



**Universidade do Minho**

Escola de Engenharia

Departamento de Informática

Carlos Brito

**Towards Procedural Music-Driven Animation**

**Exploring Audio-Visual Complementarity**

December 2017



**Universidade do Minho**

Escola de Engenharia

Departamento de Informática

Carlos Brito

## **Towards Procedural Music-Driven Animation**

### **Exploring Audio-Visual Complementarity**

Master dissertation

Master Degree in Computer Science

Dissertation supervised by

**António Ramires**

December 2017

---

## ACKNOWLEDGEMENTS

---

I would like to thank my supervisor António Ramires for giving me encouragement, support, and the creative freedom to pursue technical research rooted in personal interest and imagination. His insightfulness and clarity of instruction have contributed to my growth as a programmer and researcher and I will forever treasure his assistance and advice throughout this joint endeavour.

I would like to thank my family for their love, blind faith, unshakeable encouragement and money. They have shown me that with determination, optimism and strength one can endure and overcome any difficulty. In particular, I express gratitude to my father for his assistance throughout the final days of writing which greatly contributed to ensuring the quality of this work.

I would like to thank my girlfriend for her ceaseless support and compassion, as well as for showing me that perseverance, dedication and courage are our best tools when fighting our way through adversity.

I would like to thank all my close friends for their continued support. Those whose friendship and comradeship are founded upon mutual respect, trust and admiration, and others on mutual insult.

I would like to thank my therapist and friend Ana, for shaping my way of thinking and offering advice, insight and self-awareness all the while enduring my endless ramblings.

And last but certainly not least I would like to thank my loyal and fierce canine companion, Ché, who teaches me that to reach one's goals one must have an unyielding commitment, eternal patience and plenty hours of sleep.

---

## ABSTRACT

---

This thesis intends to describe our approach towards developing a framework for the interactive creation of music driven animations.

We aim to create an integrated environment where real-time musical information is easily accessible and is able to be flexibly used for manipulating different aspects of a reactive simulation. Such modifications are specified through the use of a scripting language and include, for instance, geometrical transformations and geometry synthesis, gradual colour changes as well as the application of arbitrary forces.

Our framework thus represents a proof-of-concept for converting musical information into arbitrary modifications to a dynamic simulation, producing a variety of animations. This is possible due to a bargaining between control and automation, where control is present by allowing the user to program these modifications with a scripting language and automation is present by using physics and interpolation to estimate the visual effects resulting from those modifications.

The particular test case for our system was the animation/simulation of a growing tree reacting to wind. In order to control or influence both the tree growth and wind field, as well as other visual parameters, the system accepts two different but complementary representations of music: a MIDI event stream and raw audio data. Different musical features are obtainable from each of these representations. On one hand, by using MIDI, we are able to discretely synchronise visual effects with the basic elements of music, such as the sounding of notes or chords. On the other, using audio, we are able to produce continuous changes by obtaining numerical data from basic spectral analysis. Our framework provides a common interface for the combined application of these different sources of musical information to the generation of visual imagery, under the form of procedural animations.

We will describe algorithms presented in multiple research papers, namely for tree generation, wind field generation and tree reaction to wind, briefly detailing our implementation and architecture. We also describe why each of these particular methods was chosen, how they are organised in our platform and how their parameters may be modified from our scripting environment leading to what we regard as the procedural generation of animations.

By allowing the user to access musical information and give them control of what we have come to refer to as animation primitives, such as wind and tree growth, we believe to have taken a first step towards exploring a novel concept with a seemingly endless expressive potential.

---

## RESUMO

---

Esta dissertação descreve o desenvolvimento de uma plataforma para a criação interativa de animações dirigidas por música. Focamo-nos em desenvolver um ambiente integrado onde vários aspetos de uma animação podem ser controlados pelo processamento em tempo real de informação musical, com recurso a uma linguagem de script.

O caso de teste específico da nossa aplicação consiste na animação de uma árvore em crescimento capaz de reagir a um campo de vento dinâmico. De forma a controlar ou influenciar quer o crescimento da árvore, quer o campo de vento, o sistema aceita como *input* duas representações diferentes, mas complementares, de música, uma sequência contínua de eventos MIDI e áudio.

Realçamos a distinção entre estas duas representações visto que apesar de serem ambas referentes a música, são fundamentalmente diferentes em termos da informação que contém. Eventos MIDI contém informação simbólica relativa à interpretação da música, nomeadamente os tempos de começo e final de notas. Por outro lado, informação áudio consiste num sinal contínuo, que resulta da gravação de um instrumento ou de uma atuação musical. Com MIDI, a nossa plataforma é capaz de sincronizar alterações discretas à simulação, com base nos elementos fundamentais da teoria musical, como o soar de notas ou acordes. Com informação áudio, é possível produzir alterações contínuas com base nos dados numéricos obtidos por análise espectral elementar do sinal de áudio.

Neste documento serão descritos vários algoritmos apresentados em artigos de investigação, nomeadamente para a geração de árvores, geração de campos de vento e reação da árvore ao vento. Iremos descrever os motivos que levaram à sua escolha, a sua organização na nossa plataforma e os vários parâmetros que podemos modificar a partir do nosso ambiente de *scripting*.

Em suma, a nossa plataforma pode ser descrita como um sistema que converte informação musical em alterações arbitrárias a um ambiente, que por sua vez influencia uma simulação reativa, produzindo animações. Foi estabelecido um compromisso entre controlo e automação de forma a tornar esta abordagem possível. O controlo provém da capacidade de programar as modificações que ocorrem no sistema, sendo que é utilizada automação de forma a estimar o movimento resultante de tais modificações.

Ao fornecer ao utilizador informação musical em tempo real e oferecer-lhe controlo sobre o que nos referimos como "primitivas de animação", como o controlo sobre vento e o crescimento da árvore, consideramos que demos um primeiro passo no que toca à exploração de um novo conceito, com um potencial expressivo aparentemente infinito.

---

## CONTENTS

---

<b>1</b>	<b>INTRODUCTION</b>	<b>1</b>
1.1	Context	1
1.2	Motivation	2
1.3	Objectives	2
1.4	Document Structure	3
<b>2</b>	<b>STATE OF THE ART</b>	<b>4</b>
2.1	Sound and Music Background	4
2.1.1	Sound	5
2.1.2	Music	5
2.1.3	Western Music Notation	8
2.1.4	Pitch	9
2.1.5	Rhythm	10
2.1.6	Dynamics	11
2.1.7	Melody	12
2.1.8	Harmony	13
2.1.9	Timbre	13
2.2	File Formats	13
2.2.1	MIDI	14
2.2.2	MusicXML	17
2.3	Visual Music	17
2.4	Algorithmic Music visualisation	20
2.4.1	Focus on Audio-Visual Complementarity	20
2.4.2	Focus on Musical Analysis	26
2.4.3	Audio and Visualisation Software	27
2.4.4	Demoscene	28
2.5	Procedural Generation	29
2.5.1	Plant Modelling	30
2.6	Tree Response to Wind	37
<b>3</b>	<b>TOWARDS PROCEDURAL MUSIC-DRIVEN ANIMATION</b>	<b>40</b>
3.1	Proposed Approach	41
3.1.1	Tree growth, Interpolation and Tweens	42
3.1.2	Wind field generation	43
3.1.3	Tree response to wind	44
3.1.4	Scripting	44

3.2	Overall Architecture	45
3.2.1	Attributes Module	46
3.2.2	Script Module	47
3.2.3	Music Module	47
3.2.4	Animation Module	48
3.2.5	Tree Module	48
3.2.6	Wind Module	49
3.3	Attributes System	49
3.4	Scripting	51
3.4.1	Script File Structure	52
3.4.2	Closures	56
3.4.3	Sol2	58
3.5	Animation	59
3.5.1	Timeline	59
3.5.2	Tweens	60
3.5.3	TweenSequence	62
3.5.4	DataLink	62
3.5.5	Cue	62
3.5.6	Application to Tree Growth	63
3.6	Audio and musical input	63
3.7	Tree Modelling	64
3.7.1	Space Colonization Algorithm	65
3.7.2	Tree Representation	66
3.7.3	Tropisms	67
3.8	Tree Physics Simulation	67
3.8.1	Dynamics Calculation	68
3.8.2	Integration of Movements	71
3.8.3	Tree Parameters	71
3.9	Tree Construction	73
3.9.1	Limitations	74
3.10	Wind Field Simulation	74
3.10.1	Uniform Flow	77
3.10.2	Source/Sink Flow	77
3.10.3	Vortex Flow	77
3.10.4	Drawbacks	77
4	USAGE AND EXPERIMENTS	78
4.1	Using AuxIn	78
4.1.1	Obtaining a MIDI File	78

4.1.2	Creating the Script File	80
4.1.3	Creating Our Musical Animation	81
4.1.4	Adding Wind	84
4.1.5	Adding Leaves	87
4.1.6	Audio Reactivity	90
4.1.7	Colour and More Wind	97
4.2	Kotowari	100
4.3	Navorski	102
4.4	Discussion	102
5	CONCLUSION	105
5.1	Prospect for Future Work	106



---

## LIST OF FIGURES

---

Figure 1	(a) Waveform of the first eight seconds of a recording of the first five measures of Beethoven’s Fifth (b) Enlargement of the section between 7.3 and 7.8 seconds (Müller (2015)). . . . .	6
Figure 2	Examples of patterns formed by wave vibrations on metal plates (Jenny (2001)). . . . .	6
Figure 3	Sheet music representation of the first five measures of Symphony No. 5 by Ludwig van Beethoven in a piano reduced version. . . . .	8
Figure 4	(a) Staff. (b) Staff with G-clef. (c) Staff with F-clef. . . . .	8
Figure 5	(a) Chromatic circle. (b) Shepard’s helix of pitch (Müller et al. (2006)) . . . . .	10
Figure 6	Symbols for most common rest (a) and note (b) durations. . . . .	10
Figure 7	Notation of time signature. (a) Four quarter notes per measure. (b) Six eighth notes per measure. . . . .	11
Figure 8	(a) Musical score of a C-major scale starting with C <sub>4</sub> and ending with C <sub>5</sub> . (b) Key signature consisting of three flats converting the sequence into a C-minor scale. . . . .	12
Figure 9	Various symbolic music representations of the first twelve notes of Beethoven’s Fifth. (a) Sheet music representation. (b) MIDI representation (in a simplified, tabular form). (c) Pianoroll representation. (Müller et al. (2006)) . . . . .	15
Figure 10	Textual description in the MusicXML format of a half note E <sub>b</sub> <sub>4</sub> . The clef, key signature, and time signature are defined at the beginning of the MusicXML file. . . . .	17
Figure 11	<i>An Optical Poem</i> frames by Oskar Fischinger . . . . .	19
Figure 12	<i>Permutations</i> frames by John Whitney . . . . .	19
Figure 13	Music visuals in Mitroo et al. (1979) . . . . .	20
Figure 14	Sound to Light Generator for the ZX Spectrum . . . . .	21
Figure 15	Cthugha Music Visualiser . . . . .	22
Figure 16	Frames generated by Milkdrop2 . . . . .	23
Figure 17	Discrete branching pattern by Ulam: (a) simple pattern with the illustration of different generations; (b) more complex branching pattern (Deussen and Lintermann (2005)). . . . .	30

Figure 18	Continuous branching structures by Cohen: (a) branching restriction introduced by minimal distance to end of branches; (b) vertical alignment of growth; (c) use of an attractor (square) and of an inhibitor (star) (Deussen and Lintermann (2005)). . . . .	31
Figure 19	Branching structure by Honda: (a) Projection to xz-plane; (b) to yz-plane; (c) to xy-plane (Deussen and Lintermann (2005)). . . . .	31
Figure 20	(a) Bloomenthal’s model of the maple tree (b) The “ramiform” primitive allows for smooth branch junctions (Bloomenthal (1985)). . . . .	32
Figure 21	Oppenheimer’s fractal models (Oppenheimer (1986)). . . . .	32
Figure 22	Greene’s Voxel Space Automata (Greene (1989)). . . . .	33
Figure 23	Hilbert curve in three dimensions (Prusinkiewicz and Lindenmayer (1990a)). . . . .	34
Figure 24	Koch Snowflake: (a) generator; (b) initiator; (c) illustrations after 1, 2, 3 and 7 rewritings (Deussen and Lintermann (2005)). . . . .	34
Figure 25	A simple branching structure Prusinkiewicz and Lindenmayer (1990a). . . . .	35
Figure 26	Synthetic landscape with self-organising trees (Palubicki et al. (2009)). . . . .	36
Figure 27	The static tree model on the left is converted into a developmental model that encompasses the ability to create arbitrary intermediate stages between a very young model and the given geometry (Pirk et al. (2012a)). . . . .	37
Figure 28	The largest model presented Quigley et al. (2017), composed of over 3 million articulated rigid bodies. . . . .	38
Figure 29	A broad overview of our system’s organisation . . . . .	46
Figure 30	Timeline interface . . . . .	59
Figure 31	Plot of the easeInQuad and easeOutQuad functions, as well as the linear function for reference. . . . .	61
Figure 32	The space colonization algorithm Runions et al. (2007) . . . . .	65
Figure 33	Tree nodes and their local coordinate system. x, y and z axes are represented in red, green blue, respectively. . . . .	67
Figure 34	The effect of tropisms in our implementation: (a) Left tropism (b) Right tropism (c) Right tropism followed by left tropism. . . . .	68
Figure 35	(a) Resting Position (b) Dynamics calculation (c) Integration of movements . . . . .	72
Figure 36	Tree mesh generation using generalized cylinders. . . . .	73
Figure 37	(a) Cylindrical coordinate system, each point is characterised by $(r, \theta, z)$ (b) Spherical coordinate system, each point is characterised by $(r, \theta, \phi)$ . . . . .	76
Figure 38	(a) Uniform Flow (b) Sink Flow (c) Source Flow (d) Vortex Flow . . . . .	76
Figure 39	2D slice of a 3D wind vector field (a) Sink Flow (b) Source Flow (c) Vortex Flow . . . . .	76
Figure 40	Our simple “melody”. . . . .	79

Figure 41	Examining the MIDI file on AuxIn . . . . .	80
Figure 42	Auxin with a spherical point cloud. . . . .	84
Figure 43	Frames from our first animation. . . . .	85
Figure 44	Adding a simple chord to our MIDI file. . . . .	86
Figure 45	Our example with a blast of vertical wind at the end . . . . .	88
Figure 46	Our animation now ending with the growth of leaves. . . . .	91
Figure 47	The audio resulting from our simple MIDI. . . . .	92
Figure 48	The evolution of the audio signal's spectrum in a logarithmic scale, as calculated by AuxIn. . . . .	93
Figure 49	Our animation with audio reactivity. . . . .	95
Figure 50	Gradual addition of leaves and color change. . . . .	99
Figure 51	A single slice of our new vortex wind primitive. . . . .	100
Figure 52	Our example with vortex wind. . . . .	101
Figure 53	A concept mock-up of the tutorial project under a visual programming paradigm. . . . .	104

---

## LIST OF LISTINGS

---

3.1	Constructor of our wind class . . . . .	50
3.2	Adding a wind primitive in lua . . . . .	51
3.3	The constructor of our UniformFlow, a derived class of AttributeSet . . . . .	51
3.4	The handlers table from our example project in section 4.1.2. . . . .	53
3.8	Typical implementation of the onUpdate function. . . . .	55
3.9	Default onNoteOn implementation. . . . .	55
3.10	Closure for calling a function during a given note interval. . . . .	56
3.11	Closure for calling a function every N notes. . . . .	57
3.12	Example use of closures in our handler structure . . . . .	57
3.13	A concrete example usage of closures . . . . .	57
3.14	The definition of make_wind_push . . . . .	57
3.15	Exposing easing functions to lua. . . . .	58
3.16	Using sol2's new_usertype for exposing the UniformFlow primitive to lua. . . . .	58
3.17	Tween update implementation . . . . .	60
3.18	Implementation of the quadratic <i>easeIn</i> functions . . . . .	61
4.1	Minimal auxin lua script file. . . . .	80
4.2	The full_growth function. . . . .	81
4.3	Assigning our function to the correct note interval. . . . .	83
4.4	Adding uniform wind. . . . .	84
4.5	Wind pull. . . . .	84
4.6	Wind pull. . . . .	86
4.7	Add leaves function. . . . .	89
4.8	Placing leaves at the final note. . . . .	89
4.9	Assigning the spectrum average to trunk displacement. . . . .	94
4.10	Caching AudioAnalyser's attributes. . . . .	94
4.11	Implementing audio reactivity with a DataLink. . . . .	96
4.12	Placing leaves at every note. . . . .	97
4.13	Changing the colour of the tree's leaves . . . . .	97
4.14	Changing leaf colour at the sounding of the final chord. . . . .	98
4.15	Placing leaves at every note. . . . .	98
4.16	Triggering the vortex wind . . . . .	98



---

## INTRODUCTION

---

Computer colours joined in harmony with music offer the composer and the artist alike a precious new amalgam of dynamic art resources. Extraordinary possibilities remain untried, unknown, even barely imaginable.

---

*John Whitney*

The creation of audio-visual compositions is an interdisciplinary subject which connects art, science and engineering. Establishing links between sound and vision has been a topic of active exploration with origins dating back to ancient Greece (Caivano (1994)).

### 1.1 CONTEXT

Sound and light are two entirely unrelated phenomena that behave in similar ways - both can penetrate materials, radiate equally in all directions, and diminish with distance following the square-cube law (Betancourt (2015)). As such, it is not surprising that artists and scientists alike are naturally drawn towards establish connections and associations between them.

The interplay between visual effects and music provides a vibrant means of artistic expression and many performances have been conducted with it in mind, providing a complementary experience to the audience. Dance, as a performing art, shows that a strong connection exists between musical rhythm and motion. Regarding colour, recent studies indicate that music-color associations are mediated by emotion (Palmer et al. (2013)).

Another valuable use for audio-visual compositions is that of teaching and aiding in the understanding of the music. A great deal of academic research follows this approach and often results in frameworks which are able to display various music theoretical aspects in graphical form. Other platforms exist which strive for a purely aesthetic visual accompaniment of music, some of which referred to as music visualiser software. This software

often employs the algorithmic generation of abstract imagery which bears some relationship to the music being played since it draws information from the underlying audio-signal. Music, however, contains much richer structure than what can be easily extracted from an audio signal. Unfortunately, this information is seldom used for animation as it is far less available and accessible.

## 1.2 MOTIVATION

Modern algorithmic visualisations of music are only possible due to the advent of digital computers. However, these frequently do not go beyond the use of the physical characteristics of sound, such as loudness and frequency spectrum.

Music contains a large amount of underlying structure (Bergstrom et al. (2007)), which music theory describes by assigning characteristics to concepts such as intervals, chords and rhythm. However, we found the visual representation of these concepts has been mostly used in the domain of teaching and aiding in the understanding of music, as opposed to artistic expression.

Although musical structure is thoroughly described by a formal theory it is often experienced in an intuitive, emotional or intellectual way.

Research into human perception of auditory-visual mappings has had a great deal of cross-disciplinary interest, namely that of visual perception and cognitive psychology. Statistical evidence has been found to indicate that there exist intuitive auditory-visual mappings common among the general population. These results encourage the design of computer music tools that support the intuitive exploration of sound-spaces (Giannakis (2006)).

## 1.3 OBJECTIVES

Our main goal consists in exploring the potential that elementary musical features hold for producing engaging animations. We do this by creating a proof-of-concept framework which allows for musical features to be connected, in a configurable and intuitive way, to having a direct impact on a reactive animation. The animation is obtained by creating a physically-based simulation, which is able to be interactively modified. These modifications may be manually described from a scripting environment with direct access to a real-time stream of musical information.

The particular case study we selected is that of an animation of a growing tree under adjustable wind conditions. We selected trees as our object of interest for three main reasons. First, the procedural generation of trees is a well-studied field. Second, trees possess several mutable aspects, such as topology, growth rate and colour, which can be used as a "canvas"

for projecting different musical aspects and third, by modifying the wind field we are able to create a variety of swaying motions which arguably resemble a primitive form of dance.

#### 1.4 DOCUMENT STRUCTURE

In chapter 2 we present a general overview of relevant scientific and artistic topics related to the visualisation of music. Afterwards, in chapter 3, we proceed to describe our proposed approach, as well as the implementation details we considered the most significant. Chapter 4 contains a tutorial on the usage of our application, as well as a brief description of two examples that we have conceived in order to explore some of the potential of our framework. Lastly, chapter 5 summarises our work and presents some possible avenues for future research and exploration.



---

## STATE OF THE ART

---

This project draws knowledge from distinct fields, ranging from music visualisation as an art form, to the mathematical and algorithmic field of procedural generation. In order to join these areas in a cohesive work, diverse research was required, as will be presented in this chapter.

We begin with an introductory section on the most basic theoretical aspects of sound and music, followed by a condensed historical perspective on the art of visualising music, the technology that supported it and some of the most influential works.

We then narrow our focus on approaches which employ electronic and digital tools. This includes the perspective of musical visualisation for impressing and entertaining, as well as for teaching and understanding musical structure. Afterwards, we visit the cultural phenomenon that is the *Demoscene*, since it bears a strong resemblance to our project by employing algorithms for the creation of audio-visual presentations.

Finally, we step into the particular problem of modelling trees, which are our chosen object of interest for this project. In order to obtain dynamic animations, this also entails simulating the interaction between trees and their environment, in particular, the wind.

### 2.1 SOUND AND MUSIC BACKGROUND

Music is an art form and cultural activity whose medium is sound and silence organised in time. Music is found in every known culture, past and present, varying widely between times and places. The origins of music date back to at least 35 000 years ago, when crude flutes were found to be carved from vulture wing bones (Wilford (2009)).

Rather than giving a comprehensive overview of musical and acoustical concepts, our goal for this chapter is to build some intuition on the subjects while introducing some basic terminology relevant for the remainder of this document.

### 2.1.1 *Sound*

A sound is generated by a vibrating object such as the vocal cords of a singer, the string and soundboard of a violin or the diaphragm of a kettledrum. These vibrations cause displacements and oscillations of air molecules, resulting in local regions of compression and rarefaction. The alternating pressure travels through the air as a wave, from its source to a listener or a microphone. At its destination, it can then be perceived as sound by the human or converted into an electrical signal by a microphone.

This signal consists in what is referred as an audio representation of sound encodes all information needed to reproduce an acoustic realisation of a piece of music. This includes the temporal, dynamic, and tonal microdeviations that make up the specific performance style of a musician.

Graphically, the change in air pressure at a certain location can be represented by a pressure–time plot, as shown in figure 1, also referred to as the waveform of the sound. The waveform shows the deviation of the air pressure from the average air pressure (Müller (2015)).

The digital representation of this waveform results in storing a large number of amplitude measurements usually referred to as **samples**. These samples relate time with the amplitude of the original signal, which then gives rise to the notion of sampling rate, the number of samples recorded by unit of time.

The most common sampling rate for audio is 44.1kHz. This number results from the sampling theorem, which states that an analogue signal can be reconstructed perfectly from its sampled version, if the analogue does not contain any frequencies higher than twice the sampling rate. This is referred to as the **Nyquist Frequency**. A poor choice of sampling rate leads to artefacts known as **aliasing**. By choosing 44.1kHz the human hearing range from 20Hz to 20 kHz is fully covered. However, in studios, it is common to use much higher sampling rates, such as 88.2kHz, 96kHz, 192kHz to reduce the probability of sampling errors (Gallagher (2015)).

The direct reaction of sound frequencies to matter give rise to intriguing figures, collectively studied by the field of *Cymatics*, as can be observed in figure 2<sup>1</sup>.

### 2.1.2 *Music*

Music can be represented in many different ways and formats. For example, a composer may write down a composition in the form of a musical score. In a score, musical symbols are used to visually encode notes and how these notes are to be played by a musician. The printed form of a musical score is also referred to as **sheet music**.

<sup>1</sup> For a musical performance exploring these phenomena please visit <http://nigelstanford.com/Cymatics/>

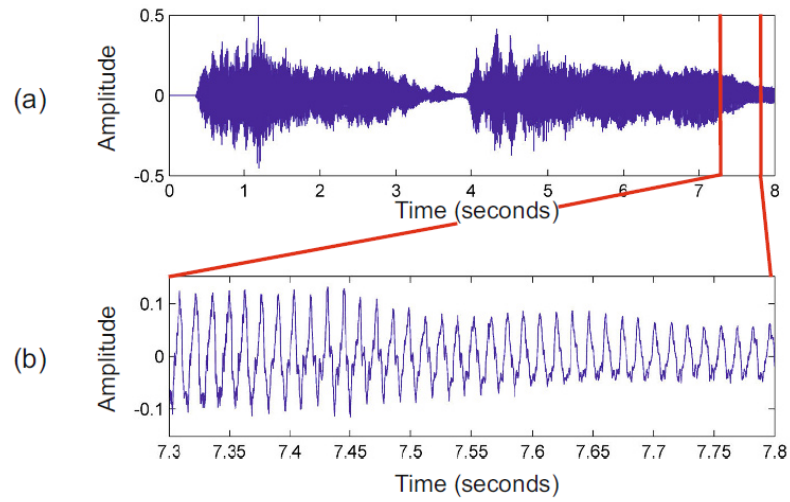


Figure 1: (a) Waveform of the first eight seconds of a recording of the first five measures of Beethoven's Fifth (b) Enlargement of the section between 7.3 and 7.8 seconds (Müller (2015)).

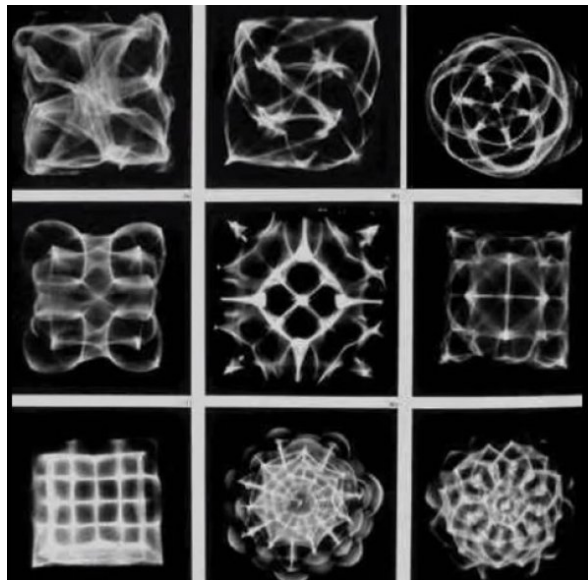


Figure 2: Examples of patterns formed by wave vibrations on metal plates (Jenny (2001)).

We will follow the convention used in Müller (2015) and use the term *symbolic* to refer to any machine-readable data format that explicitly represents musical entities. These musical entities may range from timed note events, as is the case of MIDI files (see 2.2.1), to graphical shapes with associated musical meaning, as is the case of music engraving systems.

These symbolic representations are distinct from audio representations such as WAV or MP3 files, which do not explicitly specify musical events. Such files directly encode the electrical signal captured by microphones as stated in section 2.1.1.

Each of these representations reflects certain aspects of a musical object, but no single representation encompasses all its properties.

Music is much more than a description of the notes to be played. Music is about making, creating, and shaping sounds. When musicians start delving into the music, the playing instructions recede into the background. The musical meter turns into a rhythmic flow, the different note objects melt into harmonic sounds and smooth melody lines, and the instruments communicate with each other. Musicians get emotionally involved with their music and react to it by continuously adapting tempo, dynamics, and articulation. Instead of playing mechanically, they speed up at some points and slow down at others in order to shape a piece of music. Similarly, they continuously change the sound intensity and stress certain notes. All of this results in a unique performance or an interpretation of the piece of music (Müller (2015)).

Music theory accompanied the evolution of music and includes considerations of tonal systems, scales, tuning, intervals, consonance, dissonance, duration proportions and the acoustics of pitch systems. A body of theory exists also about other aspects of music, such as composition, performance, orchestration, ornamentation, improvisation and electronic sound production.

In this section, we hope to give a very broad introduction of what is commonly referred to as the fundamental constituents of music. According to Gardner (2011), there is little dispute about the principal constituent elements of music, though experts differ on their precise definitions.

As an illustrative example of musical symbols and concepts, we will refer to the sheet music representation of Symphony No. 5 in C minor by Ludwig van Beethoven as present on figure 3. This is one of the most popular and best-known compositions in classical music. It begins with a short musical idea, the famous “short-short-short-long” motif, which is commonly referred to as the “fate motif” of Beethoven’s Fifth.



Figure 3: Sheet music representation of the first five measures of Symphony No. 5 by Ludwig van Beethoven in a piano reduced version.

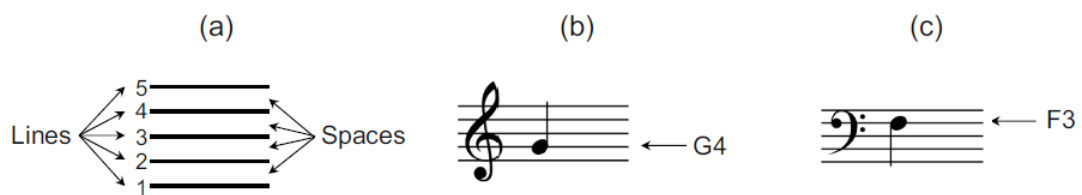


Figure 4: (a) Staff. (b) Staff with G-clef. (c) Staff with F-clef.

### 2.1.3 Western Music Notation

Generally speaking, music notation refers to a system for graphically representing music through symbols. The standard Western music notation is based on a **staff**, which is a set of five horizontal lines and four spaces each representing a different musical pitch.

Appropriate music symbols, depending upon the intended effect, are placed on the staff according to their corresponding pitch or function. Pitch is shown by the vertical placement of note symbols on the staff, sometimes modified by **accidentals**. The higher the placement within a given staff, the higher the pitch of the corresponding note. Furthermore, the duration is indicated by the shapes of the note symbols as well as additional symbols such as dots and ties.

The notation is read from left to right. A staff generally begins with a clef symbol, which indicates the position of one particular note on the staff. For example, by convention, the treble clef, also known as the G-clef, indicates that the second line is the pitch G<sub>4</sub> (see figure 4b). Similarly, the bass clef, also known as the F-clef, indicates that the fourth line is the pitch F<sub>3</sub> (see figure 4c).

Following the clef, the **key signature** placed on the staff indicates the key of the piece and lastly, the **time signature** can be found. These concepts will be detailed in the following sections.

#### 2.1.4 Pitch

Pitch determines how “high” or “low” a note sounds. It is an auditory sensation in which a listener assigns musical tones to relative positions on a musical scale based primarily on their perception of the frequency of vibration. As such, it is closely related to the physical concept of frequency, but the two are not equivalent. Frequency is an objective attribute that can be measured while the pitch is each person’s subjective perception of a sound wave.

Pitches are usually associated with frequencies by comparison with pure tones, which have periodic, sinusoidal waveforms. Playing a note on an instrument results in a (more or less) periodic sound of a certain fundamental frequency (i.e. the lowest frequency of a periodic waveform). This fundamental frequency is closely related to what is meant by the pitch of a note.

Human perception of sound is fundamentally logarithmic. As a result, two notes with fundamental frequencies in a ratio equal to any power of two (e.g., half, twice, or four times) are perceived as very similar. This interval between pitches is referred to as an **octave** and this phenomenon is referred to as octave equivalency: pitches one or more octaves apart are considered musically equivalent in many ways. Consequently, notes an octave apart are given the same note name in Western music notation. Another relevant definition is that of a **pitch class**, which is the set of all pitches or notes that are an integer number of octaves apart, such as C or D.

In order to describe music using a finite number of symbols, one needs to discretize the space of all possible pitches. This leads to the notion of a musical scale, which can be thought of as a finite set of representative pitches.

Although many other tuning systems exist, in virtually all western music, the octave has been divided into 12 logarithmically equal steps, labelled **semitones**. The resulting sequence of pitches is referred to as the **twelve-tone equal temperament chromatic scale** and thus contains every defined pitch: C; C $\sharp$ /D $\flat$ ; D; D $\sharp$ /E $\flat$ ; E; F; F $\sharp$ /G $\flat$ ; G; G $\sharp$ /A $\flat$ ; A; A $\sharp$ /B $\flat$ ; B. The next note in the scale would be C, only an octave higher than the previous. C $\sharp$  and D $\flat$ , as well as the other examples, represent the same pitch class<sup>2</sup>, even though from a musical point of view one distinguishes between these two concepts.

The term chromatic is derived from the Greek word chroma, meaning colour. In the music context, the term “chroma” closely relates to the twelve different pitch classes. For example, the notes C<sub>2</sub> and C<sub>5</sub> both have the same chroma value C. In other words, all notes that have the same chroma value belong to the same pitch class.

As previously stated, notes that belong to the same pitch class (or have the same chroma value) are perceived as similar in a certain way. This justifies the usage of the term “chroma” in the sense that notes with different chroma values have a different “sound colour”.

<sup>2</sup> This phenomenon is also known as **enharmonic equivalence**.

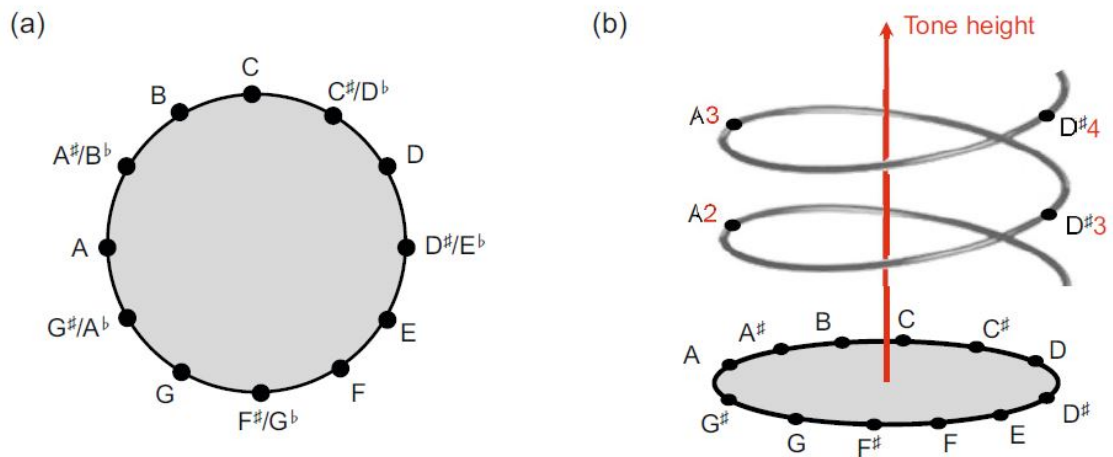


Figure 5: (a) Chromatic circle. (b) Shepard's helix of pitch (Müller et al. (2006))

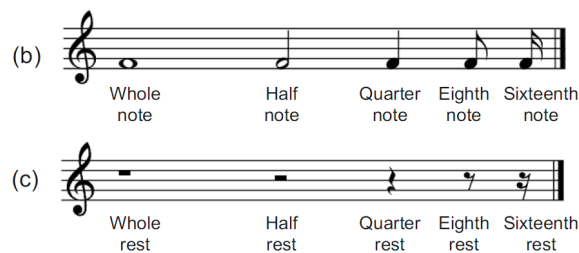


Figure 6: Symbols for most common rest (a) and note (b) durations.

