UNIVERSITY OF CALIFORNIA

Santa Barbara

Real-Time Meta-Programming for Interactive Computational Arts

A Dissertation submitted in partial satisfaction of the

requirements for the degree Doctor of Philosophy

in Media Arts & Technology

by

Graham Wakefield

Committee in charge:

Professor Curtis Roads

Professor JoAnn Kuchera-Morin

Professor Marcos Novak

Doctor Matthew Wright

September 2012

The dissertation of Graham Wakefield is approved.

Professor JoAnn Kuchera-Morin

Professor Marcos Novak

Doctor Matthew Wright

Professor Curtis Roads, Committee Chair

July 2012

Real-Time Meta-Programming for Interactive Computational Arts

Copyright © 2012

by

Graham Wakefield

ACKNOWLEDGEMENTS

First I must thank my parents for having given my potential the early chances it needed, despite the sacrifices this required, for which I will be grateful for as long as I am conscious. I thank my committee for their dedication and patience, and for being inspiring and remarkable examples of humans. Special thanks for all the fellow MATians; particularly collaborators Wesley Smith, Lance Putnam, and Charlie Roberts, for great times of work, rest and play. Much appreciation to the Cycling '74 crew, especially David Zicarelli, Joshua Clayton, and Wes again, for their understanding, and for managing to make a successful company feel like a bunch of friends having fun. Most of all I thank my wonderful other half Haru Ji, in untold ways.

The development of *LuaAV* and *Artificial Nature* have been graciously supported by the AlloSphere Research Group at the Media Arts & Technology graduate program of the University of California Santa Barbara, under NSF grants IIS-0855279 & IIS-1047678, and a graduate fellowship from the Robert W. Deutsch Foundation.

VITA OF GRAHAM WAKEFIELD

September 2012

EDUCATION

- Bachelor of Arts (Hons) in Philosophy, University of Warwick, UK, June 1997
- Master of Music in Composition (with Distinction), Goldsmiths College University of London, UK, September 2004
- Master of Science in Media Arts and Technology, University of California, Santa Barbara, USA, September 2007
- Doctor of Philosophy in Media Arts and Technology, University of California, Santa Barbara, USA, September 2012

PROFESSIONAL EMPLOYMENT

- 2012-present: Post-Doctoral Research Assistant, AlloSphere, California NanoSystems Institute, University of California, Santa Barbara
- 2008-present: Software Developer, Cycling '74, San Francisco
- 2011-2012: Research Assistant, AlloSphere, California NanoSystems Institute, University of California, Santa Barbara
- 2010-2011: Research Assistant, Sanford Burnham Center for Nano-Medicine Research, University of California, Santa Barbara
- 2008-2010: Faculty Lecturer, Southern California Institute for Architecture (SCI-Arc), Los Angeles
- 2008-2010: Research Assistant, AlloSphere, California NanoSystems Institute, University of California, Santa Barbara
- 2007-2012: Software Developer (by contract), WorldViz, Santa Barbara
- 2006-2008: Teaching Associate, College of Creative Studies, University of California, Santa Barbara
- 2005: Teaching Assistant, Media Arts & Technology, University of California, Santa Barbara

- 1999-2004: Web and Interactive Media Manager, Stafford Long & Partners, London

PUBLICATIONS

- Roberts, C., Wakefield, G., and Wright, M. Mobile Controls On-The-Fly: An Abstraction for Distributed NIMEs. Proceedings of New Instruments for Musical Expression (NIME), 2012
- Wakefield, G. and Smith, W. Cosm: A Toolkit for Composing Immersive Audio-Visual Worlds of Agency and Autonomy. Proceedings of the International Computer Music Conference (ICMC), 2011
- Ji, H., Wakefield, G. City Life. In The Urban Organism exhibition catalog (Seoul, Korea). New Media Art Research Association (NMARA), 2011
- Putnam, L., Wakefield, G., Ji, H., Alper, B., Adderton, D., Kuchera-Morin, J. Immersed in Unfolding Complex Systems. Beautiful Visualization: Looking at Data through the Eyes of Experts. O'Reilly Media, Inc., 2010
- Wakefield, G., Smith, W. and Roberts, C. LuaAV: Extensibility and Heterogeneity for Audiovisual Computing. Proceedings of the Linux Audio Conference (LAC), 2010
- Roberts, C., Wright, M., Kuchera-Morin, J., Putnam, L. and Wakefield G. Dynamic Interactivity inside the AlloSphere. Proceedings of New Instruments for Musical Expression (NIME), 2009
- Smith, W. and Wakefield, G. Augmenting Computer Music with Just-In-Time Compilation. Proceedings of the International Computer Music Conference (ICMC), 2009
- Wakefield, G. Makeshift, Machinic / Open. In Machine Dreams, ed. Jeon, B. S. and Ji, H. (ISBN: 978-89-963788-0-8). KoIAN, Seoul, Korea, 2009
- Thompson, J., Kuchera-Morin, J., Novak, M., Overholt, D., Putnam, L., Wakefield, G., Smith, W. The Allobrain: An Interactive, Stereographic, 3D Audio Immersive Virtual World. International Journal of Human Computer Studies Volume 67, Issue 11 (Special issue on Sonic Interaction Design), Elsevier, 2009
- Wakefield, G. and Ji, H. Artificial Nature: Immersive World Making, (Proceedings of the Evolutionary Music and Art Workshop, Tübingen, 2009), LNCS5484. Springer-Heidelberg, 2009

- Smith, W. and Wakefield, G. Computational Composition and Creativity. Proceedings of the Media Arts Science and Technology Conference, Santa Barbara, 2009

- Ji, H. and Wakefield, G. Artificial Nature: Research and Development. The 2nd International Conference on Media Art and Information Aesthetics (MAIA), CAFA Art Museum Beijing China, 2009

- Ji, H. and Wakefield, G. Artificial Nature. Proceedings of the Media Arts Science and Technology Conference, Santa Barbara, 2009

- Ji, H., Wakefield, G. Artificial Nature: Fluid Space. ACM SIGGRAPH ASIA 2009 Art Gallery & Emerging Technologies: Adaptation (Yokohama, Japan). ACM, 2009

- Ji, H., Wakefield, G. Artificial Nature: Fluid Space. In A.L.I.C.E. MUSEUM exhibition catalog (Seoul, Korea). Art Center Nabi, Seoul, Korea, 2009

- Smith, W., Wakefield, G.: Computational Audiovisual Composition using Lua. Transdisciplinary Digital Art: Sound, Vision and the New Screen, Springer, 2008

- Wakefield, G., Kuchera-Morin, J., Novak, M., Overholt, D., Putnam, L., Thompson, J., Smith, W.: The AlloBrain: an Interactive Stereographic, 3D Audio Immersive Environment. Proceedings of the 2008 ACM Sonic Interaction Design Computer Human Interaction Conference, Florence, 2008

- Ji, H. and Wakefield, G. Artificial Nature as an Infinite Game, (Proceedings of the 2008 International Symposium of Electronic Arts), ISEA Singapore, 2008

- Ji, H. and Wakefield, G. Artificial Nature as an Infinite Game. In Proceedings of ASIAGRAPH 2008, Shanghai, China 2008

- Ji, H., Wakefield, G. Artificial Nature: Fluid Space. In Universal Electronic Art exhibition catalog (Seoul, Korea). Seongnam Art Center, Seongnam, Korea, 2008

- Amatriain, X., Castellanos, J., Höllerer, T., Kuchera-Morin, J., Pope, S., Wakefield, G. and Wolcott, W.: Experiencing Audio and Music in a Fully Immersive Environment. Sense of Sounds, Lecture Notes in Computer Science, Springer-Verlag, 2008

- Zwick, R., Sklar, J. C., Wakefield, G., Hamilton, C., Norman, A., Folsom, D. Instructional Tools in Educational Measurement and Statistics (ITEMS) for School Personnel: Evaluation of Three Web-Based Training Modules.

Educational Measurement: Issues and Practice vol. 27(2). National Council on Measurement in Education, Blackwell Publishing, 2008

- Smith, W., Wakefield, G.: Real-Time Multimedia Composition using Lua. Proceedings of the Digital Art Weeks, ETH Zurich, 2007

- Wakefield, G., Smith, W.: Using Lua for Multimedia Composition. Proceedings of the International Computer Music Conference (ICMC) 2007, International Computer Music Association, 2007

- Wakefield, G. Vessel: A Platform for Computer Music Composition, Interleaving Sample-Accurate Synthesis and Control. M.S. Thesis, University of California Santa Barbara, 2007

- Wakefield, G.: Real-Time Third-Order Ambisonic Extensions for Max/MSP with Musical Applications. Proceedings of the International Computer Music Conference (ICMC) 2006, ICMA, 2006

ABSTRACT

Real-Time Meta-Programming for Interactive Computational Arts

by

Graham Wakefield

In the interactive computer arts, any advance that significantly amplifies or extends the limits and capacities of software can enable genuinely novel aesthetic experiences. Within compute-intensive media arts, flexibility is often sacrificed for needs of efficiency, through the total separation of machine code optimization and run-time execution. Compromises based on modular run-time combinations of prior-optimized 'black box' components confine results to a pre-defined palette with less computational efficiency overall: limiting the open-endedness of development environments and the generative scope of artworks. This dissertation demonstrates how the trade-off between flexibility and efficiency can be relaxed using *reflective meta-programming* and *dynamic compilation*: extending a program with new efficient routines while it runs. It promises benefits of more open-ended real-time systems, more complex algorithms, richer media, and ultimately unprecedented aesthetic experiences.

The dissertation charts the significant differences that this approach implies for interactive computational arts, builds a conceptual framework of techniques and requirements to respond to its challenges, and documents supporting implementations in two specific scenarios. The first concentrates on open-ended creativity support within always-on authoring environments for studio work and live coding performance, while the second concerns the open-endedness of generative art through interactive, immersive artificial-life worlds.

Contents

Chapter 1

Introduction

Computer artists make art by making and using software. What an artist or composer can do, what the artwork itself can do, and thus ultimately what aesthetic results we can experience, are all bounded by capabilities of the software. These limits include the efficiency of its real-time performance, and also the flexibility of possible paths it can take.

Any advance that amplifies or extends the limits and capacities of software for computational arts makes possible genuinely novel aesthetic experiences. A more efficient implementation of a program can synthesize more sounds, render more pixels and simulate more complex scenes. A more flexible program can better accommodate unpredictable users and move through a larger spaces of spontaneous behavior. Unfortunately however, efficiency and flexibility are conflicting concerns in conventional approaches to software development. Many systems in use today often present a *modular* interface in which a program can be flexibly assembled from component modules (representing key concepts or structures of the problem domain). Conventionally the implementation of modules is hard-wired and specialized for efficiency. This conventional approach cannot view a combination of these blocks as a computational whole, as would be the case if the entire system were written by hand without thinking in modular terms.

This dissertation explores an alternative approach that leverages techniques of *reflective meta-programming* and *dynamic compilation*. A *meta-program* is a program that reads, modifies or creates other programs, relying on the basic fact of computation that *programs are data*. A *reflective* program has some ability to observe and possibly manipulate itself; thus a *reflective meta-program* is one that can rewrite itself as it runs. A *compiler* is simply a meta-program that translates, specializes, and validates an input program in some language into a functionally equivalent output program in some other, such as fast machine code. Compilation is *dynamic* if it can be invoked and specified during the normal execution of a program. These techniques permit modular components to retain a cognitive role without sacrificing global efficiency nor the ability to continuously redefine the system while it runs.

This dissertation thus puts conventional software practice for interactive computational arts into question, motivated by the possibility to push back the boundaries of efficiency and flexibility. Where the broad term *interactive computational arts* is read, it can be clarified by reference to the following two scenarios (which have been the focus of my research):

1. *Authoring environments.* Supporting experimentation interactively with human decisions: such as a composer or artist in the studio working on the materials and structures of a composition, or a performer improvising in a live context.

2. *Open worlds.* Generative art works evolving interactively with human responses: such as evolutionary music and art, and immersive interactive

worlds.

Both scenarios involve creative results that emerge from a cyclic dialogue between human and machine, mediated by software. The implementation of this software must navigate the conflicting concerns of efficiency and flexibility in encoding and execution for the sake of immediacy and open-endedness in the human-computer interface. It must also present an interface that closely follows the concepts of the domain, without over-prescribing and overly narrowing its scope.

Although this research focuses on artistic applications, the solutions presented could have a major impact in other areas, such as scientific visualization, simulation and modeling.

1.1 Problem statement

Many of the 'creativity support tools' available to the media artist and composer today aim toward open-endedness within a real-time interface. The user can construct new systems within an always-on, immediate environment. The fast test-evaluate response time is conducive to sketching, exploratory ideas, and also improvisation. Similarly, generative artists pursuing interactive worlds of evolving complexity seek open-ended spaces of behavior with uninterrupted execution. In both cases open-endedness is in tension with efficient performance. Nevertheless, many of these systems are more rigid or less efficient than they need to be, due to conventional strategies of software design.[1]

This dissertation is concerned with an as yet rarely exploited opportunity to reconcile these concerns of performance and open-endedness in computational arts by bringing the capabilities of *reflective meta-programming and dynamic compi-*

[1]These issues are dealt with in detail in Chapter 2.

lation into real-time systems. This means efficient programs that become new efficient programs as part of their normal run-time execution. This dissertation illustrates how these capabilities can relax the conflict between goals of efficiency and open-endedness over fragments of a running program.

However, despite these potential benefits, and a fifty-year history of research in Computer Science, examples of reflective dynamic compilation remain remarkably rare within the development of compute-intensive creativity support tools, and even less within generative artworks. The principal goal of the dissertation is thus to chart the significant differences that reflective meta-programming and dynamic compilation imply for interactive computational arts, including both creative potentials to be realized and software design challenges to overcome. The motivation is to increase understanding and capabilities regarding open-ended flexibility and efficient immediacy for exploratory, creative ends; whether human-driven or artificial. The suggested implications for the two scenarios considered are broader support for creative exploration, and more genuinely open-ended, immersive worlds.

Reflective meta-programming and dynamic compilation present an opportunity to greatly increase the *computational diversity* of interactive, data-intensive media, and thus make available new ranges of possible aesthetic experiences of computational arts. Meta-programming is no stranger to generative art–grammar-based rules such as *L-Systems* (Prusinkiewicz and Lindenmayer, 1996) rely upon it–but if the real-time constraints of interactivity and immersive media can be blamed for rigidity, then the combination with dynamic compilation offers a significant contribution to the artist's palette. Through intimate interactions between human and machine, we might pursue the dream of a world that is ever being born (Dorin, 2005). The broader motivation is to encourage the development of art that Frieder Nake described as *"adequate"* to the computational medium (Nake, 2007).

The emphasis throughout is applicability to the specific domain of interactive computational arts, including mapping into existing media interfaces, algorithms and libraries, being sensitive to the temporal demands of human perception, and supporting or extending creative practices of composition and performance. The primary intended readership can be found within the artist-programmer[2] community.

Targeting a specific problem domain requires understanding the evaluation constraints of the domain (such as being sensitive to the temporal demands of human perception for real-time media processing), as well as presenting flexibility in terms of the basic conceptual structures and operations in the domain (for example, oscillators and filters in audio signal processing). These factors of evaluation and interface distinguish the task from the design of generic programming languages, where goals of flexibility and efficiency may lead to different implementation strategies.

The analysis is borne out through demonstrations of empirical practice: systems and artworks in which massive quantities of data must be processed in responsive times and in which a sense of unbroken open-ended exploration is essential. Two such scenarios in particular are considered: musical composition and performance using arbitrary synthesis processes, and immersive nature-like virtual worlds. These demonstrations make use of dynamic compilation to afford greater run-time flexibility and efficiency than could previously have been achieved. The demonstrations are described and analyzed in detail, serving as reference material

[2]A detailed investigation into artist-programmers, and their medium, can be found in (McLean, 2011).

for the broader community.

1.2 Background

Efficiency is of paramount importance for any system that involves real-time interaction with compute-intensive processes. Examples of such processes include the analysis, transformation and synthesis of audio and video streams, large geometries and particle systems in 3D worlds, and large-scale multi-agent simulations.

Generating audio-visual media to satisfy the temporal and perceptual limits of humans is a demanding task. “Ear-limited” audio requires streams at more than double our perceptual threshold: typically 44100 32-bit samples per second (or better) per loudspeaker, and multiple loudspeaker streams are needed to recreate psychoacoustic localization cues (up to triple figures in a space such as the AlloSphere, as described in Amatriain et al., 2008). Synthesis and spatialization algorithms must generate new data at this rate or better, and do so within latencies of tens of milliseconds for interactivity to feel immediate. Contemporary HD graphics run at 1920x1080 pixel resolutions per screen, with active stereographics requiring twice this number of pixels refreshed at 60 frames per second or better to maintain the illusion of continuity (through the ‘persistence of vision’ effect): nearly 250 million new red, green and blue pixel values per second for a single screen. For interaction to be comprehensible and natural, the round-trip of user input, processing, and display of results may need to occur on scales of milliseconds (MacKenzie and Ware, 1993).

It is well acknowledged that achieving these deadlines and pushing back the limits of how much can be processed while maintaining low latency introduces significant forces on software design (Dannenberg, 2005). If results cannot be computed within the available slot of time, then they must be determined in advance, i.e. approaching optimal performance requires prior specialization.

In particular, a generic program can be specialized by reducing the range of inputs, outputs and behaviors it supports, and also by rewriting it in terms of the specific hardware it will run on. The conventional technique of *(static) compilation* performs these specializations ahead-of-time: a program is entirely pre-determined and translated to machine code before execution begins. This compilation occurs separately for each different program. Since the entire space of possible control flows is specified in advance, algorithms can be restructured without changing the semantic behavior. Each resultant executable is written to better suit the program's needs in terms of the features and capabilities of the hardware, leading to greatly increased run-time performance (Aho et al., 1986).

However, satisfying performance constraints in this way results in limitations for creative, run-time flexibility.

1.2.1 Creative flexibility

Creative work must be expressed through the available tools and models of representation. Yet it is characterized by qualitative goals of *open-endedness*–variability, diversity and dynamism–for which a strict adherence to pre-defined *a priori* constraints is unacceptable (Gabriel and Sullivan, 2010). Relaxing the a priori limitations on the degree of variability in a system implies a movement of decision-making from ahead-of-time to run-time (de Campo, 2004) that conflicts with the pre-specialization of static compilation as outlined above. Creative goals are often not well-defined before starting to work (particularly in improvisation; Hamilton, 2000), but it is difficult to optimize a problem that is not yet clearly stated!

The need for “wide walls and high ceilings” has been identified as a goal of “creativity support tools” (Shneiderman, 2007): artists do not want to feel ‘steered’ by software into the production of hackneyed clichés. Not only must an artist know the limitations and capacities of models to prevent derivative work, but also a discipline must actively evolve the models available from the territories of the known or probable to those of the possible and improbable (Boden, 2007). A system whose space of possible paths is narrowly pre-defined can only lead to discovery within that pre-defined space, whereas a system that is more openly defined can lead to results far beyond any expectations of the system’s author(s).

Making changes to a pre-compiled program requires a slow process of stopping and restarting the system, and engaging with a distinct cognitive space of software development, obstructing exploratory discovery. The flexibility of run-time decision-making demands continuity of execution (without stopping and restarting). Not only is this essential for interactive installations, and novel performance forms such as live-coding (Collins et al., 2003), but even when computation is used as part of an artist’s process of composition and prototyping, a flexible ‘always-on’ environment confers significant advantages. It puts us in a perceptually more intimate relationship with our medium, while the lowered cognitive cost and rapid turnaround promotes experimentation and discovery: an in-the-moment insight can be implemented and tested immediately, before the vision is lost (Shneiderman, 2009). For example, in a presentation at the Game Developer’s Conference 2012, Niklas Frykholm of Bitsquid emphasized the importance of in-game editing to avoid the lengthy (seconds to minutes) “iteration treadmill” of rewriting, re-compiling, reloading and verifying, identifying the need for sub-second iterations for fluid design.[3]

[3]http://www.bitsquid.se/presentations/cutting-the-pipe.pdf

A similar tension recurs in certain forms of interactive art, such as immersive, evolutionary artworks. A motivating goal of such works is that artificial life organisms evolve new, emergent behaviors (McCormack, 2005), demanding an open space of discovery in which pre-determined constraints are minimized (Machado et al., 2004). Overly prescribed generative artworks have been criticized for being “crystalline and impervious”: too much predetermination of algorithms leads to fixed spaces of possible responses in which vast classes of possible behaviors are simply inaccessible (Whitelaw, 2005). The algorithmic open-endedness desirable for evolutionary, generative art finds itself at odds with the efficiency-driven static tendency of high-resolution graphics, rich audio, and large-scale simulation desirable for interactive, immersive worlds (Ji, 2011).

1.2.2 Computational flexibility

Run-time decision-making can be broadly classed into four categories of increasing flexibility in spaces of data and control flow:

1. At the simplest level, nothing is dynamic: all input is specified prior to execution, and all output is read after execution completes. This is known as ‘batch-oriented’ computing, and was the primary concern of computer science for the first decades of its evolution (Costa and Dimuro, 2005). The corresponding space of execution is an entirely predetermined flow of predetermined data.

2. The next level introduces dynamic input and output data-streams during run-time, flowing through a pre-defined process structure. This level is characteristic of stateless dynamical models, such as signal processing functions: variable data flows through a pre-defined static graph in which all paths are

active.

3. Flexibility can be increased by selecting between different data-flow and control-flow paths at run-time. Now in the executing program variable data flows through a variable subset of a pre-defined static network. This is characteristic of most of the software we use today.

4. A vastly greater degree of flexibility is possible if new control and data flow paths can be added at run-time. It allows constructive, not just parametric, transformations. This capability is usually associated with higher-level interpreted programming languages, development environments and virtual machines (but is also effectively what an operating system is). The corresponding space of execution comprises variable data flows through a graph which is itself changing; and which in the limit corresponds to the entire space of possible machines.

The latter algorithmic flexibility thus goes to the heart of what computation is. The differentiating feature of the computer as a tool is that it can, theoretically, emulate any other formally specifiable machine (Turing, 1937). Christopher Langton notes "computers should be viewed as second-order machines–given the formal specification of a first-order machine, they will 'become' that machine. Thus the space of possible machines is directly available for study, at the cost of a mere formal description: computers 'realize' abstract machines" (Langton, 1996).

1.2.3 Interpretation

The clear implication is that, in theory, a program can spawn or metamorphose into any other formally describable machine. This capability is best demonstrated by *interpreted* languages such as LISP (McCarthy, 1960) and Scheme (Sussman, 1975). In contrast to static compilation, in the interpreted approach a program is executed through a pre-compiled virtual machine (VM) tailored to the language of the input program. The VM redirects its own execution according to the input program, instruction by instruction, which offers much greater flexibility at run-time. Specifically, new chunks of code can be added with ease while a program runs, effectively transforming it into a new program, while existing state is preserved. These new program fragments could be entered by a user through a 'read-eval-print-loop' (REPL) or graphical interface. This direct engagement can move into quite different directions than had been preconceived.

Alternatively, new program fragments can be generated as the product of the program's own execution, implying a nonlinearity of program spaces analogous to the numerical nonlinearities of complexity theory. This kind of enlargement of possibility spaces is vital for generative arts, where the results of processes are fed back into influencing future decisions, and an open-ended space of exploration is considered essential (Machado et al., 2004). The biologically inspired variants of evolutionary and Artificial Life art in particular can leverage self-generation and self-evaluation of processes to more closely approach a generative creativity associated with nature (Bedau et al., 2000; McCormack, 2005).

However, since a virtual machine is not tailored to any specific program's needs, and the paths taken are indeterminate with a much higher granularity, the result is almost always significantly less efficient than a statically compiled, equivalent program.

In summary, where the interpreter favors flexibility, the compiler favors efficiency. If a compiled program is like a phrasebook of standard expressions, supporting rapid communication within a limited scope, an interpreted program is like a dictionary with which, by greater effort and time, a much broader variety of expressions can be made, including phrases that had not been expressed before.

1.2.4 A modular compromise

Some creativity support tools in use today present a compromise efficiency and flexibility, in which some parts of a program are hard-wired and specialized, while other parts remain more malleable. Implementations of this strategy may take the form of an optimized, pre-compiled run-time engine under the control of an embedded dynamic language[4], or may present a dynamically configurable interface (such as a graphical programming environment) in which a library of precompiled 'black box' pre-compiled modules can be modularly interconnected. This modular approach supports flexibility, and also brings important cognitive benefits: it enables users to break down problems into palatable sub-components, and to express domain-specific systems in terms of component concepts tailored to the domain. Nevertheless, in conventional software, the design of the modular system–the boundaries between pre-compiled and dynamic parts, the choice of pre-compiled modules, and the dynamic parameters they support–must still be determined in advance. By creating a domain-specific set of higher-level primitives implemented as atomic blocks of software, the conventional approach cannot view an implemented system with a combination of these blocks as a computational whole, as would be the case if the entire system were written by hand without thinking in modular terms.

To illustrate, audio synthesis authoring environments such as Max/MSP and PD (Puckette, 1996; Zicarelli, 2002) are considered a good examples of 'creativity support tools' (as opposed to 'productivity support tools'), since through a mod-

[4]This is the case for many game engines, such as Unity: http://unity3d.com/ (accessed 2012).

ular interface they can offer the creative user a much more flexible, open-ended space of possible algorithms (Shneiderman, 2007). A program or algorithm is represented by visual connections between pre-compiled processing objects drawn from an extensive palette of modules. The algorithm metamorphoses into a new algorithm at each edit, without pausing execution. Nevertheless, a specialized pre-compiled program for one of these algorithms will almost certainly outperform the corresponding implementation in Max or PD (hence Max and programs like it are sometimes considered prototyping environments).

Furthermore, even with this modular style, the implementation precludes whole classes algorithms. For example, most Max-like programs are incapable of constructing a user-defined audio processes involving a single-sample feedback loop (a fundamental component of audio filters) because of an implementation detail (buffered signal processing) necessary for efficient real-time audio (Dannenberg, 2005). Consequently, the discovery of new kinds of filters and many other interesting processes are precluded from the space of behaviors the interface supports. Authoring new filters requires leaving the creative environment to write and statically compile separate binary libraries in C, with a correspondingly cumbersome process of test and evaluation, and requiring non-trivial, specialized skills.

However, as we shall see, these problems are not inherent to a modular *interface*; rather they are inherent to a modular *implementation*. We are used to taking interface and implementation as one-and-the-same in everyday life. With most physical systems, including modular systems such as LEGO bricks and electronic circuits, we cannot separate the interface from the implementation. But the theory of computation is built upon the very possibility of separating specification and behavior. An important goal of this dissertation is thus to isolate the benefits of a modular interface from the restrictions of a modular implementation.

1.2.5 Reflective meta-programming and dynamic compilation

The problematic rigidity here is simply an artifact of software design practice as it has developed in the large. It is only by convention that interfaces are tied to specific implementations, that the algorithmic optimizations of compilation are separated from run-time executions, and that compilation is a process performed before starting to run a program.

Dynamic compilation (sometimes also known as run-time compilation or just-in-time (JIT) compilation) offers an alternative strategy in which the efficient native-compiled portions of a program are variable at run-time. It combines the efficiency associated with static compilation with the flexibility and temporal continuity associated with interpretation, by making a compiler available to the virtual machine and its programming interface, and inserting it within a feedback loop of code-generation and evaluation. Input program fragments can be translated into efficient representations for the hardware and the resultant machine code linked back into the same application that instigated compilation, without breaking execution. This implies that the program can extend itself with optimized capabilities that were not specified before starting execution (see Figure 1.2.1).

Dynamic compilation need not abandon the cognitive benefits of a modular interface. Instead it promises the possibility of designing and installing new efficiency-oriented modules and components during execution (regardless of whether driven by the user or generated as a result of other computations). It thus supports

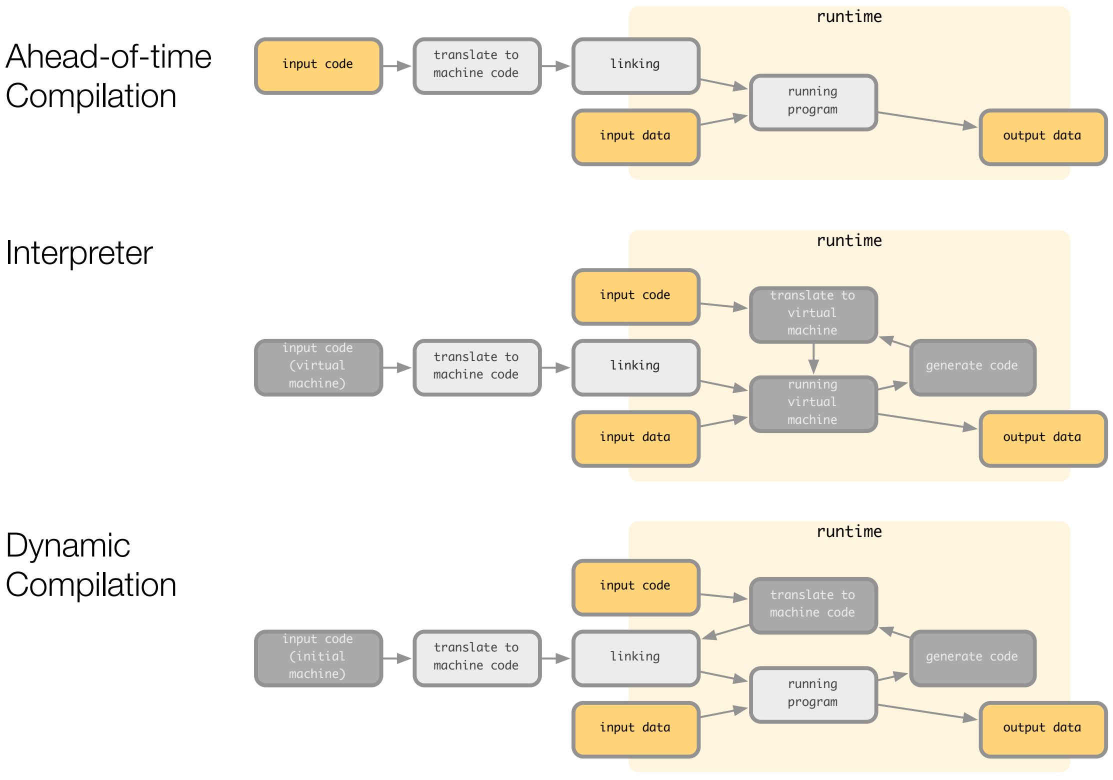


Figure 1.2.1: A comparison of the ahead-of-time / run-time workflows for static compilation, interpretation and dynamic compilation. In ahead-of-time, static compilation (top) the entire program is written and translated to machine code before execution begins, and the only level of external feedback supported in the running program is via pre-defined input data streams. Changes to the code require stopping and restarting the program. In the interpreter case (middle), code for a virtual machine is translated to machine code in advance, but user-defined code can be entered while the program runs. This input code could be written by a user or generated internally; in either case it is translated to match the functions provided by the virtual machine in order to be executed. The indirection caused by the virtual machine however reduces overall performance. With dynamic compilation (bottom), code entered while the program runs, or generated internally, is translated to machine code and directly linked into the running program, offering both flexibility and efficiency.

a 'hyper-modular' strategy, since the boundary between optimized components is itself dynamic; it presents a much smaller granularity of change, since set of static, un-modifiable 'intrinsic' elements can become very small (Piumarta and Warth, 2008) with no performance cost; and it approaches 'second-order' computing as the space of possible flows it can metamorphose into becomes vast. We can therefore consider the open-endedness of a system in terms of the space of possible machines that it can become, and the granularity at which it can change.

Nevertheless, despite a long history of research in Computer Science, this strategy has only rarely been explored within the domain of compute-intensive, interactive arts.

1.2.6 Related work

There are few direct antecedents in the literature[5], though Scott Draves's *Nitrous* project singularly stands out. In his 1997 thesis, Draves proposed an approach to treating interactive media systems as programming languages, leveraging program transformation and run-time code generation. Draves specifically intended to support "an impatient and unpredictable user who produces and consumes digital audio and video" (Draves, 1997). The central method is based on *partial evaluation*: a means to take a general program and convert it into a more specific (and faster) program by analyzing which of its inputs are fixed (static) and which are variable (dynamic), and propagating this information to both data and control flow. Specifically, *binding time analysis* identifies a sub-set of the possibility space of the original general program, and as a result a specialized program is produced. In addition, a generator program can analyze any generic program to produce a specializer for it. This specializer can then be invoked as needed to cre-

[5]Reviews of more broadly relevant research can be found in Chapters 2 and 3.

ate optimized programs. Key challenges identified included whether a system can produce 'good enough' code, whether the time spent generating code was worth its speed-up, and whether the programmer-time spent learning and using the system was worthwhile. Draves explicitly speculated on the potential of run-time code generation for for specialist tasks including designing signal processing graphs, for driving artificial evolution (Draves, 1997), and general pixel manipulation (Draves, 1998).

Conal Elliot reprised the concept of an image-manipulation program as a domain-specific language for dynamic compilation in Elliott et al. (1999); a similar insight is presented in (Tissoires and Conversy, 2008), and dynamic compilation for pixel compositing is demonstrated in (Amelang, 2008).

Video games present a particularly performance-conscious form of interactive computational art. In 1999 the *Quake III Arena* computer game embedded a dynamic code generator to speed up the game interpreters, which are primarily event dispatchers for game logic and rendering control. The generator directly translated stack-based interpreter byte-code instructions to x86 instructions at load time (Carmack, 2007). The documented performance increase was less than 50% better than interpreted byte-code, due to stability criteria and an absence of optimizing transformations: Carmack acknowledges that "the generated code is pretty grim if you look at it, in part due to the security measures (mask and add for each load/store), and in part due to the fact that it is a straight byte-code translation." (Ibid.) Despite this, and the fact that generated machine code used five times more memory and made debugging extremely difficult, run-time performance was of such critical importance to the game that these setbacks and results were worthwhile.

Dynamic compilation or machine code translation may have been utilized in many other games over the past decades, but sadly documentation is not easy to obtain. In the year of publishing this dissertation however, several game-oriented development environments have introduced dynamic code editing (without restarting the game) for the sake of improving the game development experience. The games-oriented open-source *RunTimeCompiledC++* project is primarily aimed to support run-time changes of C++ code[6], and integrates a compiler, a file watcher,[7] and a run-time object system for reflection. The commercial engine *Unreal 4* (released 2012) includes a "Hot Reload" feature allowing run-time editing and reloading of C++ code[8].

Also contemporaneous with the research undertaken for this dissertation, available literature shows a handful of other creativity support tools that have begun to leverage dynamic compilation, including those of Sorensen and Gardner (2010) and of Thieleman (2010), which will be discussed in Chapter 4.

Although several evolutionary systems have utilized code generation for bytecode virtual machines closely inspired by machine code (Rasmussen et al., 1990; Ray, 1997), to the author's knowledge, no immersive evolutionary artwork has yet utilized the generation of actual machine code to reconcile open-endedness with real-time performance.

Though less directly relevant, it should be noted that several other systems have demonstrated run-time compilation for dedicated high-performance media hardware, including audio (Scaletti and Johnson, 1988) and graphics (Stone et al., 2010).

[6]https://github.com/RuntimeCompiledCPlusPlus/RuntimeCompiledCPlusPlus (accessed 2012).

[7]A file watcher is a background process which creates an event notification whenever the modification date of a particular file changes. The notification can be used to trigger reloading or recompiling the file contents whenever it is edited and saved.

[8]http://www.unrealengine.com/unreal_engine_4 (accessed 2012).

1.3 Evaluation

The demonstrations in this dissertation are evaluated quantitatively according to the criteria of flexibility and efficiency, and qualitatively in terms of the broader implications for interactive computational arts. The measure of success of the software requires acceptable run-time performance and creatively significant advances in open-endedness: enabling smaller granularities of change, or making possible sets of solutions that were previously precluded.

