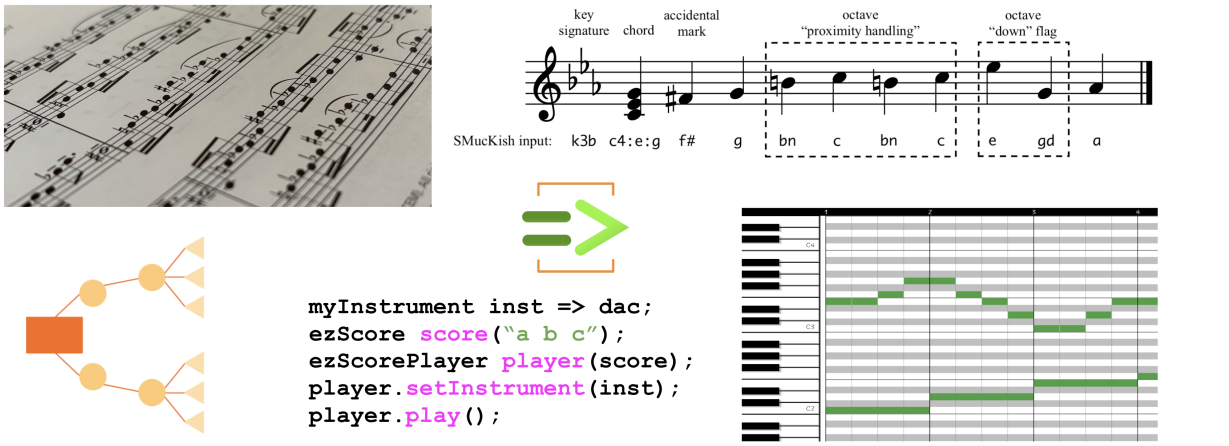
SMucK: Symbolic Music in ChucK

Alex Han
tae1han@ccrma.stanford.edu
Stanford University
Stanford, California, USA

Kiran Bhat
kvbhat@ccrma.stanford.edu
Stanford University
Stanford, California, USA

Ge Wang
ge@ccrma.stanford.edu
Stanford University
Stanford, California, USA



Abstract

SMucK (Symbolic Music in ChucK) is a library and workflow for creating music with symbolic data in the ChucK programming language. It extends ChucK by providing a framework for symbolic music representation, playback, and manipulation. SMucK introduces classes for scores, parts, measures, and notes; the latter encode musical information such as pitch, rhythm, and dynamics. These data structures allow users to organize musical information sequentially and hierarchically in ways that reflect familiar conventions of Western music notation. SMucK supports data interchange with formats like MusicXML and MIDI, enabling users to import notated scores and performance data into SMucK data structures. SMucK also introduces SMucKish, a compact high-level input syntax, designed to be efficient, human-readable, and live-codeable. The SMucK playback system extends ChucK's strongly-timed mechanism with dynamic temporal control over real-time audio synthesis and other systems including graphics and interaction. Taken as a whole, SMucK's design philosophy treats symbolic music data not only as static representations but also as mutable, recombinant building blocks for algorithmic and interactive processing. By integrating symbolic music into a strongly-timed, concurrent programming language, SMucK's workflow goes beyond data representation and playback, and opens new possibilities for algorithmic composition, instrument design, and musical performance.

Keywords

ChucK, programming, music notation, symbolic music

NIME '25, June 24–27, 2025, Canberra, Australia

1 Introducing SMucK

SMucK exists to provide both familiar and novel ways of creating music with ChucK[23]. It draws upon existing music information systems used in graphical notation and music analysis, but recontextualizes these systems within the creative workflow of ChucK.

SMucK addresses what has been absent in ChucK since its inception: built-in tools and abstractions for working with symbolic data. Normally, ChucK programmers must create their own symbolic representations of high-level musical concepts like pitch and rhythm. While, in theory, this "do-it-yourself" approach can allow for extremely customizable development, in practice the difficulty and scope of labor is often technically and artistically limiting. SMucK attempts to strike a balance between providing ready-to-use abstractions of common musical structures and allowing users to further extend and adapt these tools into their own workflows.

In addition to supporting parsing of commonly used data formats like MusicXML[2] and MIDI[16], SMucK introduces its own input syntax, SMucKish. This custom input syntax is designed to be efficient and readable, allowing users to quickly compose and edit musical material within ChucK's text-based coding environment.

SMucK goes beyond static representation of score material, taking advantage of ChucK's strongly-timed, concurrent programming model to allow for dynamic and precise control over playback. SMucK's playback system uses ChucK's *strongly-timed* mechanism to allow for dynamically editable, concurrent playback of multiple musical scores with independent timing and tempo manipulation. Moreover, SMucK introduces a new framework for virtual instruments in ChucK, allowing its symbolic scores to be linked to sound synthesis in a tightly integrated and highly customizable way.

Overall, these features make SMucK well suited for exploring compositional ideas, live-coding, and designing interactive systems. SMucK is designed for and by musicians and creative coders.

It is a tool aimed at encouraging exploration and expressivity. In this paper, we will discuss SMucK's workflow, core features, implementation, use cases, and future directions.

2 Related Work

Programming languages designed for music computation must often deal with behaviors and capabilities that can be difficult to implement in general-purpose programming languages. Music involves precise and complex organization of sounds over time, thus placing certain demands on computer music languages. Synthesizing audio signals requires sample-synchronous processing, at audio sample rates typically on the order of 44,100 samples per second. The sounds themselves may be scheduled as musical events (i.e. notes) occurring both sequentially in time and in parallel, requiring precise time scheduling and concurrent processing. Furthermore, to support real-time interactivity, languages must allow for on-the-fly control of signal processing parameters and the scheduling or manipulation of musical events as they occur.

