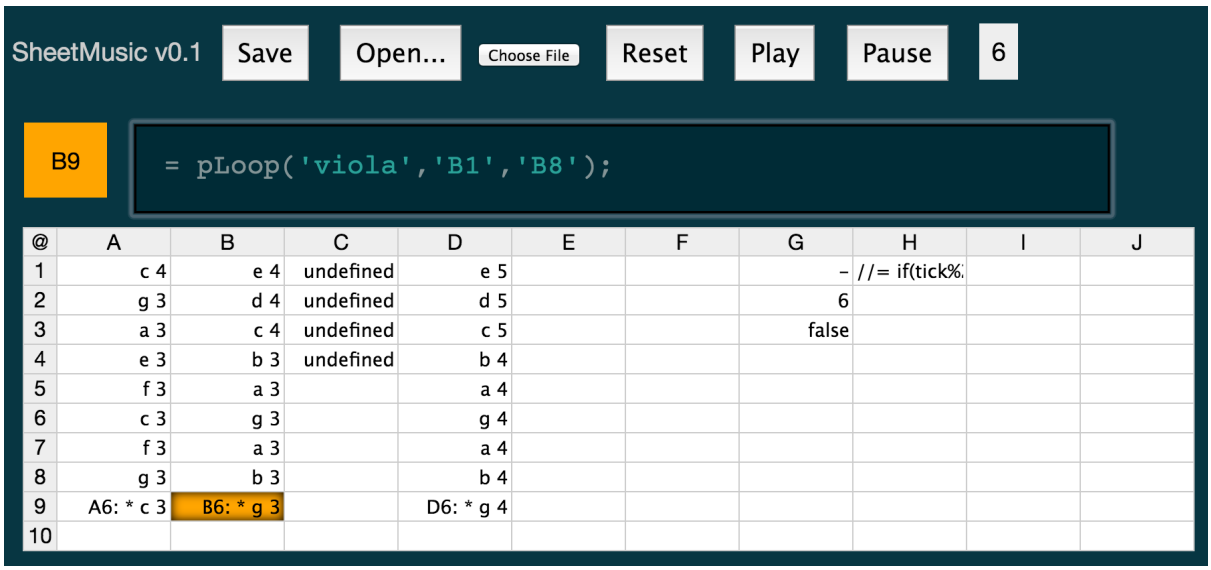
Towards spreadsheet tools for end-user music programming

Advait Sarkar
Computer Laboratory
University of Cambridge
advait.sarkar@cl.cam.ac.uk

Abstract

SheetMusic, an early-stage prototype, explores how spreadsheets can be used as accessible end-user tools for music programming and data sonification. This design probe uncovers many interesting questions: what are the primary advantages of the spreadsheet paradigm in this context? Should such a tool be regarded as a musical instrument, or as a way to create musical 'programs' with emergent runtime behaviour? How can musical experiences be 'programmatic'? How sophisticated should provisions for scripting the tool be? How can time be represented? Each issue is considered in turn, drawing on previous work in live music coding, end-user programming, and the current SheetMusic implementation.

1. Introduction



@	A	B	C	D	E	F	G	H	I	J
1	c 4	e 4	undefined	e 5			-	//= if(tick%		
2	g 3	d 4	undefined	d 5			6			
3	a 3	c 4	undefined	c 5			false			
4	e 3	b 3	undefined	b 4						
5	f 3	a 3		a 4						
6	c 3	g 3		g 4						
7	f 3	a 3		a 4						
8	g 3	b 3		b 4						
9	A6: * c 3	B6: * g 3		D6: * g 4						
10										

Figure 1 – The SheetMusic prototype.

SheetMusic is an exploration of how formulae with musical sound effects can be integrated into the spreadsheet paradigm. Figure 1 illustrates the current, early-stage prototype. Apart from the play/pause controls, it is largely indistinguishable from a regular spreadsheet. Indeed, the prototype can be used as a plain spreadsheet without any musical capability at all. It is implemented as a web page, and each cell can contain arbitrary Javascript code, consequently all native Javascript functions are available, and further libraries can be loaded in from the console.

However, this spreadsheet is also imbued with music and sound-generation capabilities, by virtue of a simple library of note and sequence synthesis functions built on top of the web audio API. The design of this library is one of the central, emergent, and open design questions raised by this exercise. Currently, functions are available which play a specified note (e.g., `p('c 5')`), or a sequence of notes (e.g., `s(['c 5', 'g 4', 'a 4', 'e 4'], 1)`), or which loop a sequence of notes defined by a range on the spreadsheet (e.g., `pLoop('viola','B1','B8')`). As with regular spreadsheets, arguments can be passed directly or can be references to other cell values. Further library functions provide easy access to arrays containing the notes of chords and scales. A global `tick` variable increments

once every tick; the time elapsed between ticks can be set by editing the global `tempo` variable. At each tick, the spreadsheet is re-evaluated. The `tick` variable is accessible in the spreadsheet, so the formula `if(tick%2==0) p('snare') else p('kick')` would produce a simple drum beat consisting of alternating kick and snare sounds. Cells are also re-evaluated upon edits, as in a regular spreadsheet. This implementation of time borrows directly from spreadsheet stream processing convention, and will be discussed in greater detail.

