# Large-Scale Physical Modeling Synthesis, Parallel Computing, and Musical Experimentation: The NESS Project in Practice

**Stefan Bilbao,[*] James Perry,[†] Paul Graham,[§] Alan Gray,[§] Kostas Kavoussanakis,[†] Gordon Delap,[‡] Tom Mudd,[**] Gadi Sassoon,[††] Trevor Wishart,[§§] and Samson Young[‡‡]**

[*]Acoustics and Audio Group
University of Edinburgh
Room 2.10, Alison House
12 Nicolson Square
Edinburgh EH8 9DF, UK
[†]Edinburgh Parallel Computing Centre
University of Edinburgh
Bayes Centre, 47 Potterrow
Edinburgh EH8 9BT, UK
[§]Nvidia, 100 Brook Drive, Green Park
Reading RG2 6UJ, UK
[‡]Department of Music
Maynooth University
Room 128, Logic House, Maynooth
Co. Kildare W23 F2K8, Ireland
[**]Reid School of Music
University of Edinburgh
Room 2.12, Alison House
12 Nicolson Square
Edinburgh EH8 9DF, UK
[††]Corso Lodi 59, Milan 20139, Italy
[§§]Department of Music
Durham University
Palace Green, Durham DH1 3RL, UK
[‡‡]17D Carmel on the Hill
Carmel Village Street
Homantin, Kowloon, Hong Kong
{sbilbao, Tom.Mudd}@ed.ac.uk,
{j.perry, k.kavoussanakis}@epcc.ed.ac.uk,
{pgraham, alang}@nvidia.com,
gordon.delap@mu.ie, gadi.sassoon@gmail.com,
trevor.wishart@durham.ac.uk, iphoneorchestra@gmail.com

**Abstract:** Sound synthesis using physical modeling, emulating systems of a complexity approaching and even exceeding that of real-world acoustic musical instruments, is becoming possible, thanks to recent theoretical developments in musical acoustics and algorithm design. Severe practical difficulties remain, both at the level of the raw computational resources required, and at the level of user control. An approach to the first difficulty is through the use of large-scale parallelization, and results for a variety of physical modeling systems are presented here. Any progress with regard to the second difficulty requires, necessarily, the experience and advice of professional musicians. A basic interface to a parallelized large-scale physical modeling synthesis system is presented here, accompanied by first-hand descriptions of the working methods of five composers, each of whom generated complete multichannel pieces using the system.

Computer Music Journal, 43:2/3, pp. 31–47, Summer/Fall 2019
doi:10.1162/COMJ_a_00517

Sound synthesis using physical modeling is, to say the least, a computationally costly undertaking. Throughout the history of computer music, it has often been the case that, during the development of new synthesis tools, there has been an initial phase during which prototypes were built in specialized hardware. Often the aim was to achieve real- or near-real-time performance using relatively established synthesis methods. Well-known examples are the "Samson Box" (Samson 1980; Loy 2013a,b) at Stanford University's Center for Computer Research in Music and Acoustics (CCRMA) and the 4N series at the Institut de Recherche et Coordination Acoustique/Musique (IRCAM), leading ultimately to the IRCAM Music Workstation (Lindemann et al. 1991), later known as the IRCAM Signal Processing Workstation. The IRCAM Workstations ran the early signal-processing version of Max via customized DSP boards in a NeXT workstation (Puckette 1991).

In the case of physical modeling synthesis, computational costs can be many orders of magnitude beyond those of standard abstract synthesis methods, particularly for complex systems. Thus, for the moment, specialized hardware is again necessary. Real-time performance was not the aim of the Next Generation Sound Synthesis (NESS) project, but rather "reasonable-time" performance—to get a glimpse of what kind of sound output is possible. Raw acceleration has thus been one of the main goals, and a variety of approaches have been taken, all relying on parallelization at different scales.

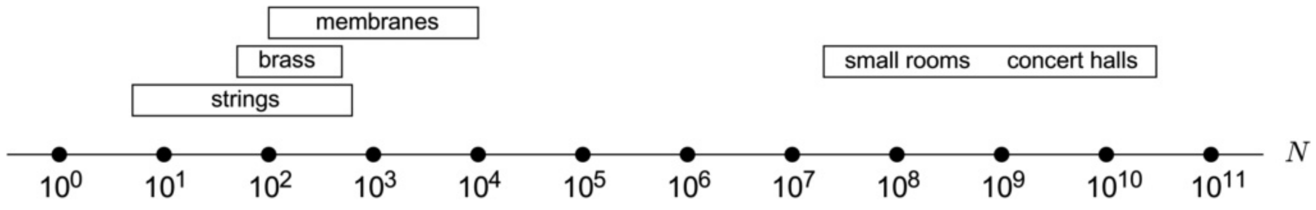
As part of the the NESS Project, sound-synthesis algorithms based on physical modeling were developed for many instrument types, ranging from emulations of existing instruments to purely virtual constructions without a counterpart in the real world. These algorithms are detailed in a companion article in the pages of this issue of *Computer Music Journal* (Bilbao et al. 2020). Though most do not operate in real time, they generate sound quickly enough to constitute a point of departure for musical exploration, which was the second, deeper and more-nebulous goal of NESS. New major issues emerged, relating to usability, interfaces, and control—the musician is faced with the large task of not only learning to play a new musical instrument but also, more often than not, designing it. Sound synthesis in parallel hardware has been explored with regard to standard, abstract synthesis techniques (Savioja, Välimäki, and Smith 2010, 2011), as well as in the case of physical modeling applications, particularly on graphics processing units (GPUs, cf. Zhang, Ye, and Pan 2005; Hsu and Sosnick-Pérez 2013) and using field-programmable gate arrays (Pfeifle and Bader 2015; Motuk et al. 2007). Alongside work on synthesis, another major thrust was towards the development of large-scale room acoustics simulations on GPU—here our approach intersects with that of work in virtual acoustics, divorced from sound synthesis applications—see, e.g., Southern et al. (2010); Mehra et al. (2012).

This article is a detailed account of the practicalities, both technical and musical, involved in bringing physical modeling synthesis into the hands of musicians. (For a more technical view of parallel computing in physical modeling synthesis, see Perry, Bilbao, and Torin 2016.)

## Physical Modeling Synthesis: Computational Complexity

Computational complexity for a physical modeling synthesis algorithm is of course highly dependent on the particular model. One very crude measure of the complexity follows from the required state size—i.e., the amount of memory required to sufficiently represent the dynamics of a given instrument, given some perceptual criterion. Interestingly, it is possible to provide a rather simple lower bound on such memory requirements. To this end, consider any acoustic entity, characterized by its material and geometry. This could be a musical instrument, a single component of an instrument, or perhaps even an enclosing space. Loss is usually low in any such system, and under linear conditions, the system may be characterized by a number of natural frequencies or modes $N(f_c)$ below a chosen cutoff frequency $f_c$. Each such mode behaves, individually, as a harmonic oscillator with two degrees of freedom, and thus requires the updating of two real numbers in memory, and thus the minimum memory requirement is $2N(f_c)$ real numbers (normally double-precision floating point). A synthesis algorithm using less than this amount of memory will necessarily be

*Figure 1. Approximate ranges of* N *in various systems in musical acoustics. The value scales with the amount of memory required to represent the dynamics of a given physical modeling system without discarding potentially audible dynamics. The systems range from "low-cost" 1-D systems, such as strings, to "high-cost" 3-D simulation.*



discarding potentially audible dynamics, and one using more is inefficient—often unavoidably so. In synthesis, $f_c = 20{,}000$ Hz is the most usual choice of cutoff and represents a basic perceptual criterion—the upper limit of human hearing. Although the aforementioned bound on memory requirements follows from an analysis in terms of modes, it is actually quite general—any synthesis method (modal, digital waveguide, finite difference, etc.) must respect this.