Various computer music languages have been developed over the past 60 years which introduce specialized semantics, syntax, data types, and workflows that aim to address different aspects of these demands. Some languages, like Faust [13], focus solely on audio signal processing, but have no provisions for sequencing of note events. In contrast, languages like ABC[21], Guido[7], Adagio[4], and MusicXML[6] are designed for rich representation of musical scores. However, these languages are intended to encode fixed compositions, often for the purpose of graphical score rendering or computational analysis of a corpus of musical works rather than sound synthesis and performance.

Other languages aim to accommodate both signal processing for sound synthesis and sequencing of musical events. Notably, the MUSIC-N family of languages [9] introduced separate "orchestra" and "score" languages, allowing virtual "instruments" defined by audio signal computing graphs to be connected to lists of note events. A number of MUSIC-N's contemporaries have aimed to more tightly integrate "orchestra" and "score" into one programming framework. SuperCollider[10] runs on a client-server architecture that separates a real-time audio engine *scsynth* from the interpreted compositional language *sclang*, and allows these separate systems to communicate via the OSC messaging protocol. Languages like Max/MSP[14] and PureData[15] allow users to work with both signal flow graphs and control messages in a graphical programming framework, but can arguably make certain programming behaviors like abstraction, functions, classes, recursion, and instancing somewhat difficult.

Nyquist[5] and ChucK[3] are examples of languages that remove the distinction between "orchestra" and "score" altogether. Nyquist offers signal processing in a functional programming style, fine-grained control over timing and tempo information, and even the ability to write score information with the Adagio notation language. Like Nyquist, ChucK allows users to program sound synthesis and control structures in a unified framework. Moreover, ChucK provides a unique concurrent, temporally deterministic, sample-synchronous timing framework. There is no explicit global control rate in ChucK; control rate is an implicit consequence of how users dynamically move through time in the language. With ChucK's model of concurrent processing, it is also possible to have multiple processes ("shreds") executed simultaneously with their own control rates. Unlike Nyquist, which is not designed for real-time interactivity, ChucK is well-suited for live-coding and interactive music performance: code can be edited and re-run "on-the-fly" without interrupting the audio stream. Furthermore, its object-oriented framework also provides a way to combine sample-level audio computation with asynchronous real-time manipulation of synthesis parameters and events.

Each of the aforementioned programming languages offers a unique approach to music computation. In designing SMucK, we draw inspiration from many of these languages in order to bridge gaps between them. ChucK provides a natural home for this syncretic approach, as it already blends sound synthesis, control structures, and real-time interactivity in one language. However, unlike Nyquist, it does not have an embedded notational language (like Adagio) that supports handling of rich symbolic information. We take inspiration from Nyquist's integration with Adagio as well as dedicated notational languages like Lilypond[12] and SCORE[18] for the design of our own ChucK-native notation syntax SMucKish.

In an interesting twist, SMucK also reintroduces the distinction between "orchestra" and "score" into a computer music language that was designed to remove this very distinction. SMucK differentiates between "orchestra" and "score" not as categorically distinct languages (as with CSound and its MUSIC-N predecessors [9, 17, 20]), but as constructs within the language. This approach aims to provide the "best of both worlds": a programmer can choose to work with SMucK's orchestra-score framework to the extent that it suits their technical and aesthetic needs.

3 The SMucK Approach

In ChucK, typical workflows for handling symbolic note events might include sequencing arrays of MIDI note numbers which can be translated to unit generator frequencies, and advancing time directly using the keyword *now*. This behavior might be encapsulated in some sort of "play note" function to be called repeatedly and concurrently during runtime. This open-ended, *do-it-yourself* approach is part of ChucK's design, and is meant to provide a flexible, expressive environment.

However, it means that certain common musical structures and behaviors can be difficult to realize: handling polyphonic material, controlling dynamic tempo changes, and inputting and editing complex scores all require bespoke, ground-up solutions. The relative difficulty of realizing these kinds of behaviors also shapes the art created with ChucK–for instance, it is easier to develop a simpler, loop-based treatment of pitch and rhythm content, while controlling signal-processing parameters governing timbre in a more granular way.

In designing SMucK, we aim to unite the representational richness of music notation-focused languages with the integrated, dynamic computation offered by ChucK. For our input syntax SMucKish, we take inspiration from languages like ABC, Lilypond, and especially Leland Smith's SCORE [18]. However, instead of tailoring the syntax towards graphical score rendering or analysis, we design our input syntax with real-time playback and manipulation of score contents in mind.

With the SMucK playback workflow, we provide high-level abstractions to handle some of the complex behaviors mentioned above: handling polyphony, dynamic tempo control, and linking of symbolic scores to audio synthesis chains. Importantly, these high-level abstractions retain fine-grained controllability and customizability, preserving ChucK's core design principles of flexible and expressive code.

4 Workflow

The overall SMucK workflow encompasses several stages which we will summarize here, before going over each part in more detail. First, users create a score by instantiating an ezScore. Next, users write or import the score content. This can be done by importing a MIDI or MusicXML file, or by writing musical material with the SMucKish input syntax.

