# GENERATIVE MUSIC WITH WEB AUDIO

*Continuous Generated Music Stream on the Web*

**Ethan Ooi**
*MEng Computer Science*
*Third Year Project at*
*University of Manchester*
*Supervisor: Toby Howard*
*April 2019*

**Abstract**

The purpose of this project is to create an endless stream of Generated Music. This will involve the implementation of ideas from Music Theory, specifically for composition. This is carried out using Javascript's built in WebAudio API. The way to build a basic music generator, as well as all the compositional techniques implemented on top of this, will be investigated in detail. An Evaluation is carried out to measure the success of this project.

## Acknowledgements

I would like to extend my gratitude to my supervisor Toby Howard, for keeping me on track during this project. Also many thanks to my good friend Nikolai Lester for providing me with his extensive knowledge of music theory.

# Contents

# 1 A Musical Context

'Generative Music', a term popularized by English Producer Brian Eno, describes a form of musical composition which creates a continuous stream of music that changes over time. A notable work, is an album by Brian Eno: Ambient 1: Music for Airports (1978) [1] , widely believed to contain the first instances of what Eno would go on to describe as 'Generative Music', which he believed had to match both these criteria:

- It must change continuously and never repeat itself exactly.
- It must last forever.

An example of one of the techniques used, for Music for Airports, Brian Eno used magnetic tapes to store sounds that he would loop into themselves. These tapes were of specific lengths that when all started at the same time, would take a long time to all fall back in sync. [2].

The cyclic long drawn out nature of this generative technique makes it very popular for creating a certain style of music- the soundscape, also commonly known as Ambient Music (a term also coined by Brian Eno).

Ambient Music uses a sparse texture, with only a few (3-4) musical voices, most of which playing long drawn out harmonious single tones, called drones. This is done to create a relaxing atmosphere for the listeners, and is commonly used as a background to meditation or studying; It also isn't uncommon to hear actual sounds from nature overlaid on the music to further calm and relax the listener. [3]

Brian Eno later went on to collaborate with programmer Peter Chilvers to create 'Scape' for the Apple iPad. This also created music in an ambient style, however, it was designed to be heavily interactive, generating and changing music in real time alongside the users non-musical abstract input. Two other notable mobile application ambient music generators are 'Wotja', and 'Reflection'.

Another way in music is generated is through Algorithmic Composition programs. These are only used to aid the composition process, giving the composer ideas and common 'helper functions' to streamline music writing, and not to generate complete audio. Some algorithms also utilize phenomena that have no musical relation, such as magnetic field measurements, or self-similar fractal shapes to generate interesting sounds.

Other methods such as Grammar based systems, or Evolutionary based musical systems also exist, however the widespread use of this is currently limited.

Mathematical Models can also be used, commonly involving stochastic processes, to produce music as a result of non deterministic methods. An example of one these processes are Markov Chains, where various aspects of the music- Melody, Harmonic Transition, and even Rhythm can be controlled by independently running Markov Chains, wherein transitional probabilities are initialized by the programmer/composer.

The style of music I would want to try and emulate would greatly influence my decision on which generational methods I would use. I wasn't too keen on generating ambient music, as it would not have contained as much musical complexity as I would have liked, and would not have pushed my musical understanding very far.

I then started thinking about the long drawn out changing nature of the musical style of minimalism, with tracks often upwards of an hour long. Notable composer Steve Reich was composing rich soundscapes without the use of long drones or excessive amounts of reverb. One key example is Steve Reich- Electric Counterpoint (Mvt. 3) [4], where it is composed of various guitar-family instruments playing repeating and ever-permuating cells of music. This creates a unique texture which can easily blend into the background, but can also be tantalizingly rich and intricate when closely listened to.

Common musical features of minimalism are as follows [5]:

- Holding or repeating the same note over and over again
- Ostinato is the basis of all minimalist music, wherein a sequence of notes or a rhythm is repeated over and over again. This sequence is often referred to as a rhythmic cell.
- Cells can variate in the notes they are playing, or in rhythm, or both. A common technique is to take the existing rhythm of the cell, and add or remove notes, gradually mutating the cell over time.
- Metamorphosis is when a few notes of the cell are changed at a time to gradually move its tonal color from one place to another
- Multiple cells layered on top of each other to create contrasting interactions between the cells. Layers can be added or removed at any time.

I was very drawn to the idea of building and growing a piece of music through layering cells on top of each other to create a rich intricate texture. I also realized that through minimalist techniques and stochastic processes, I had the basis to create one of the first generative music processes which created music in this style. It was from this point on that I decided I would use Mathematical Models to create a system where minimalism would be generated.

With this rough idea in mind, I went on to create a set of goals and ideals that I would build my music generation algorithm out of.

## 2 Project Goals

- **To create relaxing music that sounds nice**
  I would like to create music that is aurally pleasing to the listener. This is the most important goal for me, as accomplishing this would require me to learn and implement music theory knowledge.
- **To serve no more than a musical background, should not grab attention from the listener, and to have this texture slowly evolve and mutate**
  I would like to select combinations of notes, and transitions between notes which do not have musical intent (sounds like it wants to go somewhere), to avoid any change that would perk the listeners attention. However, I will also need to achieve a juxtaposed goal of making my music interesting and intricate at the same time.
  I would like components of the current sound to independently variate themselves, to add cohesion and continuity to the continuous texture of music generated. Doing different parts independently softens the effect of the change on the listener, possibly making them unaware of continuous minute changes to the music.
- **To have Simple minimalist graphics to accompany the music which give an insight into the different individual layers that build up to form the overall sound**
  Having relaxing graphics in soft, non-attention-grabbing colors will further imply what the envisioned use case for this is- as a background.

## 3 Overview of Technologies Used

- Standard Modern Javascript (ECMAScript)
  - Built in WebAudio for producing sounds on the browser
  - Built in WebGL for graphics
  - GL-matrix library for matrix transformation used in the graphics
- Basic HTML

This list is quite short, as I believe no more than this is required to implement everything I want to. The minimal technology stack makes the program fast, with high accessibility on almost any device, with users being able to go to my simple webpage on any device and start generating a musical wallpaper.

