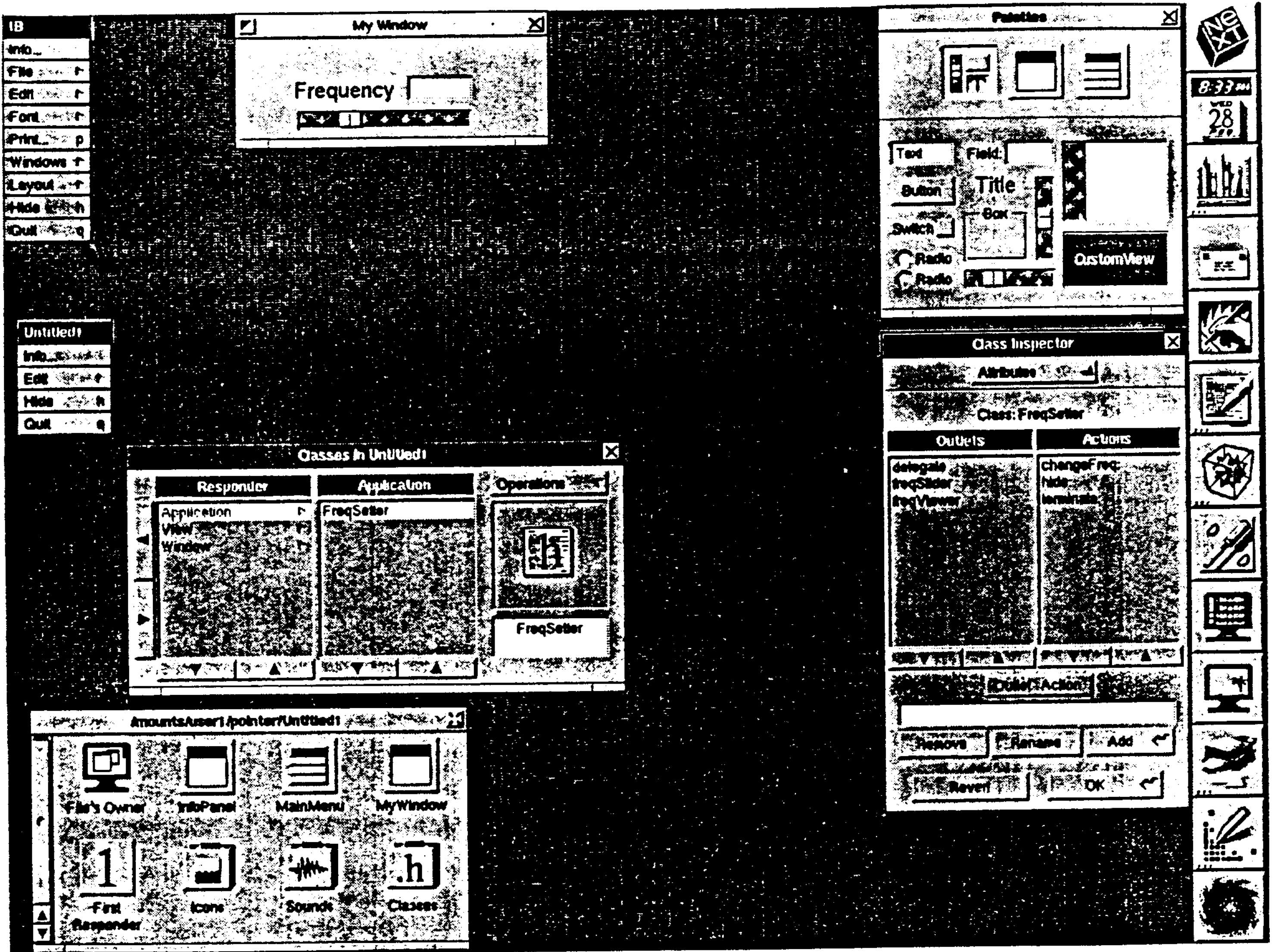


Figure 18
IB Example: Defining a Class

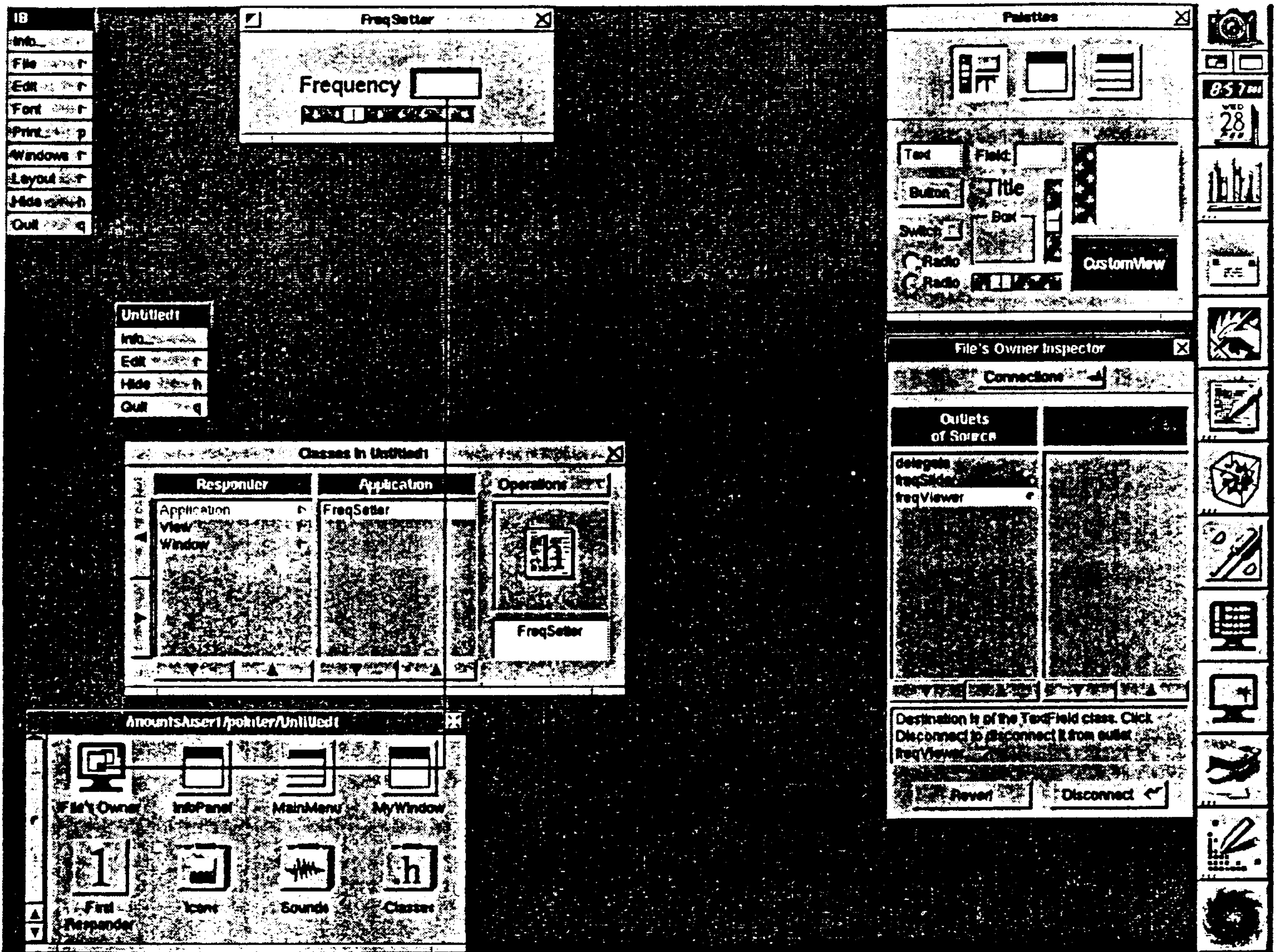


Figure 19
IB Example: Connecting Outlets

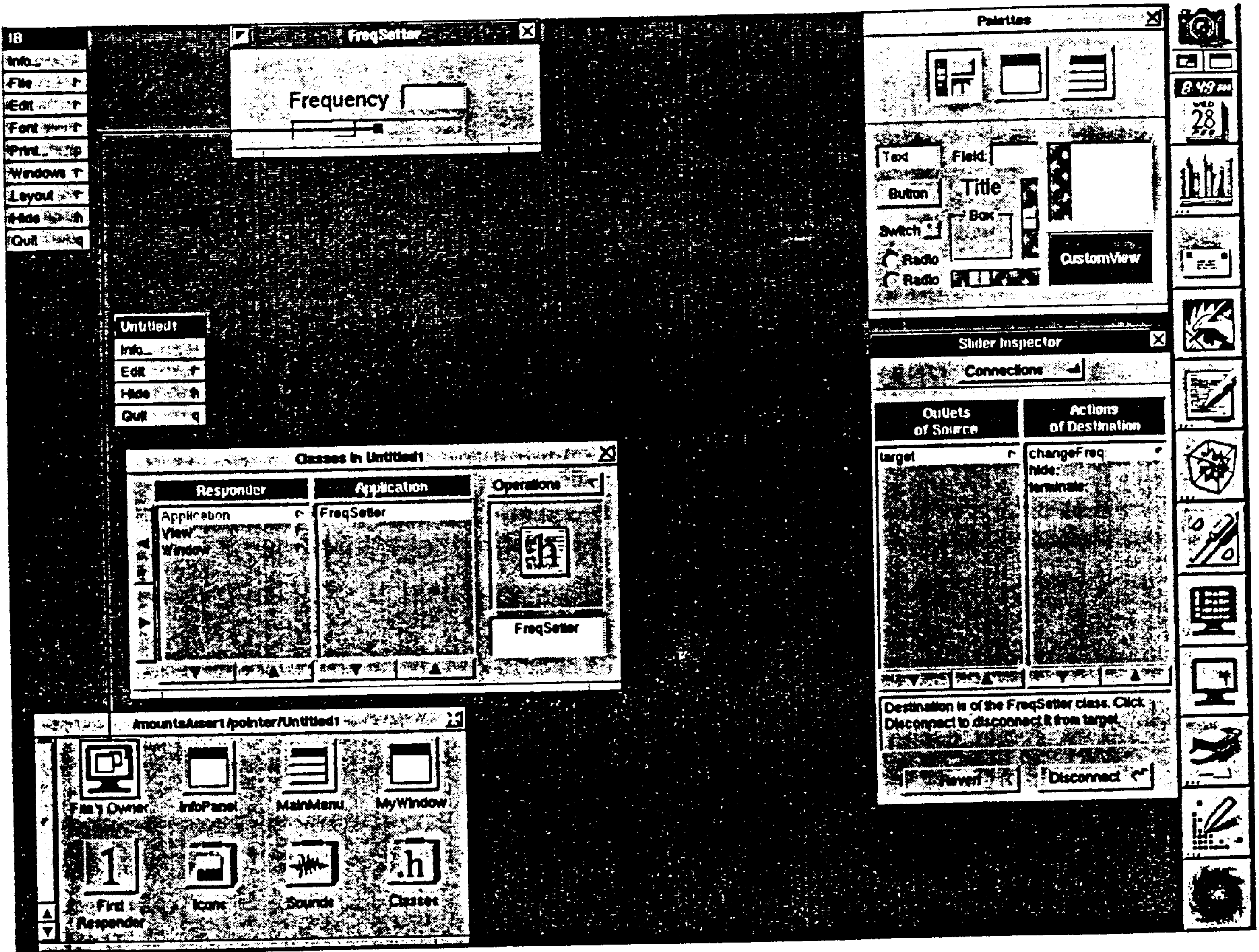


Figure 20
IB Example: Connecting Targets and Actions

```
/* File: freqSetter.h
Sample implementation of a a slider and text field which can both
be used to set the same value, and both will display its current
value.
This header file declares the objects and methods in freqSetter.
Except for the declaration of the int variable freq,and our
comments, this file is exactly as generated by Interface Builder.
*/
#import <appkit/Application.h>
/* This import file (like an include file) allows the use of
Application Kit Objects like sliders.
*/

@interface FreqSetter:Application
{
    id  freqSlider;
    id  freqViewer;
    int freq;/*this line added by hand*/
}

- setFreqSlider:anObject;