Pitches are then cyclical in nature, and to completely specify which pitch we are referred to, an additional piece of information is required: a number representing its octave. This representation of pitch is called *Scientific Pitch Notation (SPM)* and can be seen in figure 4. The cyclic nature of chroma values is illustrated by the chromatic circle as shown in 5. Extending this notion, Shepard's helix of pitch represents the linear pitch space as a helix wrapped around a cylinder so that octave-related pitches lie along a single vertical line.

### 2.1.5 Rhythm

Rhythm is the element of time in music. It comprises different concepts, such as **duration**, **tempo** and **meter**.

**Tempo** refers to the speed at which a piece of music is played. This usually translates in how many *beats* there are in a unit of time, with *Beats per Minute (BPM)* being the most common measure.

Tempo can also be described in a more subjective way, commonly using Italian words, such as *Largo*, very slow pace, *Moderato* a moderate pace and *Presto* a very fast one. Tempo is

not necessarily fixed within a piece, a composer may indicate a complete change of tempo, often by using a double bar and introducing a new tempo indication, often with a new time signature and/or key signature. It is also possible to indicate a more or less gradual change in tempo, for instance with an *accelerando* (speeding up) or *ritardando* (slowing down) marking.

Notes and rests are the fundamental building blocks of music. Rests correspond to silence and notes correspond to the sounding of given pitch. Both have a duration, which is usually a fraction of the tempo defined for the piece. The duration is represented by a specific symbol, such as a whole-note, half-note or a quarter-note, as illustrated on 6, as well as additional symbols, such as dots and ties.

Usually, these notes are organised in **measures**, also referred to as **bars**. Each measure defines a segment of time corresponding to a specific number of beats, in which each beat, or sequence of beats, is represented by a particular note or rest, and the boundaries of the bar are indicated by vertical lines. Dividing music into bars provides regular reference points to pinpoint locations within a musical composition.

The number and length of notes in each measure usually follow a pattern. This pattern is indicated at the beginning of the composition, as two stacked numbers. This is the piece's **time signature** or **metre**. The bottom number indicates what the unit duration is and the top number indicates how many of these there are in a single bar. Common time signatures are 4/4 and 6/8. In the first case, the bottom number is 4, which means the unit duration is the quarter note and the top number is 4 which means there are four quarter notes per measure. In the second case, the unit duration is the eighth note and there are six eighth notes per measure as can be seen in figure 7.

A metre may also determine recurring accent patterns, that is, how emphasis should be given to individual notes based on their position on the measure. For instance, in a triple meter (3/4), it is common to have the first beat stronger than the two that follow.

### 2.1.6 Dynamics

Dynamics refers to the relative loudness or quietness of a musical performance and may also refer to other aspects of the execution of a given piece. In written compositions, dy-

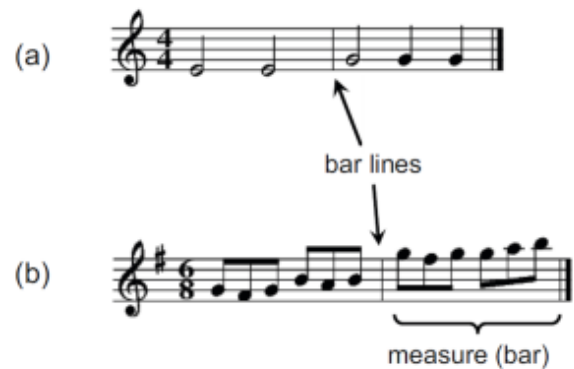


Figure 7: Notation of time signature. (a) Four quarter notes per measure. (b) Six eighth notes per measure.



namics are indicated by abbreviations or symbols that signify the intensity at which a note or passage should be played. They can be used like punctuation in a sentence to indicate precise moments of emphasis.

In classical music, the terms used to describe dynamic levels are often presented in Italian, such as *pianissimo* (very quiet), *mezzo-forte* (moderately loud) and *forte* (loud). *Crescendo* and *Diminuendo* are other fundamental concepts of dynamics, and respectively indicate that the performance should gradually become louder or quieter. Lastly, an **accent** is emphasis placed on a particular note or set of notes or chord.

An accent may alter any aspect of the note, be it attack, duration or even pitch. Due to its expressive nature, several markings exist to describe different accents. For instance, **articulation marks** are used to indicate how certain notes are to be played. For example, a *staccato* mark (a dot placed above or below a note) signifies that a note is to be played with shortened duration detached from the subsequent note, whereas a *legato* mark (a curved line placed above or below a group of notes) indicates that musical notes are played smoothly and connected.

### 2.1.7 Melody

Melody is the element that focuses on the horizontal presentation of pitch and can be interpreted as a sequence of notes. Melodies often consist of one or more musical phrases or motifs and are usually repeated throughout a composition in various forms, often making them one music's most memorable aspects.

Melody is often composed based on the *key signature* of the composition. The key signature is usually associated with a particular scale and defines the most common notes used throughout the musical composition. The key signature is represented as a set of sharp(♯) and flat(♭) markings on the staff, following the clef, specifying that certain notes are flat or sharp throughout the piece unless otherwise specified. For example, the notes shown in figure 8a are C<sub>4</sub>, D<sub>4</sub>, E<sub>4</sub>, F<sub>4</sub>, G<sub>4</sub>, A<sub>4</sub>, B<sub>4</sub>, C<sub>5</sub> thus forming a C-major scale. Using the key signature consisting of three flats as shown in figure 1.5b, the notes become C<sub>4</sub>, D<sub>4</sub>, E<sub>♭4</sub>, F<sub>4</sub>, G<sub>4</sub>, A<sub>♭4</sub>, B<sub>♭4</sub>, C<sub>5</sub> thus forming a (natural) C-minor scale.



Figure 8: (a) Musical score of a C-major scale starting with C<sub>4</sub> and ending with C<sub>5</sub>. (b) Key signature consisting of three flats converting the sequence into a C-minor scale.

### 2.1.8 Harmony

Harmony is the verticalization of pitch. Most often harmony is thought of as the art of combining pitches into chords (several notes played simultaneously as a "block"). These chords are then arranged into sentence-like patterns called progressions. Usually, the harmony is considered the accompaniment to a song, while the melody is the lead.

Harmony is often described as being *dissonant* when it produces a harsh-sounding harmonic combination or *consonant* when the combination is smooth-sounding. Dissonant chords produce musical "tension" which is often "released" by eventually transitioning to consonant chords. However, since each person's perception of consonance and dissonance differ, these terms are somewhat subjective. This leads to a discussion on tonality. A vast majority of western music is tonal, which means there is a certain key which is considered the centre of the composition and provides a sense of consonance. This is, however, a very vast subject, and a more detailed discussion would go beyond the scope of this work.

### 2.1.9 Timbre

In intuitive terms, timbre is what allows for sounds with identical pitch and loudness to be distinguishable. That is, it is what allows the same note to be differentiated among different instruments. Interestingly, timbre is also referred to as "tone colour", since it can be used analogously by a composer as colours are used by a painter. For example, the upper register (i.e. range) of a clarinet produces tones that are brilliant and piercing, while its lower register gives a rich and dark tone.

A sound generated on any instrument produces many modes of vibration that occur simultaneously. A listener hears numerous frequencies at once. A **partial** is any of the sinusoids by which a musical tone is described and the lowest partial is referred to as the **fundamental frequency**. Any frequencies which are an integer multiple above (twice, three-times, four-times) are referred to as **overtones**. The distribution of intensity among these frequencies and their variation over time is one of the key "ingredients" of timbre since it gives rise to the unique sound which uniquely characterises every instrument.

## 2.2 FILE FORMATS

In this section, we will give a brief overview of the most relevant file types for the representation of symbolic musical information. MIDI is a very condensed representation of music, due to its original intended purpose as a standard for communication between electronic musical devices such as digital keyboard and synthesizers. MusicXML, on the other hand, can be regarded as the digital equivalent of sheet music and as such provides much more

information regarding musical structure. The use of MIDI carries several benefits, as referred in section 3.6. However, we also describe MusicXML since, ideally, it could provide our system with a great deal more of musical information, as well as providing a familiar and intuitive interface for musicians.

### 2.2.1 MIDI

In this section, we intend to give the reader an overview of the MIDI representation of musical data. The most concise and intuitive introduction we found on this subject is part of the book *Fundamentals of Music Processing* by Müller et al. (2006). As such, this was our primary reference for the following text.

*“Musical Instrument Digital Interface (MIDI) is a technical standard that describes a communication protocol, digital interface and electrical connectors to allow for a wide variety of electronic musical instruments, computers and other related music and audio devices to connect and communicate with one another” (MIDI).*

MIDI was originally developed so as to allow digital electronic musical instruments from different manufacturers to work and play together. It was the advent of MIDI in 1981–1983 that caused a rapid growth of the electronic musical instrument market. MIDI allows a musician to remotely and automatically control an electronic instrument or a digital synthesizer in real-time.

It is an important fact that MIDI does not represent audio, but only represents performance information which encodes the instructions about how music is to be produced. To do so MIDI specifies a set of messages which describe precisely what notes are to be played and how.

An illustrative example is that of a digital piano. When a key is pressed on a keyboard the intensity of the sound is controlled by the velocity of the keystroke and the release the key stops the sound. Instead of physically pushing and releasing the piano key, the musician may also instruct the instrument to produce the same sound by transmitting equivalent MIDI messages, which encode the note-on, the velocity, the note-off, and other information.

The original MIDI standard was later expanded in order to include the *Standard MIDI File (SMF)* specification, which describes how MIDI data should be stored on a computer. A MIDI file contains a list of MIDI messages together with timestamps, which determine the timing of the messages.

For the purposes of this work, the most important MIDI messages are the note-on and the note-off commands, which correspond to the start and the end of a note, respectively.

In MIDI files, each note-on and note-off message is characterised by a MIDI note number, a value for the key velocity, a channel specification, as well as a timestamp.

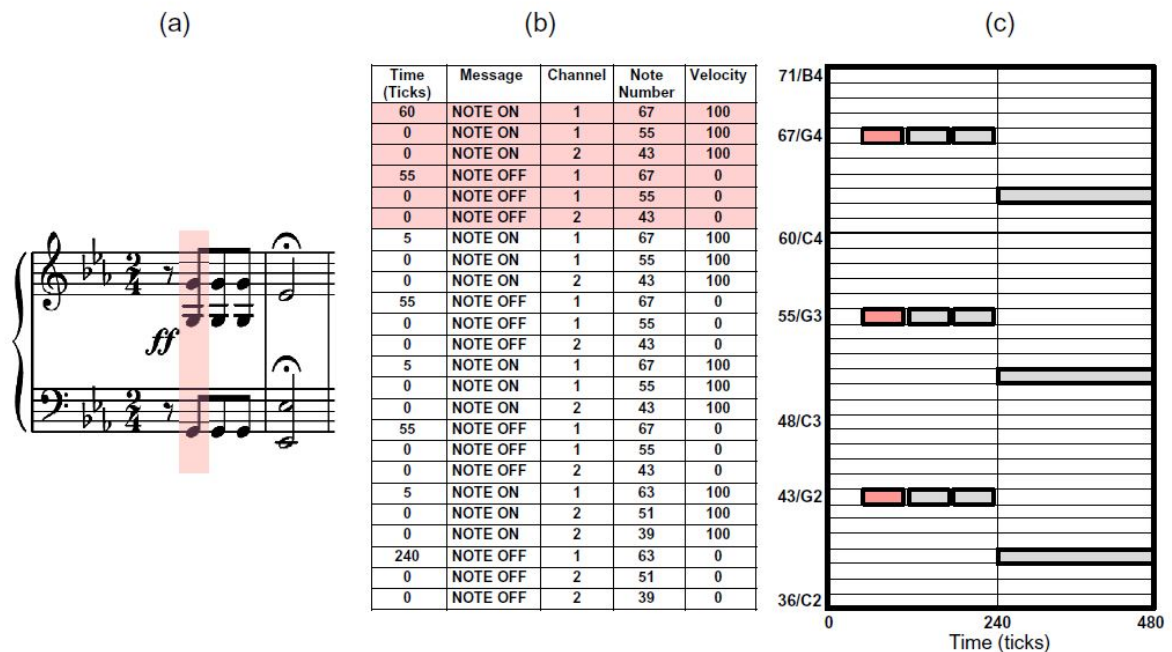


Figure 9: Various symbolic music representations of the first twelve notes of Beethoven's Fifth. (a) Sheet music representation. (b) MIDI representation (in a simplified, tabular form). (c) Pianoroll representation. (Müller et al. (2006))

The **MIDI note number** is an integer between 0 and 127 and encodes a note's pitch. MIDI pitch values are based on the equal-tempered scale as discussed in section 2.1.4. Similarly to an acoustic piano, where the 88 keys of the keyboard correspond to the musical pitches A<sub>0</sub> to C<sub>8</sub>, the MIDI note numbers encode, in increasing order, the musical pitches C<sub>0</sub> to G<sub>#9</sub>. For example, note C<sub>4</sub> has the MIDI note number 60, whereas the concert pitch A<sub>4</sub> has the MIDI note number 69.

The **key velocity** is an integer between 0 and 127, which controls the intensity of the sound. In the case of a note-on event, it determines the volume, whereas in the case of a note-off event it controls the decay during the release phase of the tone. The exact interpretation of the key velocity, however, depends on the particular instrument or synthesizer.

The **MIDI channel** is an integer between 0 and 15. Intuitively speaking, this number prompts the synthesizer to use the instrument that has been previously assigned to the respective channel number.

Lastly, the **time stamp** is an integer value that represents how many clock pulses or ticks to wait before the respective note-on or note-off command is executed.

We illustrate the MIDI representation by means of an example. figure 9 shows a (simplified and tabular) MIDI encoding of the first fate motif corresponding to the twelve notes of the score in figure 9a. In this example, the notes of the right hand are assigned to channel

1 and the notes of the left hand to channel 2. The notes specified by corresponding note-on and note-off events in the MIDI file can also be visualised by a piano-roll representation (see figure 9c).

An important feature of the MIDI format is that it can handle musical as well as physical onset times and note durations. Similarly to sheet music representations, MIDI can express timing information in terms of musical entities. To this end, MIDI subdivides a quarter note into basic time units referred to as clock pulses or ticks. The number of *pulses per quarter note (PPQN) (PPQN)* is specified at the beginning, in the header of a MIDI file, and refers to all subsequent MIDI messages.

A common value is 120 PPQN, which determines the resolution of the timestamps associated to note events. A time stamp indicates how many ticks to wait before a certain MIDI message is executed, relative to the previous MIDI message. For example, in figure 9 the first note-on message with MIDI note number 67 is executed after 60 ticks, which corresponds to the eighth rest at the beginning of Beethoven's Fifth. The second and third note-on messages are executed at the same time as the first one, encoded by the tick value zero. Then, after 55 ticks, MIDI note 67 is switched off by the note-off message, and so on.

Like sheet music representation, MIDI also allows for encoding and storing absolute timing information, at a much finer resolution level and in a more flexible way. To this end, one can include additional tempo messages that specify the number of microseconds per quarter note. From the tempo message, one can compute the absolute duration of a tick. For example, having 60000  $\mu s$  per quarter note and 120 PPQN, each tick corresponds to 5000  $\mu s$ . Furthermore, one can derive from the tempo message the number of quarter notes played in a minute, which yields the tempo measured in BPM. For example, the 60000  $\mu s$  per quarter note correspond to 100 bpm.

While the number of pulses per quarter note is fixed throughout a MIDI file, the absolute tempo information may be changed by inserting a tempo message between any two note-on or other MIDI messages. This makes it possible to account not only for global tempo information but also for local tempo changes such as *accelerandi* and *ritardandi* (Müller (2015)).

### *Limitations*

MIDI was originally designed to solve problems in electronic music performance and is limited in terms of the musical aspects it represents. For example, MIDI is not capable of distinguishing between a  $D\flat_4$  and an  $E\sharp_4$ , both of which have the MIDI note number 63. Furthermore, MIDI does not define a note element explicitly; rather, notes are bounded by note-on and note-off events. Rests are not represented at all and must be inferred from the absence of notes (Müller (2015)).

```

<note>
  <pitch>
    <step>E</step>
    <alter>-1</alter>
    <octave>4</octave>
  </pitch>
  <duration>2</duration>
  <type>half</type>
</note>

```




Figure 10: Textual description in the MusicXML format of a half note Eb4. The clef, key signature, and time signature are defined at the beginning of the MusicXML file.

### 2.2.2 MusicXML

MusicXML has been developed to serve as a universal format for storing music files and sharing them between different music notation applications. Following the general *Extensible Markup Language (XML)* paradigm, MusicXML is a textual data format that defines a set of rules for encoding documents in a way that is both human and machine readable. It is currently supported by most score writing programs, including *Finale*, *Sibelius*, as well as *Optical music recognition (OMR)* programs and sequencer programs.

This representation contains explicit information regarding musical symbols such as the staff system, clefs, time signatures, notes, rests, accidentals, and dynamics. Compared with MIDI, this format is much closer to what is actually shown in sheet music and as such the MIDI limitations referred in section 2.2.1 are no longer present in a MusicXML representation.

A minimal but complete example can be found in [Musicxml tutorial](#). As with most XML-based formats, MusicXML is particularly verbose, leading a large file size. In order to address this issue, MusicXML 2.0 added a compressed zip format which can make files roughly 20 times smaller than their uncompressed version.

## 2.3 VISUAL MUSIC

*Visual music* is a term used to refer to a broad range of artistic practices united by a common idea: that visual art can aspire to the dynamic and nonobjective qualities of music. This translates into a strong influence of musical structures in the visual arts, from paintings to films - and now to computer programs - the manifestations of visual music have evolved along with the technology available to artists ([Jones and Nevile \(2005\)](#)).

Interest in audiovisual compositions dates back to ancient Greece when philosophers Aristotle and Pythagoras had already speculated that there must be a correlation between

the musical scale and the rainbow spectrum of hues (Caivano (1994)). Sir Isaac Newton and many others have proposed different connections of sound and vision through the wave properties of colour in light and pitch in music, usually in the form of mappings between hues and pitch (Alves (2005)). However, due to the subjective nature of both music and colour perception, a definite answer is yet to be found and remains a subject of great debate and artistic exploration.

The earliest registered example of technology being used to aid in this form of expression was invented by French Jesuit monk Louis-Bertrand Castel, who proposed the idea of *Ocular Harpsichord* in the 1730's. This device was similar to a piano in construction, except for also having small coloured glass panes, each with a curtain that would lift briefly to show a flash of corresponding colour when a key was struck (Moritz (1997)). By the late 19th century, the American inventor Bainbridge Bishop and the British painter A. Wallace Rimington had separately created musical devices that employed carbon-arc lamps to produce bright flashes of colour. In the 20th century, numerous reinventions of the device emerged, each nearly identical to its predecessors save for their differing schemes for mapping colours to notes (Betancourt (2015)). These devices were collectively labelled as *colour organs*.

In the 1920s Walther Ruttmann and Oskar Fischinger were pioneering visual music films in Germany. Oskar Fischinger is regarded as one of the most influential artists within visual music. He created a series of highly acclaimed films which were some of the earliest examples of abstract animation to display an intuitive connection between musical and visual form (Moritz (1997)). Built entirely with stop-motion techniques, by the direct colouring film, they consisted mostly of delicately animated fluid forms engaging in harmonious motions or transformations inspired by the accompanying music (see figure 11). Fischinger was also involved with the animation of Disney's musical *Fantasia* (Searle (2013)) and went later on to invent the *Lumigraph*, a device which produced imagery by pressing objects or hands into a rubberised screen that would protrude into coloured light. In 1964 the Lumigraph was used in the science fiction film *The Time Travelers*, in which was renamed to *lumichord*.

Also noteworthy is the work of John and James Whitney, pioneers in the computer animation industry, who developed music-driven films using intricate home-made analogue computers (see figure 12). Their techniques inspired the traversal of the space-time continuum scene in Kubrick's famous sci-fi movie "2001: Space Odyssey" (Youngblood and Fuller (1970)). John Whitney was a strong believer that computers held tremendous potential for merging music with visual art and dedicated himself to establishing a theory of how sound and musical properties could mathematically relate to motion and animation (Alves (2005)).

Many more noteworthy artists have used this form of expression, however, a rigorous analysis would go beyond the scope of this dissertation. A comprehensive survey, both



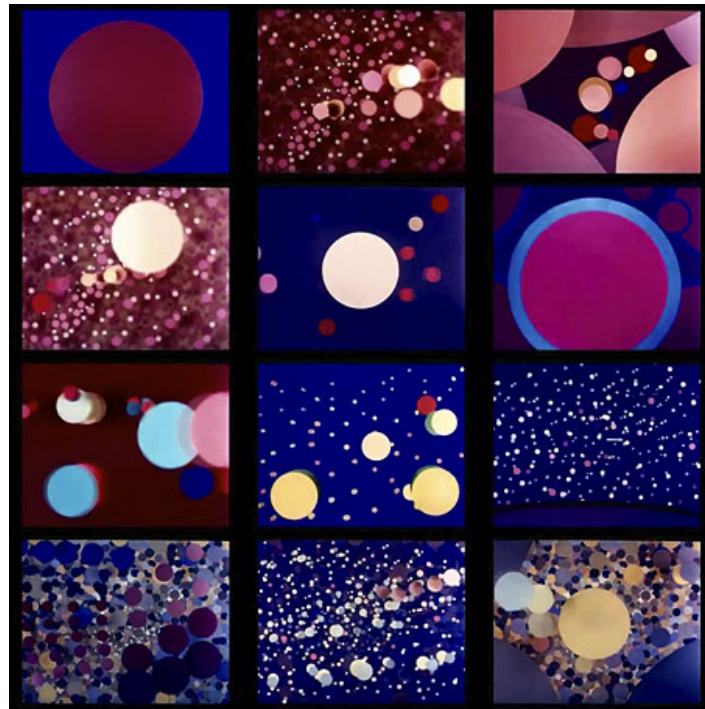


Figure 11: *An Optical Poem* frames by Oskar Fischinger

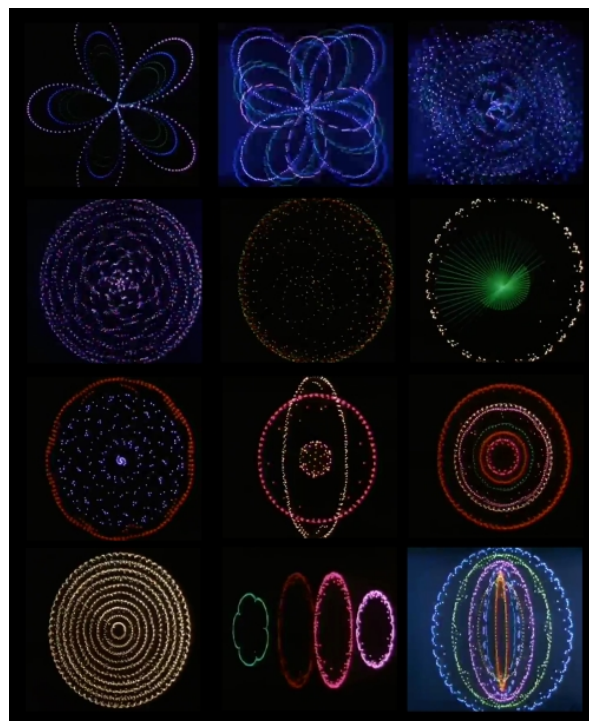


Figure 12: *Permutations* frames by John Whitney



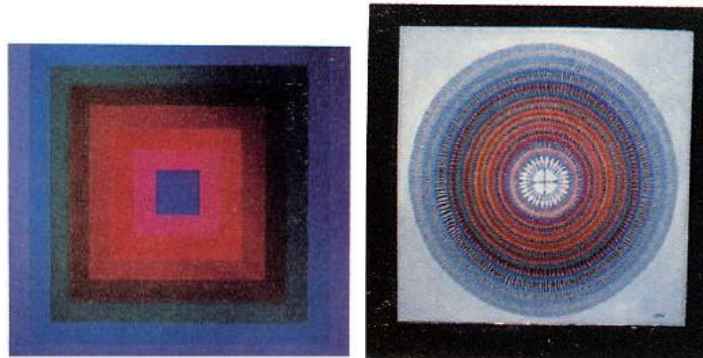


Figure 13: Music visuals in [Mitroo et al. \(1979\)](#)

on artists and the history of visual music, can be found in [Behravan \(2007\)](#). An excellent review of visual music is also presented in [Evans \(2005\)](#).

## 2.4 ALGORITHMIC MUSIC VISUALISATION

### 2.4.1 *Focus on Audio-Visual Complementarity*

The earliest examples of electronic music visualisers were “light organs”. These were devices which automatically converted audio signals into rhythmic lighting effects and usually functioned by decomposing the audio signal into several frequency bands, whose intensity then regulated the intensity of differently-coloured lights. The first music visualiser resembling modern software was developed by Atari in 1976. It consisted of a device which received input from a Hi-Fi system and produced simple imagery based on waveform information. It was, however, unsuccessful and was removed from the market after a single year ([Edwards \(2016\)](#)).

Among the earliest approaches to use a computer-based approach for musical visualisation was presented in [Mitroo et al. \(1979\)](#), where the authors developed software for generating animations and images from music. The theory of music visualisation proposed by artist Nancy Herman was implemented. It comprises the particular association between the amount of white present in the colour and the musical octave of each note. Images were produced starting at the centre of the canvas, with notes or chords appearing as concentric areas in a basic shape, such as circles or squares, as can be seen in [figure 13](#)

Soon after the appearance of personal computers, the first real-time visualisers appeared for the ZX Spectrum as shown in [figure 14](#).

From the mid-1990s onward, several music players began incorporating visualiser software and the concept became widespread.

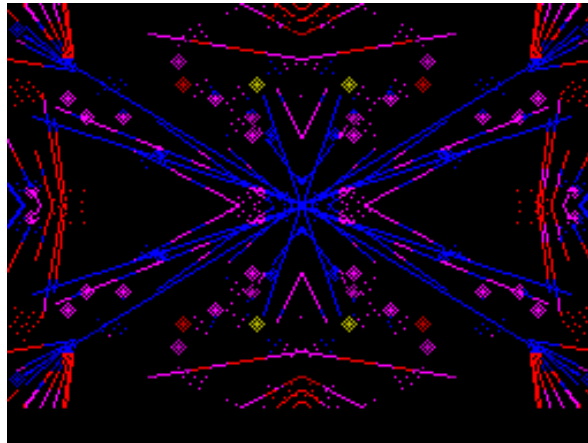


Figure 14: Sound to Light Generator for the ZX Spectrum

Concurrently, *Video Jockey (VJ)* culture began to emerge, as artists began specialising in producing visual content for concerts, nightclub and music festivals (Correia (2013)).

The earliest example of visualiser software is *Cthugha*, an open-source and cross-platform visualiser released initially in 1993. *Cthugha* calls itself “an oscilloscope on acid” and may have been the forerunner – either in inspiration or possibly even as source-code – of the numerous and varied visualisers that followed. In 1997 *Cthugha* was extended to three dimensions, which we show in figure 15.

One of such visualisers is Ryan Geiss’ *milkdrop* (Geiss (2013a)) which was featured in the popular WinAmp music player (Geiss (2013b)). Released originally in 2001, *Milkdrop* is able to generate a myriad of fluid psychedelic effects. It operates with a beat-detection algorithm coupled with *presets*, which consist of script files describing the variation of colour in every frame, according to the input audio signal. These files can describe effects ranging from simple particle effects to intricate fluid and fractal-like shapes and terrain. They contain sections akin to fragment shader code, in the sense that a colour value is generated for every pixel of every frame.

In 2007 *Milkdrop2* was released featuring support for hardware accelerated fragment shaders which allowed to push the possibilities even further.

Due to its vast versatility, *Milkdrop* remains popular to this day with a dedicated community who continue to re-invent presets, exploring the endless possibilities. Originally implemented in Direct3D, *Milkdrop* has also been re-implemented in OpenGL and is also available as a mobile application under the name *ProjecM*.

Overall, *Milkdrop2* is able to produce an endless variety of effects (as illustrated on figure 16). Due to being rooted on beat detection and constantly changing of presets, its connection to music, from a synchronisation standpoint, becomes particularly evident and engaging in music genres with fast tempos, strong beats and pronounced bass lines, such as trace and other electronic music. Other types of music, such as classical or jazz may

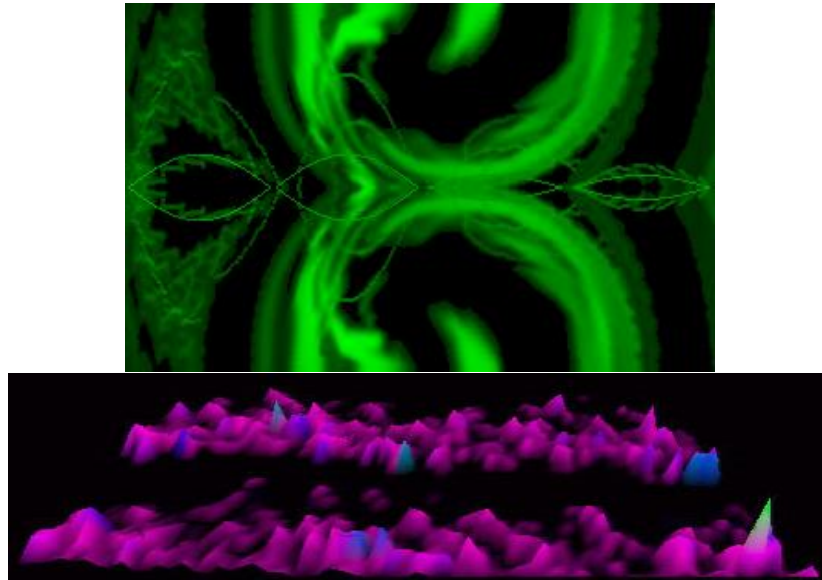


Figure 15: Cthugha Music Visualiser

occasionally synchronise but in general produce poorer displays, where randomness is mostly uncontrolled.

Also noteworthy is the G-force visualisation plug-in<sup>3</sup>. It is a popular commercial visualiser, originally released in 2001 and featured as the iTunes default visualiser, (*Soundspectrum*). From then on, it's been under constant development, accompanying numerous live performances, of great artists such as Herbie Hancock, Journey, George Michael and Aerosmith. It has also been used in various commercial enterprises as live decoration and has recently been released as an iOS application. It is a plug-in more suitable for interactive use, as it allows for tuning of the effects with a graphical user interface, as opposed to Milkdrop's more technical approach, of programming these effects.

#### *Using Musical Data as Input*

The Animusic animation company creates entrancing digital animations deeply rooted in music ever since its founder, Wayne Lytle, made his first MIDI-based animation *More bells and whistles* in 1990<sup>4</sup>. Animusic's production pipeline involves a custom software entitled MIDImotion for deriving motion from music, which combines the output generated by several algorithms and takes into account current, past and future notes. These factors are combined to derive "intelligent", natural-moving, self-playing instruments (*Animusic Company*).

Ubiquitously mentioned in music visualisation literature is Stephen Malinkowski's *Music Animation Machine*. His work consists in what he refers to as "animated scores", that

<sup>3</sup> An image gallery is available at <https://www.soundspectrum.com/g-force/screenshots.html>

<sup>4</sup> <https://www.youtube.com/watch?v=xhwwKXLQSeu>

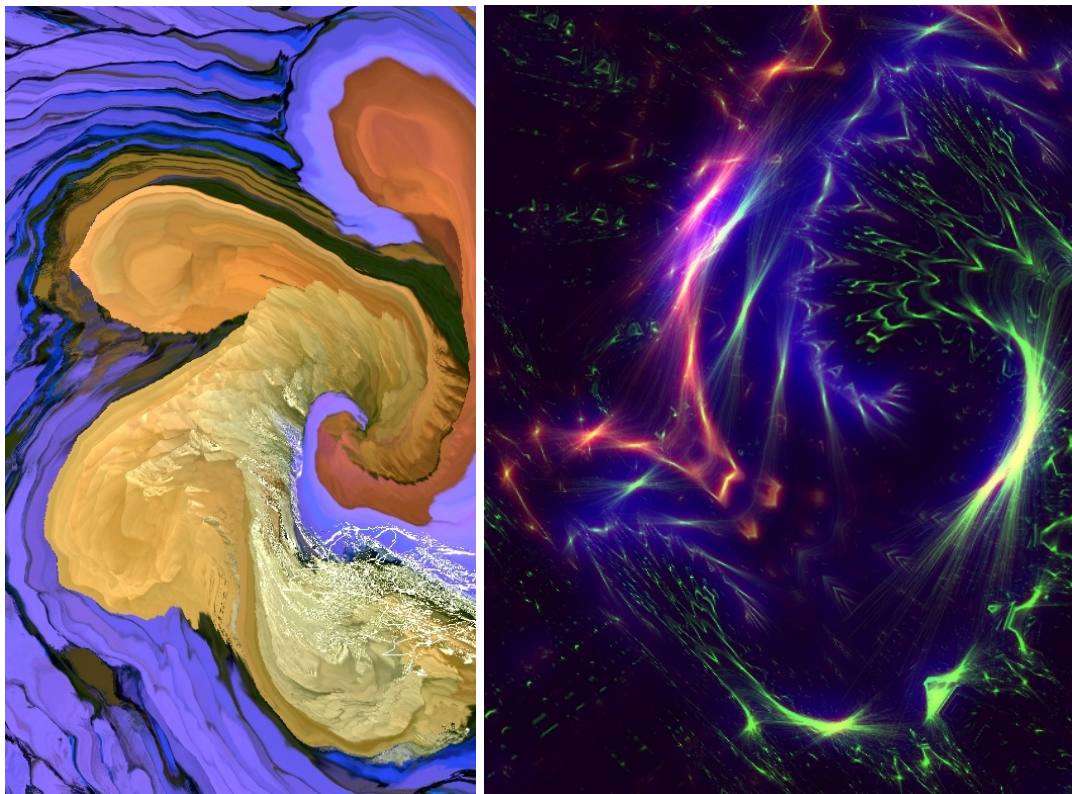


Figure 16: Frames generated by Milkdrop2

is, real-time animations of a musical sheet synchronised with a real performance. His animations resemble the piano-roll representation of music, where the pitch is mapped to height and time is mapped to width, with animated transitions inspired by the musical piece in question and colours based on musical structure. His work dates back to 1985 and has been in constant improvement and exploration since. The piano roll part of the animation is generated automatically via custom software, however, transition effects are added manually. Synchronisation between audio and MIDI files is performed in a semi-automated way by manually setting key-points. A vast library of animations has been produced and is available on Stephen Malinkowski's YouTube channel [smalin](#).

