IMUSIC: Programming Non-Music Applications with Music

Alexis Kirke[1] and Eduardo Miranda[1]

[1] Interdisciplinary Centre for Computer Music Research, School of Humanities, Music and Performing Arts, Faculty of Arts, University of Plymouth,
Drake Circus, Plymouth, UK
{Alexis.Kirke, Eduardo.Miranda}@Plymouth.ac.uk

Abstract. This paper discusses the implementation of a tone-based programming/scripting language called MUSIC (the name is an acronym for Music-Utilizing Script Input Code). In a MUSIC program, input and output consists entirely of musical tones or rhythms. MUSIC is not focused on music-programming per se, but on general programming. It can be used for teaching the basics of script-based programming, computer-aided composition, and provided programming access to those with limitations in sight or physical accessibility. During implementation MUSIC evolved into IMUSIC (Interactive MUSIC) which has its own Audio User Interface (AUI) in which the program structure is continuously represented by layers algorithmically generated tunes. This leads to the programmer almost 'jamming' with the computer to write the code. Although in itself fairly simplistic, IMUSIC could introduce a new paradigm for increasing the subjective experience of productivity, increasing flow in programming in general. In this paper the system is demonstrated using live Bongo drums: with two pieces of example code, whose live entry and execution are provided online as videos.

Keywords: Programming, human-computer interaction, computer music, tone-based programming, performance audio-user interface

1 Introduction

In this paper the implementation a new scripting language is described and demonstrated, called IMUSIC, standing for Interactive Music-Utilizing Script Input Code. IMUSIC is a language whose elements and keywords consist of groups of tones or sounds, separated by silences. The tones can be entered into a computer by whistling, humming, or "LA-LA"ing. Non-pitched sounds can be generated by clicking, tutting or tapping the table top for example. The keywords in MUSIC are made up of a series of non-verbal sounds, potentially with a particular pitch direction order. IMUSIC utilizes a simple script programming paradigm. It does not utilize the MAX/MSP-type or visual programming approach often used by computer musicians,

as its tone-based input is designed to help such programmers access and learn script-based approaches to programming.

IMUSIC evolved from MUSIC. The original proposal for MUSIC can be found here [1]. During its implementation Interactive MUSIC involved in which the program structure is continuously represented by algorithmically generated tunes. When the user triggers the programming interface the system plays an underlay – drums or an arpeggio. Then if the user enters a Repeat command the system adds another riff indicating it is waiting for the user to enter the number of repeats. Once this number is entered the system will change the overlay to a riff with the number of elements equal the number of repeats chosen. This riff will continue until an End Repeat command is entered. There are similar riffs for different command structures, and the riffs are transposed based on the number of indents in the code structure.

The purpose of IMUSIC is to fivefold: provide a new way for teaching script programming for children, to provide a familiar paradigm for teaching script programming for composition to non-technically literate musicians wishing to learn about computers, to provide a tool which can be used by blind adults or children to program, to generate a proof of concept for a hands-free programming language utilizing the parallels between musical and programming structure, and to demonstrate the idea of increasing flow thorugh real-time rhythmic interaction with a computer language environment. The language is also amenable to usage on mobile phones where users have their headphones in and uses the microphone or clicker to program with, while the phone is in their pocket. Furthermore it will be seen that these environments also provide a way of highlighting bugs through tempo and key-mode transformations.

2 Related Work

There have been musical languages constructed before for use in general (i.e. non-programming) communication – for example Solresol [2]. There are also a number of whistled languages in use including Silbo in the Canary Islands. There are additionally whistle languages in the Pyrenees in France, and in Oacaca in Mexico [3, 4]. A rich history exists of computer languages designed for teaching children the basics of programming exists. LOGO [5] was an early example, which provided a simple way for children to visualize their program outputs through patterns drawn on screen or by a “turtle” robot with a pen drawing on paper. Some teachers have found it advantageous to use music functions in LOGO rather than graphical functions [6].

