THE NYQUIST COMPOSITION ENVIRONMENT: SUPPORTING TEXTUAL PROGRAMMING WITH A TASK-ORIENTED USER INTERFACE[*]

Roger B. Dannenberg
Carnegie Mellon University
School of Computer Science
Pittsburgh, PA, USA

ABSTRACT

Nyquist is a programming language for sound synthesis and music composition. Nyquist has evolved from a text-only programming language to include an integrated development environment (IDE) that adds graphical support for many tasks. Nyquist is also hosted by Audacity, a widely used audio editor that can invoke Nyquist functions written in the form of scripted plug-ins. This article shows by example how task-oriented interface design can augment a text-based language.

1. INTRODUCTION

Nyquist is a programming language for sound synthesis and music composition. It has evolved continuously since its first version in 1989 [2]. Although the basic Nyquist engine has remained the same since 1993 [3], the program as it appears to users has undergone quite a few changes. In particular, the text-based language is now supported by an extensive integrated development environment (IDE). The goal of this paper is to provide an update on these developments and to illustrate how graphical interfaces can work to support a text-based computer music system.

One might view these developments in the context of work by Eaglestone, et al. [6] on cognitive styles and the design of electroacoustic music software. This work suggests that some composers will feel more comfortable working with, for example, text-based systems, and others will be attracted to direct manipulation graphical interfaces. An integrated system that supports different cognitive styles might be easier to use and support a wider range of users.

I will begin with an overview of the basic concepts of Nyquist for readers who are not familiar with Nyquist or early articles describing it. Sections 3 through 5 describe new features of Nyquist and the Nyquist IDE. Subsequent sections describe implementation choices, the Audacity environment, and future work. A concluding section summarizes what we have achieved and learned.

2. NYQUIST BASICS

Nyquist is fairly unique among sound synthesis languages in that it adopts a strongly functional style of programming (but see also the Faust language [11]). In particular, synthesis in Nyquist is performed by functions that operate on virtual streams of audio samples. Other synthesis systems do create a similar illusion; for example, unit generators in csound [16] appear at first glance to be functions operating on streams. However, closer examination reveals that csound "streams" are really blocks of samples, and unit generators are not functions but procedures that read from sample blocks and store results to sample blocks.

In Nyquist one can write in a functional style:

```
osc(pitch) * env(0.1, 0.2, 0.3, 1, 0.5, 0.2)
```

In this case, it is fairly easy to show that this can be compiled to a more procedural form, where `osc`, `env`, and `*` are unit generators and `b1`, `b2`, `b3` are sample blocks:

```
b1 = osc(pitch)
b2 = env(0.1, 0.2, 0.3, 1, 0.5, 0.2)
b3 = b1 * b2
```

This, in a nutshell, is the difference in terms of language design and semantics between Nyquist and csound (or your favorite Music N language).

The functional form of Nyquist sound synthesis expressions has some very interesting features. Sounds in Nyquist are values that can be passed as parameters, stored in variables, and returned from functions. While Music N sounds only exist as a small buffer of samples representing the current instant of time, Nyquist sounds are accessible at any time and fully reusable.

The functional style encourages modular programs and allows users to extend the system with their own set of personal "unit generators" or synthesis instruments. It is true that csound (and Music N) programs allow users to define instruments using an "orchestra" language, but instruments can only be invoked from a score, whereas

*Originally published as: Roger B. Dannenberg, "The Nyquist Composition Environment: Supporting Textual Programming with a Task-Oriented User Interface," in *Proceedings of the 2008 International Computer Music Conference,* San Francisco, CA: The International Computer Music Association, August 2008. © 2008, Roger B. Dannenberg.

Nyquist makes no distinction between scores and orchestras. Thus, instruments can be defined in terms of other instruments, and scores can be nested hierarchically. This generality leads to great flexibility for composers. For example, a control envelope can extend over the duration of a score such that timbres of sound objects in the score evolve according to the overarching control envelope.

To accomplish all this, Nyquist uses a sophisticated lazy evaluation scheme: sounds are only computed on demand. Sounds are represented as a linked list of sample blocks to avoid the need to keep entire sounds in memory. Normally, sounds are not computed until a play command is executed, and sounds are computed incrementally, with garbage collection recycling samples that are no longer needed in memory. The result is that very little memory is required even when long sounds are computed. Although the internal mechanisms are quite complex, the user's view is clean and elegant, and the execution speed is considerably faster than existing Music N implementations [5].

