Background Report: Loopy, A Domain Specific Language for Electronic Dance Music Creation

Edward Cai c4n0b@ugrad.cs.ubc.ca
Kwang Hee Park m8b9@ugrad.cs.ubc.ca
Shobhit Bhatia j9k0b@ugrad.cs.ubc.ca
Ro Lee n3l0b@ugrad.cs.ubc.ca
Benjamin Willox s2g1b@ugrad.cs.ubc.ca

ABSTRACT

Our proposal is a domain specific language, called Loopy, built for EDM (electronic dance music) creators. Loopy will be designed such that it provides solutions to the repetitive nature of EDM, such as loops with various notes. This report will go through an introduction on the background and the rationale of Loopy and explain the technical approaches in building it through going in detail of the concrete, abstract syntaxes and the interpretation.

1. OVERVIEW

Our project will create a domain specific language called Loopy. We will focus on the value of Loopy as an alternative music creation language primarily for EDM artists. We will illustrate the value by providing a brief overview of the EDM creation market including a look at competing languages. We will examine the features that our language has that will be of particular value to programmers.

We will create and annotate a concrete syntax that is easy to understand for the end programmer, an abstract syntax that will be interpreted into a song, and an interpreter that can provide the necessary framework for doing this. We will explore various design choices such as the forms that our concrete syntax can take and the various features of the RSound library. The repetitive and harmonic nature of EDM will influence the development of our language.

This report will examine the technical aspect of our language, provide sample programs that may be turned into music and illustrate why our DSL is worth creating.

2. VALUE PROPOSITION

2.1 Introduction

EDM (electronic dance music) has been around historically since the 1960s, but only ever reached popularity here in North America around the early 2010s after American music industries pushed to rebrand rave culture with EDM[1]. Artists such as Hardwell, Skillrex, and Steve Aoki hit millions of views as EDM rose to popularity and eventually triggered Billboard to introduce a new EDM-focused Dance / Electronics Song Chart in 2013 [2].

At a time where EDM is expanding its territories and artists are experimenting tools to create music more efficiently, our team wanted to come up with a tool that can help artists easily create electronic dance music through programming. We, specifically, wanted to target the repetitive nature of EDM which is comprised of numerous loops and notes that elevate and diminish in both volume and pitch.

2.2 The Target Market

Our target market is specific: anyone who produces EDM or applications thereof that finds text-based approach more efficient than a visual patch-based approach.

The history of text-based music creation goes back to 1957 when Max Matthews first created MUSIC[3], the first computer program to generate digital audio waveforms through direct synthesis. This spawned a whole family of computer programs and programming languages dedicated specifically towards creating text-based music creation.

The question still looms however: Are there any text-based music creation approach specifically geared towards fast, efficient EDM creation?

2.3 Alternatives

There are a couple of major DSP (short for digital signal processing) programming languages out there currently: Csound, Pyo and SuperCollider.

The problem with these alternatives are that they all offer too much. What we are aiming for is simplicity; a language with clear and efficient music creation. Below we will break down each of the competitors and discuss potential flaws with each.

Csound: Csound is a sound library for C. It is hailed as one of the most versatile yet complicated sound engines. It has functionality ranging from simple note creation to physical sound modelling. One thing that all Csound functions share is that they are very complicated. As illustrated in the figure below, simply setting up the framework to call any Csound functionality takes an extensive knowledge of C[4].

```
<Cabbage>
form size(460, 300), caption("Cart To Polar Vbap"), pluginID("Vbap")
hslider bounds(307, 0, 137, 123), channel("spread"), text("Spread"), range(0, 100, 0)
xypad bounds(4, 0, 280, 292), channel("X", "Y"), rangex(-1, 1, 1), rangey(-1, 1, 1)
</Cabbage>
<CsoundSynthesizer>
<CsOptions>
-d -n
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2  ;More than 2 chans crashes Cabbage, you can edit script with 8 chans use CsoundQT
0dbfs = 1

instr 1   ;vbap
kspread  chnget "spread"
kX       chnget "X"
kY       chnget "Y"

; Calculating Radius / Distance
kradius = sqrt((kX^2)+(kY^2))
aVol = aIn*kradius              ; Decrease Volume with more distance

;tan(xy)    for 0<θ<90° ... i.e. both x & y positive
if kX>0 && kY>0 then
kangle = taninv(kX/kY)

;θ=180°+arctan(xy)     for 90°<θ<270° ... i.e. y negative
elseif kY<0 then
kangle = 180+taninv(kX/kY)

;θ=360°+arctan(xy)     for  270°<θ<360° ... i.e. x negative, y positive
elseif kX<0 && kY>0 then
kangle = 360+taninv(kX/kY)

endif

printk 1, kangle  ; Test to see what angle is output

a1, a2, a3, a4, a5, a6, a7, a8 vbap aVol, kangle, 0, kspread
outo a1, a2, a3, a4, a5, a6, a7, a8
endin
</CsInstruments>
<CsScore>
i1 0 [60*60*24*7]
</CsScore>
</CsoundSynthesizer>
```