Another advantage is that everything is extremely lightweight, 42KB of my own things, and 100KB for the gl-matrix helper functions; Though it isn't that big of an advantage nowadays in the times of cheap storage and cheap networking.

# 4 Development Process

## 4.1 Ways in which design philosophy changed throughout the project

In the beginning, the idea was to have a user-input-centric approach to designing features. An early idea that I was exploring was to have users touch on big color coded squares on the screen in order to change the direction/parameters of music generation, and have it become a sort of 'aided musical improvisation' platform. However, as I gave more thought to how much effort/learning it would take the average user to produce good sounding music, I realized taking control away from the user would allow the programmer to have more control over the overall direction, and more importantly, tighter control over the quality of the music that would be produced.

From this point on, I realized that letting the user have a large degree of control would overall detract from the music produced, especially if a user would have to continuously interact with the program to generate music. From then on my goals changed to my current philosophy of having a completely automated music generation system.

## 4.2 Design Philosophy

The design philosophy of the project is to create pleasing, intricate, yet not attention grabbing music.

Ideally it should be easy for beginner users to start generating music which sounds pleasing, and should avoid scenarios where user input can cause distasteful, unorderly music. However, more advanced users should be able to influence the directionality of the piece, and expert users should be able to input their own material within this framework.

This should be accompanied by relaxing graphics representative of the music being played.

## 4.3 Back to Basics

The basic ingredient for everything to work is to have a simple working function:

```
playNote(frequency, startTime, duration)
```

Which just plays a certain frequency from startTime to startTime+duration. I would focus on building this one function as a useful abstraction to begin building the rest of the project on, and this where the majority of WebAudio itself is being used.

One thing I found out quite early on was that a note had to be played by an Oscillator object, a special datatype from the WebAudio API.

Each Oscillator is only able to be played once, and stopped once. I will have to create a new Oscillator for every single note that I played, and there would be huge memory problems if I kept every single one.

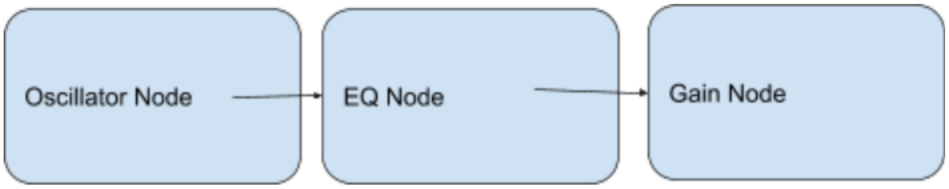


Figure 1: How Nodes should be connected

The problem is further exacerbated because each Oscillator Node had also to be passed to brand new EQ and Gain Nodes.

My solution was to utilize circular (ring) buffers- I created three arrays to store each of the Node types, and I determined that the size of 30 for each ring buffer was sufficient to avoid any collisions. I have a variable to track where the next free space is, which is incremented and mod 30 every time a note is played.

When a note is to be played, the space in the circular buffer is nulled to queue it for collection by the Javascript GC, and new Oscillator, EQ, and Gain nodes are put in the free spaces.

I now have a method which can produce sound at a given frequency and length of time.

### 4.4 Helper Functions

Helper functions were written in order to connect various parts of the program:

getScale(key, scale) - Constructs the set of notes in that scale for that key

scatter(inputnotes) - Initialize the octave value for all notes to random octaves

key_transpose(note,setting)- Calculate what the resulting note is after going up or down <setting> number of semitones.

findDistance(note1,note2) - Get the distance in semitones between the two notes

notetoFreq(note)- Returns the frequency of a given note

## 4.5 The different principles of music used

The following describes the different musical theory principles used in order to choose notes to play, with respect to my goals. A very rudimentary knowledge of music theory is required, which can be gained through reading the quick guide found in Appendix A.

### Polyrhythms

A polyrhythm is a rhythmic artifact which emerges when one unit of time is divided in different ways among different voices. Multiple Layers of polyrhythms can be combined to create a dense web of sound. The contrasting ways that the same amount of time is split up creates a fragmented texture among the voices, which allows multiple voices cohesively play different things together without them interfering with each other. This is a commonly used technique, especially in Minimalism.

Figure 2: An example of 5 equally spaced notes, and 3 equally spaced notes, each spanning the same one unit of time.

### Chord Construction

A Chord is a collection of notes played within a short period of time. How does one select a collection of pitches that sound good together? Which octave should each of the pitches be? Chords are a useful model which abstractifies the complexity of selecting individual notes.

Lets take a simple C Major7 chord for example, which contains the notes C E G B.

It could be voiced like this: [C2, E3, G4, B4] or voiced like this: [C4, E3, G5, B2] How do we know which of these chords is best?

The answer is- it depends.

We must first look at how chords move from one to another on the lowest level: How each individual note of the chord moves to a note in the next chord.

### Multiple Voices

A Voice is a single melodic line, only able to play one note at a time. Music with only one voice would get very boring very fast, and thus, composers of Classical Music from the 1400s to the 1700s would begin refining a framework for interactions of more than one voice- describing how the interactions of a collection of simple melodies could create beautiful rich harmonies as a whole. This field of study is related to the rules of Voice Leading.

When going from one chord to the next, we can imagine this as the first note of the first chord moving to the first note of the second chord, and so on... Voice leading is a powerful heuristic measure to determine the quality of a transition. In my implementation, this is done through employing two principles of voice leading in order to best select transitions:

- **Principle of least movement**- Minimize the distance of each note transition to make movements less jarring
- **Forbid voice crossing**- Maintaining a strict hierarchy of the order of voices, forcing voices to select transitions that do not cross the boundary of another voice. This makes the syntactic breaking down of the different voices even more apparent.

