Virtual Music:

Developing a System of Music Notation and Performance for Virtual Reality

Nathan Leon Cummins

Doctor of Philosophy

Elder Conservatorium of Music

Faculty of Arts



THE UNIVERSITY of ADELAIDE

July 2021

Contents

1 Introduction **1**

1.1 Introduction . . . 3

1.2 The Approach . . . 9

1.3 Why Virtual Reality? . . . 12

1.3.1 The Ultimate Display . . . 13

1.3.2 Virtual Reality, Music, and the Gamification of Music . . . 16

1.4 Music Notation and Preservation . . . 21

1.4.1 The Fundamental Elements of (Virtual) Music Notation . . . 23

1.4.2 Alternative Notation Systems . . . 25

1.5 Supporting Accessibility . . . 32

1.6 The Development of Software for Music . . . 35

1.7 Methodology . . . 36

1.7.1 Software Development Methodology . . . 36

1.7.2 Waterfall . . . 39

1.7.3 V-Model . . . 43

1.7.4 Iterative and Incremental . . . 45

1.7.5 Evolutionary Prototyping . . . 47

1.7.6 Comparison and Analysis . . . 49

1.7.7 Applying the SDLC Method as a Research Method . . . 51

1.8 Format and Chapter Outline . . . 54

1.8.1 Chapter 2 Overview . . . 55

1.8.2 Chapter 3 Overview . . . 56

1.8.3 Chapter 4 Overview . . . 57
1.8.4 Chapter 5 Overview . . . 57

2 Developing Virtual Music **59**

2.1 Requirements and Specifications . . . 61
2.1.1 Supported Hardware and Software . . . 61
2.2 Initial Design . . . 68
2.3 Pitch, Rhythm, and Dynamics . . . 71
2.3.1 v0.1.0: Étude 1 . . . 71
2.3.2 v0.1.1: Étude 1 . . . 81
2.3.3 v0.1.2: Étude 1 . . . 87
2.3.4 v0.1.3: Étude 2 . . . 92
2.3.5 v0.1.4: Étude 2 . . . 98
2.4 Polyphonic Ability and Timbral Alteration . . . 108
2.4.1 v0.2.0: Étude 3 . . . 108
2.4.2 v0.2.1: Étude 4 . . . 116
2.5 Articulations and Extensibility . . . 124
2.5.1 v0.3.0: Étude 5 and v0.3.1: Étude 6 . . . 124
2.6 Improvements to the Synthesiser . . . 135
2.6.1 v0.4.0: Étude 7 . . . 135
2.6.2 v0.4.1: Étude 8 . . . 145

3 Developer Manual **153**

3.1 Get Started . . . 155
3.1.1 How to use this Get Started Guide . . . 155
3.1.2 Prerequisites . . . 155
3.1.3 Preparation of Required Components . . . 156
3.1.4 Install Virtual Music . . . 156
3.1.5 Next Steps . . . 157
3.2 Shorthand Notation . . . 159

3.2.1 Multi-stave Limitations . . . 160
3.2.2 Settings Blocks . . . 161
3.2.3 Music Symbols . . . 166
3.3 Piece . . . 178
3.3.1 Adding a Piece to a Scene . . . 179
3.3.2 Giving a Piece Notation . . . 180
3.3.3 Setting the Title Information . . . 182
3.3.4 Changing the Piece's Direction . . . 183
3.3.5 Altering the Location of the First Note . . . 184
3.3.6 Altering the Dynamics . . . 185
3.3.7 Altering the Note Lengths . . . 186
3.3.8 Changing the Appearance of Generated Notation . . . 187
3.3.9 Giving All Notes Custom Components . . . 193
3.3.10 Overriding the Location of the Information Slide . . . 194
3.3.11 Prohibiting Automatic Generation . . . 196
3.3.12 Detecting When a Piece Has Finished . . . 196
3.3.13 Destroying a Piece When Finished . . . 197
3.3.14 Triggering Custom Functionality . . . 198
3.3.15 Setting the Piece's Synthesisers . . . 203
3.3.16 Accessing Notes by their Note Flag . . . 204
3.3.17 Accessing Notes or Staves Directly . . . 204
3.4 Note . . . 205
3.4.1 The Note Types . . . 205
3.4.2 Retrieving Information about the Note . . . 206
3.4.3 Implementing Custom Functionality . . . 207
3.4.4 Prohibiting a Note from Being Played . . . 207
3.4.5 Retrieving Notes from a Stave . . . 208
3.5 Stave . . . 208
3.5.1 Retrieving Stave Objects from a Stave . . . 209