Technically, adding a compiler to an existing program is necessary but not sufficient: techniques germane to ahead-of-time compilation may not translate well to run-time situations. Implementation challenges (discussed further in Chapter 3) include:

- how to specify new algorithms and introduce them at run time,
- how to manage an embedded compiler from within the program,
- how and where to link dynamically-compiled native code into the executing program's address space and event systems,
- how to remove parts of a running program,
- how to transform existing portions including transferring state and contextual connections,
- how to retain abstract specifications alongside native code,
- how to manage memory of dynamic objects, functions and code, and
- how to achieve all of the above efficiently (both minimizing latency during changes, and producing optimal machine code as a result of each change).

1.3.1 Authoring environments

To serve as valuable contributions to the domain of creativity support tools, authoring environments must:

1. Demonstrably match or improve upon efficiency measurements for existing techniques.

2. Confer greater algorithmic flexibility to the user, through the provision of lower-level primitives, a smaller granularity of change, and supporting the construction of algorithms that were previously infeasible.

3. Do so without causing interruptions in the execution of the software, and without requiring a cognitively distracting interface or the development of difficult new skills.

Two concrete examples that satisfy the above criteria are presented in Chapter 4 and evaluated in Chapter 6; both examples extend existing authoring environments that utilized the modular strategy (Max/MSP and LuaAV).

1.3.2 Open worlds

It is more difficult to provide a common measure in the domain of open-ended generative artworks. In order to make the evaluation concrete, the dissertation considers a specific artwork (*Time of Doubles*, a collaborative project between Haru Ji and the author, Wakefield and Ji, 2009), and a specific exhibiting context (the *AlloSphere*, an immersive facility at the California NanoSystems Institute, University of California Santa Barbara, Amatriain et al., 2009). *Time of Doubles* constructs nature-like experiences by presenting an ecosystem of thousands of evolving organisms, swimming in a fluid simulation, with which multiple users can interact (Ji, 2011). The AlloSphere uses multiple high-resolution stereographic projections and numerous loudspeakers to construct deeply immersive, interactive worlds within a three-story spherical space (Putnam et al., 2010). Satisfying the high-resolution requirements of the AlloSphere (in which one of the rendering machines renders 4x HD resolution at 60 frames per second) and the complex simulation and large population of the ecosystem approaches the computational limits of the hardware.

The evaluation criteria of the demonstrated additions to *Time of Doubles* documented in Chapter 5 must continue to satisfy the performance demands of the AlloSphere, while also greatly expanding the possibility space of the ecosystem. Specifically, the parametric evolution of prior versions of *Time of Doubles* (where control-flow paths are predetermined before execution begins, and only numeric parameters vary between organisms) is replaced by fully evolutionary programming (where each newborn organism has a distinct piece of machine code with distinct control-flow paths). This work is detailed in Chapter 5 and evaluated in Chapter 6.

1.4 Overview of the dissertation

The dissertation begins with a background analysis of computation and creativity in Chapter 2, focusing in on issues of creative software design, the trade-off of open-endedness and efficiency, continuing to the specifics of reflective metaprogramming and dynamic compilation in particular and resulting in a general framework of requirements and techniques in Chapter 3.

The demonstrations of empirical practice within the two scenarios of authoring environments and open worlds are documented in Chapters 4 and 5 respectively.

Each chapter describes software produced according to the model and method of the dissertation. Each chapter describes software produced according to the model and method of the dissertation. All demonstrations are evaluated in Chapter 6, along with additional results and speculations regarding future directions of work.

Chapter 2

Background

Computer artists make art by making and using programs. Writing programs is a process of expressing intentions in terms of what a machine can do: a pragmatic translation of semantics into syntax. For computational arts, creative activity must become specific executing computational flows actualized on hardware. What the artist can do, what the artwork can do, and ultimately what we can experience is bounded by the real-time performance and the possible paths of execution implied by the design of the software used. However, many researchers have observed that software practice has evolved inessential habits that may not be appropriate to creative work (Sussman, 2007; Gabriel and Sullivan, 2010; Lee, 2009; Hewitt and Jong, 1986; Kampis, 1993).

Thus we begin with a re-examination of the root concepts and properties of computation in specific terms of creative values. This chapter considers the nature of software and how it is constructed, and the specific challenges and issues for creative work. The efficiency/open-endedness trade-off emerges as an inherent problem, generally dealt with using a modular strategy. By understanding the trade-off as a static/dynamic division, it becomes clear that the modular strategy remains static in its boundaries, while a strategy employing reflective meta-programming and dynamic compilation allows the division itself to become increasingly flexible and open-ended without sacrificing efficiency. While clearly beneficial for creativity support tools, it has further implications for interactive generative arts.

2.1 Open-ended computation

The formal notion of computing was first put forward by Alan Turing in terms of universal Turing machines (Turing, 1937), Alonzo Church in terms of the lambda calculus (Church, 1941), Emil Post in terms of Post systems (Post, 1943). The core concept is that a procedure for operating on data can be described formally as an abstract machine (a program). A description may be, for example, a discrete list of decisions and instructions to specify at each step what must happen next (Turing suggested the metaphor of a recipe), or a recursively defined, discrete ordered structure of functions to produce results. The Church-Turing thesis holds that these (and other) formalizations are equivalent, and encompass the class of *universal computing machines* (the behavior of all possible determined machines). Though they may differ in implementation, the set of machines that they can actualize is always functionally the same: any machine expressible in the lambda calculus can be expressed a single-taped Turing machine, or vice versa. The first universal computer constructed was the *Z3,* by Konrad Zuse in 1941.

Essential to the innovation of computability theory is that these descriptions of abstract machines are expressed as information: *programs are data.* This relies on the conception of information theory that any semantic description can be encoded as data, so long as all the elements of interest can be drawn from a pre-defined set of primitives. Program data is usually written in terms of a language grammar and vocabulary of primitives, but computation is not necessarily linguistic: any chunk of data whatsoever can be interpreted as a program, so long as its structure matches the encoding expectations of a given computer (Ikegami, 1999).[1]

Theory is only half the story: as Shagrir notes, a computer is a physical process occurring in time, embodying the logical abstractions we design (Shagrir, 1997). Physical computing machines execute programs by actualizing them as discrete behavioral *processes*. Langton for example described computing hardware as a "second-order machine" that can "realize abstract machines" (Langton, 1996). It is important to observe however that traditional computability theory defines what classes of machines can realize each other, but says nothing about how much physical resource and time are used in the *process* of doing so; it is concerned with the final state rather than the journey taken.

A further consequence of the fact that programs are data is that they can be the subject and product of other programs: "von Neumann noted that computer programs are equivalent to data, that programs alter data, and thus programs can alter programs." (Rasmussen et al., 1992) A program whose process includes the manipulation of programs is called a *meta-program*. Thus computers not only *realize* abstract machines; they can also *create* and *transform* them.

Strictly defined Turing machines are closed, deterministic systems in which each step is fully specified by the previous, and the first step by the initial conditions. Nevertheless a *program-as-process* weaves a contingent history that is not always predictable from the *program-as-data*. For example, one of the principal results of computability theory is the inability to predict from the description of

[1]The ability to interpret any data as process and any process as data has sometimes been rather directly exploited in computer and media arts, such as Hubert Hohn's *Ceci n'est pas une pomme* and various works of glitch art (Cascone, 2000; Magnusson, 2002).

an arbitrary program whether it will run forever or halt (the 'halting problem'). Computation in principle is open-ended due to a kind of *dynamical complexity*, since "the trajectory of a randomly chosen initial condition for a universal computation system cannot be known a priori." (Rasmussen et al., 1992)

Early computing was dominated by the character of discrete tasks and proofs, in which the two worlds of program construction and program execution were entirely dissected in time: input is only specified before starting to run, and the only results of interest are given after the program halts. The *program-as-data* forms part of the 'initial conditions' of the computing machine's dynamical system. However, it would be a mistake to infer that the run-time process constitutes nothing more than an in-time realization of an outside-time organization (Lee, 2009). There is nothing restricting the program data itself being from modified while the program continues to run. It could, for example, operate upon and extend itself, as a *reflective* meta-program.

Fisher identified this kind of self-modification as an essential but infrequently used form of control-flow. Briefly, the control structure of a digital computer is a sequence of instructions, with three exceptions: "conditional and unconditional control transfer (branch) operations, and operations on the program as data." (Fisher, 1972) With conditional and unconditional branches, the gamut of control structures including GOTO, IF, WHILE, FOR, etc. are possible. The third kind allows the modification of the instruction series itself. "Although instruction modification in theory provides a major source of power for the digital computer, the technique is seldom used in modern machines except for initial loading and translation of programs... In early machines, it was a widely used technique; but today, dynamic instruction modification is generally considered bad coding practice, even in machine and assembly languages." (Ibid.) The principal reason for discouraging self-modification in conventional software engineering is the extreme difficulty in predicting long-term behavior.

2.1.1 Interaction machines

One might object that according to the Church-Turing thesis, a self-rewriting system can–in theory–do no more computation than any other universal Turing Machine. But as soon as a program is made sensitive to conditions external to it, it escapes the confines of theoretical determinism.

"The behavior of the computer at any moment is determined by the symbols which he [the computer] is observing, and his 'state of mind' at that moment" (Turing, 1937). Hewitt notes how this is "intensely individual and sequential," unable to encompass concurrency and with indeterminacy: "An operating system cannot be implemented using Turing Machines." [2]

Today of course we have programs (including operating systems, and also web servers) that run in continual interaction with the world they inhabit (including human beings), modulating, adding and removing parts of themselves as they run. Interactive computing brings in an unpredictability of a wholly different kind than deterministic, dynamical complexity: one that can only be characterized statistically, and never reduced to the mechanism of a machine.

Several authors have therefore distinguished machines that can interact with their environment as *open systems*, and controversially argued that their capabilities are *hypercomputational*, i.e. transcending the capabilities of classical Turing machines (Wegner and Goldin, 2003; Wegner, 2005; Costa and Dimuro, 2005; Goldin and Wegner, 2008). The controversy frequently boils down to differences of opinion regarding the definition of the term *algorithm*. Davis for example dis-

[2]http://knol.google.com/k/carl-hewitt/what-is-computation/pcxtp4rx7g1t/30#

putes the utility of an algorithm that cannot recognize when it has computed the result (Davis, 2006). But the 'result' of an operating system, web server, or interactive computational artwork is not a terminal value, it is the ongoing, unbounded constructive process itself.[3]

An interactive program behaves like a classical Turing machine between each interaction point, but is unpredictably non-deterministic across interaction points. Nevertheless, although human input to an interactive system cannot be predicted deterministically, humans do not act randomly. Rather, the relation is cybernetic: future user interactions are not known but they will in some part depend on how the user perceives and reacts to the ongoing output of the program. To the computer our actions may be inexplicable, but there will nevertheless be statistical structures to be found. The tighter and richer the modes of interaction, the more likely a system may find a potential point of symbiotic resonance with the human.

Human-computer interaction taken as a whole is thus a non-deterministic but non-random process. Many have argued that it is the combination of interaction and computation, not just computation alone, which has dramatically impacted our society: "This form of cooperation between humans and computers, and among humans via computers, is a vital necessity in many contemporary applications, where realistic results can be achieved only if human intuition and common-sense is combined with formal reasoning and computation." (Arbab, 2006)

[3]Turing also posited a purely theoretical, transcendent machine called an Oracle, or O-machine, which extends the deterministic universal Turing machine with an additional infinite tape of read-only data corresponding to the results of an otherwise undecidable question (such whether a machine halts given its description, Turing, 1939). The intention of the O-machine was to capture formally defined but indecidable mathematical relations, but O-machines can also be used to theorize interactive computing, since the state of an interactive input cannot be computed (reduced to a deterministic procedure) but only represented as an observation: a question with a binary result, such as "is the joystick button currently pressed?" An interactive program reading sensor inputs performs a measurement, evaluates a decision, which translates the external world's incomputable differences of kind into actualized differences of degree.

2.1.2 Static and dynamic computation

Many common tasks don't need solutions that are so vastly open-ended. Most problems are specific, not general. Any computational decision whose outcome is invariant during execution can be made ahead of time.

The predominant software development workflow today uses a *compiler* meta-program to perform this ahead of time, *static* decision-making. Making decisions in advance makes program behavior easier to predict, which is beneficial for issues of security, stability, and so on. A compiler can also restructure the implementation of a program according to static invariants. For example, operations on constant inputs can be performed compile-time, recursively reducing complex expressions to pre-calculated values, and complex algorithms to reduced sub-sets ('constant folding'). Entire branches of a program can be removed if it is known that they will be unused ('dead code elimination'). Common code paths can be concretized as compound routines with branching removed (at the expense of using more memory), and repeated tasks can be structured to operate over larger batches of data ('buffer processing'). Finally, the compiler can specialize the program for the specific target context, such as the CPU architecture. As a result of these operations, compiling programs ahead of time can result in orders of magnitude better performance at run time. Put simply: unpredictable indeterminism is the enemy of efficient software.

This kind of high performance is a central concern for interactive computational arts. The real-time nature of interaction implies immediate pressures on software design: how much can be achieved in a given slot of time? Real-time pressures are tightly bound to the temporal/perceptual limits of humans: video frames must refresh at a sufficient rate to maintain the illusion of continuity (the persistence of vision effect); audio samples must be produced at a steady frequency more than double our perceptual threshold (Dannenberg, 2005); and the round-trip of user input, processing, and display of results needs to occur on scales of milliseconds to be perceived as immediate, comprehensible and natural: "Interactive systems have to be efficient. In particular, graphical rendering must be fast enough so as to avoid hindering the user perception/action loop" (Tissoires and Conversy, 2008).

In summary, computation is an inherently dynamic process and also a remarkably flexible and open-ended medium: it can become any of a vast space of possible machines, given only their descriptions; it can modify and generate these descriptions itself, even while running them; it can behave unpredictably, even if it is deterministic; and it can also be open to indeterminate interactions with its external world. But the sheer necessity of efficiency encourages the use of ahead-of-time computation and the habit of considering programs-as-data as essentially static. By doing so, the interactive accessibility of the open-ended space of computation is, to a greater or lesser degree, sacrificed.

2.2 Context 1: Creativity support tools

As a *meta-medium*[4], computation inherits from information theory the capability of mapping interactive connections between media, people, and systems of thought that were previously distinct, and infuses within it autonomous logic, process and behavior. Cybernetic pioneers saw this as a radical opportunity for man-machine discovery and the augmentation of human intellect (Ashby, 1956; Licklider et al., 1960): "What we have done in the development of our augmentation means is to

[4]As described by Lev Manovich in Abstraction and Complexity: *http://www.manovich.net/DOCS/abstraction_ complexity.doc* (accessed 2011)

construct a superstructure that is a synthetic extension of the natural structure upon which it is built.“ (Engelbart, 1962) By the 1970's the notion of the computer as augmenting creative work was established (Dietrich, 1986): "The chisel, brush and canvas are passive media whereas the computer is an active participant in the creative process." (Youngblood, 1970)

Through the subsequent decades office and home computing industries made computer-assisted creative work commonplace, but at the same time more streamlined toward the anticipated productivity needs of industry and consumer users. The commodification of computing led to emphasis on interactive *programs*, rather than interactive *programming*. Established techniques of many creative practices have been emulated in widely used tools of productivity support, including industrial products such as Adobe Photoshop (static image editing), Apple Final Cut Pro (video editing), Digidesign Pro Tools (multi-track audio editing), and so on. However, "to state that computers can offer an unimaginably greater world of possible forms than these products is not techno-optimism; as computers are provably capable of simulating any other machine, it is mathematical fact."[5]

There have always been researchers keen to go beyond supportive roles of computing to explore its more creatively open-ended potential: "A growing community of innovative tool designers and user interface visionaries is addressing a greater challenge and moving from the comparatively safe territory of *productivity support tools* to the more risky frontier of *creativity support tools*." (Shneiderman, 2009) Fischer and Scharff describe this as a shift from a consumer mindset to a designer mindset, calling for principles of 'meta-design' or '*design for designers*' (Fischer and Scharff, 2000).

[5]Golan Levin, Pedagogical Statement, 2003: www.flong.com/texts/essays/statement_pedagogy

2.2.1 Creative exploration

Gabriel and Sullivan (2010) emphasize how creative work, in both art and science, is essentially underspecified; more characterized by problem-generating than problem-solving, by heuristic than algorithm.[6] Thus if "one of the main aims of man-computer symbiosis is to bring the computing machine effectively into the formulative parts of problems" (Licklider et al., 1960) and imaginative work presupposes a capacity to entertain, reason with and explore hypothetical scenarios (Harris, 2000), then the computing interface must support incompletely specified tasks as well as unpredictable solutions: "Poincaré anticipated the frustration of an important group of would-be computer users when he said, 'The question is not, "What is the answer?" The question is, "What is the question?"''' (Licklider et al., 1960)

Among twelve principles for creativity support tools, Schneiderman proposes supporting open-ended interfaces ('high ceilings and wide walls'), with many paths and styles, and open interchange (Shneiderman, 2007, 2009). Similarly, the *Cognitive Dimensions* framework of analysis has shown that creative interfaces should have a low 'viscosity' (resistance to change), and reduce 'premature commitments' which would narrow paths of further development (Green and Petre, 1996). In short, open-ended exploration presupposes a space larger than what can be immediately seen or predicted, which the generality of computing can provide, allowing us to explore beyond the barriers of predictability.

Exceeding predictability introduces the value of the *accidental*: "the discovery of something we didn't anticipate, couldn't even imagine before we found it." (Turchi, 2004) The value of the accidental that it forms "the boundary between

[6]Stafford Beer characterized heuristics as "a method of behaving tending towards a goal which cannot be precisely specified because we know what it is but not where it is..." (Beer, 1994)

the known and the unknown, the expected and the unexpected; the 'accidental' happens where waves break on the beach of knowledge" (Hayles, 1998). In fact accidental discovery can be so highly valued that "some artists use artificial mechanisms to ensure that the starting place and subject matter are substantially random and unplanned for in order to explore fruitfully." (Gabriel and Sullivan, 2010) For example, some artists use tools in deliberately unconventional ways, for which they were not intended (Cascone, 2000). We can compare this with Green and Petre's observation that while a software interface filled with 'hidden dependencies' may be counter-productive to fluid editing, it may be productive if errors fail gracefully and dependencies promote accidental discovery (Green and Petre, 1996).

Another long-celebrated technique of de-familiarization and promotion of accidental discovery is the introduction of formal rules as poetic constraints into the creative process, from the template-driven methods of serialist and minimalist composers, to the experimental literature of the Oulipo (constraints as "a commodious way to move from language to writing," Benabou, 1983). But while arbitrary constraints can be expedients to reveal new ideas, most artists do not want to be prevented from doing something because it wasn't 'programmed in.' Creative work is about breaking existing rules as much as making new ones: "Art is the roadbuilding habit (Zeno). It ruptures, then rebuilds, the edge of thought." (Novak, 1998) In particular, revealing and removing the implicit constraints hidden within a system can lead to more revolutionary discoveries:

> "Exploring a conceptual space is one thing; transforming it is another. What is it to transform such a space? One example [is] Schoenberg's dropping the home-key constraint to create the space of atonal

> music. Dropping a constraint is a general heuristic, or method, for transforming conceptual spaces." (Boden, 2007)

To clarify, Boden proposed three classifications of creativity. *Combinatorial creativity* is the creation of new combinations of existing elements in a conceptual space. *Exploratory creativity* involves the following of pre-defined search strategies, heuristics, procedures and evaluation measures to arrive at novel results within this conceptual space. *Transformational creativity* depends on transforming the space itself by rewriting the rules (search strategies): a new transformation enables new types of exploration. Wiggins refined and formalized this model, revealing that changing the rules of evaluation may also lead to new discovery, and demonstrating that transformational creativity is exploratory creativity raised to a meta-level (Wiggins, 2006).

Put another way, the formalization of a space should not be overdetermined or immutable. To properly ground exploration, structures should be built incrementally through 'progressive evaluation', since "only after discovery can the work be properly structured... If we attempt to map the world of a story before we explore it, we are likely either to (a) prematurely limit our exploration, so as to reduce the amount of material we need to consider, or (b) explore at length but ... arbitrarily omit entire realms of information." (Turchi, 2004)

A cycle of generation, evaluation, and integration thus lies at the heart of the creative process. The goals of experimental artists "are imprecise, so their procedure is tentative and incremental" (Boswell, 2008). Even the more conceptually-driven artists, who state goals and methods precisely before production, engage in a process of ramification and discovery as a work is realized: "This is the full process of art—exploration, discovery, and then verification or understanding—and it's also the full process of science. Naturally, it's not a linear process but one with cycles of explore-discover-understand within any of its steps." (Gabriel and Sullivan, 2010)

2.2.2 Programming on-the-fly

> "All studies of programming show that changes and revisions infest the whole course of programming activity, from specifying to designing to coding. This is not because programmers and designers are stupid and careless, but because of how human reasoning works: we need to see a sketch and have it 'talk back to us'." (Green and Petre, 1996)

Interactivity has long been valued by computer artists: "The computer makes execution fast enough to be done interactively with further human decisions, accurate enough to avoid mistakes, and cheap enough to afford a great deal of experimentation.“ (Knowlton, 1976) Furthermore, "the exchange, while the process of creation is happening, seems more alive because of its instantaneity. It is almost like having a conversation with someone from which something visual results." (Vilder, 1976)

Interactive exploration and conversational dialogue has also been a clearly-stated goal for several programming languages including LOGO (Papert, 1971)[7], Scratch (Resnick et al., 2009), Smalltalk (Ingalls, 1981), Self (Ungar and Smith, 1991), and most flavors of Lisp: "The average Lisp user writes a program as a programming experiment." (Sandewall, 1978) Interactive programming supports adding a code segment Q (in some programming language) to an existing, already running program P; where Q runs in the same address space of P, can use the

[7]LOGO's influence extends through John Maeda's *Design by Numbers (Maeda, 2001)* to the now widely used *Processing* environment (Reas and Fry, 2006).

shared data and components of P, and can modify the structure/logic of P in some manner or other. In effect, the addition of Q to the running program P evolves it into a new program P' (Wang and Cook, 2004).

Interactive programming naturally lends itself to the creative cycle of exploration, discovery, and understanding: "Many problems ... would be easier to solve, and they could be solved faster, through an intuitively guided trial-and-error procedure in which the computer cooperated, turning up flaws in the reasoning or revealing unexpected turns in the solution." (Licklider et al., 1960) Allowing a program to be rewritten at run-time is a shift away from the notion of the machine as a tool toward its conceptualization as an always-active workspace in which "the programmer lives and acts in a consistent and malleable world." (Smith and Ungar, 1995) In this 'conversational' mode of interaction a program gradually evolves while preserving intermediate state, supporting experimentation by allowing commitments to be made as late as possible. Potter called it *just in time programming*: "The goal of just in time programming is to allow users to profit from their task-time algorithmic insights by programming... the user recognizes an algorithm and then creates the software to take advantage of it just before it is needed, hence implementing it *just in time*." (Potter, 1993)

The user thus operates as a "spontaneous algorithm creator" (Sorensen and Gardner, 2010), however to a good extent what is discovered and created depends upon the properties of the interface. Traditionally interactive programming was presented though a textual console providing a 'read-eval-print loop' (REPL), but today it can be driven by a broad diversity of graphical and textual interfaces, including visual data-flow diagrams and spreadsheets. Potter's user-centered analysis identified that just in time programming is obstructed by static, closed interfaces and limitations on computational generality, "because there is no way to predict which of the vast array of possible algorithms users might envision being useful." (Potter, 1993) Furthermore, reducing the temporal delays between programming actions and perceived results may be essential to creative discovery: "An exploratory programming environment increases programmer productivity by giving immediate feedback to all programming actions; the pause-free interaction allows the programmer to concentrate on the task at hand rather than being distracted by long compilation pauses." (Holzle, 1994)

2.2.3 Live coding

The real-time performance aspects of exploratory programming are perhaps most intensified in the performance practice of live coding (Collins et al., 2003). In a live coding performance, artists continuously write, modify and re-write a program which continuously produces sounds and/or images. The program is typically written from scratch at the start of the performance in a largely improvised manner, and the editing often made visible to the audience.

> "A live performance that includes authoring code diverges from any perceived determination... This includes the potential for mistakes to positively affect the course of the performance, that no longer expresses control but is open to the vagaries of feedback." (Cox et al., 2004)

The programming interface becomes a performance instrument, with many of the corresponding human-computer interface issues (Magnusson, 2005) including requirements of responsiveness and robustness.[8] However, while traditional performance instruments are dominated by parametric and modal flexibility, a

[8]The first reported on-stage live coding performance (by Ron Kuivila using Forth in 1985) resulted in a system crash, a climax which has been repeated by many live-coders since (Blackwell and Collins, 2005).

programming-language-as-instrument uniquely emphasizes the construction of autonomous processes whose spaces of possible behavior are defined and re-defined on-the-fly.

Nevertheless, interface biases become apparent very rapidly in live coding performance, when there simply isn't time to do undertake an exploration that is complex to articulate. Some live-coding systems utilize visual programming interfaces, which generally adhere to a formal semantic convention such as events flowing through a graph (such as *Max* (Puckette, 1996; Zicarelli, 2002) and *Nodal* (McCormack et al., 2007)), agents in a space (such as some *IXI* instruments (Magnusson, 2006) and *Al-Jazari* (Mclean et al., 2010)), and cellular grid processes (*Betablocker*), while others embellish textual code with visual cues (*Scheme-Bricks*) (Mclean et al., 2010). Visual conventions can make systems more robust, easier to get started and more aesthetically stimulating than text, but analysis confirms the loss of abstraction and increased viscosity this implies, often falling back onto textual features to overcome these restrictions (Green, 2000).

Wang and Cook draw attention to the need for an explicit and accurate *representation* of time in the programming interface (Wang and Cook, 2004). Sorensen and Gardner expand upon this infusion of tight temporal semantics within programming environments, suggesting the term "with-time programming" to focus on the cyber-physical nature of the computer-human ensemble (Sorensen and Gardner, 2010).

Live coding juxtaposes human spontaneity and algorithmic art: "Programs are instruments that can change themselves."[9] While some authors have downplayed this generative character, others openly celebrate it (Brown and Sorensen, 2009). For example, Cox and McLean describe a live-coded program which has the ability

[9]TopLap Manifesto (Draft), http://toplap.org/index.php/ManifestoDraft, accessed 2012.

Listing 2.1: Three iterations of a trivially self-modifying program (in Perl), by Alex McLean (http://yaxu.org/words/yaxu/feedback.html, accessed 2009).

```
sub bang {
    my $self = shift;
    # Feedback
    $self->code->[3]=~ s/e/ee/;
    $self->modified;
}

sub bang {
    my $self = shift;
    # Feeeeeeedback
    $self->code->[3]=~ s/e/ee/;
    $self->modified;
}

sub bang {
    my $self = shift;
    # Feeeeeeeeeeeeeeeeeeeeeeeedback
    $self->code->[3] =~ s/e/ee/;
    $self->modified;
}
```

to modify its own code (see Listing 2.1):

> "These modifications happen directly to the code being edited in real-time, opening up the possibility for the code to fundamentally modify its own behavior. Of course, this has major implications upon the act of programming. The programmer must now think not only of what the software will do and how it will interact, but also how it will modify itself and remain functional, and continue to be active." (Cox et al., 2004)

2.3 Modular strategies

We have seen that creative computation is usually associated with complex, indeterminate and exploratory activities resistant to a priori characterization, full of provisional manners and unbounded methods; for which the open-ended character of computation is well-suited. However we have also seen that the most efficient code is usually the least flexible, having performed much of the decision-making required in a compilation stage before the software is run (the combination of prior specialization to tasks and for specific hardware explains the prevalent use of compiler tools for languages such as C/C++ and assembly). To strike a balance between dynamic flexibility and desirable efficiency, design compromises must be made.

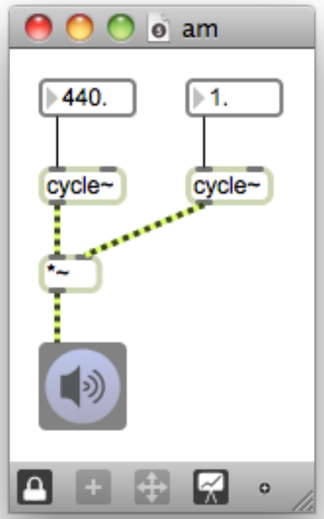
The compromise taken by most relevant systems today utilizes modularity: expressing components and rules in optimized machine code compiled ahead of time, while letting decisions regarding their combinations be deferred until run-time: "The need for modularity is even more important with interactive systems. Making software modular minimizes the cost of modification. As designing good interactive systems requires designers to implement, test, and tweak a large set of alternative solutions iteratively, modular software maximizes the quality" (Tissoires and Conversy, 2008).

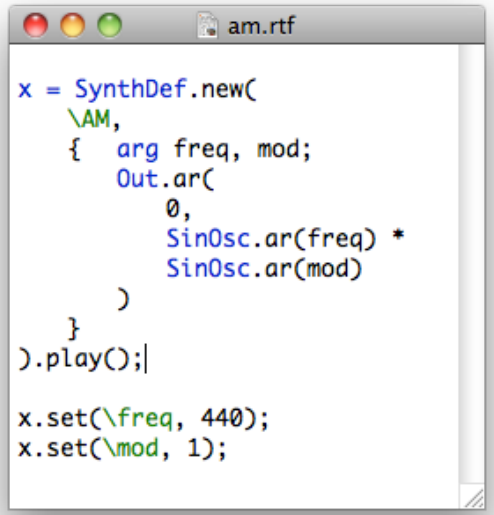
This compromise often follows a kind of Pareto principle[10]: since usually around 80% of run-time activity is performed by 20% of the code-base, it makes sense to aggressively optimize that 20%, and leave the remaining 80% amenable to run-time malleability. The dynamic portion may be a run-time data-structure presented via a graphical interface, or may be a dynamically interpreted language; the static portion is a collection of optimized, pre-compiled functions (which are represented as 'black box' primitives in the dynamic portion) and a run-time engine or virtual machine.

The modular strategy works well when it can be known clearly in advance which portions will be the efficiency bottlenecks. There are limitations however - it may not be possible when designing software to fully anticipate which portions will be heavily used at run-time. The strategy breaks down as soon as a change is desired that is not aligned to the modular boundary, whether by modifying the contents of a black box unit or library function, or in creating new ones. The boundary of flexibility is itself inflexible, and the set of black boxes remains fixed. The full power of computation as a meta-machine does not reach the granularity of the media being processed. Furthermore, the loss of efficiency at the dynamic boundary may be significant when many components are in use. Choosing a coarser granularity (few large components rather than many small ones) reduces this, but also reduces flexibility.

It is important to distinguish a modular style of interface from a modular

[10]Also known as the 80/20 Rule: originally stated that roughly 80% of the wealth is owned by roughly 20% of the people, but widely generalized since then.





```
x = SynthDef.new(
    \AM,
    {   arg freq, mod;
        Out.ar(
            0,
            SinOsc.ar(freq) *
            SinOsc.ar(mod)
        )
    }
).play();

x.set(\freq, 440);
x.set(\mod, 1);
```

Figure 2.3.1: Modular graphs of unit generators in Max (left) and SuperCollider (right). Two black box operations to generate sine wave oscillations (cycle~ in Max, SinOsc in SuperCollider) are multiplied together (*~ in Max, * in SuperCollider). In Max the operator connections are explicitly visualized as patch cord lines, in SuperCollider they are stated as function or operator arguments.

implementation as described above. We have seen that computation is built upon the separation of descriptions and implementations; thus there is no reason why a modular interface must be implemented along the same lines of division. It is also important to consider how modularity also provides cognitive benefits. It allows us to express a domain-specific system in terms of the core concepts of the domain, to decompose problems into manageable sub-problems, and re-use general solutions in multiple specific instances. Nevertheless, these benefits depend on the modules following the conceptual structure of the domain, not the structure of the implementation.

2.3.1 Example: audio synthesis

To clarify further, let us consider the audio synthesis case in more detail. Interactive audio synthesis is extremely sensitive to real-time performance due to the high sample rates (tens of thousands per second) and low latencies (tens of milliseconds) desired. Failure to meet performance deadlines results in unpleasant sound. Systems therefore generally operate on large batches or 'buffers' (from tens to thousands of samples per buffer).

To introduce flexibility in the specification of a synthesis algorithm, the majority of software in use today takes a modular approach which can be traced back to the work of Max Mathews in the 1950s, connecting black box *unit generators (ugens)* into aggregate processes by connecting their input and output ports together into a directed graph (Mathews, 1961). Mathews' pioneering innovation remains apparent in the Max (Zicarelli, 2002) or PD (Puckette, 1997) *patcher*, the SuperCollider (McCartney, 2002) *SynthDef*, the CSound/SAOL (Scheirer and Vercoe, 1999; Lazzarini, 2005) *instrument,* and so on.[11]

Each ugen performs a specific signal-processing task using buffers of samples, potentially storing state between each sample processed. These intermediate buffers of samples belong to the connections between ugens rather than the ugens themselves, providing a degree of indirection which allows the structure of ugen data flow to by modifiable at run-time (whether by graphical 'patching' or textual coding, see Figure 2.3.1).[12]

However, this approach presents several problems:

1. *No new primitives.* The set of algorithms that can be expressed is limited to

[11]The ugen paradigm of Mathews' *Music-N* is described in detail in section 4.1.

[12]There is a strong analogy between a modular synthesis engine and an interpreted language: each ugen corresponds to a primitive instruction type for the interpreter's virtual machine, but operates over buffers rather than single values.

the set of unit generator modules in the system, and the set of parameters they expose. New additions to the set of primitives may be possible through a separate software development kit (SDK) and application-programming interface (API), typically involving complex low-level and time-consuming programming that is far removed from creative goals.

2. *Structurally closed.* Operations at a lower level than the black box ugens can be modified parametrically, but not structurally. One consequence of this is that constant or disconnected input ports and optional modes still pay the full cost of being dynamic despite being unchanging or unused.

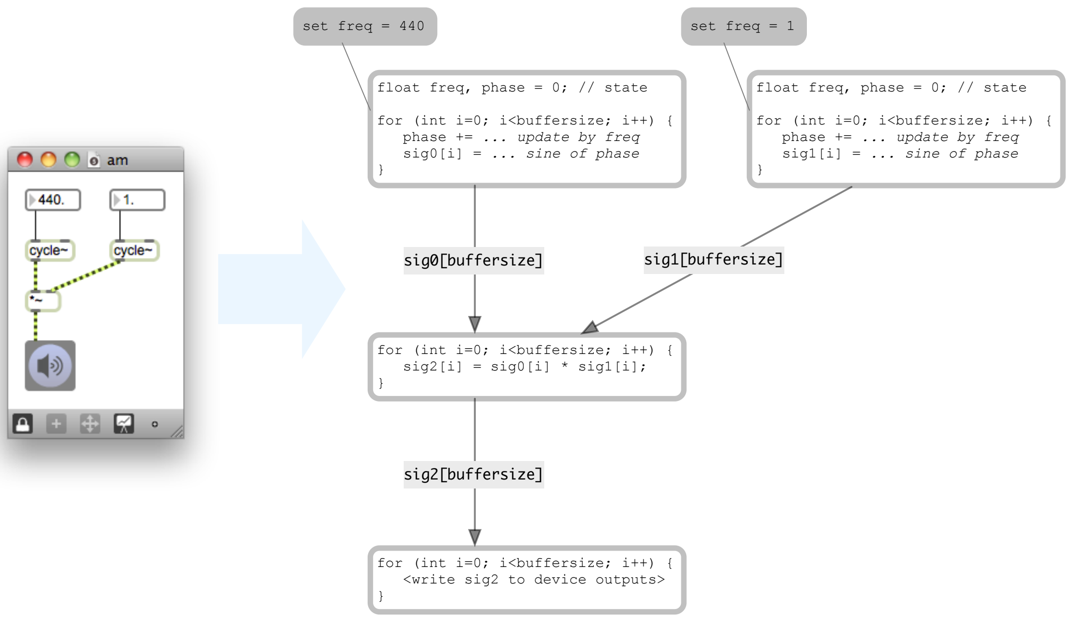
3. *Static granularity.* The choice of granularity between what can and cannot be modified is not granted to the user. This granularity of the components also limits efficiency: the wider the set of possible combinations can be made, the less optimally these combinatory wholes are expressed in machine code, leading to less resolution, less complex algorithms, and other constraints on the aesthetic results.[13]

4. *Memory overhead.* Since intermediate buffers need to be stored between each ugen process (see Figure 2.3.2), it requires reading and writing to heap memory (which is slower to access than results held on the stack would be).[14]

5. *Inappropriate concepts.* The buffering of audio is an implementation detail unrelated to the domain concepts, however it introduces limitations on what can be expressed through modules and in some interfaces even becomes an explicit part of the interface language. For example, to support feedback

[13]The temporal granularity of graph changes is often also limited by the buffer size, though this can be partially circumvented by varying buffer boundaries (Wakefield, 2007).