In a more automated approach, [Wood-Gaines \(1997\)](#) described a system to model the expressive movements of performing musicians. The case study was that of animating a 3D virtual drummer according to drum-tracks present in a MIDI file. The primary motions were generated using kinematics based on MIDI information. Motions could be later customised by an animator in order to convey the dynamics and expression of the piece being played.

Another art form that is very relevant to the visualisation of music is dance. As such, there have been some efforts in the generation of motions and choreography for virtual dancers. In [Perlin \(1995\)](#), for instance, pseudo-random noise is used as the driving force for generating motions for a virtual human dancer, coupled with constraints to prevent impossible body movements.

[Cardle et al. \(2002\)](#) present an approach to interactively edit motion curves with cues from musical information. The key to this approach is the use of music analysis techniques on both MIDI and corresponding audio signals to automate the generation of synchronised musical animations. These two sources complement each other, as extracting note information from pure sonic data is a very difficult task as opposed to MIDI, from which it is trivial. On the other hand, MIDI does not encode any sound information whatsoever, so timbre and dynamics information is lost. The approach consists in taking pre-defined motions, created by an animator or resulting from motion capture, which are then able to be algorithmically emphasised according to musical cues, such as chord changes, repeating patterns or variations in loudness.

[Shiratori et al. \(2006\)](#) define an approach for synthesising dance performance matched to input music, based on the emotional aspects of dance performance. A database of motion captured dance movements is used and processed. This step segments motions into sets of smaller sections, to which is assigned a measure of *effort* based on a cumulative displacement of the various limbs. This value is used to estimate the intensity and rhythm of the motion. The audio signal is also segmented based on the identification of repeated sequences of notes, which themselves are obtained through spectral analysis. Intensity and rhythm information are extracted as well. Finally, a sophisticated matching is made, by



assigning motions to musical sequences, assuring connectivity and producing remarkable results.<sup>5</sup>

In Taylor et al. (2005) a system was built to enable a musician to interact with life-sized virtual characters within a virtual environment. The system's goal was to enhance the experience of live musical performances by having a virtual character reacts in real-time to the music being played. The Animus framework presented in Torres and Boulanger (2003) was used, which specialises in creating believable characters that behave and react expressively to different stimuli. The original system was comprised of three layers responsible for creature behaviour: perception; cognition and expression. In short, the perception layer is responsible for sensing stimulus, which is propagated to all creatures using a *blackboard* approach. The cognition layer handles high-level cognitive processes, namely temporal memory, personality, emotions and goals. Finally, the expression layer is the animation engine in charge of showing the inner state of the creature to the audience. In this project, a fourth layer was added, for the extraction of musical features from a microphone and a MIDI-enabled Keyboard. This layer was implemented in Max/MSP and obtained attributes such as the vocal pitch and amplitude harmonic spectra of the user's voice as well as chords being played on the keyboard. The information is then forwarded to the perception layer for processing. There is, however, a great deal of manual work involved, since it is the designer who establishes the musical features characters are attentive to in the virtual world's *blackboard*, as well as choosing which animated poses to translate between based on the character's evaluation of the musical input, and also how sharply or smoothly these transitions should take place based on the general mood of the music. The main limitation identified was the need for the cognition layer to be very sophisticated in order to simulate an emotional understanding of music, as well as the need for a large number of poses and expressions to react accordingly.

Sauer and Yang (2009) enable the interactive creation of Celtic dance motion using information extracted directly from audio signals, namely tempo, beat onsets, and dynamics. The system relies on a pre-defined set of dance moves which are combined to create dance routines. A script file is used to establish the sequence of desired moves, which are then automatically timed and adjusted according to the musical information extracted.

The authors in Ng et al. (2014) have developed a system for audio-visual mapping for the accompaniment of live orchestral performances. Different colour mappings were implemented based on the synesthetic experiences reported by renowned composers, such as Messiaen, Ligeti and Sibelius, as well as recent neurological studies on synaesthesia. A flexible score-following interface was implemented to ensure synchronisation with the music sheet using a microphone, allowing, as well, for manual adjustments in real-time by the conductor. The music score was interpreted in Music21 representation and allowed for

---

<sup>5</sup> [https://www.youtube.com/watch?v=\\_tSoLONjAJg](https://www.youtube.com/watch?v=_tSoLONjAJg)

manual selection of important events and structures within the composition. The final projected visualisation was generated by [Processing](#) and custom stage lighting controls were also generated.

AVVX ([Correia \(2013\)](#)) is a tool for live visual and audiovisual performances designed to explore music's connection to abstract visual effects. The work of both Fishinger and Whitney is cited as inspiration. AVVX is browser-based and allows the composition of animations based on geometrical shapes with input from [Scalable Vector Graphics \(SVG\)](#), a widely used representation for vector images. Programming behaviours are made possible by javascript. The use of open technologies such as [SVG](#) and javascript ensure cross-platform compatibility and create an easily accessible, and free environment.

#### 2.4.2 Focus on Musical Analysis

A significant portion of current research into generating music visualisations in an autonomous or semi-autonomous way has been conducted with educational or analytical purposes in mind. As such, several software implementations exist which aim to reveal or make apparent various aspects of music's internal structure, such as chord progression and recurring motifs. Although most of these systems aim to be aesthetically pleasing, this is generally not their main purpose and as such, they have different fundamental goals from our project. However, they are relevant by providing valuable insight into what information is possible to computationally extract from music, and how to map it in a useful and relevant way to visual aspects.

In [Wattenberg \(2002\)](#), the author introduces *arc diagrams*, a visualisation method for representing sequence structure by highlighting repeated subsequences using a string matching algorithm. These patterns are visualised through translucent arches that connect the repeating sections. The diagram is applicable to any sort of data which can be encoded as text, such as DNA, however, they have been shown as particularly useful in representing the recurrent structure in music, using MIDI data as input. The results are available in [Shape of Song](#).

ImprovViz ([Snydal and Hearst \(2005\)](#)) is another visualisation method targeting music students, particularly jazz students. The outcome is a diagram which brings to light the signature patterns of a jazz musician's improvisational style. ImproViz consists of two parts: *melodic landscapes* which show the general contours of musical phrasing and *harmonic palettes* which represent the musician's tendency to use a particular combination of notes in a given part of the song. The approach is algorithmic, however, it was presented as a manual process. In spite of this, valuable insight is gained from the resulting diagrams which possess both informative and aesthetic properties.

The IsoChords framework, presented in Bergstrom et al. (2007), aims to visualise musical structure. It does so by conveying information about interval quality, chord quality, and the chord progression synchronously during playback of digital music. MIDI is used as input and the main goal is to assist music theory students by representing the evolution of chord progressions throughout a song. Notes are represented in triangular isometric coordinate grid invented by Euler called *Tonnetz*, which allows for intervals within a chord to be represented by connected adjacent nodes, following a predetermined pattern.

MuSA.RT 2 (Chew and Francois (2005)) is an interactive multi-scale music visualisation system that tracks and displays the trajectory of the tonal content and context of music in real-time. It applies principles defined in Chew (2000) and is the culmination of extensive research. In this context, the purpose of tonal visualisation is to reveal pitch structures in the music, including the pitch classes present, the chords, and the keys. It employs the spiral array model which consists of a collection of nested spirals in three-dimensional space, each comprising of a type of tonal object. Similarly to the tonnetz grid, distance is mapped to the length of intervals.

MoshViz (Cantareira et al. (2016)) is one of the most recent efforts in the visualisation of musical structure. A key feature lies in creating a high-level model of the musical data and highlighting aspects of interest, which enables a detail+overview interpretation. The software focused on analysing guitar solos. To this effect, some metrics are introduced to measure aspects such as note density (notes played by unit of time), "stability" as a result of unpredictable note and rhythm variation and "complexity" as an estimation of learning difficulty.

### 2.4.3 Audio and Visualisation Software

#### *Pure Data*

**Pure Data** is an open source visual programming language created and maintained by Miller Puckette. Pd is used to process and generate sound, video, 2D/3D graphics, and interface sensors, input devices, and MIDI. Algorithmic functions are represented in Pd by visual boxes called objects placed within a patching window called a canvas. Data flow between objects is achieved through visual connectors called cords. Each object performs a specific task, which can vary in complexity from very low-level mathematical operations to complicated audio or video functions such as reverberation, FFT transformations, or video decoding.



*Max/MSP*

One of the popular software packages among multimedia artists is *Max/MSP*. It belongs to the same family as Pure Data and was originally authored by the same developer. It is designed with live performance in mind and as such, allows for easy interfacing with external hardware such as MIDI keyboards and synthesisers, as well as stage lighting. Like Pure Data, Max adopts the paradigm of visual programming so as to allow for flexible modifications of running programs in real-time. Regarding visual effects, Max comes bundled with the Jitter package which allows for low level 2d/3d graphics programming as well as multimedia playback.

*VDMX*

*VDMX5* is a program that allows for assembly of custom real-time video processing applications. It is advertised as VJ software and is for exclusive use on the Mac platform. Its focus is on video manipulation from multiple sources with the addition of custom effects and filters. Supports projection mapping as well as custom shading operations using the ISF format.

*Resolume Avenue*

*Resolume* is yet another commercial VJ software which focuses on the real-time generation and improvisation of live visuals. It supports Projection Mapping and DMX lighting.

*Other Software*

There are several alternatives to the aforementioned software, namely:

- *vvvv*
- *Avmixer pro*
- *Arkaos GrandVJ*
- *Modul8*

2.4.4 *Demoscene*

The *Demoscene* is an international computer art community that creates digital art in the form of *demos*: small, self-contained computer programs which generate audio-visual clips, rendered in real-time. It has its roots in the late 1970s when affordable computers appeared in stores and were for the first time in history available to the masses (*Reunanen and Silvast (2007)*).

The beginning of the Demoscene is believed to be due to the advent of software cracking. Cracker groups illegally distributed games and other software, which was often coupled with digital "signatures", small executables containing digital presentations. Eventually groups started competing for the best presentation which led the making of stand-alone demos to become a specialised field on its own, and eventually split from the cracker culture to become what is now called the Demoscene (Reunanen et al. (2010)).

A small community, estimated 10 000 members, exists until this day who display their demos on regular events. Demos are evaluated on a multitude of aspects, such as technical proficiency, graphical effects and music synchronisation.

Demos often resemble the aesthetic of experimental (electronic) music video clips, since both of them combine electronic music and a flow of abstract imagery but often without a valid plot line. Additionally, demos employ basically the same programming techniques as computer games, although they remain intentionally non-interactive so as to preserve the synchronised visual and sound experience, which is considered an important part of the creation itself (Doreen (2010)).

A key aspect of demos is a self-imposed limit on the file size of the executable. At the beginning, this limitation was due to actual hardware constraints. Now it's kept not only because of tradition, but mostly as a means to boost creativity.

## 2.5 PROCEDURAL GENERATION

Procedural generation refers to any use of a formal algorithm to generate data algorithmically. Its initial application for creating digital content dates back to computers games from the early 1980's and new methods are actively developed and researched to this day, spanning numerous disciplines, ranging from computer graphics and artificial intelligence to psychology and linguistics.

Several different game aspects can be procedurally generated, from concrete aspects, such as levels, buildings and vegetation, to more abstract aspects, such as behaviour and story. As a direct result, methods for generating such content vary greatly and are usually not general purpose (Hendrikx et al. (2013)).

A prominent example is *SpeedTree*, a group of commercial vegetation programming and modelling software products that generate virtual foliage for animations, architecture and in real-time for video games.

The concept, and application, of procedural generation techniques goes beyond creating game content. For instance Ebert et al. (2000) applied procedural shape generation techniques, namely fractal detail generation, superquadrics, and implicit surfaces for multi-dimensional data visualisation with promising results.

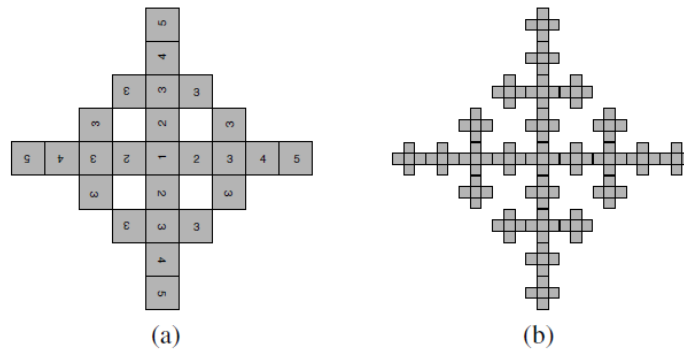


Figure 17: Discrete branching pattern by Ulam: (a) simple pattern with the illustration of different generations; (b) more complex branching pattern (Deussen and Lintermann (2005)).

For our work, however, we focused our investigation on methods for the generation of plant life.

### 2.5.1 Plant Modelling

Computer modelling of plants has a long history spanning over 50 years. Many approaches have been developed for different purposes. Biology is interested in plant models as a means for better understanding the fundamental mechanisms that govern plant development and structure. Models may also be used in computer-assisted decision-making in horticulture, agriculture, and forestry. On the other hand, the computer graphics community is interested in plants as elements of scenery for computer animations and games (Deussen and Lintermann (2005)).

Computer-assisted simulation of natural growth processes was introduced as early as the 1960s during the time computers became more and more available to researchers. *Cellular Automata* marked the first step and were formalised by John von Neumann in the 1950s while trying to develop an abstract model of self-reproduction in biology. These models were employed by Stanislaw Ulam in Ulam (1962) for the generation of growth and branching patterns (figure 17). The first continuously growing plant model with a branching structure was presented in Cohen (1967) (figure 18).

Early models of plants were based on procedural approaches that replicated growth by repetitive application of a small set of rules to an initial structure, yielding complex results. Honda (1971), described a set of three-dimensional tree topologies using a small number of parameters. By changing numerical parameters, Honda obtained a wide variety of tree-like shapes and was later able to apply his models to investigate the branching of real trees (figure 19). In Aono and Kunii (1984) Honda's model was extended in several aspects, particularly in allowing for different branching patterns as well as taking thickness

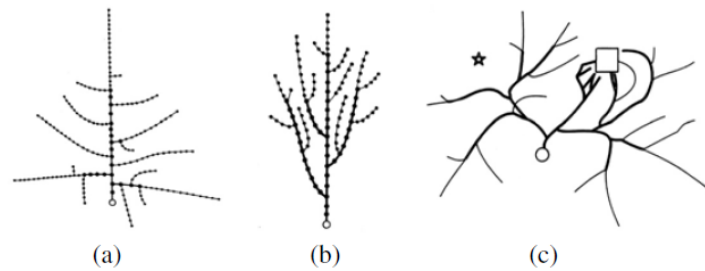


Figure 18: Continuous branching structures by Cohen: (a) branching restriction introduced by minimal distance to end of branches; (b) vertical alignment of growth; (c) use of an attractor (square) and of an inhibitor (star) (Deussen and Lintermann (2005)).

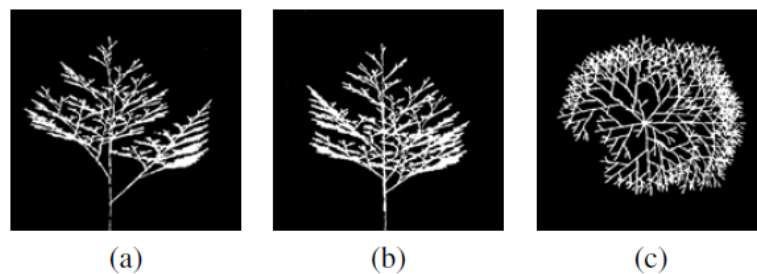


Figure 19: Branching structure by Honda: (a) Projection to  $xz$ -plane; (b) to  $yz$ -plane; (c) to  $xy$ -plane (Deussen and Lintermann (2005)).

into consideration. Aono and Kunii continuously extended their model, for instance by introducing attractors and inhibitors, which bend the models and simulate the influence of wind.

Around the same time, the first approaches with focus on visual realism were developed. Bloomenthal (1985) focused on modelling the maple tree, also resorting to a parameterized recursive algorithm for generating tree topology. Rendering was performed through generalised cylinders along a spline for shaping the branches. Smooth branching discontinuities are achieved by using a "ramiform" primitive and roots are rendered using "blobby" techniques (figure 20b). The bark texture was obtained by scanning a plaster cast of actual bark, as were the leaves. The results are remarkably realistic (figure 20a). Oppenheimer (1986) used fractal techniques for generating tree topology and focused on optimisation, achieving real-time performance. (figure 21). Moreover, he applied the same techniques for generating other fractal objects in nature, such as snowflakes and fern leaves.

An approach oriented on botanical growth was pioneered by Reffye et al. (1988). The authors simulated the growth of the shoot axes in discrete time steps from node to node, by modelling the activity of buds at discrete time intervals. Each bud carries several probabilities: the probability of dying, the probability of resting, and the probability of branching

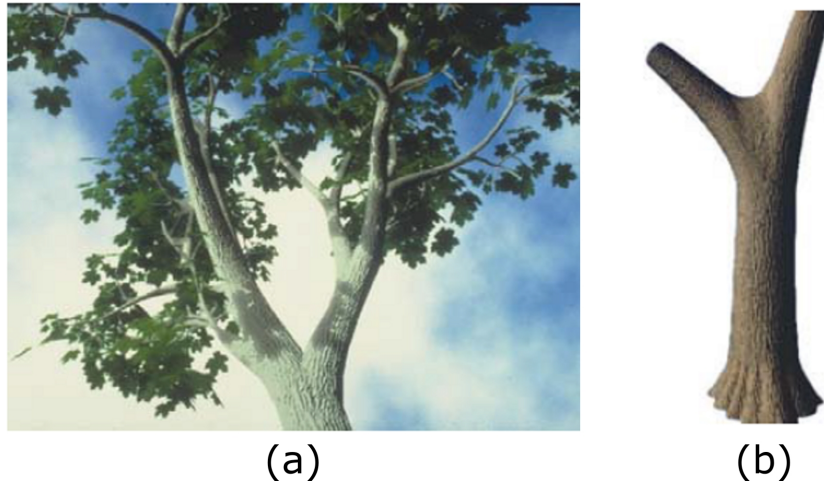


Figure 20: (a) Bloomenthal's model of the maple tree  
(b) The "ramiform" primitive allows for smooth branch junctions (Bloomenthal (1985)).



Figure 21: Oppenheimer's fractal models (Oppenheimer (1986)).





Figure 22: Greene's Voxel Space Automata (Greene (1989)).

out. This allows for modelling based on real data, thus characterising different species. In Greene (1989) the concept of Cellular Automata was extended to three dimensions and successfully applied to the simulation of climbing plants in voxel space, reacting to their environment by estimating light density in the scene. The remarkably realistic results are shown in figure 22.

Weber and Penn (1995) propose approximate, though realistic-looking solutions for tree modelling. Their procedure requires a set of approximately 50 parameters, all of which are described in their work. Methods are also presented for the interaction of the models with wind and a level-of-detail representation.

A distinct method for the modelling of plants is the usage *L-systems*, the most developed formal approach for botanical plant modelling. In Lindenmayer (1968) the concept of string rewriting was applied to the description of cellular interactions, which later evolved into the L-System framework. String rewriting is a formalism to describe transformations of a given character string throughout various iterations, according to a fixed set of rules. These rules map characters or sequences of characters to their replacements and are simultaneously applied in every iteration of the rewriting algorithm. Moreover, each character in this string can then be interpreted as a drawing command, using what is referred to as turtle geometry<sup>6</sup>. A virtual turtle is able to move and change its orientation, and as it moves a line is drawn representing its path. L-Systems assign characters to each of these commands.

Due to their intrinsic recursive nature, the simplest L-Systems, so-called DOL-systems (deterministic and context-free), were found to be suitable for modelling fractal objects, such as the Koch snowflake (figure 24) and space-filling curves such as the Hilbert curve (figure 23) (Prusinkiewicz and Lindenmayer (1990a)).

For the actual modelling of plants, however, the system had to be extended. This is due to the fact that branching structures cannot be approximated by a single line, but

<sup>6</sup> Made popular by the LOGO language, this metaphor arises since turtles are said to move similarly: if an initial position and direction are given, they usually move in a straight line, until a change of direction becomes necessary.

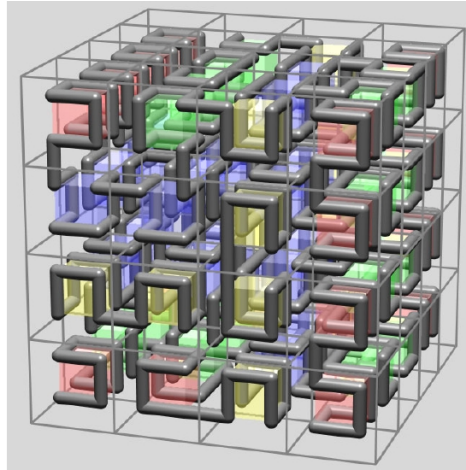


Figure 23: Hilbert curve in three dimensions (Prusinkiewicz and Lindenmayer (1990a)).

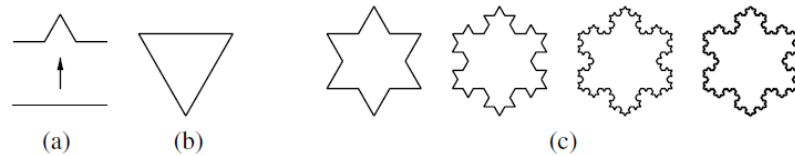


Figure 24: Koch Snowflake: (a) generator; (b) initiator; (c) illustrations after 1, 2, 3 and 7 rewritings (Deussen and Lintermann (2005)).

instead requires that individual limbs be drawn separately. In order to accomplish this, using the previously defined mechanism, it is necessary to keep track of positions where discontinuities arise, such as branching points. This can be elegantly implemented using a stack structure which stores and loads previous states of the drawing turtle.

In figure 25, we have a rendering of a basic tree built on two simple rules, illustrated in the top left corner of the image. The system went through 4 rewrite operations until reaching the last image.

The first rule states that an apex (i.e. a terminal branch segment) yields a branching structure consisting of two internodes: the apex continuing the main axis and two lateral *apices*. The second rule states that, over the same time interval, the internodes will elongate by a factor of two. In each step, all apices and internodes are subject to their respective rules, applied in parallel.

From these examples, we hope to convey that L-Systems provide an elegant way of describing complex structures. These are some of the simplest L-Systems and many extensions have been developed since. Namely, Parametric L-Systems (individual symbols have parameters associated), Stochastic (non-deterministic application of productions), context-sensitive (production application depends on neighbouring sequences) and Open L-Systems (able to interact with their environment). The support for real-time animation is presented

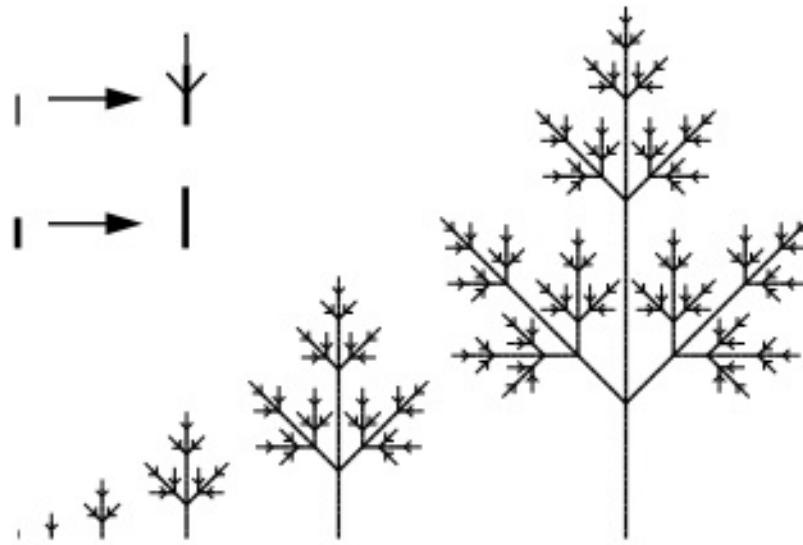


Figure 25: A simple branching structure Prusinkiewicz and Lindenmayer (1990a).

using differential equations in Prusinkiewicz and Lindenmayer (1990b) with remarkable results.

The framework is still being improved to this day (Boudon et al. (2012)) and L-Systems have proven useful beyond the botanical modelling of vegetation. For instance in the creation of game levels (Hendrikx et al. (2013)), music (Pestana (2012), Manousakis (2006)) and inspiring the CGA shape grammar for modelling of buildings (Müller et al. (2006)) which is now part of the CityEngine software.

A different approach focused on interactive modelling was originally proposed in Lintermann and Deussen (1999). Here a set of components were developed to represent different elements in plants, such as stems and leaves, as well as rules for multiplying and positioning them. These components are manually connected in a graph structure, which is transversed to generate the final geometry. This is the approach in the Xfrog Software, a commercial platform for modelling of natural environments. A survey and comparison of tree modelling methods can be found on Deussen and Lintermann (2005).

Also present are image-based techniques, which produce tree models from sets of images. For instance, in Neubert et al. (2007) an approximate voxel-based tree volume is estimated and the density values of the voxels are used to produce initial positions for a set of particles. Performing a 3D flow simulation, the particles are traced downwards to the tree basis and are combined to form twigs and branches. Also possible is the reconstruction of tree-models from *Light Detection and Ranging (LiDaR)* data, namely point clouds, as in Livny et al. (2010). In Livny et al. (2011) the authors present a lobe-based tree representation for





Figure 26: Synthetic landscape with self-organising trees (Palubicki et al. (2009)).

the modelling of trees which was found to be adequate for reconstructive processes. Sketch-based generation has also been explored in Okabe et al. (2005) and Chen et al. (2008).

The *Space Colonization Algorithm (SCA)* presented in Runions et al. (2007) derives tree shape from available space and will be detailed in 3.7.1. Essentially, the algorithm operates by defining the region where the tree will grow using discrete points in space. The tree skeleton will develop iteratively while being attracted to and extended towards nearby points. This algorithm was our choice due to its elegant simplicity and versatility. By arbitrarily defining a region of space through individual points any tree shape and size is achievable. There is a small number of parameters which all have an intuitive impact on the overall tree shape. This makes the method suitable for artistic use. In fact, this algorithm has been implemented as a plug-in for the Maya modelling environment (*Grower Maya plugin*).

In this algorithm, the tree develops in a botanically natural order starting at the root. This makes the algorithm suitable to animate real-time growth which was one of the effects we more thoroughly explored.

An extension to this method is presented in Palubicki et al. (2009) in which the process is further automated by generating plant shape as a result of a self-organising process (see figure 26). This process is dominated by the competition of buds and branches for light or space and is regulated by internal signalling mechanisms. This approach was later employed to allow for the interactive creation of trees using free-hand sketches, as described in Longay et al. (2012).

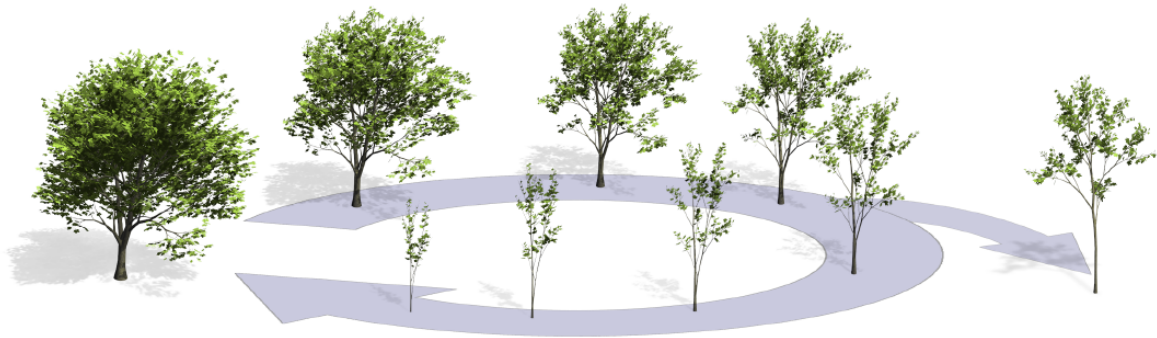


Figure 27: The static tree model on the left is converted into a developmental model that encompasses the ability to create arbitrary intermediate stages between a very young model and the given geometry (Pirk et al. (2012a)).

Pirk et al. (2012b) present a method for adapting pre-existing tree models to dynamic changes in their environment, allowing for an efficient interactive construction of complex scenes with multiple trees. Furthermore, Pirk et al. (2012a) propose a sophisticated method to compute developmental stages that approximate the tree's natural growth (as shown in figure 27), allowing for the interactive creation of animations. The method takes as input an adult tree and is able to infer positions and thickness past branches, taking into consideration different growth rates when creating the animation.

## 2.6 TREE RESPONSE TO WIND

Wind is air in motion. It is produced as a response to pressure differences within the atmosphere. The wind's velocity results from the pressure difference between an area of high pressure and an area of low pressure, with greater differences producing higher velocities. Wind velocity at a given point can be represented as a vector quantity and as such is characterised by its speed and direction.

Wind can be characterised as a viscous fluid, and as such can be described by the Navier–Stokes equations. These equations describe how the velocity, pressure, temperature, and density of a moving fluid are related. More formally the Navier–Stokes equations are a set of coupled differential equations which could, in theory, be solved for a given flow problem by using methods from calculus. But, in practice, these equations are too difficult to solve analytically. The use of computers is then common, and techniques used to derive solutions to this set of equations is collectively referred as *Computational Fluid Dynamics (CFD)* (NASA Navier Stokes Equations).

In practice, these equations are remarkably useful since they describe the physics of many phenomena of scientific and engineering interest. They may be used to model the weather, ocean currents, water flow in a pipe and air flow around a wing. The Navier–Stokes equa-



Figure 28: The largest model presented [Quigley et al. \(2017\)](#), composed of over 3 million articulated rigid bodies.

tions in their full and simplified forms help with the design of aircraft and cars, the study of blood flow, the design of power stations, the analysis of pollution, and many other problems.

Physical models for wind animations are commonly based on computational fluid dynamics. Several methods exist to integrate the flow field, however, only a small number of previous work employs them to animate tree and plant models ([Pirk et al. \(2014\)](#)).

The most physically accurate way to simulate a tree's response to wind would be to use many small volumes and apply the finite element method. However, this is unfeasible in practice since it would require a vast number of elements as well as bio-mechanically accurate constitutive models ([Quigley et al. \(2017\)](#)).

Noise-based solutions, such as [Ota et al. \(2004\)](#) and [Shinya and Fournier \(1992\)](#) were also considered, however, we required the tree to be able to respond to a controllable wind-field and not just engage in arbitrary, albeit natural looking motions. More sophisticated methods exist such as [Habel et al. \(2009\)](#), which take into account the deformation of individual branch segments.

[Quigley et al. \(2017\)](#) present a remarkably optimised approach for simulating trees, including organs such as leaves and fruits. By approximating trees as systems of rigid bodies articulated by stiff joints the authors were able to use analytic solutions to the spring dynamics equations. This allows for robust and efficient treatment of joint springs which avoids the cost of numeric integration with small timesteps (see figure 28).

In [Sakaguchi and Ohya \(1999\)](#) a segment based approach is described. Mainly employed until then as a method of a dynamic simulation for string-like objects such as lawns and hair, the authors successfully applied this segmented model to trees. Essentially a string-

like object is defined as a set of connected rigid segments, and the dynamics of the object are obtained by calculating the movements of each segment individually and combining the results at a later stage. This approach is used by recent works such as Pirk et al. (2014) and Oliapuram and Kumar (2010).

This was the approach we selected due to being physically based, thus providing realistic results, and yet remains lightweight and relatively simple to implement. It was also particularly suited to our project due to the fact that the input for the method matches our segmented representation of the tree obtained from the SCA. Further detail is presented in section 3.8.

The last element remaining for defining our simulation is the creation of the **wind field**, which is by itself a challenging research topic. Unlike Pirk et al. (2014) and Oliapuram and Kumar (2010), we did not employ CFD techniques to approximate the wind field. We took an approach closer in complexity to Sakaguchi and Ohya (1999), who defined wind using a manually crafted voxel grid which assigns wind velocity vectors to each region of space. Our selected approach automates this process and was based on the method presented in Wejchert and Haumann (1991), which defines a set of continuous *flow primitives* which can be easily combined to generate complex wind flows, assuming an ideal physical fluid. This will be further detailed in sections 3.1.2 and 3.10.

---

## TOWARDS PROCEDURAL MUSIC-DRIVEN ANIMATION

---

The problem we face consists in allowing for the flexible generation of animations strongly choreographed to music. In order to make this possible, configurable algorithms for procedural generation are necessary, as well as input for musical information. To join these two parts a versatile strategy to create mappings between them is also required. Real-time performance was also considered so as to allow for the possibility of live accompaniment for performances.

Given that music is the result of artistic effort, we hypothesise that the generation of any accompanying visual effects should not be derived entirely from algorithmic processes. Instead, such algorithms should provide flexible and reactive building blocks from which animations can be built.

Most music is structured around a fixed tempo and time signature, and cycling chord progressions. Such patterns occur in all levels of music, including melody, rhythm, harmony, and texture (Cardle et al. (2002)). Traditional *keyframing* animation, which is based on the concept of manually drawing every frame, becomes prohibitively costly and possibly inaccurate when attempting large-scale synchronisation with a musical piece (Animusic Company). The fact that such animation has to be created in advance also makes it unsuitable for live accompaniment. These were two of the main motivations behind the development of our framework.

Since any pre-defined mapping of musical to visual attributes would be arbitrary, what we strive to accomplish is to allow for any mapping to be possible, within the scope we have defined. For this, we found that the use of a scripting language provided not only flexibility but also the freedom to create further abstractions, which aid in specifying more complex behaviours.

What we hope to demonstrate is the possibility of unlocking artistic possibilities and empowering creativity by supplying a high-level interface for dealing with a real-time stream of musical information. In particular, allowing it to be projected onto procedural generation algorithms.

Our system can be described as a digital choreography assistant based on musical score, in the sense that events in the animation can be timed to different sections of music, as if



they were score markings, not unlike the rhythmic and dynamics annotations we briefly discussed in section 2.1.6.

### 3.1 PROPOSED APPROACH

In order to bring our idea to life, a proof-of-concept implementation was developed.

