College of Wooster

CSCI-200 Independent Study

Musical Cellular Automata

Colby Jeffries

April 24, 2017

Contents

1 Abstract

This paper aims to show that music can be generated through the states of cellular automaton. As background, this paper provides a survey of cellular automaton including history, applications and rule sets. Some basic music theory is presented as background for electronic music generation. Finally, the music generation software is presented.

2 Introduction

Cellular automaton are a form of automaton that determine their state based on their neighbors states. This simple concept, along with a few basic assumptions and rules, can be used to model many physical phenomena and generate many interesting patterns. This paper discusses cellular automaton, their history, rules and concepts. As background for the following software, some basic music theory is presented. This paper culminates in the presentation of software that generates music based on cellular automaton. The implementation, shortcomings and possible future additions to the software are discussed as well.

3 Cellular Automata

A simple definition of cellular automaton: "an entirely discrete version of a physical field" [12]. This definition is a bit ambiguous, so a more detailed description of the traits of all cellular automaton is more useful: A cellular automaton can be described as a discrete model that contains a finite number of states. The "world" in which these automaton reside is made up of a discrete grid of automaton that can have any finite set of dimensions. The state of each cell, is determined by the states of the cells in it's neighborhood, which can be defined in a number of ways, but usually includes the cells

directly around it [12]. Cells are updated in discrete time steps, usually called generations [5]. These basic traits are common to all cellular automaton, but are very general, leading to huge varieties in automaton shape, neighborhood shape and size, and grid dimensions.

3.1 History

Cellular automaton were originally conceived as a method to model the motion of liquid by Stanislaw Ulam and John von Neumann in the 1950s. The idea was to calculate the movement of one discrete piece of liquid using the behavior of its neighbors [7]. Research continued on cellular automaton, primarily as a means of modeling physics, until an explosion of interest, spiked by John Conway's "Game of Life", pushed the field to new heights [5]. Stephen Wolfram, founder of Wolfram Research and Mathematica, is also well known in the field for completely classifying cellular automaton using a few limiting assumptions [7]. His classifications allow the categorization of different automaton and easy statement of their rules, exemplified by our discussion of the 1-D rule set.

3.2 Conway's Game of Life

Conway's Game of Life is considered to be one of the most influential and important cellular automaton models created. Originally published in the October 1970 issue of Scientific American, Conway's model was originally described as a "solitaire pastime": a game [9]. Conway's game was much more than a game: it fascinated people with its changing patterns, cycles and gliders. This fascination led to further research and application of cellular automaton. Conway's game is relatively simple: the automaton are square and form a grid that can be infinite, toroidal or as in the original specification, finite (as it was meant to be played on a checker board). There are only three rules to determine automaton state:

1. Every alive cell with two or three alive neighbors survives to the next generation.

2. Every alive cell with less than two ore more than three alive neighbors dies.

3. Every empty cell with exactly three alive neighbors becomes alive in the next generation.

With these simple rules, the game can be played [9]. Figure 1 shows some

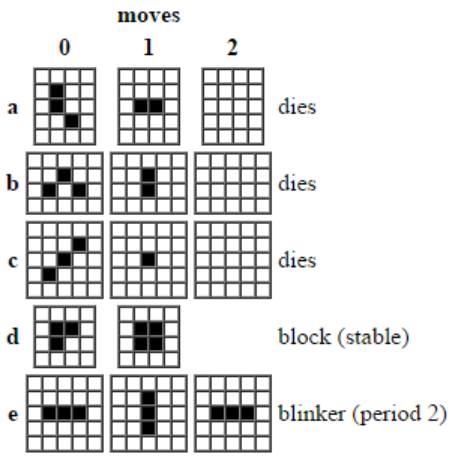

Figure 1: Basic examples of Conway's Game [9].

basic examples of Conway's game in action. What makes Conway's game so interesting, and often beautiful, is that cyclic patterns often form seemingly randomly (Figure 2). Sometimes cyclic patterns that translate across the grid called gliders (Figure 3) form as well. Conway's Game of Life is just one of many rule sets of two dimensional cellular automaton, which are explored further in the next section.

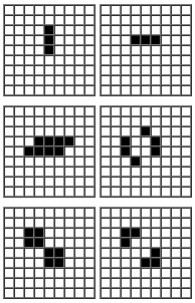

Figure 2: Basic cyclic patterns in Conway's Game.

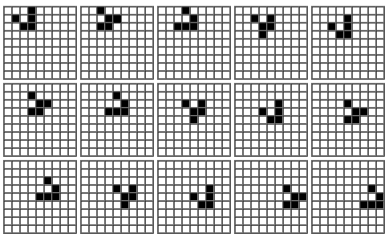

Figure 3: The smallest glider in Conway's Game.

3.3 Other Forms of Cellular Automata

In exploring other types of cellular automaton, its best to start with their one dimensional form, more commonly known as elementary cellular automaton. As Wolfram discovered, there are k^{k^s} rule sets given k states and s cells in the neighborhood [7]. Thus when looking at elementary cellular automaton,

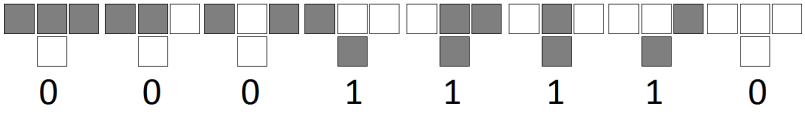

Figure 4: Rule 30 binary representation.

there are $2^{2^3} = 2^8 = 256$ possibilities. The reason the rules are called 0-255 is because that there are 8 possible neighborhoods, and whether or not each leads to cell survival/birth can be indicated in binary (Figure 4) leading to rule 0 (00000000 in binary) to rule 255 (11111111 in binary). These cellular

automaton usually generate basic, predictable patterns (Figure 5).