- setFreqViewer:anObject;
- changeFreq:sender;

@end
```

Figure 21
Objective-C Header File For IB Example

```
/*  File: FreqSetter.m
Sample implementation of a a slider and text field which can both
be used to set the same value, and both will display its current
value.
This is the main file which implements the methods declared
above.  It is as created by Interface Builder, except for the
implementation of the changeFreq method and the comments.
*/
#import "FreqSetter.h"
/* Header file as above.*/

@implementation FreqSetter
/*The following two methods associate outlets created graphically
in Interface Builder (IB) with variables in the Objective-C code.
```

```
Methods like these are created automatically by IB for every
outlet.
*/
- setFreqViewer:anObject
{
    freqViewer = anObject;
    return self;
}

- setFreqSlider:anObject
{
    freqSlider = anObject;
    return self;
}

/* For both of the above objects their target was set in IB to be
an instance of FreqSetter and the action its changeFreq: method.
Now whenever either the slider or the textView is changed, it
will trigger the changeFreq: method to be executed.
*/

- changeFreq:sender
{
    freq = [sender intValue];
/*textViews and sliders both understand the message intValue, so
whichever one sent the message will respond to the message with
the integer value it currently holds.  We set the Objective-C
program's internal variable freq to that value.
*/
    [freqViewer setIntValue:freq];
    [freqSlider setIntValue:freq];
/* Update both the slider and the textView to reflect the current
value.  One of them is already current, since it sent the
message, so another implementation would be to check which one
sent the message and only update the other one.
*/
    return self;
}

@end
```