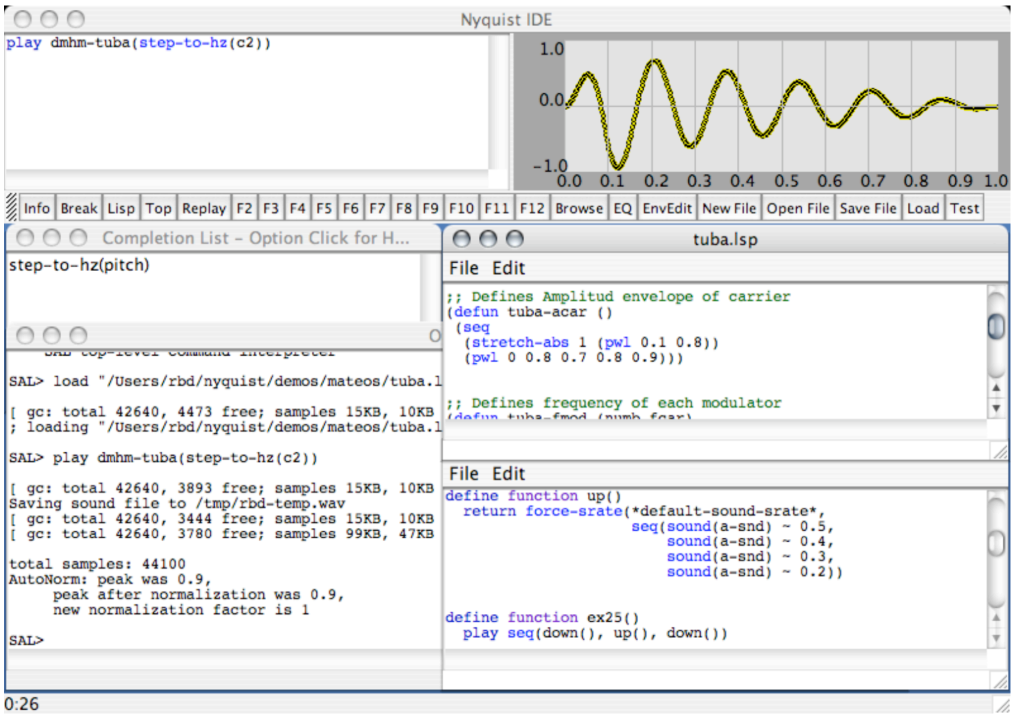
3. THE EVOLUTION OF NYQUIST

Originally, Nyquist was a command-line-only program that extended a small Lisp interpreter, XLISP. Users typed in expressions or loaded files, and results were written to sound files or printed on the console. I started teaching classes with Nyquist in 2002, and immediately, student feedback showed a desire to make Lisp programming easier. Ning Hu implemented an integrated development environment (IDE) for Nyquist on Microsoft Windows including syntax-directed editing and the ability to plot signals. This served as a model for a cross-platform Java-based IDE originally written by students Chris Yealy and Derek D'Souza.

The presence of a Java-based IDE has led to many new developments. Increasingly, Nyquist is serving as a "rendering engine" controlled at a higher level using graphical design tools in the IDE. In parallel, Nyquist has been extended with new synthesis capabilities including ports of STK [1] instruments, linear prediction, piano synthesis, digital audio effects, a Minimoog emulator, and new libraries for algorithmic composition.

The most recent development is an implementation of SAL, a language based on standard functional notation, infix operators, and familiar-looking control constructs such as loops and if-then-else. Now, users who are put off by Lisp syntax may find Nyquist more accessible and easier to learn.

