UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

PEDRO HENRIQUE AUGUSTIN

Aplicação Web para compor músicas no estilo Chiptune utilizando Composição Algorítmica e Generativa

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia da Computação

Orientador: Prof. Dr. Marcelo de Oliveira Johann

Porto Alegre
2022

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos André Bulhões
Vice-Reitora: Profª. Patricia Pranke
Pró-Reitora de Graduação: Profª. Címtia Inês Boll
Diretora do Instituto de Informática: Profª. Carla Maria Dal Sasso Freitas
Coordenador do Curso de Engenharia de Computação: Prof. Walter Fetter Lages
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

"If you hit a wrong note,

then make it right by what you play afterwards."

— Joe Pass

AGRADECIMENTOS

Agradeço aos meus pais por sempre me apoiarem em minhas decisões, por sempre estarem presentes em todos os momentos e por me incentivarem a criatividade. À toda minha família pelo amor e acolhimento. Agradeço a todos os meus amigos pelo suporte e pelos maravilhosos anos de amizade que se passaram e pelos que estão por vir. À Julia por todo amor e risadas e por sempre ter me apoiado e me motivado a seguir em frente.

Agradeço aos meus colegas com quem pude compartilhar esta jornada. Aos professores Marcelo de Oliveira Johann , Marcelo Pimenta e Rodrigo Schramm por me mostrarem que a música e a computação podem andar juntas e que não existe um limite para a criatividade.

RESUMO

A composição musical varia muito de acordo com diferentes contextos culturais e sociais. Grande parte das músicas que ouvimos no nosso dia-dia seguem um conjunto de formalismos e são compostas por diferentes partes que incluem sons, ritmos, melodias e repetições. O processo de compor uma música pode ser visto como um modo de combinar essas partes, seguindo um conjunto de "regras", para gerar um resultado musical. Um dos meios de automatizar esse processo e formalizar suas "regras"é através da composição algorítmica. Um algoritmo pode ser entendido como um conjunto de instruções predefinidas que visam resolver um problema em um tempo finito. Logo, a composição algorítmica pode ser entendida como o processo de fazer música através de processos formais predefinidos. A composição automatizada muitas vezes é realizada de forma estocástica, ou seja, onde a chance dos eventos ocorrerem são completamente aleatórias, como o Jogo de Dados de Mozart. Hoje em dia a tecnologia e a grande capacidade de processamento dos computadores abrem muitos novos caminhos para explorar os antigos e novos métodos de composição algorítmica. Este trabalho propõe o encontro de sons antigos de videogame com recursos e ferramentas atuais para criar uma aplicação web que utilize a composição algorítmica e a Web Audio API com o objetivo de gerar músicas no estilo *Chiptune* que possam ser utilizadas em qualquer computador com acesso a internet.
Palavras-chave: Composição algorítmica. chiptune. web audio. videogame.

Web Application for composing Chiptune music using Algorithmic Composition

ABSTRACT

.

Musical composition varies deeply according to cultural and social contexts. Most of today's music follow some kind of formalism and are composed by different parts such as sounds, rythms, melodies and repetition. The process of making a music consists of assembling these parts together following a set of "rules" to generate a piece of music. This process can be automatized in many ways, one of them beeing the algorithmic composition. An algorithm is a finite set of instructions used to solve a specific problem in a finite amount of time. So, algorithmic composition can be understood as the process of using some formal process to make music. The process of putting the pieces together can be completley random like the famous Dice Game from Mozart. The current technology allow us to explore new ways of creating musical pieces with the aid of computers using algorithmic composition. The main goal of this work is to bring together the old sound of videogames and the new technology to build a web application that uses web audio and algorithmic composition to compose ever changing Chiptune music.

Keywords: Algorithmic Composition, chiptune, web audio, videogame.

LISTA DE ABREVIATURAS E SIGLAS

PSG	Programmable Sound Generator
AC	Autômato Celular
SID	Sound Interface Device
ADSR	Attack/Decay/Sustain/Release
MIDI	Musical Instrument Digital Interface
BPM	Batidas por minuto
JSON	JavaScript Object Notation
DAW	Digital Audio Workstation
SPA	Single Page Application
API	Application Programming Interface

LISTA DE FIGURAS

Figura 2.1 Diagrama representando o exemplo 16
Figura 2.2 Diagramas de ritmos euclidianos 17
Figura 2.3 Automato celular elementar que utiliza a regra de número 30 18
Figura 2.4 Os principais formatos de onda gerados pelos PSGs 19
Figura 2.5 Contexto de áudio 21

Figura 3.1 Tela inicial do WolframTones 22

Figura 4.1 Fluxograma do desenvolvimento 24
Figura 4.2 Exemplo de um compasso com notas 26
Figura 4.3 Ciclo de quintas 28
Figura 4.4 The Ultimate Soundtracker 29
Figura 4.5 Resultado do método Euclidiano 31
Figura 4.6 Dez Gerações geradas pelo Autômato Celular 31
Figura 4.7 Interface visual da aplicação 37

Figura 5.1 Aplicação rodando 38
Figura 5.2 Opções de customização 38

LISTA DE TABELAS

Tabela 5.1 Tabela detalhando as funções extras 39

SUMÁRIO

1 INTRODUÇÃO **11**
2 CONTEXTO **13**
2.1 Composição Automatizada **13**
2.1.1 Composição Estocástica 14
2.1.2 Composição Algorítmica 14
2.1.3 Ritmos Euclidianos 15
2.1.4 Autômatos Celulares 17
2.2 Chiptune **18**
2.3 Web Audio API **20**
3 TRABALHOS RELACIONADOS **22**
3.0.1 WolframTones 22
3.0.2 Generative.fm 23
4 PROJETO E IMPLEMENTAÇÃO **24**
4.1 Visão Geral do Desenvolvimento **24**
4.2 Objetivo **25**
4.3 Escolha dos Métodos **25**
4.3.1 Representação 25
4.3.2 Composição 27
4.3.3 Reprodução e Visualização 28
4.4 Escolha das Ferramentas **29**
4.4.1 Tone.js 30
4.4.2 Total Serialism 30
4.4.3 Vue.js 32
4.5 Implementação **32**
4.5.1 Sintetizadores 32
4.5.2 Composição 34
4.5.2.1 Baixo 35
4.5.2.2 Arpeggiator 35
4.5.2.3 Melodia 1 e Melodia 2 36
4.5.2.4 Percussão 36
4.5.3 Interface Visual 36
4.5.4 Hospedagem 37
5 DEMONSTRAÇÃO DE USO **38**
6 CONCLUSÃO E TRABALHOS FUTUROS **40**
REFERÊNCIAS **41**

1 INTRODUÇÃO

Uma música é composta por vários elementos como melodia, harmonia, ritmo e repetições. As repetições são agradáveis aos nossos ouvidos porém costumam ser irritantes a partir de um ponto. Muitas vezes em trilhas sonoras de jogos temos um tema que repete em um *loop* constante. Essa repetição de uma trilha sonora é potencialmente usada para preencher espaços auditórios vazios, porém causa uma grande diminuição na tensão e no aspecto surpresa do ouvinte. Uma técnica para contornar essa perda de atenção é o uso de música adaptativa, ou seja, que muda constantemente com a interação do jogador. Outro modo é o uso de música generativa ou música algorítmica. A música algorítmica é aquela que é criada através de uma automatização. No caso dos jogos ela pode prover uma música única e infinita que se adapta em um nível muito maior do que a música composta adaptativa. Apesar do seus potenciais benefícios, a música algorítmica ainda não é muito utilizada nos video games (PLUT; PASQUIER, 2019).

A composição automatizada refere-se ao uso de um processo formal para criar músicas com a menor intervenção humana possível. Isso geralmente é realizado em um computador com a ajuda de vários formalismos, como geração de números aleatórios, sistemas baseados em regras e vários outros algoritmos (ALPERN, 1995). A ideia de algoritmo já existe há muitos anos, com evidências de uso por diferentes culturas antigamente. O uso de formalismos para resolver problemas tem sua origem atrelada a álgebra e matemática. Pitágoras de Samos, na Grécia Antiga, já acreditava que os sons expressos na música estavam diretamente relacionados com as leis da natureza e os números. Estes formalismos e regras para compor música existem há muito tempo, porém o termo *composição algorítmica* é relativamente novo.

Muitas técnicas de composição automatizada foram criadas e evoluídas juntamente com a tecnologia para criar músicas. Atualmente as regras e métodos, antes definidas e executadas por seres humanos, podem ser definidas e executadas por um computador. A composição de músicas realizada por um computador já tinha sido pensado por Ada Lovelace no século 19. Segundo ela, se as relações fundamentais entre sons e frequências nos termos de harmonia e composição musical estivessem sujeitas a expressões e adaptações, a máquina poderia compor peças musicais científicas e elaboradas de qualquer grau de complexidade e extensão (ALPERN, 1995).

A técnica de composição algorítmica apresentada neste trabalho segue um padrão estocástico, onde todas as escolhas são feitas a partir da chance, porém, em vez de ser

interpretada por seres humanos, a composição é interpretada pelo próprio computador. A composição e a reprodução das músicas foram baseadas no estilo *Chiptune*, que remete ao som dos primeiros consoles de *video game* e aos primeiros sons reproduzidos por um computador.