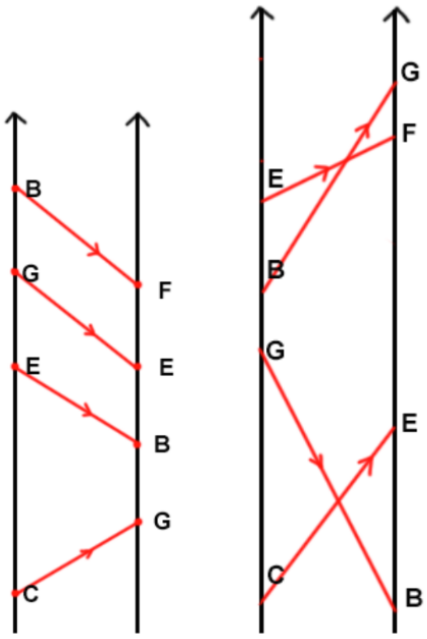


Figure 3: Examples of good voice leading versus Poor voice leading

These two simple principles massively improves the quality of music generated by further emphasizing the separation of voices, minimizing jarring voice movements, and having each of the voices select their transitions based on the (near) past and (near) future actions of the other voices, giving rise to a self-interacting evolving system. [6]

### Chord Transition

In every single key, there contains 7 diatonic chords; 1 for each note of the key. Given a certain key and a current chord, how is the next key and chord selected? For example, in the key of C Major, any arbitrarily length sequence of the chords: Am, Bm, C, Dm, Em, F, G are valid sequences.

How can I determine a good sounding sequence of chord transitions?

#### Modulating

Modulation looks at the problem of- When should I transition into another key? What key should that be?

Thinking about chords as states in a determinate finite automata, we can define transitions between chords in a key, as well as transitions from states in one key to states in another, each as a probability of transition. The probabilities will have to be inserted manually as a program parameter, however this can be done as somebody with experience in music.

### 4.6 Implementing these principles in code

#### Polyrhythms

A working rhythm engine had to be first developed that would subdivide up a bar into n equally sized slices, and encode in these slices weather or not a voice should be playing, holding a note, or not playing a note at all.

The engine would then have to run through the slices of each bar, and queue notes to play or hold that the appropriate times.

I developed a shorthand notation system using these 4 symbols: I, i, o, x, where rhythm would be encoded as a string.

- I- Play the note loudly
- i - Play the note softly
- o - Hold the note currently being played
- x - No note played

An illustrating example would be the string: 'Ioooxxxi', which encodes:

Divide the bar into 8 equal lengths of time. On the first slice, play the note loudly, and hold the note for the duration of 3 additional slices. Stop playing for the next 3 slices, and play the note softly on the last slice.

The part of my program reading these strings will go through them token by token, and trigger the playNotes function at the correct times, adding the correct amount of time to hold each note, denoted by 'o'.

#### Multiple Voices

I created an object type for a Voice, encoded with information about what note it should play, as well as the rhythm at which it should play it. The Voice objects also have a class method which take as input the time at which the rhythm should be played, and at what tempo it should be played.

I can then create an array with each of these Voice objects, from which I could add or remove voices, or modify their notes or rhythm.

With the ability to add multiple voices, I now set each of the voices to rhythms of differing lengths.

An example of a 10 over 6 polyrhythm can be encoded as:

- Voice 1- 'Ixixixixix'
- Voice 2- 'Iooixx'

### Rhythmic Variation

To construct and generate rhythmic strings of various lengths, I encoded weighted probabilities for various small rhythmic cells, ie- Io or ii or ix ... , and using the probability of each to append enough of these units to fill a desired length for every voice.

It is weighted in such a way that it emphasizes down beats, making the pulse of each polyrhythm more apparent.

### Chord Construction

I had to create a function which generated chords based on multiple factors.

```
Input: (Key, Degree, Height, Scale)
Output: Array containing <<Height>> amount of notes.
```

I chose to generate my chords with a simple interval stacking method. In the key of C, the 1st degree is C, the second degree is D, and so on..

Likewise in the key of G, the 1st degree is G, the second degree is A and so on.

In order to generate say a D Major chord in G major, we first look at the scale of G Major:

```
G, A, B, C, D, E, F#, G, A, B, C, D, E, F# ...
```

Major is the scale, and is used in the most commonly heard music. I have implemented other scales as well, but for understandability I will focus on the major scale, however the algorithm works exactly the same in other scales.

D is the 5th degree of the G scale. I would like to stack 4 notes on top of each other, so the height will be set to 3.

The function input will look like: (Key: G, Degree: 5, Height: 3, Scale: Major)

We push the first note onto the stack, now the chord looks like: [D] We then go up the desired interval (determined by scale), in the case of the major scale, we skip one, and push the next one: [D, F#]

This is repeated until height+1 notes exist in the array: [D, F#, A, C].

Now that Pitch class has been determined, we now need to figure out what octave each of the notes should be. This is the most important problem to solve, as it determines the voice leading.

The two principles of voice leading outlined above were implemented to determine an acceptable configuration for the next chord:

### Principle of Least Movement

When any voice moves from one note to the next, a decision has to be made on which octave that note will be played on.

I implement this principle by selecting the closest target note to the previous note as my chosen note. Worked Example:

One voice has to go from E4 to any G. I measure the distance in semitones (1/12 of an octave) from E4 to each of G3, G4, G5, and select the closest one.

### Forbidden Voice Crossing

The selected notes are then evaluated for voice crossings, to ensure the notes selected are within the bound drawn by the lowest note of the voice above it, and the highest note of the voice below it. This overwrites the rule of least movement, and if a crossing is detected, the resulting note is readjusted in terms of its octave, to fit the bounds.

### Modulating and Chord Transition

This focuses on determining the input parameters for the chord constructor.

Primarily, which degree chord from which scale should be generated? Each of the 7 chord degrees for each key can be represented as nodes in a markov chain graph (With the labels 1-7).