The thesis of this paper is that a task-oriented working environment can improve the usefulness of a language (like Nyquist) for sound synthesis and music composition. In other words, there can be synergy between a language and the environment in which it is used. This should be no great surprise, as other systems such as Max MSP [18] and SuperCollider [9] certainly benefit from their development environments. In addition, several non-real-time synthesis and composition systems incorporate graphical controls and interfaces to complement text-based programming. Examples include Cecilia [12] and Siren [13]. Still other languages and systems have focused on graphical rendering of compositions and musical data, including SEE [7], Audicle [17], and FOMUS [14]. This paper will concentrate on describing new features of the Nyquist development environment. This work suggests that we think about how visual interfaces, task-oriented support tools, and language


Nyquist IDE
1.0
0.0
-1.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
Info Break Lisp Top Replay F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F12 Browse EQ EnvEdit New File Open File Save File Load Test
Completion List – Option Click for H...
tuba.lsp
File Edit
File Edit
0:26

```
play dmhm-tuba(step-to-hz(c2))
```

```
step-to-hz(pitch)
```

```
SAL> load "/Users/rbd/nyquist/demos/mateos/tuba.l

[ gc: total 42640, 4473 free; samples 15KB, 10KB
; loading "/Users/rbd/nyquist/demos/mateos/tuba.l

SAL> play dmhm-tuba(step-to-hz(c2))

[ gc: total 42640, 3893 free; samples 15KB, 10KB
Saving sound file to /tmp/rbd-temp.wav
[ gc: total 42640, 3444 free; samples 15KB, 10KB
[ gc: total 42640, 3780 free; samples 99KB, 47KB

total samples: 44100
AutoNorm: peak was 0.9,
     peak after normalization was 0.9,
     new normalization factor is 1

SAL>
```

```
;; Defines Amplitud envelope of carrier
(defun tuba-acar ()
 (seq
  (stretch-abs 1 (pwl 0.1 0.8))
  (pwl 0 0.8 0.7 0.8 0.9)))


;; Defines frequency of each modulator
```

```
define function up()
  return force-srate(*default-sound-srate*,
                     seq(sound(a-snd) ~ 0.5,
                         sound(a-snd) ~ 0.4,
                         sound(a-snd) ~ 0.3,
                         sound(a-snd) ~ 0.2))


define function ex25()
  play seq(down(), up(), down())
```

Figure 1. The Nyquist IDE.

design can be further coordinated and extended to provide better computer support for creative activity.

4. THE NYQUIST DEVELOPMENT ENVIRONMENT

The main impetus behind Nyquist's new trajectory has been the IDE (Integrated Development Environment), which adds an interface layer between the user and Nyquist. In the beginning, the IDE provided only an integrated text editor, graphical display of waveform and control functions, and some menus and buttons for common commands. Figure 1 illustrates these features of the IDE. Commands are typed in the upper left text-entry box, with "hints" displayed just below, and a typescript of Nyquist output is displayed at the lower left. Additional text windows are for editing files. Buttons perform commands such as "save the current file and load it into Nyquist," and "break from the interactive debugger and return to command entry."

The IDE introduced the idea that communication with Nyquist could be mediated through a graphical user interface. This in turn has inspired many new features. Most of the IDE features are oriented around specific tasks that users have felt could be supported better through graphical interfaces than by text-based data and program entry.

4.1. Command Completion and Hinting

One interesting feature is command completion or "hinting" that examines text as the user types it. A list of Nyquist and XLISP functions are searched, and the full names and parameter names or matching functions are displayed as suggestions to the user. For example, in Figure 1, when the user types `step`, the Completion List window just below displays:

```
hz-to-step(freq)
step-to-hz(pitch)
```

The user can then click on a line to enter the expression where the user is typing (expanding `step` to a complete function call). If the user right-clicks on the hint, the reference manual entry for the function is displayed in a browser. Since the IDE includes an editor for SAL and LISP files, the command completion operates during file editing as well as for command entry. The function lists are extracted automatically from the documentation, and translated automatically into either SAL or LISP syntax based on what the user is typing. This facility, originally implemented by my student, Austin Sung, has been a great help to novices and experienced users alike.

4.2. Envelope Editor

An example that provides graphical editing to augment text-based programming is the Envelope Editor. Nyquist provides a number of envelope functions for piece-wise linear, piece-wise exponential, and a few other forms of general shapes described by parameters. As might be expected, many users find it clumsy to edit numbers rather than editing a graphical representation. The Envelope Editor, a built-in component of the IDE, is a simple but flexible tool for this task. Envelopes created here are automatically converted to/from text representations, and can be edited in either form. Then, they can simply be referenced by name whenever the envelope is needed in a Nyquist program. Figure 2 shows the Envelope Editor, which opens as a sub-window within the IDE.