Consider first the basic case of a vibrating string, of fundamental frequency $f_0$ Hz. Here, $N(f_c) = f_c/f_0$, which is on the order of about 10 to 500 for musical strings, which is relatively small. At the other extreme, consider the case of a room of volume $V$ m$^3$, and where the sound speed is $c$ m/sec. Now, $N$ is approximately $4Vf_c^3/c^3$. For a medium-sized room ($V = 2{,}000$ m$^3$), and where the speed of sound $c = 343$ m/sec, then $N \approxeq 10^9$, which is very large indeed, but reasonable-time simulation is still within the realm of possibility on specialized hardware, such as GPUs. Ranges of values of $N$, based on a cutoff of $f_c = 20{,}000$ Hz, are given for a variety of acoustic objects in Figure 1. Notice in particular the large gulf between problem sizes for 1-D and 2-D musical instrument components, and for 3-D modeling; parallelization strategies for 3-D modeling have a different character, due to the problem size.

Operation counts are more dependent on the particulars of the system at hand—in most nontrivial cases, however, the number of arithmetic operations per second, for a system with a state size of $2N$ and with a sample rate of $f_s$ will scale as at least $O(Nf_s)$. (Of course, from sampling considerations, $f_s \geq 2\,f_c$.) The digital waveguide is a notable exception in this regard, requiring $O(1)$ operations per time step—an efficiency advantage linked specifically to wave propagation over 1-D lossless homogeneous media, with the ideal string and acoustic tube as examples. As will be seen, however, actual run times are very much dependent on the particular implementation and the possibility of parallelization. More details of the relevant computational structures appear in the following section. Furthermore, such estimates are very crude—they do not take into account more-detailed information about auditory perception, which could be used to further reduce computational cost.

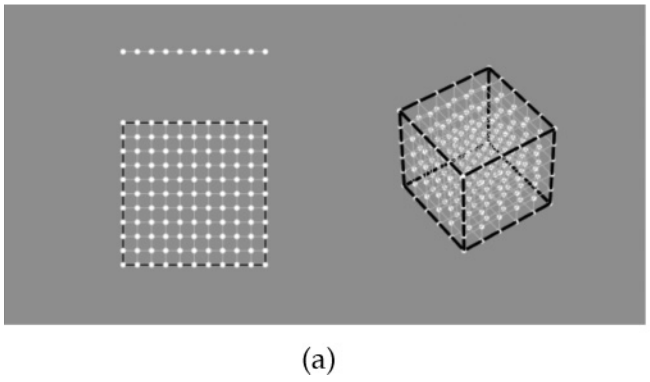
## General Algorithm Structure and Key Operations

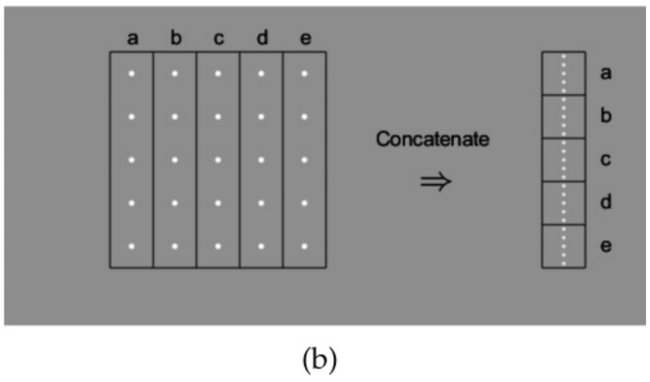
The majority of the time-stepping methods used in the NESS project share many common features across different instrument types. Although it is impossible to describe here the complete workings of all the physical modeling sound synthesis algorithms described in this and in the companion article (Bilbao et al. 2020), an attempt is made here to give the reader some notion of the technical challenges involved, particularly keeping in mind, from the previous discussion, that in some cases the problem size can be very large. (For a more technical presentation of the implementation of such time-stepping methods, see Bilbao et al. 2014.)

### Representing State

As a first step towards understanding the implementation of such methods, consider the representations of the state in typical time-stepping algorithms. Depending on the system at hand, the state (at an integer time step $n$) consists of the values representing the physical variables of the complete system. Such values could represent, for example, the displacements and velocities of a string or membrane,

*Figure 2. Spatial grids in one, two, and three dimensions (a). Concatenation of a 2-D grid into a vector (b).*


(a)



(b)

defined over a grid, or pressure and velocity values within an acoustic tube, or a combination of values representing the state of a heterogeneous system made up of a combination of components (see Figure 2a). For the purposes of the discussion below (and not necessarily in practice!) it is useful to concatenate the entire state, at a given time step $n$, as a vector $\mathbf{u}^n$ (see Figure 2b).

## Recursions

The main computational work across all the NESS code modules is the update of the state at an audio rate—usually chosen to be 44.1 kHz or 48 kHz, although all algorithms can produce sound output at any specified rate. At time step $n$, the state vector $\mathbf{u}^{n+1}$ must be computed using previously computed values of the state. In virtually all synthesis code modules, updates are "two-step": $\mathbf{u}^{n+1}$ may be computed using only $\mathbf{u}^n$ and $\mathbf{u}^{n-1}$. To avoid producing an excess of computed data, once $\mathbf{u}^{n+1}$ is computed, $\mathbf{u}^{n-1}$ may then be discarded (or overwritten). Here, the state vector $\mathbf{u}^n$, assembled by concatenation, is assumed to be of size $N \times 1$, where the total state size is $2N$.

Updates take on different forms, depending on the physics of the problem at hand. In Table 1, generic update equations of four different types, labeled I through IV, are shown. In this simplified representation, input and output operations are not included.

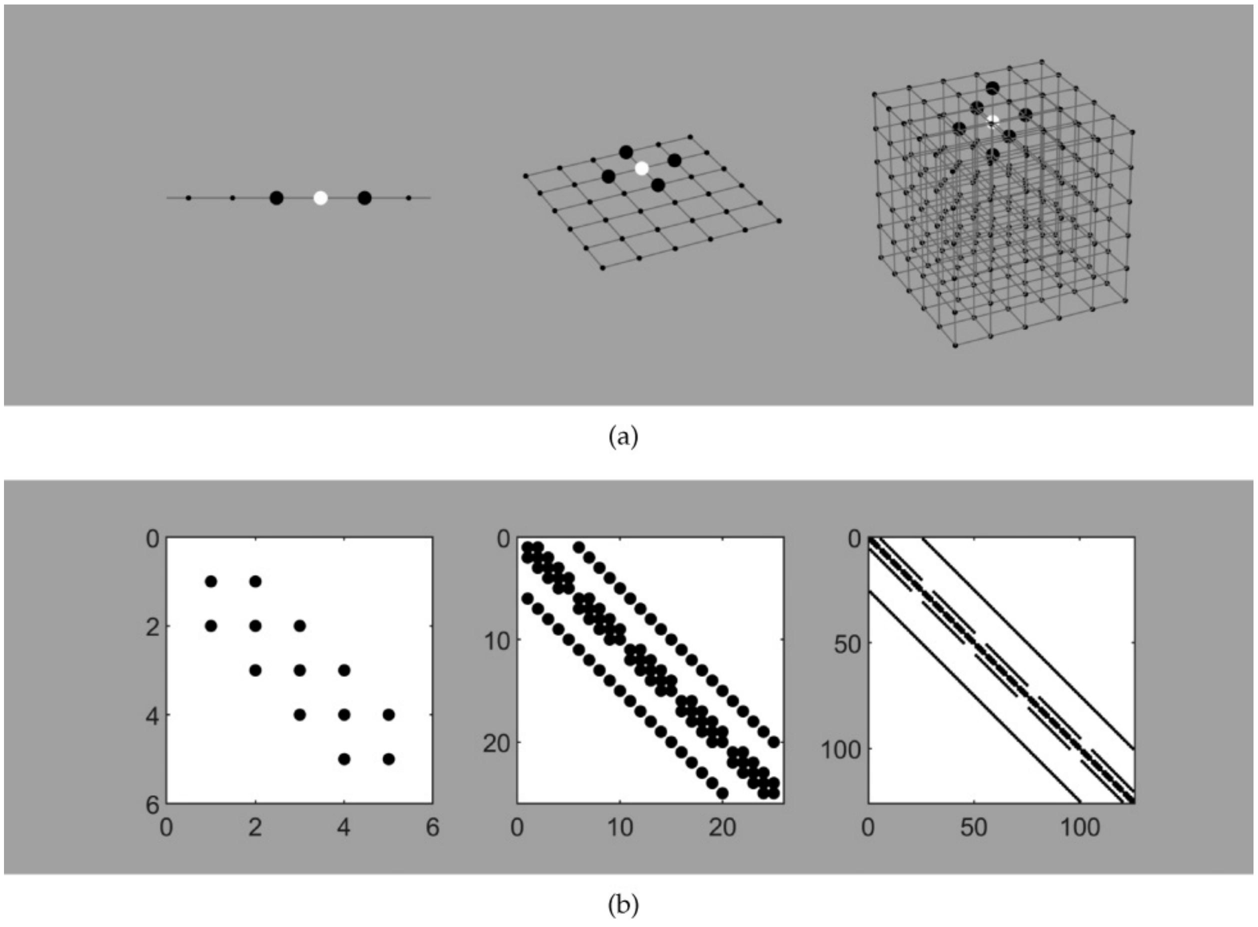
**Table 1. Generic Equations for Update Types**