Once the score is complete, users can play back the material with a two-step process. Firstly, users define ezInstrument classes which contain sound synthesis chains–these are analogous to "synth patches" which set up a signal flow that performs notes as they are received. Secondly, users create an ezScorePlayer object that connects the "orchestra" to "score". The score player sets up a virtual timeline for the score material and sends note data to the ezInstrument objects. The score player's playback rate, position, and looping behavior can all be controlled by the user. This workflow can be seen in the example code snippet below. In the next few sections, we will dive deeper into each step of this process, starting with the ezScore family of classes.

```
// create score from a simple pitch
// sequence (using SMucKish input)
ezScore score("a4 b c d e");

// Define an instrument to play back the score
class myInstrument extends ezInstrument
{
    // Sound synthesis chain
    Sitar sitar => NRev reverb => outlet;

    // Define note-on behavior
    fun void noteOn( ezNote note, int voice )
    {
        // Set the pitch
        Std.mtof( note.pitch() ) => sitar.freq;

        // Pluck the sitar
        note.dynamics() => sitar.noteOn;
    }


    // Define note-off behavior
    fun void noteOff( ezNote note, int voice )
    {
        sitar.noteOff();
    }
}

// Connect our instrument to audio output
myInstrument inst => dac;

// Create player object to play back the score
ezScorePlayer player(score);

// Set instrument(s) for player
player.setInstrument( [inst] );

// Start playback
player.play();

// Time loop
while( player.isPlaying() ) 1::second => now;
```

4.1 The ezScore Class Family

SMucK introduces a set of objects used to store musical score data, implemented as custom ChucK classes (which are therefore extensible and inspectable by users). This data can be parsed from MIDI files, MusicXML files, or from the SMucKish input syntax. These classes are organized hierarchically, representing musical information from individual notes to multi-part polyphonic scores. The hierarchy of objects is as follows:

$$\text{ezNote} \rightarrow \text{ezMeasure} \rightarrow \text{ezPart} \rightarrow \text{ezScore}$$

4.1.1 ezNote. At the lowest level are ezNote objects. Each ezNote has an associated pitch, rhythmic value, onset in beats relative to the measure start, and dynamics value. See Table 1 for a summary.

Rhythms and onsets are stored as symbolic beat values rather than durations in time. This is in contrast to data formats like MIDI messages, which rely on pairs of 'note on' and 'note off' events which are received sequentially in time offsets. Currently, ezNote objects only contain information relating to pitch, rhythm, and dynamics. However, we plan to expand this by allowing user-defined information (e.g. text annotations, abritrary numerical values) to be attached to ezNote objects.

4.1.2 ezMeasure. One level up is the ezMeasure object, which contains ezNote objects. ezMeasure objects also have a `length`(in beats) and `onset` (relative to the score's start). Each ezMeasure object can be arbitrarily long; meter/time signature is not enforced. The meaning of a measure in SMucK is looser than that of a traditional score, and can be thought of more as an intermediate-level data structure to hold a collection of notes. This is done to maintain as much flexibility in editing, generation, and playback as possible. Crucially, notes can be appended, edited, deleted, or inserted during runtime, allowing the sort of dynamic computation not available in static score representations.

4.1.3 ezPart. Above the ezMeasure level is the ezPart, which represents a single part within a score. Each ezPart contains a sequence of ezMeasures. Each ezPart can be connected to an ezInstrument during playback, assigning an audio synthesis chain on a partwise basis. This is explained further in section 4.3.2. Just like with notes within measures, the measure contents of an ezPart are editable during runtime.

4.1.4 ezScore. Finally, above the ezPart is the ezScore, which represents a multi-part polyphonic score. This top-level data structure contains a number of ezPart objects and is the object that interacts with the playback system. It is also the data structure that external data formats are parsed into: MIDI and MusicXML files can be imported as ezScore objects containing ezParts, ezMeasures, and ezNotes. Alternatively, ezScore objects may be built from the bottom up, using SMucKish to specify individual notes, measures, parts, or whole scores. In the next section, we will elaborate on how the SMucKish input system works and how it interacts with the ezScore family of data structures.

4.2 SMucKish Input Syntax

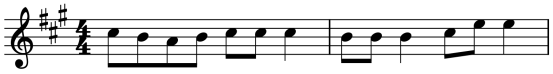
The SMucKish input syntax is a way to write musical score content within ChucK. It is heavily influenced by languages like ABC, Lilypond, and SCORE, emphasizing efficiency, local context awareness, and human readability. It can be parsed directly into arrays of pitch, rhythm, and dynamics values, or it can be parsed into the ezScore-family data structures outlined in the previous section.

SMucKish input is written as a ChucK string consisting of individual tokens delimited by spaces. Taking inspiration from SCORE, there are different conventions for specifying pitch, rhythm, and dynamics. Unlike SCORE, each of these layers can be entered and parsed separately in a "non-interleaved" format, or simultaneously with an "interleaved" format.

Table 1: ezNote properties

Property	Meaning	Data type
pitch	MIDI note number (0-127), with special value -999 to denote a rest	int
beats	Rhythmic value in beats, where quarter note = 1.0, eighth note = .5, etc.	float
onset	Onset in beats, relative to the start of the measure	float
dynamics	Value representing the dynamics of the note, ranging from 0.0 to 1.0	float

As a simple example, to encode the following musical phrase in a non-interleaved way:



The following SMucKish code could be used to represent the pitches:

```
"k3# c5 b a b c c c b b b c e e"
```

And the following code to represent the rhythms:

```
"e e e e e e q e e q e e q"
```

As an example of some of the syntactic sugar provided by SMucKish, the following code could be used to more compactly represent the same rhythms:

```
"ex6 [q e e]x2 q"
```

With the "interleaved" method, pitch and rhythm can be entered simultaneously (note the use of the `|` symbol, which "binds" pitch and rhythm tokens in the following example):

```
"k3# c5|e b a b c c c|q b|e b b|q c|e e e|q"
```

We will now specify the syntax for each layer in more detail.