A language for writing music and teaching inspired by LOGO actually exists called LogoRhythms [7]. However the language is input as text. Although tools such as MAX/MSP already provide non-programmers with the ability to build musical algorithms, their graphical approach lacks certain features that an imperative text-based language such as Java or Matlab provide.

As well as providing accessibility across age and skill levels, sound has been used in the past to give accessibility to those with visual impairment. Emacspeak [8] for example makes use of different voices/pitches to indicate different parts of syntax (keywords, comments, identifiers, etc). There are more advanced systems which

sonify the Development Environment for blind users [9] and those which use music to highlight errors in code for blind and sighted users [10].

IMUSIC is also proposed as a possible new paradigm for programming. “Flow” in computer programming has long been known to be a key increaser of productivity [11]. Also many developers listen to music while programming. This has been shown to increase productivity [12]. It is also commonly known that music encourages and eases motion when it is synchronized to its rhythms. The development environment incorporating a real-time generative soundtrack based on programming code detected from the user, and could support the user in coding rhythm and programming Flow.

In relation to this there has also been a programming language proposed called MIMED (Musically-backed Input Movements for Expressive Development) [13]. In MIMED the programmer uses gestures to code, and as the program code is entered the computer performs music in real-time to highlight the structure of the programme. MIMED data is video data, and the language can be used as a tool for teaching programming for children, and as a form of programmable video editing. It can be considered as another prototype (though only in proposal form) for a more immersive form of programming that utilizes body rhythm and flow to create a different subjective experience of coding.

3 IMUSIC Input

An IMUSIC code input string is a series of sounds. It can be live audio through a microphone, or an audio file or MIDI file / stream. If it is an audio input then simple event and pitch detection algorithms [14] are used to detect the commands. The command sounds are made up of two types of sound: dots and dashes. A dot is any sound event less than 250mS before the next sound. A dash is a sound event between 250mS and 4 seconds before the next sound event. It is best that the user avoid timings between 225mS and 275mS so as to allow for any inaccuracies in the sound event detection system.

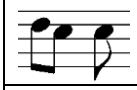
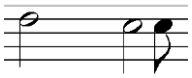
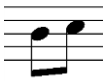
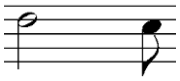
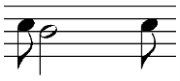
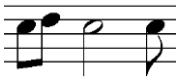
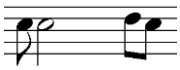
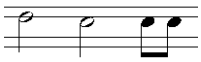
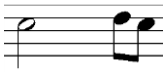
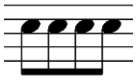
A set of input sounds will be a “grouping” if the gaps between the sounds are less than 4 seconds and it is surrounded by silences of 4.5 seconds or more. Note that these time lengths can be changed by changing the Input Tempo of MUSIC. A higher input tempo setting will reduce the lengths described above.

4 IMUSIC Overview

Table 1 shows some basic commands in IMUSIC. Each command is a note grouping made up of dots and/or dashes, hence it is surrounded by a rest. The second column gives what is called the Symbol notation. In Symbol notation a dot is written as a period “.” and a dash as a hyphen “-“. Figure 1 shows IMUSIC responding to audio input (a piano keyboard triggering a synthesizer in this case) in what is known as “verbose mode”. IMUSIC is designed from the ground up to be an audio only system. However for debugging and design purposes having a verbose text mode is useful.

Usually the user will not be concerned with text but only with the Audio User Interface (AUI).

Table 1. Music Commands Implemented

Command	**Dot-dash**	**Name**
	...	Remember
	--.	Forget
	.	Play
	..	Repeat
	-.	End
	.-.	Add
	..-.	Multiply
	.-..	Subtract
	--..	If Equal
	-..	Count
	…..	Compile

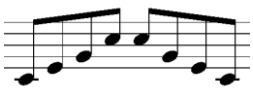
In one sense the pitches in Table 1 are arbitrary as it is the rhythm that drives the input. However the use of such pitches does provide a form of pitch structure to the program that can be useful to the user if they play the code back to themselves, if not to the compiler.