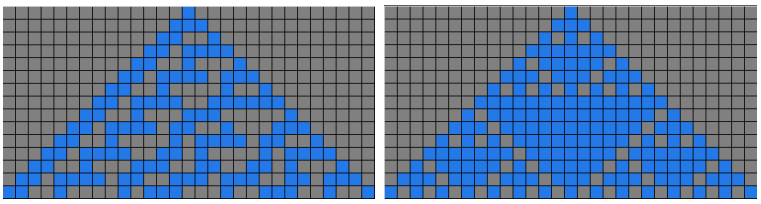

Figure 5: 15 Generations of rule 30 and rule 182.

From the Wolfram's equation, we know that moving to two dimensional cellular automaton yields many more rule sets. For example, if we consider a Moore neighborhood (the eight cells directly around a cell) and only two states, there are $2^{2^8} = 2^{256} \simeq 10^{77}$ possible rule sets. With this many possible rules, its better to just highlight a two of many interesting sets: Seeds and Langton's Ant.

Seeds was created by Brian Silverman (who also created Brian's Brain, another popular rule set) [13]. Seeds is extremely simple, and similar to Conway's Game of life, (so similar that it belongs to a class of cellular automaton called "life-like"). There are only two rules: any empty cell with exactly two alive neighbors becomes alive, and all cells that do not meet this requirement die. Despite these simple rules, there are very few gliders and

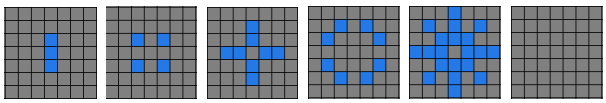

Figure 6: Seeds after a few generations.

glider guns found for this rule set. This rule set is both prone to chaos and stable configurations: an interesting and unusual combination [13].

Langton's ant was originally introduced by (and named after) Chris Langton in 1986 [11]. The concept of Langton's ant is that one cell in the discrete field is an ant, and all other cells are in one of two states (for this discussion, grey and blue). The ant moves forward with every generation, turning right

when encountering a grey cell and then leaving it blue. The ant turns left when encountering a blue cell and leaves it grey. These simple rules yield complex emergent behavior, especially when multiple ants are present [11]. A

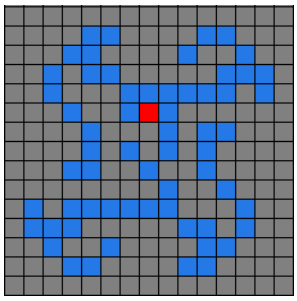

Figure 7: Langton's Ant after many generations.

single ant can yield interesting results, including building periodic self limiting structures. Multiple ants yield even larger structures, and can even work cooperatively. Langton found that his ants even replicated some phenomena of real life insect colonies [11].

3.4 Applications

Cellular automaton can do much more than create neat patterns: they can be used to model many aspects of the physical world, ranging from the movements of fluids to the growth of bacteria. In this section, different applications of cellular automaton are highlighted, ranging from the artistic to the scientific.

3.4.1 Modeling

Biology provides many phenomena that can be modeled by cellular automaton. One area that these models are effective is modeling predator-prey relationships. One such relationship, between aphids and ladybugs, has successfully been modeled in hopes of combating Citrus Sudden Death (CSD) disease in citrus trees. Figure 8 is from this study, and show the difference the

presence of ladybugs makes in aphid populations [12]. Because the aphids

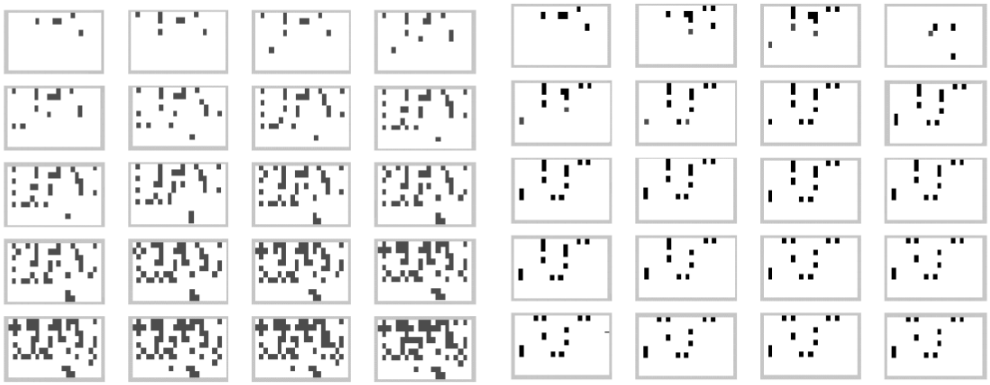

Figure 8: The amount of trees infected with aphid colonies, with and then without ladybugs present. Trees infected with aphids are grey and trees with ladybug colonies are black. [12].

are believed to infect the citrus trees, these models show that introduction of ladybugs into these areas could help control the aphid population, in hopes of reducing the spread of CSD [12]. The reason why cellular automaton work so well for predator-prey models is that they allow time and space to be taken into account. Naturally, a predator cannot eat prey that it is not close to. By making time and space discrete, these relationships can be modeled with relative ease.

For our next example, we move to earthquake simulation. These models use previous seismic information to give geographic information about the seismic activity. This geographic models help researchers better understand what these areas may hold in the future. The spacial nature of these models make cellular automaton a good fit [6]. Figure 9 shows a relative distribution of seismic activity in an circular area with radius 80kms around Skyros Island, Greece.

Our final example of a physical model comes from the world of physics. As Von Neumann and Ulam discovered, cellular automaton are effective at modeling the movements of fluids. One area where fluid dynamics are im-

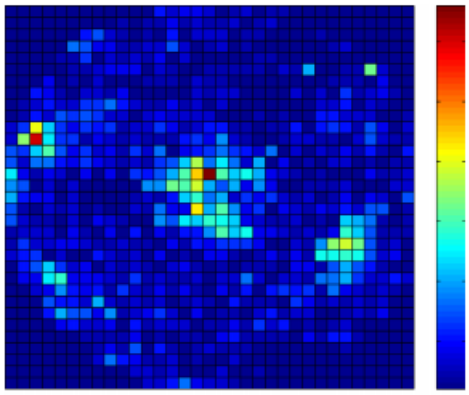

Figure 9: The distribution of seismic activity around Skyros Island, Greece [6].

portant is in gas pistons, used on internal combustion motors. Because the boundaries are moving along with the gases within, the model becomes computationally complex. With some simple modifications, cellular automaton can be extremely effective even in these situations as well [6]. Figure 10 shows a such a simulation.