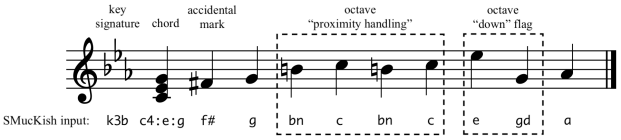
4.2.1 Pitch. Each pitch token consists of 3 parts: a pitch step, accidental, and octave. The pitch step is a single character representing the note name (e.g. `a`, `b`, `c`) and is required. The pitch step `r` denotes a rest. The accidental and octave parts may be multiple characters long and are optional.

Pitch tokens can have an arbitrary number of accidental marks (`#` or `s` for sharp, `b` or `f` for flat, and `n` for natural). Key signatures can be specified using a special token of the format `knx` where `n` is a number of flats or sharps and `x` is the accidental type (using the same characters mentioned above). For instance, the key signature token `k3#` means "key 3 sharps", or A major/F# minor. Key signatures can be set at any point in the pitch layer and persist until overridden.

Octave numbers can be explicitly set: `c5` means C5. If the octave number is omitted, SMucKish assumes the pitch is in the octave that puts it closest to the previous pitch. This "proximity awareness" can be helpful in handling octave crossing situations. For example, `a4 b c` is equivalent to `a4 b4 c5`. Additionally, the `u` or `d` flags can be used to move up or down an octave relative to the previous note.

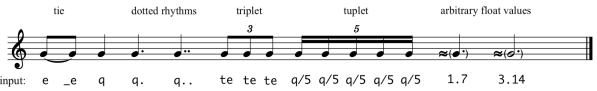
Chords can be entered by linking multiple pitch tokens with the `:` character. Here is an example showing the use of chords, key signatures, accidentals, and octave handling:

```
ezMeasure measure("k3b c4:e:g f# g bn c bn c e gd a");
```



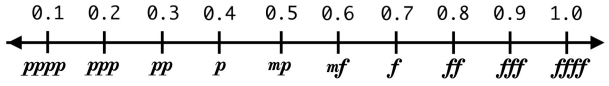
4.2.2 Rhythm. Rhythm tokens represent beat values. Basic subdivisions can be specified using `w` for whole, `h` for half, `q` for quarter, `e` for eighth, and `s` for sixteenth notes. Dotted rhythms are specified using `.` characters. Tuplets are specified either using the character `t` as a prefix, denoting a triplet, or using the suffix `/n` to specify a tuplet dividing the rhythmic value into `n` equal parts. Ties can be entered using the prefix `\_`. Lastly, arbitrary beat values can be directly written as float values. Here is an example incorporating all of these elements:

```
ezMeasure measure("e _e q q. q.. te te te
                   q/5 q/5 q/5 q/5 q/5 1.7 3.14");
```



We acknowledge that our naming convention for rhythm tokens could be confusing for users who do not think in English. We chose these tokens based upon the use of English within ChucK, and the use of similar naming conventions in other score-entry formats, such as Adagio [4] and SCORE [18].

4.2.3 Dynamics. Dynamics are represented as values ranging from 0.0-1.0. When importing MIDI files, which use integers 0-127 to represent velocity, values are normalized to this range. In SMucKish, values can be written with dynamic symbols ranging from `pppp` to `ffff`. These symbols are mapped across the 0.0-1.0 range in increments of 0.1, as seen in the figure below. Alternatively, arbitrary values can be specified directly, prefixed with a `d` (e.g. `d.5`, `d.72`).



4.2.4 Repeated Tokens and Sequences. SMucKish also allows users to repeat individual tokens or sequences of tokens in any of the aforementioned layers. By adding the suffix `xn`, a token will be repeated `n` times. Multi-token sequences can be enclosed with brackets `[` and `]`, followed by `xn` to specify how many times the sequence will be repeated.

```
ezMeasure measure("hx3 [q e. s]x2 w");
// is equivalent to
ezMeasure measure("h h h q e. s q e. s w");
```

4.2.5 Chords and Scales. SMucK also provides functions for parsing chord symbol notation and scales. Chord symbols commonly used in jazz and popular music (e.g. "G#m", "Cmaj7", "Bb7#9b13")

can be translated into MIDI note numbers. Similarly, scale names (e.g. "minor", "mixolydian", "double harmonic") combined with a given root note, can also be parsed into MIDI note numbers. Additional specifications and conventions are supported, but we will not elaborate on them in this paper. These parsing tools currently exist separately from the SMucKish input syntax, although we plan to integrate them in the future.

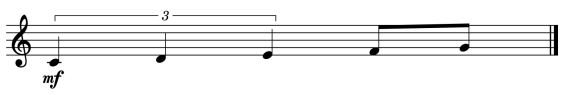
4.2.6 Putting it All Together. Each of these layers–pitch, rhythm, and dynamics–can be parsed individually into ChucK arrays or into ezNote objects within an ezMeasure. In the former case, pitches are parsed into a 2D integer array of MIDI note numbers, with the first dimension representing temporal order ("horizontal") and second dimension concurrent notes ("vertical"). Rhythms and dynamics are both parsed as float arrays. This way, users have the freedom to use SMucKish input syntax without necessarily using ezScore-family classes.

Parsing of SMucKish into ezNotes in an ezMeasure can be performed in multiple stages, or all at once using an "interleaved" format. In multi-stage entry, users create an ezMeasure object, then set each layer separately. For example, the following code:

```
ezMeasure measure;

measure.setPitches("c4 d e f g");
measure.setRhythms("tqx3 e e");
measure.setDynamics("mfx5");
```

Is equivalent to the following notated measure:

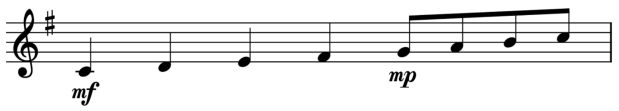


Layers can be set in any order. In the case that the number of tokens does not match between layers, missing tokens are "filled in" according to the last valid token parsed.