Every bar of the program, the chord will take a transition to another chord.The probabilities of transition between each of these states is manually inserted.

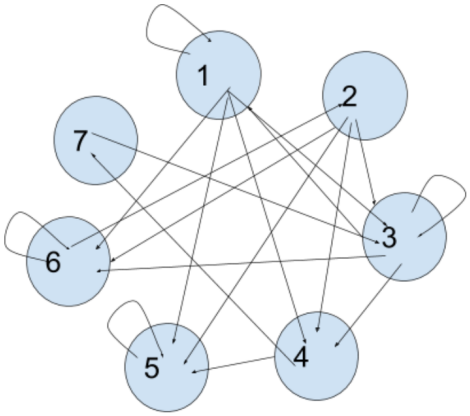


Figure 4: An example of possible intra-key transitions.

This is implemented using a switch statement with the current chord as input to denote starting node, and each case containing weighted probabilities to select the next chord.

### Inter-Key Modulation

I have chosen to implement the most common key modulations: The 5-1 and the 4-1. In the 5-1, you start by playing the 5th degree of the initial scale, you then modulate to the key of whichever note the 5th degree was, and play the 1st degree in this new key.

Table 1: 5-1 and 4-1

| 5-1 | |
|---|---|
| Key of C 5th Degree: | Key of G 1st Degree: |
| G Major Chord | G Major Chord |
| [G, B, D, F] | [G, B, D, F#] |
| **4-1** | |
| Key of C 4th Degree: | Key of F 1st Degree: |
| F Major Chord | F Major Chord |
| [F, A, C, E] | [F, A, C, E] |

Having this mutual chord between keys makes the transition between the keys far more seamless, and although they both are named the same chord, they may not contain the same notes, as changing keys makes the scale differ by one extra note with each modulation.

Therefore in my program, I connect the 4th and 5th degrees to the 1st degrees in their respective keys with a probability transition, which now fully describes movement among all 84 nodes of the 12 possible keys.

**Automatic Mutation**

In order to continuously create varied music, I need to create a system which automatically changes its own parameters over time. The main problem to solve here is determining how often these mutations should happen.

I decided to encode this probabilistically, with the probability of a mutation occurring on every bar. However, I only want auto-mutation to occur on even numbered bars, as most western music groups phrases into bars of length powers of 2.

I also wanted the probability of the mutation occuring to initially be low, and then gradually rise the longer it's been without a mutation. I achieved this by creating a function which selected a random number from a growing pool of a range of integers, and compare this to a slower-growing value. I have found the parameters 4 and 1.5 to produce a good result:

```
if (getRndInteger(1,times*4) > times*1.5){ //do mutation stuff };
```

Where 'times' is the number of bars since the last mutation (1,2,3...) and getRndInteger(x,y) returns an integer in the range (x,y)

Once that condition has been satisfied, I have now chosen to mutate the next bar. There are independent probabilities to be set as parameters. These control the following:

- Tempo Increase
- Tempo Decrease
- Add Layer
- Remove Layer
- Modify Layer(s)
- Switch Key

Tempo, Layers, and Key switching are all independent of one another, and may all occur at once or only one at a time.

#### Tempo

The tempo will move by a certain range of beats per minute, either upwards or downwards. There is an acceptable lower and upper bound for the program to move the tempo to.

#### Key Modulation

This Key switching is seperate from the additional probability that the program will 5-1 or 4-1 modulate. There is a small chance for the program to modulate into a neighboring key on any bar. I think this varies the directions the music can move in a lot, and makes it less repetitive.

#### Layering

One layer at a time may be automatically added or removed. The new added layer will be initialized with a random polyrhythm. If the program chooses to modify existing polyrhythms, it will either re-roll a polyrhythm of the same length (same rhythmic family) or put in a brand new polyrhythm entirely.

The combination of my mutation engine, and the chord and key transition chains, provide for a vast degrees of freedoms, making it able to generate great lengths of unique music.

### 4.7 How I Connect Everything (The event Loop)

My entire program is controlled by a running event loop. This loop gets triggered once every 4 beats of music, and it determines what should be played in those 4 beats.

A conversion calculation is done to determine the length of time a bar is at any given tempo.

This calculation is as follows:

```
Time of a Bar = Eighth Note Time * 8
Eighth Note Time = Quarter Note Time / 2
Quarter Note Time = Beats Per Minute (Tempo) / Seconds per Minute (60)
```

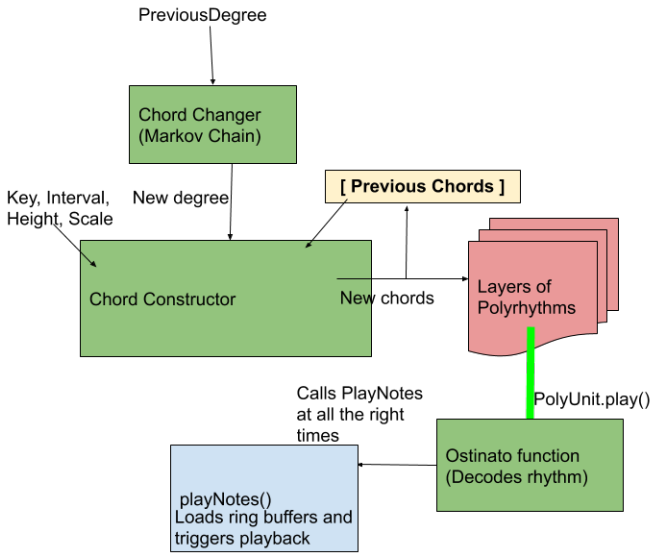


Figure 5: Structure Diagram for Music Generation

The event loop starts by taking the note degree from the previous bar, and using the Chord Changer to determine the next degree and key to play. These two, along with the chords from the previous bar, are used as input to my chord constructor to determine which notes to play next.

Each of the layers of the polyrhythms stack is assigned with one of these notes depending on which voice each layer is.