Our system has at its core a procedural generation algorithm, which iteratively produces a growing tree. Additionally, this tree reacts in real-time to its environment, particularly wind conditions and available space. Its growth may also be biased by the application of arbitrary tropisms. The algorithms used to implement these behaviours are described in sections 3.7 and 3.8. Furthermore, we require a stream of musical information to provide us with, for instance, precise timing regarding the occurrence of musical events which are, in its simplest form, the beginning and the end of notes. Different representations of what we collectively refer to as musical information can be found in 2.2.

The usage of musical features is one of the aspects which distinguishes our approach from most visualisation attempts. The main advantage is that some musical cues are not estimated from audio data but in fact known in advance or received in real-time in the case of a live performance<sup>1</sup>.

Both wind and tropisms, as well as all other parameters, such as colour or leaf growth, can be modified in real-time. All modifications are directed by a script file and, in the examples, we have developed, are usually triggered at particular moments in the song, so as to provide a sense of synchronization and attempt to match the expression contained in the music.

To generate an animation, three elements are then necessary: an audio source, a music information source and a script file.

Regarding audio, our system has support for most common file formats, namely ogg, wav and aiff<sup>2</sup>. Ideally, the audio source would always consist in one of such audio files (or live audio). However, it is difficult to obtain audio files for which synchronised musical information is readily available. To solve this problem two practical solutions were identified: one could manually create this information with specific software by transcribing the music or synchronising another transcription; one could work directly with the artist who would supply said music information. A third solution also exists which is to extract this information from the recording. This, however, is far from a simple task and is still under active investigation. In particular, for polyphonic music where the components of various musical tones interfere with each other and intermingle this task becomes particularly difficult (Müller (2015)).

<sup>1</sup> Live accompaniment is unfortunately not yet supported

<sup>2</sup> All formats supported by [libsndfile](#)

The popular [Ableton Digital Audio Workstation \(DAW\)](#) is able to assist in this task through a semi-automated process where the user contributes by placing markers on the audio file ([Ableton MIDI](#)). Other software products claim to obtain workable results, but all with a large margin of error. Except for very simple music, the automatic conversion of a music performance into score notation by a computer and is still a largely open problem despite decades of research ([Müller \(2015\)](#)). For a collection of edited articles on this topic, we refer to [Durrieu et al. \(2010\)](#).

We chose MIDI as our primary musical information source for this project, which tackles some of the issues we have discussed so far. MIDI is a standard bearing 30 years of active use, as such not only are many MIDI files readily available online, as MIDI output is available on most keyboards or digital pianos ([Chng \(2016\)](#)), which would simplify the task of interfacing with instruments during a live performance. Additionally, it is possible to produce audio from MIDI by resorting to the operating system's audio synthesizer, which provides a convenient fallback for when there is no synchronised audio file available.

Evidently, resorting to MIDI audio as fallback yields poorer results, since synthesised sound lacks the richness of an actual instrument and a recorded set of musical events lacks the expressive variations found in human performances. In spite of this, we found that using MIDI as an audio source produces acceptable results and is able to illustrate the potential of what our system can accomplish.

Lastly, we implemented scripting with the *lua* programming language. For more details please consult [3.4](#).

### 3.1.1 *Tree growth, Interpolation and Tweens*

Tree growth was one of the earliest aspects we implemented in our system. It is made simple due to the nature of our algorithm, which generates branches in a natural root-to-tips order. One of the first challenges consisted in allowing for the continuous growth of branches over time since our algorithm only provides us with the growth of the tree skeleton in discrete steps. For details on the method, the [SCA](#), please refer to [3.7.1](#).

We can represent this problem as obtaining the length of a branch segment over time, given a minimum and maximum length. This is a specific case of **interpolation**, which is, more formally, a method of constructing new data points within the range of a discrete set of known data points.

Interpolation is of particular relevance to the practice of animation in general, where it could arguably be referred to as *Inbetweening*. In traditional animation, an artist will first draw keyframes, which represent important moments in the animation and then draw the frames in between so as to produce smooth transitions ([Falkowski \(1999\)](#)).

In our framework, it is very relevant to allow for inbetweening with a satisfactory degree both of automation and control, since we will wish to synchronize with musical events which are often characterised by a start time and a duration. One possible solution is the use of *Tweens* and *easing functions*, as featured in the popular *Adobe Animate CC* software. Our simplified implementation of this method consists essentially in using linear interpolation, whose constant rate of change is then adjusted based on a particular function with a suitable variation. However, this could be further refined by allowing the user to place keyframes, as found in the *Animate CC User Guide*.

We used this approach to control the growth of branches and by selecting one of several predefined ease functions<sup>3</sup> we were able to subjectively adjust the growth animation according to its context, such as a note's particular duration. As the project development continued we found tweens to be a flexible enough solution to control almost every aspect of our animation. More detail is provided in section 3.5

### 3.1.2 Wind field generation

Wind is the motion of air, and as such, it can be simulated as a moving fluid. Mathematically, the state of a fluid at a given instant of time is modelled as a velocity vector field: a function that assigns a velocity vector to every point in space. In the simplest case, this field can be generated as a single vector quantity, which will then result in a constant, uniform and one-directional wind.

Recent literature on tree generation and wind-response simulation, such as *Pirk et al. (2014)* and *Oliapuram and Kumar (2010)*, has generated wind fields by solving the Navier-Stokes equations. To this end, *Pirk et al. (2014)* use *Smoothed Particle Hydrodynamics (SPH)* while *Oliapuram and Kumar (2010)* employ a discrete solver as described in *Stam (1999)*. Both these approaches offer some degree of control regarding the wind field. *Pirk et al. (2014)*, in particular, allow for the arbitrary placement of "wind emitters", which are 3d rectangles from where wind flow originates. *Oliapuram and Kumar (2010)* allow for control by exerting external forces upon the wind field.

Our priority regarding the wind field is, however, fundamentally different from these works since both strive for physical correctness, while for the generation of animation the requirements are not so strict. *Pirk et al. (2014)* focus on the combination of developmental tree models with complex wind fields and *Oliapuram and Kumar (2010)* focus on the optimised real-time animation of entire forests. In our work intuitive user control of the animation is the main goal, so we required a mechanism which is both easy to use (and precise) for specifying the wind field.

<sup>3</sup> Visit <http://easings.net/> for a gallery of interactive ease functions



As a result, we opted for the approach described in Wejchert and Haumann (1991). The concept of **uniform flow** is used to simulate wind and a set of **flow primitives** is defined. Each of these can be regarded as a building block of flow: **uniform flow**, which flows in a given direction; **sink/source flow**, which flows towards/away from a given position and lastly **vortex flow**, which flows around a given axis. Each of these primitives is also characterised by a strength attribute. The final velocity field is obtained from the combination of any number of these primitives. This method was particularly suited to our system due to the fact that these primitives' attributes, for instance, **strength** (which is common to all primitives), can be easily modified in real-time. This gives a further degree of control, by allowing arbitrary changes to any primitive to occur at specific times. In theory, a primitive mechanism such as this could be implemented using the force-based approaches of Pirk et al. (2014) and Oliapuram and Kumar (2010). However, by applying this method we avoid solving the Navier-Stokes equations due to the fact that the aforementioned primitives are represented as linear equations, leading to a less numerically intensive solution. This approach is further detailed in section 3.10.

### 3.1.3 *Tree response to wind*

Our tree is approximated as a set of branch segments. As such, a large number of hierarchical relationships is formed, which together represent the tree's skeleton. Since several segments approximate a single body, the whole structure must remain connected, acting as a rigid surface. As a result, the movement of a branch segment is strongly influenced by the movements of both its parent and child segments.

Similarly to Pirk et al. (2014) and Oliapuram and Kumar (2010), we implemented the model from Sakaguchi and Ohya (1999). Our segmented representation of the tree closely matches the input of their method, and as such, it was relatively straightforward to implement. Essentially, every segment is assumed to be a rigid stick with one fixed end (the parent branch side), and only rotational movements about the fixed end are calculated. The method operates by calculating the movement of segments individually, followed by combining their positions so as to ensure connectivity.

The algorithm is described in more detail on 3.8

### 3.1.4 *Scripting*

The script module is the key binding element of this project, as it is responsible for coordinating exactly what happens within the animation and when. In short, it is what grants configurability to animations and thus establishes our project as a tool with infinite possibilities rather than a particular set of audio-visual mappings.

Similarly to the way music sheet is annotated with dynamics and rhythmic information, our scripting module allows the user to annotate sections of the music with the desired actions to perform. To do so, the script receives musical events as they occur. The most basic musical events are the beginning and ending of notes, which are currently the ones used by the framework. However, we built the system so that new events can be added, and these could be of any nature provided they can be extracted from musical information, such as key and tempo changes, starting and ending of repeating motifs or changes in articulation.

The most basic action the script can perform is the assignment of values to particular attributes. Using only a single uniform flow primitive, for instance, one can easily associate a change in wind strength to the sounding of a particular note. It is worth noting that notes are not the only elements of synchronisation, one can use time as well, for instance by changing the wind direction on every beat.

Instant changes provide a good starting point, however, it very often required in animations to gradually modify a value with a given rate. Tweens were found to be a flexible strategy for dealing with these scenarios and as such, they were made usable from our scripting environment and can control virtually any property of our animation, providing that the interpolation is adequately defined. In our case, we have interpolation available for floating point numbers, as well as simple strategies for both vectors and colour values. As such any aspect of the animation represented by any of aforementioned structures can be manipulated over time with minimal effort. This includes, for instance, all attributes of wind primitives, thus allowing for a convenient manipulation of the wind field.

One of the most useful properties of using a scripting language is the ease with which new effects can be implemented. This configuration of effects could technically have been implemented in a simpler format, such as a plain text configuration file. However, this would be more time-consuming in terms of implementation. Furthermore, embedding Lua allowed us to take full advantage of its advanced mechanisms and data structures in order to produce gradually more abstract functions which contribute to concisely express more sophisticated effects.

## 3.2 OVERALL ARCHITECTURE

In this section, we wish to give the reader a brief overview of our system's internal organisation as well as introduce the different problems our system tackles. To do so, we will present the modules and software components with the most relevant functionalities for the final product. This is not a detailed presentation of the program's implementation, but rather a discussion on how the system was divided into modules and how their communication is established.

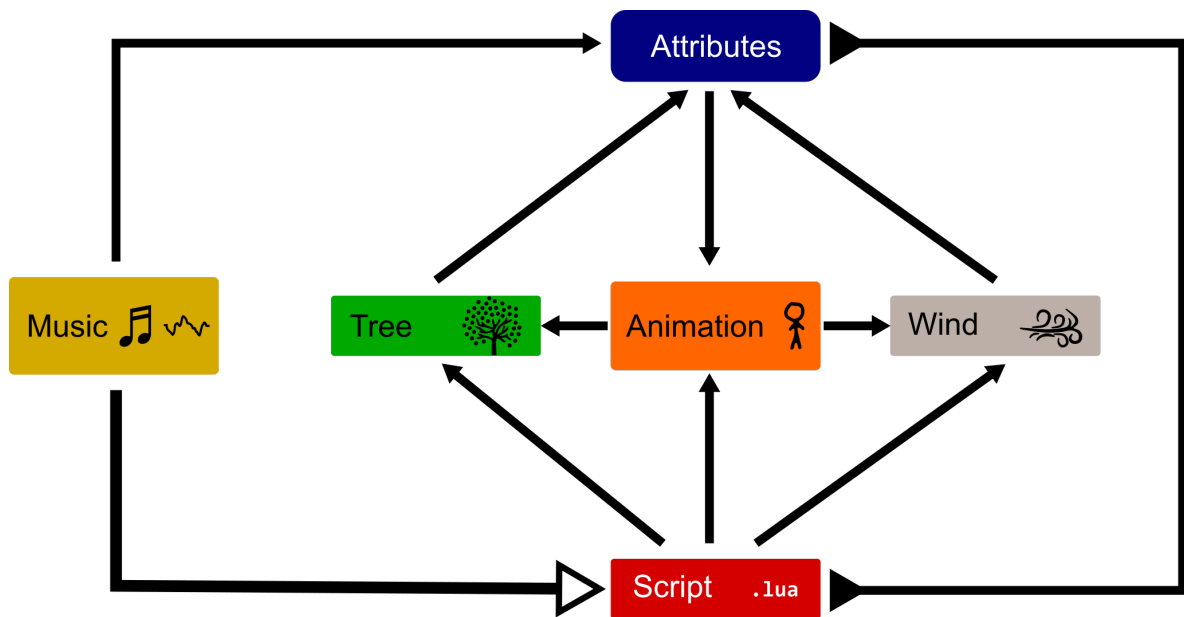


Figure 29: A broad overview of our system's organisation

The most challenging aspect of the implementation of our system is the simultaneous processing of multiple aspects which direct the animation in real-time namely, tree growth and response to wind, MIDI processing, script execution and tree rendering. Another challenge is the organisation of, and interaction between, the distinct software components carrying out these tasks while striving for extensibility and real-time performance.

We attempted to follow the *Separation of Concerns (SoC)* design principle so as to maintain our system well organised, simplifying its development and maintenance, however, due to the limited available time we do not claim to have achieved an optimal solution.

In figure 29 we illustrate the modules which will be discussed in the following sections, as well as their interaction.

### 3.2.1 *Attributes Module*

The Attributes module is the backbone of our system, as it provides a common interface for accessing every other module's functionalities. This includes both the manipulation of parameters, such as tree colour or wind strength, as well as the invocation of functions, such as for triggering the tree growth, structurally modifying the wind field or for obtaining audio features, such as the frequency spectrum.

In order to achieve this communication, virtually every module in the system is connected to the Attributes module and will, at initialisation, register the set of functions and parameters which will be accessible from it.

The Attributes module is strongly coupled with the Script module since it is exactly through this common interface that the Script module is granted access to all the features implemented by the host program.

### 3.2.2 Script Module

The Script module implements the loading and execution of **Lua** scripts. As mentioned in section 3.1.4, an animation in our system is configured through a script file, which defines what effects occur as the song progresses. To do so, each of these files must define a set of predetermined functions in order to correctly interface with our program. As a result, we have created a small *lua* library which provides the default implementations for most required functions, as well as a set of helper functions and structures, which we found to be common to most of our animation projects.

As illustrated in figure 29, the *Script* module is connected to the *Tree*, *Animation* and *Wind* modules.

The connection to the *Tree* module results in the script being able to directly modify the tree both visually and structurally, as is further described in sections 3.2.5, 3.7 and 3.9. Similarly, the connection to the *Wind* module allows the script to alter wind flow as described in sections 3.2.6 and 3.10. Lastly, the connection with the *Animation* module allows the Script module to create instances of animation primitives such as *Tweens* and *DataLinks*, which specify continuous changes in any of the aforementioned attributes and which will be further described in sections 3.2.4 and 3.5.

The Script module itself is described with more detail in section 3.4.

### 3.2.3 Music Module

The music module implements all aspects relating to music processing and playback. In particular, this includes the loading of audio files and MIDI files.

As can be seen in figure 29, the music module is strongly linked to the *Script* module. This results from the fact that our animations react in real-time to musical information, particularly note information, in order to direct the creation of visual effects. This information is provided by the Music module. This is implemented by using the timing information contained in MIDI files in order to simulate a MIDI stream. When fired, MIDI events are converted to our intermediate representation and then sent to the active script file, which will react according to the rules it defined. This mechanism is further described in sections 3.4 and 3.6.

The Music module is also responsible for the real-time analysis of audio data. At present, the only audio feature we are extracting is the frequency spectrum, as is detailed in section

3.6, which can be queried at any time during the script's execution, again through the use of the *Attributes* module.

#### 3.2.4 Animation Module

The Animation module is used to implement all effects which are not the result of physical simulation, but are also calculated based on the passage of time.

The animation module is comprised by a small set of animation primitives.

As shown in figure 29, the Animation module is connected to the *Tree*, *Wind* and *Attributes* modules. This is the case since these changes in values are, once again, mediated by the *Attributes* module (similarly to the *Script* module). Most animation primitives are bound to a particular attribute, and while active will continuously change its value according to the arguments supplied at their construction.

The most widely used animation primitive is the *tween*, as mentioned in 3.1.1. This structure is essentially a versatile strategy for two-point interpolation and is conveniently able to gradually modify any attribute exposed by the program. This includes, for instance, branch length and thickness, wind strength and leaf colour.

Another relevant primitive is the *DataLink*, which we employ to manipulate real-time data, such as the frequency spectrum and use it to alter any value available in the script environment, which includes, but is not limited to, the exposed *Attributes*.

The Animation module is further described in section 3.5.

#### 3.2.5 Tree Module

The Tree module is responsible for all aspects relating to our dynamic tree system, namely its growth, graphical representation as a set of smoothly curved limbs and reaction to wind.

The growth of the tree is implemented using the *SCA* algorithm, which iteratively extends the tree's skeleton using a cloud of points which signal space for the tree to grow into and is further described in section 3.7. The function for triggering a growth iteration is exposed to the script environment, as well as specific *tween* instances which control the gradual elongation of the segments and the growth of leaves.

The graphical representation of the tree is built by extruding a curve which interpolates all of its segments' endpoints, and is detailed in section 3.9. Visual aspects of the tree as exposed to the script environment, such as branch and leaf colour, as well as global coefficients for adjusting branch thickness and leaf size.

Lastly, the tree's reaction to wind is performed by simulating the movement of its individual segments and then combining the result. This is achieved by defining a set of forces which act simultaneously upon the tree, which leads to a final equation which is continu-

ously integrated in order to estimate the resulting motion. This simulation is detailed in section 3.8.

The tree is thus composed of several connected segments, which are also individually accessible from the script environment. Furthermore, the script is able to arbitrarily select subsets of the tree and manipulate them independently, which provides another degree of flexibility for the creation of visual effects, such as only growing part of the tree or adding leaves selectively.

Although our physics calculation method was implemented with focus on the interaction with the wind field, it is also able to react to arbitrary forces applied to any individual tree segment.

### 3.2.6 *Wind Module*

The Wind module is responsible for the generation of the wind field. As stated in section 3.1.2, the wind field is represented as a three-dimensional vector field, where each vector indicates the wind velocity at its location. We construct this vector field using the simple physical model of uniform flow, which applies a great deal of simplification to the equations that govern fluid mechanics in order to create physically plausible vector fields from linear equations. Each of these equations corresponds to a flow primitive.

Our system defines three types of elementary flow, namely uniform flow (section 3.10.1), source/sink flow (section 3.10.2) and vortex flow (section 3.10.3). As previously stated, the vector field pertaining to each of these flows is calculated by a simple linear equation. As a result they may be combined with simple addition. We detail our implementation of this method in section 3.10.

## 3.3 ATTRIBUTES SYSTEM

As stated in section 3.2.1, our attribute system is the most fundamental module of the framework, as it is what provides a common interface for accessing and manipulating the parameters of all of the described algorithms, namely tree growth, wind control and animation. It is, as such, particularly intertwined with the scripting module. We took the approach of building the system as minimal and simple as possible. Since the system is very compact, we will describe it at a more technical level.

We were able to implement this mechanism using C++ templates in a single header file of about 150 lines of code. The system consists of four main classes, `AttributeBase`, `Attribute`, `Operation` and `AttributeSet`.

*AttributeBase, Attribute and Operation*

`AttributeBase` is the base class of both `Attribute` and `Operation` which are templated class wrappers over member variables and member functions respectively. These are collectively managed by the `AttributeSet` class. In `Attributes`, in particular, access is encapsulated by getter and setter functions, which may be equipped with additional function calls to perform subsequent update operations following changes in the attribute. These two small classes encapsulate most interaction between the script environment and the host platform.

*AttributeSet*

The `AttributeSet` serves as base class for all the main classes responsible for the framework's subsystems. As such, it encapsulates all the features required for establishing most of the communication between the scripting environment and the host program.

This is accomplished by storing instances of `Attribute` and `Operation` in an associative container using text strings as keys.

When deriving from `AttributeSet` all of our subsystems define which of their methods and variables will be managed by the Attribute system, which is automatically exposed to *lua*.

The `Wind` class, for instance, exposes its methods for adding new primitives to the current wind field as seen in listing 3.1.

```

Wind::Wind(sol::state& lua)
:
AttributeSet("Wind", lua),
...
{
bind("add_uniform", (std::function<UniformFlow* (const vec3&, float)>>)
    [this](const vec3& dir, float str) { return addUniform(dir, str); });
bind("add_source", (std::function<SourceFlow* (const vec3&, float)>>)
    [this](const vec3& pos, float str) { return addSource(pos, str); });
bind("add_vortex", (std::function<VortexFlow* (const vec3&, const vec3&, float)>>)
    [this](const vec3& pos, const vec3& dir, float str) { return addVortex(pos, dir,
        str); });
}

```

Listing 3.1: Constructor of our wind class

This particular call to the `AttributeSet`'s constructor takes as input the *lua* state. This is required since the `Wind` class will be used as a **singleton** and as result, a global table named `Wind` will be available in the *lua* environment.

The next calls to `AttributeSet::bind(...)` internally create a new `Operation` instance with the appropriate lambda function and string identifier, which will be accessible from the `Wind` table in *lua* as shown in listing 3.2, an excerpt of our examples in section 4.1.4.

```

function onInit()
    ...
    uniformWind = Wind["add_uniform"](vec3.new(0.0, 1.0, 0.0), 0)
end

```

Listing 3.2: Adding a wind primitive in lua

As previously stated, `FlowPrimitive` and its subclasses are also derived from `AttributeSet` so as to expose their attributes, as shown in listing 3.3.

```

struct UniformFlow : public FlowPrimitive
{
    UniformFlow(const vec3& dir, float str)
        : direction(dir), strength(str)
    {
        bind("direction", direction);
        bind("strength", strength);
    }

    vec3 direction;
    float strength;

    ...
}

```

Listing 3.3: The constructor of our `UniformFlow`, a derived class of `AttributeSet`

Note that here, the constructor, `AttributeSet(const std::string& name, sol::state& lua)` used in listing 3.1 is not called, but instead, the default constructor is used. This is due to the fact that since multiple instances `UniformFlow` are allowed it no longer makes sense to create a global table in *lua*. In this case, `strength` and `direction` are created as `Attribute` instances instead of `Operation`, since they are member variables and not methods.

### 3.4 SCRIPTING

We implemented scripting in our system so as to allow for flexible control over all the methods and algorithms present in our framework. The scripting environment receives musical events as they occur and has constant access to the audio features we extract. As mentioned in section 3.1.4, scripts are what grant endless configurability to our application, being responsible for establishing the link between parts of the song and the corresponding changes in the animation.

We have selected `Lua` mostly due it to being specifically designed for embedding in larger applications, as well as having a rich feature set and being both mature and time proven.



To do so, we made most aspects of our framework accessible from the scripting environment as mentioned in sections 3.2.1, 3.2.2 and section 3.3.

Every animation project within our framework has its own script file which follows a set of conventions for interfacing with our program, specifically the implementation of a set of predetermined functions. These functions are then invoked under specific circumstances and form the protocol we have established for the communication between our framework and the scripting environment. For instance, the `onUpdate` function is called on every update cycle and the `onNoteOn` function is called whenever a note begins playing.

Internally, our framework maintains a queue containing the different musical events detected between update time steps. This queue is emptied on every update cycle and each event will trigger the appropriate function call defined in the script file. Since we selected MIDI as our musical source, our current event types refer to a subset of MIDI events, namely note-on and note-off events. However, an intermediate layer, dubbed `SongEvent`, was created so as to allow for the addition of event types. Following this approach, one could add classes of `SongEvent`, for instance, a `KeyChange` event, defining what data it contains, implementing its detection and specifying the corresponding handling function to be implemented in script files.

#### 3.4.1 Script File Structure

Currently our script files are required to define the following functions:

- `onInit`
- `onUpdate`
- `onNoteOn`
- `onNoteOff`

In order to describe our systems functioning we proceed to describe each of them.

##### *onInit*

The `onInit` function is called once at the start of the animation. This is where we define all the mappings and create instances of objects which will be used throughout the animation, such as the wind primitives we mention in section 3.2.6 and 3.10, and exemplify in section 4.1.4.

As previously mentioned, the main goal of the script system is the definition of actions to perform when specific musical events occur. In order to do so, we must be able to label

these moments with an identifier. We can use time to do this, however, we found that most effects we intended to define were triggered by the playing of specific notes.

Our system was built around MIDI files and in MIDI files a note can be uniquely identified by the track it belongs its position (i.e. index) on that track. We found this number to suffice for our purposes. However, situations exist where this approach's limitations become evident, such as tracks with simultaneous notes, which we often wish to treat as if they were a single note. In order to mitigate this issue, we implemented a rudimentary form of chord detection in *Lua*, which will further detailed and demonstrated on section 4.1.4 in the context of a practical example.

This labelling system is defined exclusively in *Lua*, so it is not very difficult to replace it for a more elegant mechanism, for instance by using musical measures. However this would require the file to have a time-signature defined, which is not always the case, and might become more complicated if the time-signature were to change throughout the song.

Now that we have established that we will identify each note by its index in the track, we can proceed to the general solution we have implemented for triggering actions at specific events.

For each track we wish to create effects for, a list of tuples is created. These tuples contain a predicate function and a function for each of the supported events, namely `note_on`, the start of the notes, and `note_off`, the end of the notes. The predicate function evaluates both the incoming event and the track's state (including its current note) and decides whether the corresponding function present in the tuple is to be executed. As such, we define a list of these tuples for every track we wish to animate. This set of lists is represented internally, as everything in lua, as a table which we named the `handlers` table and constitutes the most relevant structure present in our animation's script files.

```
handlers =
{
  [2] = {
    {pred = noteRange(1, 16), note_on = full_grow }
  }
}
```

Listing 3.4: The `handlers` table from our example project in section 4.1.2.

This line of code in listing 3.4 indicates that on track 2, when the note index falls between 1 and 16 (i.e. when `noteRange` returns true), the function `full_grow` will be called as each note starts. This function in particular triggers an iteration of our growth algorithm, as will be detailed in section 3.5.6. We may add more effects to this track by adding new tuples to the list as we show in listing 3.5.

```

handlers =
{
  [2] = {
    {pred = noteRange(1, 16), note_on = full_grow },
    {pred = singleNote(16), note_on = place_leaves }
  }
}

```

Listing 3.5: The handlers table from our example project in section 4.1.5.

This new line follows the same pattern as before, except our predicate function now only returning true at note 16 and invoking a different function, `place_leaves`.

Lastly, for animating other tracks the extension to the previous table is intuitive, as shown in listing 3.6.

```

handlers =
{
  [2] = {
    {pred = noteRange(1, 16), note_on = full_grow },
    {pred = singleNote(16), note_on = place_leaves }
  }
  [3] = {
    {pred = noteRange(0, 3), note_on = onChord(3, wind_pull) }
  }
}

```

Listing 3.6: The handlers table from our example project in section 4.1.5.

Although this effect makes use of another mechanism in the `note_on` function, which we detail in section 3.4.2, the pattern remains unchanged with the specification of track 3 instead of track 2.

The `onInit` function is also responsible for initialising another table, which is the `info` table (listing 3.7), which stores each track's state, namely the current note index. As such we are required to initialise this structure for every track in which effects are used. This table is also used for implementing our chord detection.

```

info =
{
  [2] = { note = 0, lastTime = 0, lastNote = {}, chordNotes = {} },
  [3] = { note = 0, lastTime = 0, lastNote = {}, chordNotes = {} }
}

```

Listing 3.7: The handlers table from our example project in section 4.1.5.

*onUpdate*

The `onUpdate` function is called on every update step and takes as single argument the time elapsed since the previous update, *dt*. Although the `onUpdate` function can perform arbitrary modifications on the simulation we have most commonly used it for accumulating the global time in the predefined global variable *t* (listing 3.8). On some of our examples, however, we have time-based effects which use an auxiliary `clock` structure we defined in lua. The clock structure requires updating with the *dt* parameter, so its update method is also-called on the `onUpdate` function.

```
function onUpdate(dt)
    t = t + dt
end
```

Listing 3.8: Typical implementation of the `onUpdate` function.

An example usage is provided in section 4.1.6.

*onNoteOn and onNoteOff*

The `onNoteOn` and `onNoteOff` are called response to the respective `NoteOn` and `NoteOff` `SongEvents`, built from the original MIDI events, but with extra information, the **duration** which we are able to calculate since we are working with MIDI files and not a live MIDI stream.

These two functions are implemented by default in our lua library and are responsible for accessing the `handlers` structure described in section 3.4.1. For completeness, we show the most relevant part of the implementation in listing 3.9.

```
function onNoteOn(track, key, vel, duration)
    ...
    info[track].lastNote = {key, vel, duration}
    info[track].lastTime = t
    info[track].note = info[track].note + 1

    local actions = handlers[track]

    if (actions ~= nil)
    then
        for k, v in pairs (actions)
        do
            if (v.note_on ~= nil and v.pred(info[track], key, vel, duration))
            then
                v.note_on(info[track], key, vel, duration)
            end
        end
    end
end
```

Listing 3.9: Default `onNoteOn` implementation.

The `onNoteOff` definition is equivalent but differs by invoking the `note.off` function instead of `note_on`.

By showing this definition we also reveal the signature of our predicate and effects functions, which both consist of all the information contained in our `NoteOn` event. This provides a great deal of flexibility particularly to the predicate function since it can evaluate the event in its entirety in order to determine whether the event function should be called. For instance, we could easily define a predicate to only return true if the note's duration is longer than a given threshold or if the note played belongs to a given set of pitches. This advantage also applies to effect function, since it can relate any of the aspects of the `NoteOn` event when defining its corresponding action.

### 3.4.2 Closures

Closures are a very useful language feature in `Lua` which allows us to easily define functions which build other functions and are thus able to abstract common behaviour or functionality<sup>4</sup>. Essentially, a closure consists in a function which has its own local state. This is possible because `Lua` supports **lexical scoping**, which means that when a function is written enclosed in another function, it has full access to local variables from the enclosing function. `Lua` also defines functions as **first-class values**, which means that functions are handled just like regular variables, and as such can be stored in data structures (as we have seen in the `handlers` structure in listing 3.4), passed as arguments, or used in control structures.

The `noteRange(note1, note2)`, `singleNote(note)` and `onChord(n, f)` are all examples of functions which build closures.

The implementation of `noteRange` is actually quite simple:

```
function noteRange(note1, note2)
  return function (trackInfo, key, vel, duration)
    return trackInfo.note >= note1 and trackInfo.note <= note2
  end
end
```

Listing 3.10: Closure for calling a function during a given note interval.

The `noteRange` function operates by building specific predicate functions from its arguments. As mentioned at the beginning of this section, the enclosed anonymous function has access to the arguments passed to `noteRange`, which are referred to in `lua` as **upvalues**, which allow us to compare them to the actual arguments this function receives when it is called.

<sup>4</sup> For more information of closures in `lua` refer to <https://www.lua.org/pil/6.1.html>

The `noteRange` condition only requires two note indexes to evaluate its condition, so we are able to return a new function which has the correct signature for being invoked generically in the `onNoteOn` function and yet ignores unnecessary arguments, improving readability and conciseness.

Overall we found closures to be a very expressive mechanism for defining behaviour. Another useful condition, for instance, is calling a function every *n* notes:

```
function everyNNote(steps, note1, note2)

  return function (trackInfo, key, vel, duration)
    return trackInfo.note >= note1 and trackInfo.note <= note2 and (trackInfo.note
      -note1) % steps == 0
    end
  end
end
```

Listing 3.11: Closure for calling a function every *N* notes.

Closures are not only useful for predicates, they can be used in event functions as well. In listing 3.6 we have the following line:

```
{pred = noteRange(0, 3), note_on = onChord(3, wind_pull) }
```

Listing 3.12: Example use of closures in our handler structure

This is an example of a closure used in an event function. In this case, the `onChord` function builds a new function which only executes `wind_pull` if a three note chord is detected. This mechanism is further detailed in section 4.1.4.

In one of our examples, we have the line displayed in listing 3.13.

```
...
{pred = singleNote(38), note_on = make_wind_push(500)},
...
```

Listing 3.13: A concrete example usage of closures

This is another example of using a closure as an event function. In this particular case, the function `make_wind_push` creates a “`wind_push`” closure with 500 strength. These are the kinds of parameters which a user would tweak in order to achieve the intended effect in their animation. By placing these numbers exactly where the effect is defined we make the code easier to adjust. But more importantly, the general definition of the behaviour is written in `make_wind_push` and can then be instantiated for different usages, in different contexts. For completeness, we also include the definition of `make_wind_push` in listing 3.14.

```
function make_wind_push(strength)

  return function (trackInfo, key, vel, duration)
    local tween = TweenF.create("wind_push", uniformWind:attr_float("strength"),
      strength, 0, t, duration, easings["easeInQuad"])
  end
end
```

```

    tween:setEnabled(true)
    timeline:addItem(tween)
end
end

```

Listing 3.14: The definition of `make_wind_push`

This code displays a very common pattern in our script code, which is the creation of a tween to control an aspect of the animation. In this particular case, a tween is created to vary the strength attribute of a wind primitive named `uniformWind` from the argument `strength` to 0.

We could further generalise this behaviour by accepting more of the tween's arguments as argument to the `make_wind_push` function, such as the wind primitive instance and the easing function.

### 3.4.3 *Sol2*

We used the `Sol2` library for binding Lua to C++. Ideally, this binding could be automated by using advanced language features such as reflection<sup>5</sup>. However, since C++ offers no native support for this, we opted for a more manual approach made possible by the `Sol2` library. This library makes extensive use of template meta programming techniques to expose user defined functions and classes in C++ to Lua with minimum hassle. For instance, registering the easing functions used on the `make_wind_push`, as shown in listing 3.15, is one of the simplest examples.

```

sol::table easings = mState.create_named_table("easings");

easings["easeNone"] = easeNone;
easings["easeInQuad"] = easeInQuad;
easings["easeOutQuad"] = easeOutQuad;
easings["easeInOutQuad"] = easeInOutQuad;
...

```

Listing 3.15: Exposing easing functions to lua.