Alguns dos jogos que utilizam temas em *Chiptune* sofrem do problema de repetição do mesmo trecho sonoro. O objetivo deste trabalho é contornar este problema criando uma aplicação que crie trilhas sonoras únicas e infinitas utilizando composição algorítmica. A aplicação foi feita na *web* para que possa rodar em qualquer computador com acesso a internet. O intuito é ser executada no *background* enquanto o usuário joga algum jogo ou realiza qualquer outra atividade.

Este trabalho é organizado da seguinte maneira: O capítulo 2 contextualiza a Composição Algorítmica, o *Chiptune* e o *WebAudio*, que são os conceitos principais para o entendimento do trabalho. O capítulo 3 revisa alguns dos trabalhos já existentes na área que utilizam composição automatizada na *web*. O capítulo 4 detalha a construção da aplicação e suas funcionalidades. O capítulo 5 discorre sobre o uso da aplicação e o capítulo 6 aponta as conclusões e os trabalhos futuros.

2 CONTEXTO

2.1 Composição Automatizada

Os primeiros registros de composição musical auxiliada por um computador datam nos anos 50. Um dos primeiros resultados pode ser ouvido em *Illiac Suite* (1957) onde Lejaren Hiller e Leonard Isaacson, da universidade de Illinois, utilizaram o computador Illiac para compor a peça musical. O computador compôs a partitura da música que mais tarde foi interpretada por um quarteto de cordas. Este mesmo processo foi utilizado mais tarde, no final dos anos 50, por Hiller e Robert Baker no MUSICOMP, um dos primeiros computadores para composição automatizada. O MUSICOMP foi feito como uma biblioteca de sub rotinas que poderiam ser utilizadas pelo usuário e facilitavam a composição. A ideia de implementar pequenas funções bem definidas se provou eficiente e permitiu que o sistema tivesse um grau de flexibilidade e de generalização (ALPERN, 1995). Mais tarde, nos anos 60, Iannis Xenakis criou um programa que produzia dados que eram usados para compor música de maneira estocástica. Xenakis utilizou o computador para deduzir uma composição a partir de notas com densidades e pesos probabilísticos diferentes. Assim como no caso do Illiac, a composição foi criada pelo computador mas foi interpretada por seres humanos.

Nos anos seguintes surgiram outros trabalhos notáveis como o computador dedicado de Olson (1961), que conseguia compor novas melodias a partir de melodias previamente alimentadas utilizando cadeias de Markov, e também o trabalho de Gill's (1963) relacionado a IA, que utilizava uma busca hierárquica com *backtracking* para guiar um processo composicional. A medida em que os computadores foram ficando mais baratos e com poder de processamento maior, a composição algorítmica foi lentamente ganhando conhecimento e ficando mais popular. (FERNáNDEZ; VICO, 2013)

O termo *composição automatizada*, as vezes também chamado de *som generativo*, engloba muitos outros conceitos que incluem composição algorítmica, procedural, IA, além de outras. Como estes são termos abstratos, muitas vezes eles podem significar a mesma coisa. Existem diferentes métodos dentro dessas áreas que podem ser aplicados a um programa para criar uma composição automaticamente. Os primeiros trabalhos na área utilizavam uma metodologia estocástica, que envolvem a aleatoriedade, enquanto trabalhos mais atuais já buscam utilizar a inteligência artificial para compor novas peças.

2.1.1 Composição Estocástica

Este tipo de composição está profundamente conectado com processos matemáticos e probabilísticos. Os sistemas estocásticos utilizam dados aleatórios e caóticos para produzir um resultado. Eles podem filtrar dados aleatórios para obter alguma ordem através de regras de estatística ou de distribuição (FARNELL, 2007). Seria como tirar notas aleatórias de dentro de um chapéu e juntá-las para formar uma composição. Por isso, este método é um dos mais simples de se aplicar.

As distribuições de dados são normalmente nomeadas de acordo com padrões de teoria estatística como uniforme, linear, exponencial e Gaussiano. Cada uma delas tem uma aplicação particular na composição, seja determinando a duração das notas, na densidade melódica ou na própria síntese do som (FARNELL, 2007). Um sistema estocástico pode ser tanto generativo como interativo, com ações e dados de entrada determinados pelo usuário.

2.1.2 Composição Algorítmica

A composição algorítmica normalmente se refere ao processo que evolui de acordo com um simples conjunto de regras fáceis de entender. Um algoritmo é definido como um conjunto de regras para resolver um problema e um número finito de passos (FARNELL, 2007). O que importa neste tipo de composição são as regras definidas previamente e os resultados parciais que elas vão gerar. Os resultados parciais são mais importantes do que os finais, visto que estes tipos de sistemas são feitos para rodar pelo maior tempo possível.

Existem inúmeras regras que podem ser aplicadas para que um sistema algorítmico seja capaz de criar uma composição. Dependendo da linguagem usada e do programa que a irá interpretar, algumas poucas linhas de código ou poucos caracteres já podem definir várias horas de som generativo. Assim como a síntese de um som, um sequenciador algorítmico utiliza equações para produzir funções no tempo mas, diferentemente de ondas sonoras, elas são raramente periódicas. As funções são feitas para seguir padrões que podem ser melodias ou harmonias musicalmente agradáveis. Os algoritmos utilizam a complexidade, similaridade e um pouco de periodicidade para produzir padrões com um certo grau de ordem (FARNELL, 2007). Muitos desses sistemas utilizam equações matemáticas para definir suas regras e garantir a generatividade como algoritmos euclidianos, sequencias de Fibonacci e autômatos celulares.

2.1.3 Ritmos Euclidianos

Um dos modos de representar ritmos na música é com o uso de sequências binárias, onde cada bit é considerado uma unidade de tempo e possui dois estados: o bit zero que representa o silêncio e o bit um que representa uma nota. TOUSSAINT propôs que um dos meios de encontrar ritmos de diferentes partes do mundo e representá-los é através do uso do algoritmo euclidiano.

O algoritmo euclidiano é um dos mais antigos algoritmos conhecidos. Ele é utilizado para computar o maior divisor comum entre dois números inteiros. A ideia é bem simples. O menor número é subtraído do maior até o maior ser zero, ou até o maior ser menor do que o menor, que nesse caso é chamado de restante. Este restante então é repetidamente subtraído do menor número até obter um novo restante. Este processo continua até o o restante ser zero. Então o maior divisor comum é o último restante antes do zero (TOUSSAINT, 2005). O pseudo algoritmo a seguir exemplifica a função recursiva com os parâmetros x e y onde x > y.

```
1. euclidean(x, y)
2.     if y = 0
3.     then return x
4.     else return euclidean(y,x mod y)
```

Agora imagine um problema onde é preciso construir uma sequencia binária de *n* bits com *k* bits um, onde esses *k* bits um precisam estar distribuídos o mais uniformemente possível. A solução é simples se *k* é divisor de *n*. Por exemplo, se *n* for 8 e *k* for 4 então a sequencia resultante é [1,0,1,0,1,0,1,0]. Porém, o problema fica um pouco mais complicado quando os números não são divisores um do outro.

Uma das soluções para resolver este problema foi proposta por Bjorklund, enquanto estudava problemas de física nuclear. Um dos exemplos de Bjorklund, citado por TOUSSAINT, consiste no seguinte: imagine uma sequência onde temos um vetor de tamanho *n* com *k* bits um. Nós começamos considerando uma sequência de 5 bits um seguidos por 8 bits zero:

[1 1 1 1 1 0 0 0 0 0 0 0 0]

Nós começamos movendo os zeros, posicionado um zero após cada um. Após isso ficamos com cinco sequências de dois bits com três zeros sobrando.

[10] [10] [10] [10] [10] [0] [0] [0]

Após isso, nós distribuímos os zeros restantes da mesma maneira, posicionando cada [0] após cada [10] e ficamos com o seguinte resultado.

[100] [100] [100] [10] [10]

Então nós ficamos com três sequências de 3 bits cada e um restante de duas sequências com 2 bits cada. O mesmo processo é aplicado e agora ficamos com o resultado.

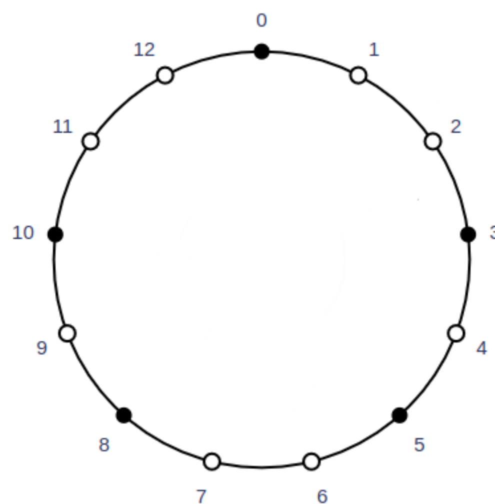
[10010] [10010] [100]

O processo termina quando o restante consiste de apenas uma sequência, ou quando não temos mais zeros. A concatenação das últimas sequências representa a sequência final.

[1 0 0 1 0 1 0 0 1 0 1 0 0]

Esta sequência também pode ser visualizada em um diagrama em forma de polígono como na Figura 2.1, onde o zero denota o início e o tempo é contado no sentido horário. Dois ritmos são diferentes se a sequência de zeros e um difere de um para o outro, começando do primeiro bit, em qualquer representação (DEMAINE et al., 2009).