The precise specification of music/sound-generating library is still emergent. As each cell can be used as an environment for arbitrary Javascript code, it may be argued that the library is irrelevant as the user can embed any desired functionality directly into the spreadsheet, or inject it into the runtime environment through the browser console. However, rich standard library support (or the lack thereof) is as much a part of the experience of programming in a certain language as the language syntax itself. This fact is not lost on the programming community; standard library support is what allows, for instance, the programming language Python to boast of having "batteries included". Consequently, future versions of SheetMusic will still retain the property of being arbitrarily-scriptable, but it should be possible to create sophisticated musical programs without extensive custom Javascript code, showcasing clearly the advantages of the spreadsheet paradigm.

This direct use of spreadsheets provides two major advantages for music coding, beyond the mere fact that implementation can be *live* (Tanimoto, 1990), allowing for direct feedback when the program is edited. The first is that the spreadsheet paradigm is well-known to be an excellent interface for novice end-user programming. The second is that the grid formalism allows for rich secondary notation to be expressed through the layout of the spreadsheet. This is already common in business applications, where a single sheet may contain several separate 'regions' of cells with strong semantic connotations, separated by blank cells or highlighted in different colours, even though this separation is unnecessary for the correct functioning of the spreadsheet.

2. Related work

Manhattan (Nash, 2014) is a grid-based music sequencer which shares many properties of spreadsheets. In particular, the interface is laid out in rows and columns of cells, where cells can contain a number of object types, including formulae. Where it deviates from spreadsheets is that some of the layout is used to enforce a notion of time; each column represents a parallel stream of execution, and going downwards through rows within a column indicates the flow of time; lower rows are 'after' higher rows in the sequence. This basic control flow can be enriched using standard programming constructs such as conditional branching and loops, through formulae. Unlike spreadsheets, each cell can contain multiple formulae. Other grid-based languages, such as Piet and Al-Jazari (McLean, Griffiths, Collins, & Wiggins, 2010) exist, but these are unlike spreadsheets, as the 'cells' of the grid cannot freely store data and code interchangeably, and control flow is intimately linked to the layout of the cells. Visual dataflow languages such as Texture (McLean & Wiggins, 2011) also have some commonalities with spreadsheets.

Sonic Pi (Aaron & Blackwell, 2013) is a live coding platform for music, implemented as an embedded Ruby DSL, with sounds generated by the SuperCollider synthesis server. As novice end-users form part of the core audience for Sonic Pi, it is coupled with an IDE meant to mitigate the complexity associated with programming. Sonic Pi is nonetheless a textual programming language, with the control flow of music corresponding directly to the control flow of program execution. Other textual languages include Impromptu (Sorensen & Gardner, 2010), Tidal (McLean, 2014), and ChucK (Wang, Cook, et al., 2003).

3. Programming language or musical instrument?

A central design question is whether SheetMusic is intended as a composition tool, a musical instrument, or a programming language. This is a false trichotomy (Blackwell & Collins, 2005), but is a simplified expression of the question: *what sorts of artefacts are users expected to produce with this tool?*

I would argue that variable output is the defining characteristic of a program. A program whose output is fixed is merely a clever way of compressing that output data. In live music programming, it is possible to represent a musical piece by explicitly coding individual notes as a sequence, but this is not the standard approach. Instead, programming constructs such as data structures, conditionals, loops, and subroutines are employed, which has a number of advantages: it makes the structure of the piece explicit, it compresses the notation required to represent the piece (reduces *verbosity*), and increases the flexibility of the program (reduces *premature commitment*). That being said, if the output of running this program is fixed and independent of input, it is essentially equivalent to a hard-coded sequence of notes. It is more interesting to consider cases where the musical output of a program is variable, and dependent on its input. A program whose output is largely unaffected by a change in its input is less 'programmy' than one whose output is affected more. A closely related concept is cyclomatic complexity (McCabe, 1976), which quantifies the number of linearly independent execution paths through a program.

Moreover, I would argue that the defining characteristic of a musical instrument is a *player with intent and agency to musically affect the output* of the instrument. Consequently, live music programming is an instance of playing a musical instrument, as even though the intermediate source code instances may be 'fixed' programs with invariable output, the player has the agency to change the program. This distinguishes musical instruments from mere playback mechanisms; we would not consider someone pressing the 'play' button on a record player to be playing a musical instrument, but someone hitting 'play' and 'pause' rapidly in order to create a slicing effect has made a musical instrument out of the record player through *intent and agency* to musically affect the output.

SheetMusic could be used to store a sequence of notes (used as a composition environment), or play a sequence of notes with agency (used as a musical instrument). However, since it also has the potential to be used for designing music as an interactive, reactive, data-dependent experience (used as a programming language), it is this latter use case that the design will be focused towards.