3.5.2 Retrieving Information about a Stave . . . 210
3.5.3 Implementing Custom Functionality . . . 211
3.6 VMManager . . . 211
3.6.1 Accessing The Current Piece . . . 212
3.6.2 Accessing the Current Environment . . . 212
3.6.3 Prohibiting Automatic Play . . . 213
3.6.4 Prohibiting Automatic Pause Control . . . 213
3.6.5 Turning on Debug Mode . . . 214
3.6.6 Overriding the Default Synthesisers . . . 215
3.6.7 Set an Overall Mixer . . . 215
3.6.8 Set Global Audio Effects . . . 216
3.7 Instrument . . . 216
3.7.1 The Control Scheme . . . 217
3.7.2 The Performance Modes . . . 222
3.7.3 The Pitch Guide . . . 225
3.7.4 The Instrument Types . . . 226
3.7.5 Retrieving the Current Pitch Degree . . . 227
3.7.6 Retrieving an Instrument . . . 228
3.7.7 Assigning the Instrument to a Mixer . . . 228
3.7.8 Adding an Audio Effect to an Instrument . . . 230
3.7.9 Following the Location of Another Object . . . 230
3.8 Synthesiser . . . 231
3.8.1 The Synthesiser Types . . . 231
3.8.2 Filtering the Output of a Synthesiser . . . 232
3.8.3 Creating a New Synthesiser . . . 232
3.8.4 Retrieving the Current Synthesisers . . . 237
3.9 Environment . . . 238
3.9.1 Retrieving Current Performance Information . . . 238
3.9.2 Retrieve the Note Object Height . . . 239

3.9.3 Changing the Time Scale . . . 239
3.9.4 Converting Height to Pitch Degree or Frequency . . . 240
3.10 VMPerformer . . . 241
3.10.1 Setting the Performance Area . . . 242
3.10.2 Setting the Performer's Dominant Hand . . . 242
3.10.3 Saving and Loading Preferences . . . 243
3.10.4 Retrieving the Location of the Performer . . . 244
3.11 The Events System . . . 244

4 Case Studies **251**

4.1 Case Study 1: Gaming . . . 253
4.1.1 Background . . . 253
4.1.2 Summary of Desired Features . . . 256
4.1.3 Creating the Video Game . . . 257
4.2 Case Study 2: Art and Experiential Music . . . 264
4.2.1 Planning and Creating Key Functionality . . . 268
4.2.2 Implementing Virtual Music . . . 274

5 Conclusion **277**

5.1 Summary . . . 279
5.2 Future Work . . . 290
5.2.1 Therapy and Accessibility . . . 291
5.2.2 Beginning Music Education . . . 294
5.2.3 Ensemble Performance . . . 295
5.2.4 The (Procedural) Generation of Music Compositions . . . 298
5.2.5 Preservation and Widespread Distribution . . . 299
5.3 Reflecting on the SDLC Method as a Methodology . . . 302
5.4 Conclusion . . . 303

