Musical Programming

Live Textual Performance

Zhi Chen
Grinnell College '17
chenzhi17@grinnell.edu

Erin Gaschott
Grinnell College '17
gaschott17@grinnell.edu

1. Introduction

Poetry and music have a long history of association. In the Western world, the close relationship between the two has been classically embodied through Euterpe, the ancient Greek muse of music, song, and poetry. Throughout history, music and poetry have often been performed together. In the modern era, the emergence of live coding as a performance art, especially in the discipline of functional programming, presents a new and novel means of expressing the historic relationship between poetry and music with computer science. Live coding, as a means of producing improvised computer music, can be utilized to highlight and to convey the fundamental connection between poetry and music, rhythm. By bringing forth rhythm, or the sound of music and poetry, with computer science through live coding performance in Haskell, we aim to explore beyond the confines of their traditional domains and incorporate the natural sciences with the humanities in a celebration of a truly liberal arts endeavor.

2. Prior Work

There has been relatively little to no prior work on the expression of rhythm in poetry and music through live coding performance. Live coding performance has largely remained a separate and exclusive entity concerned mainly with live musical performance. The work that does exist intends to combine music, text, and static programming to convey the emotions of various pieces of literature.

Within the sphere of living coding, there is a diversity of products, services, and communities devoted to musical performance without the incorporation of poetry or text. Many of these communities are united by the (The (Temporary | Transnational | Terrestrial | Transdimensional) Organisation for the (Promotion | Proliferation | Permanence | Purity) of Live (Algorithm | Audio | Art | Artistic) Programming), otherwise known as TOPLAP. This group promotes live coding and provides resources and guidances to the general public on live musical coding and performance. Packages for live coding in Haskell are dominated by the exclusive theme of music. Common packages include Vivid, a package for music and sound synthesis using SuperCollider, a MIDI interface; and Midair, a package for livecoding that allows users to exchange parts or the whole of a Functional Reactive Programming graph while the graph is running. Domain-specific languages for live music performance include Euterpea, used more in an academic setting, and TidalCycles, a more popular language specifically designed for performance. Due to the availability and accessibility of reference materials and guides, our project uses TidalCycles.

The ideals of a live coding performance are outlined in the TOPLAP manifesto, and our project strived to embody the spirit of live coding in our work. Listening to a live or documented coded performance using our text to code generator is the only true way to experience our work, but we also attempt to create understanding through words alone in our summary. The most salient theme throughout the manifesto is that of transparency, the idea that the code and algorithms should be shown to the listener at the same time they are hearing it. In a live coding performance, this means a screen mirror of the performerâĂŹs code projected visibly to the audience as the main focus of the stage. With the performerâĂŹs work so visible and exposed, the art of skillful typing is also on display and is to be celebrated as part of the work.

One of the successful examples of incorporating music, text, and programming is TransProse, developed by Hannah Davis of New York University. TransProse finds different emotions throughout the text of a novel and programmatically creates a song that has a similar emotional tone. In this algorithm, eight different emotions (joy, sadness, anger, disgust, anticipation, surprise, trust, and fear) and two different states (positive or negative) are tracked chronologically in a piece of literature. The emotion density data is then used to determine the tempo, key, notes, octaves, and other musical components for a piece depending on rules and parameters. Unlike TransPose that highlights emotions in large textual works by statically generating music, our project intends to use live coding musical performance to highlight rhythm, the fundamental basis for both music and poetry.

3. Examples

The ideals of live coding shaped the design of how the user interacts with the program. We tried to keep the improvisation and intellectualism of live coding as strong as possible instead of hiding behind an algorithm that composes everything for you. With this in mind, we designed our program to accept a line of text and output a rhythm based on the length of the line and the number of syllables in a word, as detailed in the Implementation section. We showed in class the ability of the code to create a whole poem, but by outputting one line at a time, the artist is offered the freedom to express themselves however they choose, opening the door to more exploration and innovation in how text-based rhythms are incorporated into live performance. For example, a rhythm created from text could be used as the starting beats for the performer to expand upon, as an accent used one time only, or, as we demonstrated, in combination with multiple lines into a poem. Our program continuously handles the input of a line of text and converts it into code, which the artist can then copy into their work by copy-pasting or by retyping if they want to fully embrace the live coding values.

For example, let us look at this poem by Scott Hicks.

there is no jealousy
between
grains of sand

An artist could start out with a simple beat, like so:

```
d2 $ density 2 $ sound "bd hh
```

They then translate the first line into a rhythm using our program, returning `[[sf*1] [sf*1] [sf*1] [sf*3]]`.