| | |
|---|---|
| Type I | $\mathbf{u}^{n+1} = \mathbf{B}\mathbf{u}^n + \mathbf{C}\mathbf{u}^{n-1}$ |
| Type II | $\mathbf{A}\mathbf{u}^{n+1} = \mathbf{B}\mathbf{u}^n + \mathbf{C}\mathbf{u}^{n-1}$ |
| Type III | $\mathbf{A}(\mathbf{u}^n, \mathbf{u}^{n-1})\mathbf{u}^{n+1} = \mathbf{B}(\mathbf{u}^n, \mathbf{u}^{n-1})\mathbf{u}^n + \mathbf{C}(\mathbf{u}^n, \mathbf{u}^{n-1})\mathbf{u}^{n-1}$ |
| Type IV | $\mathbf{g}(\mathbf{u}^{n+1}, \mathbf{u}^n, \mathbf{u}^{n-1}) = \mathbf{0}$ |

In the simplest case, the update step is of Type I and requires only matrix multiplication. This corresponds to the case of explicit finite-difference (FD) schemes for linear and time-invariant systems. The $N \times N$ matrices $\mathbf{B}$ and $\mathbf{C}$ may be precomputed at the setup stage, and the internal structure of these matrices follows directly from the physics of the system. They are generally very sparse; the sparsity follows from the use of FD approximations, according to which derivatives are approximated using neighboring values on a grid. This is the ideal case for a parallel implementation. See Figure 3, illustrating typical sparsity patterns, in the case of a matrix $\mathbf{B}$ utilizing a basic approximation to the Laplacian.

It is sometimes the case that the update corresponds to the solution of a linear system, as in Type II, for a known constant $N \times N$ matrix $\mathbf{A}$. This corresponds to the case of implicit FD schemes for linear and time-invariant systems. The matrix $\mathbf{A}$ may be precomputed at the setup stage, but the solution to the Type II case is generally much more problematic than that of Type I. As the size of $\mathbf{A}$ is potentially large, it is inadvisable to precompute

*Figure 3. Spatial grids in one, two, and three dimensions, showing the action of an approximation to the Laplacian at a given location, centered at the white point, and drawing on values at neighboring black points (a). Matrix forms of the Laplacian operator (b). The matrices operate on a state vector arrived at through concatenation, as illustrated in Figure 2.*



the inverse of **A**—though **A** is usually sparse, $\mathbf{A}^{-1}$ will generally not be, and storage requirements can quickly become unmanageable for all but the smallest systems. Depending on the particular structure of **A**, different approaches to linear solution are available. Some, such as Jacobi iteration (Strikwerda 2004), can easily be parallelized, but require additional conditions on **A** (such as diagonal dominance), which are not always satisfied. In the worst case, it may be necessary to use standard general methods, perhaps targeted towards sparse systems (Saad 2003).

In some cases involving nonlinearities of geometric type (such as those occurring in the case of gong vibration, or string vibration at high amplitudes), updates of Type III are necessary. Now the matrices **A**, **B**, and **C** are dependent on the previously computed state $\mathbf{u}^n$, and thus cannot be precomputed. This leads to extra computational cost during the execution of the runtime loop; usually, however, **A** remains sparse, and the linear system solution may be carried out as for systems of Type II.

The most general case is that of an update of the form of Type IV for a known nonlinear vector-valued function **g**(). This arises when nonlinearities of a more-complex form are present, as in the case of collisions (e.g., in the string–fretboard interaction or the mallet–membrane interaction). In this case (with the exception of certain highly pathological situations), iterative methods, such as the Newton-Raphson method (Press et al. 1992), are usually required and offer only scant opportunity for parallelization.

In practice, for realistic and complex designs of musical instruments, the update is often a mixture of the forms above. More precisely, it is often possible to partition the state $\mathbf{u}^n$ into subvectors, over which updates of different types are performed. Such a partition follows naturally from the physics of the problem under consideration. For example, in the case of the 3-D model of the snare drum, updating of the acoustic field is of Type I; that of the membrane of Type II; and that of the snares themselves of Type IV, owing to the highly nonlinear collision mechanism.

### Inputs and Outputs

Inputs and outputs to the synthesis algorithms, obviously essential for any meaningful musical work, have been not been described above. In physical modeling synthesis, the excitation mechanism driving a given instrument design will generally be nonlinearly coupled to the instrument dynamics. This is the case, for instance, in all of the standard playing modes (bowing, striking, plucking, or blowing). The case of output is simpler: Generally, values may be drawn from the appropriate location in the state, as it evolves, in the manner of a pickup or microphone. In the NESS code, simultaneous multiple inputs and outputs are possible. As a result, these synthesis methods are geared towards multichannel composition, and they became a large influence on both the mode of work of the associated musicians and the ultimate forms that their compositions took, as will be described later.

Under greatly simplified conditions, the entire input/output system may be viewed in a form resembling a state-space form (Kailath 1980). For example, considering the case of the simple linear form given as Type I above, an extension to incorporate I/O may be written as

$$\mathbf{u}^{n+1} = \mathbf{B}\mathbf{u}^n + \mathbf{C}\mathbf{u}^{n-1} + \mathbf{J}_i\mathbf{w}^n$$
$$\mathbf{y}^n = \mathbf{J}_o\mathbf{u}^n.$$

Here, the subscripts $i$ and $o$ refer to input and output, respectively. The vector $\mathbf{w}^n$ is an $N_i \times 1$ input vector consisting of a set of $N_i$ independent input signals at time step $n$; $\mathbf{J}_i$ is an $N \times N_i$ matrix, each column of which specifies the grid point or set of grid points to which the input will be added. Similarly, $\mathbf{y}^n$ is an $N_o \times 1$ vector of output signals, and $\mathbf{J}_o$ is an $N_o \times N$ matrix, each row of which selects a particular set of grid points in the physical model from which output will be drawn (with scaling included).

Generally, the computational cost of input and output operations is minimal compared to the raw calculation that must be carried out over the entirety of the computational grid. Extensions to the time-varying case are possible as well—input locations can be variable (as in, e.g., the case of bowing or finger-stopping in the guitar).