Figura 2.1 – Diagrama representando o exemplo

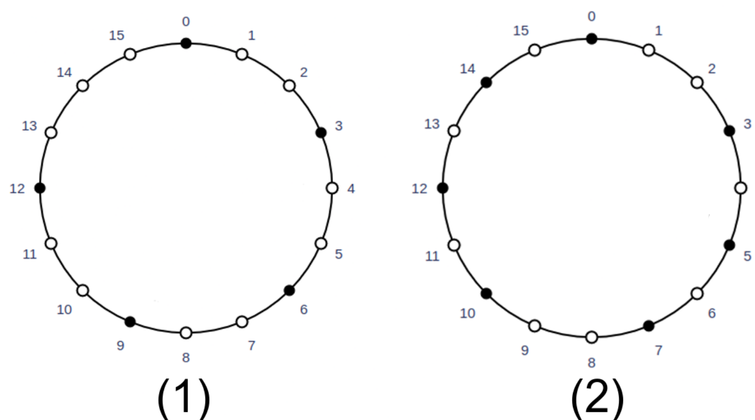


Fonte: O Autor

O método de Bjorklund e o algoritmo euclidiano seguem a mesma estrutura, e por isso TOUSSAINT nomeou estas sequências de *Ritmos Euclidianos*, que podem ser representados como *E(k,n)*, onde *k* é o número de bits um e *n* é o tamanho total da sequência.

Este método nos permite encontrar diversos ritmos diferentes mudando apenas dois parâmetros, como visto na Figura 2.2. O diagrama (1) representa o ritmo *E(5,16)* que é o ritmo da Bossa Nova, enquanto o diagrama (2) representa o ritmo *E(7,16)* que é o ritmo do Samba, de acordo com TOUSSAINT.

Figura 2.2 – Diagramas de ritmos euclidianos



.

Fonte: O Autor

2.1.4 Autômatos Celulares

Autômatos Celulares são idealizações matemáticas de sistemas naturais. Eles são representados como uma coleção de células, dispostas em uma grade de certo formato, que evoluem através de uma série de passos discretos de acordo com um conjunto de regras, que são baseadas nos estados das células vizinhas. As regras são aplicadas de forma iterativa por quantas vezes precisar (WOLFRAM, 1994).

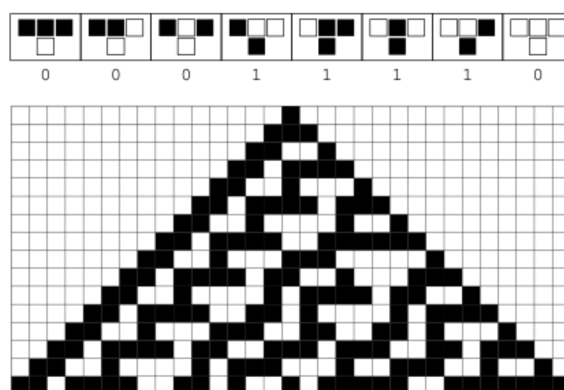
Um autômato pode ter várias formas. O formato da grade pode ser uma simples linha unidimensional como também pode ser formas bidimensionais tais como quadrados ou triângulos, entre outras formas. Dependendo do autômato, uma célula pode também assumir vários estados. No autômato mais básico uma célula pode assumir apenas dois estados (podemos pensar como 0 ou 1).

O autômato mais básico é chamado de *automato celular elementar*. Nele as células são dispostas em uma grade unidimensional e só podem assumir dois estados, 1 ou 0. A regra para determinar o estado da célula na próxima geração depende apenas da própria célula e seus dois vizinhos. Existem 2^3 possíveis configurações para os estados de uma célula e seus dois vizinhos. Sabemos que a célula da próxima geração só pode assumir

dois estados, logo, existem 2^{23} (256) regras que um autômato elementar pode seguir.

WOLFRAM propôs um sistema para numerar estas regras. O sistema consiste em escrever, para cada configuração possível (111, 110, 101...), o estado resultante. Esse resultado é então interpretado como uma representação binária de um número entre 0 e 255. Por exemplo, a Figura 2.3 representa a evolução de acordo com a regra de número 30 (00011110).

Figura 2.3 – Automato celular elementar que utiliza a regra de número 30



Fonte: Mathworld

Existem diversas maneiras de empregar autômatos celulares na música. É possível utilizá-los tanto na síntese sonora, como é o caso do sintetizador granular Chaosynth (MIRANDA, 2001), quanto na própria composição musical, como o CAMUS (MIRANDA, 2001). Uma forma básica de utilizar um autômato é interpretando-o como uma simples sequência de notas, onde o bit 1 significa uma nota sendo tocada e o bit 0 significa o silêncio. Com isso podemos criar diversas sequências de notas diferentes, que vão evoluindo por qualquer quantidade de tempo, de acordo com qualquer regra do autômato elementar.

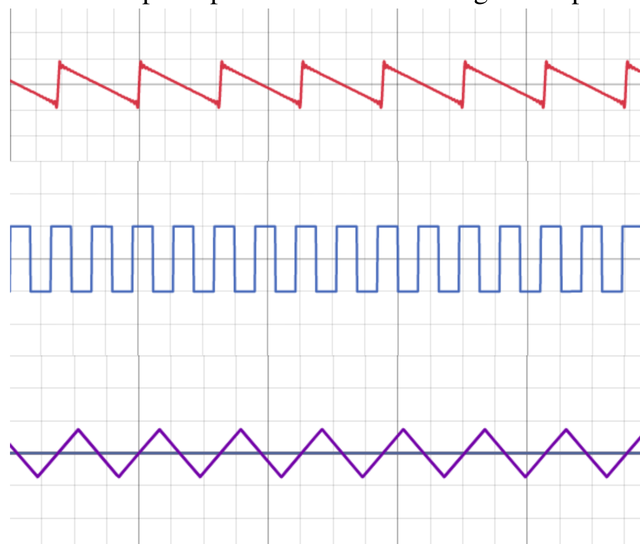
2.2 Chiptune

A criação e reprodução de trilhas sonoras de *video games* em consoles das décadas de 1970 e 1980 foi limitada pela tecnologia da época. Essa limitação fez com que as músicas e os sons ganhassem uma característica única e permitiu que trilhas sonoras criadas 40 anos atrás ainda estejam gravadas na memória de muitas pessoas nos dias de hoje. Atualmente estes sons, por mais antigos que sejam, ainda são relevantes e estão presentes em diversas músicas e trilhas sonoras para *video games* sob o título de *Chiptune* ou *música 8-bit*. O termo *Chiptune* refere-se à música que é composta utilizando hardwa-

res de computadores e consoles de jogos antigos baseados em *microchips* (DRISCOLL; DIAZ, 2009), mas atualmente não depende necessariamente desse hardware antigo para criação das músicas, visto que a tecnologia atual permite emular ou *samplear* estes sons.

As primeiras máquinas 8-bit, incluindo algumas máquinas de *pinball*, utilizavam *programmable sound generators* (PSGs) para gerar seus sons. Os PSGs eram chips de síntese subtrativa que ofereciam pouco controle sobre o timbre e eram em maioria restritos a forma de onda quadrada (COLLINS, 2007). Devido às limitações, os PSGs, normalmente, só conseguiam gerar formas de onda básicas, que podem ser vistos na Figura 2.4. Os formatos de onda gerados eram as ondas quadradas, com harmônicas ímpares, ondas triangulares (que também contém harmônicas ímpares, porém é mais suaves que a onda quadrada) e ondas dente de serra (que tem um timbre mais áspero e contém harmônicos ímpares e pares). Estes formatos de onda dão ao som um timbre "eletrônico", e junto de algumas técnicas como *arpeggios* e *loops*, definem o gênero *Chiptune*.

Figura 2.4 – Os principais formatos de onda gerados pelos PSGs



Fonte: O Autor

Um dos chips mais famosos foi o do console Commodore 64. O chip, fabricado pela *MOS Technology*, foi apelidado de SID (*Sound Interface Device*) e permitia três vozes, cada uma com seu próprio oscilador. Cada oscilador podia reproduzir uma forma de onda diferente: quadrada, triangular, dente de serra, pulso e *noise*. Além disso, o chip contava com envelopes ADSR para as vozes e um filtro que ajudava a reproduzir vários timbres diferentes. Todas essas características fizeram com que o SID tivesse um papel muito importante na história e na definição do gênero de músicas de *video game*.

Apesar do *Chiptune* ter suas origens em uma tecnologia de anos atrás, muitos artistas e desenvolvedores de jogos ainda utilizam esse estilo para compor suas peças.

Artistas como *Bit Shifter*, *Adhesive Wombat*, *Crystal Castles* e muitos outros incorporam elementos do *Chiptune* em suas músicas, e jogos lançados nos últimos anos que fizeram sucesso como *Shovel Knight* (2014), *Undertale* (2015) e *Celeste* (2018), têm toda sua trilha sonora, ou possuem vários trechos dela, em *Chiptune*.