The IMUSIC AUI is based around the key of C major. The AUI has an option (switched off by default) which causes the riff transposition point to do a random walk of size one semitone, with a 33% chance of moving up one semitone and 33% of moving down. However its notes are always transposed into the key of C major or C minor. This can be used to provide some extra musical interest in the programming. It was switched off for examples in this paper.

When the IMUSIC Audio User Interface (AUI) is activated, it plays the looped arpeggio – called the AUI Arpeggio - shown in Figure 2, the AUI riff. All other riffs are overlaid on the AUI Arpeggio.

```
IMUSIC started, please enter code...

..
[IMUSIC: <repeat>]

[IMUSIC is listening...]
...
[IMUSIC: <repeat> data]

[IMUSIC is listening...]
.
[IMUSIC: <play>]

[IMUSIC is listening...]
.--...

[IMUSIC is listening...]
-.
[IMUSIC: <end>]

[IMUSIC is listening...]
```

Figure 1: Example IMUSIC input response – verbose mode



...

Figure 2. The AUI riff

In this initial version of IMUSIC, if an invalid command is entered, it is simply ignored (though in a future version it is planned to have an error feedback system). If a valid command is entered the AUI responds in one of three ways:

- Waiting Riff
- Structural Riff
- Feedback Riff

A Waiting Riff comes from commands which require parameters to be input. So once MUSIC detects a note grouping relating a command that takes a parameter

(Remember or Repeat), it plays the relevant riff for that command until the user has entered a second note group indicating the parameter.

A Structural Riff comes from commands that would normally create indents in code. The If Equal and Repeat commands effect all following commands until an End command. These commands can also be nested. Such commands are usually represented in a Graphical User Interface by indents. In the IMUSIC AUI, an indent is represented by transposing all following Riffs up a certain interval, with a further transposition for each indent. This will be explained more below

All other commands lead to Feedback Riffs. These riffs are simply a representation of the command which has been input. The representation is played back repeatedly until the next note grouping is detected. Feedback Riffs involve the representation first playing the octave of middle C, and then in the octave below. This allows the user to more clearly hear the command they've just entered.

For user data storage IMUSIC uses a stack [15]. The user can push melodies onto – and delete melodies from - the stack, and perform operations on melodies on the stack, as well as play the top stack element. Note that most current testing of IMUSIC has focused on entering by rhythms only, as the available real-time pitch detection algorithms have proven unreliable [16]. (This does not exclude the use of direct pure tones or MIDI instrument input, however that is not addressed in this paper.) So when entering data in rhythm only mode, IMUSIC generates pitches for the rhythms using an alleatoric algorithm [17] based on a random walk with jumps, starting at middle C, before storing them in the stack.

5 IMUSIC Commands

The rest of IMUSIC will now be explained by going through each of the commands.

5.1 Remember

After a Remember command is entered, IMUSIC plays the Remember waiting-riff (Figure 3 – note all common music notation is in the treble clef). The user can then enter a note grouping which will be pushed onto the stack at execution time.

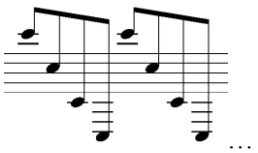


Figure 3: The looped Remember-riff

Once the user has entered the data to be pushed on to the stack, IMUSIC loops a feedback riff based on the rhythms of the tune to be memorized, until the next note grouping is detected.

5.2 Forget

Forget deletes the top item in the stack, i.e. the last thing remembered. After this command is entered, IMUSIC loops a feedback riff based on the Forget command tune's rhythms in Table 1, until the next note grouping is detected.