The "interleaved" approach to writing SMucKish uses the | character to connect tokens from different layer types (pitch, rhythm, and dynamics, in that order). This makes it possible to specify a measure's contents in a single string. If one of the layers is not specified for a given token, its value will be inferred from the previous note, as in multi-stage input. For instance, the following code:

```
ezMeasure measure("k1# c4|q|mf d e f g|e|mp a b c");
```

Is equivalent to the following notated measure:



From here, users can build up a longer, multi-part score. Individual measures can be added to an ezPart, and multiple parts can be packed into an ezScore. Here is a simple example of building a score with a single part and two measures:

```
// Create a score
ezScore score;

// Create a part
ezPart part;

// Create a measure using SMucKish add it to the part
ezMeasure measure("a b c d");
part.addMeasure(measure);

// Create another measure and add it to the part
ezMeasure measure2("c d e f");
part.addMeasure(measure2);

// Add the part to the score
score.addPart(part)
```

This process is flexible and can operate at any level of the ezScore class hierarchy–users can construct an ezScore directly from an interleaved SMucKish string, set an ezPart in multiple stages using the non-interleaved format, and so on.

4.3 Playback

Once users have input their score data into an ezScore object using the methods described above, they can play back the score. SMucK's playback system is designed to handle several complex behaviors "under the hood". It connects the symbolic score data to sound synthesis, handles timing of note events, and allocates voices for polyphonic material, among other behaviors. This is done through the ezScorePlayer object. Users create an ezScorePlayer and assign an ezScore to be played back.

4.3.1 Score Previewing. Users can immediately hear their score played back without designing any sound synthesis chains by using the `.preview()` function. This uses a built-in "default instrument" that uses sine oscillators with an amplitude envelope. This can be accomplished in just three lines of code:

```
ezScore score("[a4:c:e|q.]x2 a3:d:f|q");
ezScorePlayer player(score);
player.preview();
```

The `.preview()` function allows users to instantly translate their encoded score material into sound. This can be useful for debugging their score entry or rapidly iterating on compositional ideas.

4.3.2 Creating an ezInstrument. To go beyond the default sound design of `.preview()`, users can define "instruments" by extending SMucK's ezInstrument base class. Once defined, these instruments are connected to the ezScorePlayer on a part-wise basis–each part within the score can be played back by a different instrument. Each ezInstrument has a `noteOn()` and `noteOff()` function which receive ezNote data from the ezScorePlayer during playback. In the base class, these functions are empty–they are designed to be overridden by the user, who decides how to map the ezNote parameters to their own signal processing chain. As an example of a fully implemented ezInstrument, here is a simple polyphonic instrument similar to the one used in `.preview()`:

```
class ezPreviewInst extends ezInstrument
{
    // Set up signal flow
    20 => n_voices;
    SinOsc oscs[n_voices];
    for(int i; i < n_voices; i++)
    {
        oscs[i] => outlet;
    }

    // User-defined noteOn function
    fun void noteOn(ezNote note, int voice)
    {
        // map note's pitch to oscillator's frequency
        Std.mtof(note.pitch()) => oscs[voice].freq;

        // map note's dynamics to oscillator's gain
        note.dynamics() => oscs[voice].gain;
    }
    // User-defined noteOff function
    fun void noteOff(ezNote note, int voice)
    {
        // set the oscillator's gain to zero
        0 => oscs[voice].gain;
    }
}
```

Users don't need to call the `noteOn` and `noteOff` functions themselves–these functions are automatically called by the ezScorePlayer, which passes the current note information and routes it to individual "voices" in the signal chain if the score is polyphonic. Once users define their ezInstruments, they connect them to individual parts. Suppose the user loads in a MIDI file with two parts, and has defined two ezInstruments called `myInst` and `myInst2`. Here's how they could link their custom instruments to the score:

```
// Load in a MIDI file with two parts
ezScore score("bwv784.mid");

// Instantiate the two user-defined
// instruments, "myInst" and "myInst2"
myInst inst1;
myInst2 inst2;

// Create the score player
ezScorePlayer player(score);

// Set myInst to part 0, and myInst2 to part 1
player.setInstrument(0, inst1);
player.setInstrument(1, inst2);

// Start playback
player.play();

// Advance time
while(player.isPlaying())1::second => now;
```

4.3.3 Playback Control. Once the ezScorePlayer is set up with an ezScore and ezInstruments, its playback behavior can be controlled dynamically. Users can change the playback rate, go to different positions in the score, and loop sections, all in real-time. Because SMucK scores are editable during runtime, this is a natural setting for live-coding: users could loop playback, make changes to the score contents, jump to different positions, and swap instruments all on-the-fly. Additionally, dynamic rate change is normally a challenge when dealing with concurrent processes, but with SMucK's playback system it is as easy as setting the ezScorePlayer's `.rate()`. In fact, since playback is handled for each ezScorePlayer independently, it is possible to play multiple scores concurrently with different tempi and control each of these tempi in real-time. The implementation of SMucK's "virtual playhead" mechanism is discussed further in sections 5.1 and 6.1.

With SMucK, the act of creating a score, setting up the score player, and playing back the score can be accomplished in a few lines of code. The setting up of ezInstrument objects is the part that requires more "work" on the programmer's behalf. This is intentional–SMucK's focus is on providing the ability to represent scores and handle their *interaction* with synthesis, not on the synthesis itself. This gives creative freedom for users to decide what kind of behavior they would like to happen as symbolic score data is read through time.

In the most common use case, users might use incoming notes' pitch and dynamics values to control frequency and gain of unit generators. However, users could also map dynamics values to a filter cutoff, or play a certain sound file if the note's pitch is higher than Ab5, or generate graphical elements when the pitch belongs to a certain scale. Our hope is that building a framework full of abstractions and conveniences actually enables and inspires users to do *more* with the language, not less.