[14]Register allocation algorithms can be used to reduce the total heap memory used, as in the implementation of SuperCollider's *SCSynth* (Bencina, 2011).



```
float freq, phase = 0; // state

for (int i=0; i<buffersize; i++) {
   phase += ... update by freq
   sig0[i] = ... sine of phase
}
```

```
float freq, phase = 0; // state

for (int i=0; i<buffersize; i++) {
   phase += ... update by freq
   sig1[i] = ... sine of phase
}
```

```
for (int i=0; i<buffersize; i++) {
   sig2[i] = sig0[i] * sig1[i];
}
```

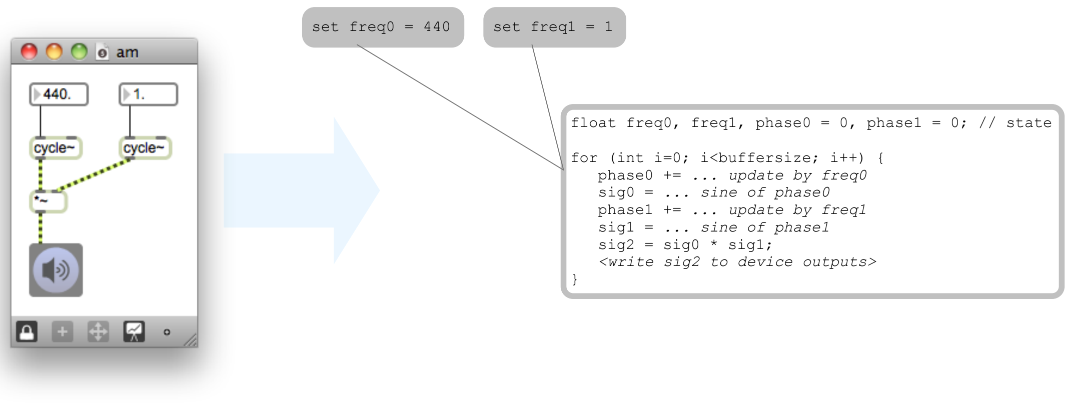
```
for (int i=0; i<buffersize; i++) {
   <write sig2 to device outputs>
}
```

Figure 2.3.2: Sketch of the underlying code of the unit generator graph in Figure 2.3.1 using the modular strategy.

loops between ugens, intermediate buffers must be also stored between successive passes over the whole graph. Feedback loops with periods shorter than the buffer size are simply not possible between ugens in this modular design. This prevents the user creating whole classes of essential signal processing systems, such as filters and physical models.

2.3.2 A hyper-modular strategy

Given the description of an algorithm, such as a graph of ugens, why not convert the whole of it into machine code at run-time? Why can't the efficient native-compiled portions of a program be variable at run-time? The result would be a single, optimal object encapsulating exactly the operations and state required. It is clear by comparison of Figures 2.3.2 and 2.3.3 how this would avoid the wasted memory and slower accesses of buffering, as well as reduce the number of loops



```
set freq0 = 440
set freq1 = 1

float freq0, freq1, phase0 = 0, phase1 = 0; // state

for (int i=0; i<buffersize; i++) {
   phase0 += ... update by freq0
   sig0 = ... sine of phase0
   phase1 += ... update by freq1
   sig1 = ... sine of phase1
   sig2 = sig0 * sig1;
   <write sig2 to device outputs>
}
```

Figure 2.3.3: Sketch of the underlying code of the unit generator graph in Figure 2.3.1 using the dynamic compilation strategy.

performed and make intermediate results local to where they are used. It would allow global optimizations between operations, such as removing or replacing unused paths with constants.

Achieving this with flexibility requires separating the modular *interface* from its *implementation*, and bringing aspects of the modular design itself into the runtime system, through the capability of *dynamic compilation*: extending a program with optimized machine code generated as it runs.[15] This strategy would avoid an exponential explosion of code size by generating special cases *lazily,* on an as-needed basis. The resultant program would evolve in response its changing and unpredictable environment.

Furthermore, the system need not abandon the cognitive benefits of the modular interface. Instead the modules can become more flexible: the granularity of primitives can vary between high- and low-levels without any impact on performance. Accordingly, the set of expressible algorithms can become much broader; in the case of real-time audio synthesis for example, it can support arbitrary systems with feedback below the global buffer size.

[15]The application of this strategy to ugen graphs is examined in detail in Chapter 4.

	Optimization granularity	**Run-time flexibility**
Interpreter	micro only	fully open-ended
Dynamic compilation	meso only	partially open-ended
Static compilation	whole program	pre-defined

Table 2.3.1: Comparing interpretation, dynamic and static compilation. Note that these systems are not exclusive: an interpreter is itself typically implemented as a program in a statically compiled language, while a statically compiled program may embed an interpreter (the traditional modular strategy). Similarly, a dynamically compiled system will include some statically compiled components, and may embed interpreted components.

2.3.3 A brief overview of dynamic compilation

Dynamic compilation can be better understood by reference to the two predominant paradigms for the translation of a program description into machine execution. In static compilation, a program is translated to machine code as a whole by a separate *compiler* meta-program before execution begins, providing many opportunities for optimization. In interpretation, the smallest fragments of a program (statements and expressions) are handled by a separate *interpreter* meta-program one at a time during execution, looking up the corresponding machine code instructions to implement them. With interpretation, there is little opportunity for optimization, but the program can be flexibly modified and rewritten at any point of its execution.

Dynamic compilation can be seen as reconciling these two approaches (see Table 2.3.1). It translates large fragments of a program to executable machine code after the program itself has already started running, without breaking execution. Unlike static compilation the whole program is not required in advance. Instead the initial program must itself supply and control the capabilities of a compiler, and properly link the machine code it generates back into the running process.

Dynamic compilation is referred to under a confusing mixture of terminology. It is sometimes called *lazy* or *just-in-time compilation* (JIT), to convey that translation to machine code is performed as late as possible, however the use of the term JIT is inconsistent. While some authors restrict this use to situations aimed specifically to optimize code with information only known at run-time (Arnold et al., 2005), others use it to refer to load-time rather than run-time compilation, such as the slim binaries of (Franz and Kistler, 1997). Kamin (2003) classified JIT compilers as the subset of code generators which operates at a binary-level, in contrast to source-level generators (such as lexer and parser generators, and meta-programming in general), but this observation does not take into account the role of linking that a run-time JIT compiler must also perform. In other contexts, JIT is used to refer to lazy linking of previously translated and cached code.

A related term, *self-modifying code,* usually refers to a specific sub-set of dynamic compilation in which modifications only occur in-place (swapping bits in the program code). Self-modifying code has been used to hide copy protection, and also both to conceal and to counteract malicious code (Anckaert and Madou, 2007).[16] However the term is sometimes more generally applied to adaptive software (such as self-repairing systems in Tschudin and Yamamoto, 2006).

Dynamic compilation has also been referred to as *deferred compilation*, *incremental compilation* (although the latter sometimes instead indicates a static compiler only recompiling the files that have changed, rather than all files of a project), *run-time code generation* (though this does not strictly imply the generation of optimized machine code), and *interactive compilation*[17].

Although not a mainstream technique of software development, dynamic com-

[16]Viruses may contain metamorphic code, which introduce changes to the viral program (possibly including the metamorphic engine itself) before being copied into a new host file. This is done to avoid pattern-recognition by anti-virus software, but also to infect different operating systems and machine architectures. The connections between self-modifying code in computer viruses, and self-modifying code in the interests of Artificial Life, has also been very clearly made (Spafford, 1989).

[17]http://www.cs.bham.ac.uk/research/projects/poplog/primer/node14.html (accessed 2011)

pilation can trace a very long history (see Table 2.3.2, and also Aycock, 2003). It may come as no surprise that the earliest example of run-time compilation is thought to be in LISP (McCarthy, 1960), a language in which both code and data share the same notation. Soon after, dynamic assembly and loading was utilized on IBM hardware (Thompson, 1968), and around the same time the use of run-time compilation for exploratory, interactive computing was being explored in the LC^2 environment (Language for Conversational Computing, (Mitchell, 1970)). LC^2 is considered a precursor of Self and Smalltalk, in which compiled code is produced as a *side effect* of the interpreter. Deutsch and Schiffman's implementation of Smalltalk (Deutsch and Schiffman, 1984) compiled lazily to memory-managed native code in a process akin to macro expansion. Inspired by this, Hölze and Ungar produced a landmark implementation of Self supporting lazy compilation to efficiently deal with the highly dynamic features of the language. The third generation of Self amortized the delay of compilation by furnishing an unoptimized function first, and later replacing it with an aggressively optimized version produced in a background process (Hölzle and Ungar, 1994).

One of the challenges of effectively using dynamic compilation is the choice of boundaries of compiled fragments. Hölze noted that the choice of when and where to compile and inline was far more critical than the method of compilation used (Holzle, 1994). Hansen suggested creating adaptive systems by *implicitly* compiling their most actively used 'hotspots' (Hansen, 1974); this approach later became a focus of industrial research for the Java Virtual Machine. The implicit, run-time informed approach can operate on individual functions ('method at a time') or ignore the original code structure and simply track the actual paths taken at run-time ('trace compilation') (Gal and Franz, 2006), a technique now heavily used for languages such as JavaScript and LuaJIT. Implicit compilation is convenient, but the unpredictably can be undesirable in some domains. At the other extreme, the HiPE compiler for Erlang is makes invocation of the compiler *explicit* in the user language, allowing fine control over the time/space trade-offs in the system (Armstrong, 2007).

Though it is the focus of this dissertation, exploratory interactive programming has not been the only motivation for dynamic compilation. Franz proposed storing libraries as high level code and compiling to machine code *at load time* for more portable and adaptive capabilities (Franz and Kistler, 1997), and Kistler extended this idea to continuous background optimization and eventually also trace compilation. Code synthesis in the kernel of operating system fundamentals was thoroughly explored by Massalin (1992), and today the idea of link-time optimization is being revisited in the LLVM project.[18]

In summary, systems using dynamic compilation can be characterized according to the following distinctions:

- Self-modifying (in-place modification) and/or self-extending
- Motivated by optimization (preserving semantics) or motivated by open-endedness (exploratory programming).
- Load-time and/or run-time invocation
- Mono- or poly-executable (whether all code is compiled, or there is a mixture of compilation and interpretation; see Aycock, 2003); the former must compile everything, latter can choose which parts to compile and interpret the rest)
- Implicit (background process) and/or explicit (user-invoked) compilation.

[18]For example http://llvm.org/docs/LinkTimeOptimization.html (accessed 2012).

1960	LISP (in-memory code)
1966	IBM 7090 (runtime assembler/loader)
1968	IBM 7094 (JIT of regular expressions)
	LC2 (interactive computing)
1970	APL (lazy evaluation)
1973	APL (mixed compiled/interpreted code)
1974	FORTRAN
1976	BASIC (throw-away compilation, hot spots)
1977	APL/3000 (type-directed run-time compilation)
1984	Smalltalk (lazy JIT)
	ML (interactive compilation)
1989	Self (method specialization)
1991	Self (deferred compilation of uncommon cases)
1992	Synthesis Operating System (code-generated kernel)
1994	Self (type feedback)
	Oberon (slim binaries)
	Prolog (dynamic compilation)
1996	Java (hotspots)
1997	Scheme (continuous optimization based on run-time flow)
1994-1999	ML (staged static + dynamic compilation)
1997	Java (translation not enough, optimization necessary too)
	‘C (Tick C: Scheme-like quasiquotes in C)
	Nitrous (run-time compiler generation for media)
1999	O'Caml (runtime macro opcodes)
2000	O'Caml (program specialization)
	Erlang
2003	LLVM (next-generation architecture)
2004	GLSL (GPU)
2005	LuaJIT
2007	JavaScript TraceMonkey
	CUDA (GPGPU)
2008	OpenCL (GPGPU)
2009	NativeClient

Table 2.3.2: Milestones in the history of dynamic compilation.

Implicit compilation can be specified by pre-determined schema or run-time metrics.

- Method-at-a-time, based on traces, or some other schema of program fragmentation.

- Serial or concurrent: i.e. whether main program execution is paused during compilation.

Supporting dynamic compilation may increase the size of a program (to support the compilation, linking and reflection capabilities), and adds at least two extra delays to the run-time: the additional start-up cost of the compiler, and the performance cost of compilation when invoked.

Today the cost embedding a compiler in a desktop application is no longer prohibitive, and efficient dynamic compilation is readily available though projects such as LLVM[19], OpenCL[20], GNU lightning[21], nanojit[22], TCC[23] and libjit[24]. Furthermore, an increasing number of projects generate high-level target code for JIT-capable virtual machines, such as CoffeeScript and ClojureScript (McGranaghan, 2011). In addition dynamic compilation is being explored in different deployment environments, from graphical hardware GPUs (GPGPU of CUDA and OpenCL (Stone et al., 2010)), web browsers (JavaScript (Gal et al., 2009) and NaCl (Yee et al., 2010; Ansel et al., 2011)) and embedded devices. The Viewpoints Research Institute, headed by Alan Kay, relies on dynamic compilation for a more revolutionary approach to computing that cuts across the whole stack of programming,

[19]http://www.llvm.org/
[20]http://www.khronos.org/opencl/
[21]http://www.gnu.org/software/lightning/
[22]https://developer.mozilla.org/en/Nanojit
[23]http://bellard.org/tcc/
[24]http://www.gnu.org/software/dotgnu/libjit-doc/libjit.html

which is sometimes characterized as ‘late-bound everything’ (Kay et al., 2008).

However, because of the danger of viruses, and concerns of stability, security, and intellectual property (due the difficulty in predicting or controlling what generated code can and cannot do), dynamic compilation is a capability often ’considered harmful’ and suppressed. In 2012, the use of writable machine code is explicitly restricted to Apple and Windows supplied software for iOS and Windows 8 for ARM machines respectively. Nevertheless, there remain many industry supported languages and systems that rely upon dynamic compilation for performance, including Java, JavaScript, .Net, and NaCl.

2.4 Context 2: Interactive generative art

We have presented the potential of reflective meta-programming and dynamic compilation for creativity support tools, but there are also other implications for media-intensive, interactive generative arts.

Generative artworks are produced by utilizing systems of rules that are to a certain extent autonomous, such as computer programs (Galanter, 2003; Boden and Edmonds, 2009). McCormack proposes a biological analogy: the *genotype* is what the artist specifies (a set of instructions, axioms, interaction rules etc.), while the *phenotype* is work as experienced by a viewer (McCormack, 2002).

McLean notes that classifying artworks as generative invokes questions not only regarding *how* the work is made, but inevitably also *why* (McLean, 2011). Generative techniques represent *an intentional degree of controlled separation* of the artist from the work (and a concomitant ambiguity regarding the locus of creativity). There can be many rationales for an artist to ‘remove themselves’ from their work and find creative behavior in autonomous systems. It could be a *pragmatic concern*: a convenience for leveraging the inhumanly vast numbers of parameters computational media can involve. It could be a *conceptual stance*, such as a critical commentary on the social contexts of contemporary art culture or the broader techno-industrial culture we live in (Whitelaw, 2005), or an attempt at understanding art and art-making through objectification (Cohen, 1995; Cope, 1992).

Nevertheless a survey of generative arts today also reveals a prevalent motivation to explore the field of complex autonomous systems as *an art material* of its own right, for the unique, distinctive and apparently open-ended behaviors it can spawn, and perhaps to explore *the fascinating sublime of the infinite and alien* in ways that were never before possible (McCormack and Dorin, 2001). Put another way, an important criterion for many generative art projects is to engineer an open-ended system which in some unpredictable way continues to differ from and surpass itself, producing results which exceed the original specification.

2.4.1 Emergent systems and open worlds

Computational generative art thus reprises an age-old tension between the mechanistic and the mysterious, which persists today in the debate regarding emergence (Hein, 1972; Brodu, 2006). Briefly, emergence refers to qualities of a system that are not fully reducible to terms of their parts (synchronic emergence), or of their antecedents (diachronic emergence). Emergence can also be described in terms of *downward causation*: where superstructures have irreducible influence on the substructures from which they arose (Hofstadter, 1979).

The ontological status of emergence remains unresolved but hotly debated, particularly in philosophy, physics, biology, neuroscience and complex systems research. The weakest and most easily defended stance, known as epistemological emergence, states that a system is emergent if its behavior forces us to revise our model of it; i.e. they surprise us, we cannot predict them *in practice*. A stronger form of emergence holds that these properties and capacities are not predetermined *in principle*.

Several authors have recast the argument in terms of open-endedness. DeLanda for examples compares the properties of a knife, such as the robustness of its edge, as quantifiably reducible to the properties of the molecules it is made of; however the capacity *to cut*, which makes a knife what it is, is unbounded: there are always more things that could be cut (DeLanda, 2011). Within certain kinds of sets, such as the set of possible organic molecules, there is always a possibility to create a novel structure from what currently exists, thus the set is unbounded or *transfinite*. However, at any time only a small fraction of possible molecules can exist (even in the whole duration of the universe, only a small fraction can exist (Kauffman, 2002)). Furthermore, the subset which does exist is largely determined by the contingent history of the whole, thus generalist theories do not capture the important features and behaviors actually in evidence. The more appropriate attitude is known in philosophy as *actualism* (Mullarkey, 2006).

It is easy to see the theoretical parallels between unpredictability (in practice or in principle) and contingent actualism, with the different open-ended properties of computation discussed in Section 2.1, but how can these be made concrete? We could perhaps draw inspiration from one of the best-known examples of a continuously self-differentiating, self-surpassing system: life itself.

2.4.2 Open worlds through computational biology

Many favorable comparisons between biology and computation have been made over the last century. Christopher Longuet-Higgins suggested introducing computer science to theoretical biology in 1969, in order to model constructive organization (Longuet-Higgins, 1969), but the kinship between biology and computation can be traced further back to the parallels between genetic development and John von Neumann's theoretical study of self-replicating machines in the late 1940s (Neumann, 1966).

In order to design a machine which could surpass itself, von Neumann theorized a 'sea' of machines and data (or "tapes"), and demonstrated a set of machine-tape assemblages capable of replicating themselves, and potentially evolving.[25] The biological analogy compares genetic information (genotypes) to the tapes, and organisms (phenotypes) to the machines. Von Neumann proposed his first prototype using a cellular automaton in 1956, and in 1979 Christopher Langton demonstrated a much simpler self-replicating cellular automaton that did not require embedding a pre-defined 'universal constructor' machine (Langton, 1990). A remarkably thorough early example of biologically inspired computation was presented by Nils Barricelli's "Symbioorganisms", demonstrating evolution, cross-breeding, growth and self-repair of computer programs (Barricelli, 1957).

Today computation is increasingly used as a tool of biological science, such as the fast-growing fields of computational biology and bioinformatics (Kasabov et al., 2005), however the simulation of self-surpassing nature is perhaps most clearly pursued in the field of Artificial Life (Langton, 1996; Bedau et al., 2000; Wheeler et al., 2002). Langton announced the field of Artificial Life with its

[25]The image of a soup of replicating universal computers recurs in the recent field of Artificial Chemistries (Dittrich et al., 2001).

first conference in 1989, with a presentation of its goal–to better understand life by recreating it in digital or other media–predicated on the assertion that life is a property of organization independent of any particular substance, and thus not limited to the particular example within which we have grown. Specifically, Langton characterized it as a study of *life-as-it-could-be.*

Etxeberria has usefully categorized Artificial Life projects as falling under three general categories according to their mode of evaluation (Etxeberria, 2002). The engineering aspect produces tools, such as the genetic algorithm, which are valued in terms of their efficiency in solving well-defined problems. The scientific aspect produces models, which are valued in terms of their ability to produce theories of predictive capacity regarding life as we know it. The third aspect is described as speculative or exploratory, producing worlds and instantiations which are more intuitively defined in terms of the 'lifelike', with a diffuse boundary between the natural and artificial, and which are value for their capacity to *sustain creative novelty* or increases in complexity. These worlds are new creations of ontology that offer "the most radical ways of exploring life-as-it-could-be" Etxeberria (2002).

What kinds of models are used to simulate, model or create life in computation? Traditionally dynamical systems are the scientific modeling tool of choice, and they have been used to model everything from ecosystemic population dynamics (Lotka, 1910) to morphogenetic pattern formation (Turing, 1952). Nevertheless, numerous scientists have claimed that dynamical systems alone are inadequate (Fontana et al., 1994). A dynamical system describes the relations between a number of important degrees of freedom, but this set of degrees of freedom must be specified in advance, defining an a priori 'phase space' through which a system moves. *The phase space itself never changes.* Giavitto and Michel accordingly suggested a model of 'dynamic systems with a dynamic structure' combining dynamical systems with rewriting systems (such as symbolic grammars), in which "the structure of the phase space must be computed jointly with the current state of the system", and "the organization of this set is subject to possible drastic changes in the course of time." (Giavitto and Michel, 2001)

In fact Artificial Life researchers have repeatedly emphasized the need to minimize a priori constraints. As practical examples, genetic programming overcomes limits of the more static genotype representation in genetic algorithms (Koza, 1990), co-evolution and ecosystemic environmental dynamics help avoid problems of local minima due to static fitness functions (Adami, 2008; Bousquet and Lepage, 2004; McCormack, 2007), and developmental modeling (Giavitto et al., 2002; Kitano, 1998; Miller and Thomson, 2003) can incorporate the mechanisms of evolution into what is evolved (the system's *evolvability*, Altenberg, 1994). More theoretically, Kampis noted how nature can be easily partitioned into hierarchical systems for specific problems, however doing so eliminates the constructive and polysemous potentials that allow a system to discover new qualities; he proposed instead the use of dynamically connected ensembles of self-modifying systems (Kampis, 1996). Ikegami revisited von Neumann's model of tapes and machines as the interaction of genotypes and phenotypes, noting that although in life a strong separation is maintained to protect the genotype from noise in the environment, this separation is never absolute: "self-interference is required for self-reproduction. In other words, life has to observe itself in order to replicate itself." (Ikegami, 1999)

The claim that an overly pre-determined space is detrimental for (autonomous) creativity also mirrors the observations made regarding creativity support tools in Section 2.2. For example, as a direct parallel of human and biological creativity, Machado et al. applied Wiggins' formalization of Boden's *transformational cre-*

ativity (see section *2.2.1*) to evolutionary computing, to show that systems can be *transformationally* creative if components that are usually static are allowed to change their rules, including the genotype-phenotype mapping, genetic operators, selection procedures, replacement strategies and evaluation functions (Machado et al., 2004). In summary, a vital, characteristic quality of autonomously creative systems is a need to be able to continuously change the rules. Just as dynamical systems need dynamic structure, computational programs may need to continuously rewrite themselves in order to create open-ended worlds.

2.4.3 Artificial life art, artificial natures

The advancements in computational biology and artificial life have persistently fascinated artists, and many techniques utilized in generative art today draw direct inspiration from mathematical and computational models of nature (and artificial life in particular), such as fractals, evolutionary algorithms, and swarm systems (McOwan, 2009; Dorin, 2005; Penny, 2010; Rinaldo, 1998; Sommerer and Mignonneau, 2009; Whitelaw, 2004). Despite using tools of simulation, such art works do not aim to accurately portray the world-as-we-know-it, but are speculative instantiations of worlds-that-could-be: “The artist searching for the computational sublime would find it, perhaps in its ultimate form, by generating an experience of open-ended complexity governed by processes instantiated on a computer... the system would build its own structures and define its own universe." (Dorin, 2005) Whitelaw, Guglielmetti and Innocent describe such worlds as ‘strange ontologies’ that present alternative models of ‘being-in-time’ and relationships of individual and environment: “In works using simulation and related techniques, abstract generative art performs cosmogeny: it brings forth a whole artificial world, saying, here is my world, and here's how it works." (Whitelaw et al., 2009)

Nevertheless, such generative worlds have been criticized for *not being generative enough*, since "entities ... are identical, or belong to a set of pre-defined types, and their properties and behavior are static over time... The environment here is (literally) a blank canvas, inert, empty space... [the agent is] a clone in a crowd, unchanging, with no traction on the space it inhabits, existing in an ongoing, perpetual present." (Whitelaw, 2005) Such works do not sufficiently surpass themselves because they are too static: "the underlying systems themselves are crystalline and impervious." (Ibid.)

The suggestion pursued in this dissertation is that to better achieve self-surpassing worlds we should avoid a priori constraints and design systems that rewrite themselves as they run, within contingently evolving regions of transfinite spaces of possibility. With the feedback of interactivity, the "structure must be adaptive implicitly or physically, to accommodate the spectator's responses, in order that the creative evolution of form and idea may take place." (Ascott, 1964).

Ji proposed the term "artificial natures" to describe complex artificial life worlds that recreate nature-like experiences within immersive and interactive media (Ji, 2011). However, when generative worlds are presented in such interactive, immersive formats, the real-time efficiency of performance becomes a critical factor in the feasibility of a work: the quantities of data processed are high, but the throughput latencies must remain low. A strategy integrating meta-programming and dynamic compilation may be essential to maintain performance while supporting the self-rewriting needed to avoid being "crystalline and impervious". A specific scenario of such an "artificial nature" is explored in Chapter 5.

Chapter 3

Method

The aim of this chapter is to determine the theoretical design underlying the practical work in the subsequent chapters. Although this chapter is presented before the practical work, in reality both theory and practice have evolved mutually. It begins with an analysis of the problem in technical terms, drawing largely upon relevant literature in computer science theory and programming practice, collecting insights leading to the presentation of a general model.

Dynamic compilation has the potential to reconcile goals of flexibility and efficiency; but adding a compiler to a program is not the whole story. The model must support unbroken continuity between changes in program and must deal with challenges of dynamic memory management (including the memory of generated machine code). It must also present the open-ended capability to users in a cognitively agreeable interface, including where to place the division between what is modulable and what is atomic.

The general model presented is closer to a virtual machine than a task-oriented tool: not a fixed solution to a fixed problem, but a starting point from which to adapt to a variable environment. This machine hosts a population of executing code fragments as *components*; in an audio synthesis tool these components may correspond to instruments, or in a virtual world they may correspond to artificial life organisms, for example. The population varies over time through an interleaving of component *life cycles* which are punctuated by a dynamic *workflow*. The virtual machine must provide sufficient capabilities to make these life cycles and workflows natural and efficient.

This chapter follows the model and workflow to outline its requirements and strategies of implementation, deriving from prior research as much as possible. Reflection, meta-programming and compilation are not mainstream knowledge in the intended readership of this dissertation, so it is hoped that this chapter may also serve a pedagogical role.

3.1 Strategic model

3.1.1 Augmentable domain-specific virtual machines

As noted earlier in Chapter 2, the original modus operandi of programming was to perform a specific task and solve a specific problem. Even a compiler performs a very well-defined task: translating from a user language to a machine language. But in the domains of interest to this dissertation–creativity support tools and interactive generative arts–tasks are not sharply defined in advance and continue to evolve during execution. At best they can be characterized as addressing specific problem *domains*.

We have seen that interpreted languages, and the virtual machines that implement them, can address problems not fully stated in advance. Although many interpreted languages are designed to be *generalist* by default, and agnostic to the vastly different characteristic features and cognitive models of the problem domains to which they could be applied, some languages, known as domain-specific languages (DSLs), are expressly designed to map to the cognitive models and recurrent challenges of a specific problem domain.[1] For example, languages specific to 3D graphics (such as GLSL) often include specialized syntax constructions to make vector and matrix operations easier to express.

We saw that in domains of interactive arts the performance of media processing is critical. Although this is not an important concern for the cognitive representation of the domain, it does impose very strong constraints on software implementations; constraints that are often very difficult to reconcile with an interpreted strategy. However we also saw that, through the use of translating meta-programs such as compilers, the language in which a program is stated can be independent of the language in which it is implemented. By utilizing dynamic compilation, we can augment an existing program with newly defined algorithms specified in one representation (such as a domain-specific language) which are implemented distinctly as efficient machine code routines.

We therefore suggest a strategy for creating creativity support tools and interactive generative artworks through the construction of *augmentable domain-specific virtual machines.* Each time that a new routine is compiled, it effectively augments the virtual machine with a new *component,* which is integrated with and made available to the ongoing processes of execution.[2]

This can be seen as an extension of the modular strategy outlined in Section 2.3 in that the specifications of new components are expressed in terms of

[1]Domain-specific languages can be implemented by extending (or constricting) a general language, or written from scratch to address the problem domain.

[2]Augmenting with components could be viewed in terms of extending to the instruction set of the virtual machine, however components are usually significantly more complex than the instructions of interpreters (else the performance advantage would not be so appealing).

domain-specific modules and features. The modular approach offers many advantages, such as ease of conceptual decomposition and code re-use. However, with a workflow based on dynamic compilation, the separations between modules at the level of component description can be eliminated during the translation to component implementation. The machine code for new components is not limited by the modular boundaries in the description, and can better address performance concerns. The description itself becomes more declarative than constitutive, and by being liberated from implementation details it can more closely be fitted to domain concepts and become more flexible in users' hands. Accordingly, the intrinsic modules in terms of which the components are expressed can be very low-level without compromising performance.

Nevertheless, since compilation can have significant overhead, choices of when and how to fragment a program into components remain critical.

3.1.2 When does a program change?

A general strategy in modeling and solving problems is to divide the problem domain into those deterministic parts which cannot change (the facts and constraints) and those indeterminate parts which can change (the free variables, flexible relationships and open endings). The division centers around decisions: deterministic decisions can be made once and for all, and the result stored as program data before execution begins ("baked in" as constant values, type definitions and linear instruction sequences); what remains are the decisions which must be re-evaluated many times as the program runs (expressed through variables, data-structures and control-flow).

This way of breaking down problems is central to many deductive program optimization techniques. For example, if all inputs to an expression are provably constant, the result is constant also, and the entire expression can be replaced with its value (constant folding); likewise if an expression result is provably never used in any other expression, it can be removed entirely from the program (dead code elimination). It lies at the heart of the Partial Evaluation technique (Consel et al., 1996; Draves, 1998).

When code can change during run-time however, the strategy can be refined according to frequencies of change:

1. Static parts that can never change.
2. Dynamic parts which change during execution.
    (a) Parts that change infrequently, with a relatively stable mixture of general and particular variations.
    (b) Parts that change frequently, with a high degree of particularity.

A biological analogy may be helpful. The basic chemical and evolutionary laws never change (category 1), but cells exist in a soup of ever-changing concentrations of chemicals (category 2(b)). Nevertheless, the genetic information which directs a cell's activity is generally considered to be the same throughout the lifespan (category 2(a)).

Clearly category 2(a) is the appropriate site for dynamically compiled components, where the benefits of optimized machine code can outweigh the overhead of generating it.

However it is important to note that grafting new components into an executing process can occur only at pre-existing points of dynamic indirection, such as dynamic data structures and dynamically bound functions. Change must be anticipated. Furthermore, not only must points of potential change be explicit in code, but the system must also reflexively know how to manipulate these points, how to create and apply changes to itself.

3.1.3 How does a program change itself?

Reflection is the aspect of meta-programming in which an executing program process operates over itself, reading and possibly modifying itself: "the ability of a program to manipulate as data something representing the state of the program during its own execution" (Malenfant et al., 1996). In more formal terms, aspects of program P in language L are represented explicitly in L and available in P, with causal connection, such that modification to one affects the other. It is the feedback path by which the *program-as-process* connects to the *program-as-data.*

To do so, parts of the executing code may include references to data structures that describe their semantics. For example, the in-memory representation of an object may contain a pointer to a data structure representing properties of its class. However reflective capabilities need not reside in objects directly, but can be provided by reflective wrappers or fully separated meta-objects (using the *proxy* and *traits* design patterns of Gamma et al., 1993). A meta-object is an entity that creates, manipulates or implements another 'base' object, and stores information about the base object.

Bracha and Ungar suggest several key terminologies to characterize reflective capabilities (Bracha and Ungar, 2004):

1. *Introspection*: a program can examine its own structure.
2. *Self-modification*: a program can change its own structure. This includes the evaluation & execution of code not necessarily available before the program

began running.

3. *Intercession.* a program can modify the semantics of the underlying language from within the language itself (Kiczales and Des Rivieres, 1991).[3]

Malenfant et al. on the other hand proposed the term *behavioral reflection* to describe the ability of a program to modify itself at run-time, including both structure and language (Malenfant et al., 1996). The set of possible machines that a dynamic program can become is limited by the reflective language and the degree to which the initial program is subject to its own capability to change.

Supporting behavioral reflection raises many interface design questions, including: what semantic aspects should to be visible to the program, how are they represented, and to what extent should new changes affect previously compiled components? For dynamic compilation, it is essential that the compiler knows how to generate code which understands the reflective protocols. Generated code must also accurately represent basic protocols for existing function addresses, type sizes, calling conventions and so forth, in order to implement reflective features.

The host program requires reflective capabilities at the anticipated points of change in order to invoke a *workflow* of code generation and receive the new components produced. It must also be able to refer to reflective properties of a newly loaded program component to know how to use it, and the loaded component itself may need to use the reflective capabilities of the host to know what it can do: "A subsystem needs to have self-knowledge in order to function effectively in open systems, in order to understand its own abilities and well as the limits of its knowledge and power." (Hewitt and Jong, 1986)

[3]Semantic intercession could be achieved by modifying the behaviors of meta-objects, or the grammar actions of the language. Piumarta and Warth have described such languages as "engines of their own replacement" (Piumarta and Warth, 2008) in their explorations of late-bound syntax, semantics and pragmatics.

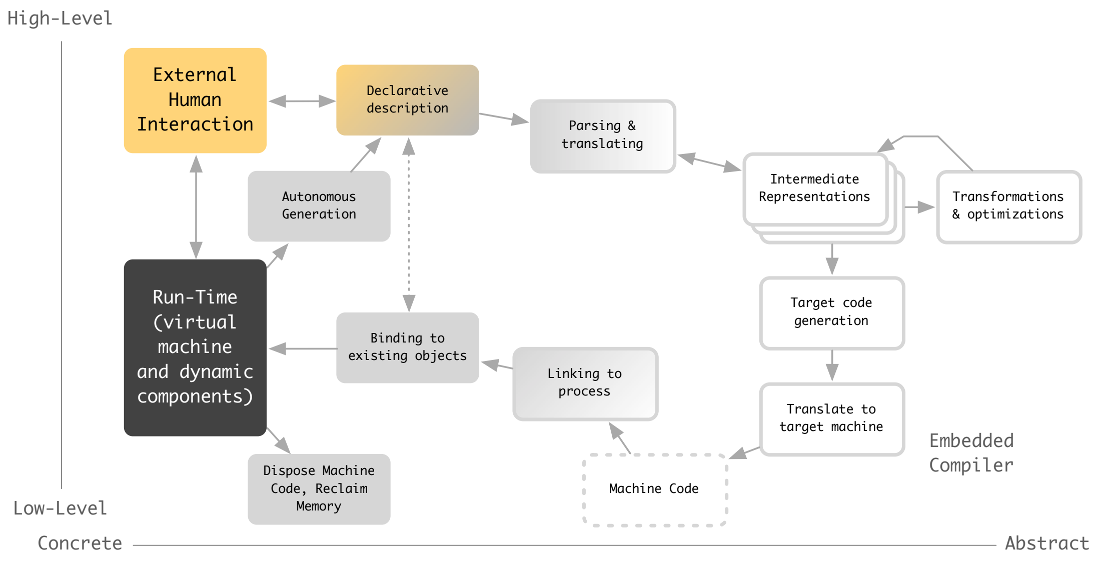

High-Level
External Human Interaction
Declarative description
Parsing & translating
Intermediate Representations
Transformations & optimizations
Autonomous Generation
Run-Time (virtual machine and dynamic components)
Target code generation
Binding to existing objects
Linking to process
Translate to target machine
Embedded Compiler
Dispose Machine Code, Reclaim Memory
Machine Code
Low-Level
Concrete
Abstract

Figure 3.2.1: The workflow of introducing components to the virtual machine, laid out in an approximate field with axes of low-level (machine friendly) to high-level (human friendly) and concrete (run-time specific) to abstract (generic) activity. Colored boxes denote parts of the external (human) interface, the dark box denotes the accessible state and behaviors of the run-time (the static virtual machine and dynamically added components), the light boxes denote parts of the system which mirror a static compilation workflow, and the mid-grey boxes denote the dynamic and reflective meta-programming processes that bind these systems together.