5.3 Play

This is the main output command, similar to "cout <<" or "Print" in other languages. It plays the top item on the stack once using a sine oscillator with a loudness envelope reminiscent of piano. After this command is entered, IMUSIC loops a feedback riff based on the Play command tune's rhythms from Table 1, until the next note grouping is detected.

5.4 Repeat

This allows the user to repeat all following code (up to an "End" command) a fixed number of times. After the Repeat instruction is entered, IMUSIC plays the Repeat Waiting Riff in Figure 4. The user then enters a note grouping whose note count defines the number of repeats. So for example the entry in Figure 5 would cause all commands following it, and up to an "end" command, to be repeated three times during execution, because the second note grouping has three notes in it.

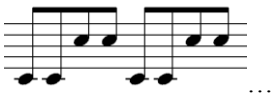


Figure 4: The looped Repeat-riff

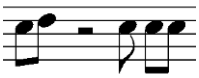


Figure 5: Input for Repeat 3 times

Once the note grouping containing the repeat count has been entered, the Repeat Waiting Riff will stop playing, and be replaced by a loop of the Repeat Structure Riff. This riff contains a number of notes equal to the repeat count, all played on middle C. The Repeat Structure Riff will play in a loop until the user enters the matching "End" command. Figure 6 shows the example for Repeat 3.

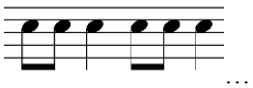


Figure 6: Repeat Structure Riff for "Repeat 3"

5.5 End

The End command is used to indicate the end of a group of commands to Repeat, and also the end of a group of commands for an "If equal" command (described later). After "end" is entered the Repeat Structure Riff will stop playing (as will the "If equal" riff – described later). End is the only command that does not have a Riff, it merely stops a Riff.

5.6 Add

The Add command concatenates the top of the stack on to the end of the next stack item down. It places the results on the top of the stack. So if the top of the stack is music phrase Y in Figure 7, and the next item down is music phrase X in Figure 6, then after and Add command the top of the stack will contain the bottom combined phrase.

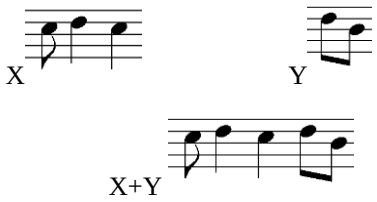


Figure 7: Results of the Add command

After this command is entered, IMUSIC loops a feedback riff based on the Add command tune's rhythms from Table 1, until the next note grouping is detected.

5.7 Multiply

The Multiply command generates a tune using the top two tunes on the stack, and stores the result at the top of the stack. If the top tunes is called tune Y and the next down is called tune X, then their multiplication works as follows. Suppose $X = [X^p_i, X^t_i]$ and $Y = [Y^p_j, Y^t_j]$. The resulting tune XY has a number of notes which is the product of the number of notes in X and the number of notes in Y. It can be thought of as tune X operating on tune Y, or as imposing the pitch structure of X onto the pitch structure of Y. The new tune XY is defined as:

$$XY^p_k = X^p_i + (Y^p_j - 60)$$
$$XY^t_k = X^t_i$$

For example suppose Y has 3 notes, and X has 2 then:

$$XY^p = [X^p_1 + (Y^p_1 - 60), X^p_1 + (Y^p_2 - 60), X^p_1 + (Y^p_3 - 60),$$
$$X^p_2 + (Y^p_1 - 60), X^p_2 + (Y^p_2 - 60), X^p_2 + (Y^p_3 - 60)]$$

and

$$XY^t_k = [X^t_1, X^t_1, X^t_2, X^t_2, X^t_3, X^t_3]$$

Figure 8 shows an example.

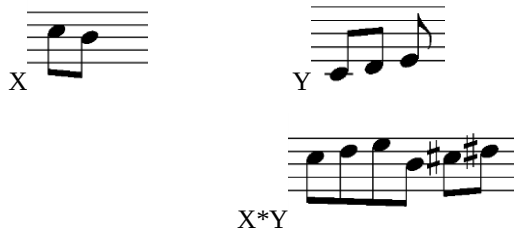


Figure 8: The result of a Multiply command when the top of the stack is Y and the next item down is X.