## Synthesis Environments

A variety of distinct environments developed over the course of the NESS project, based around the investigations of different instrument families described in the companion article (Bilbao et al. 2020). Acceleration strategies depend highly on the particular system, and are described subsequently, but all environments have ultimately been made available to musicians through a Web interface.

The Zero Code environment, developed in 2013, was the first attempt at a large-scale modular synthesis network. The basic units are plates, of dimensions and material properties as specified by the user. Connections between objects are point-like, and behave as nonlinear mass–spring–damper systems, where the nonlinearity is of cubic type. This restriction to cubic nonlinearities eases computational requirements considerably, as it is possible to develop stable algorithms of Type III, where the linear system to be solved is diagonal (thus requiring simple scalar divisions in the run-time loop). Input can be specified through a list of events, each of which corresponds, roughly, to a strike at a given location on a given object at a given time and of a given force. Each such event is translated, ultimately, into a short force signal fed into the network. Other input types are a bowing gesture, described through a breakpoint function for bow force, velocity, and position, as well as audio input, if the network is to be used as an effect (emulating, e.g., plate reverberation). Multichannel output may be obtained by reading velocities at a set of locations throughout the network, with the option for the normalization of individual channels in the case that output amplitudes are widely disparate. These modular environments are described in Bilbao (2009).

The Brass Code environment, developed in 2013–2014, allows for the flexible construction of brass and brass-like instruments, with arbitrary bore profiles and valve configurations; control is through mouth pressure and lip stiffness, as well as a set of control signals representing valve positions. Because of its rapid execution time (several times faster than real time) it has been a very popular choice among composers, and extensively explored. Part of the reason for the fast execution time is the relatively small state size (for reasonably sized instruments and at an audio rate, $N$ is on the order of 100 to 400), but another is that a Type I explicit update is used for the bulk of the calculation. Control is of a different type from that used in the Zero Code, in that continuous data streams are necessary, rather than a discrete list of events. Breakpoint functions (piecewise linear here, but easily generalized) are used to specify such control streams. The complete brass synthesis environment is described by Harrison et al. (2015).

The single- and double-membrane drum code and the gong code were developed between 2013 and 2015. They incorporate many realistic features, including membrane–plate nonlinearity, mallet interactions, and snares, and they are ultimately embedded in three dimensions, allowing for spatialized sound output. This was the closest approach to fully virtual musical instruments, in that it is possible to embed multiple instruments within a potentially large space. The calculation is immense, however, as it relies on updates of Types I, II, III, and IV simultaneously; and for particularly large state sizes it is relatively slow (taking minutes to calculate several seconds of output), which makes this approach more difficult to explore musically. As in the case of the Zero Code environment, input is specified through strike forces and locations over multiple objects. In contrast, however, full multichannel output may be drawn from the 3-D simulation of the acoustic field. Such virtual percussion instruments are described by Torin, Hamilton, and Bilbao (2014) and by Bilbao and Webb (2013).

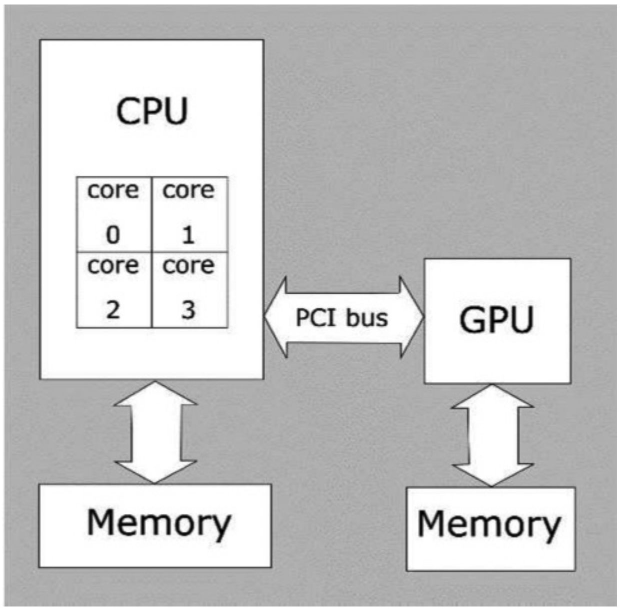
The Guitar Code environment was developed from 2014 to 2015, and was perhaps the most elaborate instrument attempted in terms of control. The user specifies string parameters, such as tension, length, stiffness, and $T_{60}$ times for $N$ strings. In addition, the user supplies a common backboard profile, against which strings will collide, as well as the positions of an arbitrary number of frets. The user must further specify the properties of fingers that will interact with the instrument, in particular their masses and stiffness. From the control point of view, breakpoint functions are used to specify the positions of the fingers along the length of the string, as well as the forces exerted by the player on each string over time. In addition, the user specifies plucking or striking events along the string. The guitar environment is described by Bilbao and Torin (2015).

The last modular code developed, in 2016, was Net1. It differs from the Zero Code in that plates are not included, and that bar and string behavior are both unified into a single model, and thus the model can behave as a string or percussion instrument. In addition, nonlinear connections are of a very different type, allowing for intermittent loss of contact of a type reminiscent of a rattle, and leading to a much larger variety of timbres. Input and output is much the same as in the case of Zero Code. Because of the form of the nonlinearity, iterative methods of Type IV are required, leading to much slower run times than in the case of the Zero Code environment.

## Acceleration in Parallel Hardware

Acceleration in parallel hardware proved to be a significant challenge, and various different methods were combined to achieve good performance. In all cases, prototyping was carried out in MATLAB. The next step was to port the code to generic C++ as a starting point for further experimentation with parallelization techniques. All of the models were ported into a common code framework so that

*Figure 4. GPU server architecture, showing the main computational and memory components, and the buses connecting them.*



useful generic functionality could be shared among them.

The appropriate strategies for speeding up a given piece of code are dependent not only on the code itself, but also on the target hardware: Large scale supercomputers and clusters demand a different approach from desktop computers and smaller servers. The optimized NESS code was targeted primarily at multicore Intel Xeon servers with multiple NVIDIA Tesla GPU cards, since it was used to run the NESS service for the composers, but it was also designed with portability in mind and can run on most Windows, MacOS, and Linux machines (see Figure 4).

## Multicore

One form of parallelism that is almost ubiquitous in computers of all sizes today, and that is conceptually quite simple, is multicore processing. Modern CPUs contain multiple cores, each of which can be running an independent thread of program execution at the same time. These cores share a single memory space, so the program threads can operate on the same data, though care must be taken to ensure synchronization and that the threads do not interfere with each other.

This type of parallelism raises the question of how best to divide up the program's work between the various threads. Two different approaches were tested for the NESS code modules:

1. Assigning each distinct component of the simulation to its own thread.
2. Decomposing the domain of the simulation across multiple threads. For example, in the case of a membrane with 1,000 elements, the first 500 elements could be updated on one thread while the last 500 are simultaneously updated on another.

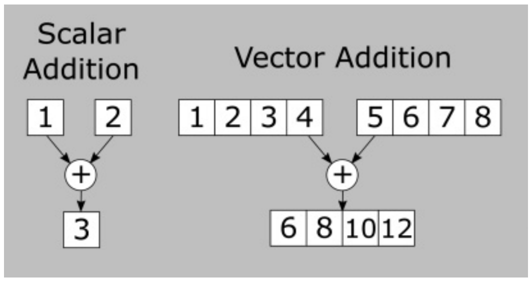
An example of the first approach is the Net1 Code environment. In this environment, each string, bar, and connection is processed on a separate thread. In the case where the components outnumber the available CPU cores, each thread may have to process multiple components in sequence. An example of the second approach would be a membrane with 1000 elements, in which the first 500 elements could be updated on one thread while the second set of 500 elements are simultaneously updated on another.