3.4.2 Art

Artistic examples of cellular automaton (much like the software presented later), are not nearly as useful as the physical models, but no less interesting. Cellular automaton in any setting provide an interesting visual, but when used specifically for visual aesthetics they ascend to another level. A simple example of this using irregular geometries with Conway's game of life. When using these rules on a Penrose tiling, the results are beautiful (Figure 11) [6].

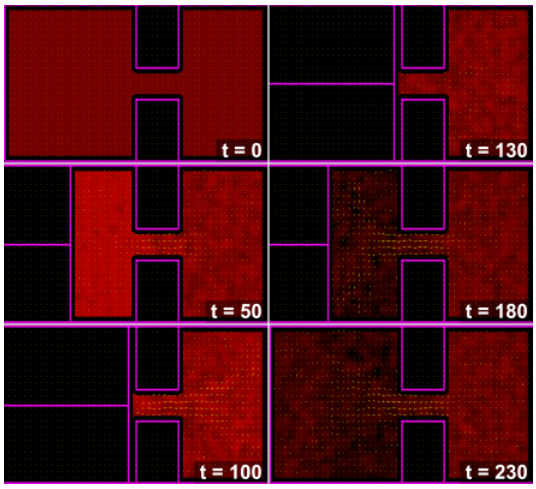

Figure 10: Simulation of piston motion using lattice-gas cellular automaton [6].

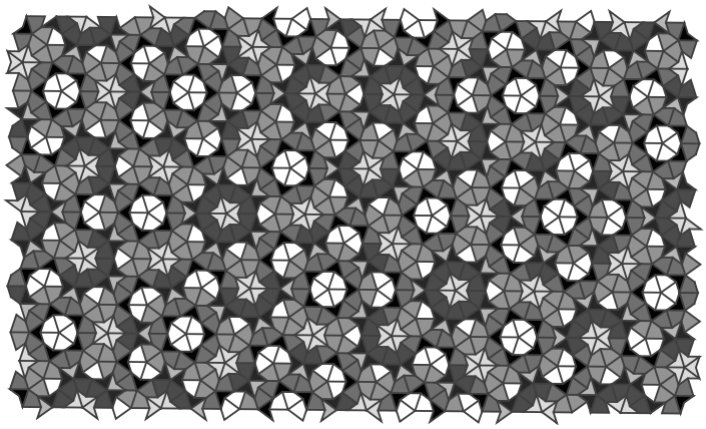

Figure 11: Conway's game of life on Penrose tilings [6].

Basic multi-state cellular automaton take on interesting appearances with their many colors. An interesting visual example is a cyclic cellular automaton using von Neumann neighborhoods with 38 states in 3D (Figure 12) [6].

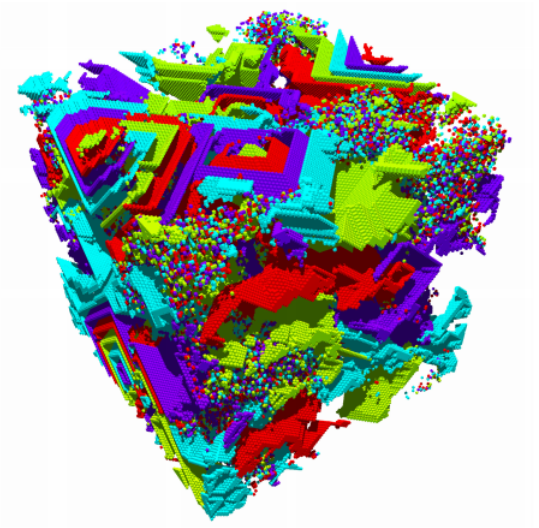

Figure 12: 3D cyclic cellular automaton [6].

Our final artistic example is derived from the work of geneticist Sewall Wright. The field is initialized in a random state, and at each iteration, a fair coin is flipped at every cell. If the coin is heads, the cell randomly takes the color of one of its neighbors, if it is tails, the cell remains unchanged.

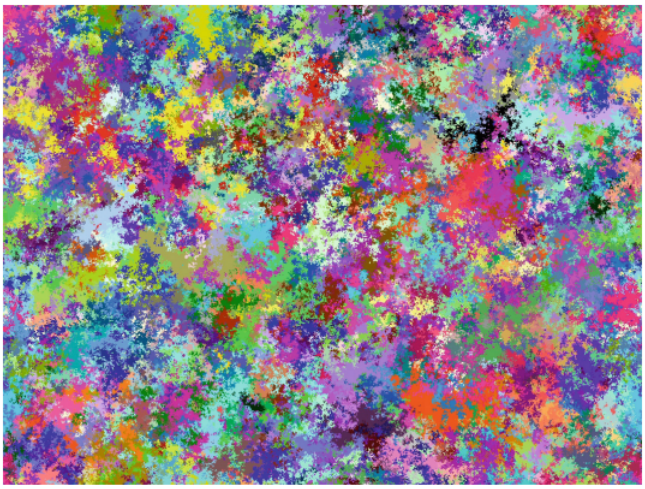

Figure 13: "Stepping Stone" cellular automaton [6].

This is model is based on Wright's prototype model for neutrally selective competition. As the iterations progress, large uneven patches of colors start to form, until the board is eventually taken over by one color [6]. Figure 13 represents an intermediate iteration.

3.4.3 Computer Science

Despite being the result of computers and some of the greatest computer science minds, cellular automaton have not had much usage in computer science. With the recent interest in cryptography and random/pseudo-random number generation, cellular automaton have been seeing increased interest. The often chaotic nature of these algorithms make them prime candidates for cryptography algorithms and pseudo-random number generation.