[Figure 1]

We view this required knowledge as not typical of the average music producer. Furthermore, Loopy is not in direct competition with Csound as they target completely different markets; the former targeting casual EDM artists while the latter is more targeted towards programmers or professional sound engineers.

Pyo (Python sound): Just like Csound, Pyo is a sound library for Python. With respect to our analysis, it shares many qualities with Csound; mainly being overly functional and requiring a great deal of programming experience. Pyo is also built around the functionality of connecting to external hardware devices. This is completely beyond the scope of our project and evidence that Pyo and Loopy are targeted towards distinct markets. Interestingly enough, Pyo was actually written in C. Below is an excerpt of Python code showing what is required to play a few binary sounds. It takes a great deal of experience in both Pyo and Python to be able to write loops that will append these sounds into a tangible song.

```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ========================== RESTART ====================
>>>
Binary Representation of sound
Enter the duration of each note (in ms)?
e.g. 200
>200
Enter a 4-bit binary note
Or more than one note separated by spaces
Notes:
0000 = no sound
0001 = Low C
0010 = D
0011 = E
0100 = F
0101 = G
0110 = A
0111 = B
1000 = High C
e.g:
0101 0101 0101 0010 0011 0011 0010 0000 0111 0111 0110 0110 0101
>
```

[Figure 2]

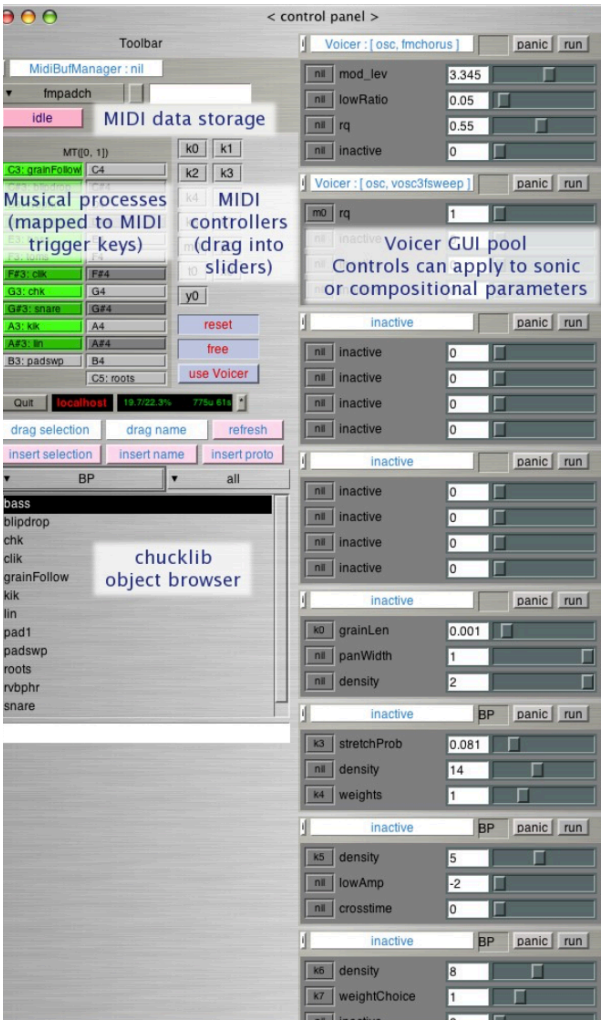
SuperCollider: SuperCollider is an environment and programming language originally released in 1996 by James McCartney[5]. It provides the functionality of real time music synthesis and audio compression.