An important part of the design is the interface between graphical object editors and text programming. In the Nyquist IDE, editable objects are generally named by the user, so in Figure 2 you see a box (upper left) to name or rename an envelope, and a drop-down menu (upper right) to select by name and edit a different envelope. After clicking the "Save" button, the user can access the edited envelope as a function, e.g. by calling `env-1()`.

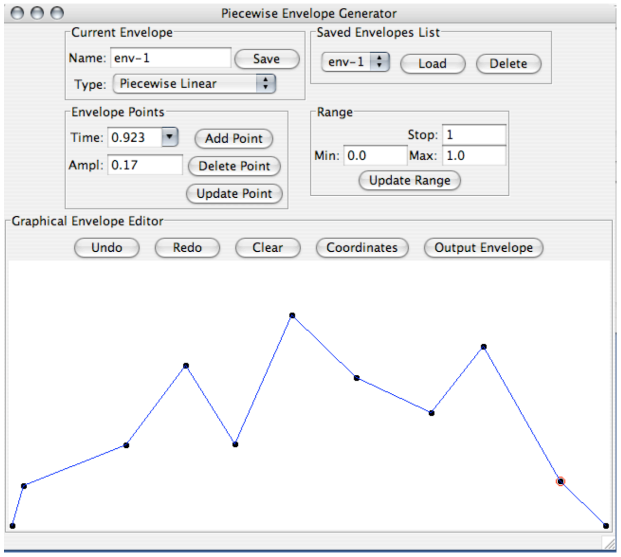

Piecewise Envelope Generator
Current Envelope
Name: env-1
Type: Piecewise Linear
Saved Envelopes List
env-1
Envelope Points
Time: 0.923
Ampl: 0.17
Range
Stop: 1
Min: 0.0
Max: 1.0

Figure 2. The Nyquist IDE Envelope Editor.

Envelopes have a double life: they are callable as functions but editable as data, both graphically and as text. This raises several questions: Where are envelopes saved when Nyquist exits? How is data accessed by the IDE, by a text editor, and by Nyquist programs? The data (mainly breakpoint coordinates) for all envelopes are stored in the Nyquist process as a list. Each envelope is also defined in LISP as a function.

When the Envelope Editor is opened, it tells the Nyquist process to print the data, and it "spies" on Nyquist's output to capture the envelope data. Each time the user clicks the editor's "Save" button, the modified envelope data is sent back to Nyquist. Within the Nyquist process, envelope updates are stored as data on a list and also redefine the corresponding function, e.g. `env-1()`.

Finally, when Nyquist exits, the envelope data is preserved (as textual data) in a file called the *workspace*. This is a general mechanism for saving any data, including scores, from one run of Nyquist to the next. User desiring access to the text can easily save the workspace, edit envelopes as text, and reload the workspace to update the envelope function definitions. The IDE monitors workspace activity and tries to help the user avoid mistakes. For example, if the workspace is not loaded before opening the envelope editor (which might prevent a synchronized view of data in the LISP and IDE worlds), a warning is issued, and the IDE offers to load the workspace.

4.3. EQ Editor

Equalization and filter design is another task that can be supported by graphical tools. While "old school" users may be comfortable typing and fine-tuning equalizer gains as text, Nyquist users can now call up an editing window to specify multi-band equalization. Figure 3 shows the EQ editor.