Each one of these layers have their .Play() function called, which in turn each call the Ostinato function, which decodes the rhythmic string in each polyrhyhm, and calls the function to play each note at the correct time denoted by the rhythmic string.

The event loop will then get Input from the UI, to determine if the user has requested any changes to the Graphics settings, Key, or Tempo, and use these as parameters for the next iteration of the event loop.

On even numbered bars, the automutation engine is run to determine (if) what should be changed in terms of layers, keys, tempo, or rhythm of the next upcoming bar. The end of the EventLoop function is a setTimeout to call the EventLoop function again at the time of the start of the next bar.

**Overview of the Event Loop:**

```
Generate Chords
For every Polyrhythm layer(unit) call PolyUnit.play()
Get Input from UI (graphics, key, tempo, state)
If barcount is even: {Run Automutate}
setTimeout(Eventloop(),timeofnextbar)
```

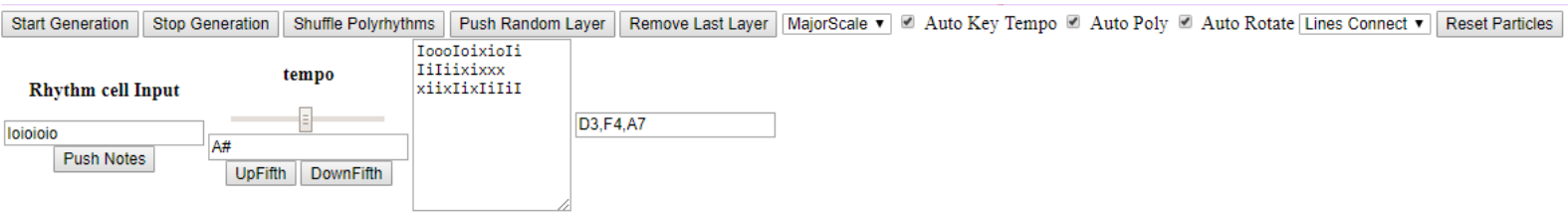


Figure 6: Final state of User Input

### User Input

As stated by my goals, my aim is to limit the amount of user input required to start generating good music, however also allow a high degree of flexibility for more advanced users. I would also like to have minimal inputs with powerful features to precisely manipulate music generation.

There are the Start and Stop buttons, followed by buttons to modify the layers. There are checkboxes for Polyrhythm or Key/Tempo Mutations, and one for Rotating the camera on the graphics. The Rhythm Cell Input allows users to input their own rhythms as layers. Users are also able to modify the tempo and key of the piece.

The Layers are shown on the right, with each rhythmic cell in a stack, and the correpsonding notes are displayed beside for each voice.

Basic users only need to interact with two buttons: Start Generation and Stop Generation. This alone is enough to see my program work at full potential. I find this highly effective, as there is essentially no barrier to entry to start listening to my program.

Intermediate users can control the Tempo, Scale, and Key of the music generated, as well as which graphical mode to visualize the music in. This allows the user to control the general direction of the music, without having the lowest level of control.

Advanced Users are able to encode their own rhythms as new layers, as well as adding or removing layers. Advanced users are able to disable the automation features of the program, granting them an undisturbed playground to manually alter the direction of the piece through key or tempo changes.

### 4.8 The Graphics Engine

The graphics are conceptually simple, yet effective. The graphics are built on a simple particle simulation model, where points can be drawn in a 3D volume. The simulation is done using WebGL to draw vertices using a vertex shader, and to connect and color these vertices using a fragment shader.

The particles are all stored in an array, which contains all rendered particles of the system. Each particle has the properties: X coordinate, Y coordinate, Z coordinate, A triple containing velocities in the X, Y, Z directions, and a Cr Cb Cg triple for color information.

Each particle also has a time-to-live property, which is initialized as a program parameter, and decremented by one every tick of the graphics simulation. Particles with a time to live less than or equal to 0 will be removed from the system. Since multiple particles may reach 0 time-to-live at the same time, I also have an array holding the indexes of all the particles waiting to be removed.

All this is running on a seperate graphics event loop, running at a much faster rate to the music event loop to update the graphics in real-time. With every iteration, every particle is updated by first updating its position with respect to its component velocities, and then each of the velocities is updated with respect to the gravitational strength and direction of the system. The time to live is then decremented by one.

When rhythm is decoded in the ostinato engine, a call is made every time 'I' or 'i' appear (a note is being played at this time), which pushes a new particle onto the particles list. The component velocities of the particle conform to a gaussian distribution with a mean of 0, and a deviance which is set as a program parameter. The position in which the particles is placed depends on the number of voices in the system.

When the function to place a particle is called every time a note is played, it is made known which voice in the polyrhythm stack is being played. I place the notes with respect to equally spaced voices around a circle. This is done using simple trigonometry.

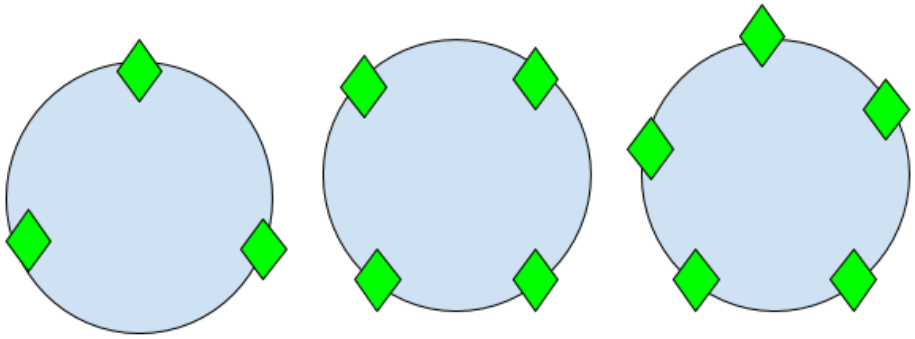

Figure 7: Showing the initial positions of the particles representing each voice in 3, 4, and 5 voices.