Exposing an entire (derived) class is also intuitive (listing 3.16).

```

mState.new_usertype<UniformFlow>("UniformFlow", sol::constructors<>(),
    "direction", &UniformFlow::direction,
    "strength", &UniformFlow::strength,
    "attr_float", &AttributeSet::attribute<float>,
    "attr_vec3", &AttributeSet::attribute<vec3>,
    "attr_vec4", &AttributeSet::attribute<vec4>,
    sol::base_classes, sol::bases<AttributeSet>()
);

```

<sup>5</sup> <http://www.randygaul.net/2014/01/01/automated-lua-binding/>

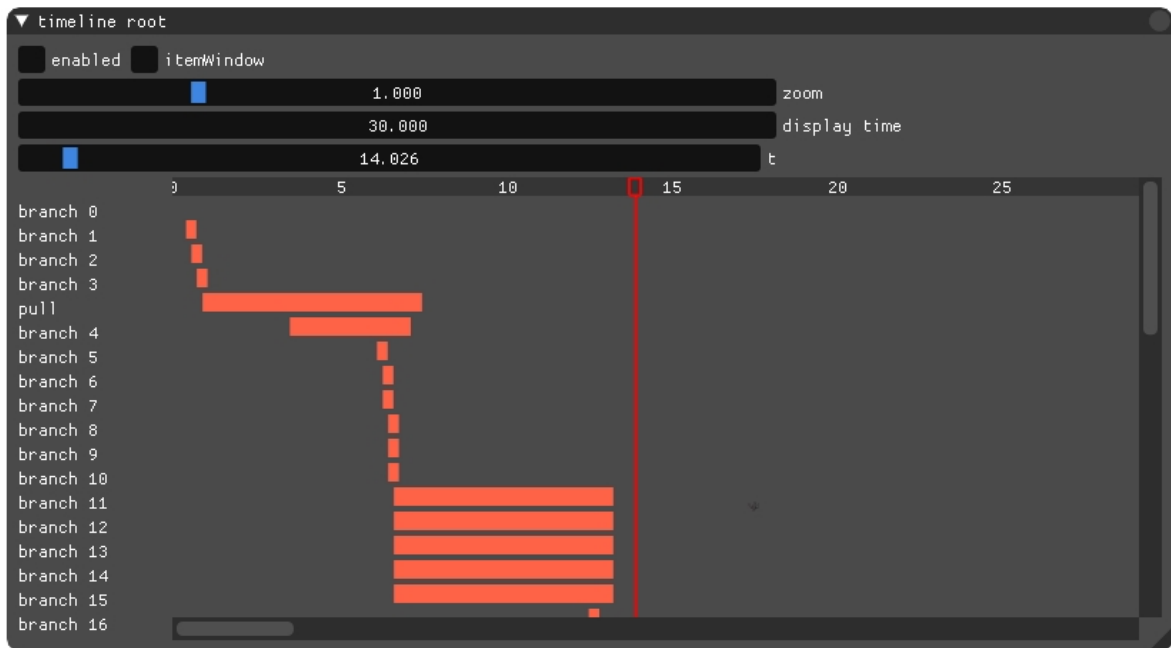


Figure 30: Timeline interface

Listing 3.16: Using sol2's `new_usertype` for exposing the `UniformFlow` primitive to lua.

Where `mState` is Sol2's wrapper over the Lua interpreter object.

Another very useful feature is the automatic handling of *Standard Template Library (STL)* containers, as well as smart pointers and lambda functions.

### 3.5 ANIMATION

The animation module is responsible for all changes in the animation which are not physically driven but instead established and customised by the user. Examples include the continuous growth of branches and changes in wind strength or direction.

#### 3.5.1 Timeline

The timeline is yet another fundamental part of our system. It is responsible for keeping track of time elapsed since the animation began and correctly updating all effects on every frame (the interface is shown on figure 30).

It is advantageous not only for centralising the processing of every time-dependent effect but also allows for a convenient piano-roll like visualisation of all effects acting on the



animation. We refer to these effects in abstract using the term *timeline item*, which is any effect which requires an update every frame and is usually characterised by a start time and a duration.

### 3.5.2 Tweens

Tweens are the most widely used timeline item in our examples due to their simplicity and versatility. Tweens are essentially a simplified form of **keyframing** which is a technique originating from traditional animation and which remains present in every animation today.

In traditional animation, the animator will draw frames of important moments in the animation, dubbed **keyframes**, and then draw the frames in **between** so as to allow for a smooth transition representing the illusion of motion.

In our implementation tweens partially automate this process by defining two key values and interpolating between them. Any type of value can be used provided that the interpolation algorithm is suitably defined. Although different interpolation schemes are possible, we found that using linear interpolation sufficed since we adjust its constant rate of change by applying what is commonly known as an **easing function**.

Easing functions are functions which take a single argument  $t$  in the  $[0,1]$  interval and return a value in the same interval.

The most common naming convention is `ease(In|Out|InOut) [type]` depending on how the rate of change varies. The rate of change of `easeIn` functions constantly increases as  $t$  increases while in `easeOut` it decreases. `easeInOut` are usually the combination of the first cases, which means the rate of change typically increases until half-way when it starts decreasing. The last portion of the name usually references the mathematical primitive upon which the easing function is based on, for instance, `easeInQuad` uses a quadratic polynomial( $t^2$ ) and `easeInExpo` uses exponentiation ( $2^t$ )<sup>6</sup>. To demonstrate the simplicity of this concept we present a condensed version of the Tween class' update method in listing 3.17.

```
void Tween::update(float)
{
    mProgress = clamp(mEase(animationTime(true)), 0.0f, 1.0f);

    if (mActive && mTarget)
        *mTarget = mInterpolate(mMin, mMax, mProgress);
}
```

Listing 3.17: Tween update implementation

<sup>6</sup> For an interactive gallery of common easing functions visit <http://easings.net/>

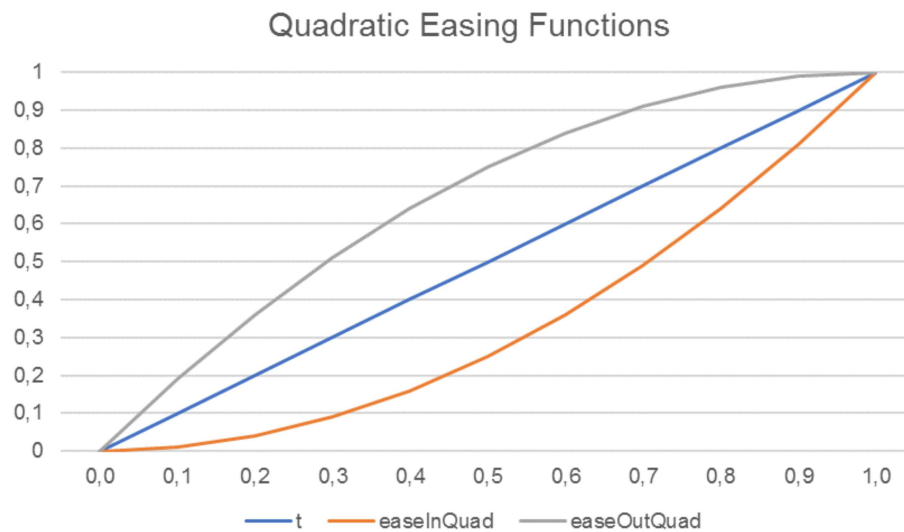


Figure 31: Plot of the `easeInQuad` and `easeOutQuad` functions, as well as the linear function for reference.

`animationTime(true)` returns the normalized linear progress of the animation. We modify this value with the tween’s easing function and then use this value for the final interpolation.

The simplest versions of easing functions are arguably the quadratic `easeInQuad` and `easeOutQuad` shown in listing 3.18.

```
inline float easeInQuad(float t)
{
    return t*t;
}
inline float easeOutQuad(float t)
{
    return -t * (t - 2);
}
```

Listing 3.18: Implementation of the quadratic *easeIn* functions

These functions are represented graphically in figure 31 and portray the general behaviour of the *easeIn* and *easeOut* family of functions.

Due to their abstract nature, we were able to use these functions to control several distinct aspects of our animations, particularly every aspect where interpolation is applicable.

The most common effect we explore in our examples is synchronising a musical note with a tween of the same duration. This tween may control attribute variations such as branch growth, wind strength or colour changes. By using easing functions we are able to expressively match, for instance, short notes with sudden effects and longer notes with softer, more gradual effects. In summary, we found that these small mathematical helpers

are versatile enough to provide a fair expressive power while granting a significant degree of automation.

### 3.5.3 *TweenSequence*

*TweenSequence* objects are simply an encapsulated set of tweens which refer the same attribute. As such, they are adequate for describing effects which require more than two keyframes<sup>7</sup>. As described in section 3.5.6, a particular usage example of *TweenSequence* is tree growth. Every *GrowNode* contains a *TweenSequence* to control its length. This allows us to grow only a fraction of the segment at each time, since at present our growth algorithm defines a uniform maximum length for every segment, as we mention in section 3.7.1.

### 3.5.4 *DataLink*

The *DataLink* is our basic primitive for the creation of continuous effects. It is a pure abstraction of the concept of mapping, consisting of a structure containing only two function pointers, which we termed an *extract* and an *apply* function. On every update cycle, a value is extracted using the *extract* function and passed as an argument to the *apply* function. It is a templated class, and its single type argument is used to simultaneously determine the return type of the *extract* function and the argument type of the *apply* function. These functions may be defined in Lua and as such, this object provides a very flexible approach for continuously executing an effect, which is based on the arbitrary transformations of any value. We use this object in particular for representing information obtained through the *AudioAnalyser*, since it is continuous in nature. This primitive will be revisited in the tutorial section 4.1.6.

### 3.5.5 *Cue*

The *Cue* class is the simplest of all *TimelineItems*, as its only function is to call an arbitrary function once. This can be used, for instance, to change the visibility of an object or trigger an activation of a wind primitive.

---

<sup>7</sup> In the future, we would like to transition to a more robust approach, namely spline interpolation, which we already employ for the construction of the tree.

### 3.5.6 Application to Tree Growth

The growth of tree segments and leaves is a particular use case of tweens, since each segment contains tween objects to control each of those attributes, length and leaf size. Segments, in particular, contain a `TweenSequence`, which we describe in section 3.5.3.

We took the approach of coupling tree segments directly with tweens because when creating examples we found that the pattern of creating a tween for branch length to be incredibly common. As such, in order to avoid instantiating hundreds of tween objects per growth iteration from the script file, we chose this approach, which creates them in advance.

## 3.6 AUDIO AND MUSICAL INPUT

Thanks to `SFML`, our system has support for most common audio file types. It also gives us access to every sound sample contained in the file as it is played. From this data, we can extract low-level audio features, such as the frequency spectrum, which we obtain directly by using the *Short Time Fourier Transform (STFT)*. The sampled audio signal is in a time-amplitude domain, and what the *STFT* allows is the conversion to the frequency-magnitude domain. In essence, this transform operates by comparing the input signal with sinusoids of various frequencies. Each of such sinusoids or pure tones may be thought of as a prototype oscillation. As a result, we obtain for each considered frequency interval a magnitude coefficient whose value indicates the degree of presence of those frequencies in the original signal.

The final result is a histogram where each bin corresponds to a frequency interval and the height to those frequency's intensity over the sampling window. This type of low-level feature is also employed by Milkdrop in order to increase the reactivity of the visualisations. In their case, this spectrum is divided into 3 parts, namely the low, mid and high frequencies. These values can then be used within presets for any desired effect. Although this is a low-level feature, the human eye can often identify the individual instruments due to their timing, intensity and frequency range.

However, in an audio representation, note parameters such as onset times, pitches or durations are not given explicitly and extracting this information is not a simple task, in particular regarding polyphonic music, where different instruments and voices are superimposed upon each other. In order to bypass this problem, we chose to add a musical input source to our system, specifically MIDI. This decision was made for the following reasons:

- MIDI is widely supported and time-tested, allowing for communication with musical input devices, such as keyboards;

- A vast amount of MIDI files are freely available online;
- It is a very compact representation of music, which makes it both lightweight and simple to process.
- It allows for direct playback by controlling a synthesiser.
- Higher level formats such as MusicXML can be converted to MIDI representation.

By using MIDI we allow users to access low-level musical features and use them directly in animations. This way we begin unlocking access to the musical structure. For instance, we were able to quickly implement rudimentary chord recognition with MIDI streams since individual notes are readily available.

It is also relevant that the output of algorithms for extracting musical information from audio will ideally produce the same information as contained in MIDI, so by using MIDI, we are not imposing limitations on our system.

The ideal situation for an animation to be generated is having both a MIDI file describing every track, as well as a synchronized audio file. This combines the best of both worlds, as specific musical analysis can be performed from the data in the MIDI file and dynamics and timbre information are more accessible on the audio data.

### 3.7 TREE MODELLING

The first approach for the procedural modelling of trees was implemented with *L-systems*, the most widely used system for the botanical simulation of plant growth. L-Systems are rule-based, which makes them suitable to model a wide range of plants with very different morphologies, usually consistent branching patterns, however complex. In this project, however, capturing the morphology of different plants was not as much of a priority as was allowing their structure to be dynamically influenced by external factors. Moreover, L-Systems are notoriously difficult to control, as any small change to a single rule may radically change the entire structure.

Producing convincing models displaying some variety is sufficient for the broader problem we are approaching, which is why we found the *SCA* presented in [Runions et al. \(2007\)](#) to be a perfect fit for this project. Using the algorithm we are able to interactively generate a large variety of tree models based on the concept of having several points in space which direct tree growth. The input parameters also correspond to visually relevant tree characteristics and offer convenient control of tree shape and structure. The tree is produced with a natural, base-to-tips order, which makes it suitable for animation.

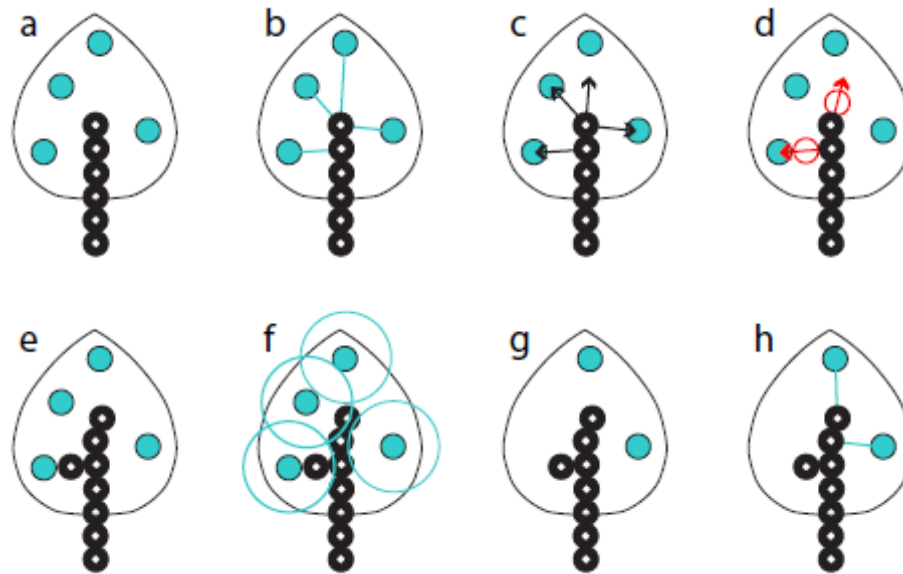


Figure 32: The space colonization algorithm Runions et al. (2007)

### 3.7.1 Space Colonization Algorithm

The beauty of the *SCA* lies in its simplicity. As previously stated, the key aspect of this algorithm is the use of a cloud of points to signal the available space for a growing tree. These points effectively act as attractors to the branching structure and are thus referred to as *attraction points*.

In the original article, a three-dimensional envelope is given as an input. This envelope represents the overall shape of the tree crown. The authors used a surface of revolution obtained by rotating a planar generating curve. Afterwards, the inside of this envelope is populated with random attraction points.

In our implementation, we simplify this step by obtaining points through a uniform sampling of simple geometric primitives such as a box or a sphere. Given the attraction points, the tree skeleton is formed in an iterative process, beginning with a single node at the base of the tree. In each iteration, new nodes, delimiting short branch segments extend the skeleton in the direction of nearby attraction points. In our implementation, we take this length to be constant.

The algorithm is parametrized by the **segment length**  $l$ , **radius of influence**,  $d_i$  and the **kill distance**,  $d_k$  as well as the number of attraction points created,  $N$ , and their distribution. The radius of influence defines a spherical region surrounding every attraction point. Any tree nodes within this region will be extended with new nodes on the next iteration. The kill distance, on the other hand, defines the minimum distance between a node and an attraction point, that when reached triggers the removal of the attraction point.

Figure 32 illustrates a single iteration of the algorithm. We begin following its operation at the stage when the tree structure is already composed of six nodes (black disks with white centres) and there are four attraction points (blue disks) (figure 32a). First, each attraction point is associated with the tree node that is closest to it, provided that the node is within the **radius of influence** (figure 32b, blue lines); this establishes the set of attraction points that will influence each node. The normalised vectors from each tree node to each influencing point are calculated (figure 32c, black arrows). These vectors are added and their sum is normalised again (figure 32d, red arrows), providing the basis for locating new tree nodes (figure 32d, red circles). The new nodes are incorporated into the tree structure, in this case extending the main axis and beginning a lateral branch (figure 32e). The region surrounding each attraction points is now tested for the inclusion of (the centres of) tree nodes, using the **kill distance** as the radius (figure 32f). The region of the two leftmost points has been reached by the new branches and the attraction points are thus removed (figure 32g). The tree nodes closest to these points are now identified (figure 32h), beginning the next iteration of the algorithm.

All parameters have somewhat intuitive effects on the tree structure. So as to limit the parameter space, we define both  $d_i$  and  $d_k$  as directly proportional to the unitary segment length,  $l$ . Decreasing  $N$  or increasing  $d_k$  will yield crowns that are increasingly sparse. Regarding the radius of influence,  $d_i$ , as its value decreases, branch tips tend to alternate radically between attraction points, coming into, then leaving their zones of influence which results in a wiggly or gnarly appearance. On the other hand, if the value of  $d_i$  increases the resulting skeleton will have smoother branches since more attraction points will be used for calculating their direction.

Due to the stochastic nature of point selection, this algorithm allows us to generate an endless variety of trees, even with the same parameters and attraction point configuration.

Branch thickness can be obtained from a botanically accurate model, such as the pipe model in [Shinozaki et al. \(1964\)](#), which relates the cross-section of a limb below a branching point to the combined cross-sections of the limbs above. However, we opted for a more ad-hoc solution based on a specified initial thickness and an arbitrary easing function, as in section 3.8.3

### 3.7.2 Tree Representation

Our tree models are a hierarchically organised modular structure, represented internally by an actual `tree` data-type. Each node on this data structure represents a possible branching point, which has as its parent the preceding branching point. This connection establishes a small branch segment, often referred to as an *internode* in botany. Each chain of vertices thus represents a full branch.

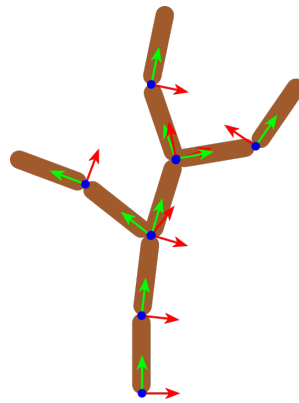


Figure 33: Tree nodes and their local coordinate system.  $x$ ,  $y$  and  $z$  axes are represented in red, green blue, respectively.

Each internode is approximated as a cylinder and has its own local coordinate system (as can be seen on figure 33) where physics calculations are performed. Each cylinder is placed at the end of its parent internode end and may only rotate with respect to that end. The physics simulation step is described in 3.8.

As such, each node is characterised by several physical parameters detailed in 3.8.3.

Lastly, we store leaves as children of internodes, initially aligned perpendicularly to their respective internode and then having their orientation disturbed by a random rotation. As such each node stores each of its leaves' relative position.

### 3.7.3 Tropisms

Tropisms in trees are the tendency of branches to turn in a particular direction. Several tropisms have been identified by botany, for instance, phototropism, a turning towards light and gravitropism, the turning according to gravity (positive in stems and negative in roots).

Due to this nature of the algorithm, approximating the effects of tropism can be implemented by biasing the growth direction, as can be seen in figure 34.

## 3.8 TREE PHYSICS SIMULATION

When simulating tree response to wind, we sought for an approach which was physically based, in order to obtain realistic results, as well as lightweight and simple to implement. The algorithm we selected was the one presented in Sakaguchi and Ohya (1999). Recent literature on tree generation and wind-response simulation, such as Pirk et al. (2014) and Oliapuram and Kumar (2010) made use of this approach with good results, which was our primary motivation its selection.



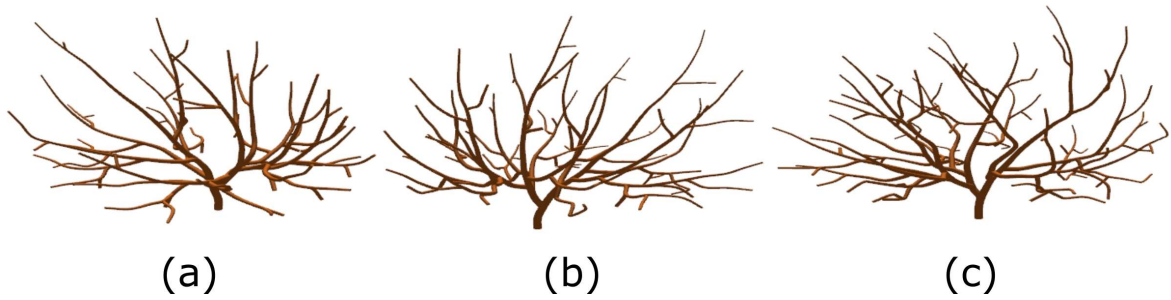


Figure 34: The effect of tropisms in our implementation: (a) Left tropism (b) Right tropism (c) Right tropism followed by left tropism.

Gravity and air resistance are not considered in this simulation. Gravity could have a strong impact in the simulation, however, since it was not taken into account when creating the tree, including it in the simulation would make the entire tree droop in a non-natural way. Air resistance is considered not to have a significant impact on strongly rigid structures, such as branches, which are under the influence of strong binding forces.

The method operates in a two-step process, the **dynamics calculation phase** and the **integration phase**. The dynamics phase is responsible for calculating the individual rotational motion of each segment, while the integration phase joins the individual motions and ensures the segments remain connected.

### 3.8.1 Dynamics Calculation

In the dynamics phase, we make use of the segments obtained from the SCA and model each of them as a fixed cylinder, positioning its base at the end of its parent and allowing only rotational movement about that end.

Each of these cylinders resides in its own local coordinate system with the origin at the fixed end and the y-axis aligned with its resting direction, as calculated by the SCA and shown on figure 33.

We calculate the movement of the tree by considering several forces acting upon each segment, namely the force applied by wind  $F_w$ , the restoring force  $R$ , the axial damping force  $D$  and the back-propagation force from the child branches  $P$ , which we will discuss in the following sections.

We take the classical Cartesian coordinate system  $(x, y, z)$  as our basis. As such, angular orientation,  $\theta$ , angular velocity,  $\omega$  and angular acceleration  $\alpha = \frac{d\omega}{dt}$  all have three components corresponding to the rotation about each of the main axes. As such, the final axis of rotation is found by normalising  $\theta$ ,  $\hat{\theta}$ , and the angle corresponds to its magnitude,  $|\theta|$ .

Lastly, since the local coordinate system of each node is aligned with its resting direction, a segment will be in resting orientation when  $\theta = (0, 0, 0)$ .

*Wind Force*

This is the external force exerted by moving air particles colliding with the branch segment. It is directly proportional to the wind velocity  $v_w$  obtained from the wind field, the surface area of the branch segment facing the wind  $S_f$  and a constant representing the viscosity coefficient of air  $\sigma$ . The final formula is then:

$$F_w = v_w S_f \sigma \quad (1)$$

*Restoration Force*

The restoration force is the internal force which attempts to restore the segment to its resting position. This force is directly proportional to the segment's rigidity coefficient,  $k$  as well as the segment's angular distance from its resting orientation.

$$R = -k\theta \quad (2)$$

*Axial damping force*

The axial damping force suppresses the motion of branches as a result of the strong binding forces between branch segments. As such it acts against the angular velocity and its magnitude is also proportional to the segment's damping coefficient.

$$D = -d\omega \quad (3)$$

*Backpropagation Force*

The backpropagation force is the accumulation of the forces exerted by child segments on the current segment. Each child segment contributes to this force with a fraction of its own restoration force, which per Newton's third law is of inverse direction since the force being propagated is the reaction force.

To obtain this force, all of the child branches' forces must be known in advance. Consequently, the force calculation is performed recursively from the outer edges to the root, that is in tips-to-root order. The fraction of the restoration force which is propagated is determined by the back-propagation coefficient, which depends on a fixed parameter and the thickness ratio between the parent and child segment. This models the fact that thicker branch segments will exert a stronger backpropagation force. Formally this can be written as:

$$P = - \sum k_i K_i \quad (4)$$

where  $k_i$  is the propagation coefficient of the force, as defined by:

$$k_i = bp_c \frac{Th_i}{Th_{i-1}} \quad (5)$$

Where  $bp_c$  is the fixed propagation coefficient and  $Th_i$  and  $Th_{i-1}$  are the thickness of the child and the parent segment respectively.

#### *External Force*

Our system also considers another force, which we refer to as the external force,  $F_e$ . As any of the previous forces, it is a vector quantity and it is exposed to our script system so as to allow for arbitrary forces to be applied to the tree model.

#### *Total Force*

The final force can be expressed as the addition of all previous forces:

$$F = F_w + R + D + P + F_e \quad (6)$$

After the total force has been calculated, we must find the effective change in angular velocity and orientation. Since we are dealing with the rotation of an object around a fixed point we must first convert this force into torque.

The torque is calculated as the cross product of the total force  $\mathbf{F}$  and the edge vector  $\vec{e}$ , that is the vector from the cylinder's origin to its end. This is the same model one could apply to determine the torque of a wrench acting on a nut.

$$\tau = F \times \vec{e}. \quad (7)$$

Since we have both these quantities we can calculate the torque. Now it is necessary to relate the torque to the angular acceleration, that is, the temporal derivative of the angular velocity.

For a fixed cylinder the following approximately holds:

$$\tau = I \frac{d\omega}{dt}; I = \frac{mr^2}{3} \quad (8)$$

where  $I$  is the moment of inertia,  $m$  is the branch mass and  $r$  is the length of the edge  $e$ . The actual values of the mass and length are calculated from the tree geometry obtained from the [SCA](#). In our implementation, the length is equal for all tree segments and as it is a parameter for the [SCA](#), while the mass is obtained by calculating the volume of the approximating cylinder and multiplying it by a density coefficient which varies according to the tree species.

We now have all the terms required to find the angular acceleration. To do so we simply rewrite the previous equation to:

$$\frac{d\omega}{dt} = \frac{\tau}{I'} \quad (9)$$

Finally, we must relate the angular acceleration to both angular velocity and angular orientation. To do so we use the equations of angular motion:

$$\begin{aligned} \theta' &= \theta + \omega(\Delta t) + \frac{1}{2}\alpha(\Delta t)^2 \\ \omega' &= \omega + \alpha(\Delta t) \end{aligned} \quad (10)$$

These are evaluated at every time-step of the simulation and are responsible for the movement of the tree as a whole. More specifically, we are integrating the equations of motion using the traditional forward Euler method.

### 3.8.2 Integration of Movements

The force that propagates from the child branch to the parent branch is defined in 3.8.1, but it is also necessary to consider the opposite case, that is, the influence of movement propagating from the parent branch to the child branch.

The back propagation from the child branch to the parent branch is expressed by forces, but the propagation from the parent branch to the child branch does not need to be expressed by forces. Even though the movements of the parent branch segment change the absolute position of the child branch, their relative positions remain unchanged. As such the integration of movements can be performed simply by placing the fixed end of child segments at the free end of their respective parents and accumulating their rotation.

This behaviour is simply the accumulation of geometrical transformations commonly found in scene graph implementations, which we had already implemented when establishing the tree representation, as described in section 3.7.2 (35c).

The dynamics calculation phase (figure 35b) provides the local transformation of each branch and since all segments are already hierarchically connected in a tree data structure it is only necessary to ensure that individual transformations are correct and the desired behaviour is obtained.

### 3.8.3 Tree Parameters

Each branch is parametrised by **length** and **thickness**, as well as **rigidity** and **damping** coefficients. Finding an adequate method for calculating these parameters and obtaining

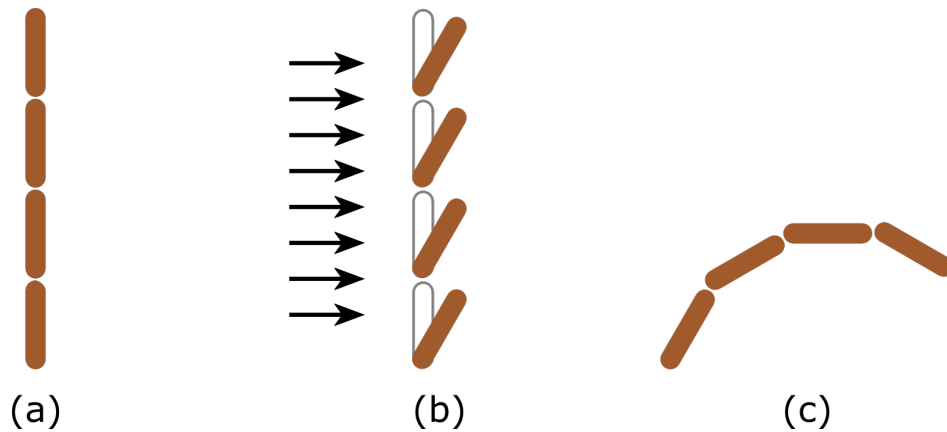


Figure 35: (a) Resting Position (b) Dynamics calculation (c) Integration of movements

satisfactory behaviour was actually one of the most challenging aspects of the implementation.

Each of these parameters has a dramatic effect on the overall motion of the tree, so we searched for a flexible approach which allowed for obtaining different behaviours.

Ideally, we would find an adequate formula for connecting these physical attributes (thickness, length or mass) with both the damping and rigidity coefficients. However, all our attempts to craft and use such a formula led to unstable simulations. In order to overcome this difficulty, we took a more heuristic approach.

The practical solution we applied was to find an attractive ratio between rigidity and damping, establishing minimum and maximum values and then interpolating between them. This is plausible since both these coefficients will become smaller as branches become thinner, that is, a thin branch will offer less resistance to wind and will continue oscillating longer after the wind has stopped. The minimum and maximum values are not based on thickness but instead on the depth level of the segment. Unfortunately, this requires the user to estimate the maximum depth of the tree, by previously simulating its growth. However, this interpolation can be adjusted by, once again, using easing functions as described in section 3.5. For completeness, we include the equation which translates this approach determining the damp and rigidity values. The ratio we found was of 4:1 in rigidity to damping respectively, the maximum tree depth is approximated as 30, and the maximum values follow the ratio to obtain the values 20 and 5:

$$\begin{aligned} rigidity &= 20 \times \left(1 - easeOutCubic\left(\frac{depth}{30}\right)\right) \\ damp &= 5 \times \left(1 - easeOutCubic\left(\frac{depth}{30}\right)\right) \end{aligned} \tag{11}$$

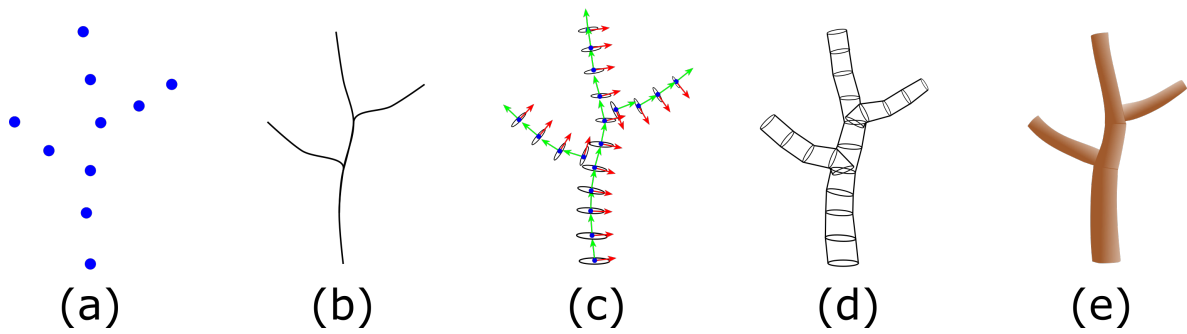


Figure 36: Tree mesh generation using generalized cylinders.

### 3.9 TREE CONSTRUCTION

As stated in 3.7.2, our tree model is stored as a collection of nodes, each representing a branch segment. We can also regard these nodes as vertices of the tree skeleton representing every possible branching point. Each node has numerous attributes, namely coefficients for the physics simulation, however relevant for construction are the node's position, thickness and leaves.

We implement the construction of the tree structure by simplifying the approach described in Bloomenthal (1985). As such, we make use of **generalized cylinders** whose axes are splines which interpolate the tree skeleton. This approach can also be described as an extrusion of the interpolating spline.

The key aspects of the algorithm are illustrated in figure 36. Since each of the nodes resides in its own coordinate space we obtain their global positions by accumulating all transformations of their parent nodes (figure 36a).

We then create as many interpolating splines as there are branches (chains of vertices) to smoothly join the nodes. We employ *Catmull-Rom* splines as described in Twigg (2003) due to their property of passing through the supplied control points (figure 36b).

Afterwards, we discretely step through the spline according to a resolution parameter and build a coordinate frame at every step with its forward vector aligned with the curve tangent. The thickness at every step is found by interpolating a separate one-dimensional spline. The coordinate frame is then used to place a disk at each interpolated position with the interpolated thickness as its radius (figure 36c). Every vertex on the disk is then "stitched" to its predecessor and successor, forming a continuous cylinder or tube (figure 36d). The result is a single mesh representing the tree (figure 36e).

### 3.9.1 Limitations

In our implementation, although the final result of the construction algorithm is a single mesh, the different branches are not effectively joined, but rather overlaid, as we show on figure 36d. A solution to this is described and implemented in Bloomenthal (1985), which consists in using a "ramiform" primitive, which is a dedicated mesh crafted for the sole purpose of joining two branches together (see figure 20b). A different solution is also considered in Runions et al. (2007) which is the use of the method described in Galbraith et al. (2004). This method makes use of implicit surfaces ("blobs") to deal with branch discontinuities. Although this yields impressive results, at the time of publication would take under 1 hour to render complex models at a moderate resolution. In the article, the authors claimed this cost to be prohibitive for interactive use. As such, due to this statement, and the complexity of the implementation, we did not explore this approach.

## 3.10 WIND FIELD SIMULATION

A wind field can be described by the Navier-Stokes equations, which can be solved by a variety of algorithms. In our system, we greatly simplify the original equations by using linearized fluid flow, as described in Wejchert and Haumann (1991).

This simplification arises from three assumptions, the fluid is **inviscid**, **irrotational** and **incompressible**, which is a reasonable model for air at normal speeds when it does not exhibit turbulence Wejchert and Haumann (1991).

When two fluid layers move relative to each other, a friction force develops between them and the slower layer tries to slow down the faster layer. This internal resistance to flow is quantified by the fluid property **viscosity**, which is a measure of internal "stickiness" of the fluid. Viscosity is caused by cohesive forces between the molecules in liquids and by molecular collisions in gases. As such all real fluid flows involve viscous effects to some degree. However, in many flows of practical interest, there are regions (typically regions not close to solid surfaces) where viscous forces are negligibly small and this assumption greatly simplifies the analysis without much loss in accuracy, as described in EngArc **Inviscid Flow**.