The first approach was adopted as it had two major advantages: First, it is very simple to implement and requires minimal communication among the threads, since each thread is working on a mostly self-contained entity; and second, because the component update itself is unchanged from the serial case, it works with any algorithm.

In contrast, the second approach requires changes within the update code, which adds complexity and is difficult to code efficiently for implicit or nonlinear (Types II, III, and IV) updates. The main disadvantage of the chosen method is that it is useless for simulations that cannot be broken down into multiple components, but these are generally either lightweight enough to run efficiently even on a single core (brass instruments, for example), or are large room simulations that are better run exclusively on graphical processing units (as will be discussed later in this article).

The standard for high-performance multithreaded code is OpenMP (cf. Dagum and Menon 1998), a

*Figure 5. Scalar and vector addition. The vector operation performs four separate additions in the same time period as the single scalar addition.*



directives-based framework that allows work to be easily shared across multiple threads in a shared-memory environment. This proved, unfortunately, to be unsuitable for the NESS code; OpenMP imposes a certain overhead at each simulation time step, and this overhead becomes problematic due to the large number of time steps required when working at audio rate. We created a simple threading framework, which on Windows uses Win32 threads and on Linux and MacOS uses the low-level thread libraries (Butenhof 1997) provided by the operating systems. This provided full control over the behavior of the threads, allowing them to be kept in a state where they could respond instantaneously when needed.

In the best-case scenario, the performance of multicore code scales with the number of CPU cores available. In practice this is rarely achieved, because of synchronization overhead and imbalances between the work assigned to each thread. Owing to the minimal interthread communication required, the NESS multicore code does approach the best-case scaling in certain cases.

## Vectorization

Modern CPUs also provide opportunities for finer-grained parallelism within each CPU core. While a traditional machine instruction (the most basic building block of all computer programs) performs a single calculation, a vector instruction can perform multiple calculations simultaneously. Figure 5 gives a simple overview of this. As a concrete example, the vector instructions contained in the SSE2 instruction set (Raman, Pentkovski, and Keshava 2000), present on all 64-bit CPUs produced by Intel and Advanced Micro Devices, can perform two double-precision or four single-precision floating-point calculations at once, and Intel's Advanced Vector Extensions (AVX) instructions extend this to four double-precision or eight single-precision operations.

This is extremely useful for looping through an array of numbers and applying the same operation to all of them, a pattern that occurs frequently in the NESS simulations (especially Type I updates, but also when computing the matrices in Type III). Modern compilers can sometimes vectorize such code automatically. Vector instructions can also be used to speed up operations that are more complex, but this takes more manual effort, typically requiring the use of compiler intrinsics (special functions that translate directly into specific machine instructions).

A four-element vector operation will be at most four times faster than the equivalent scalar operations, but often this upper performance bound cannot be reached in practice, owing to memory bandwidth or other limitations. For NESS, AVX instructions were used extensively in the main state update of the brass code, giving a 2.5-fold increase in speed compared to the serial C++ code, thereby allowing many instruments to be simulated faster than real time. The triangular solve operation used in the preconditioner of the linear-system solve, used in the Type III updates, was vectorized using SSE, again giving an approximate speed increase over the original C++ version by a factor of 2.5. (In this case SSE was actually faster than AVX because of technical limitations of the AVX instruction set).

## GPU Acceleration

Graphics processing units were originally designed for rendering real-time graphics for games, but in recent years there has been much interest in exploiting them for other purposes, and GPU vendors provide tools like Nvidia's Compute Unified Device Architecture (CUDA, cf. Nickolls et al. 2008) and the open standard OpenCL to facilitate this. Graphics processing units are essentially massively parallel processors containing hundreds or even thousands of cores, though each individual core is primitive and slow compared to a traditional CPU core.

Because the power of GPUs comes from their extreme parallelism, they are best suited to algorithms with large numbers of operations that can be performed simultaneously, and much weaker for running sequential code. In addition, they typically have their own memory space and are separated from the rest of the computer system by a relatively slow bus (PCI Express); care is required to design code modules to minimize the amount of data that needs to be transferred between CPU and GPU.

Graphic processing units are therefore well suited to running the NESS algorithms that perform a relatively simple linear operation, such as a Type I update, across a large domain—either a 3-D room or a large 2-D plate or membrane. One-dimensional and smaller two-dimensional entities are generally too small to benefit from GPU acceleration, and updates that are more complex (Types III and IV) require too many sequential computations on a single thread (Bilbao et al. 2013; Perry, Bilbao, and Torin 2016).

A flexible approach was taken in the NESS framework, allowing all simulation components to take advantage of the GPUs if beneficial, or to remain on the CPU if not. Additionally, GPU versions of most of the input, output, and modular interconnection algorithms were provided, allowing these operations to be performed directly on the GPU when required, instead of having to run them on the CPU and then copy the resulting data across the slow PCI bus.

Although the effort involved in porting code to CUDA is relatively high, the performance gains can be significant. For example, updating the acoustic field surrounding the bass drum is roughly six times faster on the GPU than it is on the CPU (Perry, Bilbao, and Torin 2016).

## Algorithm Tuning and Optimization

In addition to the parallelization methods already described, some NESS code was sped up significantly using other optimizations. One example was the Newton-Raphson solver used in the guitar code to simulate collisions between strings and other items. In its original form this involved solving a relatively large, sparse linear system, which can be a time-consuming operation. In most cases this can, however, be converted to a much smaller, dense linear system by rank reduction. This small, dense system (typically containing fewer than 20 unknowns) can then be solved rapidly by direct factorization.

In addition, most of the NESS code makes extensive use of sparse matrix and vector operations such as multiplication and addition. In the general case these operations can be expensive as they need to cater to every possible type of matrix, but they can often be sped up dramatically by using an algorithm and a matrix storage format more suited to the specific matrices involved. For example, the generation of matrices in the Type III algorithm originally used generic operations and compressed sparse row matrix storage. This was replaced by a custom banded-matrix format and specialized operations, often combining multiple operations into a single step so that intermediate results did not have to be written to memory and later read back again. The overall effect was to improve performance in this part of the algorithm by a factor of ten.

A dramatic increase in speed (by a factor of about 300) was obtained for the bowed string code by replacing a simple linear search of a table with a more sophisticated algorithm that performed a binary search on a sorted version of the table, and mapped the results back onto the original.

## Acceleration Summary

As described above, the various optimization methods all have their strengths and weaknesses and are better suited to some algorithms than others. Most of the NESS code was optimized using one or two of these methods. The Bass Drum Code and Gong Code were more challenging, however, and required a hybrid approach making use of all four methods:

1. The simulation of the surrounding acoustic field is GPU-accelerated;

**Table 2. Acceleration Relative to Single-Core MATLAB Implementation**

| *Code* | *Multicore* | *Vectorization* | *GPU* | *Algorithm Tuning* | *Speedup* |
|---|---|---|---|---|---|
| Brass | | ✓ | | | 15× |
| Bowed String | ✓ | | | ✓ | 300× |
| Guitar | ✓ | | | ✓ | 21× |
| Bass Drum | ✓ | ✓ | ✓ | ✓ | 60–80× |
| Zero Code | ✓ | | ✓ | | 25× |
| Net1 | ✓ | | | ✓ | 20× |

2. The plates and membranes each run on their own CPU core;
3. Some elements of the plate and membrane code are vectorized with SSE2; and
4. Extensive algorithmic changes were implemented to speed up generation of sparse matrices.