5 Implementation

SMucK is implemented entirely in ChucK, as opposed to C++ modules or "chugins" to the language. SMucK's ezScore family of data structures are custom ChucK classes and the SMucKish input syntax parsing happens via operations on ChucK strings. It would certainly have been possible to implement these features in a lower-level language, but also not strictly necessary. SMucK's playback system, however, is more intimately ChucKian in its treatment of time, and so building out this component within ChucK itself was helpful. We will now briefly discuss some of our methods with respect to the playback system.

5.1 Symbolic Time

At a high level, SMucK's playback system utilizes a virtual playhead, which moves across a score, to notify each instrument what notes to play or release at any given moment in time. The playhead does not move continuously in time, but takes small discrete steps. We refer to the step size as a *tatum*, inspired by Jeff Bilmes' term which describes "the fastest pulse present in a piece of music" [1]. The playhead's position is updated repeatedly at a constant period, which we refer to as a *tick*, but the step size (i.e. *tatum*) itself depends on the playback rate. A faster playback rate demands a longer *tatum*, and a slower playback rate requires a shorter *tatum*. Even reverse playback is made possible with a negative *tatum* value.

The difference between a *tick* and *tatum* is related to the concept of *logical time*. ChucK distinguishes between *logical time* and *actual time*: *logical time* is "the deterministic accounting of time internal to an audio environment" (i.e. through counting audio samples), while *actual time* corresponds to "the continuous flow of time as we perceive it" (e.g. system clock or timer) [23]. One of ChucK's unique features is that it allows users to control *logical time* via the `now` construct, in a deterministic and sample-synchronous way. Logical time does not advance until the user explicitly tells it to do so.

SMucK builds on top of this by adding what we call symbolic time. Our playback system is concerned with progressing through a symbolic score representation. As such, the virtual playhead that indicates current position within that score lies on a further level of abstraction from ChucK's logical time. The *tick* corresponds to an actual movement forward in ChucK's logical time, but the *tatum* corresponds to a movement in SMucK's symbolic time. This update to symbolic time is essentially a fractional beat value, and can move backwards, as opposed to logical time. The playback rate is represented by a float value that scales the tatum length relative to the tick length, determining the mapping between symbolic and logical time.

A simplified version of the internal update function in the ezScorePlayer can be seen below:

```
// Update the tatum size
tick * rate => tatum;

// Advance symbolic time by one tatum
tatum +=> playhead;

// Advance logical time by one tick
tick => now;
```

The *tick* and *tatum* are both of type "dur", which is a ChucK primitive type that represents a duration of logical time (e.g. 1 millisecond). The resolution of the *tick* is set to 1 ms by default, but can be set directly by the user. This means that the resolution of score playback could theoretically be sample-accurate at the

audio sample rate (e.g. with a *tick* of 1 sample and playback rate of 1.0), although in practice this could lead to performance issues.

On top of the virtual playhead mechanism, our implementation of score playback includes a number of other components, such as voice allocation for polyphony and interaction with external devices and software via OSC [24] and MIDI [16]. However, these are outside the scope of this paper so we will not detail them here.

6 Use Cases

SMucK has a wide variety of use cases in the realm of live performance, instrument design, interactive graphics, score following, and more. At the time of writing, SMucK has not been officially released publicly. This means that we cannot yet showcase a large body of work created with SMucK. However, in this section we briefly share a few of our own case studies that capture some of what is possible with our system.

6.1 Dynamic Playback

In ChucK, it is straightforward to play musical notes at a fixed tempo, since their durations are decided ahead of time. However, it is surprisingly challenging to dynamically alter tempo during playback, since it requires modifying the duration of the notes in the midst of their synthesis. However, SMucK facilitates dynamic-tempo playback of complex scores, opening rich new possibilities for musical expression in ChucK.

As an example, let's observe how we can use an external controller, called a "GameTrak", in conjunction with SMucK's playback system to create an expressive, dynamic-tempo performance (see Figure 1). The GameTrak controller can be used to translate physical gestures to a stream of numerical data, and is frequently used in the Stanford Laptop Orchestra (SLOrk) [22].

```
// create score from a single-part MIDI file
ezScore score("my_midi_file.mid");

// create an instrument
class myInstrument extends ezInstrument
{
    // design whatever instrument we want!
}
myInstrument inst => dac;

// begin playback of our score
ezScorePlayer player(score);
player.setInstrument([inst]);
player.play();
player.loop(True);



// use the GameTrak controller to control the playback
// rate in real time
while(true)
{
    // retrieve the GameTrak position value
    // (between -1.0 and 1.0)
    get_gametrak_position() => float position;

    // map the GameTrak position to the playback rate
    player.rate( position * 5 );

    // wait 10 ms before the next GameTrak measurement
    10::ms => now;
}
```

This example might raise the question: why not just play a sound file and adjust the playback rate in real time? With audio files, we can control playback rate, but have little control over individual notes and their synthesis process. Conversely, in native ChucK we can algorithmically modify our notes and sound synthesis process, but dynamic tempo playback is challenging. But with SMucK, we have control over playback rate, note-level parameters, and synthesis all at once.

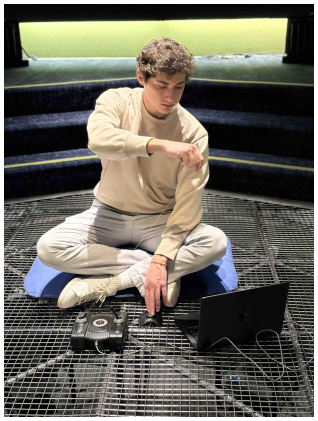


Figure 1: A student uses a GameTrak controller to dynamically control playback rate

6.2 Note-aware Instrument Design

The ezInstrument framework makes it possible to map note parameters to a wide variety of other behaviors. This can lead to potentially unconventional and artistically interesting interactions between symbolic data and synthesis. Here are a couple of examples.