The inviscid simplification leads to the valid assumption of irrotationality. Rotation of a fluid particle can only be caused by a torque applied by shear forces on the sides of the particles. Since shear forces are absent in an inviscid fluid, the flow of such fluids is essentially irrotational. Although the fluid may travel in circular trajectories (as is the case with vortex flow described in section 3.10.3) the fluid itself does not rotate (Massey and Ward-Smith (1998)).

Generally, when the flow is viscid, it also becomes rotational. This is due to the fact that viscosity introduces velocity gradients and introduces distortion and rotation of fluid particles. The condition of irrotationality can be expressed mathematically using the **curl** operator ( $\nabla \times$ ). Curl describes the infinitesimal rotation of a vector field, that is, at every point in the field, the curl of that point is represented by a vector whose length and direction characterize the rotation at that point. A vector field  $v$  representing irrotational flow satisfies  $\nabla \times v = 0$ .

Lastly, incompressibility dictates that the **density** of the flow remains constant. This can be expressed using the **divergence** operator ( $\nabla \cdot$ ), which assigns each point in the vector field with a scalar value representing the field's tendency to converge toward or repel from that point. Similarly to irrotationality, we can formulate incompressibility as  $\nabla \cdot v = 0$ .

A vector field which satisfies these restrictions will also satisfy **Laplace's equation**:

$$\nabla \cdot v = \nabla \cdot \nabla \phi = \nabla^2 \phi = 0 \quad (12)$$

Where  $\phi$  is the potential function. Potential functions are also known as harmonic functions and are characterised by having continuous second-order partial derivatives. In this context, we obtain three elementary vector fields (i.e. flow primitives) by applying the gradient operator to a set of potential functions and obtain what the authors refer to as **flow primitives** which are guaranteed to satisfy the Laplace equation.

Since equation 12 is a linear differential equation, if we find two analytical solutions then their linear combination is also a solution. It is this property which allows us to combine multiple primitives in order to create complex flows. Although the mathematical reasoning behind this model is somewhat intricate, the practical implementation is both elegant and intuitive.

Wind vector fields are then obtained from the combination of *flow primitives* (figure 38). Each of these primitives corresponds to a flow building block: **uniform flow**, which flows with a given strength in a given direction; **sink/source flow**, which flows towards/away from a given position and lastly **vortex flow**, which flows around a given axis. These primitives can be represented as three-dimensional linear equations and thus can be solved analytically and combined through simple addition. Thus we can add the primitives to create more complicated flows, as shown in figure 39.

Flow primitives can be concisely described in both cylindrical coordinate and spherical systems, as illustrated in figure 37.



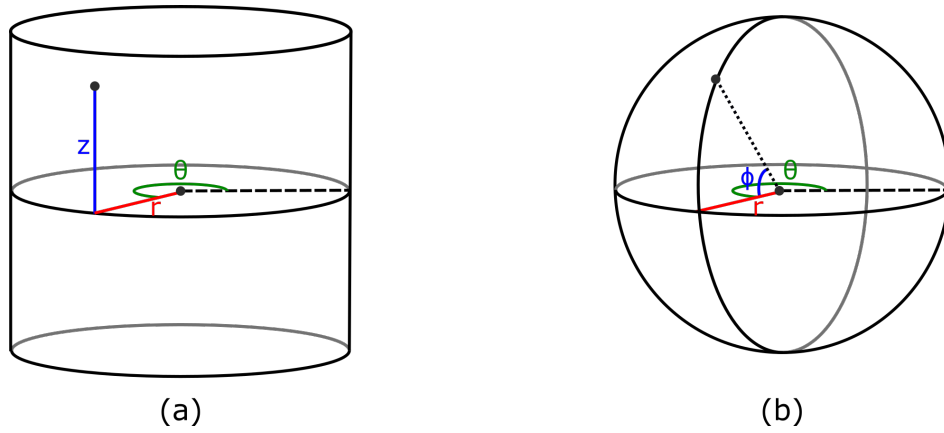


Figure 37: (a) Cylindrical coordinate system, each point is characterised by  $(r, \theta, z)$   
 (b) Spherical coordinate system, each point is characterised by  $(r, \theta, \phi)$

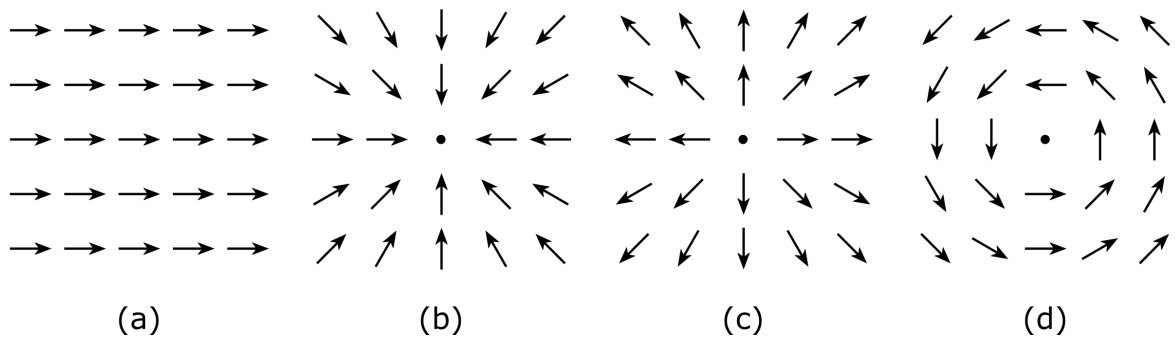


Figure 38: (a) Uniform Flow (b) Sink Flow (c) Source Flow (d) Vortex Flow

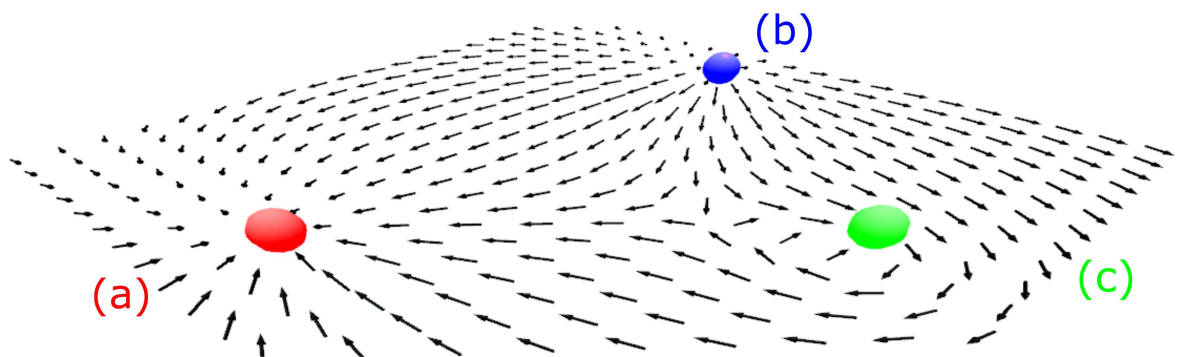


Figure 39: 2D slice of a 3D wind vector field (a) Sink Flow (b) Source Flow (c) Vortex Flow

### 3.10.1 Uniform Flow

Uniform flow (figure 38a) is the simplest flow possible and can be visualised as a one-directional vector field. It can be expressed simply as a function which assigns the same vector to every point in space.

### 3.10.2 Source/Sink Flow

Source flow (figure 38b) flows away from a given position while sink flow flows towards a given position depending on whether its strength  $s$  is positive or negative, respectively (figure 38c). It is expressed in spherical coordinates  $(r, \theta, \phi)$  at the origin as:

$$v_r = \frac{s}{2\pi r} \quad v_\theta = 0 \quad v_\phi = 0 \quad (13)$$

This equation indicates that the velocity vectors have no angular component, which translates to a set of vectors pointing directly away or towards the origin.

### 3.10.3 Vortex Flow

Vortex flow (figure 38d) flows in concentric circular trajectories around a given axis with strength  $s$ . It is expressed in cylindrical coordinates at the origin as:

$$v_r = 0 \quad v_\theta = \frac{s}{2\pi r} \quad v_z = 0 \quad (14)$$

Contrary to source/sink flow, this equation dictates that the vectors only have an angular component, leading to a set of vectors which circle the origin.

### 3.10.4 Drawbacks

Simplification is never without its limitations. In this model, turbulence is not simulated, which is relevant for the chaotic behaviour of wind fields in reality. However, since our goal was maximum control, this was not particularly severe and it would be possible to extend our current system with noise (Bridson et al. (2007)) to simulate the effect of turbulence. Another limitation is that we only model one-way coupling between the wind and the tree, that is, the wind influences the tree but the inverse does not occur. This would also be relevant for a realistic simulation since wind flow can be dramatically changed by collision with the complex geometry of trees and leaves, producing intricate vortices and displaying chaotic behaviour. Pirk et al. (2014) were able to achieve this by using SPH.

# 4

---

## USAGE AND EXPERIMENTS

---

Simple things should be simple,  
complex things should be possible.

---

Alan Kay

In this chapter, we will present the end result of our project, which is the application itself, named *AuxIn*<sup>TM1</sup>. The name comes from a wordplay between *Auxiliary Input Ports*, commonly abbreviated to *AUX*, which are audio connections which allow devices to receive sound from any media player with a normal headphone socket, and *auxins*, which are the plant hormones produced in the stem tip that promote growth.

First, we will present an introductory tutorial which will showcase the application's main features while going through a basic example. We follow this section with a brief description of the two main examples we created throughout the development of the application.

### 4.1 USING AUXIN

*AuxIn* is a 3D application with a graphical user interface for the creation of sound-reactive animations. We will introduce its most relevant features at present, going through the complete creation of a very basic animation. As described in section 3.1, three elements are used for creating an animation in our system, a MIDI file, an optional audio source and a script file.

#### 4.1.1 Obtaining a MIDI File

A MIDI file can be obtained in various ways. The best one would arguably be working directly with a musician. The simplest case would be that of a pianist since most digital pianos are able to produce MIDI output. This way, not only do we automatically obtain MIDI data, but we also have the guarantee that it will be perfectly synchronised with the

---

<sup>1</sup> not actually trademarked

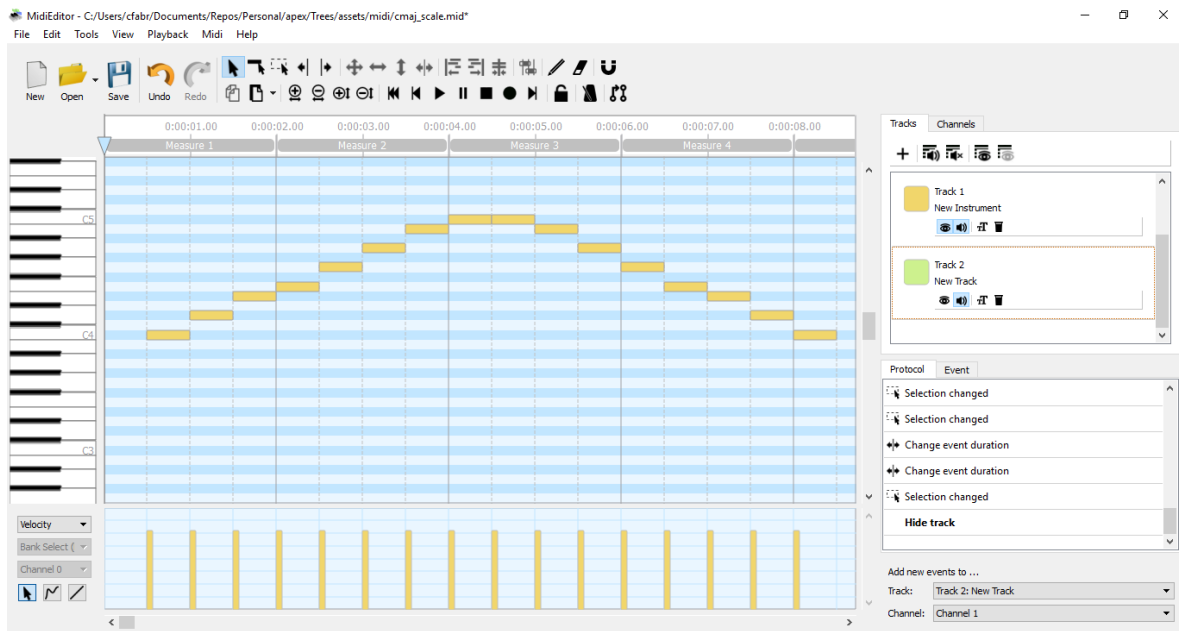


Figure 40: Our simple “melody”.

piano’s audio. Other musicians could transcribe their performance in such a way that a digital sheet music becomes available. The conversion to MIDI can then be automated using specific software. Another way would be to obtain a MIDI from the internet, where they are abundant. However, this poses the problem of synchronisation if one wishes to use real audio, as we described in section 3.1. Since our system provides MIDI playback, one can use the MIDI data itself to produce the audio by using a software synthesizer, which is available on most operating systems. This was the approach we took in our second example “Navorski”. We will also take this approach for the purposes of this tutorial and we will additionally be creating a MIDI file from scratch.

To do so, we will use the freely available open-source software [MidiEditor](#).

To keep the example minimal we decided to create a simple ascending and descending C major scale in straight quarter notes. We leave the tempo and time signature in their default settings, which are 120 BPM and 4/4 time. The final result in MidiEditor is shown in figure 40

We save this file as `cmaj_scale.mid` and we are done with creating our MIDI.

Now let us load it in *AuxIn* to check if everything is correct, this is shown on figure 41.

At first, it seems odd that we have two tracks when we only created one, but this is due to MidiEditor’s behaviour of creating a default track for selection of the time signature and the tempo. Since there is no note information on this track, it is correctly displayed as blank. We could adjust the file so that the scale also belongs to the first track, but this represents minimal changes in our *AuxIn* script.

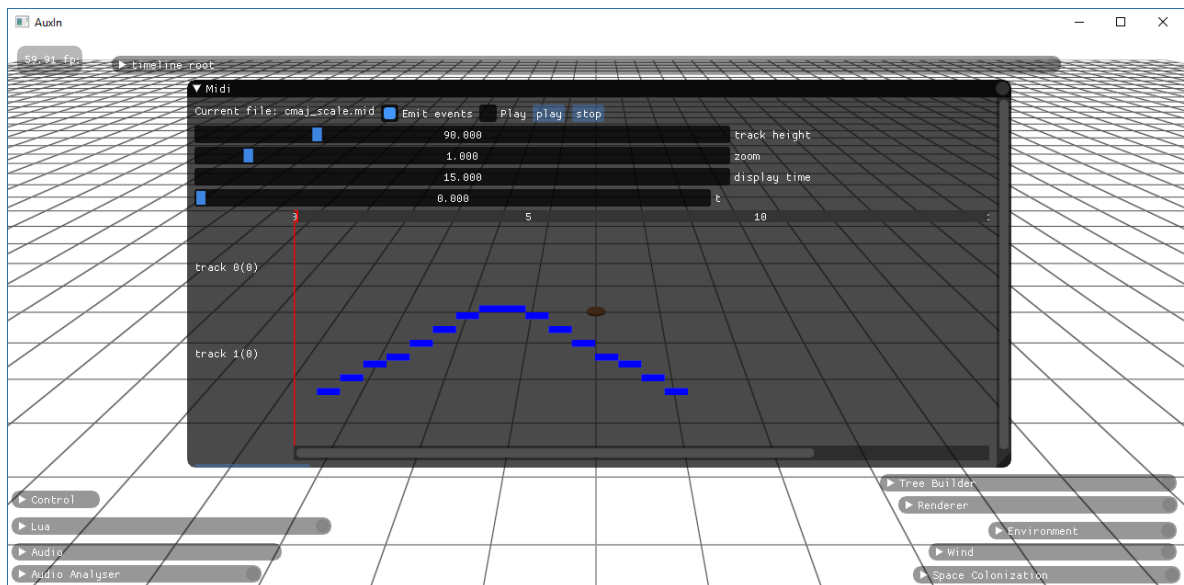


Figure 41: Examining the MIDI file on AuxIn

Note that in both figures 40 and 41, the notes are not evenly spaced. This is to be expected since we created a major scale, whose notes are not uniformly spaced, unlike the chromatic scale mentioned in 2.1.4, but instead follow a particular interval pattern. This pattern can be represented by the character sequence *W**W**H**W**W**W**H* where *W* stands for whole step<sup>2</sup> (i.e. 2 semitones) and *H* stands for half step (i.e. 1 semitone). This is the same interval pattern as followed by the white keys on conventional pianos, since this scale contains neither sharp nor flat notes, which correspond to black keys.

#### 4.1.2 Creating the Script File

Since we will be generating audio from the MIDI file, we may now move on to create our script file, which will determine exactly what happens during the animation. Since this is plain-text lua source file, any editor can be used to create it. Let us begin with a small step above the smallest example possible shown in listing 4.1. (This code would be equivalent to having both the handlers and info tables empty).

```
require "auxin"

function onInit()
    print ("C major scale")

    handlers =
    {
        [2] = {}
    }
}
```

<sup>2</sup> Also referred to as a *major second interval*.

```

    info =
    {
        [2] = { note = 0, lastTime = 0, lastNote = {}, chordNotes = {} },
    }

    timeline:setEnabled(true)
    t = 0.0
end

function onUpdate(dt)
    t = t + dt
end

```

Listing 4.1: Minimal auxin lua script file.

The `auxin.lua` file which we import on the first line consists of a small lua module for interfacing with our program. Although this import is not strictly required to create a valid script file for our program, it greatly simplifies this task. It contains our standard definitions for the `onNoteOn` and `onNoteOff` functions (which usually do not require modifications between projects), as well, as all auxiliary functions for defining note ranges (as discussed in 3.4) and other helpers.

What remains to be implemented are the functions specific to our example, namely the `onInit` and `onUpdate` functions. `onInit` will be called once, when the animation begins and `onUpdate` will be called on every frame.

In the `onInit` function, we initialize our `handler` structure with no behaviour, as well as the `track info` structure, whose purpose will be detailed later. The `handler` structure will have a subtable for each of the tracks defined in the MIDI file, which we intend to animate, as we describe in section 3.4.1. We also enable the global `timeline`. Since this function may be called repeatedly in order to restart the animation, we also take care to reset the global time, `t` to zero.

The `onUpdate` function has the standard behaviour of accumulating the time step `dt` in the global `t` variable so we can keep track of absolute time (i.e. elapsed time since the start of the animation).

This file is the minimal working example and as such could be considered as a script skeleton for any animation.

#### 4.1.3 *Creating Our Musical Animation*

As for our example mapping, in order to keep everything minimal, we will trigger a full iteration and the growth of all tree branches at the start of every note on our scale. To do this, we will borrow the `full_growth` function from our other examples:

---

```

function full_grow(trackInfo, key, vel, duration)
  nodes = TreeGrower["iterate"](0)

  for k, v in pairs(nodes)
  do
    v.lengthSequence:setStartTime(t)
    v.lengthSequence:addTween(TreeGrower["full_length"]:get(), t, duration,
      easings["easeOutCubic"])
    v.lengthSequence:setEnabled(true)

    timeline:addItem(v.lengthSequence)
  end
end

```

Listing 4.2: The full\_growth function.

This function's signature is common to all functions which react to a `NoteOn` event. `trackInfo` refers to the subtable of the `info` structure we show in listing 4.1 and is usually used to access the current note index (i.e. its first field). Next, we have the three attributes of the `NoteOn` event, namely the note number, key, the velocity (i.e. intensity) and the event's duration.

This note-to-growth assignment is used in both our examples, which will be briefly describe in sections 4.2 and 4.3.

The first line requests an iteration of the `SCA` and stores the set of newly produced nodes on the global variable `nodes`. Since we want every branch to grow to their maximum length in a single step, we only need to add a single tween to each branch's `lengthSequence` `TweenSequence` object (see section 3.5.3 for more detail).

The next step is iterating the new segments adding the aforementioned tween, using each node's `addTween` function.

Lua's unusual implementation of the `for` loop comes from its core concept that everything is a table. Even plain arrays, for instance, are tables with integer keys. As such it is standard to use the `pairs` function when iterating a table, which returns a new key/value pair on every iteration. In our case, `v` is the `GrowNode` instance we wish to access, and `k` is its key in the collection, which we do not need<sup>3</sup>.

The `addTween` call appends a new tween to the `TweenSequence` object of each node, which we mention in 3.5.3.

The first argument for `addTween` is the **target value** (length in this case), to which we assign the max-length parameter, obtained from `TreeGrower`'s `full_length` attribute<sup>4</sup>. Next we have the **starting time**, which we set to the global time `t`, followed by the **duration**, obtained from the event, and lastly the desired **easing function**. Since this `TweenSequence`

<sup>3</sup> It is generally regarded as good practice to replace unused return values with `_` for clarity, as such this condition would become `for _, v in pairs(nodes)`

<sup>4</sup> Since this value is unlikely to be changed during the animation it could be cached in a global variable at initialisation.

object was already created referring to the segment's length, this information does not need to be specified in the `addTween` call.

By assigning the starting time to `t`, and `duration` as the note event's duration, this tween will start immediately, that is exactly as the note rings, and will last until it ends.

Now, the only remaining step towards having a working animation script is determining *when* this behaviour is triggered. To do so, we assign it to the intended note interval by modifying the handlers structure.

```
require "auxin"

function onInit()
  print ("C major scale")
  t = 0.0

  handlers =
  {
    [2] = {pred = noteRange(1, 16), note_on = full_grow }
  }

  info =
  {
    [2] = { note = 0, lastTime = 0, lastNote = {}, chordNotes = {} },
  }

  timeline:setEnabled(true)
end

function onUpdate(dt)
  t = t + dt
end
```

Listing 4.3: Assigning our function to the correct note interval.

As mentioned in section 3.4, the `pred` field is a predicate function which determines if the corresponding `note_on` will be called at each received event. In this example, this field is assigned a *closure* which will return true if the note index is between 1 and 16 (for more information on closures see section 3.4.2).

All we are missing to get our first animation is to ensure the call to `iterate` works as intended by defining the set of attraction points for the *SCA*. This is done in the application<sup>5</sup>. We obtain our points from the uniform sampling of a sphere as we show in figure 42.

And that is all, the animation is working as intended and some frames of the final animation are shown in 43<sup>6</sup>.

<sup>5</sup> We plan to move this process to the script file.

<sup>6</sup> The video is available at [https://youtu.be/tWdd\\_f5jYzs](https://youtu.be/tWdd_f5jYzs)



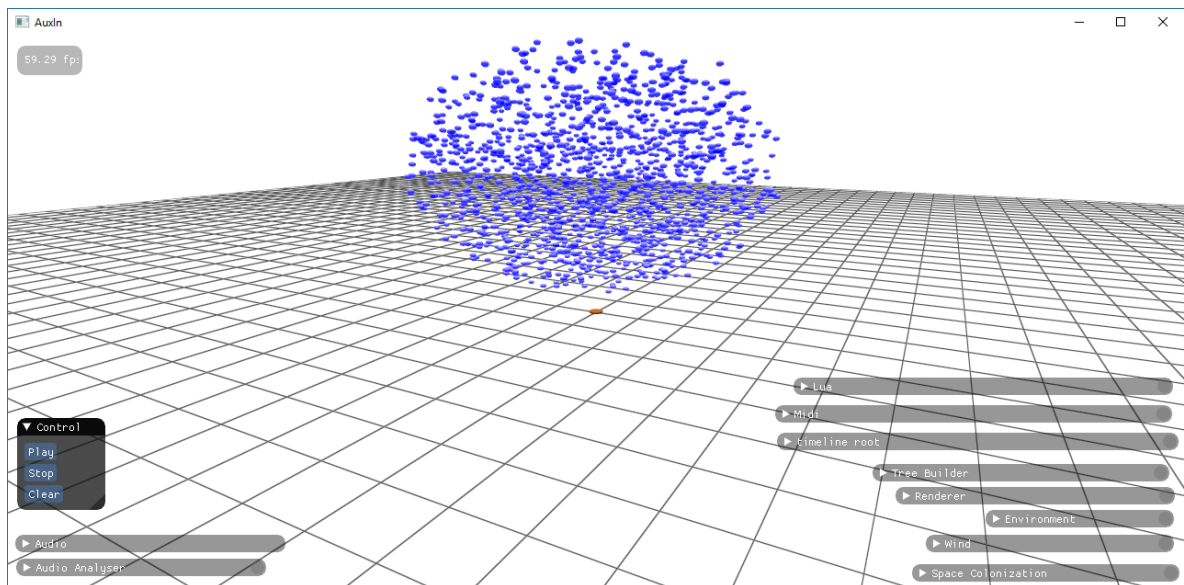


Figure 42: AuxIn with a spherical point cloud.

#### 4.1.4 Adding Wind

The previous example showed how to make the tree grow synchronised to each note. Now we will add the other main effect we implemented, which is wind. To do so, we revisit [MidiEditor](#) in order to add a chord which rings simultaneously with the last note of the scale. We also place this chord in a new track, as can be seen in figure 44.

We made sure to add this chord at the end of the scale since by then tree has completed its growth and the wind effects will be most noticeable.

In order to add wind our current script file requires three modifications. We first need to create a wind primitive, as referred in section 3.10, which we will do at the `onInit` function.

```
function onInit()
  ...

  uniformWind = Wind["add_uniform"](vec3.new(0.0, 1.0, 0.0), 0)
end
```

Listing 4.4: Adding uniform wind.

With this instruction, the global variable `uniformWind` is available and refers to a purely vertical wind, initialised with no strength.

Now, we move on to creating the function that triggers the actual "blast" of wind, using this primitive. It is more succinct than the `full_growth` function since we can implement it by creating a single tween.

```
function wind_pull(trackInfo, key, vel, duration)
```

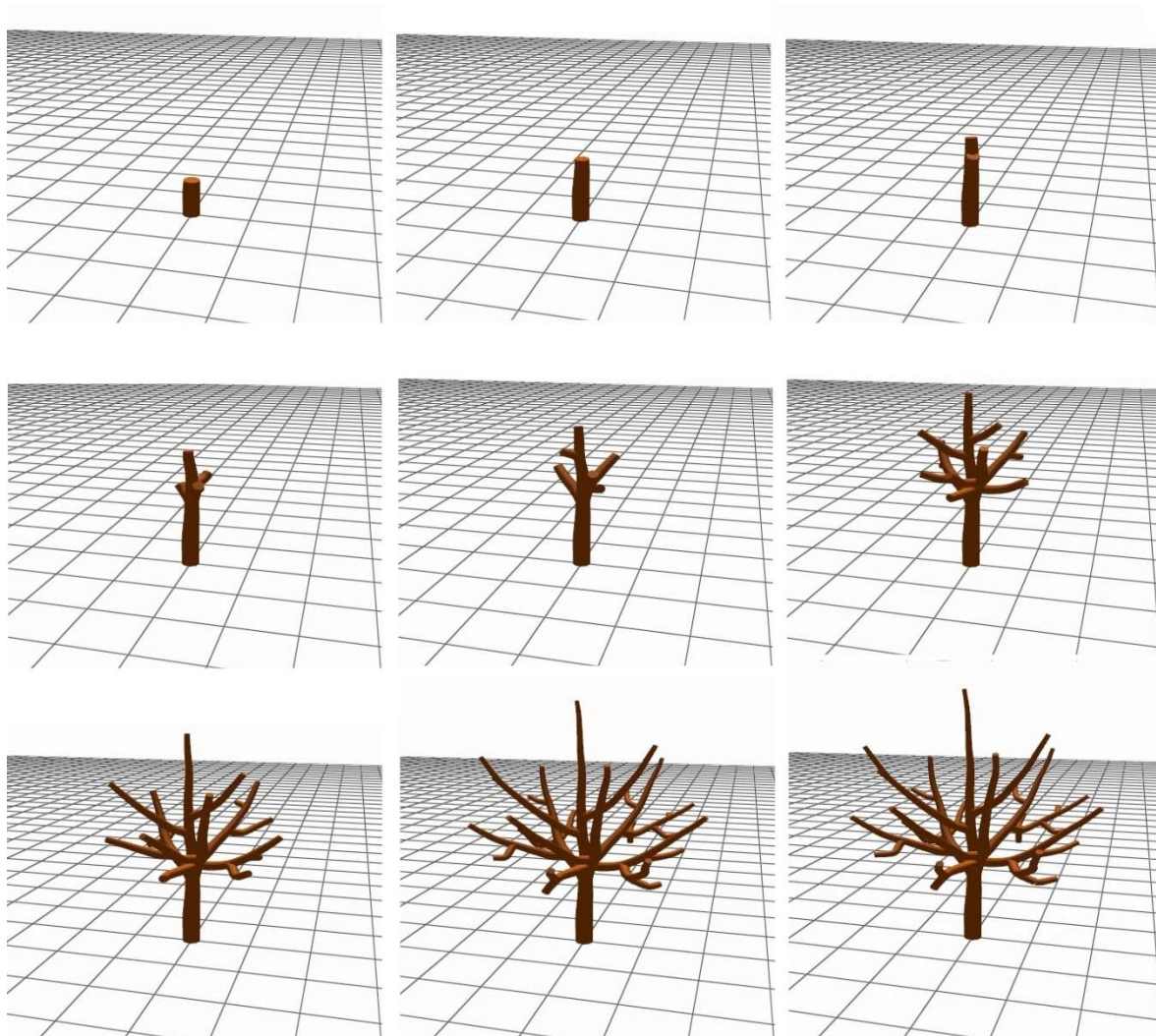


Figure 43: Frames from our first animation.

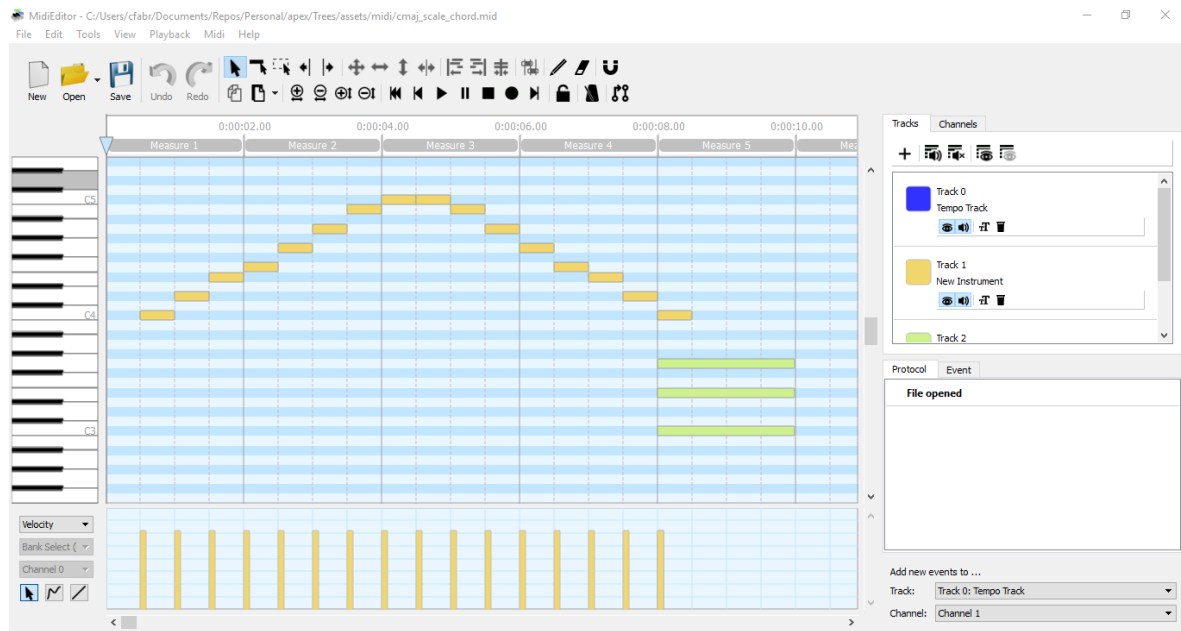


Figure 44: Adding a simple chord to our MIDI file.

```

local tween = TweenF.create("wind_pull", uniformWind:attr_float("strength"),
    1000, 0, t, duration, easings["easeOutCubic"])
tween:setEnabled(true)
timeline:addItem(tween)
end

```

Listing 4.5: Wind pull.

The arguments for the creation of a tween have already been described in section 3.5.2. Comparatively to the growth tween of the previous section, we must now specify the actual *attribute* this tween acts upon, which we obtain by requesting the **strength** attribute from our global wind primitive. We assign it the initial value of **1000**, which will drop to **0** throughout the event's **duration**, using a **cubic ease out** function. This choice of easing function translates into a wind force which begins at its peak intensity, followed by a fast decay.

Lastly, as we did with the tree growth, we must assign this change to be triggered at the correct moment, so we revisit the handler and info structures.

```

handlers =
{
  [2] = {
    {pred = noteRange(1, 16), note_on = full_grow },
  },

  [3] = {
    {pred = noteRange(1, 3), note_on = onChord(3, wind_pull) },
  }
}

```

```

}
info =
{
  [2] = { note = 0, lastTime = 0, lastNote = {}, chordNotes = {} },
  [3] = { note = 0, lastTime = 0, lastNote = {}, chordNotes = {} }
}

```

Listing 4.6: Wind pull.

Since we wish this effect to occur in the third track, we must add an extra entry to both tables.

These are all the required changes to add wind to our previous example. We include frames of the resulting animation in figure 45<sup>7</sup>.

We also take this opportunity to discuss the limitations of using the note count as the index for triggering effects, as well as our practical solution, which revolves around the `onChord` closure in listing 4.6.

Since we currently process events individually, lua receives three `NoteOn` events when our chord is sound. As a result, their order of arrival is undetermined. If we wish for a function to be called only once during these three notes we make use of our rudimentary chord detection. This method operates by continuously storing notes which occurred within a very short time interval. This is why the three last fields in the info subtables exist. The `onChord(3, wind_pull)` translates to "only execute `wind_pull` if a three-note chord has been detected". Unfortunately, limitations remain since this method makes it impossible to distinguish between a complete 2-note chord from an incomplete 3-note chord. This could be solved by a preprocessing step by part of the host program, which would wait for a very small amount of time before firing every event, in order to distinguish individual notes from chords. In this particular example, since we only have one chord, another practical solution would be to treat it as a single note. However, this would not extend to an actual chord track without individually specifying a single note for each chord (this could be achieved using the `noteSet` closure, which accepts a list of notes).

#### 4.1.5 Adding Leaves

We make another addition to our example by adding the growth of leaves at the end of the animation. The same pattern from the two previous effects applies: we first define an appropriate function and then determine when to invoke it.

This function is the most complex so far since it makes use of a few more advanced features of our framework, particularly the selection (i.e. filtering) of tree nodes, which is where leaves are placed.