An interesting (and perhaps a bit less useful) application of Conway's Game of Life in computer science was the building of a Turing machine. Figure 14 shows the constructed Turing machine that uses three states and three symbols.

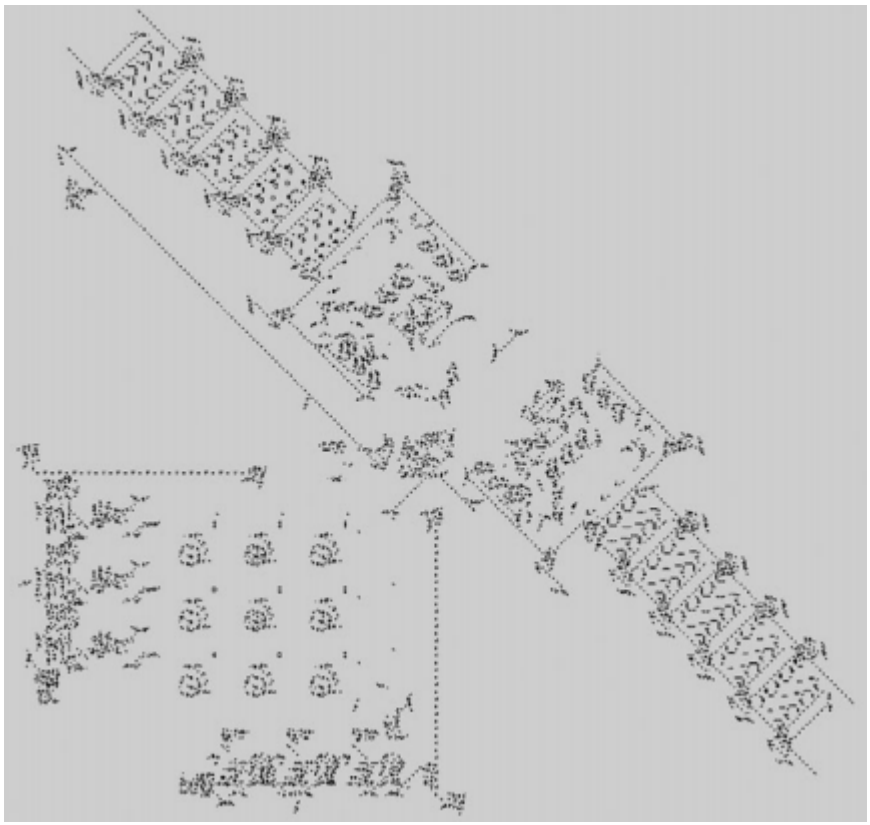

Figure 14: Conway's Game of Life Turing machine [6].

Beyond this simple Turing machine, it has been shown that a Universal Turing Machine can be constructed in Conway's Game of Life, showing that Conway's Game of Life is Turing Complete [6].

4 Background Music Theory

Then rest of this paper is dedicated to a specific artistic usage of cellular automaton: music. In order to understand how cellular automaton can be used to generate music, we must first introduce some basic physics and music concepts. Sound is a compression wave. The frequency of the wave determines the pitch of the sound, the shape of the wave is responsible for the timbre (or

"texture") of the sound and the amplitude of the wave indicates the volume. However, sound alone does not necessarily constitute music. Music, is the arrangement of sounds to be pleasing to the ear, a sort of "sonic art". This loose definition of music allows a large variety of sounds and arrangements to be considered music. Moving forward with this discussion, we are going to be talking generally about more western music. Notes, scales and chords will be discussed, along with some basics about progressions.

4.1 Notes

We start with the most basic building block of music: the note. A note can be thought of as a specific frequency, and in western music a note's frequency is usually denoted by a alphabet character from A to G. Something interesting happens however, when a frequency is doubled: the note remains the same, it is just one octave higher. Octaves are called octaves because notes are one octave apart when they are eight jumps apart on a major scale (which will be discussed in the next section). These notes sound very similar, so similar

C4	=	261.63 Hz
C4#	=	277.18 Hz
D4	=	293.66 Hz
D4#	=	311.13 Hz
E4	=	329.63 Hz
F4	=	349.23 Hz
F4#	=	369.99 Hz
G4	=	392.00 Hz
G4#	=	415.30 Hz
A4	=	440.00 Hz
A4#	=	466.16 Hz
B4	=	493.88 Hz

Figure 15: The Fourth Octave [10].

that people often have difficulty distinguishing them. Notes in western music are seperated by steps in frequency. A step is defined to be one twelfth the

distance between octaves. This distance is also called a semitone. A whole tone is two semitones [10]. With these concepts in mind, we can start to construct more complicated structures of notes.

4.2 Scales

A scale can be seen as a series of notes, that follow a specific pattern. Every scale has what is known as a key note, the first note of the scale, and the note that the scale is focused around. When discussing scales with a specific key note, say A, that scale is often said to be in the key of A. Scales are usually constructed from notes in one octave, so that they can played in any octave [10]. There are many patterns used to generate scales, so instead of going through all of them, we will just discuss the Major Ionian scale (a diatonic scale), which is widely used in western music. This scale contains seven notes, all in one octave. To express this pattern, it is easiest to lay out all semitones in an octave and label which belong to the scale:

Notes	I		II		III	IV		V		VI		VII	I
Semitones	0	1	2	3	4	5	6	7	8	9	10	11	12

As we can see, the jumps between notes are as follows: whole tone, whole tone, semitone, whole tone, whole tone, whole tone, semitone. In the key of C, this scale is CDEFGAB, which are just the white keys on a piano [8]. This explanation of a scale is about as basic as possible, as there are many other types of scales, which are used in wide varieties of music.