6 References **307**

7 Appendices **323**

A The complete commit history of the Virtual Music repository 325

B Issue Log 345

C v0.1.0 349

C.1 Release Notes 349

C.2 Performance Test (Étude 1) 349

D v0.1.1 351

D.1 Release Notes 351

D.2 Performance Test (Étude 1) 352

E v0.1.2 355

E.1 Release Notes 355

E.2 Performance Test (Étude 1) 357

F v0.1.3 361

F.1 Release Notes 361

F.2 Performance Test (Étude 2) 363

G v0.1.4 367

G.1 Release Notes 367

G.2 Performance Test (Étude 2) 369

H v0.2.0 371

H.1 Release Notes 371

H.2 Performance Test (Étude 3) 374

I v0.2.1 377

I.1 Release Notes 377

I.2 Performance Test (Étude 4) 379

J v0.3.0 383

J.1 Release Notes 383

J.2 Performance Test (Étude 5) 386

K v0.3.1 391

K.1 Release Notes 391

K.2 Performance Test (Étude 6) . . . 392
L v0.4.0 . . . 397
L.1 Release Notes . . . 397
L.2 Performance Test (Étude 7) . . . 399
M v0.4.1 . . . 403
M.1 Release Notes . . . 403
M.2 Performance Test (Étude 8) . . . 405
N Preparing Unity for Virtual Reality Development . . . 409
O Preparing the Unity Project for Virtual Music . . . 411
P Piece . . . 413
Q Note . . . 423
Q.1 Native members . . . 423
Q.2 Inherited Members . . . 429
R Stave . . . 433
S Instrument . . . 437
T Environment . . . 445
U The PersonSpawner Script . . . 451
V Video Recordings of All Etudes Through Performance Tests . . . 453
W Creative Works . . . 455
X Demonstrating the Dynamic Audience Noises . . . 457

8 Pieces Composed for Virtual Music 459

A Étude 1 . . . 461
A.1 Virtual Music Shorthand Notation . . . 461
A.2 Equivalent Stave Notation . . . 463
B Étude 2 (Variation 1) . . . 465
B.1 Virtual Music Shorthand Notation . . . 465
B.2 Equivalent Stave Notation . . . 467
C Étude 2 (Variation 2) . . . 469

C.1 Virtual Music Shorthand Notation 469
C.2 Equivalent Stave Notation 471
D Étude 3 473
D.1 Virtual Music Shorthand Notation 473
D.2 Equivalent Stave Notation 475
E Étude 4 477
E.1 Virtual Music Shorthand Notation 477
F Étude 5 479
F.1 Virtual Music Shorthand Notation 479
F.2 Equivalent Stave Notation 481
G Étude 6 483
G.1 Virtual Music Shorthand Notation 483
H Étude 7 485
H.1 Virtual Music Shorthand Notation 485
H.2 Equivalent Stave Notation 487
I Étude 8 491
I.1 Virtual Music Shorthand Notation 491
I.2 Equivalent Stave Notation 493
J Peril (But Not Quite Doom) 497
J.1 Virtual Music Shorthand Notation 497
J.2 Equivalent Stave Notation 505
K Yurndu akanatha 515
K.1 Virtual Music Shorthand Notation 515

List of Figures

1.1 The first system of *Clarinet Concerto in A major* 8

1.2 A violinist interpreting Virtual Music Notation 11

1.3 Three VR applications allowing users to experience new worlds or art forms 13

1.4 A screen-shot from *Curious Alice* 14

1.5 *Beat Saber*: a popular music rhythm video game 19

1.6 A short phrase notated using traditional stave notation 24

1.7 The same short phrase without the fundamental elements 24

1.8 Page 11 of *Threnody to the Victims of Hiroshima* by Krzysztof Penderecki 28

1.9 Alison Knowles performing *Make a Salad* (Walker Arts Center) 30

1.10 Page 23 of *Treatise* by Cornelius Cardew 32

1.11 *Wikipedian Protester* by Rundall Munroe 34

1.12 A typical Waterfall phase structure 40

1.13 The problem with late testing 41

1.14 A typical V-Model phase structure 43

1.15 The Iterative and Incremental life cycle 45

1.16 The Evolutionary Prototyping life cycle 47

1.17 The reflective practice approach 52

2.1 The initial spherical appearance of the ‘stimuli’ 69

2.2 The only exception to the octave principle 73

2.3 The play area 74

2.4 Notes of a stave that approaches from the side remain at a constant distance from the performer 85

2.5 Demonstrating the touch registration problem 87

2.6 Constructing shapes in Unity . . . 104
2.7 The house-like octave up shape . . . 118
2.8 Four crotchets followed by four crotchets with staccatos . . . 128
2.9 Six crotchets, four of which are accented . . . 129
2.10 A screenshot from the *Étude 5* Performance Test . . . 130
2.11 A screenshot from the *Étude 6* Performance Test . . . 131
2.12 A sudden change in amplitude when retrieving a sample as a function of time . . . 142
2.13 The phase of a sine wave . . . 142
2.14 The waveform of phase-tracking oscillator with changing frequencies . . . 144