Figure 22
Objective-C Main File For IB Example

Filling in the Objective-C code as in Figures 21-22 and compiling the application will result in an application which allows the frequency value to be set either with the slider or by typing into the textField; both user-interface objects will reflect the current value of the variable. Naturally, a application would need to do something with this value to be useful. Demonstrations such as the Digitizer (Figure 11 above) get values from the user in this manner, then generate graphics in Mathematica using routines like Figure 16 above. Applications using sound get input from the user in this manner, then use the Sound and Music Kits to apply the user's choices to the audio demonstration.

The Sound Kit. Our sound examples using the Sound Kit are a combination of pure object-oriented programming and machine-dependent bit-manipulation. The Sound Kit classes provide methods which normally keep the sound-using programmer shielded from the implementation of sound storage and manipulation. However, the methods provided are geared toward simple recording and playing of sounds; the manipulations we wanted to do with the sound data required getting at the data and modifying it directly. For example (Figure 23), to simulate the effect of quantizing a sound to different numbers of bits, we went into the representation of the sound and truncated the bytes used to store the sound. Our method is thus tied to the particular storage implementation NeXT is using, although we support other sound formats by converting them to the one we use. Alternatively, we could have defined a subclass of Sound and hidden these details inside the quantize method, but then this demonstration could not use a Sound object created elsewhere.

```
- quantize:sender
{
    if ( ( [ self isQuantizable /*a custom function*/]  == NO)
&& ( [self makeQuantizable /*a custom function*/] == NO) ) {
        NXRunAlertPanel("Quantize error", "Sound not
quantizable","OK", NULL, NULL);
    }
    else {
        /*if here, then sound is quantizable:*/
```

```
        /* OK to proceed with poking at the sound*/
        id theSound = [view sound];
        SNDSoundStruct *soundStruct = [theSound soundStruct];
        unsigned char *soundData  = [theSound data] ;
        int dataSize = [theSound dataSize];
        int i;

        for ( i = 0; i < dataSize; i += 2) {
        /*leave the object-oriented paradigm and exploit the
        implementation of the Sound object*/
             *soundData++ = (*soundData & upper_8_bit_mask);
             *soundData++ = (*soundData & lower_8_bit_mask);
        }
    }
    return self;
}

- setMask: (int)numBits
{
/*leave the object-oriented paradigm and exploit the
    implementation of the Sound object*/
/*bits is declared an int in the header file*/
bits = numBits;
/* ~(~0 << n) creates a mask with 1's in the rightmost n bits,
0's elsewhere*/
upper_8_bit_mask = (~(~0 << 16)) << (bits > 8? 0 : (8 - bits));
lower_8_bit_mask = (~(~0 << 16)) << (bits < 8? 16 : (16 - bits));
}
```

Figure 23
Objective-C Code Fragment:
Poking Inside the Sound Kit to Simulate Quantization

The Music Kit. We used the Music Kit for only one demonstration, an extension of the PlayNote application in NextDeveloper/Examples/MusicKit/PlayNote. The sound portion of this demonstration plays a sinusoid of varying frequency and amplitude. Its implementation is similar to that of the PlayNote application, so we will not discuss it here. In contrast to the Sound Kit example described above, our sine wave generator did not require leaving the object-oriented paradigm. We had planned to import recorded files from the Sound Kit and manipulate them in the Music Kit, but this turned out not to be possible in the current release of these kits.

Evaluation of DISIPLE System

In this section we evaluate the system against its design goals.

A first comment to make is that the system is incomplete. Only parts of the syllabus are implemented, both in terms of text and demonstrations. This means that the system may not yet be useful as a self-contained tutor, but could be used in conjunction with another text or to demonstrate specific topics. Also, because the system is incomplete there has been little feedback from potential users. The comments herein are our own.

The user-interface of the system is generally a success. A novice user interacts with a Mathematica notebook in much the same way as with a book, so that there is some familiarity with the format of the system. The user can launch some relatively sophisticated graphics and sound demonstrations with no typing at all, and these demonstrations allow the user to vary parameters at will and to return to any demonstration at any time. A user who could not or would not do the necessary programming can still experience some real-time audio and video demonstrations tailored to his/her liking.