SuperCollider offers studio level music creation. It has three main parts that interact: a background framework based on code, a control panel, and a mixing window.

As you can see from *Figure 3* and *Figure 4*, SuperCollider is complex. One must learn how to master each aspect before they can start to create actual music. A very basic understanding of

SuperCollider is noted as taking 12 weeks of practice[6].

[Figure 3]



[Figure 4]

Furthermore, proper music creation in SuperCollider requires hardware. It is recommended to use both physical sliders for the control panel and the mixer. Clearly, SuperCollider and Loopy are targeted towards different markets. SuperCollider is for professional music creation while Loopy is for clear and efficient EDM production with no background knowledge.

Echoing the analysis above, Loopy offers what these alternatives do not; music creation for artists with little or no coding experience.

3. THE LOOPY LANGUAGE

3.1 Concrete Syntax

The concrete syntax of a language provides the set of rules that define the way programs look to the programmers. When we were creating our concrete syntax our goal was twofold; firstly to provide an easily understandable syntax and secondly to efficiently represent the functionality of our language. We decided upon the following:

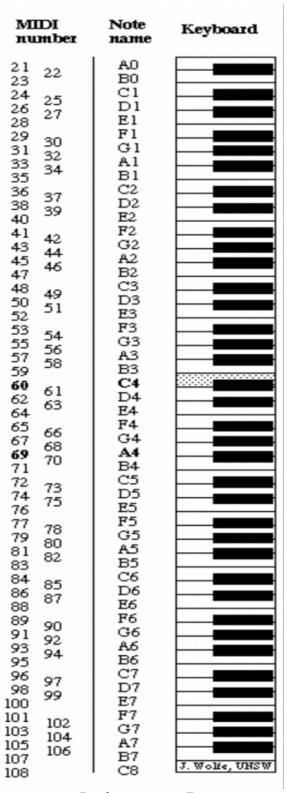
```
<L-Expr> ::=
| {note <num> <num> <num>}
| {loop <L-Expr>* <num> <num> <num>}
| {segment <L-Expr>* <num>}
| {<num> <L-Expr>}
```

The first form is the basis of all songs the user can create. It represents a synthetic musical note.

```
{note <num> <num> <num>}
```

The first parameter is the musical instrument digital interface or MIDI number of the note to be played. We decided to use MIDI note numbers rather than conventional musical notation since it is already widely used in the aforementioned alternative programming languages.

Figure 5 shows a conversion of a typical note and octave into the corresponding MIDI number[7]. The second and third parameters are the start and end bar in which the note will be playing for. A bar, in musical notation, is the segment of time



[Figure 5]

corresponding to a specific number of beats[8], and by default, all bars in Loopy will consist of 4 beats, or in musical notation, follow a time signature of $\frac{4}{4}$, meaning there will be 4 beats of quarter notes playing in one bar. All musical notation aside, to keep it simple to understand for the users we designed it so that all the user needs to provide is the start bar position and the end bar position for the synthetic musical note.

Our second form is the representative feature of our language which directly targets the repetitiveness of EDM: loops.

```
{loop <LExpr>* <num> <num> <num>}
```

The parameters that a loop takes are: 1) a list of `LExpr` values to be looped, and like the note, 2) a start bar and 3) an end bar, and finally 4) the number that signifies the amount of iterations that the loop will take. This will allow artists to repeat sounds an arbitrary amount of times.

The third form that our musical expression can take is a segment.

```
{segment <LExpr>* <num>}
```

A segment takes two parameters: a list of `LExpr` values to be played and a number representing the number of total bars of the segment. The main goal of the segment is to provide structure. Structure in the sense that users can see how a series of notes and loops creates certain melodies or sections of their musical piece. And by providing the total number of bars, users can ensure that the loops defined within cannot accidentally go over the length that they desired for a segment. This validation will take place in the interpretation stage where the segment will actually clip the sound to the length of its total bars.