3.1 The contents of the 'Virtual Music' repository and folder . . . 157
3.2 A short staccato phrase in traditional stave notation . . . 169
3.3 The Information Slide . . . 182
3.4 A pitch guide being displayed for both instruments . . . 226
3.5 Creating a new synthesiser . . . 233
3.6 The oscillator GUI . . . 233
3.7 The schematic of an envelope . . . 234
3.8 The synthesiser GUI . . . 237
3.9 How many events is too many events? . . . 249

4.1 The *Guitar Hero* performance interface . . . 255
4.2 *Guitar Hero* . . . 255
4.3 The finished stage . . . 258
4.4 The audience member prefab . . . 258
4.5 The score screen . . . 262
4.6 The 'Experience Theater' (Heilig 8) . . . 265
4.7 Henry and his birthday cake . . . 266
4.8 The main musical motifs for *Yurndu akanatha* . . . 269
4.9 A skybox with a cosmic texture . . . 270

4.10 Demonstrating the interaction of a directional light with a procedural skybox Shader 271
4.11 Creating the environment 273

5.1 The five 'pitch degrees' of the *Guitar Hero* notation 286
5.2 The Xbox Adaptive Controller 293

List of Tables

1.1 The strengths and weaknesses of Waterfall 42

1.2 The strengths and weaknesses of V-Model 44

1.3 The strengths and weaknesses of Iterative and Incremental 46

1.4 The strengths and weaknesses of Evolutionary Prototyping 48

1.5 Comparing the characteristics of the investigated SDLC methods 49

1.6 The identified common characteristics of the investigated SDLC methods in relation to Virtual Music 50

2.1 A comparison of considered VR systems 62

2.2 A comparison of considered game engines 65

2.3 The pricing models for the considered game engines 65

2.4 The number of Stack Overflow results for each game engine's 'tag' 67

3.1 The dynamic values and their equivalent markings in traditional stave notation 174

4.1 A summary of features common in the *Guitar Hero* series 256

List of Equations

2.1 The formula to convert the linear height to frequency 75

2.2 Calculating the frequency with a root of two 78

2.3 Calculating the note's true frequency including any octave differences . . . 111

2.4 Lowering the first pitch degree by two octaves 111

2.5 Retrieving a sample of a sine wave as a function of time 141

2.6 Converting frequency to angular frequency 143

3.1 Calculating the frequency of a pitch degree 165

3.2 Calculating the number of available pitches 170

3.3 Calculating the frequency of a note with an octave difference 172

3.4 Calculating an even and distributed note height 239

List of Code Samples

2.1 Getting the heights for the possible tones 75

2.2 Implementing the linear height to frequency formula 76

2.3 The greatly simplified GetFrequency method 78

2.4 A first attempt at procedural audio generation 92

2.5 The EventManager class and its container 96

2.6 The default note lengths and dynamics 97

2.7 An excerpt of the ShapeType class showing the VertexPositions Dictionary referencing all corners of a square 101

2.8 Constructing a note using the ShapeType class 103

2.9 Overriding the new Synthesiser class with object-oriented principles 107

2.10 The Regex pattern to generate notes as of v0.2.0 111

2.11 The new Regex pattern for the first and second notes 119

2.12 Methods returning the current musical attributes or progress through a note 122

2.13 An excerpt from the VMManager source code providing lazy loading . . . 123

2.14 An excerpt from the VMManager source code which provides the singleton pattern 124

2.15 The events in the Note class as of v0.3.0 134

2.16 The Apply method of the Envelope class 137

2.17 Methods of the Oscillator class calculating the amplitude and frequency as of v0.4.0 139

2.18 Retrieving a sample of a sine wave at a given time 141

2.19 Increasing the phase of a sine wave according to the requested frequency . 143

2.20 A square wave oscillator that tracks phase 145

3.1 Setting the date of composition 183
3.2 Changing a piece's dynamics 186
3.3 Changing a piece's default rhythmic values 187
3.4 Adding a basic particle system by using CustomNoteComponents 194
3.5 Using the AddCustomNoteComponents method of a different piece 195
3.6 Supplying a method to a piece's finalised event 197
3.7 Setting a note to a random colour upon activation by using PostActivationActions 200