There is also deviance in the exact positions that the particles are emitted from, which follows a gaussian distribution, with deviance as a program parameter. This further enforces the separation of voices in the music, by separating where the particles from each voice emanate from. I find this to be a very useful visual aid to conceptually separate the different voices heard.

Another way that I use graphics to differentiate the voices is that each polyrhythm will have a set color. The color of that layer will only change if either the rhythm inside it is modified, or if the layer is removed, and a new layer put in its place. This helps with visual concurrency, as the colors remain consistent with each voice, only changing color when an audible difference takes place.

My program renders graphics in 4 different modes:

- POINTS emits one particle for every one note played.
- LINES_CONNECT emits one particle for every note played, and draws a line from the previous particle to the current particle.
- LINES emits two particles for every one note played, and connects these two points into a single line
- TRIANGLES emits three particles for every note played, and connects them together and fills in the resulting surface.

In these different modes, it is important to remove as many particles as you put in, so for example- the removal of particles in the TRIANGLES mode will have to remove dead particles in the system in groups of three. Not doing so may cause shapes lose their vertices one at the time, rather than have the whole shape disappear as expected.

# 5 Evaluation

## 5.1 Survey Results

The fruits of my program are measured subjectively, by different metrics of how the music makes the listener feel. Therefore the best way to evaluate this is a survey with selected examples from my program.

My survey consists of three questions, which gives me an indication of how well the music-generation goals of the project are achieved. Each Question was rated out of 5, with 5 being the most favorable.

The questions Were as following:

- Music sounded Good
- Music was Relaxing
- Music suited as a background

For the survey, participants are shown three different screen-recordings of my program at work. The program has been tweaked in each of these examples to try and measure the effects of certain techniques on fulfilling my musical goals. These screen recordings are available in Appendix B.

The first example shown is a version of my program which doesn't implement chord transition rules (Selects random scale degrees every bar), and does not adhere to voice transition rules (The voices jump about and move with no regard to the other voices). This example focuses on highlighting the effects of note selection.

The second example shown is a truly random selection of notes, with my program playing notes of any pitch, with no regards for chords or key. This is the control test, to give a baseline to compare the scores against.

And the third example shown is an excerpt from the submission state of the program, with all musical rules implemented. This will give an indication on how effective Chord Transition, and Voice Transition rules are at accomplishing my goals.

**Results and Analysis (n=9)**

| Example/Question | Music sounds Good | Music was Relaxing | Sits in Background |
|---|---|---|---|
| Example 1 | **3.00** | **2.78** | **2.44** |
| Example 2 | **1.33** | **1.78** | **1.78** |
| Example 3 | **4.22** | **4.44** | **4.11** |

Figure 8: Results from conducted survey

Example 2, the example with no constraints on the notes generated, scored the lowest as expected. Although fragments of music emerge from time to time through chance, it does not consistently produce pleasing music- as indicated by the low score of 1.33 for the first question. Continuity between bars here is lowest, which would explain the score of 1.78 in the background measure- The listener's focus is frequently broken due to jarring changes in the music. The effects of these jarring changes can also be seen in how relaxing the music is perceived to be- implying that jarring changes put listeners on edge, and make it difficult for them to relax.

In Example 1, the generation is less chaotic. Chords are generated using interval stacking, to create music that sounds good on a bar to bar basis- as seen by the higher score of 3.00 for aurally pleasing- however little is preserved between bars, which breaks continuity on scales larger than a single bar. This can be seen in the low score of 2.44 for the background criteria, which indicates that the listener still loses focus. The higher score of 2.78 for relaxation can be attributed to the fact that the notes played are all 'Technically Correct', all being within the Key and Scale that the bar is in- which can create a wash of musically-sounding noise that people find relaxing. The large (+1.67) increase in how 'good' the music sounds from Example 2 to Example 1 can be largely assumed to be an effect of having note selection rules, and not have notes selected purely at random with no regard to key or scale.

In the final example shown, we see that adding Voice and Chord transition rules increases Relaxation (+1.66) and Background (+1.67) scores from Example 1 quite a bit. However, how good the music is perceived to be does not rise as much (+1.22). This is supportive of my hypothesis that implementing voice leading and chord transition rules would make moving from bar to bar feel less jarring. This in turn makes the listeners more relaxed by minimizing the movements of each note, and less aware of changes over time, making it more suited for background listening. This data also suggests that note selection is the most important aspect of making the music sound 'good', however note selection alone will not be able to create a slow-evolving texture which would invoke relaxation and focus.

## 5.2 Evaluation on Features used to achieve goals

The first iteration of a working Event Loop generating music had the same chord generation features as the one seen in Example 1. At that point in time, building a fully working continuous stream of music was one of the largest milestones, after which I could begin to refine the generated music.

I first implemented the Chord Transition function. This restricted the degrees of the scale which could follow other degrees, which I tuned to the most widely known chord changes, which made the random disparate bars of music sound more cohesive in direction (relative to the aural traditions of most popular music).

I then implemented rules for voices, starting off with restricting voice movements from crossing the boundaries of other voices. The effect that this had on the overall music generated was insurmountable, as now each voice could be clearly distinguished from the others, maintaining identity even across many bars- allowing listeners to observe each voice on its own, and track its movement over time. Furthermore, by later minimizing movement of each voice to notes in nearby octaves, the movement of each voice is made less sporadic, making each chord change even less jarring, and less mentally obtrusive.

The final cherry on top was the auto-mutation engine. This helped break up the monotony that would arise from insufficient user-input controlled automation. By dynamically Adding/Removing layers and modifying the key and tempo, the overall sound is allowed to begin at a point, and gradually meander through a vast quantity of different musical landscapes as set by the tempo, scale, and rhythmic layers.

At the time, I believed these to be acceptable solutions to the problems presented by music generation, and looking back, while they may have produced acceptable results, with my newly acquired knowledge of musical generation techniques, I would have approached some of these problems differently.