4.3 Chords

At the most basic level, a chord is multiple notes being played together. Chords are often notes that have consonance, that is they sound pleasing together, often due to constructive interference between the sound waves. Chords can also be dissonant, the opposite of consonant, but are not used in our current context. At the most basic level, chords are based off of scales,

usually using a root note, and its third and fifth. If the root note is the key note of the scale, these three notes are considered the "major triad" of a scale [10]. These triads alone makes nice sounding chord, but when using notes of the triad in multiple octaves, it generates a nice full sound.

Music that consists of a single chord alone often creates a feeling of stagnation, that the music "goes" nowhere. To give music a more dynamic feel, chord progressions are used. These are changes in chord as the song progresses. At the most basic level, a progression can be composed of simply two chords. On of the most common two chord progressions is I-V, where the chord changes back and forth between the key note and its fifth. Three chord progressions are also very common, but usually have a duplicate of one of the chords in order to give the progression symmetry. An example of such a progression would be I-I-VI-V. Chord progressions often sound pleasing when they resolve back to the key note or its fifth. Thus, many progressions involve starting at the key-note, moving to another chord such as the fourth or sixth, then resolving with the key note or its fifth [10]. These 4-chord sections can be extended to longer forms like the popular AABA 32 bar form or the 12 bar blues (I-I-I-I IV-IV-I-I V-VI-I-I).

5 Musical Cellular Automata

A possible application of cellular automaton, as mentioned earlier, is music generation. Due to the chaotic yet predictable nature of cellular automaton, mapping their states to musical notes could yield interesting, and perhaps beautiful results. In hopes of doing that, cellular automaton were implemented that play music based on the state of the board. A GUI (Figure 16) has be created that allows users to specify the nature of the music along with the cellular automaton rules that are used to create it. This GUI also provides a visualization of the cellular automaton.

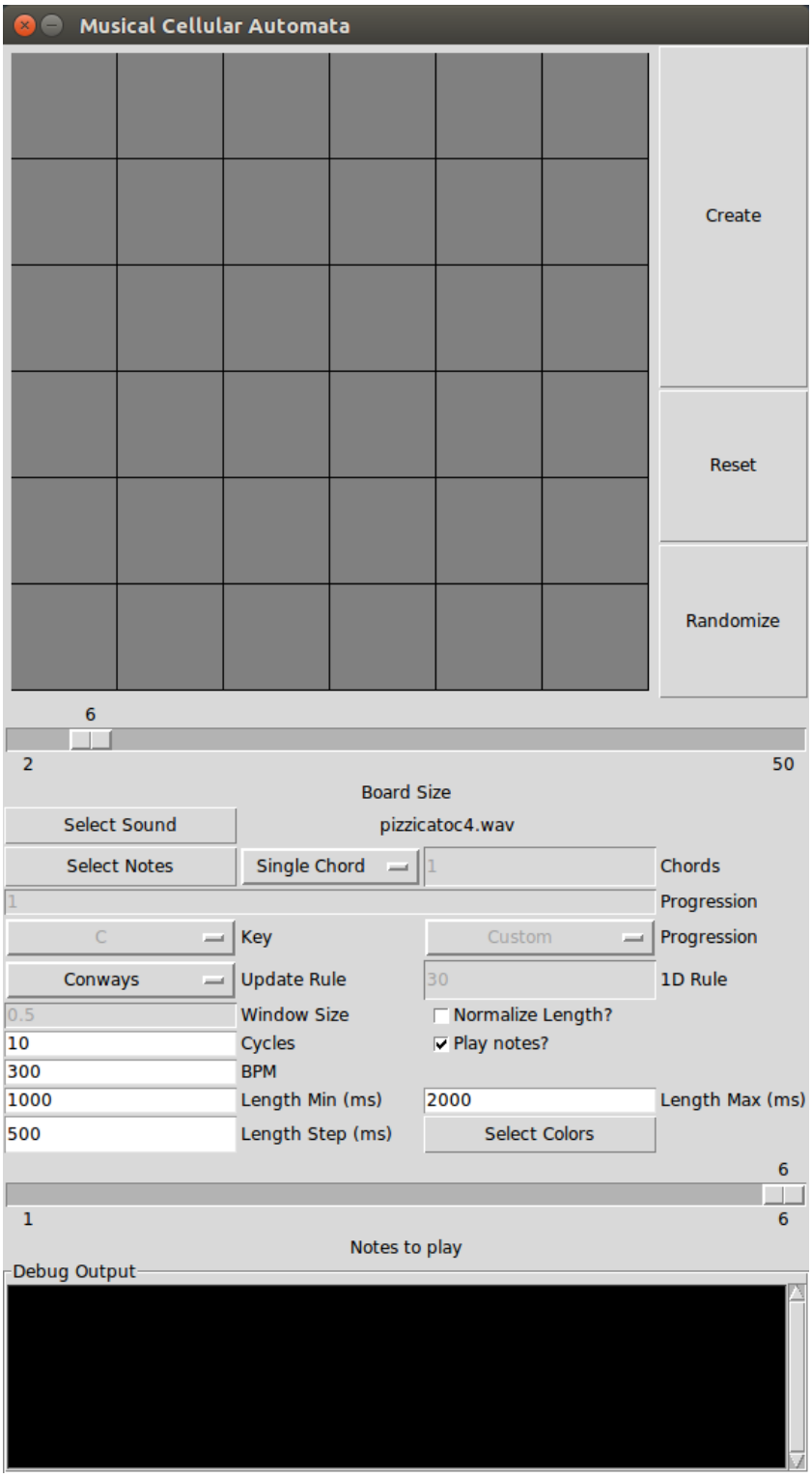

Figure 16: Musical Cellular Automata GUI.

5.1 Concept

The idea of this software is pretty simple: make sound a discrete field and use cellular automaton to modify it. The world of the cellular automaton is toroidal, the y axis represents time steps and the x axis are notes. This means that with every time step, all of the alive cellular automaton in a row play their note. The board is either updated with every time step or when the current step reaches the end of the board, depending on the rule. Beyond this, the software allows complete modification of both the sounds and the cellular automaton.