<sup>7</sup> Video available at <https://youtu.be/bA4EW4AmJMs>

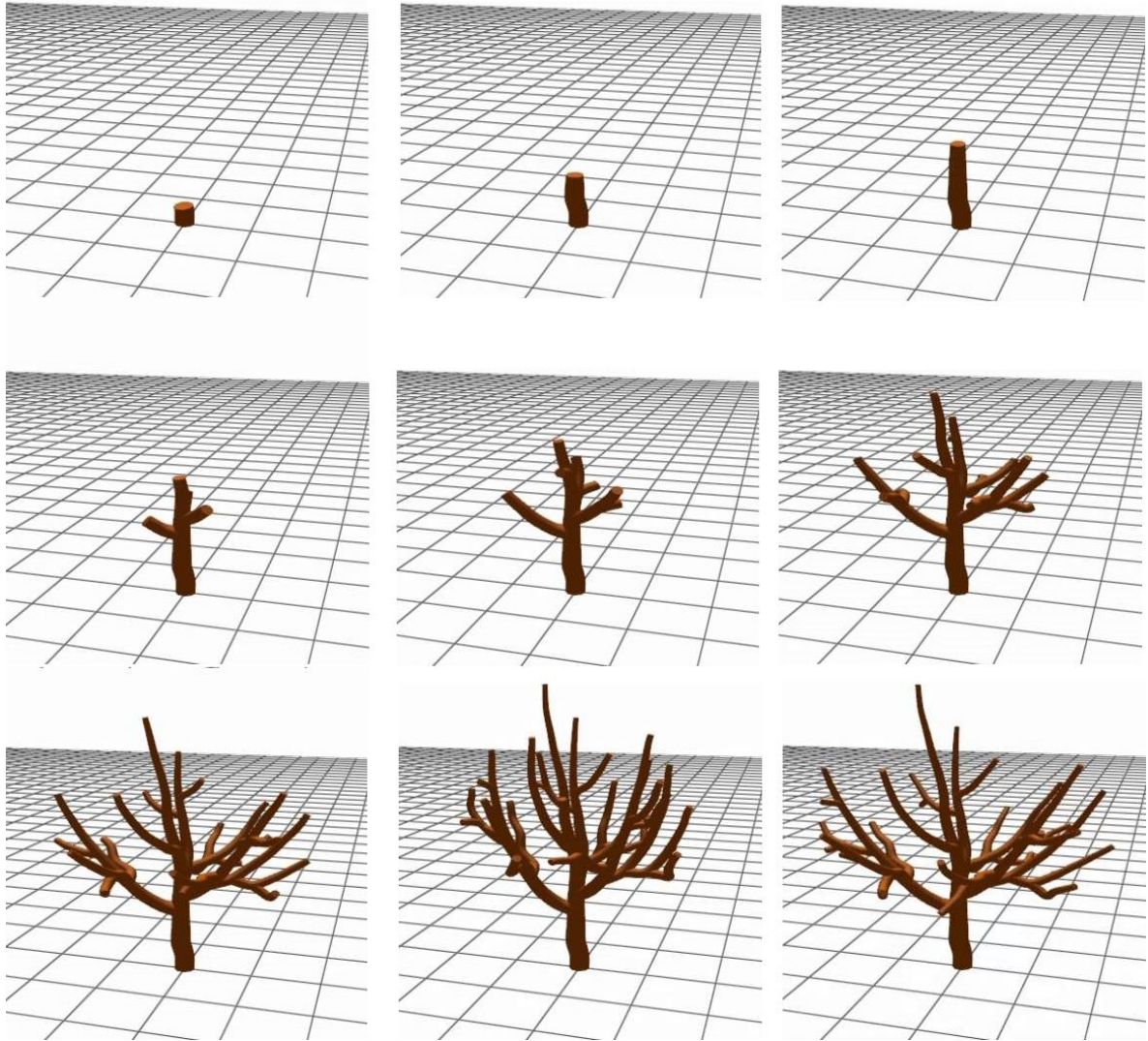


Figure 45: Our example with a blast of vertical wind at the end



```

function place_leaves(trackInfo, key, vel, duration)
  local tree = TreeGrower["tree"]()

  local nodes = tree:filter(
    function (node)
      return node.order >= 1
    end)

  for k, v in pairs(nodes)
  do
    if(not v.leavesTween:isEnabled())
    then
      v:placeLeaves(math.random(3, 5))

      v.leavesTween:setStartTime(t)
      v.leavesTween:setDuration(duration*2)
      v.leavesTween:setEaseFunction(easings["easeOutCubic"])
      v.leavesTween:setEnabled(true)
      timeline:addItem(v.leavesTween)
    end
  end
end
end

```

Listing 4.7: Add leaves function.

First, we must acquire a list of all nodes we want to place leaves on. We do this by obtaining the root `GrowNode` and then selecting all its descendants which satisfy a given predicate using the `filter` function. The condition, in this case, is having an order of branching greater or equal to one<sup>8</sup>. This effectively prevents the addition of leaves to the main trunk.

Afterwards, we iterate over the selected nodes, calling the appropriate `GrowNode` function, `placeLeaves` which takes as its single argument the number of leaves, which we randomise using lua's built-in pseudo-random number generator<sup>9</sup>. We then proceed to initialise the `GrowNode`'s `leavesTween` which is similar to `lengthSequence`. However, this is a not a `TweenSequence` object, but instead a plain `tween`. As such, we individually set its multiple parameters and add it to the timeline.

We now move on to the second half our last effect, which, as usual, consists in adding the new function to the handler structure:

```

handlers =
{
  [2] = {
    {pred = noteRange(1, 16), note_on = full_grow },
    {pred = singleNote(16), note_on = place_leaves }
  },

```

<sup>8</sup> In this context, order refers to the **classic stream order**, which begins at 0 for the tree trunk and increases with every branching level.

<sup>9</sup> The arrangement of leaves is determined automatically.

```
[3] = {
  {pred = noteRange(1, 3), note_on = onChord(3, wind_pull) },
}
}
```

Listing 4.8: Placing leaves at the final note.

The effect is now functioning, and the resulting frames are shown in figure 46<sup>10</sup>. Note that changing this effect to, for instance, adding leaves with every note could be achieved simply by altering the predicate function from `singleNote(16)` to `noteRange(1, 16)`. This is possible because our `place_leaves` ensures not to add leaves to nodes which have had leaves previously added<sup>11</sup>.

#### 4.1.6 Audio Reactivity

All we are missing in order to showcase most of the possibilities we implemented, is the use of audio information. This poses a small problem since currently we only have MIDI information available and our system does not have access to the audio data it produces using the operating system's software synthesizer. In order to bypass this problem, we will resort to yet another excellent freely available open-source software, *Audacity*. Audacity is able to capture all audio sent to the system's audio devices, so we will use the MIDI playback feature provided by *MidiEditor*, and record it in an actual ogg audio file, as shown in figure 47. Although this data is not as interesting as real audio data, it bears a very strong resemblance and will suffice to illustrate our system's abilities.

Another problem arises, as it often does with audio data, which is that of synchronisation. Although in this case, the problem is not severe since every note is correctly spaced. As such, we only need to adjust the beginning of the first note, to match that of the MIDI. We can determine the start time in seconds of the first note using a little bit of music arithmetic. Our intricate composition has its tempo set at 120 BPM. Having a tempo of 120 beats per minute means there are two beats per second, and conversely half a second per beat. Since the time signature is 4/4 time, then the length of a beat corresponds the quarter note. This means each quarter note corresponds exactly to 0.5 seconds. Since our scale begins playing exactly after a quarter rest, then all we need to do in order to synchronise our audio file is select the region beginning at the first note and shift it to start at 0.5s.

As we mention in section 2.1.1, an audio representation is very rich in information, as it encodes the real-time temporal, dynamic, and tonal microdeviations present in a musical performance. Currently, our system explores only the spectral representation of the audio

<sup>10</sup> Video available at <https://youtu.be/Vb3qPSItAdw>

<sup>11</sup> This check could also be performed directly on the `filter` function so as to exclude nodes with leaves from the selection.

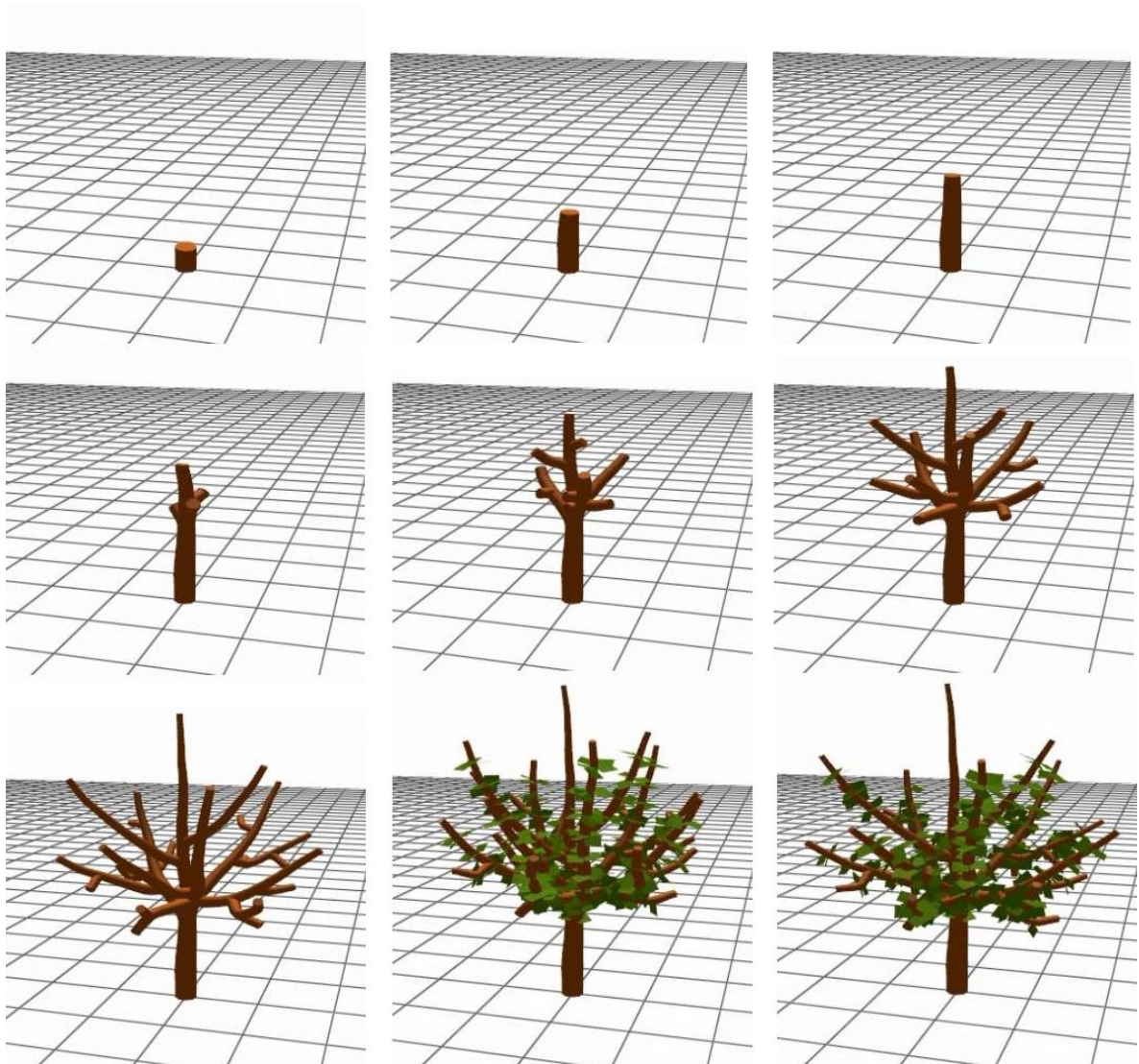


Figure 46: Our animation now ending with the growth of leaves.



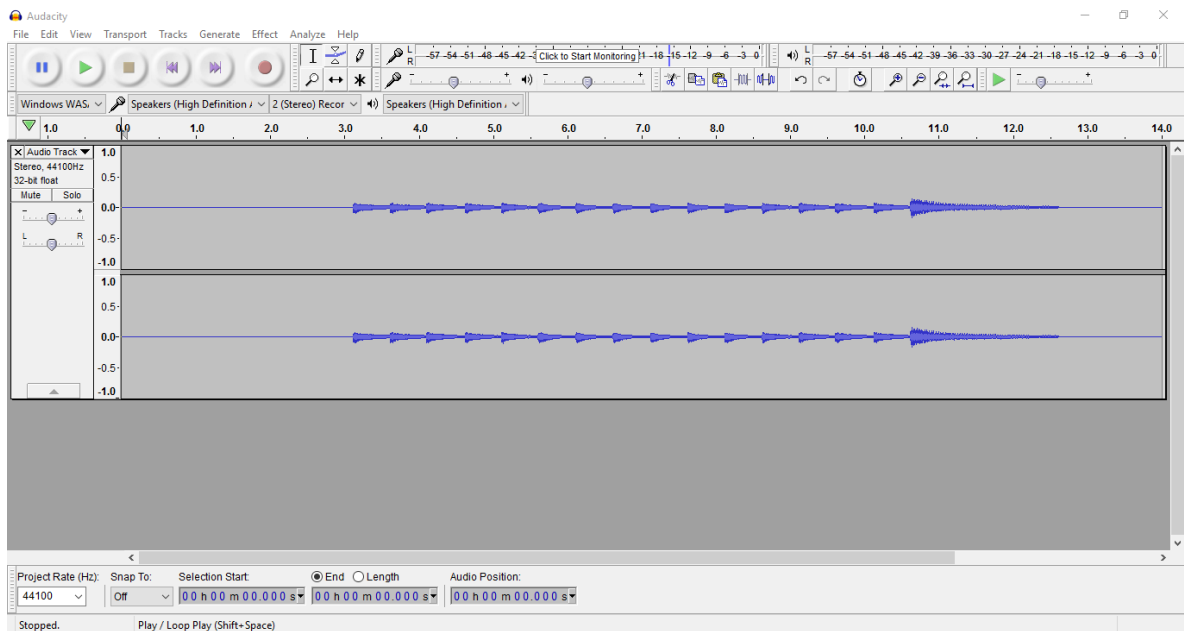


Figure 47: The audio resulting from our simple MIDI.

signal, obtained from the [STFT](#), which by itself already provides interesting information. This data is calculated every frame, as we show in [figure 48](#).

As is observable in [figure 48](#), it is clear when each note rings and when the chord is reached. In every note, several peaks are present, each of which corresponds to a partial, as we have discussed in [2.1.9](#). On the last two frames these peaks are particularly abundant, since they correspond to the more intricate superimposition of all the three notes ringing simultaneously.

Internally, this data is simply a large array which in practise assigns a magnitude coefficient to every frequency. The vast topic of signal processing is beyond the scope of this tutorial, but we have used 16384 samples for each execution of the [STFT](#). This carries little overhead in our system and already provides a fairly detailed spectrum.

We can use this data in any way we wish. For our purpose we found the global spectrum average to be an interesting candidate, since it provides a measure of intensity of the audio, as it peaks when each note sounds, and in particular at the ending chord.

Now the question arises on how to represent this number in a visually appealing way. Since it is a value calculated at every frame, candidates would be values whose variation is also continuous, such as wind strength or growth rate. However, these attributes are already being controlled by tweens.

For this example, we chose to use the **displacement** attribute defined in the tree builder module. This attribute is simply a fixed numeric value which scales the thickness of every branch in the tree. As such it produces a swelling/shrinking effect whether it is greater or



Figure 48: The evolution of the audio signal's spectrum in a logarithmic scale, as calculated by AuxIn.

smaller than one, respectively. Since this value affects the whole tree, mapping the spectrum average to it provides a captivating pulsating effect.

In order to implement this effect, we will make our first changes to the `onUpdate` function.

```
function onUpdate(dt)
  t = t + dt

  local spect_avg = AudioAnalyser["spectrogram_average"]()

  TreeBuilder["displacement"]:set(0.5 + spect_avg * 2.0)
end
```

Listing 4.9: Assigning the spectrum average to trunk displacement.

Those two lines are all that is necessary to add basic audio reactivity, and frames are displayed on 49<sup>12</sup>. We give the displacement a minimum value of 0.5, which is why the tree appears “thinner” than in the previous images: its thickness is halved when the audio intensity is low. Unfortunately, the pulsating effect is not illustrated in the captured frames.

We give the displacement a minimum value of 0.5 to which we then proceed to add the spectrum average times 2. These sort of “magic numbers” can be determined by experimentation to find the most interesting combinations.

A quick but relevant optimisation is caching access to both the `spectrogram_average` operation and displacement attribute, by assigning each to a global variables.

```
getSpectAvg = AudioAnalyser["spectrogram_average"]
displacement = TreeBuilder["displacement"]
...

function onUpdate(dt)
  t = t + dt

  local spect_avg = getSpectAvg()

  displacement:set(0.5 + spect_avg * 2.0)
end
```

Listing 4.10: Caching AudioAnalyser’s attributes.

This avoids a map retrieval operation on every frame, which is significant.

Many more possibilities exist. For instance, this effect can be achieved identically for leaves, using `leaf_size` instead of `displacement`. Another example would be colour. We could determine two colours and interpolate between them according to a modified spectrum average, or adjust one of its components, such as brightness or saturation. Implementing effects such as these would not pose many more difficulties than this example.

<sup>12</sup> Video available at <https://youtu.be/pRI5vRG1nEY>

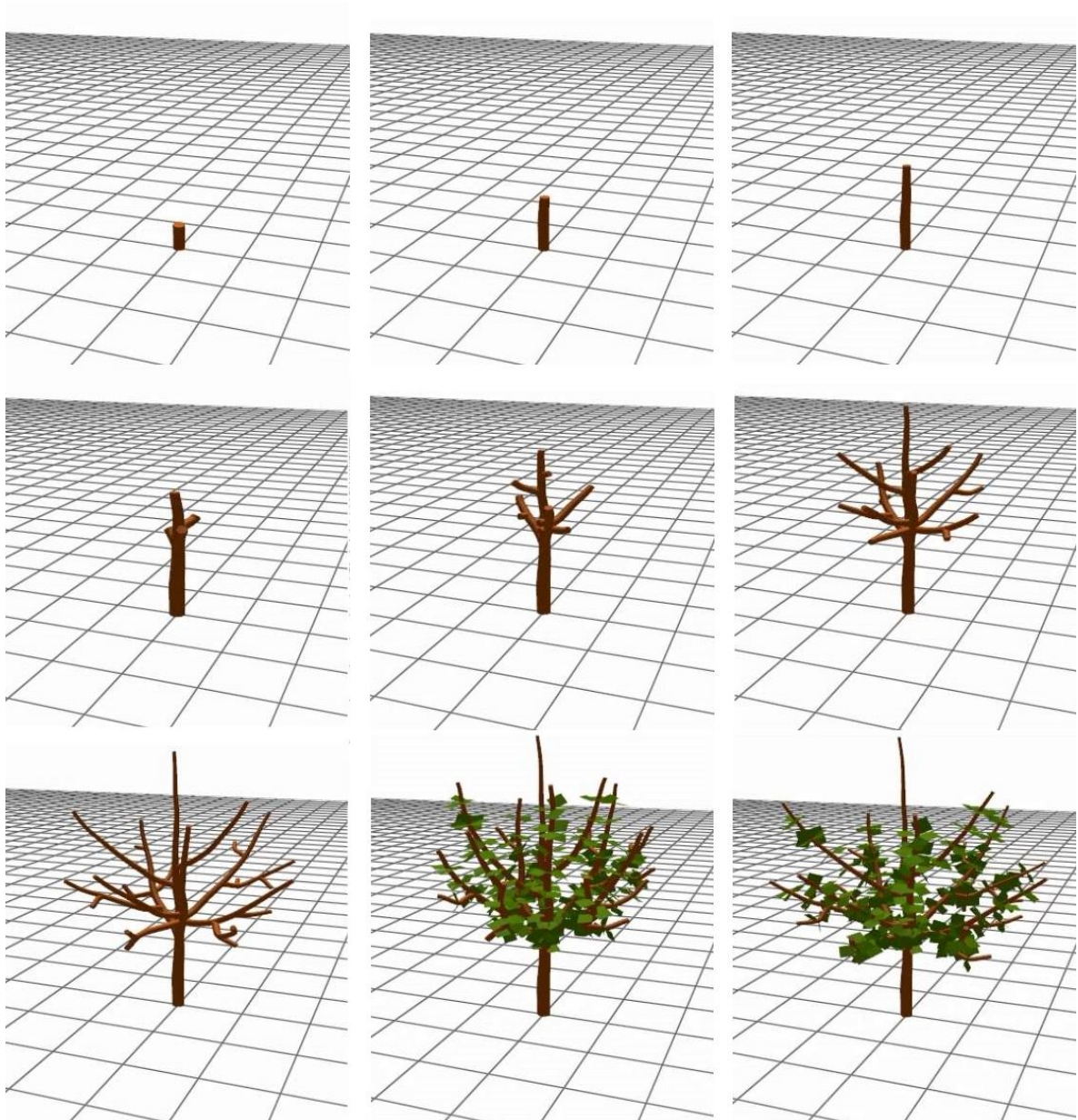


Figure 49: Our animation with audio reactivity.

A limitation of implementing this effect directly on the update function is the fact that it will always be active, unless we use a lua mechanism, such as a *Boolean* flag. Since we have a system that thrives on the timed addition of effects, in order to assist in doing the same for this family of continuous changes, such as audio reactivity, we have developed another concrete derivation of *TimelineItem* called *DataLink*. These objects are templated and are equipped with two function pointers, which we dubbed *extract* and *apply*. As their name suggests, the *extract* function is a function which retrieves a value and the *apply* function does something with it, as we describe in section 3.5.4.

Keeping in mind that we could create a *DataLink* under any circumstance (as we do with *Tweens*) we will create it at initialisation in order to maintain the same behaviour as before:

```
function onInit()
  ...

  uniformWind = Wind["add_uniform"](vec3.new(0.0, 1.0, 0.0), 0)

  local pulseLink = DataLinkF.create("pulse_link",
    function (dt) return getSpectAvg() end,
    function (dt, v) displacement:set(0.5 + v * 2.0) end)

  pulseLink:setEnabled(true)
  pulseLink:setInfinite(true)

  timeline:addItem(pulseLink)
  timeline:setEnabled(true)
end
```

Listing 4.11: Implementing audio reactivity with a *DataLink*.

The *pulseLink* variable is handled similarly to the other *TimelineItems*. Since we won't be changing it throughout the animation we declare it as local to initialisation, but if we were to declare it as global it could be manipulated anywhere in the script. Using separate *extract* and *apply* functions we individually capture the two distinct behaviours which were previously merged, providing a more decoupled implementation. A possible improvement for *AuxIn* would be allowing these "magic numbers" to be adjusted in real-time. To do so it would be necessary to extend our program so as to give the script module access the program's graphical user interface and manipulate it. Should this be accomplished, the lua file would be able to define custom user interfaces for each project, which could provide many interesting possibilities for live accompaniment.

#### 4.1.7 Colour and More Wind

We conclude our example with two final touches. First, we will make leaves gradually change their colour at the end of the animation. Second, we will add a second wind primitive, now a vertical vortex wind, which will cause the tree to spin around its trunk.

First, we will have our leaves appear with every new note, as opposed to only once at the end. We mentioned how small a change this is in terms of code in section 4.1.5:

```
handlers =
{
  [2] = {
    {pred = noteRange(1, 16), note_on = full_grow },
    {pred = noteRange(1, 16), note_on = place_leaves }
  },
  [3] = {
    {pred = noteRange(1, 3), note_on = onChord(3, wind_pull) }
  }
}
```

Listing 4.12: Placing leaves at every note.

Leaf colour is a global attribute present in our `TreeBuilder` class. It is represented as a `vec3`, that is, a three-component vector, representing red, green and blue components respectively. Modifying it is no different from modifying any other parameter and since linear interpolation is also defined for `vec3` objects, we can easily produce a simple colour gradient. We follow the same pattern as we have thus far. We first define our new `leaf_color` function:

```
function leaf_colour(trackInfo, key, vel, duration)
  local tween = TweenVec3.create("leaf_color", TreeBuilder["leaf_color"],
    TreeBuilder["leaf_color"]:get(), vec3.new(1, 0, 0), t, duration, easings["
    easeInQuad"])

  tween:setEnabled(true)
  timeline:addItem(tween)
end
```

Listing 4.13: Changing the colour of the tree's leaves

Now that we are dealing with a `vec3` object, we must use the appropriate Tween constructor, namely `TweenVec3`.

We access the attribute object as we did for displacement in section 4.1.6, by indexing `TreeBuilder`'s global table. As initial value, we specify the current value and we make the change to pure red.

All that's left, as usual, is defining when to call this function. We schedule it to occur simultaneously with the `wind_pull` function.

```

handlers =
{
  [2] = {
    {pred = noteRange(1, 16), note_on = full_grow },
    {pred = noteRange(1, 16), note_on = place_leaves }
  },
  [3] = {
    {pred = noteRange(1, 3), note_on = onChord(3, wind_pull) },
    {pred = noteRange(1, 3), note_on = onChord(3, leaf_color) }
  }
}

```

Listing 4.14: Changing leaf colour at the sounding of the final chord.

The effect is complete and its frames are displayed on 50<sup>13</sup>. Note that due to our `place_leaves` implementation shown on listing 4.7, leaves will only effectively appear when segments with *order* greater or equal to 1 are present, that is, after the first branching occurs.

The vortex wind is added similarly to the uniform wind, except for the different initialising function.

```

function onInit()
...

uniformWind = Wind["add_uniform"](vec3.new(0.0, 1.0, 0.0), 0)
vortexWind = Wind["add_vortex"](vec3.new(0.0, 0.0, 0.0), vec3.new(0.0, 1.0, 0.0), 0)
...
end

```

Listing 4.15: Placing leaves at every note.

The vortex wind is characterised by different parameters, namely the point of origin, where vortex strength is maximal, which we place at the origin, and the axis, which we make vertical. We show a slice of the wind field for illustrative purposes in figure 51.

Now that the primitive has been created we need to make a function to trigger the activation of this wind field, similarly to `wind_pull`, which we will call `wind_spin`:

```

function wind_spin(trackInfo, key, vel, duration)
  local tween = TweenF.create("wind_pull", vortexWind:attr_float("strength"), 200
    * key, 0, t, duration, easings["easeOutCubic"])
  tween:setEnabled(true)
  timeline:addItem(tween)
  vortexWind.direction.y = -1 * vortexWind.direction.y
end

```

Listing 4.16: Triggering the vortex wind

<sup>13</sup> Video available at <https://youtu.be/HGdNHirLLfM>



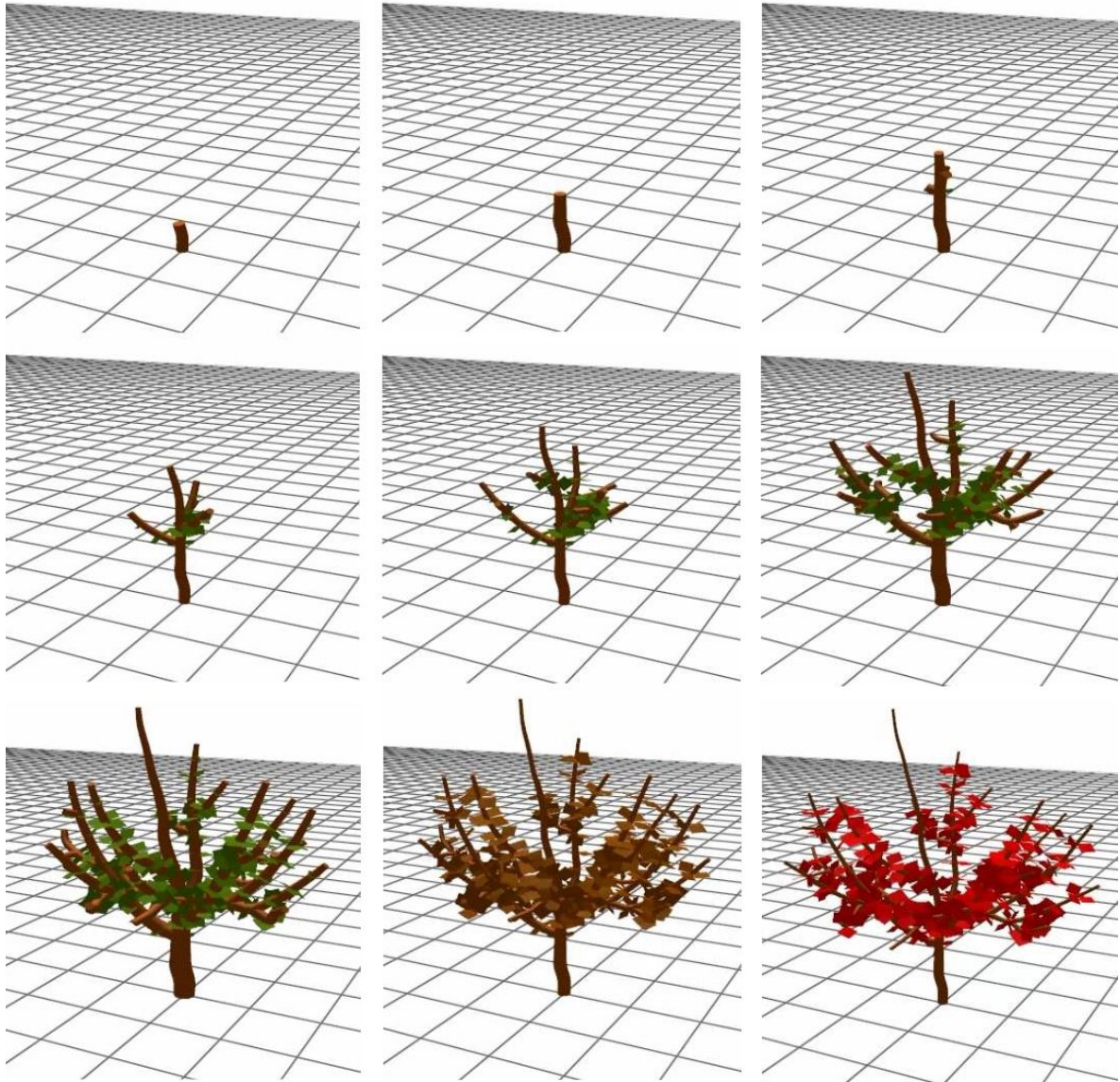


Figure 50: Gradual addition of leaves and color change.



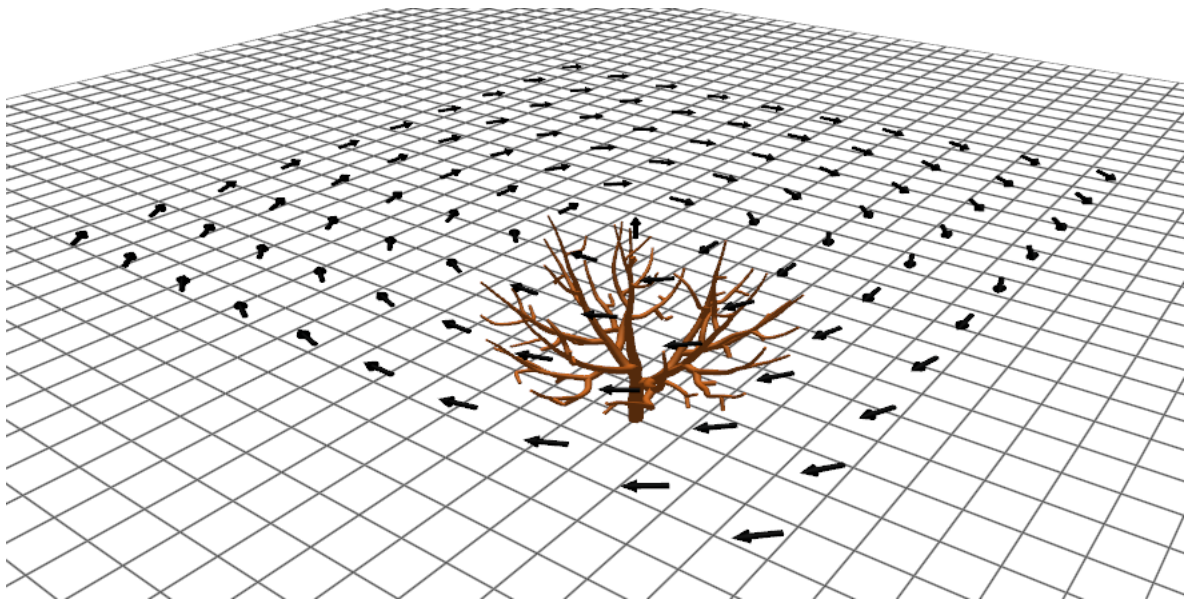


Figure 51: A single slice of our new vortex wind primitive.

We also take this opportunity to use the key argument for the first time. This is the MIDI note number. Since we are playing the C major scale, we go from C<sub>4</sub> up to C<sub>5</sub>. This translates to the key value varying from 60, up to 72. By using this value as our starting wind strength at every “wind spin”, the tree will spin most intensely as we reach the highest note. We also flip our vortex by inverting the y component of its direction, which translates to alternating between spin directions on every note. Although the effect is not very noticeable in the still images, we show frames of the animation on figure 52.<sup>14</sup>

## 4.2 KOTOWARI

*Kotowari* is the eighth song of the soundtrack from the Japanese anime television series, *Mushishi*. This song was used as our running example throughout the implementation.

We found *Kotowari* to be particularly suitable as a test case due to its small duration of 70 seconds, as well as having two main contrasting melodic lines which are rich in expressiveness and consequent interpretation. The two melodic tracks allowed us to explore the two effects we dedicated our implementation to, namely the animation of tree growth and its real-time reaction to a changing wind field.

In order to use this song, we obtained a freely available transcription which we later synchronised manually to the audio data. This allowed us to use features from both representations.