If we want an instrument that increases vibrato with note duration, we can design an ezInstrument with the following noteOn function:

```
Bowed bow => outlet;
...
fun void noteOn(ezNote note, int voice)
{
    // longer notes -> increased vibrato
    (note.beats / 16) => bowed.vibratoGain;

    // design rest of noteOn() function as normal
}
```

If we want an instrument that plays a score "upside down" (lowest pitches become the highest, and vice versa), we can design an ezInstrument with the following noteOn function:

```
SinOsc osc => outlet;
...
fun void noteOn(ezNote note, int voice)
{
    // low notes become high, and vice versa
    Std.mtof(127 - note.pitch) => osc.freq;

    // design rest of noteOn() function as normal
}
```

There are endless possibilities for designing your instruments, and using the symbolic music data in creative ways. The combination of SMucK's instrument design and dynamic playback capabilities presents users with an exciting new sandbox to build expressive performances.

6.3 Graphical and Interactive Scores

Another powerful use case of SMucK is the ability to tie graphics and interaction to a score's symbolic data and playback. As a case study, let's examine a 3D platforming game named *The Trebled C*, written using ChucK's new audiovisual programming framework ChuGL [25].

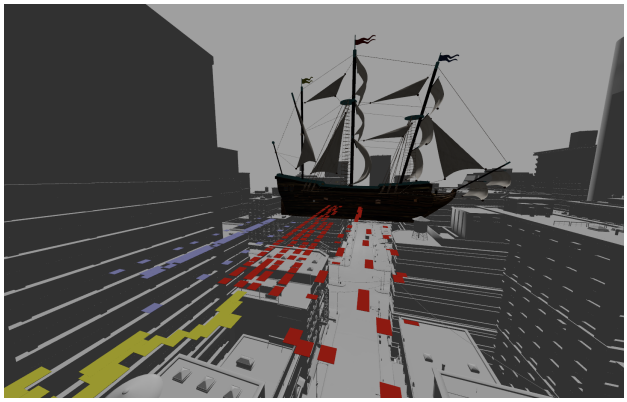


Figure 2: Screenshot from *The Trebled C*

In *The Trebled C*, the player must run away from a flying pirate ship by jumping across a series of platforms. The backing music and platform placement are linked to an ezScore, whose playback is responsive to the player's movement – as a player moves forward, the music begins and platforms are placed in front of their feet. Additionally, the position and scale of these platforms is linked to the score, such that the platforms are organized in the form of a piano roll from a bird's eye view. (See Figure 2) Here's how we drive graphics in *The Trebled C* with SMucK:

```
class platformInstrument extends ezInstrument
{
    // set up sound chain
    ...
    fun void noteOn(ezNote note, int voice)
    {
        // compute platform position
        // using note.pitch() and note.onset()

        // compute platform size using note.beats()

        // set platform 3D position and size

        // play note!
    }
}

ezScore score("level_one.mid");
ezScorePlayer score_player(score);
platformInstrument inst => dac;
score_player.setInstrument([inst]);
score_player.play();

while (true)
{
    // use player's in-game velocity to control the
    // score playback rate
    avatar.velocity() => score_player.rate();

    // advance graphics/audio frame
    GG.nextFrame() => now;
}
```

The playback of the score is tied to the player's movement, and the placement of the platforms occurs as a result of playback. Each call to noteOn() during playback spawns a new platform, which is offset to the left or right based on the note's pitch. This project exemplifies the ways SMucK's score representation and playback can be used beyond the sound domain, and drive audiovisual interactive systems.

7 Reflections and Future Work

> *"If Human Computer Interaction (HCI) research strives to give people greater and better access to using the computer, then perhaps computer music language design aims to give programmers more natural representation of audio and musical concepts."*

The above sentiment comes from a 2005 paper from the creators of ChucK [3] outlining the philosophies underlying the language's design. It perfectly captures the ethos of SMucK–at its heart, our project is about providing natural representation of musical concepts. SMucK is neither the first nor the most comprehensive system for representing musical information symbolically. However, we would argue that it certainly provides an accessible, versatile, extensible, and indeed natural workflow within ChucK's programming environment.

SMucK is fundamentally about bridging different kinds of workflows used in computer music. It draws inspiration from graphic notation systems, symbolic score entry languages, digital audio workstations, and interactive live-coding environments. Each of these offers unique strengths and weaknesses. The overarching goal of SMucK is to draw upon the affordances of all of these systems: it borrows the compactness and readability of notation-focused languages, the hierarchical organization of score information found in graphic score editors and DAWs, and the customizability and dynamic computing of live-coding frameworks. ChucK, with its strongly-timed concurrent programming model, provides the perfect home for SMucK to bring these workflows together.

Our project is still in its infancy. It has not been widely tested and used by musicians and coders, and its features are still actively being improved upon and added to. In the future, we plan to introduce our tools in computer music courses at Stanford University's Center for Computer Research in Music and Acoustics (CCRMA). We also plan to document works created using our system and collect feedback about students' experience with SMucK, both from a technical and artistic standpoint. We hope that, as people start to use our tools, we will gather additional insight on what SMucK should become and how it can best serve the computer music community.

There are many features we plan to implement in the future. On the representation front, we are working on allowing ezNote objects to carry more kinds of information, including MIDI-CC-like messages, additional articulation and expression markings, text annotations, and user-defined data structures. We are also considering including ABC and Lilypond as usable input syntaxes alongside SMucKish.