3.2 Workflow

Introducing new components to the run-time involves a workflow involving many stages. The demonstrations described in following chapters implement this workflow differently, but share a general structure broadly depicted in Figure 3.2.1. A round-trip of the workflow is triggered by the request to create a new component according to a declarative description (whether human- or machine-generated). The workflow proceeds through a series of transformations to translate this description through several abstract forms into an in-memory machine code implementation, and then to package this machine code appropriately and integrate it through reflection mechanisms within the ongoing, executing run-time.

Since several requirements of the model are well known to interpreted languages, with solutions in the form of dynamic binding/dispatch, polymorphism, garbage collection, lazy evaluation etc., some of the requirements of the workflow can be satisfied in practice by leveraging an existing dynamic language. The demonstrations in the following chapters make use of the Lua programming language (Ierusalimschy, 2009a) to orchestrate the workflow process. Lua is fast, very legible, and supplies many dynamic features inherently. Lua is also easily bound to C code, which is important when embedding the workflow within a larger host program or integrating with media-processing libraries.

3.2.1 Conventional compilation workflows

Many stages of the workflow are very familiar from the static compilation workflow from description to code generation (Aho et al., 1986), except that what is being compiled is an isolated component, rather than a whole program, and that new program components may be specified at run-time, either directly by the user, or generated by the program itself.

Declarative descriptions

We saw in section 2.1 that programs (and thus program components) are expressed in terms of formalisms, which could include textual languages, visual languages, and languages of data structure. In this section we thus take a perspective related to model-driven engineering (MDE, Schmidt, 2006), in which software design begins from a more abstract, declarative level of models, closer to the features of the domain in question than to specific features of computing and software.

A formal language is specified by a model which identifies a set of elements (primitives), rules for how they can be legally combined (syntax), and indications as to what the elements and combinations mean (semantics). A *concrete form* of a program is how it is materialized (such as text, graphics or data structures), while the *abstract form* or *abstract syntax* of a program is the logical, syntactic structure it follows.[4]

Primitive elements of a concrete form may be manifest as characters and lexemes in a textual language, graphemes and geometric parts in a graphical language, or field types in a database. In textual languages, the syntax is usually expressed as a grammar of terminals and non-terminal productions. In a visual programming language, the constraints of the syntax are not necessarily captured by a tree structure, and may require a more general graph formalism than a grammar. The syntax of a data formalism is the rules by which compound structures can be constructed, and the primitive elements are those members of data structures which are not compound.[5]

The same abstract form may have many concrete forms, and acts as the mediator between these concrete forms and the actual, operational semantics. A parser can recognize a program in one of these concrete forms, and return the equivalent program in its abstract form. When the source language is textual, the destination structure often follows the tree-like structure of the source language's grammar, and is therefore sometimes called a "parse tree" or "abstract syntax tree" (AST). The semantics of a formalism may be provided by documentation, by executable implementations, or by translation to another language (as implemented by a compiler, for example).

[4]The term abstract syntax derives from natural-language research, as the underlying hidden structure of sentences (Chomsky, 1965).

[5]The essence of the LISP language is a succinct example. The simplest representation of a fragment of LISP code in memory is a hierarchy of lists. A list is defined (and represented in memory) as an ordered set of members that are either references to other lists or irreducible primitives such as symbols and numbers.

In many situations the concrete forms of languages that humans like to use tend to be full of redundancy and diversity, with many ways to articulate the same result. Over its history programming has transitioned “up the language-generation ladder”, from machine and assembly languages up through imperative and functional to contemporary styles including visual, constraint, logic and most of all multi-paradigm, multi-level languages that are far removed from the hardware (Harel, 2008), an evolution built upon ever more powerful parsers, compilers and other translators, able to render these high-level intentions back into machine code.

Intermediate representations, analysis and transformations and optimizations

Neither the high-level human-readable languages nor the machine code languages are easy to operate over, since both include many concrete irregularities and redundancies. Meta-programming is easier if the program representation to be manipulated is translated into more abstract, homogenous, regular, normalized forms. An Intermediate Representation (IR) is a language used to represent a program internally to a compilation pipeline[6]; thus a high-level, human-oriented language is translated into IR, where it undergoes various transformations, and is finally translated into assembly, byte-code, or machine code. Due to its abstract nature, an intermediate representation can also serve as a portable ’common language’ between distinct source languages, distinct compilation processes and tools, and distinct targets, as originally proposed with the UNCOL language (Conway, 1958) and realized today by LLVM (Lattner and Vikram, 2007).

[6]Some development tools also support the direct execution (interpretation) of intermediate representation, which can be helpful in debugging.

An intermediate representation is typically designed to make important features of the program, such as control flow and data flow, explicit in the representation, making forms of analysis common to typical transformations and optimizations easier and more efficient to implement. For example, operations may be limited to only a few minimal types, since it is easier to manipulate a verbose but regular language than a concise but irregular one. It is also common in IRs to abstract away hardware limits, such as treating the number of data registers as theoretically infinite. Some formal models for intermediate representations include static single assignment (SSA), administrative normal form (ANF) and continuation passing style (CPS) (although these can be shown to be equivalent(Kelsey, 1995; Kennedy, 2007)).

Many optimizing transformations can be applied while a program is in an intermediate representation. In general, an optimization is a transformation of the behavior of a program into an equivalent representation (i.e., one that produces the same output behavior), with the exception of placing less demand upon CPU, memory or other practical resources. Making a program more efficient is specializing it to better fit the run-time environment. Optimization is generally driven by inferential logic: utilizing what is known and re-arranging what is unknown, as noted in Section 2.1.2. They often require a prior stage of analysis in order to generate *annotations*: additional derived facts at each stage of the algorithm (such as types and ranges of variables) stored locally in the intermediate representation, which can be used to prove the validity of an optimizing transformation. Standard optimizations are well understood (Aho et al., 1986) and need not be discussed further here.

The last stage of the conventional compilation workflow is translation to machine code. It is another example of *translational semantics:* the semantics of the source intermediate representation are expressed in terms of fragments of the target language. In summary, the predominant situation is that a source program in a high-level, human-readable language is *parsed* into a data-structure closely related to the source language's abstract model (such as an AST). By translating this structure through one or more *intermediate representations* more amenable to meta-programming tasks of analysis and transformations (including *optimization*), a compiler finally utilizes a process of *code generation* to produce a semantically equivalent but more performant piece of machine code. These are the terminologies generally used, however we have seen that this can also be simply viewed as a series of translations between language formalisms.

3.2.2 Embedding a compiler

Rather than targeting machine code directly, an appealing option is to generate a higher-level target language for which good machine code compilers already exist. For example, many functional languages compile to C as the target language and then rely on the power of existing C compilers for portability and performance.

Today there are several very well-developed and portable libraries that can support compilation to machine code at run-time, as detailed in Section 2.3.3. Target languages include C/C++ for Clang (LLVM), Lua for LuaJIT, JavaScript for the virtual machines in recent web browsers, Java for the JVM, and so on. The author chose to leverage the LLVM compiler infrastructure libraries in the experiments in this dissertation, but has also experimented with using LuaJIT and OpenCL as meta-programming target languages. LLVM is open-source, cross-platform, strongly-backed and relatively easy to integrate into other C-based systems (Lattner and Vikram, 2007).

Generating a higher-level target language can offer more than portability and performance. A human-readable target code is beneficial for understanding errors during development. It is also easier to generate code binding to the reflective features in the host environment, especially if these are also expressed in the same language. For example, generated C/C++ code can directly include the same *header* definitions against which the static portions of the host application were compiled, providing a degree of reflection for free.[7]

On the other hand, the run-time overhead of introducing additional layers in compilation must be taken into account. In addition, static code in the host application may be needed in order to direct the embedded compiler within the workflow.

Luaclang

To provide application-driven services of machine code generation, the author created an interface to LLVM and Clang usable within the Lua dynamic language (the *luaclang* library).[8] This makes dynamic compilation of C++ code available to be controlled by a dynamic language, and encapsulates many of the complicated aspects of machine code generation into a relatively simple workflow (see Listing 3.1).

Luaclang also takes care of memory management of machine code. All machine code associated with a particular compilation kernel is stored in a garbage-collectible "jit" object; and furthermore each function pushed into Lua retains a reference to this object. In this way, only when none of the functions nor the "jit" object itself are reachable from Lua is the machine code freed.

[7]A more tangential benefit is that at any moment the target code can be exported for use in a more traditional static workflow; a dynamic program can be frozen into a static one.

[8]An open-source project available at https://github.com/grrrwaaa/luaclang

Two further features are provided by the luaclang library. First, luaclang includes a string of C++ code which can be compiled to provide a binding to the LLVM C++ API, allowing the construction of LLVM IR directly, rather than by compiling C/C++ in Clang. Secondly, luaclang includes an FFI (Foreign Function Interface) declaration of LLVM/Clang headers that can be utilized within LuaJIT. Both features are aimed toward reducing the overhead of invoking the LLVM compiler.

3.2.3 Code generation

Unlike grammars for language parsing, there is no widely accepted theoretical framework for code generation. Nevertheless several themes can be identified.

The workflow of a generator must find a match between the structure of the source and the structure of the target, but in many cases these structures are not isomorphic. For example, the member variables of an object might be defined in only one location of the source model, but may need to be referenced in multiple locations of the target code, including object type declarations, constructor, initializer or destructor methods, member setter and getter methods, and so on. Generally the construction of this match can be dominated by the the source or by the target structure, however these target- and source-driven strategies can be combined: at the large scale, a template-driven approach has clear advantages; while at the small scale, a source-driven approach is more flexible.

Source-driven code generation is dominated by the structure of the abstract source model, and may also be described as *push-driven*. The general method recursively visits nodes of the source model to produce fragments of target code. This may be implemented according to the *Visitor* design pattern (Gamma et al.,

Listing 3.1: A minimal sample of using the *luaclang* module in Lua to compile a new native function written in C++. The sample function is written to match Lua's C ABI, so that it can be immediately demonstrated within the Lua script; however the *luaclang* module will compile functions of any signature (and make them available as pointer userdata in Lua). All created objects are implicitly garbage collected, but *luaclang* automatically ensures that the machine code for the generated **timesTwo** function is not collected until the function itself is no longer reachable.

```
-- load the luaclang module:
clang = require "clang"

-- create a compiler:
cc = clang.Compiler()

-- create a string of C/C++ code:
code = [[
    // declare necessary parts of Lua's C ABI (lua.h):
    typedef struct lua_State lua_State;
    double lua_tonumber(lua_State *, int);
    void lua_pushnumber(lua_State *, double);

    // define a function conforming to Lua's C ABI:
    extern "C" int timesTwo(lua_State * L) {
        double n = lua_tonumber(L, 1);  // first
           argument
        lua_pushnumber(L, n * 2.);      // first result
        return 1;                       // one result
    }
]] -- end of C/C++ code declaration

-- compile C/C++ code into an LLVM Module:
cc:compile(code)

-- optimize the Module:
cc:optimize("O2")

-- compile the Module to native machine code
-- wrapped in the "jit" object:
-- (the compiler can now be re-used)
jit = cc:jit()

-- retrieve a function (in this case as a Lua function):
timesTwo = jit:pushcfunction("timesTwo");

-- call the generated function:
print(timesTwo(2))          --> prints 4
```

1993), in which each node type has a corresponding code generator routine. However the source-driven generator must also know how to locate the proper parts of the target for code interpolation (worse still, they may refer to parts that have not yet been generated). For example, adding an oscillator operation to an object method may also require adding a declaration to the object for the oscillator phase, and an initialization of this phase, which may require the addition of another method... Clearly, fragments cannot be combined immediately, since the order of traversal of the source does not necessarily match the order of the final output code. Source-driven generators may instead generate the target language in its abstract form, which can be easily made concrete at a later stage. Deferring concretization not only allows non-linear insertion, but also allows rearrangements of the target code while traversing the source. However it requires defining a somewhat abstract model of the target language to insert into.

Target-driven code generation is dominated by the structure of the desired output code. In particular, it is often specified in the form of a template, in which invocations to convert source model components are written inline, using special syntax, within a larger fragment of static target code. Template-driven methods mix static text with annotated variable insertion points, naturally capturing the invariant text between variant rules, with advantages of readability at the macro scale. These variable points become specialized according to values that are found in the source model. Inline invocations in the template include requests for data from the current location in the source model, requests to change the current location in the source model, and possibly more complex operations over the data such as conditionals and loops.

Template-driven code-generation has been widely used in web development, such as inserting data-driven variations within otherwise static HTML code (see

Listing 3.2: A minimal example of a template for generating HTML. The **{$title}** invocation indexes the data model for the value of the "title" field and substitutes it into the output code (and similarly for **{$username}**).

```
<html>
    <head>
        <title>{$title}</title>
    </head>
    <body>
        Hello {$username}.
    </body>
</html>
```

Figure 3.2). They are appealing to designers, since they closely resemble the final code produced. It is relatively straightforward to start from a static example of target code, and gradually insert template components to modularize the template. However, if multiple locations of the template refer to the same locations in the source, computations and derivations from the source may be unnecessarily repeated. The input tree may have semantic constructs that cannot be expressed in the target language (e.g. some control flow structures require a *goto* instruction, which is not available in languages such as GLSL). Code generation of a node may depend on type annotations and distant contextual information to be valid.

A simple template-driven approach therefore does not scale as gracefully as the source model increases in flexibility and complexity. The appealing readability of a template begins to evaporate, and computations of even moderate complexity simply cannot be expressed in a single linear template. A natural extension to accommodate complexity is therefore to define a template recursively, in terms of a hierarchy of sub-templates with one root template. In a sense this closely mirrors the structure of a grammar, whose production rules contain concrete syntax as well as references to other non-terminal rules. This grammar is effectively an application-specific specialization of the target language's grammar.

Parr contributed toward the formalization of code generation by proposing a strict Model-View separation for template-driven generators, in which the template invocations are strictly limited to copying data and testing for existence (Parr, 2004). The goal is that the template should not contain any executable code or be capable of producing side-effects, effectively making it a purely functional language. Instead of performing computation within the template directly, Parr suggests a multi-pass model in which the source model is *preprocessed* to produce an *intermediate model* that closely matches the template. Placing all computation prior to templating avoids repetition and reduces fragility, without sacrificing expressive power.

A multi-pass approach to code generation also presents the opportunity to reconcile the anisomorphisms between the source and model structures: the translation from original source model to intermediate model is source-driven, while the interpolation of the intermediate model into the template is target-driven. Compact components of the source that have multiple consequences in the target can be 'expanded' first in the intermediate model. Expanding operations here is an abstract equivalent of macro expansions in concrete code, and similar issues of macro hygiene apply: any new variable names introduced must not clash with existing names (this is relatively easy to ensure with a global **gensym()** function to generate unique names, and a memoization pattern to map generated names to locations in the source model). The intermediate model becomes a kind of 'reduced normal form' which is trivial for the template to handle (though it will still require invocations to insert elements and navigate the structure of the model). Note that even if the source is a tree, the intermediate model need not be: a graph with multiple references to certain nodes may be far easier for the template to use.

The code generators described in Chapters 4 and 5 target C code, and in some cases also Lua code. These generators use a multi-pass approach with target-driven workflows at the macro-scale, and source-driven workflows at the micro-scale. The code generator first converts the received source model into an abstract data structure representing the program (the intermediate model) in a form that fits the constraints of a pre-defined hierarchical template (including generating expanded operations with hygienic names), traverses the template hierarchy to invoke rules of string generation using this intermediate model, and merges the generated structure of strings into a single fragment of target code at the last step. This target code is then translated to machine code using *luaclang*.

3.2.4 Linking, binding, registration and reflection

Once machine code has been generated, it must be bound into the run-time. Making the machine code executable and addressable is part of the general task of a dynamic compiler (equivalent to a linking stage for a static compiler), but to be actually called from the running program, it must also be integrated with the active state and behaviors through some form of dynamic registration and reflection. If the new code constructs objects, these objects may also need to be interfaced with the run-time.

The run-time itself needs to provide some 'loose coupling' features with which new components can be registered. There are many ways this can be achieved: a globally accessible dictionary under which components are registered by unique indices, publish/subscribe and other 'event-based integrative' registration systems, a pre-defined plug-in style of registration or polymorphic object model to which the generated code template conforms, and so on. Registration systems must also require appropriate means to un-register.

Whatever the registration system, a minimum level of low-level protocol glue is needed to implement it. By far the most widely used today is the C language application binary interface (ABI), which specifies the memory layout and calling conventions for defined data structures and functions. Many dynamic languages provide a foreign-function interface (FFI), which defines an interface by which routines in one language can call precompiled routines in another. In most cases the FFI uses the system's C application binary interface (ABI). Generated target code can use the same strategy. If the target language is C or C++, it can be as simple as making the appropriate "extern" declarations or including the original headers, allowing the template to freely integrate existing library and application functions and types.

Allocated objects may also need to import existing state from the run-time. For example, if the new component is a direct replacement of a previous component, care must be taken to disconnect all references from the old instance and connect them to the new one, and copy across important state, in order to present an apparently seamless change.

It may also be desirable for generated code to maintain a copy of the high-level description from which it was constructed, so that it can be copied or mutated at a later point in execution. It may even retain a reference to an object representing the machine code or the dynamic compiler itself, such that dynamically generated components can know how to inspect, remove, and even extend themselves.

3.2.5 Additional requirements

Memory management

Systems that are highly dynamic in structure also incur additional complexities of memory management. Dynamically allocated memory for objects in regular programs must be de-allocated when no longer used, so that the memory can be re-used and total memory requirements stay low. The complexity of memory management is the web of references between dynamically allocated objects and code: data structures and functions must remain valid at their address for as long as that address is reachable by active code. Garbage collection is the means by which memory addresses which are no longer reachable by the run-time are released for re-use. With dynamic compilation, the generated machine code resides in memory and must also be managed. This memory must remain valid for as long as there are potential execution paths that reach the functions it contains. Some policy must be in place to ensure that unreachable dynamically compiled code is garbage collected.

Different policies for managing garbage collection are available. It can be left entirely in the programmers' hands to ensure that allocated memory is properly freed. Dynamic languages tend to use many more dynamic memory allocations, and often provide automatic reference management and *garbage collectors* to remove the burden of memory management from the user. Objects can use reference-counting mechanisms to retain knowledge of whether they are still reachable, or an external tally can do the job for them. Alternatively, policies may be in place to logically ensure the duration of objects in relation to other objects (such as the *arena* design pattern, (Gamma et al., 1993)). Often before releasing an address, certain 'clean up' code must be run to release resources uniquely referenced by the objects being freed. Triggering clean-up code and actually releasing memory could be the responsibility of the referencing mechanism, or a separate process.

Performance

In the context of this dissertation, an important goal is that the entire workflow presented above, from parsing through code generation to linking and binding is both cheap and fast: "Compilers, linkers, analyzers and the like are all traditionally considered to be "out of time" processes. However, when integrated into a dynamic, real-time, causal development process, such tools must be reactive, complying to task-domain time and concurrency constraints." (Sorensen and Gardner, 2010)

The performance benefits of dynamically compiled code must outweigh the cost of adding a compiler to the program as well as the cost of running the compiler for that particular piece of code. For a real-time domain, the time taken to compile a new code fragment must achieve a minimum deadline to be viable at all.[9]

We could take the potential of behavioral reflection and dynamic compilation to the limit, such as modifying the semantics of the language and its objects at run-time (Kiczales and Des Rivieres, 1991; Piumarta and Warth, 2006), however for the domain-specific problem area of media processing we don't necessarily need every step of a program to be modulable and we can provide strong hints regarding anticipated points of change. Similarly, many languages supporting dynamic compilation do so implicitly: a function is first interpreted, which may (sooner or later) trigger compilation of itself, and replace the interpreted implementation (or part of it) with the optimized one. However for this dissertation we do not

[9]It is for this reason that explicitly driven compilation, rather than background implicit or hotspot compilation, is appropriate to many real-time domains.

consider implicit compilation appropriate: if a function describes performance-critical operations such as audio signal generation, interpretation is often simply not viable at all.

Failing gracefully

For real-time interactive arts, it is essential to gracefully handle errors in the workflow without crashing or invalidating the whole system due to memory leaks, infinite loops and so on. For creativity support tools, verification throughout the workflow can trap problems and report them to the user. For example, type analysis within intermediate representations can be a powerful aid to ensuring validity of target code. For generative artworks however, appealing to the user is not possible, and all errors must be otherwise handled gracefully.

Not all errors can be detected at compile-time. Retaining human-readable information (such as line numbers, variable names etc.), which is otherwise irrelevant to the machine, is essential for run-time feedback to indicate to the user which part(s) of the source may be the cause of a problem. Classic debuggers use embedded reflective marker information (debug symbols) within the machine code to provide this reversibility for the user.

Chapter 4

Practice: Authoring Environments

This chapter details experiments using meta-programming and dynamic compilation to extend the capabilities of existing creative authoring environments. The investigations in this chapter concentrate on the specific sub-domain of digital audio synthesis for convenience, though the issues can be readily extended to other media. Audio is an excellent test case: even the smallest failure to meet stringent real-time performance demands can be painfully obvious. The general domain problem is the flexible specification of a data-flow of signal processing, and its efficient implementation within a buffered context.

Specifically, this chapter describes the *audio.Def* module within the *LuaAV* application, and the *gen~* extension within the *Max/MSP/Jitter 6* application, both of which allow the run-time creation and playback of optimized audio synthesis processes from arbitrary and low-level components.

The trade-offs of the modular 'black box' approach and buffered processing, and the advantages dynamic compilation promises, were discussed previously in section 2.3. Briefly, dynamic compilation may circumvent problems with the modular approach by allowing the definition of new native code processes at run-time, and buffered ‘block-rate’ memory usage and feedback limitations by allowing these processes to embed arbitrary yet efficient low-level processing within single-sample feedback loops.

4.1 The Music-N Paradigm

> “Software must remain sufficiently general, and must not impose too many restrictions on the sounds that can be obtained from it. It should be simple to use, and it should permit the easy specification of single complex sounds or of series of complex sounds. Finally this software should be efficient, and should not require considerable computing time.” (Risset, 1971)

It was noted already in Section 2.3.1 how the unit generator concept prevalent in audio synthesis and composition software traces its origins to Max Mathews' *Music-N* series (and *Music III* in particular). The influence of this work remains relevant half a century later, most evidently in *CSound* (Boulanger, 2000; Lazzarini, 2005; YI and Lazzarini, 2012), and is well worth reviewing in greater detail.

Mathews' *Music III* introduced the unit generator concept within a familiar musical analogy: to design an *orchestra* of multiple *instruments*, which can be individually invoked from a *score*. Both instruments and score are specified declaratively, resulting in the generation of corresponding programs, which when run together generate digital audio data. Pope thus likens the structure of a composition directly to a program to produce audio samples, in which instruments of the orchestra are subroutines to be invoked in the instructions of a score, with reference to shared data such as sampling rate and wavetables (Pope, 1993).

Efficiency and flexibility were both clear motivating concerns in the design of *Music III*: "The program, as written, is a compromise between efficiency, flexibility, and simplicity... By virtue of the general nature of the compiling program a great variety of instruments may be produced, and the instrument programs are quite efficient in terms of computer time." (Mathews, 1961) Furthermore, the instrument specification itself utilized a form of code generation: "The compiling program was greatly simplified by the use of *macro* instructions, which specify a sequence of computer instructions by means of a single statement. In this way each unit generator can be specified by a single macro statement." (Mathews, 1961).

The instrument specifications are entered into the computer using these macros in a more human-friendly form of byte-code (input via punched cards), and the output is an *orchestra* program (in assembly code) that can interpret scores. Instruments are specified in terms of a small number of primitive signal generators and filters (the 'unit generators'), whose inputs and outputs are connected by further declarations. An individual macro specifies a unit generator type (oscillator, adder, acoustic output etc.), the instrument number, the unit generator number within the instrument, the output connection, and the wavetable references and default input parameters specific to the unit generator type. The combinations of macros for a given instrument produce assembly for a closed subroutine, whose jump label (subroutine name) is derived from the instrument name.[1] This subroutine is called for each active voice of orchestra, for each sample of output. A separate macro added at the end of an instrument definition creates the connections between note parameter indices (to be read from the score) and unit

[1]Incidentally, Mathews notes that, since the macros produce assembly, it is perfectly possible to intersperse assembly instructions and macros together. "This ability to fall back on basic machine language is always desirable in a compiler." (Mathews, 1961)

generator inputs. This results in another subroutine, called the 'setter', which is invoked at the start of a new note in order to initialize the note parameters.

Along with the instruments, the orchestra includes a score-reading and sound-generating program, which can directly address the instrument subroutines. The score-reader sorts the input note data for a given measure, and then executes the instrument subroutines sample-by-sample for each interval within the measure. This continues for each measure in the score.

In summary, we can identify several levels of abstraction:

1. The music environment knows how to 'run' a score program and what to do with the audio sample output (e.g. write to tape/disk).

2. The score is constructed of commands to initialize and execute instrument subroutines.

3. The instruments are specified by initialization and execution subroutines, constructed by means of unit-generator macros.

4. The unit generator macros compose pre-compiled lower-level instructions.

The longevity of the *Music-N* approach can perhaps be attributed to it being a good fit to how we think about music traditionally, the differing characteristic problems of sample synthesis versus macrostructure, the generality offered by the instrument/unit-generator abstraction, and relative ease by which it can be implemented efficiently. On the other hand, the division between articulation of processes (the orchestra of instruments) and musical data (the score of instrument invocations) has at times been considered problematic (Puckette, 2004).

4.1.1 Interactive Audio Synthesis

Where in the early days of computer music it could take hours to synthesize a few seconds of material, advances in computing power through the following decades made faster-than-real-time digital audio synthesis, and thus also interactivity, increasingly feasible. As noted in Section 2.2, interactivity is not only of creative value in the studio context, but also makes possible performance and improvisation.

However supporting real-time audio synthesis requires the unit generator and instrument code to accommodate additional performance constraints: interactive audio synthesis is one of the most time-critical problem areas in computing. In particular, audio signals are not processed one sample at a time, but instead in buffers (of typically between 2-25ms duration). A sweep of the program's entire synthesis algorithm must occur within these deadlines. On general-purpose computing hardware the CPU shares time between many tasks, and the cost of switching context between tasks would be prohibitive if performed per sample. Interfacing with digital-analog converters typically requires buffering in any case. Audio synthesis must reliably produce the samples of a buffer within the fraction of the duration granted by the CPU. Most audio software today therefore places all audio signal-processing in a dedicated, high-priority system thread to improve reliability. Within this thread, systems calls of unpredictable duration (such as file accesses, memory allocations and user interface activity) should be avoided (Dannenberg, 2005). Managing communication between audio and other application threads can add significant complexity to program implementations.

Given these constraints, audio software may be immediately interactive by processing audio streams received externally, from sources such as microphones and instruments, but also from other software (such as audio plug-ins).

Audio software may also be interactive by allowing the score to become stream-based, receiving new notes and parameter control events incrementally while it runs. For example, the widely-supported MIDI protocol is explicitly oriented to this task. Today implementations of CSound, a computer music framework strongly inspired by Music-N, also support streaming events in this fashion. Other systems generalized the score into a full-fledged textual programming language, capable of invoking instruments according to arbitrary algorithms, such as *SuperCollider 3* (McCartney, 2002).

Some software also extend run-time reconfigurability to the orchestra by implementing unit generators as pre-compiled modules which can be dynamically created and connected, as noted in Section 2.3.1. Internally the 'graph' of unit generators is either traversed as a data-structure at each callback, or pre-serialized into a list to be iterated over. In systems such as Max, the notion of orchestra (as pre-generated program) and score (as pre-written data-structure) have been effectively replaced by an entirely interactive, stream- and message-based visual programming environment (Puckette, 2004). However implementing unit generators as dynamically connected libraries of pre-compiled code propagates the buffering constraint to the connections between unit generators: "In the limit, this results in an interpreter with vector primitives. There are three problems with this alternative: accessing the buffers consumes memory bandwidth; the latency is increased because the first result is not ready until an entire buffer has been processed; and handling dynamic, sample-dependent control-flow becomes problematic." (Draves, 1997)

4.1.2 Code generation: a return to *Music III?*

The demonstrations of *audio.Def* and *gen*~ presented later in this Chapter present a return to the code-generation approach of *Music III*. Rather than treating the orchestra as a structure of dynamic connections between pre-compiled objects, we can consider it as the specification of a program to be generated on-the-fly. The traversal of an orchestra specification is computed once while generating the code, rather than continuously as the program executes.

Nevertheless, as noted in Chapter 3, since compilation can have significant overhead, the choice of how to fragment the program, and when to compile, is critical. The program must be divided into:

1. Those parts which change rapidly (not suitable for compilation),
2. Those parts which change infrequently (suitable for dynamic compilation),
3. Those parts with never change (suitable for static compilation).

Within a *Music-N* paradigm, updated for real-time performance, the score player (audio driver and scheduler engine) is largely invariant and can be statically compiled, the instrument specifications can change but not usually very frequently, while the note (or 'voice') events and parameters may change rapidly (particularly for granular synthesis).

Exactly where to divide signal processing networks into distinct 'instruments' cannot be implicitly predicted, so this decision can be granted to the user of the system. As a heuristic, an instrument would be that part of the audio processing in which all unit generators are created at the same time, and share the same duration. The same instrument structure may be used repeatedly but instantiated frequently and concurrently, such as the voices of a polyphonic synthesizer or the grains of a granular synthesizer. Voices that have different data but the same procedural structure can share the same machine code.[2]

It is relatively easy to modulate voice parameters without needing to recompile code, using dynamic parameters (or 'ports'). Many such parametric controls can remain fully dynamic in the generated code. With some effort these can also accommodate both numeric and signal-based connections. With signal-based ports, idioms of dynamically routable mixing busses can be accommodated. Although this re-introduces some of the disadvantages of the modular strategy, the control of its boundary is in the hands of the user rather than pre-defined.

The basic unit of compilation thus corresponds to the *Music-III* instrument, with internal initialization and signal processing functions, but which implements a buffered interface for real-time input/output signals, with configuration parameters and dynamic ports. New instruments can be created while the program runs, and the entire audio scene at any time is a collection of voices: distinctly parameterized instances of the currently available orchestra of instruments, connected by signal busses and driven by the static audio scheduler engine. User interaction can thus define new instruments, create voices from them, and modify the parameters and interconnections of voices.

Implementations must still handle the threading concerns of real-time audio. In particular, compilation time cannot be guaranteed, and therefore should not occur in the same thread as signal processing. Some degree of inter-thread communication and synchronization is necessary. Implementations must also supply the ability for generated code to bind with the host environment, including registration and deallocation of new voices, registering port interfaces for dynamic

[2]Recompiling the entire synthesis graph for every micro-sonic grain, where the grains vary only in numeric parameters, would be extremely wasteful.

configuration, and access to globals and shared resources (such as wavetables). To a certain extent these features are part of the statically compiled host, and to a certain extent they must be part of the code generation template(s).

4.2 LuaAV and the *audio.Def* module

Before detailing the use of dynamic compilation in LuaAV, it is pertinent to describe some of the properties of LuaAV system in general, and its audio engine in particular.

4.2.1 LuaAV

LuaAV is a platform for audio-visual composition, live coding and research with which users can author and execute scripts. The LuaAV application provides an extension of the standard Lua virtual machine (Ierusalimschy et al., 2005) incorporating a deterministic timing scheduler for Lua and C functions, and many extensions for audio synthesis, windowing and events, OpenGL graphics, MIDI, OSC and so on. User scripts make use of the grammar and vocabulary of the Lua language[3] along with these extensions. LuaAV is principally developed by the author and Wesley Smith (Wakefield et al., 2010).[4] LuaAV also incorporates the *luaclang* module detailed in section 3.2.2.

[3]The Lua programming language (Ierusalimschy et al., 2007) is a predominantly imperative language with strong functional capabilities, including first-class functions and coroutines (one-shot continuations). It was designed as an 'extensible extension language': easy to learn, easy to embed, and easy to extend for domain-specific tasks (Ierusalimschy et al., 1996). It also supports powerful textual string processing through the LPEG parsing expression grammar module (Ierusalimschy, 2009b).

[4]The author was largely responsible for the scheduler, all aspects of the audio engine, and several other modules.

Precursor & motivation

With Vessel, a precursor to LuaAV, the author's original intention was to reduce the barriers between the score and unit generator layers by presenting both in a common language, and creating a scheduler which avoids block-based limitations on the creation and connection of unit generators (Wakefield, 2007). Unit generators could be created, parameterized, destroyed, connected and disconnected at arbitrary sample-accurate times, fully interleaving the underlying machine-code synthesis routines with any other algorithmic and temporal processing in the score-level Lua language. However the resultant performance of this modular approach was poor for complex synthesis algorithms, and there was no way to define new unit generators. The LuaAV *audio.Def* module described below was designed to resolve these problems by introducing an instrument abstraction closer to the *Music III* macro concept.

Strongly timed

LuaAV inherits the 'strongly timed' language feature from Vessel. In LuaAV, every script has a master scheduler attached to the top-level main coroutine, driven by a microsecond timer. Changes to audio processing are synchronized to the timer with sample accuracy, supporting a wide variety of micro-sonic and granular approaches that are not feasible in most related systems.

Temporal programming is achieved by scheduling new coroutines using **go()** (see Listing 4.1). Scheduled coroutines can yield at any point in the imperative code using **wait()**, to be resumed by either synchronous or asynchronous events. Synchronous events are given to **wait()** as durations for the scheduler, while asynchronous events are specified using arbitrary string names, triggered using

Listing 4.1: Basic examples of temporal and event-based scheduling in *LuaAV*.

```lua
-- define a function to print "tick", "tock"
-- alternately every half of a second:
function clock(name)
    -- repeat forever:
    while true do
        print(name)
        -- wait for 1 second:
        wait(1)
        -- trigger any events waiting on "alarm":
        if random(30) == 1 then
            event("alarm")
        end
    end
end

-- start a coroutine using it immediately:
go(clock, "tick")

-- start another coroutine half a second later:
go(0.5, clock, "tock")

-- some time later also print "SURPRISE!"
go("alarm", function()
    print("SURPRISE!")
end)
```

the **event()** call. This allows a tightly temporal form of programming which can interleave both visual and sonic synthesis.

Audio engine and timing

The LuaAV audio engine runs in a separate high-priority thread and avoids unbounded calls to maintain real-time deadlines and avoid audio drop-outs. Communication between the main Lua thread and the audio thread is achieved through time-stamped first-in, first-out (FIFO) message queues. These queues are lock-free to make a fine granularity of messaging viable.

LuaAV uses a latency window between Lua and the audio engine, which supports deterministic timing so long as the main thread can process events within this window (the failure scenario is delayed events, but ordering of events is preserved). The window is initially configured at 50ms, but can be modified as desired.

Each active script has a corresponding *Context* object in the audio engine, triggered by callbacks from the audio device driver. The *Context* contains a list of voices. Adding and removing voices to the *Context* list follows a generic plug-in interface with data pointer, synthesis and de-allocation functions. The engine ensures that addition and removal of voices is sample-accurate to the event timestamp.

4.2.2 The *audio.Def* module

The *audio.Def* module extends LuaAV to support the creation of arbitrary new and efficient audio synthesis nodes at run-time, by means of dynamic compilation, directly within a Lua script.

The use of *audio.Def* generally follows the following stages (a brief example is given in Listing 4.2):

1. Use the **Def{}** function to define an *instrument* constructor. The definition must include a synthesis expression, constructed from built-in expression generators. Expressions can be nested, and can also include constants, infix operators, named ports, and multi-channel lists. The definition may also include dynamic port specifications and feedback loop assignments.