Probably the biggest problem with the user-interface of the system is the number of instructions we must give. Even a computer-literate user must adjust to the NeXT machine somewhat, for although its interface is internally consistent, it takes some learning (for example, using menus, using a mouse, and closing windows). The DISIPLE user must adjust not only to the NeXT machine, but to some aspects of Mathematica and the particular demonstrations we offer. The implementers of Mathematica themselves suggest working with a simpler program on the NeXT (such as a word processor) before trying to use Mathematica at all [Wolfram Research 89]. The DISIPLE interfaces are straightforward, but many instructions still must be explicitly given in the text to a novice user.

One shortcoming of DISIPLE has to do with its textual presentation of mathematical ideas. Mathematical conventions such as summations were developed to represent large quantities of information in a concise symbolic form, and it is difficult to teach mathematical topics without them (as, ironically, one must do in Mathematica).

The system's animations are not very effective, for two reasons. The first is that the user must manually set various parameters of an animation each time it is invoked. This limits the effectiveness of animations greatly, particularly the fact that one can't specify in advance whether the animation should cycle through the frames or double back at each end. An unfair burden is placed on the user, who must try to adjust the animation to demonstrate a topic s/he doesn't yet understand. The second problem with the animations is that even when properly adjusted, they are not always effective at illustrating the point they are intended to illustrate. The ability to animate graphics doesn't guarantee that animating one's graphics will produce a result worth having, any more than the ability to compile C code guarantees that one will write a useful program. The animations that work best are either very straightforward (for example, the simulation of a string vibrating sinusoidally) or were produced by trial and error, varying the input graphics until the animation produced a result we wanted.

The sound and graphics demonstrations are effective as far as they go, but are limited. For example, most of the demonstrations deal with only a single sine wave, manipulated in various ways. It is a good introduction to topics like aliasing and the discrete Fourier transform for a user to use sine waves as test cases, but it would be more effective eventually to use real sampled data, or at least more complicated input, as test cases for these topics. One of the most effective demonstrations is the quantization demonstration where a user can hear and see the effects of quantizing his/her voice to an arbitrary number of bits, and its strength stems largely from its use of real data as input.

The ability to vary parameters of a demonstration and perceive the result in both sound and graphics is effective, and should be retained in future systems. For example, the demonstration allowing the user to change the frequency and amplitude of a sinusoid and both hear and see the difference helps the user quickly get an intuitive feel for sinusoids. However, a problem with all of the interactive graphics is the amount of time it takes Mathematica to compute and image graphics. It is unlikely that a user would spend enough time with an interactive demo, varying parameters and observing the results, to get appreciably more value than from a static demo.

DISIPLE has not yet achieved the desired status of a complete DSP demonstration environment. One reason has simply been that its implementation has not progressed that far. A more worrisome reason is that it has proved hard to expand the range of choices available to the user without also rendering the interface too complicated to meet our design objectives. Even now, the user-interface for the sampling and quantization demo shown above (Figure 11) borders on overwhelming. It would be possible to hide various choices in pop-up windows, but hiding choices implies a longer learning curve for demos which are intended to be used only for a short time. In the demonstration environment we had envisioned, where the entire range of DSP topics we had covered would be available at once, we would not be able to present the choices among topics in a convenient way.

Another shortcoming of DISIPLE compared to its design goals is that its machine-dependent features are only partially encapsulated. All of the Mathematica features are portable, but the graphic interfaces and the sound demonstrations would have to be reimplemented to port DISIPLE to another machine. Since these comprise a large portion of the system, porting DISIPLE to another system such as the Macintosh would not be trivial.

This concludes our discussion of the DISIPLE system. In the next section we will evaluate the tools used for this project with regard to their potential for interactive multimedia systems to teach audio technology topics.

Evaluation of Tools and Capabilities

In this section we evaluate the tools we used for this project, with regard to their potential for interactive multimedia systems to teach topics in audio technology.

Mathematica is a versatile system, and is a platform we would recommend for future systems if its speed improves with new releases of software or hardware. It makes an excellent document processor for technical documents because of the notebook feature, its generation of graphics and its ability to include "live" mathematical functions. However, there are some missing features for document processing. For example, a cell may not combine two fonts in one sentence, which precludes italicizing keywords. Worse, it is not possible to print many mathematical formulae in standard form (Greek letters are not even supported); although Mathematica can give output in form suitable as TeX input, it cannot print TeX-style formulae in its own notebooks. This is troubling in a system which bills itself as a platform for teaching mathematics. Overall, Mathematica is no replacement for TeX in preparing technical books, but offers some unique advantages as a technical document processor for preparing interactive multimedia systems.