The software framework developed for NESS makes it relatively easy to combine all of these disparate methods, and the end result were increases in speed for typical instruments by factors of about 60–80 compared to the the original MATLAB versions. For example, in the case of the bass drum, for typical choices of parameters, simply porting the code from MATLAB to C++ improved performance by a factor of roughly 5.9. This was then further improved by factors of 2.4 for using a multi coprocessor, 2.3 for CUDA, 1.8 for algorithmic changes, and 1.2 for vectorization, leading to an over-all acceleration by a factor of 70.3.

Table 2 shows the methods used to optimize the code for each instrument, and the rough overall speed improvement of the final optimized C++ code compared to the original MATLAB implementation.

## User Control

Because most of the synthesis algorithms developed under the NESS project were slower than real time, there were no attempts at a sophisticated graphical interface. Rather, work on the control side was at the lower level of determining usable parameter sets that could be approached even by those who were not experts. From an early stage, it was decided to adopt the score–instrument breakdown of user-supplied input data, which has been standard across various iterations of "Music N" synthesis environments.

### Instrument and Score Files

The NESS synthesis system accepts an instrument file and a score file as input, and also possibly audio input, if the synthesis algorithm is to be used as an effect. In Figure 6, a rudimentary instrument-and-score file pair is shown, in the case of the Zero Code modular-plate synthesis network. The instrument file specifies the sample rate and the parameter sets defining a set of thin metal plates, as well as connection elements linking two plates together (or a plate to itself). For a tensioned rectangular plate, for example, the parameters are the material, thickness, tension, dimensions, as well as specifications of 60-dB decay time at DC and at 1 kHz, and an integer value specifying the type of boundary condition (in this case, clamped, simply supported, or free). For a connection, the parameters are the coordinates with respect to two different plates, one parameter each for linear and nonlinear stiffness, and an additional 60-dB decay time for the connection itself (realized as a damper). Outputs are also defined, with reference to coordinates on the plates. The score specifies the total duration of the simulation, and also strike events, as well as a bowing gesture. Audio input is also an option.

### Web-Based Interface

A simple Web-based user interface was created to allow the composers to access the NESS service

*Figure 6. Instrument and score files for the Zero Code modular-plate environment.*

Instrument File

```
# zcversion 0
# set 44100Hz sampling rate
samplerate 44100

# Define two steel plates. Arguments:
# <name> <material> <thickness> <tension> <X> <Y> <T60@0Hz> <T60@1kHz> <boundarytype>
plate plat1 steel 0.001 0.0 0.4 0.7 11.0 6.0 4
plate plat2 gold 0.001 0.0 0.3 0.6 10.0 6.0 4

# Define a connection from one point on the plate to another on this or a different plate. Arguments:
# <component> <component> <X1> <Y1> <X2> <Y2> <linearstiffness> <nonlinearstiffness> <T60>
connection plat1 plat2 0.7 0.4 0.3 0.7 10000.0 10000000.0 1000000.0
connection plat1 plat1 0.2 0.3 0.5 0.5 10000.0 10000000.0 1000000.0

# Define two outputs from the two plates. Arguments:
# <component> <X> <Y> <pan>
output plat1 0.6 0.6 -1.0
output plat2 0.3 0.7 1.0
```

Score File

```
highpass off # no high-pass filter
duration 1.0 # one second simulation

# Define two strikes. Arguments:
# <starttime> <component> <X> <Y> <duration> <amplitude>
strike 0.0 plat1 0.4 0.7 0.002 400000.0
strike 0.0 plat2 0.3 0.6 0.001 100000.0

# Define a bowing action. Arguments:
# <starttime> <component> <X> <Y> <duration> <forceamp> <velocityamp> <friction> <ramptime>
bow 0.3 plat1 0.3 0.9 4.0 2.3 2.8 1.1 0.02

# Define an audio input. Arguments:
# <file> <starttime> <component> <X> <Y> <gain>
audio drumming.wav 0.1 plat1 0.2 0.4 1.0
```

through a standard Web browser. It consists of a form allowing the user to upload an instrument file, a score file, and an optional audio input file, and then to launch the NESS code on the GPU server. A demo mode is supported for most code modules; this runs as simulation at a lower sample rate, giving users an impression of how their submission will sound, and generally completes much faster than a full-quality run. The synthesis environment to run is selected via a comment in the instrument file. After the job completes, the user is able to download the generated audio outputs (individual mono channels as well as a stereo mix) as WAV files.

## Musical Experimentation

An integral part of the NESS project was collaborative work with electroacoustic composers, which commenced approximately 18 months into the project. All work was carried out in a 16-channel space during intensive workshops with the NESS team. In most cases, the musicians worked directly through the Web-based interface to the GPU server. As described in the previous sections, the interface itself is quite raw—there were thus many opportunities for musicians to develop their own approaches to instrument design, writing sufficiently interesting and complex scores, and handling multichannel output. Approximately ten complete works resulted, using anything from 8 to 32 channels. In this section, a variety of composers provide their own insights into the experience of working with the NESS system. They are arranged approximately chronologically, and cover the period from 2013 to 2019. Many of the pieces can be heard on the NESS Web site at www.ness.music.ed.ac.uk.

## Gordon Delap: *Ashes to Ashes*

In *Ashes to Ashes* (2013), instruments were assigned certain physical properties of uranium. Perhaps understandably, this choice was received with some degree of skepticism, although the assignment of parameters relating to radioactive elements was intended principally as a conceptual starting point. More fundamentally, the work was conceived of as a reflection on life and death. It was also envisaged that the instruments would display behaviors of real-world objects while exhibiting surreal qualities, although the surreal aspect usually had more to do with nontraditional topologies than material specifications.

A vital concern was the investigation of compositional applications of the technology. It seemed appropriate, then, to set two constraints:

1. Modification of audio output via post-processing techniques was not permitted
2. Sound materials generated via external means could not be used, except where such sounds were processed by being fed through one of the NESS project's models.

These constraints presented challenges. First, the generation of audio took an extremely long time in the early stages of the NESS project. Overnight processing for instruments with large dimensions was not unusual, while fine-tuning of instruments was often difficult. Second, the composition of electroacoustic music typically allows for the deployment of a vast and powerful armory of digital signal processing techniques. Such techniques have been developed and refined over decades of experimentation and investigation. These methods could only have been accessed, however, at the expense of overshadowing unmanipulated output from the models.

From the outset, there was an awareness of the types of raw sound output that might be expected. For instance, idiophones and membranophones were readily obtainable. Blown, brass-type instruments were also under development at the time. *Ashes to Ashes* was especially influenced by the ritual music of Tibet. It seemed that many aspects of this sonic world lay within the capabilities of the physical models under development, although the composition is not reducible to this concern.

Learning how to play the brass instrument models was difficult. Embouchure parameters called for precision. Much to the credit of the model's accuracy, when this principle was violated, the outcomes sounded every bit as deficient as those produced in real life by unskilled brass players.

## Trevor Wishart: *Dithyramb-Kepler 63c*

The work *The Secret Resonance of Things* (2014) attempts to derive music from scientific data of various kinds (the spectra of supernova, the onset of turbulent flow, and so on). *Dithyramb-Kepler 63c*, the third movement, uses the NESS physical modeling software to conjure a musical celebration on an earth-like planet using alien instruments and an "unknown" musical style. I used quasi-random sets of values as input to the NESS models of both brass and plates, searching, initially haphazardly, for sonically interesting results. I wanted a rich sonic palette of brass instrument articulations and various different percussive sounds (e.g., wood-like, clunk-metal-like, marimba-like) to create my alien ensemble. Having found sonorities and articulations that I liked, I explored small adjustments to the parameter values to uncover sets of closely related sounds (which might conceivably have come from the same instrument, and player). Using the Composers' Desktop Project tools to adjust pitch and layer, and to otherwise expand the sounds, I was able to compose lines of complex and plausible instrumental music. The streams from these various "players" were spatialized in eight-channel surround sound to create the illusion of an encircling ensemble. Such spatialization particularly animates the rhythmic percussive sequences in the piece. As often in my compositional history, I developed the software interface (to allow brass profiles to be drawn directly, rather than entered numerically) only after I had completed the piece!