The program currently builds music from the chord up, however, I believe more rich intricate results can be produced taking the opposite approach- By beginning with the voices, and building each of them up through interval interactions. Chords would merely be an emergent property of such a system. This method of generation would require a generation program built in a completely different way to my current method, as it would reason with each and every possible note interaction for every voice present- possibly even choosing to omit voices to adhere to some grander law of music theory; I believe a music generation system of that sort would not have been feasible within the time constraints of this project, and may require the use of neural networks.

Although my method isn't perfect, I am happy with the results produced by the abstractions created through my limited understanding of the full field of music composition, and I believe it is sufficient for a project of this scope and purpose.

## 5.3 Quality of coding implementation

Throughout the development of my project, I have refactored the entire codebase a couple of times. A big reason for this is that I set out to be able to develop on a live working prototype to continuously modify it for improvement. The downside to this is that I never truly got the full scope of what was best to do until the project was fully complete, and therefore my vision of where the project would go, as well as how that should be coded, would constantly be reevaluated. However, with a project with an end goal as general as mine, I believe that this may have been the best way to produce a solid outcome, as with more experimentation came more knowledge and insights that would lead me to change the vision of my project for the better.

Examples of previous visions include- A chord progression player, An interactive composition aid, and a musical playground with shape and color abstractions to guide the inexperienced user.

## 5.4 Challenges Faced

Javascript is frequently the butt of many programming jokes, and I got to experience why first hand.

In the beginning, I was relying on Javascript's implicit type conversion to handle all my variables. This frequently produced unexpected results and the infamous NaN. This was further exacerbated by the automatic global scoping done for default variable assignments, leading me to constantly be using variables from the wrong scope. Fortunately, not too long after beginning my implementation, I discovered that modern Javascript had an option to enable "strict mode" which would not only disable all implicit type conversions- allowing me to explicitly know the types of all my variables at any point in execution, but also would prevent variables from being declared unless they were already initialized in their current scope. This would greatly increase my chances of catching unexpected behavior, gradually edging the program closer towards the bug-free ideal over time.

Another big issue was that there were significant memory leaks in the early prototypes of my program. While explicit typing and variable initialization constraints alleviated some of this, My program would still grow from 5Mb of memory up to 30+Mb of memory late into its runtime. Perplexed by this, I began using the Chrome Memory debugger to locate what needed to be garbage collected. As it turns out, every instance of the Event Loop would be considered an entirely new scoping, which meant that variables that were re-assigned in the next iteration, still had copies left over in memory from all the previous iterations.

My solution to this would be to null every variable before reassigning. This reduced my memory footprint quite significantly, however it would still grow slowly over time. I dug deeper, and realized that the SetTimeouts powering the event loops themselves were being constantly stored duplicated in memory. The solution to this would be to write the SetTimeout function as an object. At the end of every iteration, the setTimeout would be called for the next bar, and at the beginning of the next bar, this variable holding the setTimeout would be nulled, marking it for the garbage collector. The result is that the program now only takes up about 3Mb of memory at most, and doesn't grow over time.

Another big Issue that started occurring out of nowhere was after an indeterminate amount of time of my program running, the audio would suddenly become extremely distorted. At first I thought it was a memory leakage causing this, which led me to doing all the prior memory debugging, however, after fixing all of the memory leaks, the problem still persisted and did not get any better! I noticed that the problem disappeared every time I finished collecting memory snapshots for debugging, which led me to discover the manual garbage collection feature only in Chrome, my default browser, which would tell the engine to go collect all the garbage waiting to be collected, and clicking this little rubbish-bin icon would magically return my program to a functioning state.

This was very puzzling, and after some digging, it turns out that an update to Chrome that was launched mid-way through my development contained a bug that prevented the garbage collector from automatically cleaning memory that it is marked for removal. This breathed new hope into me, having trying to debug this specific issue for weeks. From that point on, I would test my program on Firefox, and even Microsoft Edge! Both of which would not produce this issue. I am still unsure of the specifics of why my program does not function correctly in Google Chrome, and further digging is required, however I would from that point on begin running and testing in Mozilla Firefox.

### Personal Evaluation of Outcome- Do I like the music?

Towards the end of my development of the program, I would actually leave the program on and listen to its music whilst debugging. The music produced was calming, soothing, and did not draw my attention away from my work. In this regard, I find my program to be successful in creating a good atmosphere for work and focus. From time to time, paying attention to the music generated would surprise me, as beautiful intricate moments would arise through sheer chance, and they would not be too far and few in between. I would say that I enjoy the music, and not merely because its grown on me through the countless of hours of listening to the same Sine and Triangle waves crashing on my ears, but also because I genuinely enjoy some of the music produced by the program, and am proud of the overall result.

# 6 Reflection

## 6.1 Lessons Learned

I learned about memory management in Javascript the hard way, and I believe a formal comprehensive study on browser memory management before starting implementation would have saved me on debugging time. However, through all the memory debugging I had to do, I learned hands-on how to trace the source of memory leaks in a large connected system, and how to greatly minimize those leaks to produce a lightweight, responsive, and reliable Javascript web application.

I also learned that testing on multiple environments is important for debugging issues, and that I should not blindly assume homogeneity across all possible runtime environments.

Working on a large system with many interconnected parts has strengthened my discipline in being explicit with scoping, and has taught me to write cleaner code. This large interconnectivity also gave me useful experience in unit testing each of the components that build up to form my project, as well as building functions and systems that conform to specifications that allow it to flawlessly integrate with the greater system.

I have also increased the scope of my music theory knowledge, and thought a lot about how music can be represented, and reasoned with in code.

## 6.2 Conclusion

Overall I would say my design and implementation was largely successful in achieving all the goals set out in the beginning, however the journey has led me to see many ways in which the end result could be improved.

And last but not least, I had a lot of fun working on this project, and faced many interesting problems along the way. Once again I would like to thank my Project Supervisor Toby Howard for not only hosting such a fun project topic, but also for his guidance, feedback, and unwavering support.