The artist decides they want to play this line, then play a space in order to isolate the repetitions from each other. Adding an extra beat subdivided the cycle, so the original rhythm is doubled in speed, so the artist decides to slow the cycle down so that the rhythm stays at the same speed as it was on its own.

```
d1 $ slow 2 $ sound "[[[sf*1] [sf*1] [sf*1]
[sf*3]] [~]]"
```

From here, the artist decides to generate the rhythm for the rest of the poem and add it to what they have. They slow down the cycle even more to accommodate for the additional subdivisions created by adding new lines.

```
d1 $ slow 4 $ sound "[[[sf*1] [sf*1] [sf*1]
[sf*3]] [[blip*2]] [[cp*1] [cp*1] [cp*1]][~]]"
```

This complete poem rhythm could be the final product, or the performer could use it as a starting point to leverage the expansive TidalCycles capabilities.

4. Program Implementation

The architecture of our program can be divided into two thematic sections. The first section manages user-input and the second section manages the text to beat conversion and processing.

Within the first section, main initiates a feedback-response I/O loop to receive and process input from the user. After a user has entered a line of poetry into the program, the string is parsed into a list of text with

Regex, keeping only alphabetic characters and removing numbers, symbols, and invalid results. The Wreq library is then used to make a GET request to the Words API to ascertain the number of syllables in each of the words. An exception handler is wrapped around the GET request to ensure that exceptions thrown by the Words API for invalid words do not interrupt the program. Additionally, the number of free requests remaining is printed to remind the user that use of the Words API is limited to 2,500 requests per day. A list of Maybe Value is returned as a result of the GET request. The list of Maybe Values is converted into a list of integers, with numeric values within Justs extracted and Nothings converted to the number one.

With a list of Ints representing the number of syllables of each of the words from the user submitted line, the program enters the second phase to convert the syllables into a rhythm. A line is written as a single cycle with a consistent tone throughout the whole line. This note is chosen based on the number of words in the line as determined by the length of the input list. There are seven distinct sounds that are selected from using a modulo function. Using the `map` function, each word within the input list is run on a procedure that repeats the chosen tone for the line as many times as the word has syllables. These words are encapsulated within brackets, giving each word equal time within the cycle.

5. Reflection

As a result of the programming language, tools, and libraries that were employed in the project, there were both eases and difficulties. However, the eases outnumber the difficulties, but the difficulties nearly outweighed the conveniences.

One of the main conveniences that we experienced was the existence of pre-existing libraries and services, such as Wreq and the Words API. Without the Words API, we would have been able to get the syllables of each word. Without the Wreq library, we would not have been able to perform convenient HTTP requests to the Words API using Haskell. The Wreq library provides additional features that were especially useful. First, managing header information such as what type of response to accept or what API keys to use was a matter of setting options. Second, the Wreq library simplified the use of lens to focus in on portions of Haskell values. For example, when the program receives a return response after making the GET request to the Words API, the ^. Operator that Wreq provides takes a value as the first argument and a lens as the second and returns the portion of the value focused on by the lens. The operator allowed us to easily access various parts of the response body such as the response status, the status code, and the actual response content. The functional nature of Haskell allows for the use as higher order procedures, which allowed us to generate very clean and concise code, specifically in the section that converted a list of syllables into rhythm.

Working with a functional programming language presented difficulties with types. To accept user input, the program works with the I/O monad and had to constantly pack and unpack values outside of their contexts. Additionally, the Wreq library accepts text while our I/O inputs were strings and returns IO (Maybe Value) for GET requests that we then had to unpack to ascertain the value. The Words API does not accept words that are incorrectly spelled and so it throws an exception that had to be dealt with. Handling the error that the exception threw within the immediate function that contained it and handling the exception outside of the function within main presented two separate challenges because of the different types of error being thrown and handled. Fortunately, our instructor, Peter Michael-Osera, with more experience in functional programming was able to resolve our difficulties and struggles.

6. References

TransProse. Available at http://www.musicfromtext.com/.
TidalCycles. Available at http://tidalcycles.org/.
Words API. Available at https://www.wordsapi.com/.
Wreq package.
Available at https://hackage.haskell.org/package/wreq.
The TOPLAP Manifesto.
Available at http://toplap.org/wiki/ManifestoDraft.

Acknowledgments

We thank Peter Michael-Osera, Grinnell College, for his support and encouragement. We would also like to thank the students of the Computer Science Department at Grinnell College for providing us with examples of poetry.