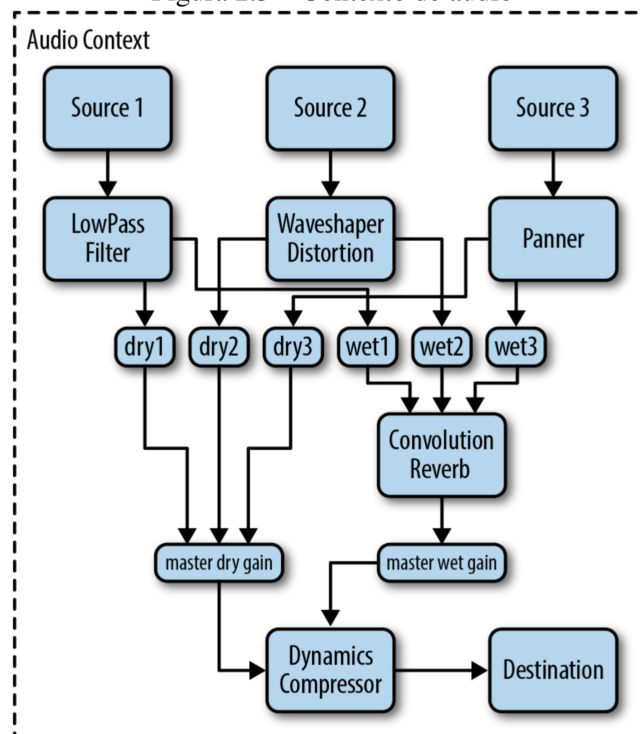
2.3 Web Audio API

No início da web, os métodos de inserção de áudio em sites e aplicativos eram escassos. O primeiro meio criado para esse viés foi a *tag* *<bgsound>*, proveniente da linguagem HTML, que permitia que os *plugins* de áudio reproduzissem sons de fundo nos sites, assim que um usuário o visitasse. Essa ferramenta possuía controles básicos sobre o áudio inserido, permitindo que alguns atributos, como volume, fossem controlados por meio de código pelo criador do domínio. Atualmente, a *tag* *<bgsound>* foi descontinuada e substituída pela *tag* *<audio>*, que possui mais compatibilidade com navegadores e não utiliza *plugins* adicionais para reprodução de áudio.

Apesar dessa substituição, a reprodução de áudio por meio de *tags* HTML ainda possui algumas limitações significativas para aplicações mais acuradas, como afirma SMUS ao listar algumas dessas limitações no seu livro, tais como: a falta de controle de *timings* precisos, limites baixos para a quantidade de sons que podem ser tocados ao mesmo tempo, a falta de uma maneira eficiente de pré-carregar um som, a não habilidade de aplicar efeitos em tempo real e a incapacidade de analisar sons. Essas limitações serviram como base para a criação do Web Audio API, que oferece uma alternativa mais moderna e apurada para controle e processamento de sons.

Web Audio é uma API JavaScript para processar e sintetizar áudio em aplicações *web*. A API se assemelha a aplicações *desktop* modernas de manipulação de áudio tendo a capacidade de mixar, processar e filtrar áudio. A API é baseada no conceito de contextos de áudio. Contexto de áudio pode ser interpretado como um grafo direcionado que define o fluxo de áudio da fonte até o destino. O áudio pode ser modificado ou inspecionado conforme ele passa entre os nodos do grafo (SMUS, 2013). O grafo pode ser extenso e complexo, como mostrado na Figura 2.5, sendo formado por vários nodos (blocos), que executam sínteses e análise, e suas conexões.

Figura 2.5 – Contexto de áudio



Fonte: (SMUS, 2013)

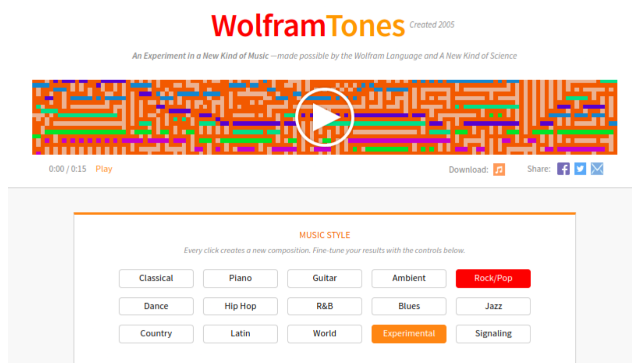
3 TRABALHOS RELACIONADOS

Este capítulo trata sobre trabalhos relacionados que utilizam técnicas de composição algorítmica e música generativa para gerar músicas na web. Existem diversas aplicações que utilizam diferentes processos para criar músicas. As aplicações mais atuais utilizam inteligência artificial e técnicas de aprendizado de máquina para realizar suas composições. Apesar disso, ainda existem aplicações *web* relevantes que não estão relacionadas com aprendizado de máquina e que usam técnicas similares às apresentadas neste trabalho. Dois exemplos são o WolframTones e o Generative.fm.

3.0.1 WolframTones

Criado em 2005, o WolframTones é baseado no trabalho de Stephen Wolfram. A aplicação utiliza ACs em conjunto teoria musical para criar músicas em diversos estilos, com diferentes instrumentos. O método que ele utiliza para criar músicas é tratar o AC como se fosse uma grade com notas, onde cada linha do autômato se refere a um tom, e as notas são representadas pelo estado de uma célula. A grade colorida da Figura 3.1 representa as notas que serão tocadas, e pode ser vista como a evolução das células de um autômato celular. As notas são filtradas de acordo com a escala escolhida e tocadas em ordem. A aplicação permite alterar parâmetros antes de criar as músicas. Entre eles é possível escolher até cinco instrumentos para tocarem ao mesmo tempo, escolher escalas, gêneros, ritmos e tempos diferentes. As músicas são totalmente dependentes da regra escolhida para o autômato, isso faz com que algumas músicas tenham alguns padrões estranhos que tornem a música *estranha* devido à evolução do AC

Figura 3.1 – Tela inicial do WolframTones



Fonte: O Autor

3.0.2 Generative.fm

O Generative.fm é uma aplicação web que utiliza composição generativa para criar músicas. As ferramentas utilizadas para criar as músicas são semelhantes as usadas neste trabalho e envolvem Web Audio e bibliotecas de Javascript para produzir áudio. A maioria das músicas geradas seguem o gênero de música ambiente, com o foco em músicas para serem ouvidas no *background* enquanto o usuário realiza outras tarefas. Cada composição utiliza um método diferente, mas a maioria dos métodos são técnicas estocásticas onde cada nota, escala e tempo é escolhido aleatoriamente baseado em algumas regras pré definidas.

4 PROJETO E IMPLEMENTAÇÃO

O objetivo deste capítulo é discorrer sobre o projeto e sua implementação.

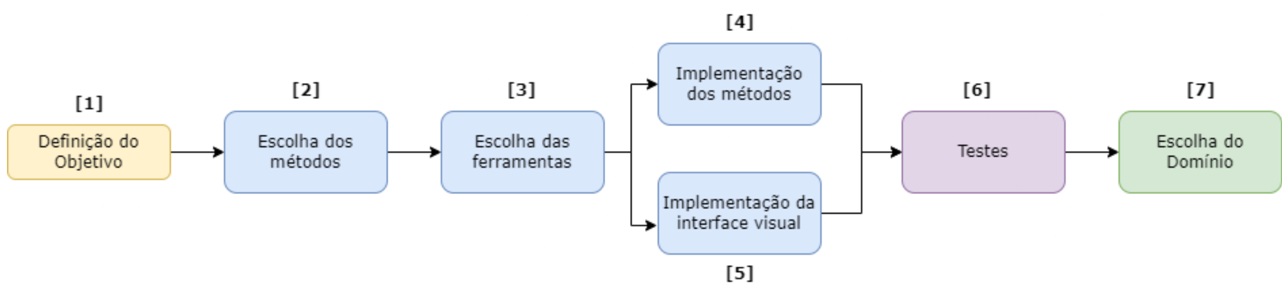
As seções seguintes irão apresentar o projeto e os seus objetivos e irão detalhar a sua implementação com os métodos e ferramentas escolhidos.

4.1 Visão Geral do Desenvolvimento

O desenvolvimento da aplicação foi divido em diversas etapas. Todas as etapas estão ilustradas na Figura 4.1.

A primeira etapa [1] foi a definição do objetivo. O objetivo nada mais é do que o resultado esperado para a aplicação, ou como ela deve se comportar e quais as funções que ela irá possuir. A etapa [2] foi o estudo dos métodos de composição algorítmica e do gênero de música *Chiptune*. Nesta etapa foram escolhidas as técnicas de composição que foram utilizadas mais tarde e o tipo de som que seria gerado pela aplicação. Após a definição das técnicas, foram escolhidas as ferramentas [3] que auxiliaram na implementação das técnicas para gerar o som. Nesta etapa foram escolhidas as linguagens e as bibliotecas utilizadas. A seguir vieram as etapas de desenvolvimento. As etapas [4] e [5] foram realizadas em conjunto e consistiram em utilizar as ferramentas para programar os métodos escolhidos e ao mesmo tempo construir uma interface visual para permitir a interação e para exibir os resultados. A penúltima etapa [6] foi onde foram realizados testes para comprovar o funcionamento correto da aplicação. E por último foi escolhido o domínio onde foi realizado o *deploy* da aplicação.

Figura 4.1 – Fluxograma do desenvolvimento



Fonte: O Autor

As próximas seções e subseções irão detalhar cada uma das etapas do desenvolvimento.