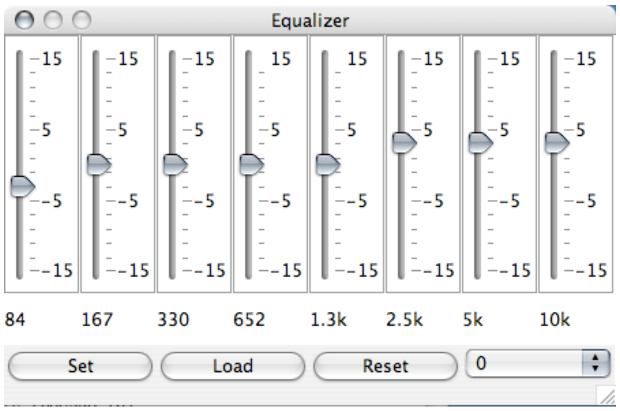


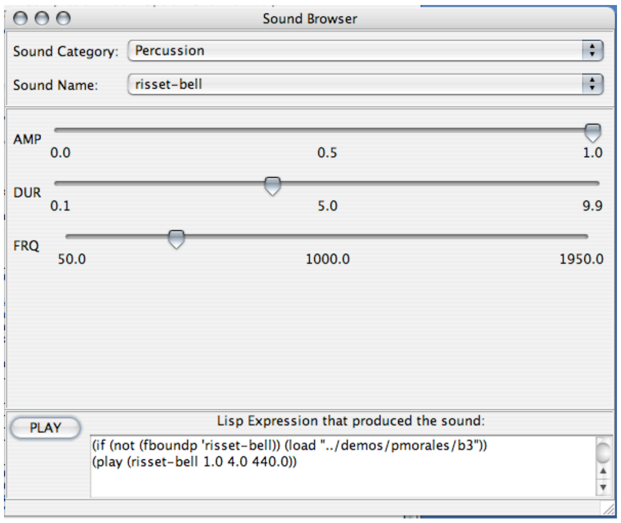
Figure 3. Equalizer interface from the Nyquist IDE.

As with the envelope editor, the EQ editor communicates with the Nyquist process to maintain a consistent view of a set of equalizers. In this case, equalizers are numbered, and the user applies an equalizer by calling, e.g. `eq-0`(*sound-expression*).

4.4. Browser

Nyquist has a number of different instrument definitions and libraries. While in the past, contributions were simply added to sub-directories named "demos" and "lib" included in the release, there is now a user-friendly browser that offers an index to sounds and examples. The browser (see Figure 4) uses Nyquist to synthesize sound examples. Examples are parameterized, and the browser displays sliders so that users can experiment with different settings. Code for any example is displayed, and it is easy to copy and save code, including custom parameter settings, for use in a composition. The browser works by loading a text description of examples; it is simple to extend the system by adding new descriptions for new sounds.

This is another way to bridge between a textual and a graphical environment. New users and novices can see directly how the graphical menu, slider, and audio interface are implemented in terms of text-based programs and commands. Everything is organized as an invitation to new users to probe deeper and to learn by doing and by example.



Sound Category: Percussion
Sound Name: risset-bell

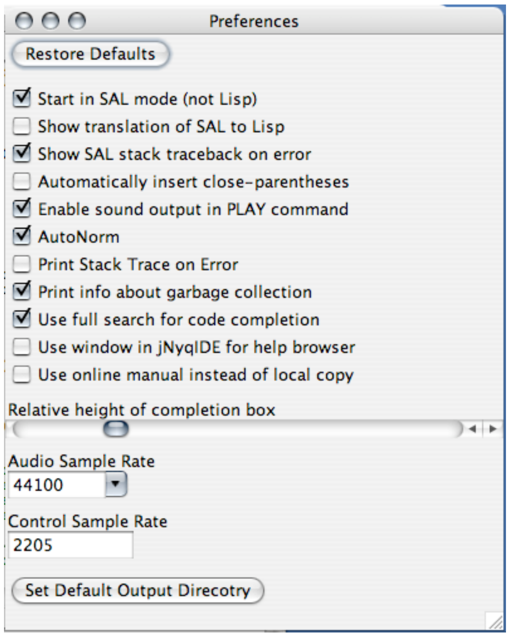
Lisp Expression that produced the sound:

```
(if (not (fboundp 'risset-bell)) (load "../demos/pmorales/b3"))
(play (risset-bell 1.0 4.0 440.0))
```

Figure 4. Sound Browser in the Nyquist IDE.

4.5. Preferences

Nyquist has many settings and options, generally controlled at the lowest level by setting global variables or calling functions. For example, the default sound sample rate is set by calling the function `set-default-sound-srate`. Many of these options are now easy to find and change in the "Preferences" window of the IDE. In addition to making changes in system behavior, settings are saved and restored each time the user runs Nyquist. This could always be accomplished by creating a custom initialization file loaded by Nyquist at start-up time, but in practice, few people ever did this, whereas users find it easy to customize the system now. Figure 5 illustrates the Preferences dialog window.



- ☑ Start in SAL mode (not Lisp)
- ☐ Show translation of SAL to Lisp
- ☑ Show SAL stack traceback on error
- ☐ Automatically insert close-parentheses
- ☑ Enable sound output in PLAY command
- ☑ AutoNorm
- ☐ Print Stack Trace on Error
- ☑ Print info about garbage collection
- ☑ Use full search for code completion
- ☐ Use window in jNyqIDE for help browser
- ☐ Use online manual instead of local copy

Relative height of completion box

Audio Sample Rate: 44100

Control Sample Rate: 2205

Figure 5. The Preferences window from the Nyquist IDE.