In the area of sound, there are three different "modes" for which the notes to be selected: single chord, multi-chord, generated-chords. Single chord is exactly as it sounds, there are no chord changes and the notes are played from one chord that is comprised of notes selected by the user, ranging from C2 to C9. Multi-chord allows for multiple single chords to be played in progression. Chord composition is selected for each chord like in single chord and the chord progression can be completely specified. Generated-chords is more algorithmic. It allows the user to select a key, which is used to generate chords from the major scale of that key. The user can then select a chord progression type, like 12 bar blues. Some of the chord progressions, like 3-chord and 32 bar, are generated randomly using basic assumptions to ensure they have a pleasing sound. Chord progression can also be specified by hand.

The notes themselves can be modified, the min and max play length of the notes can be specified, along with a step size between them. This allows for randomized note length. Because all of the different notes are generated from the same sound file, and length is changed when pitch is changed, there is an option to normalize the length of all of the different sounds. This process changes timbre of the sound, so a window size option (more on that later) allows the user to change the properties of the normalization of length to get different timbre changes.

For cellular automaton, the user can change the size of the board, the

update speed (which also functions as a sort BPM), and the rule sets used. The user can select from any of the elementary cellular automaton rules, along with a handful of 2D rules. For creative purposes, updating can be turned off altogether. When initializing a game, the user is able to hand craft the starting seed, or randomize it.

All of these possibilities together make for a powerful tool to generate randomized music. There are endless possibilities, ranging from fast shimmering chords to slower, melodic pieces.

5.2 Implementation

This software was implemented in Python 2.7, and requires a handful of dependencies:

`copy`	Used to deep copy arrays.
`math`	Used for manipulations to sound files and arrays.
`numpy`	Used to manage sound file arrays.
`os`	Used to access file paths.
`paulstretch`	Used to stretch sound files, slightly modified. Has a handful of dependencies of its own (`scipy`).
`Pygame`	Used to play and modify sounds.
`random`	Used for random selection and number generation.
`re`	Used to search for patterns in file paths.
`shutil`	Used to access operating system functions.
`time`	Used to ensure (relatively) consistent timing.
`Tkinter`	Primary GUI package.
`wave`	Used to access and modify .wav files.

Many of these libraries are included in python, or can be downloaded via `pip`. The implementation is split between two different files, `AutomataApp.py` and `SoundAutomata.py`. The first file contains the GUI classes and is the primary driving script. The latter file contains a class that manages the generation

of sounds and updates of the cellular automaton. The following sections will dissect these files in greater detail.

5.2.1 Cellular Automata

The cellular automaton are primarily implemented in the `SoundAutomata.py` file. The updates however are called in `AutomataApp.py`. The automaton states are stored an array as integers (0 - $n-1$, for n states). The different types of updates are implemented as different functions. Conway's, seeds and Brian's Brain (see Appendix A) are implemented using their standard algorithms. Langton's Ant is implemented using 10 different states, 8 of which are used for the ant (see Appendix B). The elementary rules are all implemented in one function (see Appendix C), that takes the rule number as a parameter. The rule number is converted to binary, and the binary string is used to determine when a cell is alive or not.

5.2.2 Sound

Sound in this software is a little more complicated. Everything sound related is implemented in `SoundAutomata.py`. `Pygame` is used to play all sounds. Any sound in the 16-bit wave format can be used, and sounds are selected in the GUI. Sounds are preferred to be a C4 note, as the GUI and scale generation are based around this assumption. When a `SoundAutomata` class is constructed, a function call is made to the `generateNotes` function that generates all of the needed notes (see Appendix D). The original sound file is read into a `numpy` array. The notes are generated by resampling the sound to the proper frequency ($C4 * 2^{f/12}$, where f is the number of semitones to be moved and C4 is the frequency of a C4). If normalize length is checked, the sound is then stretched back to the original length using the `paulstretch` algorithm. This algorithm takes overlapping sections of the sound array and makes them overlap more or less based on the stretch needed. This affects the timbre, but can yield pleasing results. Once the new sounds are generated,

they are saved, so they do not need to be regenerated every time they are used.

All needed sounds are then loaded into main memory, so they can be quickly accessed. Sounds are played by stepping through the current row. If the state is not 0, the note is played. On rule sets with more than two states, the higher the state value, the lower volume the sound is played. This gives some of the automaton some dynamic range. Length is selected randomly using the min and max length along with the step size. After all the notes in the row have been played, the current time is saved. After all calculations are made, a loop polls the time until the proper interval is met, then the next set of sounds are played. This ensures that the calculation have a minimal effect on the timing of the music.

Scales and chords are generated based on just a given key. The random-chords mode only uses the major Ionian scale, so generation of the scale in a given key is simple. Chords are generated from a specific tone by finding its third and fifth, and then adding those notes in a few different octaves. The key notes a few octaves lower is added to give some bass. The end result is 13 different notes in each chord, giving a large range of sounds.

5.2.3 Graphical User Interface

The GUI is implemented entirely through TKinter. The GUI is composed of a large root window that hosts all of the various widgets used to change the properties of the musical cellular automata. This window can be seen as a "launcher" for the cellular automata. Aside from this window, there are two other important windows: the note select window and cellular automata visualization window. The former is used to select what notes are in a chord and the latter shows a visualization of the musical cellular automata as it runs. TKinter code can be quite long, so only the class for the note select window is included (see Appendix E).

5.3 Limitations and Further Work

There are a few limitations of this software. The primary limitation is that Python is an interpreted language. This leads to timing issues, as background garbage collection and other Python "bookkeeping" functions take up time. The inconsistencies in timing are only really noticeably at slower speeds and very high speeds. When chords contain many notes, it becomes barely audible. The other primary limitation, is that stretch and resampling audio compromises the timbre of the sound. This can lead to pleasing results, such as the "ethereal" sound of large window sizes when normalizing the length, but makes it difficult to faithfully reproduce the sound of an instrument.