So, here's an example:

```
{segment
      {loop
            {note 50 1 4}
            {note 45 1 2}
            {note 30 2 2}
      1 4 4}
16}
```

The example is a segment that runs for 16 bars with a loop. The loop itself starts at the first bar, ends at the fourth bar, and repeats itself 4 times. The first note will be playing a #50 MIDI note from the first bar until the fourth bar. The second note will be playing a #45 MIDI note from the first bar until the second bar. Finally, the last note will be playing a #30 MIDI note from the second bar until the second bar, so for just one bar.

The fourth and final form is the speed modifier. It takes a number and a musical expression and modifies the speed of that musical expression by whatever the number given is. For example the code shown below will play the expression `{loop {note 50 1 1} 1 8 3}` twice as fast.

```
{2 {loop {note 50 1 1} 1 8 3}}
```

By twice as fast, we actually mean doubling the BPM. For default, we will go with 128 BPM since that is the average EDM tempo[9], however by providing a speed modifier, artists can freely change the BPM to whichever speed they prefer. The design as to why give it a multiplier compared to actually providing the exact BPM number is that usually in EDM, the tension before the "drop", a moment of instrumental build when the bass and rhythm hit hardest[10], has a slowly increasing speed. Instead of stating actual BPMs, our team thought that giving multipliers, for example, 1.0, 1.1, 1.2, 1.3, etc. would be a more efficient and intuitive way to state speeds.

This aligns with our goal of creating a language that artists will find useful.

3.2 RSound

Before we go into the interpretation stage, let us briefly explain about the RSound, a crucial package for our implementation of Loopy.

RSound[11] is a Racket sound engine that is described by its creators as "an adequate sound engine". It includes procedures for sound control, stream based playback, recording, sound I/O, and network based playback. Our discussion will focus mainly on rsound manipulation and single-cycle sounds because they are core to our interpreter and thus our DSL.

Here are some descriptions of the core procedures that we will be using:

```
FRAME-RATE : nonnegative-integer?
```

This is the basic default frame rate. It is set to 44,100 Hz.

```
(play rsound) -> void?
  rsound : rsound?
```

The play procedure takes an rsound and plays it through the computer's audio output device. Our DSL will use play as the final value of our interpreter; the key step in turning all the rsound values that we generate into tangible music.

```
(resample factor sound) -> rsound
  factor : positive-real?
  sound : rsound?
```

The resample procedure will be used to change the speed. The factor parameter is used to change the length of the sound while retaining the frame rate, resulting in a sped up or slowed down sound.

```
(synth-note family
            spec
            midi-note-number
            duration) -> rsound
  family : string?
  spec : number-or-path?
  midi-note-number : natural?
  duration : natural?
```

The synth-note procedure is the basis of our note and provides an EDM like synthetic note that can be modified based on parameters. For default we will resort to the "main" family and use the first case of it defined by the spec parameter. The midi-note-number refers to the actual note that will be played and, of course, the duration in frames.

```
(assemble assembly-list) -> rsound?
  assembly-list :
  (listof (list/c
    rsound?
    nonnegative-integer?))
```

This is a crucial procedure which is used to assemble all of the recursively processed sounds with each sound starting at an offset.

```
(rs-append rsound-1
           rsound-2) -> rsound?
  rsound-1 : rsound?
  rsound-2 : rsound?
```

This procedure will be used in our loop implementation to concatenate sequentially the processed rsound objects.

```
(silence frames) -> rsound?
  frames : nonnegative-integer?
```

This procedure returns an empty silent sound for the given frame. It will be used to prepend a

buffer to a given rsound that does not start at bar 1 (which is always the starting bar).

```
(clip rsound start finish) -> rsound?
  rsound : rsound?
  start : nonnegative-integer?
  finish : nonnegative-integer?
```

The clip procedure is used, as its name states, to clip the rsound from the start to finish frame. This will be used by our segment case to enforce structure; not let loops or notes go on forever by mistake.

3.3 Interpretation

Our interpretation of notes, loops, segments and speed modifiers will heavily use the aforementioned RSound functions.