From a musical perspective, Multiply can also be used for transposition.

After this command is entered, IMUSIC loops a feedback riff based on the Multiply command tune's rhythms from Table 1, until the next note grouping is detected.

5.8 Subtract

Subtract acts on the top item in the stack (melody X) and the next item down (melody Y). Suppose melody Y has N notes in it, and melody X has M notes. Then the resulting tune will be the first M-N notes of melody X. The actual content of melody Y is unimportant – it is just its length. There is no such thing as an empty melody in IMUSIC (since an empty melody cannot be represented sonically). So if tune Y does not have less notes than tune X, then tune X is simply reduced to a single note. Figure 9 shows an example.

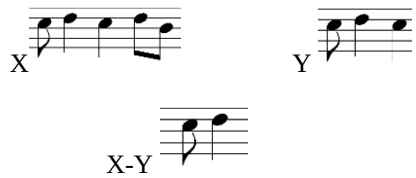


Figure 9: Results of the Subtract command

After this command is entered, IMUSIC loops a feedback riff based on the Subtract command tune's rhythms from Table 1, until the next note grouping is detected.

5.9 If Equal

This allows the user to ignore all following code (up to an "End" command) unless the top two melodies in the start have the same number of notes. After the If Equal instruction is entered, IMUSIC plays the If Equal Structure Riff in Figure 10.

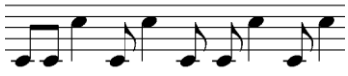


Figure 10: If Equal Structure Riff

This riff continues to play until the user enters a matching End command.

5.10 Count

The Count command counts the number of notes of the tune on the top of the stack. It then uses a text to speech system to say that number. After this command is entered, IMUSIC loops a feedback riff based on the Count command tune's rhythms from Table 1, until the next note grouping is detected.

5.11 Compile

The Compile command compiles the IMUSIC code entered so far into an executable stand-alone Python file. It also deletes all IMUSIC code entered so far, allowing the user to continue coding.

6 Examples

Two examples will now be given – one to explain the AUI behavior in more detail, and one to demonstrate calculations in IMUSIC. Videos and audio of both pieces of code being programmed and executed are provided [18].

6.1 Example 1: Structure

To explain structure representation in the AUI consider the first example in Figure 11 which shows an actual IMUSIC program which was entered using Bongo drums.

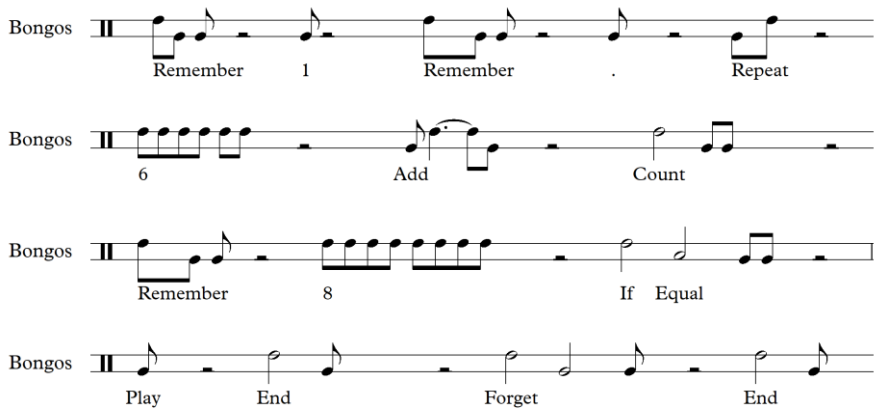


Figure 11: Example code to demonstrate Structure Riffs in AUI

This program might be written in text as (with line numbers included for easy reference):

```
1       Remember 1
2       Remember 1
3       Repeat 6
4               Add
5               Count
6               Remember 8
7               If Equal
8                       Play
9               End
10              Forget
11      End
```

The output of this program is the synthesized voice saying “2”, “3”, “5”, “8”, “13” then “21”. However between “8” and “13” it also plays a tune of 8 notes long. This is caused by the cumulative addition of the two tunes at top of the stack in line 4 of the code, and by the comparing the stack top length to a tune of length 8 in line 7 of the code.