Further work on this software could be to make multi-timbre instruments like a drum set or a piano that uses different recordings for each note, to make a more faithful reproduction. Another avenue of possible work would be to implement each of Wolfram's classes of cellular automaton using parameters. This is beyond the purpose of generating music, but would generate interesting visuals. In the software's current state, chord progression and generation is very simple. More complicated chords and progressions could be generated using different scales. Chord progressions could also be generated using more complicated structures to yield better randomization while maintaining pleasing sound.

5.4 Results

Determining how effective the software is at generating music is difficult. The software most certainly generates sounds that are in key, but like all "music" generated via algorithms, there is something distinctly artificial about it. The primary reason in this instance is that the sounds are all generated from the same original note, causing the timbre to be exactly the same, note to note. This is different from a normal instrument, which has a variety of timbres. The best sounds that come from the software are from large seed

sizes, fast update speeds, longer chord progressions and normalized sounds. The normalized sounds lose some of the sharper parts of the timbre and create full, shimmering, almost ethereal, chords.

6 Conclusion

There are many types of cellular automaton, ranging from simple elementary automaton to the more complex and chaotic "life-like" games. Cellular automaton are important for modeling various physical phenomena. These automaton can also be used generate both visual, and as shown in this paper, sonic art. The chaotic nature of many of the cellular automata rule sets allow the creation of highly randomized and cyclic music. The software in this paper is just one primitive way to generate music and hopefully will be used as a stepping stone towards generating more realistic music.

A Brian's Brain Update Function

```
## Updates the cellular automata according to the rules of Brian's Brain.
def bBUpdate(self):
        for i in range(self.size):
            for j in range(self.size):
                count = self.countneighbors(i,j)
                while switch(self.gameBoard[i][j]):
                    if case(1):
                        self.gameBoardTemp[i][j] = 2
                        break
                    if case(2):
                        self.gameBoardTemp[i][j] = 0
                        break
                    if case(0):
                        if count == 2:
                            self.gameBoardTemp[i][j] = 1

                        break

        self.gameBoard = copy.deepcopy(self.gameBoardTemp)
```

B Langton's Ant Update Function

```
## Updates the cellular automata according to the rules of Langton's Ant.
def langtonsUpdate(self):
        for i in range(self.size):
            for j in range(self.size):
                current = self.gameBoard[i][j]
                while switch(current):
                    if case(2):
                        self.gameBoardTemp[i][j] = 0
                        if self.gameBoard[(i+1)%self.size][j] in
    [0,6,7,8,9]:
                            self.gameBoardTemp[(i+1)%self.size][j] = 9
                        else:
                            self.gameBoardTemp[(i+1)%self.size][j] = 3
                        break
                    if case(3):
                        self.gameBoardTemp[i][j] = 0
                        if self.gameBoard[i][(j-1)%self.size] in
    [0,6,7,8,9]:
                            self.gameBoardTemp[i][(j-1)%self.size] = 6
                        else:
                            self.gameBoardTemp[i][(j-1)%self.size] = 4
                        break
                    if case(4):
                        self.gameBoardTemp[i][j] = 0
                        if self.gameBoard[(i-1)%self.size][j] in
    [0,6,7,8,9]:
                            self.gameBoardTemp[(i-1)%self.size][j] = 7
                        else:
```

```
                                self.gameBoardTemp[(i-1)%self.size][j] = 5
                            break
                        if case(5):
                            self.gameBoardTemp[i][j] = 0
                            if self.gameBoard[i][(j+1)%self.size] in
    [0,6,7,8,9]:
                                self.gameBoardTemp[i][(j+1)%self.size] = 8
                            else:
                                self.gameBoardTemp[i][(j+1)%self.size] = 2
                            break
                        if case(6):
                            self.gameBoardTemp[i][j] = 1
                            if self.gameBoard[(i+1)%self.size][j] in
    [1,2,3,4,5]:
                                self.gameBoardTemp[(i+1)%self.size][j] = 3
                            else:
                                self.gameBoardTemp[(i+1)%self.size][j] = 9
                            break
                        if case(7):
                            self.gameBoardTemp[i][j] = 1
                            if self.gameBoard[i][(j-1)%self.size] in
    [1,2,3,4,5]:
                                self.gameBoardTemp[i][(j-1)%self.size] = 4
                            else:
                                self.gameBoardTemp[i][(j-1)%self.size] = 6
                            break
                        if case(8):
                            self.gameBoardTemp[i][j] = 1
                            if self.gameBoard[(i-1)%self.size][j] in
    [1,2,3,4,5]:
                                self.gameBoardTemp[(i-1)%self.size][j] = 5
                            else:
                                self.gameBoardTemp[(i-1)%self.size][j] = 7
                            break
                        if case(9):
                            self.gameBoardTemp[i][j] = 1
                            if self.gameBoard[i][(j+1)%self.size] in
    [1,2,3,4,5]:
                                self.gameBoardTemp[i][(j+1)%self.size] = 2
                            else:
                                self.gameBoardTemp[i][(j+1)%self.size] = 8
                            break
                        break


        self.gameBoard = copy.deepcopy(self.gameBoardTemp)
```

C 1-D Update Function

```
## Function that updates the automata based on the specified 1-D rule.
def oneDUpdate(self, rule):
        binRule = bin(int(rule))[2:].zfill(8)
        for i in range(self.size):
```

```
            prevVal = []
            for j in range(i-1,i+2):
                curj = j % self.size
                prevVal.append(self.organismAt((self.currentNote-1)%self.size,
                    curj))

            while switch(prevVal):
                if case([1,1,1]):
                    self.gameBoardTemp[self.currentNote][i] = binRule[0]
                    break
                if case([1,1,0]):
                    self.gameBoardTemp[self.currentNote][i] = binRule[1]
                    break
                if case([1,0,1]):
                    self.gameBoardTemp[self.currentNote][i] = binRule[2]
                    break
                if case([1,0,0]):
                    self.gameBoardTemp[self.currentNote][i] = binRule[3]
                    break
                if case([0,1,1]):
                    self.gameBoardTemp[self.currentNote][i] = binRule[4]
                    break
                if case([0,1,0]):
                    self.gameBoardTemp[self.currentNote][i] = binRule[5]
                    break
                if case([0,0,1]):
                    self.gameBoardTemp[self.currentNote][i] = binRule[6]
                    break
                if case([0,0,0]):
                    self.gameBoardTemp[self.currentNote][i] = binRule[7]
                    break
                print("The 1D rules broke. What did you do?")
                break

        self.gameBoard = copy.deepcopy(self.gameBoardTemp)
```