4. Applications

Two immediate application scenarios present themselves. The first application scenario is custom sonifiers for data; a few SheetMusic formulae could instantly create 'auditory scatterplots' or line graphs, where data values are mapped to pitches and played in rapid succession – known to be highly perceptually effective for several types of analysis (Flowers, Buhman, & Turnage, 2005). Pitch mappings could be tailored specifically to the data domain (e.g., a change in octave or key at some domain-specific critical threshold). This would enable simpler multi-modal data analysis, as well as improve accessibility to data in spreadsheets for the visually impaired. Another application is as a prototyping tool for music in interactive games such as role playing games. Music and sound are often linked in complex ways to game state, from trivial applications such as sound effects for player actions, up to more complex applications such as selecting soundtracks with different moods depending on the player's location or status in the game world. SheetMusic could be used to compose, test, and share such state-dependent musical experiences, as a subset of game variables could easily be captured as cell values which are then readily available for musical interpretation through the library functions.

5. Representing time

In SheetMusic, time passes independently of the spreadsheet, communicating its current value to all the formulae in the spreadsheet once per 'tick'. Notes and effects relying on sub-tick durations can currently be expressed through stretching/squeezing and offset parameters. However, since this is an inconvenient notation, it is expected that just as in setting the time signature and tempo of a musical score, the tempo will be adjusted so that the duration of a tick corresponds to the smallest duration required to concisely capture the majority of the piece. For instance, a musical piece set to a tempo of 40, but consisting mainly of quavers, might be recast more 'comfortably' as a piece at tempo 80 consisting mainly of crotchets – this is a matter of taste, convention, and interpretation, which does not have an effect on the music denoted literally by the score.

The idea that time elapses independently of the spreadsheet layout, and that formulae recalculate either on a publish/subscribe or polling basis is the main convention for stream processing in spreadsheets, as implemented in Microsoft Excel's native Real Time Data (RTD) feature[1], as well as in several popular stream processing add-ins.

This can be contrasted against Nash's Manhattan, which sacrifices much layout flexibility by committing columns to denote *tracks* which execute in parallel, and rows to represent *time slices* which execute sequentially going downwards. This approach was taken to build directly on the chronological grid-based design of music sequencers, as the aim was to introduce greater spreadsheet programming capabilities to users of such sequencers. SheetMusic takes the opposite approach as it has the opposite aim; to bring greater musical programming capabilities to users of spreadsheets. By decoupling time, SheetMusic frees the layout of the spreadsheet for use as arbitary secondary notation. Griffiths' Al-Jazari decouples time from the grid differently – in that system, agents are programmed to explore the grid and 'play' cells they occupy. McLean's Texture is a 2D playground which employs proximity to infer control flow – syntactic elements placed close together are automatically connected.

6. Conclusion

This paper has presented an exploration of spreadsheets as end-user music programming tools, illustrated through an early-stage prototype called SheetMusic. The primary advantages of the spreadsheet paradigm in this context are direct manipulation, liveness, and the ability to use the layout of the spreadsheet as secondary notation. SheetMusic can be regarded as a musical instrument, but has potential as a tool for creating highly data-dependent musical 'programs' with emergent runtime behaviour.

7. Acknowledgements

Thanks to Alan Blackwell and Luke Church for discussions on the topic. The author is supported by an EPSRC industrial CASE studentship sponsored by BT Research and Technology, and also by a Robert Sansom scholarship from the University of Cambridge Computer Laboratory.

8. References

Aaron, S., & Blackwell, A. F. (2013). From sonic pi to overtone: creative musical experiences with domain-specific and functional languages. In *Proceedings of the first ACM SIGPLAN workshop on Functional art, music, modeling & design* (pp. 35–46).

Blackwell, A., & Collins, N. (2005). The programming language as a musical instrument. *Proceedings of PPIG '05 (Psychology of Programming Interest Group)*, *3*, 284–289.

Flowers, J. H., Buhman, D. C., & Turnage, K. D. (2005). Data sonification from the desktop: Should sound be part of standard data analysis software? *ACM Transactions on Applied Perception (TAP)*, *2*(4), 467–472.

McCabe, T. J. (1976). A complexity measure. *Software Engineering, IEEE Trans. on*(4), 308–320.

McLean, A. (2014). Making programming languages to dance to: live coding with tidal. In *Proc. 2nd ACM SIGPLAN workshop on Functional art, music, modeling & design* (pp. 63–70).

McLean, A., Griffiths, D., Collins, N., & Wiggins, G. (2010). Visualisation of live code. *Proceedings of Electronic Visualisation and the Arts 2010*, 26–30.

McLean, A., & Wiggins, G. (2011). *Texture: Visual notation for live coding of pattern*. Ann Arbor, MI: Michigan Publishing, University of Michigan Library.

Nash, C. (2014). Manhattan: End-user programming for music.

Sorensen, A., & Gardner, H. (2010). Programming with time: cyber-physical programming with impromptu. *ACM Sigplan Notices*, *45*(10), 822–834.

Tanimoto, S. L. (1990). Viva: A visual language for image processing. *Journal of Visual Languages & Computing*, *1*(2), 127–139.

Wang, G., Cook, P. R., et al. (2003). Chuck: A concurrent, on-the-fly audio programming language. In *Proceedings of International Computer Music Conference* (pp. 219–226).

[1] `https://support.microsoft.com/en-us/kb/285339`