# 7 Appendix A- A brief introduction to music theory

In Music, there are 12 equivalence classes of pitches, denoted by these names given to them:

C – C# – D – D# – E – F – F# – G – G# – A – A# – B

These pitches are cyclic, differentiated by different octaves. For every note, we have the octaves from 0 and upwards. C0 is a wave vibrating at 16.352hz, C1 is double that frequency, C2 is double the frequency of C1, and so on. The way of denoting pitch in this way is referred to as Scientific Pitch Notation.

We can group the notes into what we call 'Key Signatures', or 'Key' for short. Each Key will contain 7 of the above notes. An example is the Key of G, which consists of - G A B C D E F#

We can select subsets of these pitches to create what we call a 'chord'; each chord invoking a different auditory feeling. A typical chord may look something like this: [G4, C5, E6, B6]

We can imagine this to be analogous to a 'color' in the visual arts. Moving or adding notes to a chord is analogous to mixing more colors in to a paint, it will create different shades of color, and mixing colors that don't go together makes an unpleasant dark amalgamation of color. Similarly, in music, adding too many notes into a chord will muddy the intent of the chord, making it sound chaotic and random. The rules of Harmony describe these interactions.

Notes in a chord need not be played at the exact same time: The effect from combining multiple notes together still persists so long as the listener still has the notes in their immediate present short term memory (roughly 3 seconds in humans). Rhythm denotes when a note is played. A standard unit length of time is denoted by the bar, which is the time taken for 4 beats to elapse with respect to the tempo (given in beats per second).

This bar can be subdivided further, into any number of equal length parts. A bar is most commonly divided into powers of 2 such as 16, or 32, however, other division sizes, such as 20 (4 beats of 5) or 28 (4 beats of 7), are often used in less mainstream music.

# 8 Appendix B- Resources

- Survey Video Examples- youtube.com/playlist?list=PL8LB87utRAzbO7kfQav5pBZGOSSVkq-SE

- The project hosted as a webpage- http://genmumsic.byethost11.com

### Rhythm Decoding Pseudocode

```
function ostinato(rhythm, notearray, tempo, startTime, type, playingvoice) {
        var time = startTime;
        let state = "";
        for (let i = 0; i < rhythm.length; i++) { //loops over each character in
            rhythm
                let lookahead = 1;
                let durationadded = 0;
                //handles adding length to held notes
                if ((rhythm[i] == "i" || rhythm[i] == "I") && rhythm[i + 1]
                    == "o") {
                        durationadded += 1;
                        lookahead += 1;

                        while (rhythm[i + lookahead] == "o") {
                                lookahead += 1;
                                durationadded += 1;
                        } //accumulates time for held notes
                }
                state = rhythm[i];
                switch (state) {
                        case "I": //play loudly
                                setTimeout(locationEmit, 1000 * ((time + i *
                                    eighthNoteTime) - context.
                                    currentTime)); //Graphics Emit

                                playNotes(notearray, OstOsc, Gains,
                                    EQArray, time + i * eighthNoteTime,
                                    eighthNoteTime, (durationadded), 1,
                                    type, context);
                                break;
                        case "i": //play softly
                                setTimeout(locationEmit, 1000 * ((time + i *
                                    eighthNoteTime) - context.
                                    currentTime));
                                playNotes(notearray, OstOsc, Gains,
                                    EQArray, time + i * eighthNoteTime,
                                    eighthNoteTime, (durationadded), 0,
                                    type, context);
                                break;
                        case "x": break;
                        case "o": break;
                }
        }
}
```

### Chord Generation Pseudocode

```
function chord_constructor(key, currentdegree, interval, number, scale)
{
        let chordtones = [];
        let notes = getScale(key, scale)
        let currdegree = currentdegree;
        \\Returns the 7 Notes in the given Key/Scale;
         for (var i = 0; i < number; i++) {
                if (currdegree == 0) {
                        currdegree += 7;
                }
                chordtones.push(notes[currdegree − 1]);
                currdegree = (currdegree + (interval − 1)) % notes.length
        }
        return chordtones;
}
```

### Auto-Mutate Pseudocode

```
function automutate(times)
{
        tempo−polarity = Math.random() < 0.5 ? −1 : 1;
        change−tempo = Math.random() < 0.3 ? 1 : 0;
        keychange−polarity = Math.random() < 0.5 ? −1 : 1;
        is−keychange = Math.random() < 0.3 ? 1 : 0;
        add−or−remove−poly = Math.random() < 0.5 ? 1 : 0;
        tempo−polarity = Math.random() < 0.5 ? −1 : 1;

        var mutate−seed = getRndInteger(1,times∗4)

        if (mutseed > times∗1.5)
                {
                        //use all the generated probabilities to alter track
                }
        else {return times+1;}
}

\\IN THE MAIN PART OF THE PROGRAM IN THE EVENT LOOP:

if (automutate−switch) //if automutate is on
{
        mutatenumber = automutate(mutatenumber); //use value
            accumulated every round
}
```

# 9 References

## References


1. Brian Eno - Music for Airports https://youtu.be/vNwYtllyt3Q

   1978 , Referenced November 2018

2. JOHN T. LYSAKER - Music for Airports 40 years later https://blog.oup.com/2018/12/brian-eno-music-for-airports/

   2018, Referenced March 2019

3. Christopher Lloyd Clarke - Psychoacoustics and the science of relaxation music

   https://enlightenedaudio.com/psychoacoustics-and-science-relaxation-music

   Referenced March 2019

4. Steve Reich - Electric Counterpoint Movement 3 (Fast) https://youtu.be/_TKVpUSWCug

   1987, Referenced November 2018

5. Russell Meyers

   http://music.meyers.me.uk/home/20th-century-music/minimalism

   1999, Referenced March 2019

6. Dr. Barbara Murphy University of Tennessee School of Music

   Four Part Writing

   https://music.utk.edu/theorycomp/courses/murphy/documents/PartWritingRules.pdf

   Referenced April 2019