The code entry will be examined in more detail to highlight more of the IMUSIC AUI. As code entry begins the AUI will play the AUI riff in Figure 1. The Riffs for the first 2 lines have been explained already. The third line’s Repeat will cause the Repeat Waiting Riff from Figure 4. After the Repeat Count is entered in line 3, Figure 12 will be looped by the AUI to indicate that all statements now being entered will be repeated 5 times.

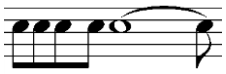


Figure 12: Repeat Count Structure Riff

Because it is a Structure Riff, it continues looping until its matching End is entered. When the user enters the next command of Add, the AUI will add Figure 13 as a Feedback Riff. It can be seen it has the same rhythms as the Add command from Table 1.

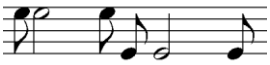


Figure 13: Add command Feedback Riff at 1 Indent

It can also be seen that this is a major third above the Repeat Count Structure Riff in Figure 12. This interval is an indication by the AUI of the single level of indent created by the earlier Repeat command. This transposition will also be applied to the Feedback Riffs of the next 3 commands (lines 5, 6, and 7). Note that each Feedback Riff stops when the next note grouping is entered. Immediately after line 7, the AUI will consist of the AUI Arpeggio, the Repeat Count Riff from Figure 12, and the Structure Riff in Figure 14. Notice how once again it is transposed a major third up compared to Figure 10.

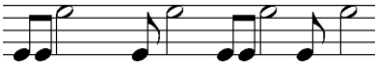


Figure 14: If Equal command Structure Riff at 1 Indent

This command embeds another "indent" into the AUI. So when the next command is entered – line 8, Play – its Feedback Riff will be a major fifth above the default position. This indicates a second level of indenting to the user, as shown in Figure 15.

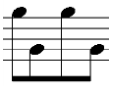


Figure 15: Play command Feedback Riff at 2 Indents

Then after the End command at line 9 in entered, the Play Riff and the If Equal Structure Riff stop. Also the indentation goes back by on level. So the Forget Feedback Riff will only be a major third above its default position. Then after the End command is entered from line 11, the Repeat Structure Riff stops, and only the AUI Arpeggio remains.

6.2 Example 2: Calculation

The second example – shown in Figure 16 - demonstrates some simple calculations, but – like all code in IMUSIC – can be viewed from a compositional point of view as well.

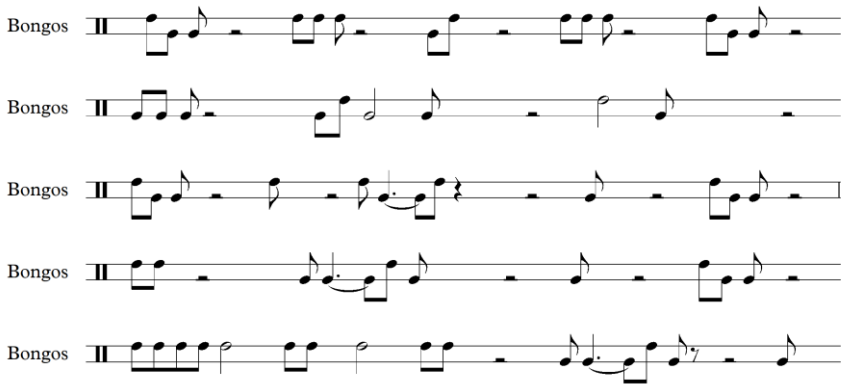


Figure 16: Example code to demonstrate calculation and "compositional" code

It was programmed using Bongo drums and Plays 3 note groupings of lengths: 82, 80 and 70 – ASCII codes for R, P and F. These are the intials of that famous scientific bongo player – Richard P. Feynman. In text terms this can be written as:

1 Remember 3
2 Repeat 3

```
3               Remember 3
4               Multiply
5       End
6       Remember 1
7       Add
8       Play
9       Remember 2
10      Subtract
11      Play
12      Remember 10
13      Subtract
14      Play
```

Note that multiple entries in this program involve counted note groupings (lines 1, 2, 3, 6, 9 and 12). The actual rhythmic content is optional. This means that as well as the output being potentially compositional, the input is as well (as note groupings are only used here for length counts). So when "pre-composing" the code, the note groupings in these lines where rhythmically constructed to make the code as enjoyable as possible to enter by the first author. This lead only to a change in the rhythms for line 12. Thus in the same way that many consider text-based programming to have an aesthetic element [19], this clearly can be the case with IMUSIC.

7 Other Features

A moment will be taken to mention some features of IMUSIC which are planned for implementation but not yet formally implemented. One is debugging. Once entered, a program of IMUSIC code – rather than being executed - can be played back to the user as music, as a form of program listing. One element of this playback will involve the pitched version of the commands. If the user did not enter the commands with pitched format, pitches can still be auto-inserted and played back in the listing in pitched format - potentially helping the user understand the structure more intuitively.

The IMUSIC debugger is able to take this one step further, utilizing affective transformations of the music. Once a program has been entered and the user plays it back, the MUSIC debugger will transform the code emotionally to highlight errors. For correct syntax the code will be played back in a "happy" way – default tempo and in C major. For code sections with syntax errors, it will be played in a "sad" way – half of default tempo and in a C minor. Such musical features are known to express happiness and sadness to listeners [20]. The Sadness both highlights the errors, and slows down the playback of the code which will make it easier for the user to understand. Not only does this make the affected area stand out, it also provides a familiar indicator for children and those new to programming: "sad" means mistake. (However this would require the pitches in Table 1 to be adjusted so that by default they include distinguishing notes for the relevant key modes.)

One other element to be mentioned is affective computation. Other work has demonstrated the use of music for affective computation, defining the AND, OR and NOT of melodies – based on their approximate affective content [21]. These are defined as MAND, MOR and MNOT ("em-not"). They are based on two dimensional

fuzzy logic, one dimension representing tempo, and one, key mode. For simplicity and space reasons the focus here will be tempo. The MNOT of a low tempo tune is a high tempo and vice-versa. The MAND of two tunes is the one with the lowest tempo. The MOR of two tunes is the one with the highest tempo. Three further commands are implemented in IMUSIC, each of which performs these operations on the top or top two melodies, in the stack. The result is pushed on to the top of the stack. The possible applications of this are beyond the scope of this paper, but are explained in [22] and include sonification and affective computing.

8 Conclusions and Future Work

This paper has discussed the implementation of a tone-based programming language called IMUSIC. In an IMUSIC program input and output consists entirely of musical tones and rhythms. IMUSIC is not focused on music-programming per se, but on general programming. It is designed to be used for teaching the basics of imperative text-based programming, computer-aided composition, and provided programming access to those with limitations in sight or physical accessibility. IMUSIC has its own Audio User Interface (AUI) in which the program structure is continuously represented by layers algorithmically generated tunes.

Although in itself fairly simplistic, IMUSIC could introduce a new paradigm for increasing the subjective experience of productivity increasing flow in programming in general. The language is also amenable to usage on mobile phones where users have their headphones in and uses the microphone or clicker to program with, while the phone is in their pocket. In this paper the system is demonstrated using live Bongo drum tests: with two pieces of example code, whose live entry and execution are provided online as videos.