[Editor's note: *Dithyramb-Kepler 63c* is included in the Sound Anthology published with the Winter 2019 issue of *Computer Music Journal*, Vol. 42, No. 4.]

**Gadi Sassoon: *Multiverse***

My investigation of the NESS environments Guitar, Net1, and Brass revolved around their application for music production and their integration with analog synthesis and acoustic instruments. Through the NESS physical models I was able to explore a space that weaves in and out of realism and abstraction, seeking to develop a textural bridge between synthesizers and orchestral instruments in pursuit of a cinematic sound that blurs the edges between synthetic and organic.

To that end, I have explored the parameter space's fringe areas of operation: With some experimentation the models allow for the rendering of instruments designed with impossible physics (e.g., gigantic brass instruments blown with very hot air, needle-thin fingers gently rattling against a string's harmonic on a fret with no loss, and lattices of bound masses vibrating infinitely). The nonlinearities of the systems provide exceptional detail and help maintain quasi-mechanical semblance even in otherworldly simulations. The ability to sample some models in more than one location makes for compelling multichannel output, allowing gestures ranging from short bursts to complex immersive soundscapes.

Interacting with the NESS environments has pushed me to develop new sonic vocabulary: The unconstrained nature of a research-oriented system presented unique challenges, requiring creative solutions, which in turn prompted previously unimagined musical ideas. The result of our collaboration over the years from 2016 to 2018 is my album *Multiverse*, comprising pieces (such as *Collision Suite* and *Black Hole Fanfare*) made entirely from code output and others (like *Life on a Tidally-Locked Planet* and *Chaos and Order*) which are a combination of NESS output, modular synthesizers and live strings. All the pieces have both stereo and 8-channel mixes. They have been performed at IRCAM, CCRMA, the 2018 conference on New Interfaces for Musical Expression in Virginia, and the 2018 Music Tech Fest in Stockholm.

A nonlinear, interactive version of the piece *Chaos and Order* was unveiled at the 2019 Sónar festival in Barcelona. Here it was presented in the form of an immersive installation using three interactive sculptures, head-tracking headphones, and the Traverse augmented reality platform. (Traverse is developed by the studios Superbright and Vrai Pictures in New York. See www.traverse.fm for further details.)

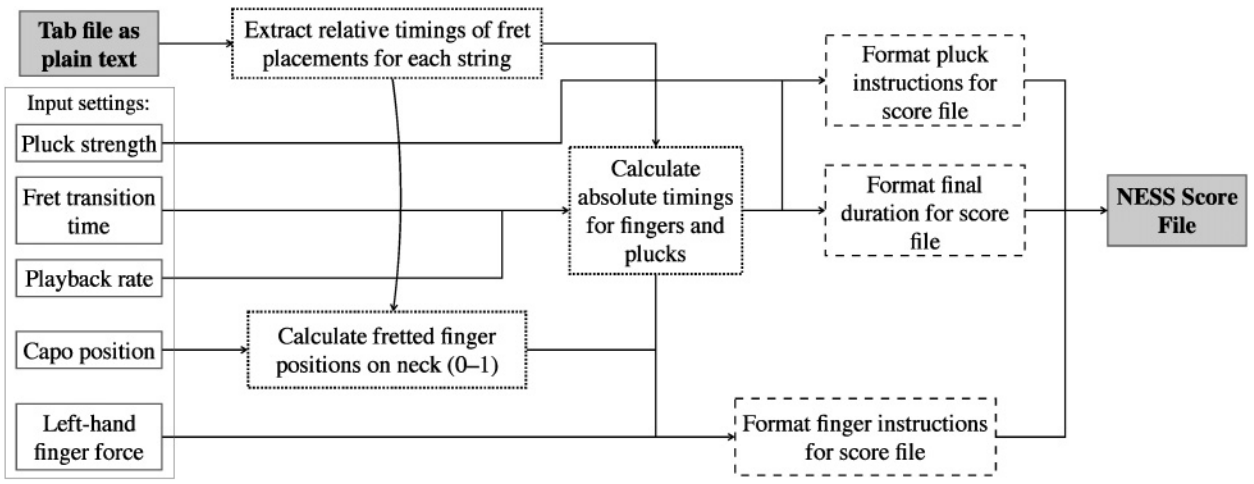
**Samson Young: *Possible Music No. 1***

I visited Edinburgh to work with NESS twice for this research, briefly in the summer of 2017 and then again for two and a half weeks in November of the same year. My collaboration with the NESS project resulted in a new commission at the Guggenheim Museum in New York city, titled *Possible Music No. 1* (2018, featuring NESS and Shane Aspegren), which was on view at the museum from May to October of 2018, and has since become a part of the Guggenheim's collection.

Using the NESS software, I created a series of musical compositions that are sometimes enriched with human vocals. These arrangements are activated according to a precise schedule that references the use of bugle calls in the military to signal orders and events. Using old and new sounds to collapse temporal and spatial divides, my aim was to interrogate our search for truth, as well as music's unsung role in shaping the birth and progress of civilization.

For this work, I first created a few dozen brass instruments with the NESS software. I designed a Pure Data patch to help me generate score files. In this Pd patch I am able to draw various musical parameters in a graphical interface, which allowed me to work visually and intuitively with the code. This process resulted in a large database of short monophonic samples, with durations from 15 to 30 seconds. In a sequencer, these short samples are combined into longer compositions that are each one minute to a minute and a half in length. Each composition consists of six to ten audio channels. Finally, these compositions were spatialized using a 10.1-channel speaker array, controlled by a custom Max patch.

*Figure 7. Diagrammatic structure of a tablature converter for the guitar code.*


Tab file as plain text
Extract relative timings of fret placements for each string
Input settings:
Pluck strength
Fret transition time
Playback rate
Capo position
Left-hand finger force
Calculate absolute timings for fingers and plucks
Calculate fretted finger positions on neck (0–1)
Format pluck instructions for score file
Format final duration for score file
Format finger instructions for score file
NESS Score File

**Tom Mudd: *Brass Cultures* and *Three Algorithms for Hans Reichel***

I have been using the NESS Brass environment since 2015 and the Guitar environment since late 2017. The material has been used for performances at Café Oto in London in 2017 and 2019, and at Saint Cecilia's Hall in Edinburgh in 2018.

The simple text-based score and instrument format used in the NESS interface makes it relatively easy to create algorithms that generate both score and instrument files. Multiple scores and instruments can be created by the same algorithm, allowing different instruments to be played in a coordinated fashion.

Each piece for the *Brass Cultures* album was created with a Python script that generates scores and instruments. Score parameters were generated according to a variety of rules: creating lip-frequency sequences from predefined sets, choosing breath pressures in certain ranges and with defined envelopes, creating different rules for difference sections of each piece, different rules for the timings of each section, and so on.

As part of the work *Three Algorithms for Hans Reichel*, which uses the Guitar environment, a converter was implemented in JavaScript that takes a plain-text representation of guitar tablature as input and creates NESS-formatted score files as shown in Figure 7. For physical modeling, tablature is a more appropriate score mechanism than other input options such as MIDI, as it denotes the specific finger placements explicitly rather than allowing the performer to decide how to assign individual notes to the guitar strings. The guitar tablature displays a row for each string and uses numbers to denote on which fret a finger is to be placed (the finger number is not specified). Duration is indicated using hyphens: For example, the following $\frac{4}{4}$ measure of tab for one string shows the first, fourth, and fifth notes being held for twice as long as the second and third notes: `|5–––5–4–5–––0–––|` (in other words, the rhythm quarter, eighth, eighth, quarter, quarter).