2. Call this instrument constructor to create active *voices* (generating sounds). Arguments to the constructor can also set input and output port assignments

Listing 4.2: A brief example of constructing and instantiating a dynamically compiled instrument in *LuaAV*. This example shows a basic frequency modulation instrument, compiled by the **Def{}** call, and demonstrates creating and modifying a voice from it.

```
-- build an instrument:
fm = Def{
    -- set port defaults:
    car = 440,
    mod = 10,
    depth = 20,
    -- define the synthesis expression:
    SinOsc{
        P"car" + SinOsc{ P"mod" } * P"depth"
    } * 0.1
}

-- create a voice:
voice = fm{ car=500, depth=0 }

-- repeat 10 times:
for i = 1, 10 do
    -- modify port value:
    voice.depth = i * 50
    -- continue for 0.25 seconds:
    wait(0.25)
end

-- stop the voice playing:
voice:stop()
```

(with constants or signal busses). Many different voices can be created from the same instrument, and their lifetimes can overlap arbitrarily.

3. Call methods on the voice objects to change their port assignments.

4. Stop an active voice (either explicitly via the **stop()** method, or automatically via envelope durations or garbage collection).

The *audio.Def* module is itself written in Lua. It depends on the *luaclang* module to compile the generated C++ code, and the LuaAV *audio* module to access properties of the audio engine, such as the sample-rate.

Creating synthesis expressions

New synthesis routines are specified by declarative constructions of compound expression generators. The *audio.Def* module includes a set of pre-defined expression generators with which a synthesis process can be defined, such as common audio waveform generators, filters and delays.

Expression generators appear similar to the concept of unit generators used in related software, however they do not encapsulate state as *objects*, rather they are purely syntactic, declarative *specifications* using the standard Lua data structure (a *table,* a combination of a hash map and array). Similarly, despite the appearance of computation in an expression such as **Saw{ 2 } * Square { 3 }**, no multiplication is carried out: instead the operator is overloaded to generate a compound expression object (see Listing 4.3). The *audio.Def* expression interface is a human-friendly language which immediately constructs a machine-friendly representation.

The arguments to an expression generator specify the inputs, which can be numeric literals (constants), strings (for named arguments such as *Ports*), other

Listing 4.3: Expression generators in LuaAV *audio.Def* are simply a syntactic sugar for the construction of standard Lua data structures. Operators are overloaded as another syntactic sugar for expression generators.

```
-- this construction:
Saw{ 2 } * Square { 3 }

-- is equivalent to this Lua data stucture:
{
    ["op"] = "Mul",
    [1] = {
        ["op"] = "Saw",
        [1] = 2
    },
    [2] = {
        ["op"] = "Square",
        [1] = 3
    }
}
```

expression objects, or lists of these (for multi-channel inputs). Expressions are thus nested in a structure determined at construction time. Any inputs that may need to be dynamically varied must use explicit named *Port* objects, using the **P** constructor.

Since expressions are simply data declarations, it is easy to create new, re-usable generators and expression fragments from the built-in set. Furthermore, it is also easy to generate new compound expressions programmatically, and subject them to further manipulations, such as genetic algorithms.

4.2.3 Compilation via the Def call

These expressions are converted into machine code by means of the **Def{}** function. Along with the expression, additional meta-data can be supplied to specify additional contextual information, such as variable types and default values.

Def{} performs several important operations, finally returning a synthesis voice constructor function (see Figure 4.2.1). This call is the heart of the *audio.Def* system.

Before beginning the parsing process, **Def{}** constructs a container *prototype* (a Lua table) to collect information during parsing and code generation. This prototype is gradually annotated with template definition dependencies, persistent state variables and names, basic blocks[5] containing the actual synthesis algorithm, and port specifications for the voice interface.

Parsing expressions

The tree of expression structures in the **Def{}** argument is traversed in depth-first order. Depth-first traversal ensures that expression inputs have valid annotations, including data type, channel-count, update rate and scope. Memoization is used to prevent multiple visits of the same sub-expression, ensure uniqueness in returned values, and handle feedback situations. String nodes are treated as *prototype* member or global name lookups, and numbers are passed through as constants.

For each expression type (as identified by the "op" field), *audio.Def* provides a corresponding parser function.[6] Each specialized parser function generally performs several of the following actions:

1. Aggregate any necessary dependencies to the *prototype,* while avoiding duplicates. Dependencies may include "extern" declarations for global symbols,

[5]A basic block is a node within the control-flow graph, with one entry point and one exit point, and between which all instructions necessarily operate in sequence. The strict definition makes them highly amenable to code transformation and analysis.

[6]Any tables with no "op" defined are understood as multi-channel arguments. This is a concise lexical convention inspired by a similar convention in SuperCollider. Any argument to a SuperCollider ugen can be specified as a list, e.g. *{ SinOsc.ar(500) }.play* is single channel, while { *SinOsc.ar([499,600]) }.play* creates two channels of output, because of the list in the first argument of *SinOsc.ar().* Similarly in LuaAV, *SinOsc{ {499, 600} }* creates a two-channel expression.

and fragments of code for type definitions (similar to the role of header file includes in a static compilation workflow).

2. Aggregate any necessary persistent state variables (such as the phase needed for oscillators, the previous input/output values needed for filters, or more complex objects declared in pre-defined headers) to the list of *prototype* members. Persistent variables are given unique names[7], and may also need initializer and destructor code fragments added to corresponding basic blocks of the *prototype*.

3. Aggregate statement objects to the basic block containers of the *prototype*. The choice of basic block, which statements to add and how many, may vary according to the number of inputs, and their type annotations. Some of these statement generation procedures are generic and used by several expression parser functions. The statement objects are also Lua tables, tagged with one of a small set (currently 21) of low-level operation names, the appropriate argument names (or numbers), and a uniquely generated output name.[8]

4. The names and type annotations of the statement(s) corresponding to the expression output(s) are returned by the expression parser function, to be used within the containing expression.

Expression parsing is thus a translation of a source language (trees of expression objects) into a target language (basic blocks of statement objects).

[7]A global **gensym()** function generates unique variable names for each invocation of **Def{}**.

[8]This is largely equivalent to a byte-code (with unlimited registers) stored as a data structure (rather than as text).

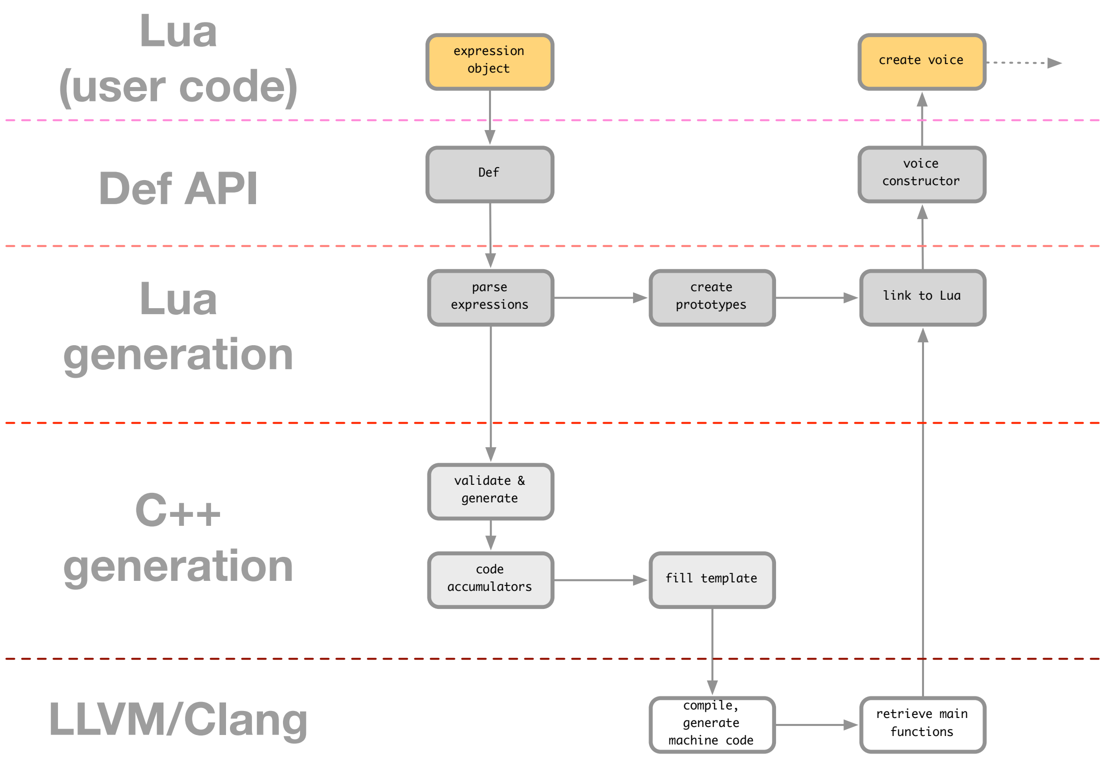


Figure 4.2.1: An overview of the workflow invoked by calling **Def()** in LuaAV. The orange areas are user-level input and output. A synthesis expression moves through several layers of language and representation to reach machine code, which is returned wrapped within higher layers of interface for the benefit of the user.

Generating code

The *audio.Def* module generates C/C++ code at the next stage of compilation. Generating C/C++ is advantageous in several respects: ease of binding with Lua (via the Lua C API), ease of binding with LuaAV's audio engine (via a C++ API), ease of templating (due to some flexibilities of the C++ language) and ease of debugging due to readability.

Code generation is achieved by populating a tree of code accumulators. An accumulator object contains a list to which strings, templates and other accumulators can be added. An accumulator can be serialized into a single string (recursively serializing any embedded accumulators).

The macro-structure of the generated code is specified by a master template, which is a top-level accumulator with nested string and accumulator sections. Each basic block and function in the template has a corresponding accumulator, including the audio synthesis routine (with separate control- and audio-rate accumulators), constructor and destructor, and the *define* and *create* functions to bind the generated machine code to Lua. Additional accumulators may be created as needed for subroutines, conditional blocks.

The *prototype*'s set of dependencies identified during parsing is iterated, appending appropriate 'header' code into the first accumulator. Next, the stateful members in the *prototype* are iterated, and C++ declarations appended to the struct definition, constructor, and destructor according to the annotations found. Setter / getter methods may also be generated and added to the root template, and code added to *define* for registration with Lua.

Finally the statement blocks are transformed into strings of actual code according to a set of C++ code generator functions corresponding to the statement tags. Which accumulator a generated statement is appended to depends on the scope and rate of its inputs and other annotations. This, for example, ensures that operations with only control-rate inputs are performed outside the sample loop, unless the operation itself is marked as audio-rate (such as a signal filter).

Translating to machine code

Only after these processes run are the accumulators serialized, resulting in a single string of C++ code. This code is further processed by Clang/LLVM (via the Lua-bound *luaclang* module) for these purposes:

1. To translate the C++ code into an LLVM module (or signal errors for invalid code).

2. To apply a series of fast optimizations on the LLVM module.

3. To translate the LLVM module into x86 machine code.

4. To retrieve function pointers from the machine code (and later to deallocate them).

These processes are somewhat standardized with Clang/LLVM. Care was taken however to only apply the optimizations strictly necessary (much of the generated code is already optimized), and to minimize external dependencies to avoid delaying the **Def{}** result.

If machine code compilation was successful, the address of the function named *define* is retrieved from the generated machine code, and made available to Lua. This function is an *"extern"* C function, conforming to the C ABI, and its type signature also conforms to the Lua C API for Lua functions, and thus *define* is directly callable from within Lua code.

Binding to Lua

As the next stage in **Def{}**, the *define* function is immediately called, passing in the current audio *Context* and the *prototype* generated earlier. The *define* function creates a new class-type (or "metatable") in Lua to be used by the instrument voices, including the binding of the standard and custom methods, setters and getters.

The *define* function returns the *create* constructor (another Lua-typed C function) as a closure, binding it with several upvalues including the audio *Context* object and the voice *prototype*.[9] The *create* function also retains an upvalue reference to the LLVM JIT object (containing the generated machine code) from which it was created. Values in Lua which are upvalues of a closure cannot be garbage collected before the closure is garbage collected, which in this case ensures that the generated machine code remains in memory for at least as long as the constructor function is accessible in Lua.

If any errors occurred during the whole **Def{}** process, they are propagated as Lua errors to the call-site. Otherwise, the *create* voice constructor function is returned, where it can be invoked in user-level code.

4.2.4 The voice abstraction & implementation

Voices in *audio.Def* are an abstraction of two objects:

1. An instance of the *Synth* class, compatible with the LuaAV audio engine, which utilizes the synthesis machine code in the audio thread.

[9]A closure is a function with reference to an environment in which any variables used not local defined in the function are to be found. Such variables are upvalues, which is to say, upvalues are references in the body of a function to mutable variables in an outer environment of the function. Calling a closure with upvalues may modify the values of these referenced upvalues in this outer environment.

2. An object in the Lua language that acts as a proxy to the *Synth* instance.[10]

Constructing voices

The C++ implementation of the voice constructor allocates both *Synth* instance and voice proxy. The machine code to initialize the Synth instance is invoked, and the voice proxy behavior is set by applying the instrument metatable created by *define*.

The voice constructor takes a table (dictionary) of arguments to initialize the ports of the *Synth* instance and voice proxy. Each port in the instrument is initialized with the corresponding constructor argument, or default value from the *prototype* if the argument was not given (see Listing 4.4).

Finally the constructor sends a message to the audio thread to add the *Synth* to the audio engine *Context,* and increments a reference count to the JIT object from which it was created. This reference count ensures that the machine code remains in memory for as long as any voice is used in the audio engine.

Listing 4.4: Constructing a voice from the definition in Listing 4.2. Since the "depth" port is not given as an argument, it will be initialized to the default specified in the **Def{}** prototype.

```
local myvoice = fm{ car=800, mod=10 }
```

Methods and properties of voices

The voice object acts as a proxy to the *Synth* instance in the audio engine *Context.* Use of a proxy object is necessary in this case to maintain deterministic timing of state changes for processes occurring in a separate thread. Attempts to change properties of the voice are redirected as messages to to the machine-code generated

[10]Implementing the *Proxy* software design pattern (Gamma et al., 1993). A proxy object operates as an interface to some other object, mediating messages between sender and proxied object in some way.

methods of the *Synth* instance, but these messages are time-stamped for temporal accuracy. Calls to property setters also store the new value within the proxy object.[11] This duplicate storage is used by getters for immediate access, and also serves a referencing role to prevent garbage collection.

For a friendly interface, the voice proxy metatable also supports a property indexing style. For example, the code in Listing 4.5 modifies the modulation frequency of the voice *myvoice* every 0.1 seconds by assigning to the *mod* property. Behind the scenes, the setter method is being invoked and messages are being sent to the *Synth* instance in the audio thread.

Listing 4.5: Invoking a setter by property assignment (to the "mod" port) on a voice, and invoking a getter by property access (a continuation of the code in Listing 4.4).

```
while now() <= 2 do
    wait(0.1)
    myvoice.mod = now() * 10
end
print(myvoice.mod)  --> prints 200
```

Multi-channel ports also accept assignments from lists. Any list entries that are missing (or have a value of **nil**) will not modify the corresponding port-channel value, to support independent assuagement to individual channels.

Stopping and destroying voices

Voices may be terminated in four different ways:

1. Self-termination within the audio signal processing thread. For example, use of the Envelope generator adds a terminating condition to a synthesis code, to destroy itself when the envelope completes. The voice proxy may

[11]Implemented using Lua userdata environments.

remain in Lua until garbage collected, but no longer forwards messages to the audio engine.

2. Explicit termination by the user through calling the **stop()** method on the voice proxy. This is not immediate due to the thread boundary, but is forwarded to the audio engine using the standard timestamped message queue with accurate logical timing, and behaves sample accurately.

3. Garbage collection: when a voice proxy is no longer reachable from Lua, the garbage collection handler can call the **stop()** method on it implicitly. Exactly when a voice is garbage collected is indeterminate however. Note that any generator with a self-terminating condition in the synthesis code (as above) is not garbage collected in this way.

4. All voices are terminated at the destruction of the running script in which they were launched, as part of general tear-down procedures.

Memory management

Special care has to be taken with the memory management of the voice instances themselves. Since they are used in both the main Lua thread and the audio signal processing thread, we cannot simply rely on Lua's existing garbage collector. A voice's *Synth* instance may still be synthesizing audio for a short time after it has become collectible from Lua. The solution employed uses a time-stamped message queue in both directions, to ensure safe release of memory only after a voice is no longer used for synthesis.

Deallocation of the object itself does not occur until *both* the instance has been terminated in the audio engine, *and* the voice proxy has been garbage collected in the Lua virtual machine. At this point, the generated machine-code destructor is invoked, releasing any memory allocated by the voice, and removing references it has, including its reference to the JIT object.

Each instrument has a corresponding JIT object retaining the generated machine code. A separate process sweeps the list of JIT modules to release memory associated with the JIT-compiled code when no longer in use. If a JIT object no longer has references (i.e., it are no active voice proxies in Lua, no active instances in the audio engine, and the *create* constructor is no longer reachable in user code), it releases the memory of the dynamically compiled machine code functions and globals, and then deallocates itself.

4.3 Max/MSP/Jitter and *gen~*

The Max family, including Max/MSP/Jitter (Zicarelli, 2002), PureData (Puckette, 1996) and several others, share a common root (the original Patcher program), and purposefully lean toward open-ended creativity support rather than productivity support: "the design of Max goes to great lengths to avoid imposing stylistic bias on the musician's output" (Puckette, 2002).

Max began its life in the late 1980's as a visual programming environment for computer music composers, allowing the connection of processing modules through a flexible, asynchronous data-flow paradigm. The open-source re-implementation, called Pure Data (PD), was released in 1996. The MSP extension was added to Max in 1997, adding support for synchronous data-flow for audio signal processing on the CPU (a forerunner, called FTS, supported audio signal processing on separate dedicated hardware). In 2003 Max added the Jitter extension, supporting processing of arbitrary 2D and 3D matrices, video, and 3D graphics. The Gen extension to Max, partly detailed in this section, introduced code generation and run-time compilation to Max in 2011.[12]

The Max family of tools quite explicitly show the compromise taken between black-box modules combined in run-time determined ways, as described in Section 2.3.1. The behavior of each box is specified by its first argument (the box class name), which identifies the algorithms it supports. The connections between boxes are represented and edited visually as lines, known as *patch-cords,* by analogy to the patch cables connected analogue modular synthesizers. Artists and composers work with Max by creating and configuring boxes, and connecting and disconnecting lines, to gradually construct a program (or *patch*) while it continues to run. The Max program effectively interprets this developing program by following the data-flow connections between the boxes, and passing control to the pre-compiled class implementations as needed to compute the effects of each box.

Each class of box is implemented in a chunk of pre-compiled machine code originally written in C, which is either built into the application itself or dynamically loaded and linked. The class includes a definition of the boxes' initialization, inlet and outlet types, the set of events it will respond to, along with the implementations of the box state and event handlers. The boxes themselves are written to a public application programming interface (API), allowing third party authors to extend the capabilities of Max with new code in C or other compatible languages. This extensibility may partly underlie the success of this tool in the wider community. Over its 20 year history Max has aggregated more than 4000 user-contributed objects.[13] However, as much as this attests to the wide use

[12]The Gen extension has been primarily developed by the author, Wesley Smith, Joshua Kit Clayton and David Zicarelli. The author has been responsible for the implementation of the gen~ object, the gen~ specific operators, code generation for gen~, and has contributed significantly to the parsing and compilation processes of the shared Gen system.

[13]As listed at the *Max Objects Database,* a website dedicated to documenting the universe of Max/MSP/Jitter objects: http://www.maxobjects.com/ (accessed 2012).

of this tool in the community, it is also reflects the problems associated with a black-box modular strategy: a significant proportion of user-contributed objects perform tasks that could be achieved using combinations of built-in objects, but with less optimal performance.

4.3.1 The *Gen* patcher

Gen extends the Max/MSP programming environment with a new kind of patcher (the *Gen patcher*), whose contents are translated dynamically into machine code each time the user makes an edit, rather than being continuously interpreted.[14] The Gen patcher can be used to program routines for different domains, including audio signal processing (gen~) as well as CPU and GPU matrix and image processing (jit.gen, jit.pix and jit.gl.pix). Each domain adds a corresponding collection of specific operators to the general set. By hosting a Gen patcher inside, the gen~, jit.gen etc. objects act as an interface between the modularly interpreted Max/MSP world and the dynamically compiled Gen world. Specifically, the Gen patcher inside a gen~ object is translated into efficient audio signal processing code and embedded into the parent Max/MSP patcher, as if the inlets and outlets of the gen~ object are directly connected to the inputs and outputs of its embedded Gen patcher.

The Gen patcher represents synchronous data-flow in a similar fashion to patching with MSP objects: rather than triggering each box's computation explicitly only when data is received at a specific 'hot' inlet, each box represents a process occurring at a global signal rate (unless provably otherwise), so there is

[14]Code generation from visual programming has been widely used previously, such as the DRAKON (ДРАКОН) system developed for the Buran space program between 1986 and 1996, which featured code generation for Java, C#, C/C++, Python, Tcl, Javascript and Erlang. http://drakon-editor.sourceforge.net/generation.html (accessed 2012).

no ambiguity of inlet/outlet ordering, and multiple inlet connections are summed. Since there is no asynchronous messaging, the interface can also take advantage of many compile-time decisions. In particular, since arguments and attributes of a box are set only at compile-time, they can include arbitrary constant expressions, and the functional implementation and box interface can be determined accordingly.

Gen also supports a mixture of visual and textual programming by using the *expr* and *codebox* objects, which allow arbitrary chunks of textual code in the *GenExpr* language. GenExpr is a C-like language with dynamic typing and local-by-default lexical scoping rules. GenExpr supports most of the same operators as the visual patching interface, along with features such as conditional branches (if), loops (for, while) and subroutines, supporting concepts that are very difficult to express in a visual data-flow programming environment (Green and Petre, 1996). All visual Gen patchers are also displayed in GenExpr equivalent code visible in a side-pane of the Gen patcher window.

4.3.2 The *gen~* domain

The gen~ domain targets single-sample processing of 64-bit audio signals. It adds a variety of audio-specific operators to the basic set of Gen operators, including waveform generators and filters, audio conversion utilities, delay lines, a single-sample feedback operator, and a variety of operators for wavetable / sample buffer writing and playback. Some of these operators closely follow familiar objects of Max/MSP, while others are unique to capabilities of gen~.

The support for single-sample feedback in particular allows for several interesting new capabilities for audio processing that would previously have demanded writing C objects, including arbitrary new filter designs, physical models, iterative map oscillators, waveset manipulations, and more (this is further elaborated in Section 6.1.2). It can also be used for Fourier-domain spectral processing when a gen~ is hosted within an MSP pfft~ object.

4.3.3 Workflow

The full workflow of a Gen patcher outlined in this section is usually invoked at each non-trivial edit, such as the creation and deletion of gen box objects, the creation and deletion of patch-cords connecting them, or the modification of text with a codebox object.[15]

The workflow is similar to, yet significantly more rich and complex than, the demonstration in Section 4.2. It proceeds by a series of translations between different representations of the desired algorithm, to finally reach machine code, which is then encapsulated within a dynamic binding proxy layer in the gen~ object (see Figure 4.3.1).

The workflow is largely general to all Gen host objects, but configured for each domain according to a *domain* object. The domain adds definitions of the output template(s), type hierarchy and behaviors, domain-specific constants, and so on, along with the dictionary of available operators. The system refers to this library of domain operator definitions (OpDefs) to potentially specialize many stages of the workflow. In a sense, the OpDefs form the large instruction-sets of the intermediate representations in the workflow. Nevertheless, many of the OpDefs share implementation code to promote consistency.

[15]Auto-compilation can be disabled if desired, and invoked manually instead.

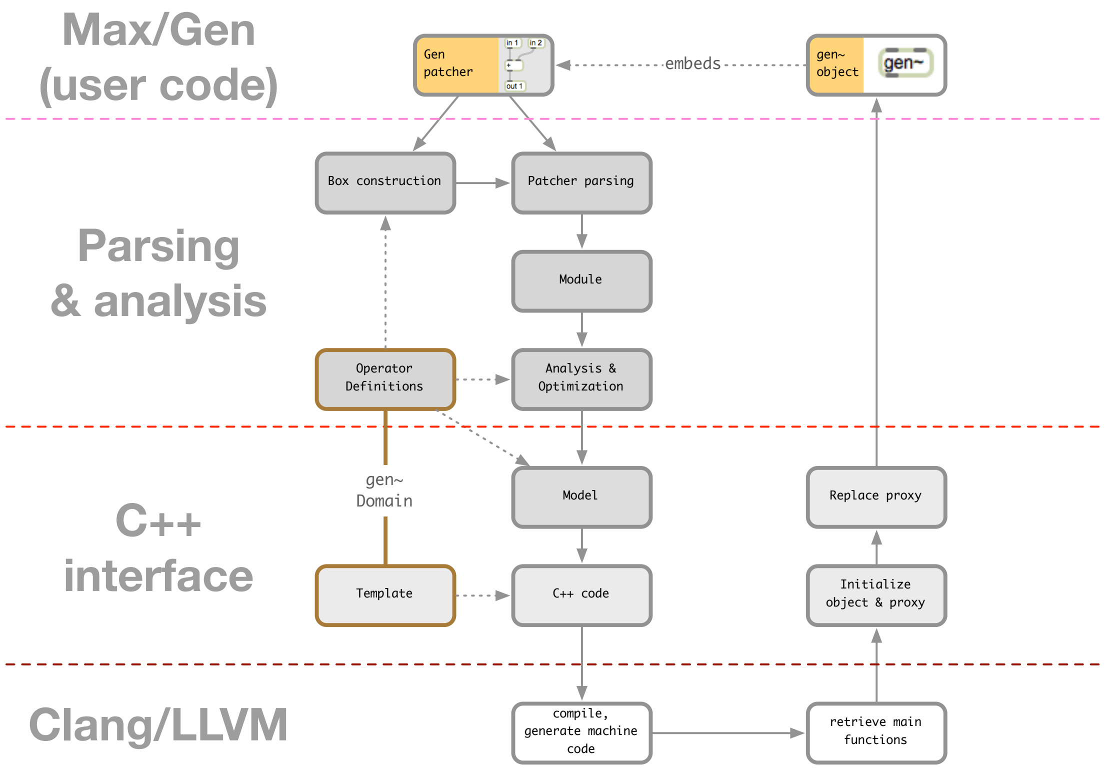


Figure 4.3.1: An overview of the workflow invoked when editing a Gen patcher embedded in a gen~ object. The orange areas are user-level input and output. An algorithm moves through several representations to reach machine code, which is returned wrapped within a proxy layer for dynamic swapping in the host gen~.

Parsing and analysis

Parsing a visual diagram presents several differences from a textual source. A textual source is essentially one-dimensional, and uses the strict ordering to carry semantic meaning (e.g. the order of statements within a function relates to the order in which they are applied). A visual source does not present a single way to be read or stored in memory however. The ordering of boxes and patch-cords in the in-memory structures may not carry important semantic meaning. Secondly, the abstract syntax of a textual program has a tree structure (due to the tree-like grammar), whereas the abstract structure of a visual programming diagram is a directed graph (which may contain cycles). The order of operations of the visual programming diagram must be derived during parsing.

Nevertheless, an important goal of Gen is that any algorithm expressible as an arrangement of visual boxes and patch cords can also be expressed using GenExpr textual code. Furthermore, a visual Gen patcher can contain fragments of GenExpr code within it, which must be parsed at the same time as the visual components. The Gen parser therefore uses a common intermediate representation, called *Module,* as the target language of parsing both visual patchers and GenExpr code. This intermediate representation closely resembles the abstract syntax of GenExpr, and is thus tree-like.

Parsing textual code to *Module* uses a typical workflow built upon a definition of the GenExpr language in the form of a Parsing Expression Grammar (Ford, 2004). Parsing visual patchers however requires several passes. Firstly, whenever a box is created or edited, it is partially parsed immediately. The first word in a box identifies the operator name, which is used to look up the corresponding OpDef in the domain dictionary. The OpDef's box constructor method specifies the inlets & outlets of a box (which may vary according to given arguments), and the stores arguments & attributes in memory. It also identifies illegal arguments and attributes, signaling an error to the Max user interface accordingly.

Then, at each significant edit, the following stages occur:

1. A *Module* structure and a list of inlet connections are initialized.

2. All patch-cords are visited, adding the source/target details to the list of inlet connections.

3. This list is used to sort the boxes by data-flow dependency. Sorting begins at patcher outlets and other zero-outlet objects (such as buffer writers). Special feedback connections (history and delay inputs) are handled last. Any illegal feedback connections signal an error to the Max user interface.

4. The sorted list of boxes is iterated. Inlet connections are resolved to singular values: inlets with no connections utilize a default value instead (specified by the OpDef), and inlets with multiple connections are converted to addition operators over these connections. Another specialized method in the OpDef is used to parse the box inputs, arguments and attributes and insert abstract declarations, statements and expressions into the *Module* accordingly. Boxes containing GenExpr code have their internal *Module* merged into the main *Module*.

The *Module* is sufficient to reconstruct the concrete syntax of GenExpr, but does not explicitly represent much of the semantic information useful for code generation. An *Analysis* object is created to store additional semantic annotations regarding the *Module* (including symbol resolution, data-flow representation, and type-inference), and perform several optimizations prior to code generation (including constant folding and dead-code elimination). These are somewhat standard processes of static analysis and optimization, and will not be detailed further, however it is worth noting that several of these processes are informed by operator-specific specializations in the OpDefs.

All results of parsing retain a reference to the source from which they were derived, whether a box in a visual patch, or a fragment within textual code. This source reference is used if an error is detected during parsing, analysis or optimization, to locate the displayed error message to the user.

Code generation and dynamic compilation

Code generation is template-driven at the macro-scale but model-driven at the micro-scale (as described in Section 3.2.3). In accordance with the model-view separation recommendations presented in (Parr, 2004), computations over source data and interpolation into the template are performed in separate passes. Specifically, the source *Analysis* object is traversed by a *Visitor-pattern* to create a *Model* dictionary data structure specific to the gen~ template.

The gen~ template is expressed as a grammar-like tree structure of template-rules, with the main template as the top-level, starting rule. Each template-rule is a string containing raw output code interspersed with both invocations of other template-rules and macro statements to be replaced by specific values in the data model. The template engine supports conditional and looping constructs, such that template-rule(s) invoked can be dependent upon values in the *Model*. Higher-level template-rules are specialized to the gen~ domain, while lower-level and terminal rules specify how to generate common C++ expressions, statements, struct declarations and so on.

Applying the *Model* to the template results in a string of C++ code. The generated C++ code is translated to LLVM intermediate representation by Clang, upon which several further optimizations are applied, and the result translated to machine code by LLVM.

Interfacing the machine code with the gen~ object

The gen~ object internally provides a Proxy-pattern plug-in architecture, which allows proxied implementations to be swapped at run-time. Specific events received by the gen~ object, including requests to synthesize audio samples, are delegated to the hosted proxy object. The proxy is a simple data structure specification, with fields for each of the event handler functions, and other fields for metadata such as the number of inputs and outputs and the names and properties of parameters.

When a new compilation is completed, one of the generated machine code functions is immediately called to allocate and initialize an object whose specification is defined in the generated code. This object incorporates all the stateful components of the user-defined algorithm, such as history values, oscillator phases, delay memory and so on. The object also includes a section which conforms to the proxy specification.

This new proxy section is sent to the host gen~ object as a notification to replace the existing proxy. When installing a new proxy, the gen~ object uses the proxy metadata to dynamically modify the number of inlets and outlets, and install dynamic attributes for parameters as needed. While running, messages to update attributes of the gen~ object are forwarded to the corresponding generated function in the proxy, which in turn updates the internal state of the generated object.

The proxy structure also keeps a reference to the generated machine code memory, so that this memory can be freed when the proxy is no longer used.

4.4 Related Work

CSound is perhaps the closest widely-used descendant of Music-N. Today CSound is also capable of running in real-time, receiving note events dynamically; but still supports a non-real time mode in which scores utilize a form of code generation. However in the current implementation CSound creates instruments as objects in memory referring to pre-compiled unit generators.

SAOL (Structured Audio Orchestra Language) embedded in the MPEG-4 specification is also strongly influenced by Music-N and CSound (Scheirer and Vercoe, 1999). The specification of SAOL is that instrument descriptions generate C code, which must be compiled and loaded by implementations. Scheirer notes the importance of distinguishing languages from implementations; SAOL is a specification of a language that fully compliant MPEG-4 systems must implement. Demonstrations of this independence are given in Section 6.2.2. Both CSound and SAOL maintain a strong distinction between control- and signal-rate operations in the instrument specification.

FAUST is another language specification for audio synthesis, however unlike SAOL it uses terse and highly functional idioms (Orlarey et al., 2004). A goal of FAUST is to support the generation of C/C++ code for multiple host (native applications, Max/MSP, PD, SuperCollider, CSound, etc.). Although there has been interest in invoking FAUST dynamically during a real-time performance, this has not yet been implemented.[16] However *Pd-faust* is a dynamic environment for

[16]Through personal communication with Yann Orlarey (2011), there are two issues to over-

running FAUST code in PD, implemented in Pure by Albert Gräf. With *Pd-faust*, new instruments can be loaded at run-time by invoking the faust compiler implicitly in a separate process.[17] An example of connecting a FAUST language parser to *gen*~ is detailed in Section 6.2.2.

Several systems have generated code for dedicated external hardware, especially when desktop computing was not yet powerful enough for serious real-time audio synthesis. For example, the Kyma system of 1988 dynamically compiled sound-generating structures to microcode for a dedicated DSP processing unit (Scaletti and Johnson, 1988). Today this approach is more prevalent in the graphics domain (compiling higher level languages such as OpenCL (Stone et al., 2010) for dedicated GPU devices).

More than a decade ago, C code generation, compilation and loading of audio instruments at run-time was being explored to solve the efficiency demands of audio in the interpreted Squeak Smalltalk virtual machine: "As the performance of the Squeak system improved, we were delighted to find that we could actually synthesize and mix several voices of music in real time using simple wave table and FM algorithms written entirely in Smalltalk. Nonetheless, these algorithms are compute-intensive, and we used this application as an opportunity to experiment with using C translation to improve the performance of isolated, time-critical methods. Sound synthesis is an ideal application for this... We observed nearly a 40-fold increase in performance: from 3 voices sampled at 8 KHz, we jumped to over 20 voices sampled at 44 KHz" (Ingalls et al., 1997). Around the same time Scott Draves was experimenting with generating machine code at run-time

come. Since dynamic compilation was not a major goal in the development of FAUST, the current code generator implementation is not conducive to being dynamically loaded, nor has it been yet optimized for fast turnaround.

[17]http://docs.pure-lang.googlecode.com/hg/pd-faust.html (accessed 2011).

from LISP-expressions for audio synthesis (Draves, 1996), as detailed in Section 1.2.6. However both of these pioneering examples also appear to have led more to follow-up work in graphical rather than audio domains.

For more than a decade the average personal computer has been capable of relatively flexible run-time audio synthesis through the modular strategy outlined in Section 2.3: an dynamic graph of buffered unit generators which Draves characterized as an "interpreter of vector primitives" (Draves, 1997). Max/MSP clearly fits this description, as do many textually-based creativity-support tools such as SuperCollider 3 (McCartney, 2002).

SuperCollider 3 (SC3) is an audio synthesis language with a syntax inspired by Smalltalk. It grants the user flexibility to specify new instruments (*SynthDefs*) from low-level unit generator classes at run-time. *SynthDefs* are specified using the interpreted *SCLang* language, and stored in a byte-code representation. By sending messages to a separate virtual machine (the *SCServer)* it can instantiate instances (*synths*) of these *SynthDefs. SCServer* interprets the byte-codes for each active *synth*, calling the appropriate pre-compiled library functions per unit generator to process block-based audio streams. Other commands to *SCServer* can modify the parameters and port connections of active *synths*, as well as remove them from the active audio graph.