<sup>14</sup> Video available at <https://youtu.be/XYZTcN5LnjU>

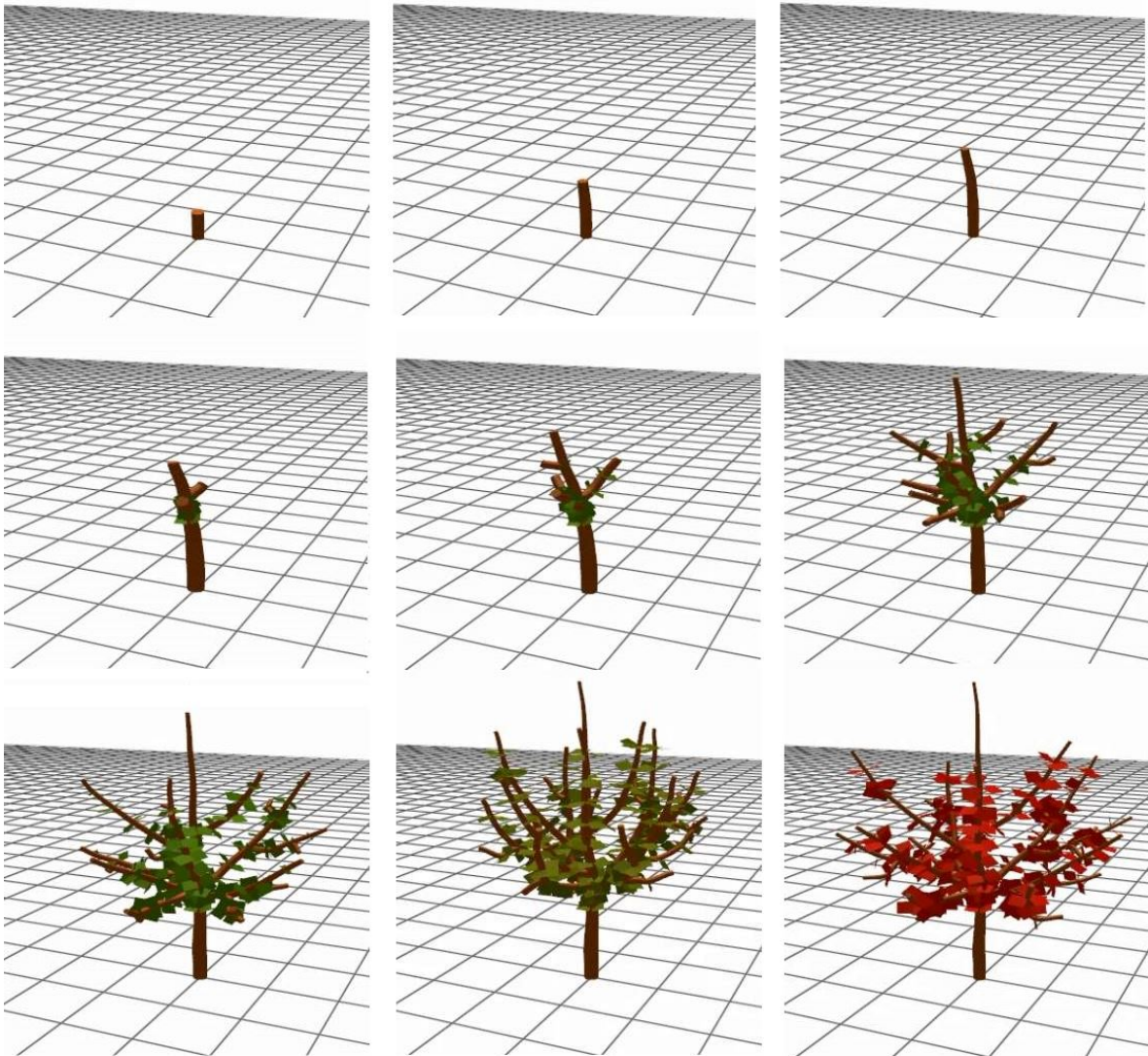


Figure 52: Our example with vortex wind.

The higher pitched melodic line was assigned to tree growth, while the lower one was assigned to control the wind field. A third melodic line exists, which corresponds to the unexpected ringing of a bell, which we mapped to the appearance of leaves.

This is, of course, our own subjective mapping of the song, and any other would also be possible, depending on the input script file. Furthermore, nothing prevents these mappings from being changed in different moments of the song, since they are established on a per-note basis as we illustrated in the tutorial section.

### 4.3 NAVORSKI

*Navorski* is our code name for the song "Viktor's Tale" by John Williams, composed for the film "The Terminal" directed by Steven Spielberg. This song is 4 minutes long and as such, the creation of a "choreography" proved to be much more challenging and time-consuming than for our previous example. A large part of the difficulty arises in engaging the audience and holding their attention.

In this example, there is a small set of rules we defined to be fixed throughout the whole song, such as a downward motion on a bass chord note, and an upwards motion on the remaining chord notes. We find that in some moments these rules interact with temporary rules to form interesting patterns which begin to resemble dancing motions.

Due to the length of the song, the code becomes significantly more extensive. In order to obtain interesting behaviours, we make use of slightly more intricate algorithms to achieve what we found to be intuitive mappings of the different aspects of the song. For instance, there are scenes where we approximate the melodic motion of the song (ascending vs descending melody lines) and map it to vertical changes in the wind direction. We also make use of the twirling motion caused by a vortex wind, as we discuss in section 4.1.7.

However, properly assigning these effects to moments in the song has proved a challenging task. This is where we are confronted with strong evidence that a more adequate interface would prove useful. We also found that the implementation of distinct effects would greatly improve the quality of the final animations.

### 4.4 DISCUSSION

With this section, we hope to have given the reader greater insight regarding the inner workings of our application, what we aimed to achieve by creating it and its potential. We believe that the concept of supplying the user with a set of sophisticated, yet intuitive animation primitives, as well as access to musical and audio information can allow for the creation of imaginative and expressive works without requiring extensive prior knowledge. Moreover, we believe that our system would have a vast leap in usability if we were to

implement an interface following the visual programming paradigm as with [Max/MSP](#) or [Pure Data](#). To illustrate this, we drew inspiration from [Pure Data](#) and present a non-rigorous mock-up of what our tutorial project could look like if represented as a hybrid between code and visual representation, shown in [figure 53](#).

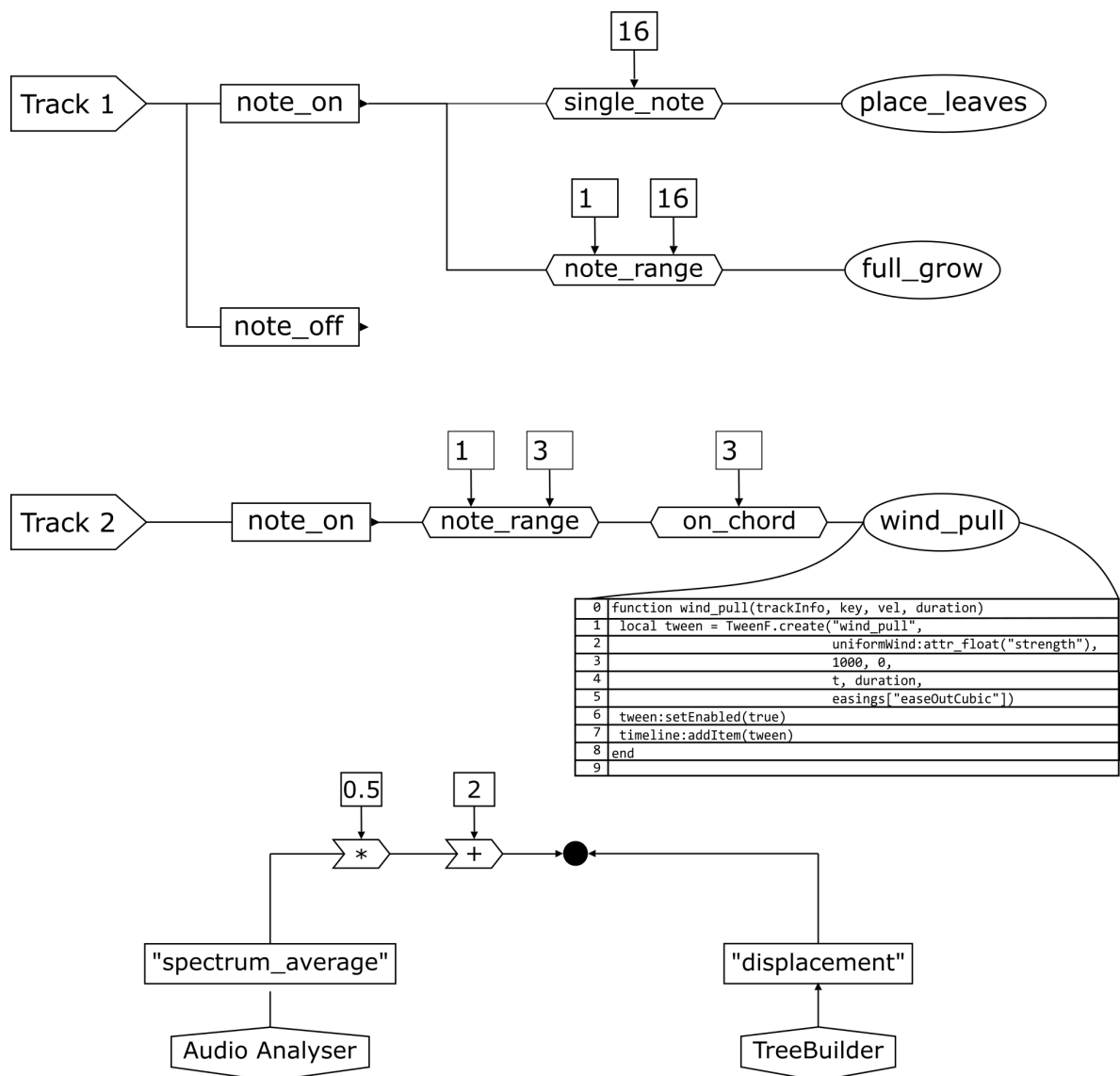


Figure 53: A concept mock-up of the tutorial project under a visual programming paradigm.

---

## CONCLUSION

---

In this project, we explored the challenge of creating a tool for deriving animation from musical information. This was made possible by three key elements: the simulation of a dynamic system, a configurable environment with which it interacts and a set of rules which direct changes in this environment from incoming data. In our case, the system is the growing tree simulation in the wind, the environment is the set of stimuli which the tree reacts to, the set of rules is represented by the script file and the incoming data consists of our supported representations of music.

In general terms, what this accomplishes is to provide a layer of abstraction from manually specifying individual changes in the animation, which is replaced with triggering changes in the environment. As such, we are effectively trading control for automation. In any context where there is abundant real-time data, this drastically lowers the effort required to translate such data into a meaningful visualisation and we believe this concept could be extended beyond music.

In our case study, we found this approach to be successful. Music structure is thoroughly described and analysed by a formal theory, however, most of its elementary aspects are perceived by the general population in an intuitive way. An example of such is dancing, which comprises the performing of motion usually associated with aspects of music, in particular rhythm, which gives rise to a particular form of expression. In a broad sense, this is what we strive for our application to be, a means of expression. In our digital approach, we provide the user with complementary, yet distinct, representations of music information and give them programmable control over a dynamic reactive system.

We found that imagining visual metaphors for music comes naturally to most people since our discussions regarding this application often brought with them fresh ideas for possible associations between our tree's reaction to moments in songs. The flow of ideas was particularly abundant from those who with greater affinity for music, and even more so from those who study it. This was both an exciting and motivating result since in this project we have barely scratched the surface of what could be accomplished by extending this concept in different directions. For instance by modelling different phenomena, introducing cinematography by capturing their evolution in meaningful ways as well as

extracting more cues from musical information. The ever-growing available power of computation continues to expand the sphere of possibilities of what computers may achieve, and many ideas remain delightfully open for future research, exploration and experimentation. We can only hope that our project will be able to incite further development into this realm of endless possibilities.

## 5.1 PROSPECT FOR FUTURE WORK

We can not over-emphasise how this project has merely scratched the surface of what we believe is possible to achieve by enriching this concept. In order to produce more captivating animations many features from many distinct fields could be incorporated into our system.

As development progressed we found countless directions in which this idea could be further extended and improved, some of which we will proceed to enumerate.

An interesting aspect of our project is that we used the approach of script-based real-time animation particularly for the visualisation of music. Theoretically, any sort of real-time data could be used instead, leading to many distinct usage possibilities.

We modelled the specific phenomena of tree motion and growth. And although we believe that overall we obtained satisfactory results, there is still ample room for improvement: improving performance and allowing for multiple trees and plants with different behaviours; the animation of leaf motion and interaction with wind; the modelling of flowers, different species of trees or even other families of plants. These are all interesting possibilities for future development. Going further with this line of thought could lead to simulation of entire ecosystems.

This takes us to what we consider to be one of the most relevant paths of future work, namely amassing a library of procedural effects and behaviours.

We have explored the movement of trees swaying in the wind. This is a very specific test case and, as such, many more exist which can be modelled by computers. Flock and swarm behaviour, fluids, kinematics, fractals, autonomous agents are some one of the few that come to mind. Biological, geological and chemical phenomena could also be the source of endless inspiration and investigation. Any artistic field would also be a valid driving force, dancing would be the most prominent candidate due to its proximity to music, but the remaining performing and visual arts should not be disregarded.

Cinematography and theatrical concepts and techniques should also be considered, since the end result of our software is effectively comparable to short films and plays. Incorporating only the most basic structural elements from these areas such as the notions of scenes and acts could already yield significant improvements. Camera control and lighting effects were painfully neglected throughout this project and would undoubtedly contribute

to widen the range of expressive possibilities available. We considered real-time performance for the possibility of live accompaniment, but this needs not always be the case, which opens to computationally intensive photorealistic (or non-photorealistic) methods, as usually found in computer-generated imagery.

Concerning music, we have only used the most its most readily accessible information and many more expressive cues could be extracted from both sound and musical structure. Regarding sound, more sophisticated signal analysis algorithms would be useful additions for generating interesting imagery even if the musical representation is not available, which unfortunately will always be the most common case. Algorithms for beat detection and pitch detection would be a valid starting point. Regarding music, the use of information such as key changes and articulation marks are just initial examples. One could analyse the use of scales, intervals or chord and take this into consideration when assigning effects. As such, receiving more musical input, such as the one found in MusicXML is definitely a priority. By using slightly more sophisticated analysis techniques and describing music in more abstract terms, such as motifs or repeating patterns, our system's usability would be greatly improved. We also believe music sheet would be ideal for complementing our system's interface since it is a time proven method for representing and analysing music. The cues for our effects could be precisely represented as annotations in music sheet, which would simultaneously benefit the user experience for musicians. The composer Alexander Scriabin, for instance, already specified colour accompaniments for his musical compositions in sheet music (Berman (1999)).

Lastly, we could take the concept another step forward, allowing for live interaction. A *Disc Jockey (DJ)* has access to hundreds of switches and knobs, which we could accept as input for our system and thus the animation would be effectively produced in real-time, and user and system can cooperate to keep the animation interesting. An orchestra conductor, for instance, could operate a system such as ours while directing an orchestral performance, using gesture recognition and augmented reality techniques. Additionally, stage lighting, special effects, such as lasers and fog machines could also be considered.

We believe to have stumbled upon what seems to be a largely unexplored concept, a union between the seemingly endless processing power of modern technology and human expressiveness. As such, "extraordinary possibilities remain untried, unknown, even barely imaginable".



---

## BIBLIOGRAPHY

---

- Ableton. Music production with live and push — ableton. URL <https://www.ableton.com/en/>.
- Ableton MIDI. Audio to midi tips and tricks. URL <https://help.ableton.com/hc/en-us/articles/209773845-Audio-to-MIDI-Tips-and-Tricks>.
- Adobe Animate CC. Adobe animate cc. URL <http://www.adobe.com/products/animate.html>.
- Bill Alves. Digital harmony of sound and light. *Computer Music Journal*, 29(4):45–54, 2005. doi: 10.1162/014892605775179982.
- Animate CC User Guide. Motion tween animations, Sep 2016. URL <https://helpx.adobe.com/animate/using/about-motion-tween-animations.html>.
- Animusic Company. Software. URL <http://animusic.com/company/software.php>.
- Masaki Aono and Toshiyasu L Kunii. Botanical tree image generation. *IEEE computer graphics and applications*, 4(5):10–34, 1984.
- Arkaos GrandVJ. About grandvj eight channels vj software for real time hd video mixing. URL <http://vj-dj.arkaos.net/grandvj/about>.
- Audacity. URL <http://www.audacityteam.org/>.
- Avmixer pro. Avmixer pro. URL <https://neuromixer.com/products/avmixer-pro>.
- Ramona Behravan. *Automatic Mapping of Emotion in Music to Abstract Visual Arts*. PhD thesis, University College London, 2007. URL <http://www0.cs.ucl.ac.uk/staff/ucacpjb/RamonaBehravanThesis.pdf>.
- Tony Bergstrom, Karrie Karahalios, and John C. Hart. Isochords. *Proceedings of Graphics Interface 2007 on - GI 07*, 2007. doi: 10.1145/1268517.1268565.
- Greta Berman. Synesthesia and the arts. *Leonardo*, 32(1):15–22, 1999.
- Michael Betancourt. Making music with color, Nov 2015. URL <https://www.theatlantic.com/technology/archive/2015/11/color-organs/414460/>.

- Jules Bloomenthal. Modeling the mighty maple. *ACM SIGGRAPH Computer Graphics*, 19(3): 305–311, Jan 1985. doi: 10.1145/325165.325249.
- Frédéric Boudon, Christophe Pradal, Thomas Cokelaer, Przemyslaw Prusinkiewicz, and Christophe Godin. L-py: An l-system simulation framework for modeling plant architecture development based on a dynamic language. *Frontiers in Plant Science*, 3, 2012. doi: 10.3389/fpls.2012.00076.
- Robert Bridson, Jim Houriham, and Marcus Nordenstam. Curl-noise for procedural fluid flow. *ACM Transactions on Graphics (TOG)*, 26(3):46, 2007.
- José Luis Caivano. Color and sound: Physical and psychophysical relations. *Color Research & Application*, 19(2):126–133, 1994.
- Gabriel Dias Cantareira, Luis Gustavo Nonato, and Fernando V. Paulovich. Moshviz: A detail overview approach to visualize music elements. *IEEE Transactions on Multimedia*, 18(11):2238–2246, 2016. doi: 10.1109/tmm.2016.2614226.
- Marc Cardle, Loic Barthe, Stephen Brooks, and Peter Robinson. Music-driven motion editing: Local motion transformations guided by music analysis. In *Eurographics UK Conference, 2002. Proceedings. The 20th*, pages 38–44. IEEE, 2002.
- Xuejin Chen, Boris Neubert, Ying-Qing Xu, Oliver Deussen, and Sing Bing Kang. *Sketch-based tree modeling using markov random field*, volume 27. ACM, 2008.
- Elaine Chew. *Towards a mathematical model of tonality*. PhD thesis, Massachusetts Institute of Technology, 2000.
- Elaine Chew and Alexandre RJ Francois. Interactive multi-scale visualizations of tonal evolution in musa. rt opus 2. *Computers in Entertainment (CIE)*, 3(4):1–16, 2005.
- Reuben Chng. How to use your keyboard or digital piano as midi controller, May 2016. URL <https://www.audiomentor.com/audioproduction/use-keyboard-digital-piano-midi-controller>.
- CityEngine. URL <http://www.esri.com/software/cityengine>.
- Dan Cohen. Computer simulation of biological pattern generation processes. *Nature*, 216(5112):246–248, 1967.
- Nuno N Correia. Avvx: A vector graphics tool for audiovisual performances. *Leonardo Electronic Almanac*, 19(3), 2013.
- Cthugha. URL <http://www.afn.org/~cthugha/>.

- Oliver Deussen and Bernd Lintermann. *Digital design of nature: computer generated plants and organics*. Springer, 2005.
- Hartmann Doreen. Computer demos and the demoscene: Artistic subcultural innovation in realtime. *Proceedings of the 16th International Symposium of Electronic Art ISEA Ruhr 2010*, 2010.
- Jean-Louis Durrieu, Gaël Richard, Bertrand David, and Cédric Févotte. Source/filter model for unsupervised main melody extraction from polyphonic audio signals. *IEEE Transactions on Audio, Speech, and Language Processing*, 18(3):564–575, 2010.
- David S Ebert, Randall M Rohrer, Christopher D Shaw, Pradyut Panda, James M Kukla, and D Aaron Roberts. Procedural shape generation for multi-dimensional data visualization. *Computers & Graphics*, 24(3):375–384, 2000.
- Benj Edwards. The Atari Video Music is a trippy, psychedelic rarity from the 1970s, 2016. URL <http://www.pcworld.com/article/3026252/consumer-electronics/this-old-tech-atari-video-music-is-a-trippy-psychedelic-rarity-from-the-mid-1970s.html>.
- EngArc Inviscid Flow. URL [http://www.engineeringarchives.com/les\\_fm\\_viscousinviscidflow.html](http://www.engineeringarchives.com/les_fm_viscousinviscidflow.html).
- Brian Evans. Foundations of a visual music. *Computer Music Journal*, 29(4):11–24, 2005. doi: 10.1162/014892605775179955.
- Jeremy Falkowski. Inbetweening/cleanup, 1999. URL [http://www.animationartist.com/columns/Inside\\_Animation/Inbetweening\\_Tutorial/inbetweening\\_tutorial.html](http://www.animationartist.com/columns/Inside_Animation/Inbetweening_Tutorial/inbetweening_tutorial.html).
- Finale. Finale music notation software products for music composition. URL <https://www.finalemusic.com/>.
- Callum Galbraith, Peter MacMurchy, and Brian Wyvill. Blobtree trees. In *Computer Graphics International, 2004. Proceedings*, pages 78–85. IEEE, 2004.
- Mitch Gallagher. 7 questions about sample rate, Sep 2015. URL <https://www.sweetwater.com/insync/7-things-about-sample-rate/>.
- Howard Gardner. *Frames of mind: The theory of multiple intelligences*. Basic books, 2011.
- Ryan Geiss. Milkdrop plug-in for winamp, 2013a. URL <http://www.geisswerks.com/milkdrop/>.
- Ryan Geiss, 2013b. URL [http://www.geisswerks.com/about\\_milkdrop.html](http://www.geisswerks.com/about_milkdrop.html).

- Kostas Giannakis. A comparative evaluation of auditory-visual mappings for sound visualisation. *Org. Sound*, 11(3):297–307, December 2006. ISSN 1355-7718. doi: 10.1017/S1355771806001531. URL <http://dx.doi.org/10.1017/S1355771806001531>.
- N. Greene. Voxel space automata: modeling with stochastic growth processes in voxel space. *ACM SIGGRAPH Computer Graphics*, 23(3):175–184, Jan 1989. doi: 10.1145/74334.74351.
- Grower Maya plugin. URL <https://github.com/joesfer/Grower>.
- Ralf Habel, Alexander Kusternig, and Michael Wimmer. Physically guided animation of trees. In *Computer Graphics Forum*, volume 28, pages 523–532. Wiley Online Library, 2009.
- Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. Procedural content generation for games. *ACM Transactions on Multimedia Computing, Communications, and Applications*, 9(1):1–22, Jan 2013. doi: 10.1145/2422956.2422957.
- Hisao Honda. Description of the form of trees by the parameters of the tree-like body: Effects of the branching angle and the branch length on the shape of the tree-like body. *Journal of Theoretical Biology*, 31(2):331–338, 1971. doi: 10.1016/0022-5193(71)90191-3.
- Hans Jenny. *Cymatics: a study of wave phenomena and vibration*. MACROmedia, 2001.
- Randy Jones and Ben Nevile. Creating visual music in jitter: Approaches and techniques. *Computer Music Journal*, 29(4):55–70, 2005. doi: 10.1162/014892605775179900.
- Aristid Lindenmayer. Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of Theoretical Biology*, 18(3):280–299, 1968. doi: 10.1016/0022-5193(68)90079-9.
- B. Lintermann and O. Deussen. Interactive modeling of plants. *IEEE Computer Graphics and Applications*, 19(1):56–65, 1999. doi: 10.1109/38.736469.
- Yotam Livny, Feilong Yan, Matt Olson, Baoquan Chen, Hao Zhang, and Jihad El-Sana. Automatic reconstruction of tree skeletal structures from point clouds. *ACM Transactions on Graphics (TOG)*, 29(6):151, 2010.
- Yotam Livny, Soeren Pirk, Zhanglin Cheng, Feilong Yan, Oliver Deussen, Daniel Cohen-Or, and Baoquan Chen. *Texturelobes for tree modelling*, volume 30. ACM, 2011.
- Steven Longay, Adam Runions, Frédéric Boudon, and Przemyslaw Prusinkiewicz. Treesketch: interactive procedural modeling of trees on a tablet. In *Proceedings of the international symposium on sketch-based interfaces and modeling*, pages 107–120. Eurographics Association, 2012.
- Lua. URL <https://www.lua.org/>.

- Lumigraph. Cvm fischinger's lumigraph. URL <http://www.centerforvisualmusic.org/Fischinger/Lumigraph.htm>.
- Stelios Manousakis. Musical Isystems. *Koninklijk Conservatorium, The Hague (master thesis)*, 2006.
- Bernard Stanford Massey and John Ward-Smith. *Mechanics of fluids*, volume 1. CRC Press, 1998.
- Max/MSP. Max connects objects with virtual patch cords to create interactive sounds, graphics, and custom effects. URL <https://cyclimg74.com/products/max/>.
- Maya. URL <https://www.autodesk.eu/products/maya/overview>.
- MIDI. Midi. URL <https://www.midi.org/>.
- MidiEditor. URL <http://midieditor.sourceforge.net/>.
- J. B. Mitroo, Nancy Herman, and Norman I. Badler. Movies from music. *ACM SIGGRAPH Computer Graphics*, 13(2):218–225, Jan 1979. doi: 10.1145/965103.807447.
- Modul8. Modul8 vj software. URL <http://www.modul8.ch/>.
- W Moritz. The dream of color music, and machines that made it possible. *Animation World Magazine*, 2.1(1), 1997. doi: 10.1017/s135577189900103x.
- Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. In *Acm Transactions On Graphics (Tog)*, volume 25, pages 614–623. ACM, 2006.
- Music Animation Machine. Music Animation Machine. URL <http://www.musanim.com/>.
- Music21. URL <http://web.mit.edu/music21/>.
- Musicxml tutorial. Hello world: A one-bar song with a whole note on middle c in 4/4 time., 2011. URL <http://www.musicxml.com/tutorial/hello-world/>.
- Meinard Müller. *Fundamentals of music processing: audio, analysis, algorithms, applications*. Springer, 2015.
- NASA Navier Stokes Equations. Navier stokes equations. URL <https://www.grc.nasa.gov/www/k-12/airplane/nseqs.html>.
- Boris Neubert, Thomas Franken, and Oliver Deussen. Approximate image-based tree-modeling using particle flows. In *ACM Transactions on Graphics (TOG)*, volume 26, page 88. ACM, 2007.

- Kia Ng, Joanne Armitage, and Alex Mclean. The colour of music: Real-time music visualisation with synaesthetic sound-colour mapping. *Electronic Visualisation and the Arts (EVA 2014)*, Aug 2014. doi: 10.14236/ewic/eva2014.3.
- Makoto Okabe, Shigeru Owada, and Takeo Igarash. Interactive design of botanical trees using freehand sketches and example-based editing. In *Computer Graphics Forum*, volume 24, pages 487–496. Wiley Online Library, 2005.
- Nimish J Oliapuram and Subodh Kumar. Realtime forest animation in wind. In *Proceedings of the Seventh Indian Conference on Computer Vision, Graphics and Image Processing*, pages 197–204. ACM, 2010.
- Peter E. Oppenheimer. Real time design and animation of fractal plants and trees. *ACM SIGGRAPH Computer Graphics*, 20(4):55–64, 1986. doi: 10.1145/15886.15892.
- Shin Ota, Machiko Tamura, Tadahiro Fujimoto, Kazunobu Muraoka, and Norishige Chiba. A hybrid method for realtime animation of trees swaying in wind fields. *The Visual Computer*, 20(10):613–623, 2004.
- Stephen E Palmer, Karen B Schloss, Zoe Xu, and Lilia R Prado-León. Music–color associations are mediated by emotion. *Proceedings of the National Academy of Sciences*, 110(22): 8836–8841, 2013.
- Wojciech Palubicki, Kipp Horel, Steven Longay, Adam Runions, Brendan Lane, Radomir Mech, and Przemyslaw Prusinkiewicz. Selforganizing tree models for image synthesis. *ACM Transactions on Graphics*, 28(3):1, 2009. doi: 10.1145/1531326.1531364.
- Ken Perlin. Real time responsive animation with personality. *IEEE transactions on visualization and Computer Graphics*, 1(1):5–15, 1995.
- Pedro Pestana. Lindenmayer systems and the harmony of fractals. *Chaotic Model. Simul*, 1(1):91–99, 2012.
- Sören Pirk, Till Niese, Oliver Deussen, and Boris Neubert. Capturing and animating the morphogenesis of polygonal tree models. *ACM Transactions on Graphics*, 31(6):1, Jan 2012a. doi: 10.1145/2366145.2366188.
- Sören Pirk, Ondrej Stava, Julian Kratt, Michel Abdul Massih Said, Boris Neubert, Radomír Měch, Bedrich Benes, and Oliver Deussen. Plastic trees. *ACM Transactions on Graphics*, 31(4):1–10, Jan 2012b. doi: 10.1145/2185520.2335401.
- Sören Pirk, Till Niese, Torsten Hädrich, Bedrich Benes, and Oliver Deussen. Windy trees. *ACM Transactions on Graphics*, 33(6):1–11, 2014. doi: 10.1145/2661229.2661252.

- Processing. Processing.org. URL <https://processing.org/>.
- ProjectM. ProjectM. URL <http://projectm.sourceforge.net/>.
- Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer-Verlag, 1990a.
- Przemyslaw Prusinkiewicz and Aristid Lindenmayer. Animation of plant development. *The Virtual Laboratory The Algorithmic Beauty of Plants*, page 133–144, 1990b. doi: 10.1007/978-1-4613-8476-2-6.
- Pure Data. Pd community site. URL <https://puredata.info/>.
- Ed Quigley, Yue Yu, Jingwei Huang, Winnie Lin, and Ronald Fedkiw. Real-time interactive tree animation. *IEEE Transactions on Visualization and Computer Graphics*, 2017.
- Phillippe De Reffye, Claude Edelin, Jean Françon, Marc Jaeger, and Claude Puech. Plant models faithful to botanical structure and development. *Proceedings of the 15th annual conference on Computer graphics and interactive techniques - SIGGRAPH '88*, 1988. doi: 10.1145/54852.378505.
- Resolume. Introducing! resolume v6! URL <https://resolume.com/>.
- Markku Reunanen and Antti Silvast. Demoscene platforms: A case study on the adoption of home computers. In *IFIP Conference on History of Nordic Computing*, pages 289–301. Springer, 2007.
- Markku Reunanen et al. Computer demos—what makes them tick. *Licentiate thesis, Helsinki: Aalto University*, 2010.
- Adam Runions, Brendan Lane, and Przemyslaw Prusinkiewicz. Modeling trees with a space colonization algorithm. In *Proceedings of the Third Eurographics Conference on Natural Phenomena, NPH'07*, pages 63–70, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association. ISBN 978-3-905673-49-4. doi: 10.2312/NPH/NPH07/063-070. URL <http://dx.doi.org/10.2312/NPH/NPH07/063-070>.
- Tatsumi Sakaguchi and Jun Ohya. Modeling and animation of botanical trees for interactive virtual environments. *Proceedings of the ACM symposium on Virtual reality software and technology - VRST '99*, 1999. doi: 10.1145/323663.323685.
- Danielle Sauer and Yee-Hong Yang. Music-driven character animation. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 5(4):27, 2009.
- Adrian Searle. Oskar fischinger: the animation wizard who angered walt disney and the nazis, Jan 2013. URL <https://www.theguardian.com/artanddesign/2013/jan/09/oskar-fischinger-animation-disney-nazis>.



- SFML. Simple and fast multimedia library. URL <https://www.sfml-dev.org/>.
- Shape of Song. Shape of song. URL <http://www.bewitched.com/song.html>.
- Kichiro Shinozaki, Kyoji Yoda, Kazuo Hozumi, and Tatuo Kira. A quantitative analysis of plant form—the pipe model theory: I. basic analyses. *Japanese Journal of ecology*, 14(3): 97–105, 1964.
- Mikio Shinya and Alain Fournier. Stochastic motion—motion under the influence of wind. In *Computer Graphics Forum*, volume 11, pages 119–128. Wiley Online Library, 1992.
- Takaaki Shiratori, Atsushi Nakazawa, and Katsushi Ikeuchi. Dancing-to-music character animation. In *Computer Graphics Forum*, volume 25, pages 449–458. Wiley Online Library, 2006.
- Sibelius. Making music with a little help from your friends. URL <http://www.avid.com/sibelius>.
- smalin. smalin. URL <https://www.youtube.com/channel/UC2zb5cQbLabj3U9l3tke1pg>.
- Jon Snyder and Marti Hearst. Improviz. *CHI '05 extended abstracts on Human factors in computing systems - CHI '05*, 2005. doi: 10.1145/1056808.1057027.
- Solz. URL <https://github.com/ThePhD/solz>.
- Soundspectrum. Soundspectrum news - g-force concert visuals, updates, shows. URL <http://www.soundspectrum.com/news.html>.
- SpeedTree. Speedtree vegetation modeling. URL <https://store.speedtree.com/>.
- Jos Stam. Stable fluids. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 121–128. ACM Press/Addison-Wesley Publishing Co., 1999.
- Robyn Taylor, Daniel Torres, and Pierre Boulanger. Using music to interact with a virtual character. In *Proceedings of the 2005 conference on New interfaces for musical expression*, pages 220–223. National University of Singapore, 2005.
- Daniel Torres and Pierre Boulanger. The animus project: a framework for the creation of interactive creatures in immersed environments. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 91–99. ACM, 2003.
- Christopher Twigg. Catmull-rom splines. *Computer*, 41(6):4–6, 2003.
- Stanislaw Ulam. On some mathematical problems connected with patterns of growth in figures. *Mathematical Problems in the Biological Sciences Proceedings of Symposia in Applied Mathematics*, page 215–224, 1962. doi: 10.1090/psapm/014/9947.

VDMX5. Plays well with others. URL <http://vidvox.net/>.

vvvv. a multipurpose toolkit. URL <https://vvvv.org/>.

M. Wattenberg. Arc diagrams: visualizing structure in strings. *IEEE Symposium on Information Visualization, 2002. INFOVIS 2002.*, 2002. doi: 10.1109/infvis.2002.1173155.

Jason Weber and Joseph Penn. Creation and rendering of realistic trees. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 119–128. ACM, 1995.

Jakub Wejchert and David Haumann. Animation aerodynamics. *SIGGRAPH Comput. Graph.*, 25(4):19–22, July 1991. ISSN 0097-8930. doi: 10.1145/127719.122719. URL <http://doi.acm.org/10.1145/127719.122719>.

John Noble Wilford. Flutes offer clues to stone-age music, Jun 2009. URL <http://www.nytimes.com/2009/06/25/science/25flute.html>.

Adam Matthew Wood-Gaines. *Modelling expressive movement of musicians*. PhD thesis, Applied Sciences: School of Computing Science, 1997.

Xfrog Software. Xfrog - 3D Trees and Plants for CG Artists. URL <http://xfrog.com/>.

Gene Youngblood and Richard Buckminster Fuller. *Expanded cinema*. Dutton New York, 1970.