5. SAL

Basing Nyquist on Lisp has been a mixed blessing. On the one hand, the XLISP interpreter at the core of Nyquist has proven to be very portable, small, extensible, and free of any licensing fees. Lisp has also facilitated the implementation of some of the advanced features of Nyquist, including lazy evaluation of sounds and automatic garbage collection. On the other hand, many potential users have asked for a more familiar language. Although Lisp has many proponents, even most Lisp programmers will admit that infix operators for math would be nice, and fewer parentheses could enhance readability. The solution now being explored is SAL, a programming language designed by Rick Taube for his Common Music system. [15] To support SAL, Taube's SAL compiler was ported to XLISP and a "SAL mode" was added to the IDE to help users by highlighting keywords, suggesting proper indentation, offering "hints" in SAL syntax, and automatically running the SAL compiler when SAL commands are entered or SAL program files are loaded. Figure 6 shows a Nyquist program in both Lisp and SAL for comparison.

```
;; a Nyquist/Lisp example
(defun ex5 ()
  (play (seq (stretch 0.25
                      (seq (env-note c4)
                           (env-note d4)))
             (stretch 0.5
                      (seq (env-note f4)
                           (env-note g4)))
             (env-note c4))))

;; a Nyquist/SAL example
define function ex5()
  play seq(seq(env-note(c4),
               env-note(d4)) ~ 0.25,
           seq(env-note(f4),
               env-note(g4)) ~ 0.5,
           env-note(c4))
```

Figure 6. Example program in Lisp and SAL.

SAL is not completely independent of either Common Music or Nyquist, so there are some differences in the two SAL implementations. Some things supported in Common Music, in particular the process, are absent from Nyquist, and therefore there is no `define process` construct in SAL for Nyquist. On the other hand, Common Music does not support signal processing, so a bit of new syntax has been added to express Nyquist-specific concepts. One example, shown in Figure 6, is the addition of infix operators for stretch (~) as well as shift (`@`). These operators come from Arctic [3], the language that inspired Nyquist. (One might ask: Why not use Arctic syntax throughout instead of SAL? The answer is that SAL was designed as a replacement for Lisp syntax, so it maps nicely onto existing Lisp function names, control constructs, and data types. Arctic could probably serve as the basis for a new syntax for Nyquist, but it seemed wiser to build upon SAL's successful introduction in Common Music.)

6. Open Sound Control

Another recent extension to Nyquist is a basic facility for controlling Nyquist sounds via Open Sound Control (OSC). Since all signal processing and control is based on the `SOUND` datatype, Nyquist offers a function, `snd-slider`, that creates a sound whose value is determined in (near) real time by OSC messages. The idea is that there is an array of virtual sliders that can be set by OSC (or the IDE) and read by Nyquist. This allows Nyquist sounds to be controlled by external physical devices or programs. In the spring of 2007, students in my Introduction to Computer Music class collaborated with students in Pamela Jenning's course on Physical Computing to build physical controllers connected to Nyquist synthesis algorithms. For this work, PIC microcontrollers sent serial data over USB to a computer running a simple serial-to-open-sound-control program (included in the Nyquist distribution), which then transferred data via OSC to Nyquist. Nyquist is not really designed to be a real-time program, so latency tends to be high, but it is nonetheless very powerful for prototyping or for getting values from the real world into Nyquist.

7. IMPLEMENTATION

Nyquist runs on Windows, Macintosh, and Linux systems. The Nyquist IDE is implemented in Java. This may seem a strange choice given that Nyquist is created with C and Lisp. Are three languages really necessary? In retrospect, Java was a very good choice. One could argue about the best system, but the cross-platform GUI-building tools are at least powerful, well-documented, and generally familiar to students. There are thousands of code examples on the web that are easily searchable. Another very important feature is Java's exception-handling. Especially in GUI-intensive programs, bugs can often be survived simply by handling an exception and continuing to run. With garbage collection, Java is very forgiving of failures to release memory structures and resources. As new components are added to the IDE (bugs included), the overall IDE continues to operate reliably. We do not recommend creating or re-leasing buggy systems, but it is much better when bugs do not result in a sudden program exit while users are at work. Java has the drawback of unreliable realtime performance, but this is also an issue with Nyquist and not really necessary for the Nyquist IDE. Overall, the typical performance feels very responsive.