The IMUSIC approach has not been properly evaluated from a user perspective. We are currently working with the Plymouth Hearing and Sight Centre. This will be helpful in obtaining feedback to further develop the language.

References

1. Kirke, A., Miranda, E.: Unconventional Computation and Teaching: Proposal for MUSIC, a Tone-Based Scripting Language for Accessibility, Computation and Education, International Journal of Unconventional Computation, Vol. 10, No. 3, pp. 237--249 (2014).
2. Gajewski, B.: Grammaire du Solresol. France (1902).
3. Busnel, R.G., Classe, A.: Whistled Languages. Springer-Verlag (1976)
4. Meyer, J.: Typology and intelligibility of whistled languages: approach in linguistics and bioacoustics. PhD Thesis. Lyon University, France (2005).
5. Harvey, B.: Computer Science Logo Style. MIT Press (1998).
6. Guzdial, M.: Teaching Programming with Music: An Approach to Teaching Young Students About Logo. Logo Foundation (1991).
7. Hechmer, A., Tindale, A., Tzanetakis, G.: LogoRhythms: Introductory Audio Programming for Computer Musicians in a Functional Language Paradigm, In Proceedings of 36th ASEE/IEEE Frontiers in Education Conference (2006).
8. Raman, T.: Emacspeak - A Speech Interface, In Proceedings of 1996 Computer Human Interaction Conference (1996).

9. Stefik, A. , Haywood, A., Mansoor, S., Dunda, B., Garcia, D.: SODBeans, In Proceedings of the 17th international Conference on Program Comprehension (2009).
10. Vickers, P., Alty, J.: Siren songs and swan songs debugging with music, Communications of the ACM, Vol. 46, No. 7, pp. 86--93 (2003).
11. Csikszentmihalyi, M.: Flow and the Psychology of Discovery and Invention. Harper Perennial (1997).
12. Lesiuk, T.: The effect of music listening on work performance, Psychology of Music, Vol. 33, No. 2, pp. 1730-191 (2005).
13. Kirke, A., Gentile, O., Visi, F., Miranda, E.: MIMED - Proposal For A Programming Language At The Meeting Point Between Choreography, Music And Software Development, In Proceedings of 9th Conference on Interdisciplinary Musicology (2014).
14. Lartillot, O., Toiviainen, P.: MIR in Matlab (II): A Toolbox for Musical Feature Extraction From Audio, In Proceedings of 2007 International Conference on Music Information Retrieval (2007).
15. Grant, M., Leibson, S.: Beyond the Valley of the Lost Processors: Problems, Fallacies, and Pitfalls in Processor Design. In Processor Design. Springer Netherlands, pp. 27-67 (2007).
16. Hsu, C-L, Wang, D. Jang, J-SR.: A trend estimation algorithm for singing pitch detection in musical recordings, In Proceedings of 2011 IEEE International Conference on Acoustics, Speech and Signal Processing (2011).
17. E. Miranda, Composing Music with Computers, Focal Press, 2001.
18. Kirke, A: http://cmr.soc.plymouth.ac.uk/alexiskirke/imusic.htm, last accessed 5 Feb 2015.
19. Cox. G.: Speaking Code: Coding as aesthetic and political expression, MIT Press (2013).
20. Livingstone, S., Muhlberger, R., Brown, A.: Controlling musical emotionality: An affective computational architecture for influencing musical emotions, Digital Creativity, Vol. 18, No. 1, pp. 43--53, Taylor and Francis (2007).
21. Kirke, A., Miranda, E.R.: Pulsed Melodic Affective Processing: Musical structures for increasing transparency in emotional computation, Simulation, Vol. 90, No. 5, pp. 606--622 (2014).
22. Kirke, A., Miranda, E.R.: Towards Harmonic Extensions of Pulsed Melodic Affective Processing - Further Musical Structures for Increasing Transparency in Emotional Computation, International Journal of Unconventional Computation, Vol. 10, No. 3, pp. 199--217 (2014).