D Sound File Generation Function

```
## Generates audio files of all of the specified pitches based on the
## original audio file. Sound length can be normalized.
def generateNotes(self):
        self.parent.write("Generating notes...")
        for i in range(-36,60):
            for key in self.key:
                if i in key:
                    if not os.path.exists(self.basicNote[:-4] + os.path.sep +
                        str(i) + '.wav'):
                        factor = 2**(1.0 * i / 12.0)
                        if self.lengthAdjusted:
                            (samplerate,smp)=paulstretch.load_wav(
                                self.basicNote)
```

```
                                paulstretch.paulstretch(samplerate, smp, factor,
                                    self.windowSize,
                                    self.basicNote[:-4]+"temp"+".wav")
                                note = pgm.Sound(self.basicNote[:-4]+"temp"+".
    wav")
                            else:
                                note = pgm.Sound(self.basicNote)

                            note.set_volume(0)
                            note.play()
                            basicNoteArray = pgsa.array(note)
                            basicNoteResampled = []
                            for ch in range(basicNoteArray.shape[1]):
                                sound_channel = basicNoteArray[:,ch]
                                basicNoteResampled.append(np.array(
                                    self.speedx(sound_channel, factor)))

                            basicNoteResampled = np.transpose(np.array(
                                basicNoteResampled)).copy(order='C')
                            noteOut = pgsa.make_sound(basicNoteResampled.astype(
                                basicNoteArray.dtype))
                            noteFile = wave.open(self.basicNote[:-4] + os.path.
    sep +
                                str(i) + '.wav', 'w')
                            noteFile.setframerate(44100)
                            noteFile.setnchannels(2)
                            noteFile.setsampwidth(2)
                            noteFile.writeframesraw(noteOut.get_buffer().raw)
                            noteFile.close()
                            self.parent.write(self.parent.notes[
                                i%len(self.parent.notes)] + "(" +
                                str(i//len(self.parent.notes)+5) + ") Generated!
    ")

        if os.path.exists(self.basicNote[:-4]+"temp"+".wav"):
            os.remove(self.basicNote[:-4]+"temp"+".wav")
```

E Note Selection Class

```
## Class that governs the window to select new notes.
class NotesSelectWindow(tk.Toplevel):
    ## Intializes all values and GUI elements.
    def __init__(self, parent, val):
        tk.Toplevel.__init__(self, parent)
        self.wm_title("Select Key/Chord " + str(val+1))
        self.parent = parent
        self.val = val
        self.noteArray = []
        self.noteCheckArray = []
        self.octaveLabelArray = []
        for i in range(-36,60):
            if i in self.parent.key[val]:
                self.noteArray.append(tk.IntVar(self, 1))
```

```
15              else:
16                  self.noteArray.append(tk.IntVar(self, 0))
17
18          for i in range(8):
19              self.octaveLabelArray.append(tk.Label(self, text = str(i+2)))
20              self.octaveLabelArray[i].grid(row = i, column = 0)
21              for j in range(12):
22                  self.noteCheckArray.append(tk.Checkbutton(self,
23                      text = self.parent.notes[j],
24                      variable = self.noteArray[i*12 + j]))
25                  self.noteCheckArray[i*12+j].grid(row = i, column = j+1,
26                      sticky = "nsew")
27
28          self.confirmButton = tk.Button(self, text="Confirm",
29              command = self.confirm)
30          self.confirmButton.grid(row = 12, column = 0, columnspan = 13,
31              sticky = "nsew")
32
33      ## Function that confirms selection and closes the note select window.
34      def confirm(self):
35          key = []
36          for i in range(len(self.noteArray)):
37              if self.noteArray[i].get() == 1:
38                  key.append(i-36)
39
40          self.parent.key[self.val] = key
41          self.destroy()
```

References

[1] Freesound database. `https://www.freesound.org/`.

[2] Numpy and scipy documentation. `https://docs.scipy.org/doc/`.

[3] Pygame documentation. `http://www.pygame.org/docs/`.

[4] Python 2.7.13 documentation. `https://docs.python.org/2/`.

[5] Adrew Adamatzky, editor. *Game of Life Cellular Automata*. Springer, London, England, 2010.

[6] Adrew Adamatzky and Martinez Genaro, editors. *Designing Beauty: The Art of Cellular Automata*. Springer, London, England, 2016.

[7] Iwo Bialynicki-Birula. *Modeling Reality: How Computers Mirror Life*. OUP Oxford, Oxford, England, 2004.

[8] Jeff Brent and Schell Barkley. *Modalogy: Scales, Modes and Chords: The Primordial Building Blocks of Music*. Hal Leonard, Milwaukee, WI, 2011.

[9] Martin Gardner. Mathematical games: The fantastic combinations of john conway's new solitaire game "life". *Scientific American*, (223):120–123, 10 1970.

[10] Leon Gunther, editor. *The Physics of Music and Color*. Springer, London, England, 2012.

[11] Christopher G Langton. Studying artificial life with cellular automata. *Physica D: Nonlinear Phenomena*, (22):120–149, 10 1986.

[12] Thomas M. Li. *Mathematics Research Developments : Cellular Automata*. Nova, Hauppauge, US, 2011.

[13] GJ Martinez. Computation and universality: Class iv versus class iii cellular automata. *Journal of Cellular Automata*, (7):393–430, 5 2012.