8. AUDACITY AND NYQUIST

Audacity [10] has become an extremely popular audio editor. It includes a copy of Nyquist, disguised as a plug-in scripting language. Users can write Nyquist expressions to generate or process sounds, which are streamed from user-selected regions of sound files into Nyquist and back into Audacity. There are plug-ins to help label silence and segment recordings, to generate custom fades, to create tones, perform granular synthesis, etc. This facility is completely independent of the Nyquist IDE, but it is worth mentioning because it is an alternative working environment. Audacity is excellent for editing and viewing digital audio, but the support for language-based synthesis and composition is very limited. Some people find the ability to load and view sounds very important to their development process and are willing to forgo the IDE to gain access to the editing functions of Audacity.

With Audacity and the Nyquist IDE, we see two ways of working. In Audacity, the focus is on audio; preparing Nyquist programs is an external function. With the IDE, the focus is on program development; viewing or editing audio is an external function. Obviously, some way to integrate the processing of audio via programming and via direct manipulation editors could be of great value.

9. FUTURE WORK

Nyquist and its IDE have come a long way, but there are still many additions envisioned. There have been several attempts to create some sort of time-line based graphical composition interface. One of these was based on the UPIC system, with very encouraging results. The prototype was created in C++ for Windows, so it is not integrated with the Nyquist IDE. Hopefully, there will be a port to Java so that a UPIC-like interface will be available within the current environment. Another student project created a prototype for a graphical score editor where boxes representing sound objects are placed along a time line, similar to the Animal system. [8] This prototype requires further development to be really useful.

Another score-like interface could be introduced for drum patterns. Currently, Nyquist includes a simple text representation for entering rhythmic patterns and assigning them to drum sounds, and there is a nice library of drum samples created by Phil Light. An effective and flexible graphical editor for drum patterns along with some libraries of ready-to-use patterns would be useful to many.

There are more graphical interface tools that could be introduced. Mechanisms for a slider interface exist, but are not yet supported by the GUI. Ideally, one should be able to link instrument parameters to graphical sliders to assist in fine-tuning sounds. This is already possible in the Sound Browser shown in Figure 4, but these sliders are configured within the IDE and are not convenient for developing new sounds. Furthermore, it is already easy to simply edit text and reload code, so to be of any use, a slider interface must be very simple and efficient to set up.

The Open Sound Control Interface has demonstrated that near-real-time interaction is possible with Nyquist. With a few more primitives for real-time interaction (perhaps modelled on those of csound), it could be interesting to support MIDI interfaces for control, especially to extend the working environment with physical knobs, sliders, and other controllers. Along those lines, the mouse and computer keyboard can also be used as effective controllers with the right support from the IDE.

10. CONCLUSIONS

Nyquist has evolved from a very powerful but rather hard-to-use text-based programming language to a powerful environment for composition and sound synthesis. Our experience indicates that a task-oriented interface can greatly enhance the usability of a language like Nyquist. It is interesting that some “tasks” are not expected ones such as designing an envelope or a filter, but programming tasks such as finding the correct name and parameter list for a function or balancing parentheses. The Nyquist IDE syntax editor and hinting mechanism are relatively simple programs, building upon existing text objects and documentation processors, but the support for even experienced Nyquist programmers is substantial. As Nyquist gains more widespread use, especially as a scripting language for Audacity, the IDE helps by providing examples, showing connections between graphical manipulation and text-based control, and by assisting with language syntax. This in turn makes Nyquist more accessible to more users. Users, especially students, have made great suggestions for improvements and implemented many of them. Thus, Nyquist and its development environment continue to grow in response to perceived needs and helpful contributions.

11. ACKNOWLEDGEMENTS

Nyquist would not exist without the contributions of many students and collaborators, including: Chris Fraley, George Polly, Peter Velikonja, Dean Rubine, Joe Newcomer, Cliff Mercer, Ning Hu, Dave Mowatt, Dominic Mazzoni, Chris Tchou, Morgan Green, Eli Brandt, Pedro Morales, Eduardo Miranda, Ann Lewis, Erich Neuwirth, Philip Yam, Dave Borel, Adam Hartman, Stephen Mangiat, Phil Light, Andreas Pfenning, Chris Yealy, Derek D’Souza, Daren Mackuck, Michael Rivera, Priyanka Raghavan, and Austin Sung. Thanks are also due to the creators of csound, Music4BF, STK, and Common Music, from which Nyquist inherits. Finally, thanks to IBM, Yamaha, the National Science Foundation, Carnegie Mellon, and others who have helped support various facets of Nyquist development through the years.