Although the tablature converter can be used to render slightly mechanical versions of existing pieces, it can also be used as a rapid way to create simple structures and patterns for use with unconventional string instruments that may or may not yield obviously pitched results.

## Concluding Remarks and Perspectives

The NESS project was intended as an opportunity to step back from the constraints of real-time performance, to fully come to grips with the entire physical modeling "production chain," from basic algorithm design to implementation and musical use. Indeed, given the computational demands of this type of physical modeling synthesis, and the unknown territory of algorithm parallelization, it seemed appropriate to use this "offline" research model, which was inspired, perhaps subliminally, by the classic working methods at Bell Labs in the 1950s and 1960s.

But the real-time challenge remains—and real-time performance is what many musicians ultimately want. A small subset of the algorithms presented here can operate faster than real time, in particular those of the Brass synthesis environment. In other cases, it is fairly clear that real-time performance will not be possible for the foreseeable future: Examples are 2-D instruments such as gongs and drums, where the strong nonlinearity forces a choice of algorithm design that can not be easily parallelized, and full 3-D room acoustics simulation, which, though algorithmically simpler, is a problem of a vast scale. Modular designs are inherently scalable, and are a good choice for real-time operation. The first modular physical modeling synthesis system based on NESS, Derailer, was released as a plug-in in 2018 by Physical Audio (www.physicalaudio.co.uk), and more advanced models are in various stages of porting from the NESS code modules.

Beyond the question of raw computation time, instrument design and control remain difficult problems, which is not unexpected, given the history of musical instrument design and performance: Both building an instrument and learning to play it have always required an inordinate amount of patience and labor. A relatively low-level mode of design and control (instrument and score files) was the point of departure in joint work with musicians—intended to allow freedom of exploration of the resulting soundspace, and, more importantly, not to over-constrain it. Any instrument should probably be difficult both to build and and to learn, if it is ever to produce musically interesting sound—but "difficult" within reason.

At this point, though, where creative concerns become central, the engineer's role must recede into the background—and this was exactly our approach. Each musician we have worked with has approached the instrument design and control issue in a distinct way, and in some cases has built higher-level working tools to intuitively generate instruments and scores. Ultimately, it is these individual approaches to design and control that give each completed piece of work its distinctive sound, and these approaches are perhaps a further step towards the mature use of physical modeling in sound synthesis.

## Acknowledgment

This work was supported by the European Research Council, under grants 2011-StG-279068-NESS and 2016-PoC-737574-WRAM.

## References

Bilbao, S. 2009. "A Modular Percussion Synthesis Environment." In *Proceedings of the International Conference on Digital Audio Effects*, pp. 321–328.

Bilbao, S., and A. Torin. 2015. "Numerical Modeling and Sound Synthesis for Articulated String/Fretboard Interactions." *Journal of the Audio Engineering Society* 63(5):336–347.

Bilbao, S., and C. J. Webb. 2013. "Physical Modeling of Timpani Drums in 3D on GPGPUs." *Journal of the Audio Engineering Society* 61(10):737–748.

Bilbao, S., et al. 2013. "Large Scale Physical Modeling Synthesis." In *Proceedings of the Stockholm Musical Acoustics Conference*, pp. 593–600.

Bilbao, S., et al. 2014. "Modular Physical Modeling Synthesis Environments on GPU." In *Proceedings of the International Computer Music Conference*, pp. 1396–1403.

Bilbao, S., et al. 2020. "Physical Modeling, Algorithms, and Sound Synthesis: The NESS Project." *Computer Music Journal* 43(2–3):15–30.

Butenhof, D. 1997. *Programming with POSIX Threads*. Boston, Massachusetts: Addison-Wesley.

Dagum, L., and R. Menon. 1998. "OpenMP: An Industry Standard API for Shared-Memory Programming." *IEEE Computational Science and Engineering* 5(1):46–55.

Harrison, R. L., et al. 2015. "An Environment for Physical Modeling of Articulated Brass Instruments." *Computer Music Journal* 29(4):80–95.

Hsu, B., and M. Sosnick-Pérez. 2013. "Realtime GPU Audio." *ACM Queue* 11(4):40–55.

Kailath, T. 1980. *Linear Systems*. Englewood Cliffs, New Jersey: Prentice Hall.

Lindemann, E., et al. 1991. “The Architecture of the IRCAM Musical Workstation.” *Computer Music Journal* 15(3):41–49.

Loy, D. G. 2013a. “Life and Times of the Samson Box.” *Computer Music Journal* 37(3):26–48.

Loy, D. G. 2013b. “The Systems Concepts Digital Synthesizer: An Architectural Retrospective.” *Computer Music Journal* 37(3):49–67.

Mehra, R., et al. 2012. “An Efficient GPU-Based Time Domain Solver for the Acoustic Wave Equation.” *Applied Acoustics* 73(2):83–94.

Motuk, E., et al. 2007. “Design Methodology for Real-Time FPGA-Based Sound Synthesis.” *IEEE Transactions on Signal Processing* 55(12):5833–5845.

Nickolls, J., et al. 2008. “Scalable Parallel Programming with CUDA.” *ACM Queue* 6(2):40–53.

Perry, J., S. Bilbao, and A. Torin. 2016. “Hierarchical Parallelism in a Physical Modelling Synthesis Code.” In G. R. Joubert et al., eds. *Parallel Computing: On the Road to Exascale*. Amsterdam: IOS, pp. 207–216.

Pfeifle, F., and R. Bader. 2015. “Real-Time Finite-Difference Method Physical Modeling of Musical Instruments Using Field-Programmable Gate Array Hardware.” *Journal of the Audio Engineering Society* 63(12):1001–1016.

Press, W., et al. 1992. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge, UK: Cambridge University Press.

Puckette, M. 1991. “Combining Event and Signal Processing in the Max Graphical Programming Environment.” *Computer Music Journal* 15(3):68–77.

Raman, S., V. Pentkovski, and J. Keshava. 2000. “Implementing Streaming SIMD Extensions on the Pentium III Processor.” *IEEE Micro* 20(4):47–57.

Saad, Y. 2003. *Iterative Methods for Sparse Linear Systems*. Philadelphia, Pennsylvania: Society for Industrial and Applied Mathematics.

Samson, P. 1980. “A General-Purpose Digital Synthesizer.” *Journal of the Audio Engineering Society* 28(3):106–113.

Savioja, L., V. Välimäki, and J. O. Smith. 2010. “Real-Time Additive Synthesis with One Million Sinusoids Using a GPU.” In *Proceedings of the 128th AES Convention*, paper 7962. Available online at www.aes.org/e-lib/browse.cfm?elib=15259 (subscription required). Accessed February 2020.

Savioja, L., V. Välimäki, and J. Smith. 2011. “Audio Signal Processing Using Graphics Processing Units.” *Journal of the Audio Engineering Society* 59(1–2):3–19.

Southern, A., et al. 2010. “Finite Difference Room Acoustic Modelling on a General Purpose Graphics Processing Unit.” In *Proceedings of the 128th AES Convention*, paper 8028. Available online at www.aes.org/e-lib/browse.cfm?elib=15325 (subscription required). Accessed February 2020.

Strikwerda, J. 2004. *Finite Difference Schemes and Partial Differential Equations*. Philidelphia, Pennsylvania: Society for Industrial and Applied Mathematics.

Torin, A., B. Hamilton, and S. Bilbao. 2014. “An Energy Conserving Finite Difference Scheme for the Simulation of Collisions in Snare Drums.” In *Proceedings of the International Conference on Digital Audio Effects*, pp. 145–152.

Zhang, Q., L. Ye, and Z. Pan. 2005. “Physically-Based Sound Synthesis on GPUs.” In *Entertainment Computing: ICEC 2005*, 3711/2005. Berlin: Springer, pp. 328–333.