4.2 Objetivo

O objetivo da aplicação foi definido a partir de uma motivação. Muitos dos jogos que são lançados hoje em dia são feitos por desenvolvedores independentes. Estes jogos, apelidados de jogos *indie*, muitas vezes contam com um pequeno orçamento para sua realização. Isso implica em uma limitação de recursos para desenvolver um jogo, e isso incluí a sua trilha sonora que muitas vezes é composta por um mesmo conjunto de notas e de melodias que ficam repetindo indefinidamente causando um esgotamento ao ouvinte. O objetivo deste trabalho é explorar uma maneira de contornar a limitação criativa de trilhas sonoras com problemas de repetição.

Para atingir este objetivo, a aplicação deve permitir ao usuário gerar músicas que serão tocadas por um tempo indefinido e que serão sempre diferentes. Para isso serão utilizadas técnicas de composição algorítmica e composição estocástica. Além disso, o usuário pode alterar parâmetros, podendo escolher tons, tempos, escalas e progressões em que a música irá acontecer. Por fim, deve permitir que o usuário possa reproduzir a mesma sequência em qualquer outro computador e possa baixar a composição em um arquivo tipo MIDI.

4.3 Escolha dos Métodos

Neste capítulo é detalhado a escolha dos métodos e técnicas que foram implementados mais tarde. As seguintes subseções apresentam a escolha do modo de como as notas são representadas, as técnicas utilizadas para compor uma música nessa representação e como elas deverão ser reproduzidas e visualizadas.

4.3.1 Representação

Antes de escolher as técnicas utilizadas para compor as músicas, foi preciso pensar em como a música seria representada. A escolha foi representar a música como uma sequência de notas ocorrendo em uma grade dentro de um tempo determinado. Essa representação pode ser pensada como um *array* de notas, onde cada nota é representada por um valor e o zero significa o silêncio. A grade é como uma linha do tempo, onde os *arrays* de notas são dispostos para serem reproduzidos. Esta escolha foi feita para

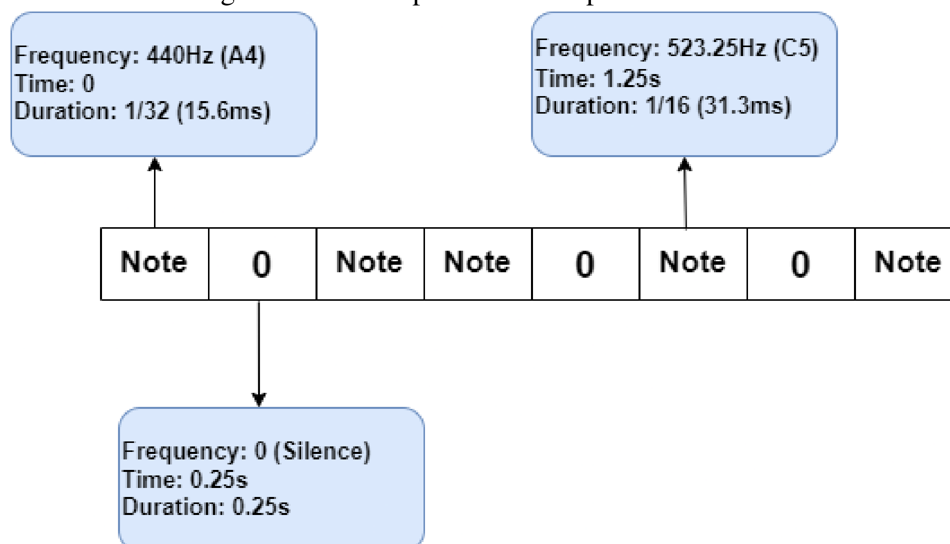
facilitar a composição com os algoritmos de ACs e Ritmos Euclideanos, visto que ambos produzem *arrays* de zeros e uns como resultado.

A linha do tempo é divida em compassos. Compassos, na música, são divisões da música em intervalos de tempos iguais, ou seja, a linha do tempo é divida em várias partes com o mesmo tamanho. A duração de cada compasso dessa linha é dada pelo tempo da música. Essa medida de tempo também indica o andamento da música, ou seja a velocidade com que ela é executada. O andamento da música pode ser dado pelo BPM. Quanto maior o BPM, menor é a duração de cada compasso, e quanto menor o BPM, maior é a duração do compasso em segundos. Por exemplo, a 120 BPM ou 120/60 = 2 batidas por segundo, um compasso de 4/4 (4 pulsos com duração de 1/4) tem duração de 2 segundos.

Cada nota pode ser tocada em uma certa fração de tempo do compasso. No caso anterior, se quisermos que uma nota toque em 16 do compasso, ela será tocada 31.3ms após o inicio do compasso. Com essa medida de tempo podemos escolher quando e por quanto tempo uma nota será tocada na linha do tempo.

Podemos pensar que uma nota é um conjunto de três parâmetros: a frequência da nota, quando ela é tocada e por quanto tempo ela é tocada. Todos esses parâmetros podem ser escolhidos por um método que, executado múltiplas vezes, gere uma sequência de notas. A Figura 4.2 demonstra um exemplo de uma sequência de 5 notas em um compasso divido em 8 partes utilizando essa representação.

Figura 4.2 – Exemplo de um compasso com notas



Fonte: O Autor

4.3.2 Composição

Como a representação de notas é um *array*, o primeiro método escolhido para gerar uma sequência de notas foi o algoritmo euclidiano, visto que ele pode ser utilizado para construir ritmos euclidianos que nada mais são do que uma sequência de notas que estão separadas igualmente. Este método é muito bom para, por exemplo, criar uma sequência de notas de baixo, que dão um ritmo à musica.

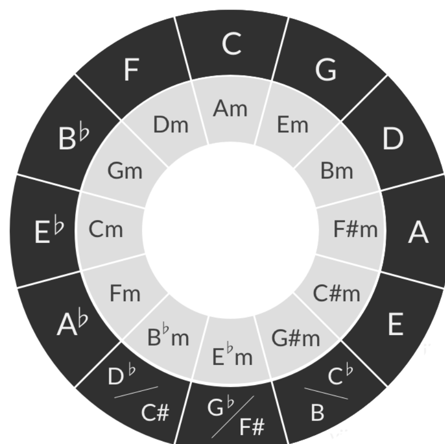
A segunda técnica escolhida para gerar um *array* de notas foi o automato celular. Com a ajuda do automato celular é possível gerar infinitas sequências de notas. Para fazer basta alimentar o autômato com um *array* aleatório com um certo número de notas. Uma regra é aplicada nesta sequência para criar uma nova geração de notas. Este processo pode ser repetido, mudando ou não a regra, indefinidamente. Assim, é possível gerar diversas sequências de notas diferentes.

As duas técnicas em conjunto fornecem diversas sequências de notas, separadas diferentemente. Porém, é preciso que estas notas façam algum sentido musical, para não serem apenas notas aleatórias dispostas aleatoriamente. Para resolver este problema é preciso um pouco de teoria musical. Uma música pode ser composta por progressões de acordes. Cada acorde é composto por múltiplas notas. Uma progressão pode ser entendida como diferentes acordes tocados em sequência dentro de uma escala musical. Ela geralmente é escrita utilizando números romanos onde cada número representa a nota dentro da escala e usa-se letra maiúscula para maior e letra minúscula para menor. Por exemplo, na escala de Dó Maior (C), a progressão I-V-vi-IV é equivalente a Dó Maior (C), Sol Maior (G), Lá Menor (Am) e Fá Maior (F). Então, uma alternativa, para que a sequência de notas faça sentido, é gerar elas dentro destas progressões de acordes, ou seja, limitar quais notas podem ser geradas em determinado tempo.

Além de variar as progressões, é preciso variar a nota principal (tônica) para que a música não fique sempre no mesmo tom. Para realizar isto é possível utilizar o ciclo de quintas e quartas (Figura 4.3). O ciclo de quintas nada mais é do que um ciclo representando notas separadas por quintas justas no sentido horário e por quartas no sentido anti-horário. Isso faz com que cada escala seguinte no ciclo de quintas tenha apenas uma nota diferente da anterior. Ele pode ser usado para realizar transições mais *suaves* ao ouvinte. Logo, para que a música não fique no mesmo tom, basta mudar a tônica da escala seguindo o ciclo de quintas e quartas no sentido horário ou anti-horário.

A transição também pode ser para uma escala relativa maior ou menor. Uma escala

Figura 4.3 – Ciclo de quintas



Fonte: O Autor

relativa é aquela que possuí as mesmas notas mas modo (maior ou menor) diferente. Por exemplo a escala de Dó Maior possui as notas C, D, E, F, G, A e B enquanto a escala de Lá Menor é composta pelas notas A, B, C, D, E, F e G. Elas possuem as mesmas notas, porém são modos diferentes. Como as notas são as mesmas, a transição também é suave.

Todas as escolhas de notas são feitas a partir de um método estocástico. As notas são escolhidas aleatoriamente dentro de suas escalas e acordes. Para que seja possível reproduzir uma mesma música novamente é preciso que seja possível gerar os mesmos números aleatórios na mesma sequência. Pensando nisso, os números aleatórios serão gerados a partir de uma *seed*. Esta *seed* é um número que será gerado a partir de uma palavra. Por exemplo, se o usuário digitar a palavra *som*, o programa vai transformar esta palava em um número que será a *seed* para a geração de números aleatórios. Se na próxima vez que o usuário abrir a aplicação ele digitar a mesma palavra, as mesmas sequências de números aleatórios irão acontecer.