12. REFERENCES

[1] Cook, P. and Scavone, G., “The Synthesis ToolKit (STK),” *Proceedings of the International Computer Music Conference,* International Computer Music Association, (1999), pp. 164-166.

[2] Dannenberg, R. and Fraley, C. “Fugue: A Signal Manipulation System with Lazy Evaluation and Behavioural Abstraction,” *1989 International Computer Music Conference*, Computer Music Association (October 1989), pp. 76-79.

[3] Dannenberg, R., McAvinney, P., and Rubine, D., "Arctic: A Functional Approach to Real-Time Control," *Computer Music Journal*, 10(4) (Winter 1986), pp. 67-78.

[4] Dannenberg, R., "The Implementation of Nyquist, A Sound Synthesis Language," *Proceedings of the 1993 International Computer Music Conference*, International Computer Music Association (September 1993), pp. 168-171.

[5] Dannenberg, R., "The Implementation of Nyquist, a Sound Synthesis Language," *Computer Music Journal*, 21(3) (Fall 1997), pp. 71-82.

[6] Eaglestone, B., Ford, N., Holdridge, P., and Carter, J., "Are Cognitive Styles an Important Factor in the Design of Electroacoustic Music Software?," *Proceedings of the 2007 International Computer Music Conference,* International Computer Music Associaation, (2007), pp. 466-473.

[7] Kunze, T. and Taube, H., "SEE--A Structured Event Editor: Visualizing Compositional Data in Common Music," *Proceedings of the International Computer Music Conference*, International Computer Music Association (1996), pp. 63-66.

[8] Lindeman, E., "ANIMAL: A Rapid Prototyping Environment for Computer Music Systems," *Proceedings of the International Computer Music Conference*, International Computer Music Association (September 1990), pp. 241-244.

[9] McCartney, J. "SuperCollider: A New Real Time Synthesis Language," *Proceedings of the 1996 International Computer Music Conference.* International Computer Music Association (1996), pp. 257-258.

[10] Mazzoni, D. and Dannenberg, R., "A Fast Data Structure for Disk-Based Audio Editing," *Computer Music Journal,* 26(2), (Summer 2002), pp. 62-76.

[11] Orlarey, Y., Fober, D., and Letz, S., "Syntactical and Semantical Aspects of Faust," *Soft Computing* 8(9), (2004), pp. 623-632.

[12] Piche, J. and Burton, A., "Cecilia: A Production Interface to Csound", *Computer Music Journal* 22(2), (Summer 1998), pp. 52-55.

[13] Pope, S. and Ramakrishnan, C., "Recent Developments in Siren: Modeling, Control, and Interaction for Large-scale Distributed Music Software," *Proceedings of the 2003 International Computer Music Conference,* International Computer Music Association (2003), pp. 5-9.

[14] Psenicka, D. "FOMUS, a Music Notation Software Package for Computer Music Composers," *Proceedings of the International Computer Music Conference*, International Computer Music Association (2007), pp. 75-78.

[15] Taube, H. "Common Music: A Music Composition Language in Common Lisp and CLOS," *Computer Music Journal 15*(2), (1991), pp. 21-32.

[16] Vercoe, B., "The Canonical CSound Reference Manual Version 5.07. Edited by J. ffitch, J. Piché, P. Nix, R. Boulanger, R. Ekman, D. Boothe, K. Conder, S. Yi, M. Gogins, A. Cabrera, F. Pinot, and A. Kozar. URL (accessed 7 Feb 2008): http://www.csounds.com/manual/html/index.html.

[17] Wang, G. and Cook, P., "Audicle: A Context-sensitive, On-the-fly Audio Programming Environ/mentality," *Proceedings of the International Computer Music Conference*, International Computer Music Association (2004), pp. 256-263.

[18] Zicarelli, D. "An Extensible Real-Time Signal Processing Environment for Max," *Proceedings of the 1998 International Computer Music Conference.* International Computer Music Association (1998), pp. 463-466.