Let's consider the example below:

```
{note 40 2 3}
```

We know that this note will play the MIDI number 40 starting from the second bar to the third bar. But, what about the "empty" sound automatically implied for the first bar?

As explained in *Section 3.2*, we will be using the silence function to prepend a silent sound and thus to fill the first bar of this sound. This way we can ensure that the note runs for 3 bars in total, playing the sound we want from the second bar to the third bar.

This will be similar in practice for loops as well; for any "empty" sounds that the loop holds due to the difference in start and end bar positions, we will fill it up with silent sounds.

Another case we need to consider is the recursive nature of Loopy. For loops and segments to properly process the given lists of `L-Expr` objects and simultaneously play them, we will have to make sure that the `L-Expr` objects are recursively processed while assembling each individual sound into one to create one sound. The recursive nature of our interpretation function will handle this effectively.

4. PROJECT POTENTIAL

4.1 Initial Implementation

To achieve this milestone, we must first create a basic parser and interpreter. At this step we plan to lay out the outline for the language's design, breaking it down into specific tasks that we can fulfill.

To complete the proof-of-concept, we plan to build a basic parser and interpreter that implements the necessary components of our language: notes and loops. This parser will allow the user to build relatively simple EDM tracks but will more importantly demonstrate that our final language is possible and useful. We chose to only include notes and loops in this milestone we feel that they are sufficient to show the underlying parser and interpreter are working to interpret our DSL.

4.2 Final Project

To complete our final project we will be expanding our parser and interpreter to implement speed changing parameters and segments. This will take our DSL from being able to build simple beats to being able to build complex electronica that has structured components.

The main hurdle that we will have to overcome in finishing this project is the interpretation and scoping of speed changing parameters. We have done some research into possible implementation techniques and have decided that working with the conversion factor between frames and seconds is the best choice. Since a duration of one second is approximately 44,100 frames, we will work with an environment to keep a speed factor that we can use to modify the duration of notes in musical expressions that follow the parameter.

Our final project will also include a tutorial package that demonstrates how EDM artists can write music. It will include some recreation of

popular songs that we think will serve as good examples.

5. SUMMARY

Our report has outlined the unique and useful features of Loopy that make it suitable for EDM development. By describing the concrete syntax, abstract syntax, parsing and interpretation, this report has demonstrated the technical aspects that our DSL will be built upon. More importantly this report has shown the thought we have put into each feature, so that our language will have true value to EDM artists.

REFERENCES

[1] Reynolds, Simon. “How Rave Music Conquered America.” The Guardian, Guardian News and Media, 2 Aug. 2012, www.theguardian.com/music/2012/aug/02/how-rave-music-conquered-america.

[2] Pietroluongo, Silvio. “New Dance/Electronic Songs Chart Launches With Will.i.am & Britney at No. 1.” Billboard, Billboard, 21 Jan. 2013, www.billboard.com/articles/news/1510640/new-danceelectronic-songs-chart-launches-with-william-britney-at-no-1.

[3] DuBois, R. Luke. “The First Computer Musician.” The New York Times, The New York Times, 8 June 2011, opinionator.blogs.nytimes.com/2011/06/08/the-first-computer-musician/.

[4] The Canonical CSound Reference Manual, http://www.csounds.com/manual/html/while.html http://newt.phys.unsw.edu.au/jw/notes.html

[5] SuperCollider https://en.wikipedia.org/wiki/SuperCollider

[6] Learning SuperCollider http://supercollider.sourceforge.net/learning/

[7] What Is a Sound Spectrum?, newt.phys.unsw.edu.au/jw/notes.html.

[8] “Bar (Music).” Wikipedia, Wikimedia Foundation, 29 Apr. 2018, en.wikipedia.org/wiki/Bar_(music).

[9] “Why Is House Music 128 BPM?” EDM Nerd, 1 Mar. 2015, www.edmnerd.com/house-music-128-bpm/.

[10] Yenigun, Sam "The 5 Deadliest Drops Of 2010". www.npr.org. NPR, 31 December 2010

[11] RSound Documentation https://docs.racket-lang.org/rsound/index.html