We also plan to add more functionality supporting live coding and algorithmic composition, including abstractions for manipulating score contents programmatically (e.g. shuffling contents, arpeggiating chords, and systems for generating SMucKish tokens). We are exploring ways to expand on our playback system to allow users to create "branching scores" that follow non-linear trajectories. We imagine a score with "transition points" with multiple potential musical pathways, where the transition conditions can be defined by the user.

In recent years, ChucK development has become increasingly active [19]; ChucK now features an extensive audiovisual programming framework in ChuGL [25], web integration with WebChucK [11], and interactive AI tools with ChAI [8]. These kinds of developments bolster each other and invite research and artmaking that links multiple methodologies and media. SMucK

contributes to this ecosystem by providing new capabilities in itself, but also in its potential integration with these new offerings.

We are excited to see what the future holds for SMucK. Our hope is that SMucK will continue to grow and mature into a powerful, versatile system that inspires new forms of expression. We believe that the affordances and constraints of a system intimately shape the creative process, and that by expanding what ChucK can do, we might diversify and enrich the art that people create with it.

8 Acknowledgements

We would like to thank the ChucK development team for their feedback, support, and company throughout the SMucK design process. We also thank Celeste Betancur and the students of Music 220B for their willingness to be the first users of our system. Additionally, we would like to thank Craig Sapp and Eleanor Selfridge-Field whose mentorship and expertise in symbolic music information inspired many of the ideas we used here.

9 Ethical Standards

SMucK has been developed with the support of CCRMA's departmental funding, curricular student research, and volunteer contributions. The authors are aware of no potential conflicts of interest.

References

[1] Jeff A Bilmes. 1993. *Techniques to foster drum machine expressivity.* Citeseer.
[2] World Wide Web Consortium et al. 2000. Extensible Markup Language (XML) 1.0 (Sec–ond Edition).
[3] Ge Wang Perry R Cook and Ananya Misra. 2005. Designing and implementing the chuck programming language. In *Proceedings of the 2005 International Computer Music Conference.*
[4] RB Dannenberg. 1998. The CMU MIDI Toolkit. Manual. *Center for Art and Technology, College of Fine Arts, CMU (Aug. 1986)* (1998).
[5] Roger B Dannenberg. 1997. Machine tongues XIX: Nyquist, a language for composition and sound synthesis. *Computer Music Journal* 21, 3 (1997), 50–60.
[6] Michael Good. 2001. MusicXML for Notation and Analysis. W. B. Hewlett and E. Selfridge-Field.
[7] Holger H Hoos, Keith Hamel, Kai Renz, and Jürgen Kilian. 1998. The GUIDO notation format: A novel approach for adequately representing score-level music. In *ICMC*, Vol. 98. 451–454.
[8] Yikai Li and Ge Wang. 2024. Chai: Interactive ai tools in chuck. In *New Interfaces for Musical Expression.*
[9] Max V Mathews, Joan E Miller, F Richard Moore, John R Pierce, and Jean-Claude Risset. 1969. *The technology of computer music.* the MIT Press.
[10] James McCartney. 2002. Rethinking the computer music language: Super collider. *Computer Music Journal* 26, 4 (2002), 61–68.
[11] Michael R Mulshine, Ge Wang, Jack Atherton, Chris Chafe, Terry Feng, and Celeste Betancur. 2023. Webchuck: Computer music programming on the web. In *New Interfaces for Musical Expression.*
[12] Han-Wen Nienhuys-Jan Nieuwenhuizen and H Nienhuys. 2003. Lilypond, a system for automated music engraving. In *Proceedings of the XIV Colloquium on Musical Informatics (konferenciaanyag).*
[13] Yann Orlarey, Dominique Fober, and Stéphane Letz. 2009. Faust: an efficient functional approach to dsp programming. *New computational paradigms for computer music* (2009), 65–96.
[14] Miller Puckette. 1991. Combining event and signal processing in the MAX graphical programming environment. *Computer music journal* 15, 3 (1991), 68–77.
[15] Miller S Puckette et al. 1997. Pure data. In *ICMC.*
[16] Joseph Rothstein. 1995. *MIDI: A comprehensive introduction.* Vol. 7. AR Editions, Inc.
[17] Bill Schottstaedt. 1994. Machine tongues XVII: CLM: Music V meets common lisp. *Computer Music Journal* 18, 2 (1994), 30–37.
[18] Leland Smith. 1972. Score-a musician's approach to computer music. *Journal of the Audio Engineering Society* 20, 1 (1972), 7–14.
[19] Marise van Zyl and Ge Wang. 2024. What's up ChucK? Development Update 2024. In *Proceedings of the International Conference on New Interfaces for Musical Expression.* 549–552.
[20] Barry Vercoe et al. 1986. Csound. *The CSound Manual Version* 3 (1986).
[21] Chris Walshaw. 2021. The abc music standard 2.1 (dec 2011).
[22] Ge Wang, Nicholas J Bryan, Jieun Oh, and Robert Hamilton. 2009. Stanford laptop orchestra (slork). In *ICMC.*
[23] Ge Wang, Perry R Cook, and Spencer Salazar. 2015. Chuck: A strongly timed computer music language. *Computer Music Journal* 39, 4 (2015), 10–29.
[24] Matthew Wright. 2005. Open Sound Control: an enabling technology for musical networking. *Organised Sound* 10, 3 (2005), 193–200.
[25] Andrew Zhu and Ge Wang. 2024. ChuGL: Unified Audiovisual Programming in ChucK. In *Proceedings of the International Conference on New Interfaces for Musical Expression.* 351–358.