The *jitlib* extension to SuperCollider provides a proxy pattern system above the *SCServer* that allows *SCLang* code to reference elements before they exist, and to modify synthesis routines without losing intermediate state. The *jitlib* name refers to just-in-time programming (as opposed to just-in-time compilation). Julian Rohruber states: "My reason to introduce the just in time programming library (*jitlib*) was to make an interface to write code while playing that removes the distinction between preparation and action, so that I could more easily change things and not lose the connection to the written representation ... One of the aims of *jitlib* here is to be able to exchange synth definitions and other processes within a referential environment conveniently so that there is less need to plan changes beforehand." (Collins et al., 2003) The additional layer of indirection makes possible the operation over variables and expressions before they have been defined, as well as the re-definition of sub-expressions while they are in use: "The fact that one can refer to an element before it exists as well as change it when it is already in use allows the refactoring of a sound algorithm at runtime." (de Campo, 2005) This flexibility however comes at a price: "This kind of on-the-fly creation is not as efficient in SC3 as it would be to prepare a performance with all needed sound functions and then interconnect them, but as long as this is not done twenty times a second the flexibility is worth the expense." (Collins et al., 2003) Although *jitlib* does not utilize dynamic compilation, it is a powerful example of the use of reflective meta-programming in a performance context. The smooth preservation of state between changes, and the flexibility due to indirection, have been inspiring to the author.

There have been discussions and experiments in the SuperCollider community as to the possibility of implementing dynamic compilation using LLVM[18], however the focus appears to be on the dynamic generation of new unit generators (e.g. from FAUST code), rather than the optimization of entire *SynthDefs*.

Other applications have supported code generation of audio routines at run-time; often for the purpose of exporting plugins for non-linear editing suites. SynthEdit is a commercial application for Windows to generate C++ code for VST plugins, by connecting *modules* in a visual programming environment.[19] It

[18]http://thread.gmane.org/gmane.comp.audio.supercollider.user/64138/focus=64225
[19]http://www.synthedit.com/ (accessed 2012).

also includes an extension system and SDK to define new modules. SynthMaker is another commercial Windows program for connecting components in a visual interface, which generates VST plugins as well as standalone applications.[20] It also supports a code component and single-sample feedback loops. Reaktor is a commercial visual programming environment for audio synthesis, running on Windows and OSX, which now includes a *Core* extension to allow users to create new unit generator modules, supporting single-sample delay.[21] Although all of these systems utilize a graphical programming interface comparable to Max, they have been generally geared more toward productivity-support in music/entertainment contexts than creativity-support of a more open-ended character.

Finally there have been several projects, presented concurrently with some of the research results in this dissertation, that revive the potential outlined by Draves and Ingalls. For example, in September 2010, Andrew Sorensen, the developer of Impromptu (a Scheme-based live-coding environment for OSX) announced a new version with an embedded compiler called *ICR* which can be used to define new natively-compiled audio instruments at run-time.[22] The ICR uses a Scheme-like language, but with some important differences. For example, it is statically typed, though type declarations can often be omitted for automatic inference. The ICR compiler is invoked using a **define-c** macro, and translates its arguments to LLVM IR code, using a sequence of type inference, IR translation, LLVM just-in-time compilation to machine code, and wrapping with a binding to the Scheme runtime. The result connects to the audio system in Impromptu using the OSX AudioUnit system, using shells that operate similarly to the proxy objects of the

[20]http://synthmaker.co.uk/ (accessed 2012).

[21]http://www.native-instruments.com/#/en/products/producer/reaktor-55/?page=1626 (accessed 2012).

[22]http://impromptu.moso.com.au/extras/ICR.html (accessed 2011).

SuperCollider *jitlib*. Users however must take care with the ICR: replacing existing functions must preserve the same argument signatures, and memory must in certain situations be manually freed by the user. An *Arena* pattern is of memory zones is provided to make this easier, but does not apply in all circumstances. Sorenson notes that the compilation time is slower than desired. In the same year, Thieleman presented a system generating machine code for audio synthesis routines written in the Haskell functional language, utilizing LLVM for dynamic compilation Thieleman (2010).

Chapter 5

Practice: Open Worlds

"The machine does not isolate man from the great problems of nature, but plunges him more deeply into them." Antoine de Saint-Exupéry, *Wind, Sand and Stars.*

This chapter focuses on creative work that is not merely using the computer as a tool to accelerate or assist something that could have been done with prior technologies, but which explores the artistic potential particular to the interactive computational medium. Specifically it focuses on the generative capability of a program to redefine itself while it runs, through techniques of meta-programming, and how this can remain efficient for media-intensive interactive arts by utilizing dynamic compilation to machine code. Meta-programming is familiar to generative artists, from the grammar productions of Lindenmeyer systems (Prusinkiewicz and Lindenmayer, 1996), to the expression productions of genetic programming (Koza, 1990). As noted in Section 2.4.2, many researchers have identified the necessity in open-ended systems, such as evolutionary models, to rewrite the rules while they run (Machado et al., 2004). Machine code generation however appears to be as yet unexplored for such real-time interactive generative art.

To narrow the scope to a concrete example, this chapter concentrates on the subdomain of immersive, interactive evolutionary ecosystems ("artificial natures", Ji, 2011), however the issues encountered may be extrapolated to other forms of interactive generative art, and perhaps also to domains such as video games and scientific visualization.

Practical concerns

To a large extent the design and implementation concerns mirror those of the authoring environments described in the previous chapter. The principal differences derive from the fact that new component descriptions are generated internally rather than received from human users. This has implications for how they are represented: since a human-friendly concrete syntax is no longer essential, new component descriptions may be directly specified in terms of an abstract syntax model or intermediate representation. Whichever format is used to represent new components, it must be amenable to autonomous generation, by some means specified by the designer.

Another important difference with creativity-support tools is that a generative artwork cannot appeal to the user to handle unexpected errors. An exhibited artwork cannot appeal to the designer; it must be autonomously robust and consistent. As with artificial life systems in general[1], it must be designed such that all generated code constructions must be valid, or handled gracefully otherwise.

Nevertheless, this should not overly restrict the open-endedness of the world. Kampis paraphrased the philosopher A. N. Whitehead (exchanging 'computation' for 'doing mathematics by real world systems') to note that no matter what in-

[1]For example, Tierra-family simulations use a bit-code representation in which any possible string is valid code, including a clever use of relative pattern-matching instead of absolute pointer addressing (Ray, 1991).

tended effect a computation may have, there are always possibilities for other unintended effects (Kampis, 1996), and that it would be nonsensical to claim that nature errs. Similarly, computation as such does not err; simply there are interactions between processes that do not meet (or supersede) the expectations of the designers. The more open-ended the problem space, the more the concept of error is replaced by polysemy and multiple interpretation.

There are also important temporal and quantitative differences with creativity support tools. Since code changes are not limited by human action and attention, the rate of new component declarations can potentially be much higher, and the set of active components could be much larger. The real-time constraints on the performance of the compilation workflow and memory management are thus much tighter.

5.1 Evolution in Artificial Natures

Computational models of evolution distinguish between an information-carrying primary form (the genotype) and a complex behavioral form (the phenotype). Variations between organism phenotypes are partly specified by variations in the genotype, which occur at the moment of reproduction.[2] In asexual reproduction, the offspring receives a slightly mutated copy of the parent genotype; in sexual reproduction a cross-over operation mixes contents from both parent genotypes to create the offspring. Evolutionary selection however operates at the level of the phenotype, according to the behaviors it performs in the environment. It is important to note that many variations and mutations have negligible effect on the 'fitness' of a phenotype, and are thus evolutionarily *neutral*. Neutral drift of

[2]Of course, phenotypes are as much a product of their environment as the genotype, however this is not always incorporated in computational models.

gene space has been considered an important source of evolutionary creativity.

Beyond a minimal complexity, it is difficult to derive the fitness of a genotype from first principles; within an interactive environment (such as an ecosystem) it is utterly impossible. Evolution proceeds with a cycle of exploration, discovery, evaluation and integration within an open-ended space.

5.1.1 Levels of rules

Constructing an artificial evolutionary system requires the specification of rules at several hierarchical levels, possibly including:

1. Rules for the dynamic environment the organisms inhabit (the deterministic component of the environment).
2. Rules for external (e.g., human) interaction (the non-deterministic component of the environment).
3. Rules for how genotypes evolve (such as genetic mutation and crossover).
4. Rules for how a phenotype is generated from a genotype (development).
5. Rules for how a phenotype behaves over its life-span.
6. Rules for selection (the choice of which phenotypes can reproduce).

We can analyze these levels in terms of frequencies of operation and frequencies of change, in order to determine appropriate sites of dynamic compilation (see Section 3.1.2). The rules in levels (1) and (2) represent the underlying laws of the world and the broader framing of the artwork. They may operate at high frequency and thus be performance-critical, requiring compilation to machine code.[3]

[3]Part of the reason these rules are performance-critical is the need for accuracy: "It is important that the physical simulation be reasonably accurate when optimizing for creatures that

In most cases we do not expect the laws of a world to change, in which case static compilation is appropriate. However, the environmental rules and framing of a work may change often during the design stage, in a similar fashion to the creativity support tools of the previous chapter. Furthermore, the philosopher C. S. Peirce ruminated over the possibility of 'laws' changing over large temporal and spatial scales (Peirce, 1931). However if the environmental rules of an artificial nature change, they do so only at low frequencies (else they would not be considered the environment). If the rules are expected to change, dynamic compilation may be valuable.

The rules in level (3) typically only operate when an organism is conceived, which is generally a relatively low frequency event and not performance-critical, so an interpreted implementation is viable. An interpreted strategy can also easily handle models in which the mode of evolution varies over time.

In many artificial evolutionary models the rules in level (4) are not given physical interpretations, and are treated as somewhat instantaneous. In this case they can be treated similarly as level (3). However, if development is taken seriously as a spatiotemporal process, it must be dealt with along with phenotype behavior in level (5).

Rules in (5) operate throughout an organism's lifetime, and are therefore another performance-critical component of the work. Implementing these rules in machine code may significantly increase the viable population size and organism complexity without compromising real-time performance requirements. However the motivation is that these rules change significantly from organism to organism, generation to generation. With static compilation, organisms implemented

can move within it. Any bugs that allow energy leaks from non-conservation, or even round-off errors, will inevitably be discovered and exploited by the evolving creatures." (Sims, 1994) in machine code can only vary parametrically between each other. A parametric model is equivalent to a dynamical system, which we saw criticized by artificial life researchers for missing out on the vast open-ended spaces of behavior of dynamical structure (see Section 2.4.2). Kauffman defined an open world as one in which there are always *new ways of making a living* to be found (a transfinite space of viable behaviors) (Kauffman, 2002). A design built upon dynamic compilation can leverage dynamical structures to result in vastly open-ended spaces of possible phenotypes, as new ways of making a living. This will be the main site of discussion in this chapter.

The rules of selection in level (6) evaluate phenotypes in terms of their 'fitness' for reproduction. In the traditional genetic algorithm, fitness measurement is made against a pre-determined function after the candidate has performed its 'task' (Holland, 1992). This occurs at a rate determined by the population size and life-span, which is usually not performance-critical, and thus can be interpreted. However, a static fitness function turns evolution into a tool of optimization, whereas the point of interest of an open-ended artificial nature is to avoid resolution; a static fitness function is not appropriate. Instead, selection may be stated in terms of *viability*: what do organisms need to achieve to remain alive, and thus partake in reproduction? Viability measures could be as simple as maintaining a positive energy store and avoiding predators, for example. However, this is an evaluation that must be made continuously through an organism's lifetime, and thus a performance-critical aspect of the work. It may be integrated into the laws of the world, or into the rules of the phenotype in level (5), with a static or dynamic compilation strategy.

5.1.2 Models of genotype representation and phenotype implementation

The literature shows several prevalent approaches to describing and modeling the behavior of artificial organisms (and more generally, artificial agents). In most cases organisms are represented in terms of sensors, effectors (actuators), and internal processing and state.

Artificial Neural Networks (ANN) build upon an analogy with biological neurons. Neurons are connected into a directed graph, in which sensors are represented as input neurons and actuators as output neurons. If the graph contains cycles, it is a *recurrent* neural network. Connections between neurons are weighted, and a neuron computes its output connection value according to a function of the values and weights of its input connections. The function used by a neuron is often simply a threshold, or a continuous sigmoid such as a hyperbolic tangent. Neural networks are widely used for adaptive learning in artificial intelligence, where the network connectivity and weights vary according to evaluation feedback, however an ANN need not be adaptive. Dewdney however noted that the functionally simple but massively parallel nature of ANNs are not an ideal match to the synchronous but diverse instruction-set architecture of general purpose computing hardware, and may consume more resources than alternative models (Koza, 1991). Generating C code from an ANN specification requires tracing the data-flow from sensors to actuators, and the addition of persistent state variables for any recurrent (feedback) connections.

Several early artificial life models utilized a finite state automaton (FSA) genotype representation, in many cases generating byte-code for a virtual machine (Ray, 1991; Rasmussen et al., 1992), a clear example of meta-programming.

Phenotype behavior is enacted by interpreting the series of byte-code instructions in the genotype, which are drawn from a pre-defined instruction set. The byte-code can be represented as a binary string, making it directly amenable to mutation/cross-over operations, however the byte-code representation must be carefully chosen to ensure that all strings produced are syntactically valid programs, and generated byte-code must be further validated to ensure semantic correctness before running (an issue which does not affect ANNs). For a sufficiently expressive byte-code (for example, one that supports conditional backward jumps) this is intractable due to the halting problem, however it could be detected at run-time and removed accordingly. Generating C code from an FSA may appear trivial (iterating the instructions, with each instruction having a corresponding C statement generator), but becomes more complex as more control-flow capabilities are added to the instruction set, and generating optimal code may require significant effort.

Genetic programming (GP) represents a phenotype program as a tree of expressions (such as S-expressions of LISP family languages, as in Koza, 1990). Encoding genotypes as expression trees offers a more open-ended space of evolution than bit-strings: "Variable length genotypes such as hierarchical Lisp expressions or other computer programs can be useful in expanding the set of possible results beyond a predefined genetic space of fixed dimensions. Genetic languages such as these allow new parameters and new dimensions to be added to the genetic space as an evolution proceeds, and therefore define rather a hyperspace of possible results." (Sims, 1994) Mutation and cross-over of phenotypes indicates the modification or replacement of a node (including its sub-tree), making it better suited to naturally hierarchical problems, as well as easier to understand (Koza, 1991). However the GP examples given by Koza can only represent tree structures of data-flow, and thus-defined are strictly less expressive than an ANN or FSA. The expression types within a GP tree can be low- or high-level operations, as with FSA byte-code. Generating C code from a tree of S-exprs is trivial in a similar fashion to FSAs, simply by traversing the tree depth-first. Programs may need to be *reduced* first however, to remove redundant expressions that have no side-effects.

Drawing upon experience with hardware robotics, Rodney Brooks proposed a *subsumption architecture* to represent phenotype behavior (Brooks and Ross, 1996). It organizes complex behavior in terms of parallel layers of simpler behavioral modules (most of which have almost direct access to sensors and motors), where the goals of higher level layers subsume the goals of lower-level, more primitive behaviors. For example, an 'exploration' module may subsume sensory modules and a 'random walk' module; the 'random walk' module in turn subsumes 'move forward' and 'turn' modules. All modules are potentially active at all times, testing their input 'applicability predicates' and generating output values if the conditions are met, but higher-level modules can suppress the inputs or inhibit the outputs of lower-level modules. It effectively provides a spectrum from immediate reflexes to longer-term objectives, degrades gracefully if one module is modified (a positive feature for evolutionary search), and promotes fast response times for task-oriented situations. The subsumption architecture is a higher-level strategy regarding organism behavior that does not specify a particular implementation, however Koza has demonstrated the use of GP to generate subsumption architecture programs.[4]

Sims used a broadly GP approach (embedding a neural network) for his groundbreaking *Evolving Virtual Creatures* (Sims, 1994). In this project the basic GP sys-

[4]http://robotics.usc.edu/~maja/teaching/cs584/papers/koza93evolution.pdf

tem is extended to include a developmental 'instancing' system, in which common tree structures can be repeated through the organism's nervous system (somewhat akin to macros or subroutines). The directed graphs used by Sims were used not only to model the phenotype neural structure and behavior, but also the phenotype's bodily structure. Re-using sub-trees therefore also introduces biologically plausible symmetries into the morphology (and also therefore the neural components embedded within them). Neurons can connect to other neurons in a neighboring body part, and in addition, a set of global neurons (not associated with a particular body part) can connect or be connected by neurons in any other part of the body. Neurons can be sensors, actuators, or internal processing nodes drawn from a set of operators including constant, sum, product, divide, sum-threshold, greater-than, sign-of, min, max, abs, if, interpolate, sin, cos, atan, log, expt, sigmoid, integrate, differentiate, smooth, memory, oscillate-wave, oscillate-saw (some of the latter operators require persistent state). The resulting neural system is thus a synchronous data-flow program. Aside from mutations adjusting numeric parameters of nodes and connection weights, mutations can also add/remove nodes, and and/redirect/remove connections. After mutation, any un-connected nodes are removed to prevent genotypes growing too large (however Sims acknowledges that retaining such 'junk' gene data could have evolutionary advantages). Cross-over replaces a sub-tree of one genotype with a sub-tree of another. After these processes, a significant amount of checking and validation is performed to ensure a valid phenotype will be generated.

5.2 Time of Doubles

Time of Doubles is the most recent of a series of *Artificial Nature* art installations involving immersive, interactive ecosystems, exhibited since by 2008 by Haru Ji and the author (Ji, 2011; Wakefield and Ji, 2009). Each work involves large populations of evolving, artificial life organisms, rendered in a dynamic, 3D environment, with which humans can interact. This section describes how the existing *Time of Doubles* artwork was extended using meta-programming and dynamic compilation to support a much richer space of evolution. It was also designed and developed using some techniques familiar from the authoring environments in the previous chapter.

The dynamic environment inhabited by organisms in *Time of Doubles* includes systems of particles (potentially food or waste products) drifting within the currents of a dynamic fluid simulation. When people visit the installation, their physical shapes and movements are captured by infrared depth cameras and projected as 'doubles' into the virtual world. The shapes become regions of high nutritious particle density, and their movements add currents to the fluid. Like most evolutionary artworks, the population evolves according to a 'selection' of which organisms can reproduce. Unlike genetic algorithms, in which a selective function to measure fitness for reproduction is specified in advance, and unlike 'artificial selection', in which a human must individually select each organism to reproduce, *Time of Doubles* uses *ecosystemic* or 'endogenous selection', as detailed in (McCormack, 2007). Each organism must maintain an energy balance to stay alive, by finding & consuming particles in the environment. Human input influences factors of the environment (location of food, introduction of fluid currents), and thus only indirectly influences the evolving populations; nevertheless as or-

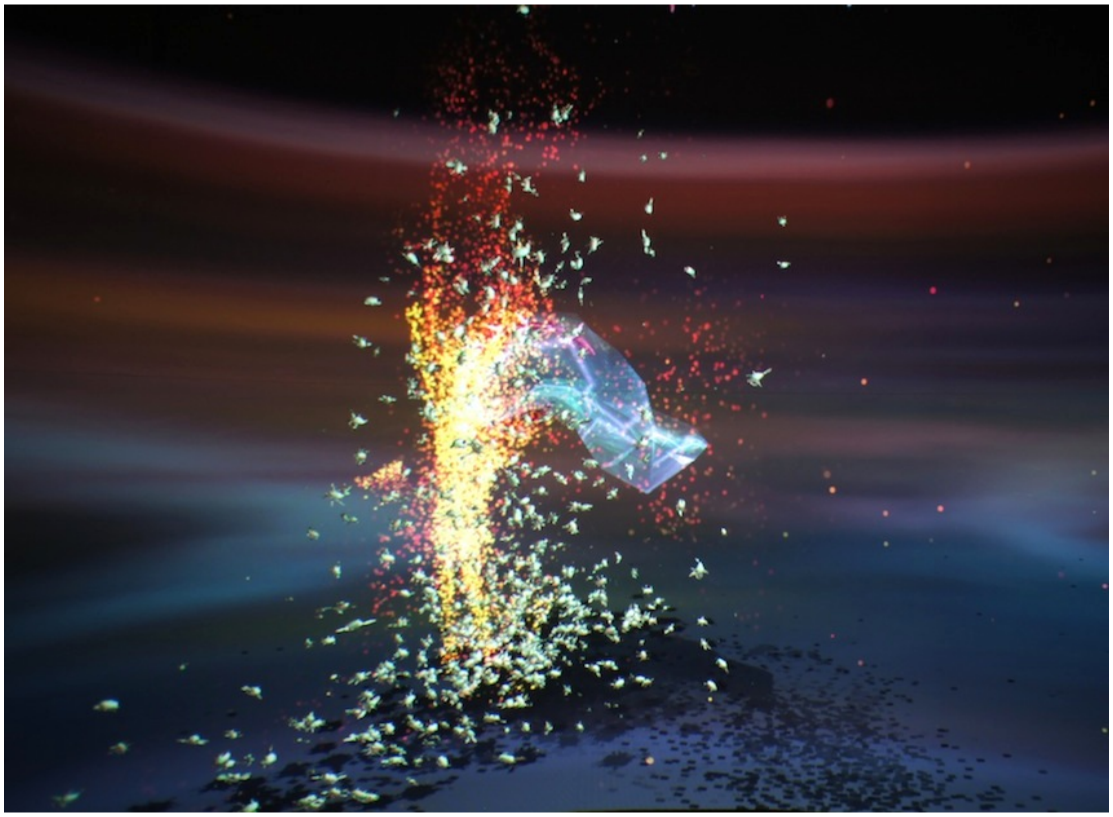


Figure 5.2.1: Screenshot of *Time of Doubles,* in which a human 'double' (yellow particles) is being consumed by a population of organisms (green objects). Image by author.

ganisms discover the locations of the high-energy particles, they are more able to survive and reproduce, leading to explosions of populations consuming their way across the human 'doubles' (see Figure 5.2.1).

5.2.1 Implementation

The *Time of Doubles* artwork is implemented as a fairly simple, statically compiled virtual machine, which provides some basic system resources (windowing, audio, user interface, sensor drivers, etc.), and hosts a Lua interpreter. At launch, this machine loads and runs scripts to create the various elements of the *Time of Doubles* world. At any time during execution these scripts can be edited and reloaded (in a similar fashion to the creativity support tools of the previous chapter) to redefine the world at run-time, while preserving other state. This proved to be very useful for the authoring of the artwork.

The underlying laws of the world and of human interaction (levels (1) and (2) from section 5.1.1) include simulation of the fluid environment, particle systems, collisions between objects (including organisms) in space, the injection of viewers' actions and depth-profiles into the space, and other high-frequency, performance-critical subsystems. Although these subsystems are not expected to change during an exhibition, they are a site of intense change during the development of the work. These layers are written in C++ code which is dynamically compiled by the main script using the *luaclang* module (see Section 3.2.2). As with the main scripts, these can be dynamically replaced without stopping execution. Less performance-critical aspects of the world, including genetic reproduction and mutation, and currently also development of phenotypes from genotypes (levels (3) and (4) from section 5.1.1), and are implemented directly in Lua.

The open-ended, performance-critical part of the work is the phenotype behavior itself. The viability constraint in *Time of Doubles* of energy balance applies throughout an organism's lifetime, and is also partly implemented in the high-frequency phenotype behavior functions. The focus of research in extending *Time of Doubles* is the generation, compilation and management of rules of phenotype behavior. The goal is to support thousands of active organisms, each one having a distinct piece of machine code to enact its behavior, while maintaining minimum performance of the system as a whole for perceptual immersion.

We will first discuss what the generated code for each phenotype organism does, and then describe how these phenotype functions are derived from evolving genotype representations.

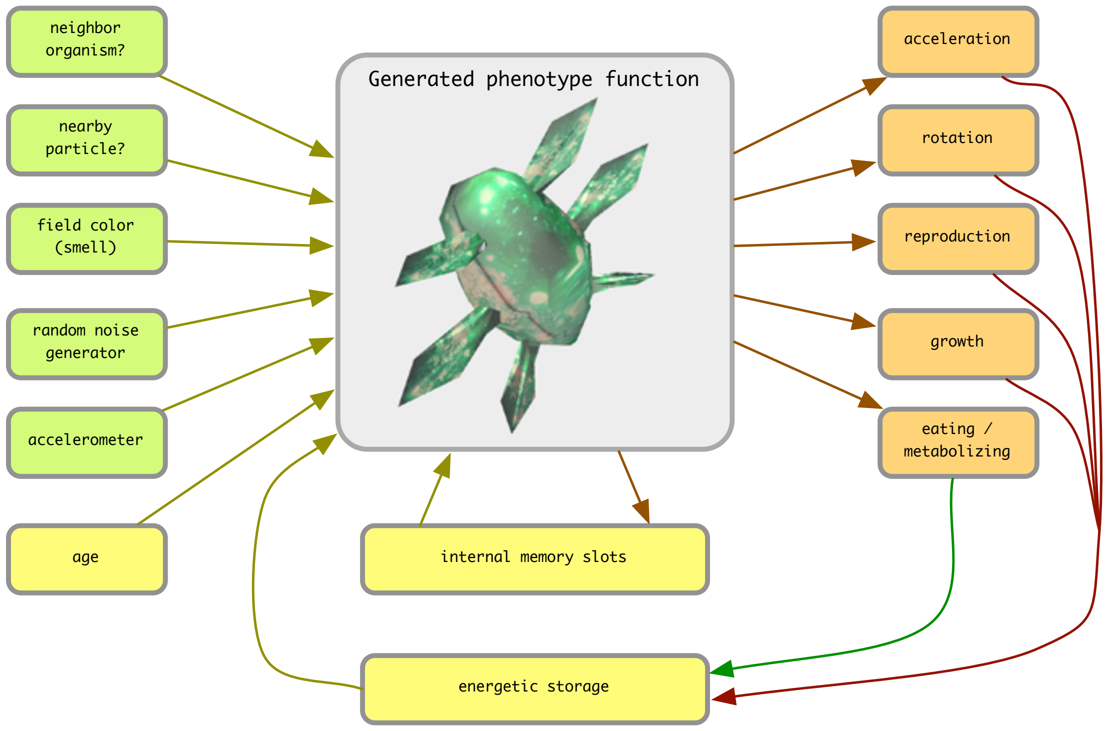


Figure 5.2.2: Semantic components of the phenotype behavior functions of an organism generated in *Time of Doubles*. Sensors are marked as green, internal state is yellow, and side-effect actuators are orange. Most actions have a negative effect on stored energy, with the exception of memory writing (negligible effect) and eating (positive effect).

5.2.2 Phenotype

The generated code for a phenotype includes a function to initialize the organism state at birth, and a function to update the state throughout the life-span (the phenotype behavior). This behavior function takes a number of inputs (external and internal sensors), applies a series of expressions, and produces a number of side-effects including motor actions, growth, reproduction, and memory storage (see Figure 5.2.2). Most actions have a negative effect on stored energy, with the exception of memory writing (no effect) and eating (positive effect). Sensors are also assumed to have negligible energetic cost.

The function is defined to conform to the C ABI, and takes as an argument a pointer to the organism state, and a pointer to an object representing the local environment of the organism. The function itself is stored within the organism state. The main simulation routine in *Time of Doubles* visits all living organism objects and calls the phenotype function of each one, passing in the organism state and a data structure configured for its the local environment. After calling the phenotype function of an organism, the main simulation routine performs "fixed pipeline" procedures including physical movements as well as procedures to remove an organism that failed viability tests.

Within the phenotype function, internal sensor values (noise generator, age, memory slots) are read directly from the organism state, and external sensor values (neighboring objects, local field intensities) are read from the local environment object. The accelerometer input computes a value according to the organism movement and the local fluid velocity. Intermediate expressions to compute action values include a variety of elementary mathematical operators, and actions are embedded within conditional structures. Actions are expressed by adjusting values in the organism state (acceleration, rotation, growth, memory updates), or by sending messages to the local environment object (reproduction) or both (eating). Energy effects are also applied as increments and decrements to the organism state object.

5.2.3 Genotype

The genotypes in *Time of Doubles* combine aspects of genetic programming, subsumption architecture and finite state automata. The behavior function is represented a graph (an example is depicted in Figure 5.2.3), upon which evolutionary mutations can occur in a similar fashion to genetic programming. The lower levels of the graph represent a control-flow tree of conditional expressions and side-effects

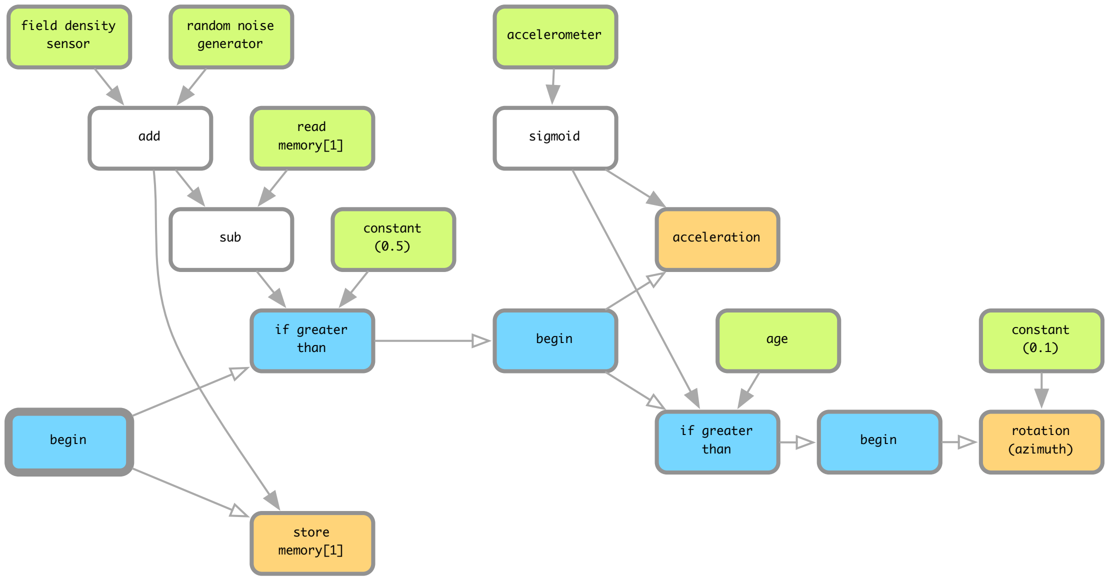


Figure 5.2.3: A small example genotype graph (the actual genotypes in use in *Time of Doubles* are generally much larger). Blue boxes and hollow arrows indicate the control-flow graph, with the heavily-bordered left-most box being the entry point. Orange boxes indicate the side-effect operations. Solid arrows depict the data-flow graph, where white boxes are expressions and green boxes are terminal inputs. Note that the control-flow graph is a tree, but the combined control-flow/data-flow graph is not, since some nodes are used in more than one branch; however it is directed and acyclic. The only feedback path is implicit, safely implemented by the memory read and store operations.

(the applicative predicates of a subsumption architecture). The control-flow tree branches always terminate in side-effect nodes (actuators and memory stores). The control-flow non-terminals are either sequence nodes (**begin**) or conditional branches (**if**). Branch conditions and side-effects refer to nodes in the data-flow graph for their arguments. The data-flow portion permits joins and disconnected graphs, and cannot be represented as a tree. The data-flow portion need only be directed and acyclic: a node in the data-flow graph may be used in more than one branch. Furthermore, the data-flow graph need not be fully connected (there may be several unconnected subgraphs). Nevertheless, feedback is not permitted (a node may not depend on a parent node). Instead, the organism memory slots are available for use in **read** (data-flow expression) and **write** (side-effect) operations.

These rules are expressed in the pseudo-grammar of listing 5.1. The specification of the graph in these terms is important to ensure validity before generating code.

When an organism reproduces, the genome of the offspring is a copy of the parent with a number of mutations applied. Several different kinds of mutations can occur to the graph. Constant nodes may have new values applied, with a Gaussian divergence. Operators may be replaced with other operators of the same arity (retaining connections). Similarly, side-effect operations can be exchanged for others of the same arity. Sub-sections of data-flow expression graph can be exchanged, so long as a feedback path is not created. A node in the expression graph can be reduced by removing an intermediate step (orphaned branches are not discarded but remain as island graphs) or augmented by the insertion of a new operator (additionally required arguments are selected from the graphs that already exist). Control-flow conditional nodes and side-effect operations may switch to a different target in the data-flow graph. Leaf nodes (side-effects) can be

Listing 5.1: The genotype representation rules as a loosely expressed EBNF grammar. The full list of sensors, operators and side-effects are omitted for brevity. The actual genotype structure does not strictly conform to a grammar, since a particular *expression* node may be found in more than one branch.

```
/* The root rule is begin: */
begin = control-flow, control-flow
control-flow = subsumption-module
             | side-effect
             | begin
subsumption-module = condition, begin
condition = "if greater than", expression, expression
             | "if nearby organism"
             | "if nearby particle"
expression = constant
             | sensor
             | operator
sensor = "random number generator"
             | "memory read", memory-slot
             | "field-density"
             | "accelerometer"
             | "age"
             /* etc. */
operator = "add", expression, expression
             | "mul", expression, expression
             | "sigmoid", expression
             /* etc. */
side-effect = "memory store", memory-slot, expression
             | "acceleration", expression
             | "rotation azimuth", expression
             /* etc. */
```

moved from any **begin** block to any another. Conditional nodes can be moved to different blocks, so long as a feedback path does not occur. These validations are performed to ensure that each kind of mutation results in a conformant genotype. Additional constraints are made to ensure an upper bound on the total number of operations and a maximal depth of conditional trees.

A routine to generate new, random (and valid) genotype graphs is used to seed the initial population of Time of Doubles. In addition, new randomized seed organisms are added to the system periodically to provide new genetic material and ensure that if extinction occurs, it does not permanently empty the installation. The rate of seeding is low, and inversely proportional to the population size in order to allow evolution to express itself.

The genotype of each organism is implemented as a data structure in the interpreted Lua language, in a format somewhat similar to the LuaAV *Def* description (see Section 4.2.2). The control-flow tree refers to data-flow nodes by name. Code generation from this data structure targets the C language (see Figure 5.2), and involves two passes. The first pass traverses the graph to identify a sorted list of data-flow nodes used in more than one location. Code generation then occurs by traversing these nodes first and producing assignment statements, and then traversing the control-flow tree to express conditional and sequence blocks. Memoization is used to ensure that an expression is not generated more than once; instead each expression generates an assignment statement whose unique name is subsequently used in place of the expression. A standard header is added to the resultant C code to add definitions of used types and non-standard operators, along with a further function to initialize the memory slots, and then *luaclang* (see Section 3.2.2) is used to compile the function to machine code, and store the results in the organism object.

Listing 5.2: A fragment of the genotype function (in the C language) generated from the example in Figure 5.2.3. The actual genotypes in use in *Time of Doubles* are much larger than this example.

```
extern "C" void update(Organisms& self, Environment& env
    ) {
    /* used sensors */
    const float density = env.field_density;
    const float noise0 = noise();
    const float memory1 = self.memory[1];
    const float accelerometer = self.accelerometer;
    const float age = self.age;

    /* operators */
    const float add1 = density + noise0;
    const float sub2 = add1 - memory1;
    const float sigmoid3 = sigmoid(accelerometer);

    /* applicative predicates */
    if (sub2 > 0.5) {
        self.acceleration = sigmoid3;
        self.energy -= sigmoid3 * ACCELERATION_COST;
        if (sigmoid3 > age) {
            self.rotation_azimuth = 0.1;
            self.energy -= 0.1 * ROTATION_COST;
        }
    }
    self.memory[1] = add1;
}
```

Chapter 6

Conclusion

"The single drawing or painting, or even a series of them, is certainly not the media adequate work of digital art. Adequate are works that need the computer medium for their existence, and would not exist without that. Interactive art is one such case. It makes adequate use of the computer when the work, in order to react to some activity of the audience, relies on symbolic action (as the execution of a program)." (Nake, 2007)

There can be no doubt that computationally driven art puts humans within reach of otherwise unattainable aesthetic experiences. The fact may be more sharply felt when systems are interactive: an intimate yet unpredictable feedback between human input and computational results can move through a sequence of complex spaces that evades prediction. To do so, a creative medium should make it easy to propose and evaluate *what-if* scenarios, and an open world artwork must itself be exploring *what-if* scenarios. This dissertation followed a motivating suggestion that a 'second-order' capacity of computation–the ability to redefine a program while it runs–provides the plasticity appropriate to an 'adequate kind' of interactive computational art.