4.3.3 Reprodução e Visualização

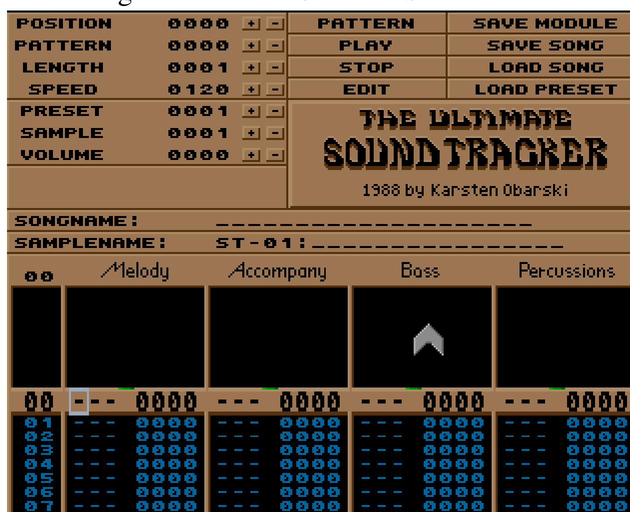
Após a representação das notas e as técnicas de composição serem definidas, foi preciso escolher como as notas seriam reproduzidas, e como as composições seriam mostradas. A reprodução das notas deve ser coerente com o estilo de música escolhido. O *Chiptune* é baseado na tecnologia dos chips de consoles antigos. Estes chips possuíam algumas limitações. Os geradores eram limitados a apenas alguns tipos de onda que geralmente eram ondas quadradas, triangulares, dentes de serra, pulso ou *pseudo-random-*

noise.

Outra limitação era o número de canais, que era em torno de 3 a 5 canais, com osciladores diferentes. Devido a esta limitação, um dos desafios era reproduzir sons polifônicos com os chips. Para contornar este problema eram utilizadas técnicas de *arpeggios* rápidos que davam a sensação de polifonia. Portanto, para reprodução das notas é preciso utilizar pelo menos 3 canais com osciladores gerando tipos de onda diferentes.

A visualização da composição foi baseada nos famosos *Soundtrackers*. O primeiro *Soundtracker* foi criado em 1987 por Karsten Obarski que estava cansado de criar músicas na mão. Ele criou um software chamado *The Ultimate Soundtracker* (Figura 4.4) que facilitou a composição de músicas para o console Amiga A500. O software representava graficamente os quatro canais do *chip* de som do console como um *piano roll* vertical. O ambiente permitia que pessoas que não tinham conhecimento em programação pudessem utilizar as ferramentas de música sem precisar aprender a programar (DRISCOLL; DIAZ, 2009).

Figura 4.4 – The Ultimate Soundtracker



Fonte: (DRISCOLL; DIAZ, 2009)

Uma visualização baseada no *Soundtracker* faz com que as notas e o tempo da música fiquem bem claros para o usuário.

4.4 Escolha das Ferramentas

Nesta seção serão detalhadas as ferramentas utilizadas para concretizar os objetivos listados na seção anterior.

4.4.1 Tone.js

O Tone.js é um *framework* de Web Audito para criar música interativa no navegador. Ele provê uma biblioteca de módulos e abstrações permitindo utilizá-los para arranjar e produzir música. Ele oferece funções que são comuns em DAWs, como uma linha de tempo global para sincronizar e programar eventos, sintetizadores e efeitos. Além disso, o *framework* também permite a edição de parâmetros para criar sintetizadores, efeitos e sinais de controle customizados.

Uma das principais vantagens desta ferramenta é que ele permite definir sintetizadores e efeitos como objetos JSON, o que facilita o controle sobre eles. Outra vantagem é poder expressar o tempo de várias maneiras diferentes. O tempo pode ser expresso em termos do *beat* da música, como por exemplo uma nota poderia tocar por "4n"que, em 120BPM, seria traduzido para 0.5 segundos. O Tone.js utiliza notação musical e rítmica, o que ajuda na hora de escolher as notas e tempos para as músicas. Com este *framework* basta duas linhas de código para conseguir produzir um som:

```
1 const synth = new Tone.Synth().toDestination();
2 synth.triggerAttackRelease("C3", "4n");
```

O Tone.js está disponível no Github sob a licensa MIT [1].

4.4.2 Total Serialism

O Total Serialism é uma biblioteca que disponibiliza uma série de métodos para gerar música proceduralmente e para transformar sequências de números. Ela é focada principalmente em técnicas de composição algorítmica e é facilmente integrada com o ToneJS. A biblioteca conta com métodos generativos, algorítmicos e estocásticos para gerar sequências de números. Além disso, o Total Serialism também conta com métodos de traduções, que são métodos para auxiliar na transformação de dados para diferentes notações, como por exemplo traduzir uma nota para uma frequência.

Os principais métodos dessa biblioteca que serão utilizados são os métodos para gerar as sequências euclideanas e os autômatos celulares. O método *euclid* permite gerar um *array* de tamanho *n* com *k* números 1 separados igualmente. O exemplo da Figura 4.5 mostra o resultado do método utilizando como parâmetros o tamanho de *array* 16 com 10

[1] https://github.com/Tone.js/Tone.js

números 1.

Figura 4.5 – Resultado do método Euclidiano

Fonte: O Autor

O método para gerar o autômato celular recebe como parâmetros um *array*, com um número determinado de células, e qual a regra para gerar a próxima geração. O código é apresentado abaixo e a Figura demonstra o resultado de 10 gerações.

```
let ca = new Algo.Automaton();
// Alimenta o automato com um array
ca.feed([1,0,1,0,0,0,1,0,0,1,0,1,0,0,1,0,1,1,0,1,0,1]);
// Define a regra com um numero decimal
ca.rule(122);
// Gera a proxima geracao
let gen = ca.next();

// Cria multiplas geracoes com um for
let gens = [];
for (let i=0; i<10; i++){
  gens.push(ca.next());
}
```

Figura 4.6 – Dez Gerações geradas pelo Autômato Celular

Fonte: O Autor

Estes dois métodos servem como base para a geração das sequências de notas da composição. O Total Serialism está disponível no Github sob a licença MIT [2].

[2]https://github.com/tmhglnd/total-serialism

4.4.3 Vue.js

O Vuej.js é um *framework* Javascript para criação de interfaces de usuário. Ele utiliza os padrões HTML, CSS e Javascript e permite uma programação declarativa e baseada em componentes. Ele foi escolhido para criar a página web pela sua simplicidade e facilidade de uso. Além de ser super simples de utilizar ele também conta com uma grande quantidade de bibliotecas oficiais e tem uma comunidade colaborativa e extensa que vem crescendo mais a cada dia.

4.5 Implementação

Está seção irá detalhar a implementação dos métodos com as ferramentas escolhidas. As seguintes subseções irão percorrer pelo processo de criação do código explicando as decisões tomadas.

4.5.1 Sintetizadores

A primeira parte a ser implementada foram os sintetizadores. A aplicação foi projetada para possuir cinco canais, cada um com um função diferente. Eles são divididos em Baixo, *Arpeggiator*, Melodia 1, Melodia 2 e Percussão. Para estes cinco canais foram implementados seis sintetizadores diferentes. A razão para haver seis é em motivo da percussão precisar de dois sintetizadores, um para o *Hi-hat* e para o *Snare* e outro para o *Kick*. Para emular o som dos *video games* antigos, os sintetizadores foram definidos como sintetizadores monofônicos com apenas um oscilador cada. Os osciladores utilizados produzem ondas quadradas, triangulares, dentes de serra e *noise*.

Para definir um sintetizador monofônico com o Tone.js basta utilizar o método *Monosynth*. Um *Monosynth* é composto por um oscilador, um filtro e dois envelopes. A amplitude e a frequência de corte do filtro são controlados pelos dois envelopes. Para tocar uma nota basta chamar o método *triggerAttackRelease* com a frequência da nota (ou a nota), a duração da nota, em que momento ela é tocada e qual a velocidade dela. A *Listing* 4.1 demonstra a declaração de um sintetizador monofônico e logo após a chamada do método para tocar uma nota.

Listing 4.1 – Declaração de um sintetizador monofônico

```
const Synth = new Tone.MonoSynth({
  oscillator: {
    type: "square"
  },
  envelope: {
    attack: 0.1
  }
}).toDestination();

Synth.triggerAttackRelease("C4", "8n");
```

Esta declaração foi repetida para os outros sintetizadores, mudando os osciladores e alguns parâmetros de envelope e de volume.

O Tone.js utiliza o *Transport* para manter a contagem do tempo. Ele permite programar eventos para acontecerem em tempos específicos. Pode-se imaginar que o *Transport* é como uma linha do tempo, com início e fim, aonde eventos podem acontecer. O *Transport* possuí métodos como *start*, *stop* e *loop*. Todos os eventos de *trigger* de notas acontecem em cima desta linha do tempo e são chamados de *ToneEvents*. Essa linha do tempo foi definida com o tamanho de 16 compassos. Estes 16 compassos foram divididos em dois *loops* de 8 compassos com 128 notas espaçadas igualmente, ou seja, no decorrer do *loop* vão ocorrer 128 eventos para cada sintetizador dentro da linha do tempo. O número de compassos foi escolhido de forma que a composição não troque tão rapidamente e não fique repetindo por muitas vezes.