As a programming language, Mathematica suffers somewhat from its attempts to include everything for everyone. First, it is slow. Second, the language is so large that it would take a long time to be familiar with all the keywords and built-in constructs. Third, it allows so many programming paradigms that it is often unclear which would be the most efficient way to implement a custom function. In the future, we would do as much computation as possible outside of Mathematica (say in C) and save Mathematica for plotting graphics and organizing the document. One problem with doing this in digital audio topics, though, is that using Mathematica to plot the results of external computations on collections of samples would require reading lists of data into Mathematica.

Mathematica's animation feature holds promise, but currently it is not a tool for production work. The reason is that the programmer can not define the speed or direction of the animation ahead of time, but must rely on the user to adjust them for the right effect. If the purpose of an

animation is to demonstrate an unfamiliar concept to a student, one cannot rely on the student to adjust the animation until it demonstrates the concept as planned! It is possible to animate PostScript graphics on the NeXT machine outside of Mathematica, but we did not investigate this for DISIPLE. Because teaching systems are likely to require animations of technical graphics, it would probably be necessary to generate the graphics in Mathematica, then cut and paste them into another window for animation. A further point on animation is that the time it takes to generate the still-frame graphics precludes allowing the user to make choices about what to show in the animation. A system where the user chooses what s/he would like to see in an animation, then waits five minutes or more for Mathematica to generate all the graphics, is not the kind of "interactive" system anyone would use.

Another problem we had with Mathematica was in communicating between it and external programs using the utilities supplied with the system. It sometimes would take several seconds more to execute a Mathematica command from without than from within. The MathTalk protocol would be worth investigating for future systems. Also, the planned implementation of Mathematica as an Application Kit object would facilitate communication with Mathematica.

Although Mathematica makes some aspects of plotting easy, it makes others difficult. Sometimes a desired effect is impossible even with manual settings of multiple options. For example, there is a graphical convention in DSP books that, when graphing the phase of a discrete Fourier transform, a small "x" is printed on the horizontal axis to represent an undefined phase. However, none of Mathematica's graphic options allow the size of an object created from graphics primitives (like an "x") to be a constant relative to the size of the graph, so we were not able to implement this convention in our plotting function. If there were a good supported drawing program on the NeXT, it would be useful at times to create sample graphics there and import them.

Turning now away from Mathematica, we found that Interface Builder and the object-oriented programming paradigm made it relatively easy to create working graphic user interfaces. The application development environment of NextStep is a major advantage of working on the NeXT machine, because the programmer is freed from much of the tedious work of application

development. Instead, s/he concentrates on the features of the application at hand.

The many (though small) sample applications provided as part of the NextDeveloper package were also useful. They demonstrated the use of certain classes and application development tools better than the NeXT manuals, and they also provided working applications the programmer can extend. Many of our demonstrations were extensions of Next-supplied sample applications, and this made their development easier than coding them from scratch.

The Sound Kit and Music Kit provide some interesting capabilities, but are incomplete. We were disappointed that the process for importing a Sound object into the Music Kit doesn't work, because we had hoped to filter the user's recorded voice (a Sound) with the DSP (via the Music Kit). (Three bugs in the Sound and Music Kits that impeded our progress are being corrected by NeXT [Jaffe 90].) Other functions simply aren't implemented, for example some of the sound format conversions that we had wanted to use. Notwithstanding these shortcomings, the Sound and Music Kits still make it relatively easy to incorporate sound into applications. Especially if future releases fix some of their flaws, an application developer could use them and occasional DSP code to incorporate some very sophisticated audio demonstrations into a NeXT-based learning environment.

The NeXT machine on the whole is a double-edged sword. For the developer, the fact that it offers so many tools in different media is a great help, but the fact that they are at times poorly documented or buggy is a source of great frustration. For the user, the current speed of the machine is a real inconvenience. Nearly everything takes an appreciable amount of time to accomplish (switching between running applications, loading files, imaging graphics), and at this point we don't expect that a user would interact with the multimedia demos for long enough to benefit from them because they are so slow.

Although the NeXT may make it easier to create interactive multimedia than most computers, there is still no integrated environment for doing so. We spent several months learning the above tools before we were able to implement a single demonstration using sound, graphics, and a graphic interface, and found that many of our anticipated demonstrations were too complicated.

Certainly the learning curve would have been shorter in an integrated environment which offers sound and graphics. We don't know of such an environment on any machine, but its absence makes the development of interactive multimedia textbooks on the NeXT a time-consuming task. Its absence also required some redundant work, for instance implementing quantization separately for Mathematica graphics and for sound (Figures 12 and 23 above). Having to use several different systems also means a steeper learning curve for a user.