Keeping the description of the program separate (immutable, out-of-time) from the execution limits its potential range of behavior. On the other hand, moving decision-making capabilities to run-time reduces efficiency. This conflict of interests is exacerbated in real-time scenarios, where processing must not terminate nor consume excessive computational resources in a given window of time. Such conditions are make-or-break for performance systems and generative installations. They are also valuable for the artist or composer at work in the studio, since reducing both restrictions and latencies between action and perception can increase fluidity with the medium.

We have seen that many creative software in which both efficiency and flexibility are a premium make a compromise by means of fixed efficient pre-compiled modular 'black boxes' that can be freely re-assembled at run-time. Immutability is confined to these black boxes with a predefined granularity hopefully suited to the domain in question. Since no such compromise can anticipate all possible situations flawlessly, this thesis suggested instead that the lines of compromise themselves be placed in the hands of users and the autonomous capabilities of generative artworks.

Specifically, this dissertation has demonstrated that, through the use of metaprogramming and dynamic compilation, efficient media-processing programs can evolve to become new efficient programs through a creative dialogue with human experience.

6.1 Evaluation

The demonstrations described in Chapters 4 and 5 have satisfied the criteria set out in the Introduction, and in addition have seen significant public acceptance.

6.1.1 LuaAV audio.Def

Dynamic compilation of audio synthesis in LuaAV was first demonstrated in 2009 (Smith and Wakefield, 2009), but audio.Def was first released in January, 2011[1], and fully open-sourced and hosted on GitHub since December 2011[2]. LuaAV has been documented in several publications (Smith and Wakefield, 2009; Wakefield et al., 2010), and has been used for virtual reality simulations, international gallery exhibits, performances of live-coding, and also as a pedagogical tool at several universities, including the *Composing for the AlloSphere* course at Media Arts & Technology, UCSB (2010).

Prior to 2011, LuaAV utilized a modular ugen-based audio system inherited from the author's previous work *Vessel* (Wakefield, 2007; Wakefield and Smith, 2007). Evaluation of audio.Def is made relative to this system and to the closely related SuperCollider 3 (McCartney, 2002) and ChucK (Wang and Cook, 2004) systems as appropriate.

Flexibility

All systems provide comparably low-level primitives. In terms of granularity of change, audio.Def provides a similar abstraction to the SuperCollider *SynthDef*, metaphorically akin to an instrument-type within an orchestra. A significant

[1] http://lua-av.mat.ucsb.edu/blog/?p=386 (accessed 2012).
[2] https://github.com/LuaAV/LuaAV (accessed 2012)

feature of Vessel and ChucK was the ability to schedule synthesis graphs at sample-accurate times ('strongly timed') to support exploration of granular and micro-sonic techniques, which is also matched by audio.Def.

Although the instrument paradigm is less flexible than the raw ugen paradigm in Vessel and ChucK, it does provide flexible input ports (which can accommodate single- and multi-channel signals, busses and constants) for sample-accurate dynamism during instrument playback.[3]

However audio.Def also supports the design of arbitrary user-defined synthesis algorithms embedding feedback loop down to a single sample within instruments (see Listings 6.1 and 6.2), such as creating arbitrary new filters and physical models, which has not been possible before.

Listing 6.1: Example of single-sample feedback in a LuaAV instrument definition. The 'Basic' instrument uses single-sample feedback as a simple phase accumulator, to drive a sine oscillator.

```
Basic = Def{
    freq = 440,

    -- define a recursive accumulator:
    phase = P"phase" + P"freq" / audio.samplerate(),

    -- use it to create a sine oscillator:
    Sin{ P"phase" * math.pi * 2 }
}
```

[3]Furthermore, incorporating a raw ugen interface into audio.Def, to use non-instrument paradigm where desired, is both feasible and partially implemented. It is not however expected to scale as efficiently as the instrument paradigm for micro-sonic synthesis.

Listing 6.2: Example of single-sample feedback in a LuaAV instrument definition. The 'KP' instrument uses a feedback path to implement a Karplus-Strong string synthesis instrument capable of high frequencies, with filters in the feedback loop.

```
-- Karplus Strong physical model, with built-in filter:
KP = Def{
    -- string fundamental frequency (Hz):
    freq = 220,
    -- feedback filter cutoff frequency (Hz):
    cutoff = 10000,
    -- string T60 decay time (in seconds):
    t60 = 2,

    -- define recursive expression (in terms of ‘fb’)
    fb = Delay{
        -- decay factor:
        0.001^(P"t60"/P"freq") *
        -- prevent DC saturation:
        DCBlock{
            -- raw input:
            P"input" +
            -- filtered feedback:
            biquad.LP{ P"fb", P"cutoff", 1 }
        },
        -- delay period:
        delay = 1 / P"freq",
    },

    -- stereo output:
    Pan2{ P"fb", pan=P"pan" }
}
```

Efficiency

Direct comparisons are difficult to make, given the different contextual environments, however LuaAV audio.Def provides comparable performance to SuperCollider 3, and significantly improved performance over the existing “strongly-timed” systems of Vessel and particularly ChucK.

For example, a bank of 80 Karplus-Strong string synthesis modules, triggered randomly by individual noise burst voices being created and scheduled with sample accuracy at 300 bursts per second, sits comfortably around 15% of available processing time on a 2GHz Intel Core i7 MacBook Pro, 4GB RAM, running OS X 10.6.8 (with LuaAV audio running at 44.1kHz and a block size of 64 samples). It may be possible to improve this performance result further with more aggressive optimizations. For example, variants of an algorithm for different port type connections (signal- and control-rate) could be compiled lazily as needed, removing a significant amount of dynamic processing overhead.

Aside from the audio synthesis performance, the compile time of instruments is on the scale of tens of milliseconds, which is acceptable (though improvements would be welcome), while the time to construct a voice is less than a tenth of a millisecond, which is certainly satisfactory.

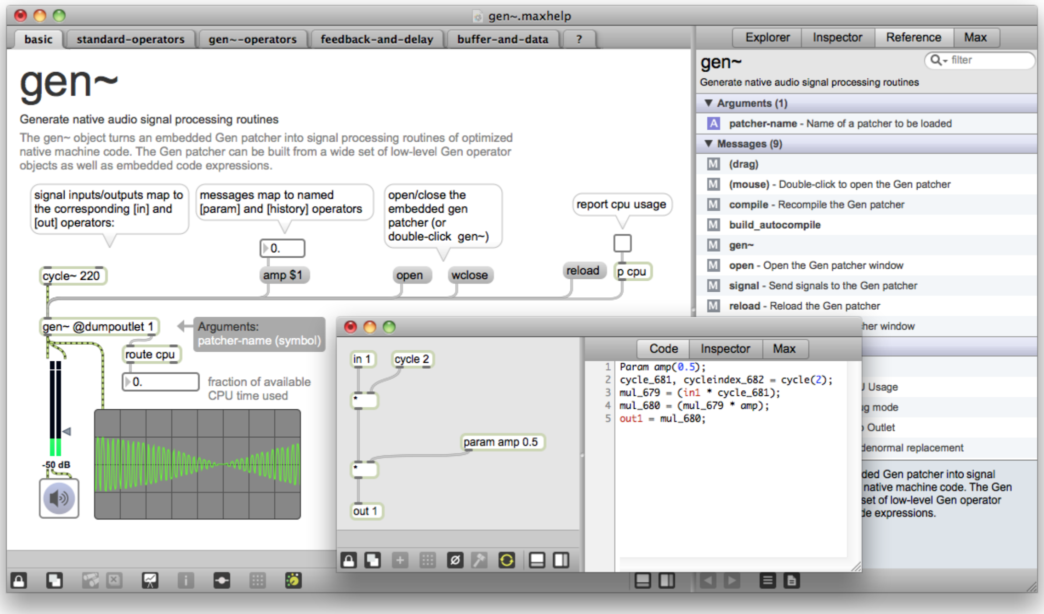
Attempts to streamline the compilation time by removing optimization passes in LLVM showed no discernible difference. Furthermore, an attempt to bypass the C++ parsing layer of Clang by generating LLVM intermediate code directly also made little difference to compilation time. This can perhaps be understood as due to the generated C++ code already being quite minimal and streamlined. The use of external #include directives, complex templated types, virtuals, function and operator overloading etc. are all avoided in the generated code, ensuring rapid parsing times. Generating LLVM intermediate code on the other hand requires greater effort due to the less flexible syntax. The other values of C++ as a target language (ease of connecting to C and Lua APIs, readability of output) easily outweigh the meager benefit of bypassing Clang.

Uninterrupted interface

LuaAV's audio.Def can successfully host an endless stream of newly defined synths while it runs. Careful garbage collection avoids memory leaking or exhaustion.

The first efforts toward audio.Def attempted to mirror the existing Vessel programming interface, however it became apparent that supporting granular approaches with a procedural approach to ugen graph construction was not feasible. Instead, the division of the whole scene into changeable units was made explicit through the use of the **Def{}** call, mirroring the style of SuperCollider 3's *SynthDef*.

The audio.Def programming interface emphasizes its declarative nature by re-using existing the data structure concept of the host language (Lua tables), requiring no new skills or special languages. Furthermore, this makes synthesis design readily available to generative and transformative techniques that would be otherwise difficult to express (such as mutating and breeding instruments).



Generate native audio signal processing routines

The gen~ object turns an embedded Gen patcher into signal processing routines of optimized native machine code. The Gen patcher can be built from a wide set of low-level Gen operator objects as well as embedded code expressions.

signal inputs/outputs map to the corresponding [in] and [out] operators:

messages map to named [param] and [history] operators

open/close the embedded gen patcher (or double-click gen~)

report cpu usage

Arguments: patcher-name (symbol)

fraction of available CPU time used

gen~

Generate native audio signal processing routines

Arguments (1)

patcher-name - Name of a patcher to be loaded

Messages (9)

(drag)

(mouse) - Double-click to open the Gen patcher

compile - Recompile the Gen patcher

build_autocompile

gen~

open - Open the Gen patcher window

signal - Send signals to the Gen patcher

reload - Reload the Gen patcher

```
Param amp(0.5);
cycle_681, cycleindex_682 = cycle(2);
mul_679 = (in1 * cycle_681);
mul_680 = (mul_679 * amp);
out1 = mul_680;
```

Figure 6.1.1: The *gen~* help patcher in Max/MSP 6.0.7 (accessed 2012).

6.1.2 Max/MSP Gen

Gen was first released to the public as an integrated optional extension to Max 6.0 in late 2011 (see Figure 6.1.1), and has since grown a good community of users.[4] It has been used as a supporting platform for pedagogy, such as the *Computer Music II* graduate seminar at Columbia University, USA, in Spring 2011.[5]

Flexibility

As well as easily representing existing algorithms, Gen achieves the goal of making possible algorithms that were previously impractical or impossible in Max/MSP, through the combination of lower-level and more generalizable primitives. The most obvious way this is achieved is through support for feedback signal paths

[4]See the dedicated forum at http://cycling74.com/forums/forum.php?id=13 (accessed 2012).

[5]http://music.columbia.edu/cmc/courses/g6611/spring2012/ (accessed 2012). Professor Brad Garton, in personal communication, attested that "some students really loved it".

Complex filters	Moog ladder
Nonlinear waveform generation	Feedback frequency modulation Feedback amplitude modulation
Chaotic waveform generation	Logistic map Lorenz attractor Navier Stokes simulator
Physical modeling	Karplus Strong string model Waveguide string model Flute model of Valimaki et al. (1992) Feedback delay networks Finite difference vocal synthesis
Waveform segmentation	Zero crossing triggered sampler Waveset synthesis Interpolated wave segmentation
Sample accurate transformations	Buffer shuffling Procedural enveloping
Microsound	Granular pitch-shifting ModFM FOF generation Pulsar synthesis

Table 6.1.1: Examples of signal processing algorithms implemented by users of gen~ that are impossible or impractical in Max/MSP.

down to a single sample using the **history** (representing the Z^{-1} operation) and **delay** operators. These have been used not only for filters and physical models, but also varieties of sample-accurate waveform segmentation. Examples contributed by the community and the author are listed in Table 6.1.1.

In addition, through the embedded *GenExpr* language, users can express control-flow that was previously cumbersome or impossible to express in the visual data-flow paradigm, including subroutines, *if* blocks, *while* and *for* loops. Previously in MSP emulated signal-rate *if* branches would execute both true and false cases and discard the unused results; with Gen only the used paths are executed. Previous to *GenExpr*, fixed-size *for* loops could be only partially emulated by duplicating sub-sections of a patch, but cumbersome to modify; signal-rate, dynamically sized *for* loops were basically impossible.

Before Gen, constructing such routines required developing specialized externals in C/C++. Using *gen*~ these examples are written on-the-fly, auditioning results as the algorithm is edited.

Efficiency

In order to compare performance of *gen*~ and MSP, a number of patchers were created using built-in MSP objects, and duplicate patchers created using *gen*~ (some representative measurements are listed in Table 6.1.2). In almost all cases individual operators in *gen*~ perform similarly to MSP (though there is a slight extra overhead in *gen*~ to support dynamic switching of compiled code). However once patchers grow in size, the performance benefits of *gen*~ are more clearly visible, and generally quite satisfactory. This can be attributed to the locality of operations in memory, availability of global optimizations, and the avoidance of heap memory buffers between operators.

Interface

Gen extends the existing conceptual model and infrastructure of Max (which is already familiar to many composers and artists) to lower-level abstractions, without sacrificing the immediate user feedback during editing.

Although a new machine code implementation is generated at each edit, externally accessible state (including values of **param** and named **history** objects, as well as external bindings to **buffer** objects) is preserved across edits to make a friendlier interface.

The small set of *gen*~ operators can implement a vast number of algorithms without requiring extensive software development expertise. Developing with *gen*~ avoids many headaches of what would otherwise require development of C exter-

Algorithm	**Instances**		μ	σ	max	σ
Single-addition	100	MSP	5.0%	0.53	10%	0.45
		Gen	5.7%	0.39	11%	0.49
Multi-addition	20	MSP	11.4%	0.26	22%	0.67
		Gen	2.8%	0.05	7%	0.49
Dual FM	20	MSP	22.1%	0.62	37%	1.10
		Gen	12.7%	0.26	23%	0.50
Sinc interpolator	10	MSP	27.4%	0.25	46%	1.97
		Gen	8.5%	0.18	18%	0.70

Table 6.1.2: Performance comparison of MSP and Gen. In both cases, a *poly~* object is used to host several instances of the algorithm (with multi-threading disabled). Test results are measured over 10 runs of 30 seconds each. Tests measure the percentage of available CPU time used by the algorithm (lower is better), in which μ represents the arithmetic mean of all runs, max is the peak over all runs, and σ is the population standard deviation in each case. The single-addition patcher contains one addition object connected to the patcher inlet and outlet. The multi-addition patcher contains 20 fully connected addition objects (a total of 60 patch-cords). The dual-FM patcher implements a two-modulator, two-carrier frequency-modulated (FM) oscillator. The sinc interpolator patcher performs 16-point sinc interpolation of a waveform. MSP and Gen patchers are constructed as close to identical as possible. Tests were performed at a sampling rate of 44.1kHz, with a signal vector size of 256 samples on a 2GHz Intel Core i7 MacBook Pro, 4GB RAM, running OS X 10.6.8, using Max/MSP/Jitter 6.0.7.

nals: no need for a separate, cumbersome development environment, C programming skills, difficulties in debugging, portability and longevity issues.

Implementations of complex algorithms can be easily shared and distributed, offering immediate portability between the operating systems Max supports, potentially extending their longevity, and supporting collaborative work in the community. Re-implementations of existing black-box Max objects using *gen~* also become available for others to extend and modify (the distribution includes many such examples).

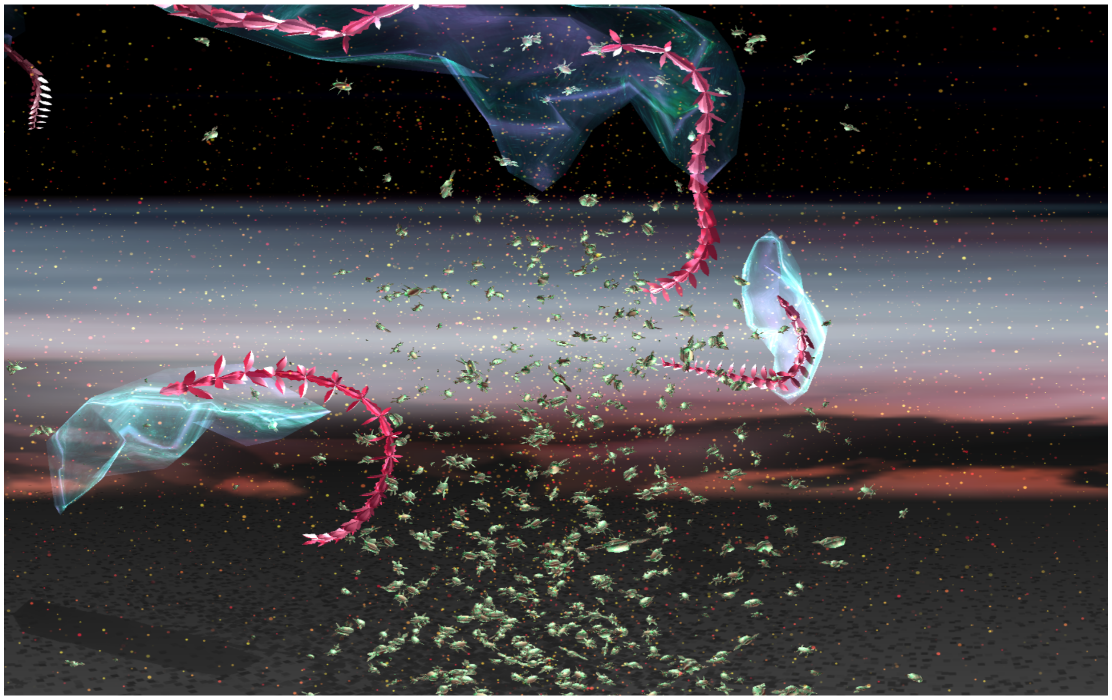


Figure 6.1.2: Screenshot of *Time of Doubles* showing populations of two major species of artificial organisms (image by author).

6.1.3 Time of Doubles

Time of Doubles is the third in a series of art installations involving immersive, interactive ecosystems, exhibited since by 2008 by Haru Ji and the author[6]. Dynamic compilation has been used in the software development of these *"artificial nature"* projects since 2010, to support rapid development in a form of remote debugging. However the *Time of Doubles: Flux* installation described in Chapter 5 was the first in the series to fully utilize meta-programming and dynamic compilation for a richer and more open-ended world of ecosystemic evolution. Prior installations in the series were not able to support such a wide diversity, due to the efficiency demands of real-time, interactive, high-resolution rendering.

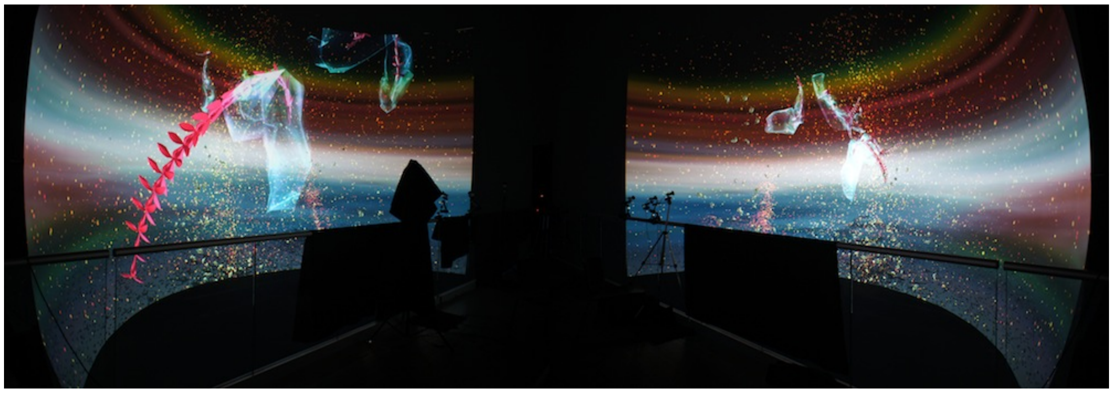
Time of Doubles was already running at near machine capacity, utilizing sev-

[6]http://www.artificialnature.net/ (accessed 2012)

eral parallel threads to manage graphical rendering, audio rendering, agent simulation, fluid / field simulation, human input data processing, resource monitoring, etc. Many prototype experiments turned out not to be viable once integrated to the full project, as the accumulated compilation tasks triggered by sudden explosions in population size would interrupt rendering. The solution taken was to also move the genetic processing and dynamic compilation into a background thread. In order to gracefully handle the added latency, newborn organisms act as largely inert 'eggs' until their machine code program is ready (on average only a handful of milliseconds, and only perceptible during large population explosions).

With this design, good performance targets were met. For example, using behavior functions with 40 intermediate operations, a population size of 2,000 concurrently active, unique organisms, with on average 30 newly reborn behavior functions generated per second, an uninterrupted frame-rate of over 50 FPS (observed for more than ten minutes) was obtained on a 2GHz Intel Core i7 MacBook Pro, with 4GB RAM, running OS X 10.6.8. This comfortably leaves time to spare for the other simulated and rendered objects, and should scale comfortably up with more powerful hardware. It easily satisfied the stringent performance demands of the AlloSphere space (as detailed in the Introduction), as tested on a 3.4GHz 8-core Intel Core i7 PC, 8 GB RAM, running Ubuntu 12.04, rendering with an nVidia Quadro 4000.

This version of *Time of Doubles* is due to be exhibited at the Daejon Museum of Art in the exhibition Energy, from September thru November 2012, with three more exhibitions scheduled internationally for 2012-2013. It is a periodic exhibit at the CNSI AlloSphere, University of California Santa Barbara (see Figure 6.1.3), where it is hoped to be extended to "full-dome" projection (spreading 12 or more projectors) during the Fall of 2012.



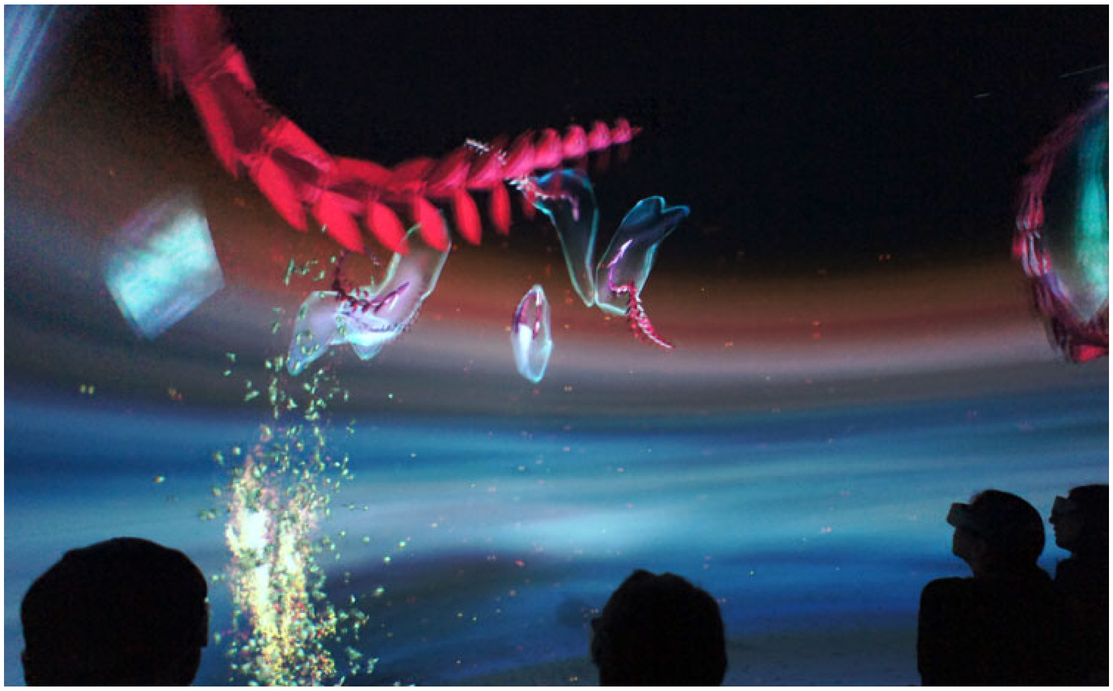


Figure 6.1.3: The *Time of Doubles* installation running on four full-HD projectors in the CNSI AlloSphere facility in stereoscopic 3D. Images by author.

6.2 Additional results

6.2.1 Explicit or implicit compilation

An important goal of creative interfaces is that implementation-specific details should not distract exploration, nor define the resultant artworks. For example, where many audio authoring environments (such as SuperCollider and CSound) make explicit reference in the interface to the creatively irrelevant implementation details of 'control' or 'block' rate processing, this becomes entirely implicit in both LuaAV *audio.Def* and Max/MSP *gen~*.

For systems utilizing dynamic compilation, an important question is to what degree it should occur implicitly. The entire compilation workflow is itself a feature that should not distract creative application, to the degree this is possible.

We saw in Section 2.3.3 that many systems and languages utilized fully implicit compilation, in many cases relying on an interpreter in the default case and replacing parts of a running program with compiled equivalents where this is relevant and supported. However, researchers noted that the choice of when and where to compile and inline is far more critical than the method of compilation used (Holzle, 1994). In domain-specific situations of audio-visual arts, we can give very strong hints in advance regarding which parts are efficiency constrained. Furthermore, if a function describes performance critical operations such as audio signal generation, interpretation may simply not feasible at all; lazy compilation is a general solution that might not always find an optimal fit for domain-specific problems.

Invocation of compilation is transparent in *gen~*, occurring at each structural edit[7], and some state (named parameters) is carried over between voices to pre-

[7]Auto-compilation can be disabled if desired.

serve the illusion of continuity. It appears implicit to user experience, however there remains an explicit separation between the host Max patcher and the Gen sub-patcher. Given that the sub-patcher is also a domain with slightly different semantics, this separation has cognitive advantages to support it.

On the other hand, invocation was made explicit in LuaAV through the call to **Def{}**. This puts the division of the audio scene by replaceable components (instruments) into the hands of the user. Although this is less implicit, it permits more flexible and generative approaches to synthesis algorithm construction and better performance with rapidly changing scenes for applications of granular synthesis.

The author explored more implicit interface models for LuaAV, matching the ugen-oriented style of its predecessor Vessel, however it is very difficult to do so while maintaining reliable high performance in all cases. For example, the author experimented with an alternative implementation of *audio.Def* generating code for LuaJIT[8] rather than for LLVM, and found performance to be generally very positive, in some cases matching LLVM performance, but far less predictable and more difficult to control. Nevertheless, insights from these experiments contributed toward development of a faster implementation of *Gibber*, a browser-based musical live-coding environment by Charlie Roberts, which generates code to target JavaScript tracing JIT compilers available in modern browsers, incrementally replacing sub-sections as the signal processing graph evolves.[9]

In *Time of Doubles*, all programming concerns are fully concealed from the

[8]The LuaJIT project is an implementation of Lua implementing an implicit tracing just-in-time compiler to adaptively optimize the hot paths of execution through a running program. It is designed as a drop-in replacement for Lua, requiring no code changes. In certain cases, LuaJIT can achieve speeds comparable to C. However, as with all trace compilers, the results can be somewhat unpredictable, and not all code of interest can be compiled; such cases fall back to the interpreter. See http://www.luajit.org/ (accessed 2012).

[9]http://charlie-roberts.com/gibber/ (accessed 2012).

spectators, while from the point of view of the developer compilation is entirely explicit, offering some degree of predictability within an intrinsically unpredictable task domain.

6.2.2 Fluidity and portability of language

The relative independence of language interface and run-time implementation was noted in Chapter 3. A parser written for a well-defined *model* language can be attributed with semantic actions to produce code for another well-defined *target* language, in a form of *source-to-source translation* (provided that the capabilities of target language are a superset of the capabilities of model). There are several implications:

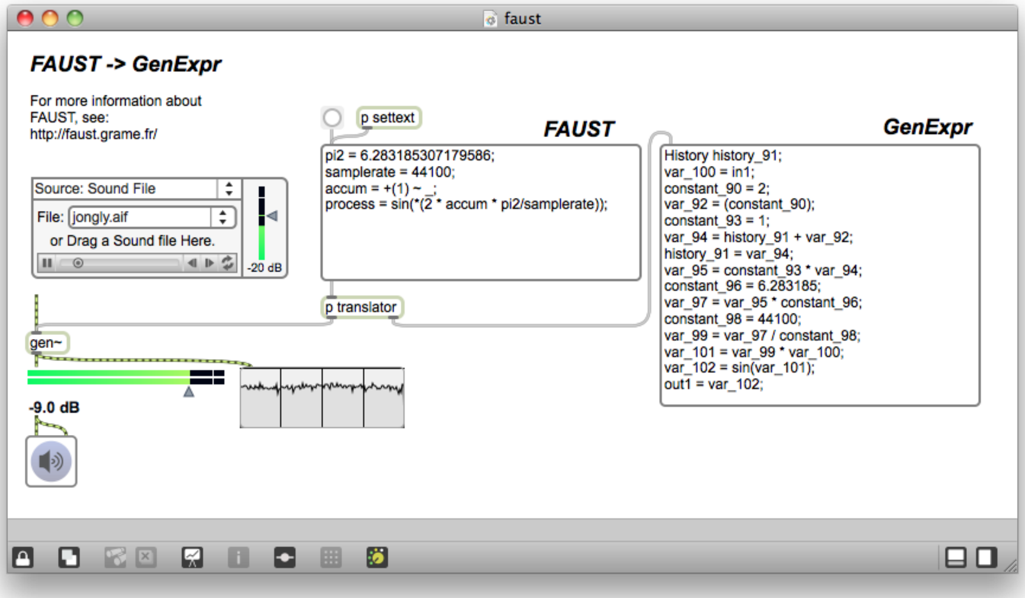
1. Artworks specified in the model language can be used within a larger infrastructure than was previously possible, or even outlive the viability of the original software.[10]

2. New domain- and even user-specific languages can be written over an existing system. Thor Magnusson for example designed a terse but expressive language specifically for live-coding, describing it as a parasite of the SuperCollider language on which it was implemented (Magnusson, 2011).

3. Interface and language design can be actively researched and prototyped in a live environment; modifying the language can be easier than modifying the implementation.

4. Alternatively, a new implementation can achieve wider acceptance by utilizing a language interface that is already familiar to users.

[10]Such as resurrecting 'dead' systems, a form of *virtualization*.



```
pi2 = 6.283185307179586;
samplerate = 44100;
accum = +(1) ~ _;
process = sin(*(2 * accum * pi2/samplerate));
```

```
History history_91;
var_100 = in1;
constant_90 = 2;
var_92 = (constant_90);
constant_93 = 1;
var_94 = history_91 + var_92;
history_91 = var_94;
var_95 = constant_93 * var_94;
constant_96 = 6.283185;
var_97 = var_95 * constant_96;
constant_98 = 44100;
var_99 = var_97 / constant_98;
var_101 = var_99 * var_100;
var_102 = sin(var_101);
out1 = var_102;
```

Figure 6.2.1: Translating FAUST code to GenExpr on-the-fly.

For example, the author constructed a completely new parser for the *FAUST* audio synthesis language (Orlarey et al., 2004), whose semantic actions produce valid GenExpr code for *gen*~. By means of this translation, the FAUST code can be edited and auditioned on the fly (a feature not yet supported with the standard FAUST compiler), while also fully integrated with the facilities of the host Max environment (see Figure 6.2.1).

The author also implemented a parser for *CSound* orchestra definitions (Boulanger, 2000), whose semantic actions produce valid data structures (rather than text) for LuaAV *audio.Def* instruments (see Listing 6.3). This allows, for example, a generative approach to CSound orchestras at run-time, which is impractical using standard CSound tools. It would be trivial to also write a parser for the *CSound* score file format, generating Lua code in which notes are triggered accurately by the LuaAV scheduler.

Listing 6.3: Parsing CSound orchestra definitions to produce LuaAV instruments.

```
local csound = require "audio.csound"
-- parse the CSound orchestra code
-- to produce LuaAV instruments
-- and store them in 'orc':
local orc = csound.parse([[
    instr synth1
        asig    oscil 0.5, p4
        asqr = asig * asig
                out asqr
    endin
    instr synth2
        asig    phasor p4
        afilt   lowpass2 asig, 400, 10
                out (afilt * 0.5)
    endin
]])
-- create two note events:
orc.synth1{ p3 = 0.5, p4 = 440 }
orc.synth2{ p3 = 0.75, p4 = 660 }
```

Both parsers were written in Lua using the LPeg parsing expression grammar library (Ierusalimschy, 2009b).

6.2.3 Potential points of failure

In prototyping self-rewriting systems for artworks, the author encountered several potential points of failure that need to be taken into account. Generated code can produce infinite loops, bad memory accesses, and other inconsistent states causing a freeze or crash in the host system. A viable workaround for infinite loops in procedural code is to add a loop counter to every generated loop and abort the loop if the counter exceeds an unreasonable count. Memory accesses can be guarded by providing run-time bounds checking where compile-time range analysis is not possible. Alternatively, memory addressing consistency can be enforced by more radical strategies, such as the address modulo of Core Wars red code (Rasmussen et al., 1990), or template matching of Tierra (Ray, 1997).

Both loop-abort counters and memory bounds checking have been utilized in the implementation of *gen*~.

Dynamically compiled code occupies memory. When new code fragments are continuously added at run-time this memory must eventually be recycled to prevent saturation. However, memory used for machine code must not be recycled as long as any active parts of the program can reach the functions it contains. Reference counting or more elaborate garbage collection schemes are needed. In *gen*~, a direct reference-counting system is utilized. In LuaAV and *Time of Doubles*, a combination of reference counting and Lua's built-in garbage collector are used. In all cases additional care was needed, since generated machine code is often being utilized in a different thread from which it is generated and recycled.

Speculating further, more powerful self-rewriting systems could pose more complex problems. A system could rewrite itself into a corner, such as an evolutionary system becoming trapped on a local fitness peak. To a certain extent maintaining interactive influence can help diffuse this possibility, and other techniques standard to evolutionary computing (such as co-evolution) can be applied to avoid an overly stabilized system. Similarly, a rewriting system could rewrite itself into a non-rewriting system, just as a self-interpreter can evolve into a fixed program by overwriting the functionality of *eval*. Preserving certain features as immutable may be necessary for stability in any case. Furthermore it may be possible for a system to bifurcate into islands that can no longer make any causal connection. Forcing the tethering of all active processes with some link back to a host environment may circumvent this possibility. Addressing these issues is an important issue for future work. The time-critical nature of real-time systems and installations have been strongly emphasized in this document, but the prevention of total loss of control is just as vital.

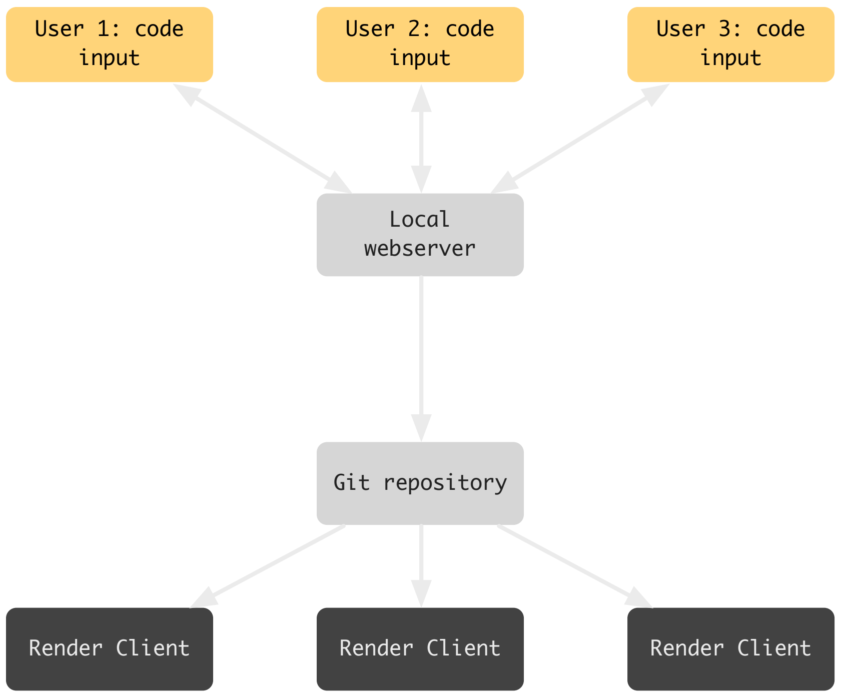


Figure 6.2.2: Live meta-programming in a multiple user, multiple renderer immersive space such as the AlloSphere at the University of California, Santa Barbara. Multiple users can edit and enter code fragments, via browser-based editors hosted by a local web server. A Git repository is used to handle merges and conflicts, pushing new edits to the repository, and notifying multiple renderer clients of the need to pull the latest changes.