Existem diversas abordagens para programar eventos em cima da linha do tempo. No caso das notas, o *framework* conta com o *Part*. Ele é uma coleção de *ToneEvents* que podem ser iniciados, parados ou em *loop*. Como argumentos ela recebe um *value* que pode ser um Objeto com diversas notas, tempos e velocidades. Esta coleção facilita bastante a reprodução das notas, visto que só é preciso montar um Objeto com quais notas irão tocar e quando elas irão tocar.

Listing 4.2 – Declaração de uma *Part*

```
const notes = [
{ time: 0, note: "C3", velocity: 0.9 },
{ time: 1, note: "C4", velocity: 0.5 },
];

const part = new Tone.Part((time, value) => {
  synth.triggerAttackRelease(value.note, "8n", time, value.velocity);
```

```
}), notes).start(0);

Tone.Transport.start();
```

O *Listing* 4.2 demonstra a declaração de um Objeto contendo duas notas que serão tocadas pelo *callback* da *Part* nos tempos 0s e 1s respectivamente.

É possível criar diversas partes que irão tocar ao mesmo tempo, com sintetizadores diferentes. Portanto, para cada sintetizador foi criado uma *Part* que fica em *loop* 8 compassos. Após definir elas, basta apenas gerar o Objeto com as notas e os tempos que serão passados para as partes tocarem.

4.5.2 Composição

A composição é a parte encarregada de gerar os Objetos que contém as notas que serão tocadas. Cada canal conta com um método diferente de geração de notas. A composição é baseada em progressões. Como o *loop* é composto por 8 compassos, cada progressão contém 4 acordes que irão tocar durante 2 compassos cada. Cada compasso do *loop* é construído em cima de um acorde que pode ser composto por múltiplas notas. Para facilitar a escrita e manipulação dos dados, todas as notas são expressas em função do seu número MIDI. Por exemplo, o C4 é dado pelo número 60 e se adicionarmos dois semitons a ele teremos o D4, que é dado pelo número 62. Essa notação facilita o uso de acordes e de escalas mais tarde.

As notas do baixo e do *arpeggiator* são baseados totalmente nos acordes das progressões, enquanto as notas da Melodia 1 e Melodia 2 seguem uma escala em cima da progressão. Uma escala é uma sequência ordenada de notas. Existem diversas escalas diferentes, por exemplo, uma escala maior possui cinco intervalos de tons e dois intervalos de semitons entre as notas que resulta na sequência tom, tom, semitom, tom, tom, tom e semitom. Essa escala pode ser expressa na notação de inteiros como [0, 2, 4, 5, 7, 9, 11, 12], onde o 0 é a nota raiz e os outros números são a quantidade de semitons acima da nota raiz. A cada 16 compassos, a nota raiz muda e todas as partes são recalculadas gerando uma nova composição.

Todas as escolhas aleatórias dependem de uma *seed*. A biblioteca do Total Serialism já possuí um método *Random* que pode receber uma *seed*, basta implementar um método para gerá-la. Para isso, foi usado um método[3] para produzir um valor *hash* de

[3] https://stackoverflow.com/questions/521295/seeding-the-random-number-generator-in-javascript

128 bits a partir de uma *string*. Este valor, então, pode ser usado para alimentar o método *Random*. Uma mesma *string* sempre irá gerar um mesmo número, e com isso é possível fazer com que uma mesma palavra sempre gere uma mesma composição. As próximas subseções irão detalhar os métodos de geração da sequência de notas de cada canal.

4.5.2.1 Baixo

A sequência de notas do baixo é baseada na nota raiz do acorde sendo tocado. Foram definidos dois padrões que o baixo pode seguir: uma sequência de notas com ritmo euclidiano, ou uma sequência de notas onde o baixo altera entre a nota raiz e a nota raiz uma oitava acima. A decisão de qual padrão deve ser utilizado é feita pela chance.

Se o padrão escolhido for o padrão euclidiano, então é gerado um *array* de tamanho 128 com um número de notas entre 60 e 100. O resultado irá definir em quais das 128 posições as notas irão tocar. Após isso, para gerar as notas em si é utilizado um *for loop* que passa por todas as posições escolhendo qual nota será tocada em cada. As notas podem ser escolhidas aleatoriamente dentro das notas de cada acorde.

Se o padrão escolhido for o da nota raiz, o *array* é gerado de forma que a cada 4 posições aconteça uma nota raiz e a cada duas posições aconteça uma nota raiz uma oitava acima. Se a nota raiz for um C4, o resultado fica da seguinte forma: [60,0,72,0,60,0,72...].

O tempo em que cada nota irá tocar é dado pelo BPM. Por exemplo, um compasso na forma 4/4 em 120BPM tem dois segundos de duração. Logo, 8 compassos tem a duração de 16 segundos. Se dividirmos 16 segundos por 128 notas, teremos que o intervalo entre as notas é de 0.125 segundos. Portanto, para montar o objeto com as notas e os tempos basta ir adicionando as notas geradas no passo acima com o tempo dado pelo cálculo anterior.

4.5.2.2 Arpeggiator

Para começar é preciso saber que um arpejo nada mais é do que um acorde quebrado em uma sequência de notas. Os consoles de *video game* antigos normalmente usavam essa técnica tocando uma sequência rápida de notas para suprir o pequeno número de canais e dar a ilusão de uma polifonia. A ideia para o *arpeggiator* é replicar esta técnica. Para isso, a sequência de notas deve conter notas que façam parte do acorde e que toquem logo uma após a outra. Logo, o *array* de notas é preenchido totalmente com notas que são escolhidas aleatoriamente dentro do acorde. Além disso, as notas podem ser escolhidas

até duas oitavas acima, para dar maior extensão ao arpejo.

4.5.2.3 Melodia 1 e Melodia 2

As duas melodias utilizam métodos diferentes para serem geradas. As notas para a primeira melodia são escolhidas utilizando o mesmo método euclidiano do Baixo, porém com diferentes parâmetros, enquanto a segunda melodia é criada utilizando autômatos celulares. A diferença entre as duas é que a primeira melodia terá as notas espaçadas igualmente, com um certo ritmo, enquanto a segunda melodia poderá ter notas espaçadas de diversas maneiras, tendo mais um comportamento de *lead*.

A criação da segunda melodia passa por três etapas. A primeira é inicializar o autômato com um *array* de tamanho 128 com números aleatórios entre 0 e 1. Após isso é preciso escolher a regra pela qual o autômato vai gerar a próxima geração, e por fim executar a regra para obter a próxima geração. A regra é decidida aleatoriamente, mas também é permitido ao usuário escolher.

4.5.2.4 Percussão

A Percussão é dividida em três partes: *Kick*, *Snare* e *Hihat*. O *Kick* é realizado por um sintetizador com um envelope de frequência, que diminui a frequência em um curto período de tempo após ser tocada, resultando em um som mais grave com um baque. O *Snare* e o *Hihat* utilizam o mesmo tipo de sintetizador de *noise*, porém o *snare* possui um *release* maior.

É preciso preencher três *arrays* diferentes, visto que são três partes diferentes. Eles são preenchidos de acordo o seguinte padrão adotado para as notas: um *Kick* a cada 8 notas, com uma chance de acontecer duas notas depois para simular um *Kick* duplo, um *Snare* a cada 4 notas e o *Hihat* no restante das notas.

4.5.3 Interface Visual

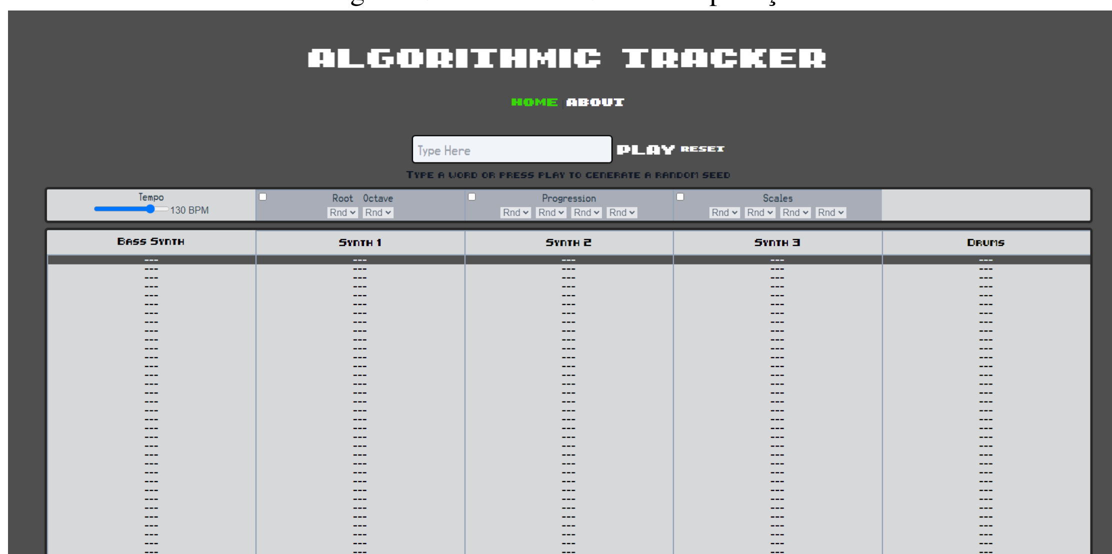
A aplicação foi projetada para ser uma SPA, ou seja, todos os eventos são carregados dinamicamente na tela sem precisar atualizar a página. O Vue.js se encarrega de realizar esse carregamento. Para auxiliar no desenvolvimento do *Frontend* foi utilizado o Tailwind CSS [4] , que é um *framework* que provê classes que facilitam o design CSS da

[4]https://tailwindcss.com/

aplicação.