Finally, it should come as no surprise that applications which make full use of the multimedia capabilities of the NeXT are not portable to other machines. For systems built as Mathematica notebooks, like DISIPLE, the notebook and Mathematica packages are portable to another machine but not the implementation of the demonstrations. Interface objects created with interface Builder, the Sound and Music Kits, and (in practice, since C++ has become the *de facto* standard for object-oriented C-programming) Objective-C itself, are all specific to the NeXT machine.

Conclusions and Future Directions

We believe that the DISIPLE system illustrates some possibilities for developing interactive multimedia systems on the NeXT machine, and for teaching topics in audio technology using interactive multimedia. The experience gained in its development will be useful in developing related systems.

Although we have concentrated on the domain of digital signal processing, these techniques can be used in other domains in music and music technology. We envision interactive multimedia systems to teach music theory, psychoacoustics, arranging, and performance, to name just a few. There are many exciting possibilities for the use of computer technology to expand our musical horizons, and we hope that this is one of many happy marriages to come between music and technology.

Appendix A

Syllabus And Notes For DSP Course

- Introduction
    - Purpose of course
        - —audience assumed interested in music or audio
        - —will use that sort of example
    - Motivation
    - Format of course
        - —will have stored sound examples (music, white noise)
        - —also a microphone to record user's voice
        - —will do everything technical in the digital world, occasionally using analog for motivation or insight
    - How to Use the System and Mathematica
- Review of Math
    - Binary numbers
    - Exponentials
    - Trigonometry & sinusoids
        - —show sinusoidal wave as well as trigonometric version of sine and cosine
        - —hear sinusoid- will be used in many illustrations
    - Complex exponentials
        - —Euler's formula
        - —cos and sin in terms of exp(jw)
        - —show pictures, movie
    - Fourier decomposition
        - —not too detailed, just the basic idea of time<->frequency
        - —motivate with music example("treble/bass")
        - —show the time representation of a signal (maybe a second of a musical tone) and then the Fourier transform (don't talk about the technique yet) to demonstrate what each representation has and lacks
- Review of the Computer World
    - Data stored in finite-length words.
    - Intuition of underflow and overflow.
    - Resources
        - —processing time
        - —storage
        - —cost can be $, or can keep you from doing things in real-time
        - —often have tradeoffs between these, or between resources consumed and functionality; many "breakthroughs" in computers are figuring out how to do things you want within the resources you have
- Getting Into the Digital Domain
    - Introduction
    - Representation of a discrete-time signal
    - Sampling
        - —show example analog->discrete signal

—allow user to choose sampling rate and listen to example
—suggest they go to very low sampling rate and listen- will motivate the aliasing discussion

Aliasing
- —wagon wheel phenomenon- show movie and allow user to choose movie's sampling rate
- —simple signal example- show & hear that sinusoids at frequencies f and 3f are indistinguishable when sampled at 2f. Let them play.
- —Nyquist Thm. Show formula for foldover frequency and hear it with simple example.
- —use complicated example (real music). show spectrum of original and aliased.

Quantization
- —pointer to above treatment of how numbers are stored in computers
- —show picture of digital signal getting quantized
- —use example and let them choosenumber of bits for quantization, then listen
- —show the noise signal and its spectrum, vs. spectrum of original

Summary of getting into the digital domain
- —must choose sampling rate carefully, or get unpleasant aliasing
- —quantization noise is more tolerable in many cases.
- —from now on, will work completely in the digital domain (occasionally using analog world for insight only).

In the Digital Domain

Introduction
- —soothe user that many people have to be exposed to all of it before any of it makes sense (i.e. z-transform, convolution, etc.); if some of it seems unclear, may be best to keep going then do the whole thing over again.

DFT
- —like the Fourier analysis in analog time, can do Fourier transform in digital domain.
- —formula.
- —simple example (just a few data points) in great detail.
- —allow user see DFT computed for one of our sound examples. (And differentiate into hearing each frequency component separately?)
- —implied periodicity.
- —mention FFT but not in detail- it's fast & needs powers of 2; used extensively.

Inverse DFT
- —same example as above - note periodicity.
- —allow them to input a sequence of components and take inverse DFT.

Z-transform
- —just justify as a useful mathematical technique; don't try to make it make sense.
- —usually on paper for insight, not for computation.
- —formula
- —convergence
- —examples
- —z-plane
- —DFT as special case (see unit circle)

Inverse z-transform
- —hard; only show the partial fraction expansion technique.