6.2.4 Continuing work

The author and Charles Roberts have begun prototyping how real-time meta-programming for authoring environments can be applied to intensely demanding multimedia environment of the AlloSphere at the University of California, Santa Barbara. The prototype utilizes a browser-based, client/server development interface to enable multiple users to edit running applications concurrently, with a local Git repository to handle merges and conflicts. The motivation is to support an novel scenario of groups of artists and scientists working together, with the ability to open a URL on their laptops and immediately see and edit the code of the simulation as it is running all around them, without any delay or loss of continuity (see Figure 6.2.2).

6.3 Future speculations

> "Speculative software has been suggested as being software that explores the potentiality of all possible programming... Software, part of whose work is to reflexively investigate itself as software... Speculative software can be understood as opening up a space for the reinvention of software by its own means."[11]

This dissertation has demonstrated the value of reflective meta-programming and dynamic compilation for interactive computational arts, but does not fully define its potential.

As dynamic compilation becomes more readily available to contemporary systems, and dynamic languages become faster and more flexible, perhaps static compilation will become the rarity rather than the norm. For example, the View Points Research Institute, headed by Object-Oriented Programming pioneer Alan Kay, propose a model of 'late-bound everything' (run-time reflection and dynamic flexibility from the ground up), as a necessary step toward the reinvention of programming (Kay et al., 2008). A prototype system is pervasively built upon dynamically generated machine code for object models that are fully self-describing, giving end-users control over the environment 'to the metal'. At any point a program may redefine everything from the structure, grammar, even to the instruction set (Piumarta and Warth, 2008): the system becomes a 'laboratory of its own redevelopment'. There may be fascinating implications for the interactive computer arts.

Furthermore, the computers of our not-so-distant future may have potentially thousands of processing cores on a single chip. The current convenience of instan-

[11]Matthew Fuller, Behind the Blip, http://amsterdam.nettime.org/Lists-Archives/nettime-l-0201/msg00025.html (accessed 2012).

taneously shared memory cannot scale to this situation, implying a new trade-off of determinism and performance (synchronizing memory in distant cores can outweigh the benefit of parallel processing). David Ungar (among others) has suggested that we need new programming paradigms of many locally reified fast "kernels" interacting within a less predictable soup, fragmenting tasks into ensembles of granular agents (actors) with redundancy and adaptive characteristics.[12] Such paradigms are remarkably similar to the speculations of meta-programmed artificial life.

Multi-user, dynamic rewriting of code also has a social impact. The popularity of user-configurable, open-ended and sharable content in games such as *The Sims*[13] and *Little Big Planet*[14] suggest a provocative bridge between general consumer products and the kinds of systems discussed in this dissertation. The efficiently open-ended creativity support tools and generative artworks of today could become a significant content industry of tomorrow.

The internet browser is becoming an increasingly viable medium for rich and complex media arts, particularly since HTML 5.0 introduced standards for synthesized audio and 3D graphics in the browser. Dynamic compilation in the browser has been supported since Java's HotSpot JVM (Aycock, 2003), but more recently leveraging recent advances in implicit just-in-time compilation of JavaScript (Kienle, 2010). Furthermore, language-independent support for dynamic compilation has been made possible using the Native Client sandbox (Ansel et al., 2011), effectively allowing dynamic compilation of C and other languages to machine code during the normal client-side operation of a web page on supported

[12]http://www.cmu.edu/silicon-valley/news-events/seminars/2011/ungar-talk.html (accessed 2012).

[13]http://thesims.com/en_us/home (accessed 2012).

[14]http://www.littlebigplanet.com/ (accessed 2012).

browsers, while maintaining important security guarantees.[15]

In theory, developing art for the browser rather than the desktop provides instant accessibility to a vast audience, portability across platforms, and ease of integration with distributed and multi-user "cloud" applications. For example, it raises the tantalizing possibility of a generative artwork which 'lives' online and never stops running, with cells appearing and disappearing on various client machines from time to time, constantly evolving. Multi-cellular organisms replace their cells (and atoms) many times in a lifetime, but "the only human artifact that is remotely like this is the Internet, which has been able to grow and replace most parts large and small without having to ever be stopped."[16] The idea of a massively distributed biological simulation over the internet has been explored before in Artificial Life[17], but not with the rich media processing possible today. A self-rewriting, diversifying online artwork unfolds a unique story, bearing fascinating implications for old questions of art in the age of mechanical reproduction (Benjamin, 2008).

[15]The only requirement is that generated code conform to NaCl's code safety verifications, such as aligned instructions. Ansel et al. noted a 25% to 60% slowdown incurred by the sandboxing safety requirements compared to untrusted code, but this was easily outweighed by the speedups of using dynamically compiled versus interpreted code (generally more than 10x faster).

[16]Alan Kay, in a message posted to the VPRI *Fundamentals of New Computing* mailing list, Jun 2011 (http://vpri.org/mailman/listinfo/fonc, accessed 2011).

[17]Thomas Ray's 'Network Reserve' proposal can be read at http://life.ou.edu/pubs/reserves/node1.html#SECTION00010000000000000000 (accessed 2012).

Bibliography

Adami, C. (2008). Ab Initio Modeling of Ecosystems with Artificial Life. *Natural Resource Modeling 15*(1).

Aho, A., R. Sethi, and J. Ullman (1986). *Compilers, principles, techniques, and tools.* Addison-Wesley series in computer science. Addison-Wesley Pub. Co.

Altenberg, L. (1994). The evolution of evolvability in genetic programming. In *Advances in Genetic Programming.* MIT Press.

Amatriain, X., J. Castellanos, T. Höllerer, and J. Kuchera (2008). Experiencing Audio and Music in a Fully Immersive Environment. In R. Kronland-Martinet, S. Ystad, and K. Jensen (Eds.), *Computer Music Modeling and Retrieval. Sense of Sounds*, pp. 380–400. Berlin, Heidelberg: Springer-Verlag.

Amatriain, X., J. Kuchera-Morin, T. Hollerer, and S. T. Pope (2009). The AlloSphere: Immersive Multimedia for Scientific Discovery and Artistic Exploration. *IEEE Multimedia 16*(2), 64–75.

Amelang, D. (2008). Jitblt: efficient run-time code generation for digital compositing. Master's thesis, University of California San Diego.

Anckaert, B. and M. Madou (2007). A model for self-modifying code. In *Proceed-*

ings of the 8th international conference on Information hiding, IH'06, Berlin, Heidelberg, pp. 232–248. Springer-Verlag.

Ansel, J., P. Marchenko, U. Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. L. Biffle, and B. Yee (2011, June). Language-independent sandboxing of just-in-time compilation and self-modifying code. In *PLDI '11: Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. ACM Request Permissions.

Arbab, F. (2006). Computing and Interaction. In S. S. D. Goldin and P. Wegner (Eds.), *Interactive Computation: The New Paradigm*. Springer-Verlag.

Armstrong, J. (2007). A history of Erlang. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III. ACM Request Permissions.

Arnold, M., S. Fink, D. Grove, M. Hind, and P. Sweeney (2005). A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE 93*(2), 449–466.

Ascott, R. (1964). Behaviourist art and the cybernetic vision. *Cybernetica: journal of the International Association for Cybernetics 9*, 247–264.

Ashby, W. (1956). *An introduction to cybernetics*. London, UK: Chapman and Hall.

Aycock, J. (2003). A brief history of just-in-time. *ACM Computing Surveys (CSUR) 35*(2), 97–113.

Barricelli, N. (1957). *Symbiogenetic Evolution Processes Realized by Artificial Methods*.

Bedau, M., J. McCaskill, N. Packard, and S. Rasmussen (2000). Open Problems in Artificial Life. *Artificial Life 6*(4), 363–376.

Beer, S. (1994). *Brain of the Firm.* Stafford Beer Classic Library. John Wiley & Sons.

Benabou, M. (1983). La regle et la contrainte. *Practiques 39.*

Bencina, R. (2011). Inside scsynth. In S. Wilson, N. Collins, and D. Cottle (Eds.), *The SuperCollider Book*, Chapter 26. The MIT Press.

Benjamin, W. (2008). The work of art in the age of mechanical reproduction. In E. Ihnatowicz (Ed.), *Artist and computer*, pp. 114–115. Penguin Books Limited.

Blackwell, A. and N. Collins (2005). The programming language as a musical instrument. In *Psychology of Programming Interest Group.*

Boden, M. (2007). Creativity: how does it work? In *Proceedings of the Creativity: Innovation and Industry Conference.* Creativity East Midlands.

Boden, M. and E. Edmonds (2009). What is generative art? *Digital Creativity 20*(1), 21–46.

Boswell, R. (2008). *The Half-Known World: On Writing Fiction.* Graywolf.

Boulanger, R. (2000). *The Csound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing, and Programming.* Mit Press.

Bousquet, F. and C. Lepage (2004). Multi-agent simulations and ecosystem management: a review. *Ecological Modelling 176*(3-4), 313–332.

Bracha, G. and D. Ungar (2004). Mirrors: design principles for meta-level facilities of object-oriented programming languages. *ACM SIGPLAN Notices 39*(10), 331–344.

Brodu, N. (2006). A synthesis and a practical approach to complex systems. *Complexity 15*(1), 36–60.

Brooks, S. and B. Ross (1996). Automated composition from computer models of biological behavior. *Leonardo Music Journal 6*, 27–31.

Brown, A. R. and A. Sorensen (2009). Interacting with Generative Music through Live Coding. *Contemporary Music Review 28*(1), 17–29.

Carmack, J. (2007). Plan (1999). Technical report, id Software.

Cascone, K. (2000). The Aesthetics of Failure: Post-Digital Tendencies in Contemporary Computer Music. *Computer Music Journal 24*(4), 12–18.

Church, A. (1941). *The Calculi of Lambda Conversion.* Princeton University Press, Aberdeen, Maryland.

Cohen, H. (1995). The further exploits of AARON, painter. *Stanford Humanities Review 4*(2), 141–158.

Collins, N., A. Mclean, J. Rohrhuber, and A. Ward (2003). Live coding in laptop performance. *Organised Sound 8*(03), 10.

Consel, C., L. Hornof, F. Noël, J. Noyé, and N. Volanschi (1996). A uniform approach for compile-time and run-time specialization. In *Selected Papers from the Internaltional Seminar on Partial Evaluation*, London, UK, pp. 54–72. Springer-Verlag.

Conway, M. E. (1958). Proposal for an UNCOL. *Communications of the ACM 1*(3).

Cope, D. (1992). Computer modeling of musical intelligence in experiments in musical intelligence. *Computer Music Journal 16*(2), 69–83.

Costa, A. and G. Dimuro (2005). Interactive Computation: Stepping Stone in the Pathway From Classical to Developmental Computation. *Electronic Notes in Theoretical Computer Science 141*(5), 5–31.

Cox, G., A. Mclean, and A. Ward (2004). Coding Praxis: Reconsidering the Aesthetics of Code. *Goriunova, Olga and Shulgin, Alexei (ed.): readme–Software Arts and Cultures, Edition*, 160–175.

Dannenberg, R. (2005). Design patterns for real-time computer music systems. In *Proceedings of the International Computer Music Conference*, ICMC '05. IMCA.

Davis, M. (2006). The church-turing thesis: Consensus and opposition. In *Proceedings of the Second conference on Computability in Europe: logical Approaches to Computational Barriers*, CiE'06, Berlin, Heidelberg. Springer-Verlag.

de Campo, A. (2004). Waiting and Uncertainty in Computer Music Networks. In *Proceedings of the International Computer Music Conference*, ICMC '04. ICMA.

de Campo, A. (2005). Algorithms today notes on language design for just in time programming. In *Proceedings of International Computer Music Conference 2005*. ICMA.

DeLanda, M. (2011). *Philosophy and Simulation: The Emergence of Synthetic Reason*. Bloomsbury.

Deutsch, L. and A. Schiffman (1984). Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, New York, pp. 297–302. ACM.

Dietrich, F. (1986). Visual intelligence: the first decade of computer art (1965-1975). *IEEE Computer Graphics and Applications 5*(7), 33–45.

Dittrich, P., J. Ziegler, and W. Banzhaf (2001, July). Artificial Chemistries—A Review. *Artificial Life 7*(3), 225–275.

Dorin, A. (2005). Enriching Aesthetics with Artificial Life. In *Artificial Life Models in Software*, pp. 415–431. New York: Springer-Verlang.

Draves, S. (1996). Compiler generation for interactive graphics using intermediate code. In *Selected Papers from the Internaltional Seminar on Partial Evaluation*, London, UK, pp. 95–114. Springer-Verlag.

Draves, S. (1997). *Automatic program specialization for interactive media.* Ph. D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA.

Draves, S. (1998). Partial evaluation for media processing. *ACM Computing Surveys (CSUR) 30*(3es).

Elliott, C., O. De Moor, and S. Finne (1999). Efficient Image Manipulation via Run-time Compilation. Technical report, Microsoft Research.

Engelbart, D. (1962). Augmenting human intellect: A conceptual framework. summary report contract af 49 638 1024 sri project 3578. Technical report, Stanford Research Institute, Stanford, USA.

Etxeberria, A. (2002). Artificial Evolution and Lifelike Creativity. *Leonardo 35*(3), 275–281.

Fischer, G. and E. Scharff (2000). Meta-design: design for designers. In *Proceedings of the 3rd conference on Designing Interactive Systems: processes, practices, methods, and techniques*, DIS '00, New York, pp. 396–405. ACM.

Fisher, D. A. (1972, November). A survey of control structures in programming languages. *ACM SIGPLAN Notices 7*(11), 1–13.

Fontana, W., G. Wagner, and L. Buss (1994). Beyond digital naturalism. *Artificial Life 1*(1-2), 211–228.

Ford, B. (2004). Parsing expression grammars: a recognition-based syntactic foundation. *ACM SIGPLAN Notices 39*(1), 111–122.

Franz, M. and T. Kistler (1997). Slim binaries. *Communications of the ACM 40*(12), 87–94.

Gabriel, R. P. and K. J. Sullivan (2010). Better Science Through Art. In *Proceedings of the ACM international conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pp. 885–900.

Gal, A., B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, and J. Orendorff (2009). Trace-based just-in-time type specialization for dynamic languages. *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, 465–478.

Gal, A. and M. Franz (2006). Incremental dynamic code generation with trace trees. Technical report, University of California, Irvine.

Galanter, P. (2003). What is Generative Art? Complexity theory as a context for art theory. In *Proceedings of the 6th Generative Art Conference*, GA '03.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1993). Design Patterns: Abstraction and Reuse of Object-Oriented Design. pp. 406–431. Springer Berlin / Heidelberg.

Giavitto, J., C. Godin, O. Michel, and P. Prusinkiewicz (2002). Computational models for integrative and developmental biology. *LaMI Rapport de Recherche*.

Giavitto, J. and O. Michel (2001). MGS:: A Rule-Based Programming Language for Complex Objects and Collections. *Electronic Notes in Theoretical Computer Science 59*(4), 286–304.

Goldin, D. and P. Wegner (2008). The Interactive Nature of Computing: Refuting the Strong Church–Turing Thesis. *Minds and Machines 18*(1), 17–38.

Green, T. (2000). Instructions and descriptions. In *Proceedings of the working conference on Advanced Visual Interfaces*, AVI '00, New York, pp. 21–28. ACM.

Green, T. R. G. and M. Petre (1996). Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing 7*, 131–174.

Hamilton, A. (2000). The art of improvisation and the aesthetics of imperfection= L'art de l'improvisation et l'esthétique de l'imperfection. *British journal of aesthetics 40*(1), 168–185.

Hansen, G. J. (1974). *Adaptive Systems for the Dynamic Run-time Optimization of Programs.* Defense Technical Information Center.

Harel, D. (2008). Can Programming Be Liberated, Period? *Computer 41*(1), 28–37.

Harris, P. (2000). *The Work of the Imagination*. Understanding Children's Worlds. Blackwell Publishers.

Hayles, N. K. (1998). How does it feel to be posthuman? In A. Broeckmann (Ed.), *The art of the accident*. V2 publishing.

Hein, H. (1972). The endurance of the mechanism—vitalism controversy. *Journal of the History of Biology 5*, 159–188.

Hewitt, C. and P. Jong (1986). Open systems. *ACM Transactions on Information Systems 4*(3), 271–287.

Hofstadter, D. (1979). *Godel, Escher, Bach: An Eternal Golden Braid*. Basic Books.

Holland, J. H. (1992). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Cambridge, MA, USA: MIT Press.

Holzle, U. (1994). *Adaptive optimization for self: reconciling high performance with exploratory programming*. Ph. D. thesis, Stanford University, Stanford, USA.

Hölzle, U. and D. Ungar (1994). A third-generation SELF implementation: reconciling responsiveness with performance. In *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, OOPSLA '94, New York, pp. 229–243. ACM.

Ierusalimschy, R. (2009a). Programming with Multiple Paradigms in Lua. In *Proceedings of the 18th international conference on Functional and Constraint Logic Programming*, WFLP'09, Berlin, Heidelberg, pp. 1–12. Springer-Verlag.

Ierusalimschy, R. (2009b, March). A text pattern-matching tool based on parsing expression grammars. *Softw. Pract. Exper. 39*(3), 221–258.

Ierusalimschy, R., L. De Figueiredo, and W. Celes (2005). The implementation of Lua 5.0. *Journal of Universal Computer Science 11*(7), 1159–1176.

Ierusalimschy, R., L. de Figueiredo, and W. Celes (2007). The evolution of Lua. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, New York, pp. 2–1–2–26. ACM.

Ierusalimschy, R., L. H. de Figueiredo, and W. C. Filho (1996, June). Lua–an extensible extension language. *Softw. Pract. Exper. 26*(6), 635–652.

Ikegami, T. (1999). Evolvability of machines and tapes. *Artificial Life and Robotics 3*(4), 242–245.

Ingalls, D. (1981). Design Principles Behind Smalltalk. *BYTE 6*(8).

Ingalls, D., T. Kaehler, J. Maloney, S. Wallace, and A. Kay (1997). Back to the future: the story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '97, New York. ACM.

Ji, H. (2011). *Artificial Natures: Creating Nature-Like Aesthetic Experiences through Immersive Artificial Life Worlds*. Ph. D. thesis, University of California Santa Barbara.

Kamin, S. (2003). Routine run-time code generation. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, New York, pp. 208–220. ACM.

Kampis, G. (1993). *Life-like computing beyond the machine metaphor.* London: Chapman and Hall.

Kampis, G. (1996). Self-modifying systems: a model for the constructive origin of information. *BioSystems 38*(2-3), 119–125.

Kasabov, N., I. Sidorov, and D. Dimitrov (2005). Computational intelligence, bioinformatics and computational biology: A brief overview of methods, problems and perspectives. *Journal of Computational and Theoretical Nanoscience 2*(4), 473–491.

Kauffman, S. (2002). *Investigations.* Oxford University Press, USA.

Kay, A., D. Ingalls, Y. Ohshima, and I. Piumarta (2008). Proposal to NSF–Granted on August 31st 2006 Steps Toward The Reinvention of Programming A Compact And Practical Model of Personal Computing.

Kelsey, R. (1995). A correspondence between continuation passing style and static single assignment form. *Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*, 22.

Kennedy, A. (2007). Compiling with continuations, continued. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming.* ACM Request Permissions.

Kiczales, G. and J. Des Rivieres (1991). *The Art of the Metaobject Protocol.* Cambridge, MA, USA: MIT Press.

Kienle, H. (2010). It's About Time to Take JavaScript (More) Seriously. *Software, IEEE 27*(3), 60–62.

Kitano, H. (1998). Building complex systems using developmental process: An engineering approach. In *Proceedings of the Second International Conference on Evolvable Systems: From Biology to Hardware*, ICES '98, London, UK, pp. 218–229. Springer-Verlag.

Knowlton, K. (1976). Ken knowlton. In E. Ihnatowicz (Ed.), *Artist and computer*, pp. 65–67. Harmony Books.

Koza, J. (1990). Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems. Technical report, Stanford University, Stanford, USA.

Koza, J. R. (1991, February 1990). Genetic evolution and co-evolution of computer programs. In C. T. C. Langton, J. D. Farmer, and S. Rasmussen (Eds.), *Artificial Life II*, Volume X of *SFI Studies in the Sciences of Complexity*, pp. 603–629. Santa Fe Institute, New Mexico, USA: Addison-Wesley.

Langton, C. (1990). Computation at the edge of chaos: Phase transitions and emergent computation. *Physica D: Nonlinear Phenomena 42*(1-3), 12–37.

Langton, C. G. (1996). Artificial Life. In M. Boden (Ed.), *The Philosophy of Artificial Life*. Oxford: Oxford University Press.

Lattner, C. and A. Vikram (2007). Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization*, CGO '04, Washington DC, USA, pp. 75–. IEEE Computer Society.

Lazzarini, V. (2005). Extensions to the Csound Language: from User-Defined to Plugin Opcodes and Beyond. In *Proceedings of the Linux Audio Conference*, LAC '05, pp. 13.

Lee, E. (2009). Computing needs time. *Communications of the ACM 52*(5), 70–79.

Licklider, J., B. Beranek, N. Inc, and M. Cambridge (1960). Man-Computer Symbiosis. *IRE Transactions on Human Factors in Electronics 1*, 4–11.

Longuet-Higgins, C. (1969). What biology is about. In C. H. Waddington (Ed.), *Towards a theoretical biology. 2: Sketches*. Chicago: Aldine.

Lotka, A. J. (1910). Contribution to the Theory of Periodic Reactions. *The Journal of Physical Chemistry 14*(3), 271–274.

Machado, P., J. Tavares, A. Cardoso, and F. Pereira (2004). Evolving Creativity. In *Computational Creativity Workshop, 7th European Conference in Case Based Reasoning*, Madrid, Spain.

MacKenzie, I. S. and C. Ware (1993). Lag as a determinant of human performance in interactive systems. In *Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*, CHI '93, New York, NY, USA, pp. 488–493. ACM.

Maeda, J. (2001). *Design By Numbers*. Mit Press.

Magnusson, T. (2002). Processor Art: Currents in the Process Oriented Works of Generative and Software Art. Master's thesis.

Magnusson, T. (2005). IXI software: the interface as instrument. In *Proceedings of the 2005 conference on New interfaces for musical expression*, NIME '05, Singapore, pp. 212–215. National University of Singapore.

Magnusson, T. (2006). Affordances and constraints in screen-based musical instruments. In *Proceedings of the 4th Nordic conference on Human-computer interaction: changing roles*, NordiCHI '06, New York, pp. 441–444. ACM.

Magnusson, T. (2011). Ixi lang: a supercollider parasite for live coding. In *Proceedings of the International Computer Music Conference*, ICMC '11, Huddersfield, UK. International Computer Music Association.

Malenfant, J., M. Jacques, and F. Demers (1996). A tutorial on behavioral reflection and its implementation. In G. Kickzales (Ed.), *Proceedings of the Reflection'96 Conference*, Volume 96, San Francisco, USA, pp. 1–20.

Massalin, H. (1992). *Synthesis: An Effcient Implementation of Fundamental Operating System Services.* Ph. D. thesis, Columbia University, New York.

Mathews, M. (1961). An acoustic compiler for music and psychological stimuli. *Bell System Technical Journal*.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM 3*, 184–195.

McCartney, J. (2002). Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal 26*(4), 61–68.

McCormack, J. (2002). Evolving for the Audience. *International Journal of Design Computing 4*.

McCormack, J. (2005). Open Problems in Evolutionary Music and Art. In F. R. et al. (Ed.), *Proceedings of Applications of Evolutionary Computing, (EvoMUSART 2005), Lausanne, Switzerland*, Volume 3449 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, pp. 428–436. Springer-Verlag.

McCormack, J. (2007). Artificial ecosystems for creative discovery. In T. D. et. al. (Ed.), *Proceedings of the 9th annual conference on Genetic and Evolutionary Computation*, Volume 1 of *GECCO '07*, New York, pp. 301–307. ACM.

McCormack, J. and A. Dorin (2001). Art, Emergence, and the Computational Sublime. In *Proceedings of the Second International Conference on Generative Systems in the Electronic Arts*, Second Iteration, Victoria, Australia. Center for Electronic Media Art.

McCormack, J., P. McIlwain, A. Lane, and A. Dorin (2007). Generative composition with Nodal. In E. Miranda (Ed.), *Workshop on Music and Artificial Life*, ECAL 2007, Lisbon, Portugal.

McGranaghan, M. (2011). ClojureScript: Functional Programming for JavaScript Platforms. *Internet Computing, IEEE 15*(6), 97–102.

Mclean, A., D. Griffiths, N. Collins, and G. Wiggins (2010). Visualisation of Live Code. In *Proceedings of the 2010 international conference on Electronic Visualisation and the Arts*, EVA'10, Swinton, UK, pp. 26–30. British Computer Society.

McLean, C. (2011). *Artist-Programmers and Programming Languages for the Arts.* Ph. D. thesis, Goldsmiths College University of London.

McOwan, P. (2009). Sodarace: Continuing Adventures in Artificial Life. In *Artificial Life Models in Software*, pp. 97–111. Springer-Verlag.

Miller, J. and P. Thomson (2003). A developmental method for growing graphs and circuits. In *Proceedings of the 5th international conference on Evolvable systems: from biology to hardware*, ICES '03, Berlin, Heidelberg, pp. 93–104. Springer-Verlag.

Mitchell, J. G. (1970). *The design and construction of flexible and efficient interactive programming systems.* Ph. D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA.

Mullarkey, J. (2006). *Post-Continental Philosophy: An Outline.* Transversals (London). Continuum.

Nake, F. (2007). Computer art: creativity and computability. In *Proceedings of the 6th ACM SIGCHI conference on Creativity & cognition*, C&C '07, New York, pp. 305–306. ACM.

Neumann, J. V. (1966). *Theory of Self-Reproducing Automata.* Champaign, IL, USA: University of Illinois Press.

Novak, M. (1998). Next babylon: Accidents to play in. In A. Broeckmann (Ed.), *The art of the accident.* V2 publishing.

Orlarey, Y., D. Fober, and S. Letz (2004). Syntactical and semantical aspects of Faust. *Soft Computing-A Fusion of Foundations, Methodologies and Applications 8*(9), 623–632.

Papert, S. (1971). Teaching Children Thinking. *MIT AI Memo*.

Parr, T. J. (2004). Enforcing strict model-view separation in template engines. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04. ACM.

Peirce, C. S. (1931). *Collected Papers of Charles Sanders Peirce.* Harvard University Press.

Penny, S. (2010). Twenty years of artificial life art. *Digital Creativity 21*(3), 197–204.

Piumarta, I. and A. Warth (2006). Open, reusable object models. *Research Note RN-2006-003-a*.

Piumarta, I. and A. Warth (2008). Open, extensible object models. *Self-Sustaining Systems*, 1–30.

Pope, S. (1993). Machine tongues xv: Three packages for software sound synthesis. *Computer Music Journal 17*(2), 23–54.

Post, E. (1943). Formal reductions of the general combinatorial decision problem. *American journal of mathematics 65*(2), 197–215.

Potter, R. (1993). Just-in-time programming. In *Watch What I do: Programming by Demonstration*, pp. 513–526. Cambridge, MA, USA: MIT Press.

Prusinkiewicz, P. and A. Lindenmayer (1996). *The algorithmic beauty of plants.* New York, NY, USA: Springer-Verlag New York, Inc.

Puckette, M. (1996). Pure Data: another integrated computer music environment. In *Proceedings of the Second Intercollege Computer Music Festival.*

Puckette, M. (1997). Pure data: Recent progress. In *Proceedings of the Third Intercollege Computer Music Festival*, pp. 1–4.

Puckette, M. (2002). Max at Seventeen. *Computer Music Journal 26*(4), 31–43.

Puckette, M. (2004). A divide between 'compositional'and 'performative' aspects of PD. In *Proceedings of the first International PD Convention*.

Putnam, L., G. Wakefield, H. Ji, B. Alper, D. Adderton, and P. J. Kuchera-Morin (2010). Immersed in Unfolding Complex Systems. In J. Steele and N. Iliinsky (Eds.), *Beautiful Visualization*, Beautiful Series. O'Reilly Media.

Rasmussen, S., C. Knudsen, and R. Feldberg (1992). Dynamics of programmable matter. In C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen (Eds.), *Artificial Life II*, pp. 211–254. Addison Wesley.

Rasmussen, S., C. Knudsen, R. Feldberg, and M. Hindsholm (1990). The coreworld: Emergence and evolution of cooperative structures in a computational chemistry. *Physica D: Nonlinear Phenomena 42*(1-3), 111–134.

Ray, T. (1991). An approach to the synthesis of life. In C. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen (Eds.), *Artificial Life II*, Volume XI of *Santa Fe Institute Studies in the Sciences of Complexity*, pp. 371–408. Addison-Wesley.

Ray, T. (1997). Evolving complexity. *Artificial Life and Robotics 1*(1), 21–26.

Reas, C. and B. Fry (2006). Processing: programming for the media arts. *AI & SOCIETY 20*(4), 526–538.

Resnick, M., J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai (2009). Scratch: programming for all. *Communications of the ACM 52*(11), 60–67.

Rinaldo, K. (1998). Technology recapitulates phylogeny: Artificial life art. *Leonardo 31*(5), 371–376.

Risset, J. (1971). Synthesis of sound by computer and problems concerning timbre. In *Report on the Stockholm Meeting, "Music and Technology"*, La Revue Musicale. UNESCO.

Sandewall, E. (1978). Programming in an Interactive Environment: the "Lisp" Experience. *ACM Computing Surveys 10*(1), 35–71.

Scaletti, C. and R. Johnson (1988). An interactive environment for object-oriented music composition and sound synthesis. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '88, New York, pp. 222–233. ACM.

Scheirer, E. and B. Vercoe (1999). SAOL: The Mpeg-4 Structured Audio Orchestra Language. *Computer Music Journal 23*(2), 31–51.

Schmidt, D. C. (2006). Model-Driven Engineering. *Computer* (6), 1–7.

Shagrir, O. (1997). Two Dogmas of Computationalism. *Minds and Machines 7*(3), 321–344.

Shneiderman, B. (2007). Creativity support tools: accelerating discovery and innovation. *Communications of the ACM 50*, 20–32.

Shneiderman, B. (2009). Creativity support tools: A grand challenge for HCI researchers. *Engineering the User Interface*, 1–9.

Sims, K. (1994). Evolving virtual creatures. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '94, New York, pp. 15–22. ACM.

Smith, R. and D. Ungar (1995). Programming as an experience: The inspiration for Self. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, London, UK, pp. 303–330. Springer-Verlag.

Smith, W. and G. Wakefield (2009). Augmenting Computer Music with Just-In-Time Compilation. In *Proceedings of the International Computer Music Conference*, ICMC '09. ICMA.

Sommerer, C. and L. Mignonneau (2009). From Artificial Life and Complexity Research to Bio and Nano Art. *Interactive Art Research*, 1–72.

Sorensen, A. and H. Gardner (2010). Programming with time: cyber-physical programming with impromptu. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pp. 822–834. ACM.

Spafford, E. (1989). Computer viruses as artificial life. *Journal of Artificial Life 1*(1).

Stone, J. E., D. Gohara, and G. Shi (2010). OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering 12*(3), 66–73.

Sussman, G. (1975). Scheme: An interpreter for extended lambda calculus, MIT AI Memo 349.

Sussman, G. J. (2007). Building Robust Systems. An Essay.

Thieleman, H. (2010). Compiling Signal Processing Code Embedded in Haskell via LLVM. In *Proceedings of the 13th International Conference on Digital Audio Effects (DAFx-10), Graz.*

Thompson, K. (1968). Regular expression search algorithm. *Communications of the ACM 11*(6), 419–422.

Tissoires, B. and S. Conversy (2008). Graphic Rendering Considered as a Compilation Chain. In *Interactive Systems. Design, Specification, and Verification*, pp. 267–280. Berlin, Heidelberg: Springer-Verlag.

Tschudin, C. and L. Yamamoto (2006). Harnessing self-modifying code for resilient software. In *Proceedings of the Second international conference on Radical Agent Concepts: innovative Concepts for Autonomic and Agent-Based Systems*, Berlin, Heidelberg, pp. 197–204. Springer-Verlag.

Turchi, P. (2004). *Maps of the imagination: the writer as cartographer.* Trinity University Press.

Turing, A. (1937). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society 2*(1), 230.

Turing, A. (1952). The chemical basis of morphogenesis. *Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences 237*(641), 37.

Turing, A. M. (1939). Systems of Logic Based on Ordinals. In *Proceedings of the London Mathematical Society*, Volume 2, pp. 161–228.

Ungar, D. and R. Smith (1991). Self: The power of simplicity. *LISP and Symbolic Computation 4*(3), 187–205.

Valimaki, V., M. Karjalainen, Z. Janosy, and U. Laine (1992). A real-time DSP implementation of a flute model. In *IEEE Internationcal Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pp. 249–252.

Vilder, R. (1976). Roger vilder. In E. Ihnatowicz (Ed.), *Artist and computer*, pp. 114–115. Harmony Books.

Wakefield, G. (2007). A Platform for Computer Music Composition, Interleaving Sample-Accurate Synthesis and Control. Master's thesis, University of California Santa Barbara.

Wakefield, G. and H. Ji (2009). Artificial Nature: Immersive World Making. In *Proceedings of the EvoWorkshops 2009 on Applications of Evolutionary Computing*, EvoMUSART '09, Berlin, Heidelberg, pp. 597–602. Springer-Verlag.

Wakefield, G. and W. Smith (2007). Using lua for audiovisual composition. In *Proceedings of the 2007 International Computer Music Conference*, ICMC '07. ICMA.

Wakefield, G., W. Smith, and C. Roberts (2010). LuaAV: Extensibility and Heterogeneity for Audiovisual Computing. In *Proceedings of Linux Audio Conference*, LAC '10.

Wang, G. and P. Cook (2004). On-the-fly programming: using code as an expressive musical instrument. In *Proceedings of the 2004 conference on New Interfaces for Musical Expression (NIME)*, pp. 138–143.

Wegner, P. (2005). The church-turing thesis: Breaking the myth. *New Computational Paradigms*, 152–168.

Wegner, P. and D. Goldin (2003). Computation beyond turing machines. *Communications of the ACM 46*(4), 100–102.

Wheeler, M., S. Bullock, E. Paolo, J. Noble, and M. Bedau (2002). The view from elsewhere: Perspectives on alife modeling. *Artificial Life 8*.

Whitelaw, M. (2004). *Metacreation: art and artificial life.* MIT Press.

Whitelaw, M. (2005). System stories and model worlds: A critical approach to generative art. *Readme 100*, 135–154.

Whitelaw, M., M. Guglielmetti, and T. Innocent (2009). Strange ontologies in digital culture. *Computers in Entertainment (CIE) 7*(1), 4.

Wiggins, G. A. (2006). A preliminary framework for description, analysis and comparison of creative systems. *Knowledge-Based Systems 19*(7), 449–458.

Yee, B., D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar (2010). Native Client: a sandbox for portable, untrusted x86 native code. *Communications of the ACM 53*(1), 91–99.

YI, S. and V. Lazzarini (2012). Csound for Android. *Proceedings of the Linux Audio Conference (LAC).*

Youngblood, G. (1970). *Expanded Cinema.* Dutton.

Zicarelli, D. (2002). How I Learned to Love a Program That Does Nothing. *Computer Music Journal 26*(4), 44–51.