A interface visual foi baseada na interface de um *tracker*. Ela é composta por colunas verticais que representam os canais e que contém as notas que estão sendo tocadas. A Figura 4.7 apresenta a interface criada. Ao usuário são apresentados um *Input*, aonde ele pode digitar o texto que irá gerar a *seed*, um botão de *Play*, um botão de *Pause* e um botão de *Reset*. Além disso, é possível customizar alguns parâmetros como Tempo, Nota Raiz, Oitava, Progressão e Escalas.

Figura 4.7 – Interface visual da aplicação



Fonte: O Autor

4.5.4 Hospedagem

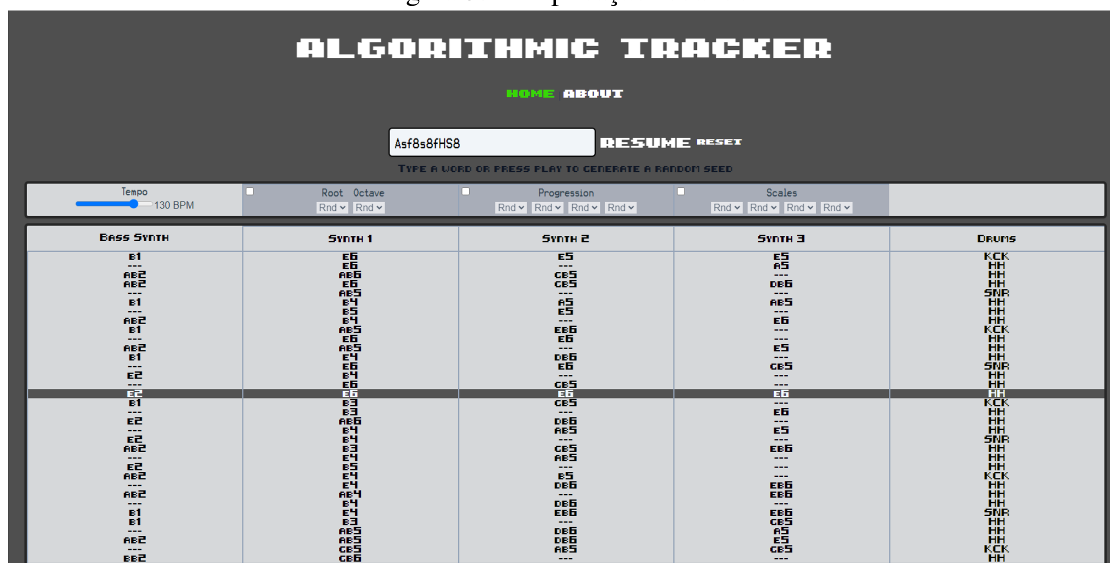
Em um primeiro momento a hospedagem do site foi realizada no GitHub Pages[5]. O GitHub Pages é um serviço de hospedagem de sites que utiliza os arquivos diretamente de um repositório no GitHub. Ele é um serviço gratuito que cumpre a sua função e é fácil de utilizar.

[5]https://pages.github.com/

5 DEMONSTRAÇÃO DE USO

Ao abrir a aplicação o Usuário irá se deparar com um *Input* textual e diversos botões. O usuário tem a opção de entrar com um texto para gerar um som, ou apenas apertar o botão *Start* para gerar uma *string* aleatória. Ao pressionar *Start*, uma composição será gerada automaticamente. Todas as notas geradas serão exibidas em seus respectivos canais e o som irá começar, como mostra a Figura 5.1.

Figura 5.1 – Aplicação rodando

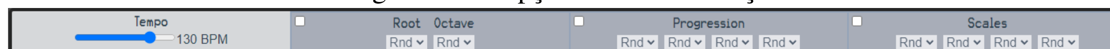


Fonte: O Autor

A cada 16 compassos uma nova composição será gerada e as notas irão trocar. Este processo irá ocorrer indefinidamente enquanto o Usuário permitir. O Usuário tem a opção de pausar a música ou retornar ao início. Para pausar a execução basta clicar no botão *Pause*, e para reiniciar a música do início basta clicar no botão *Reset*.

A aplicação permite customizar alguns parâmetros da composição. As opções são mostradas na Figura 5.2 e detalhadas na Tabela 5.1.

Figura 5.2 – Opções de customização



Fonte: O Autor

Tabela 5.1 – Tabela detalhando as funções extras

Botão	*Função*	*Valor Default*
Tempo	Controla o BPM da música. O valor mínimo é 60 BPM e o valor máximo é 150BPM	130
Root	Seleciona a nota raiz da composição	Rnd
Octave	Seleciona a oitava em que se encontra a nota raiz	Rnd
Progression	Seleciona os acordes em que a composição será baseada	Rnd
Scales	Seleciona as escalas que a Melodia 1 e a Melodia 2 irão utilizar	Rnd

Fonte: O Autor

6 CONCLUSÃO E TRABALHOS FUTUROS

Neste trabalho foi possível explorar algumas técnicas de composição algorítmica para geração de música. O resultado é uma aplicação que permite ao usuário criar diferentes composições de Chiptune a partir de qualquer navegador na web. Ela foi testada com êxito nos seguintes navegadores: Google Chrome, Mozilla Firefox e Safari. A aplicação pode ser acessada pela URL <https://predowtf.github.io/algorithmic-tracker/>.

As linguagens e APIs da web ainda possuem algumas limitações para reprodução de sons. Alguns navegadores podem não conseguir reproduzir uma música adequadamente quando introduzimos muitas funções ou muitos artefatos musicais. Uma solução para este problema é mudar a responsabilidade de realizar as funções para o *Backend* com a introdução de uma API, onde a composição e o som são gerados no *Backend* e o *Frontend* se preocupa apenas em reproduzir a música. Uma das melhorias que pode ser aplicada é a introdução de mais métodos para a composição generativa e algorítmica. Por enquanto os métodos se preocupam apenas em escolher quando as notas serão tocadas, enquanto a escolha das notas é feita apenas por chance. Isso limita as notas escolhidas e o andamento da música. Uma alternativa é o uso de inteligência artificial e aprendizado de máquina para analisar um *dataset* de composições e escolher qual é a melhor nota para ser tocada em devido momento. Outra alternativa é a introdução de cadeias de Markov para guiar as escolhas feitas , dando a elas probabilidades diferentes de acontecer.

O estudo e aplicação de composição automatizada permite explorar a interação humano-computador no âmbito criativo e com isso criar novos meios de criar e reproduzir música. Com a tecnologia atual e com o avanço da inteligência artificial, nós conseguimos criar composições que muito se assemelham a algo criado por um ser humano. Este trabalho serve como uma contribuição para a área de composição algorítmica e estará em constante desenvolvimento a partir deste momento.

REFERÊNCIAS

ALPERN, A. **Techniques for Algorithmic Composition of Music**. 1995.

COLLINS, K. In the loop: Creativity and constraint in 8-bit video game audio. **twentieth-century music**, v. 4, p. 209 – 227, 09 2007.

DEMAINE, E. D. et al. The distance geometry of music. **Comput. Geom. Theory Appl.**, Elsevier Science Publishers B. V., NLD, v. 42, n. 5, p. 429–454, jul 2009. ISSN 0925-7721. Available from Internet: <https://doi.org/10.1016/j.comgeo.2008.04.005>.

DRISCOLL, K.; DIAZ, J. Endless loop: A brief history of chiptunes. **Transformative Works and Cultures**, v. 2, 02 2009.

FARNELL, A. An introduction to procedural audio and its application in computer games. 01 2007.

FERNáNDEZ, J. D.; VICO, F. Ai methods in algorithmic composition: A comprehensive survey. **J. Artif. Int. Res.**, AI Access Foundation, El Segundo, CA, USA, v. 48, n. 1, p. 513–582, oct 2013. ISSN 1076-9757.

MIRANDA, E. R. Evolving cellular automata music: From sound synthesis to composition. In: PRAGUE UNIVERSITY OF ECONOMICS. **Proceedings of the Workshop on Artificial Life Models for Musical Applications - ECAL 2001**. 2001. Music. Available from Internet: <https://pdfs.semanticscholar.org/cb55/55ab06fd05ae1262c226e1c973b2ea055b73.pdf>.

PLUT, C.; PASQUIER, P. Generative music in video games: State of the art, challenges, and prospects. **Entertainment Computing**, v. 33, p. 100337, 12 2019.

SMUS, B. **Web Audio API: Advanced Sound for Games and Interactive Apps**. [S.l.]: "O'Reilly Media, Inc.", 2013.

TOUSSAINT, G. The euclidean algorithm generates traditional musical rhythms. In: . [S.l.: s.n.], 2005. p. 47–56.

WOLFRAM, S. **Cellular automata and complexity: collected papers**. [S.l.]: CRC Press, 1994.