Digital filters

introduction
—what is a filter?
—linear, time-invariance condition
introduce simple example to refer to- Smith's example of simplest LPF? [Smith 85]
—$\{y(n) = x(n) + x(n-1)\}$
—intuitively will pass DC, stop 1/2 sampling freq.
time-domain :
—difference equation representation
—impulse response
—show impulse response of our example
—causality criterion
—convolution
—filter output is input convolved with impulse response
—show simple example in excruciating detail
—implement a "DetailedConvolve" function which, given
user input, shows multiple delayed graphs before
adding to arrive at answer
—note that any two signals can be convolved-
need not think of one as impulse response
—note that convolution is time-consuming and not often used in practice
frequency-domain:
—transfer function
—H(z) is z-transform of impulse response
—$x*y=z \Leftrightarrow XY = Z$
—H(z) of our example
—poles and zeroes (show 3D and 2D representations)
—picking off pole and zero locations from H(z) equation
—frequency response
—specialize transfer function to unit circle
—how to intuit the magnitude from pole/zero locations
—magnitude and phase
—stability iff poles inside unit circle
—plot of magnitude and phase for our example
—run a sound bite through our example and show DFT
before and after and frequency spectrum.
—getting frequency response graphically
using vectors from poles and zeroes
show more examples in all four representations.
show some common transform pairs (delta to all-pass,etc;
note that they go both ways).
Implementing digital filters
—direct form
—cascade
—parallel
Designing digital filters
Bringing it all together
—create an environment where user can play with all these
(mostly just reminding of what functions are available,
also suggesting some ideas of mini-projects to take through several steps)
Conclusion and References

Appendix C

Distribution And Execution Instructions

For The Disiple System

For information regarding licensing of the DISIPLE software and obtaining a copy, contzact Leslie Delehanty (administrator) at the Center For New Music and Audio Technologies through any of the following channels:

Mail CNMAT

Department of Music

University of California, Berkeley

1750 Arch St.

Berkeley, CA 94709

Electronic mail cnmat@cnmat.berkeley.edu

Phone (415) 643-9990

Fax (415) 642-7918

To install the DISIPLE system from an optical disk onto another NeXT computer, copy the entire DISIPLE.app directory into the /LocalApps directory of the new host. The DISIPLE.app directory is currently (March 1990) found in /LocalApps of the host NeXT machine (named "cnmat") at CNMAT. The necessary files include the Mathematica notebook (DISIPLE.ma and DISIPLE.mb), various Mathematica packages (Plots.m, DFT.m, etc), and various demonstration applications (SineWaver, Digitizer, etc.).

To execute the DISIPLE system, simply double click on the icon of the Mathematica notebook, DISIPLE.ma.

References

Ambron, Suanne, and K. Hooper, eds., **Interactive Multimedia**, Microsoft Press, Redmond, Washington, 1988.

Cox, Brad, **Object-Oriented Programming: An Evolutionary Approach**, Addison-Wesley, Reading, Massachusetts, 1987.

Freed, Adrian, "New media for musicological research and education - The country blues in hypermedia", *Proceedings of International Computer Music Conference 1989*, Columbus, Ohio, 1989.

Jackson, Leland B., **Digital Filters and Signal Processing**, Kluwer Academic Publishers, Hinsham, Massachusetts, 1986.

Jaffe, David, personal communication, Jaunary 30, 1990.

Moore, F. Richard, "An Introduction to the Mathematics of Digital Signal Processing", pp. 1-67 in Strawn, John, ed., **Digital Audio Signal Processing: An Anthology**, William Kaufmann, Inc., Los Altos, California, 1985.

NeXT, Inc., **The NeXT System Reference Manual**, Redwood City, California, 1989.

Oppenheim, Alan V., and R. W. Schafer, **Discrete-Time Signal Processing**, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1989.

Smith, Julius O., "An Introduction to Digital Filter Theory", pp. 69-135 in Strawn, John, ed., **Digital Audio Signal Processing: An Anthology**, William Kaufmann, Inc., Los Altos, California, 1985.

Strawn, John, ed., **Digital Audio Signal Processing: An Anthology**, William Kaufmann, Inc., Los Altos, California, 1985.

Wessel, David, R. Felciano, A. Freed, and J. Wawrzynek, "The Center For New Music and Audio Technologies", *Proceedings of International Computer Music Conference 1989*, Columbus, Ohio, 1989, pg. 1.

Wolfram, Stephen, **Mathematica: A System For Doing Mathematics By Computer**, Addison-Wesley, Redwood City, California, 1988, pg. vii.

Wolfram Research, on-line help for Mathematica on the NeXT machine, Redwood City, California, 1989.