3.8 Utilising the 'bonus' note flag 202
3.9 Accessing notes by their note flag and enumerating through them 204
3.10 Prohibiting a random selection of notes from activation 210
3.11 Calculating the interval between two instruments 227
3.12 Setting the performance area 243
3.13 Subscribing to an event 247
4.1 Using the AddForce method of RigidBody with Note.Activate to make an object jump 260
4.2 The AddScore and FailNote methods 262
4.3 A script that smoothly rotates the sun between two positions 272
4.4 The first section of the Control.cs file 275

List of Piece Excerpts

2.1 A short phrase written with Virtual Music Shorthand Notation 79
3.1 An example of a Settings Block 162
3.2 A single symbol that contains most available segments 167
3.3 A short phrase with staccato indications 169
3.4 A short phrase without staccato indications 169
3.5 A phrase that contains an octave alteration 171
3.6 A phrase creates a chord via an arpeggiation effect 178
3.7 An excerpt of a piece that contains a note with a 'bonus' note flag 202

List of Acronyms

AR Augmented Reality

BPM Beats Per Minute

DMI Digital Musical Instrument

GUI Graphical User Interface

HMD Head-mounted Display

LFO Low Frequency Oscillator

NIME New Interfaces for Musical Expression

SDLC Software Development Life Cycle

VMI Virtual Musical Instrument

VR Virtual Reality

VRMI Virtual Reality Musical Instrument

Abstract

This thesis describes the creation of Virtual Music, a comprehensive system of music performance, notation, and composition using Virtual Reality (VR) technologies. Virtual Music includes Virtual Music Notation and the Virtual Music Instrument, which when combined bring music composition and performance to a new and globally distributed set of composers and performers.

Through systematic experimentation and reflective practice, a set of compositions is created specifically for Virtual Music to support and demonstrate the system's aims of accessibility, ease-of-use, and minimal required training needed to notate and perform. These compositions—and the research itself—use the Iterative and Incremental Software Development Life Cycle (SDLC) as a compositional method to review key features and major changes throughout the development process. The progress and changes between each major milestone or development area are discussed and evaluated through journalling.

Declaration of Originality

I certify that this work contains no material which has been accepted for the award of any other degree or diploma in my name, in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text. In addition, I certify that no part of this work will, in the future, be used in a submission in my name, for any other degree or diploma in any university or other tertiary institution without the prior approval of the University of Adelaide and where applicable, any partner institution responsible for the joint-award of this degree.

I give permission for the digital version of my thesis to be made available on the web, via the University's digital research repository, the Library Search and also through web search engines, unless permission has been granted by the University to restrict access for a period of time.

I acknowledge the support I have received for my research through the provision of an Australian Government Research Training Program Scholarship.

Acknowledgements

There are a number of people who supported me throughout my candidacy that I would like to thank. In particular, I would like to express my gratitude and appreciation to my supervisors, Luke Harrald and Stephen Whittington, whose ongoing feedback and support has been invaluable. I am also grateful for the support of Graeme Koehne, who encouraged the development of Virtual Music when I first discussed it with him many years ago now, guiding me in the right direction. I would like to thank Charles Bodman Rae and Kimi Koaldrake for their support as Postgraduate Coordinators. Their patience, understanding, and encouragement ensured that this project saw its completion. I would also like to thank Tina Morganella for her stellar work editing this thesis.

I am immensely grateful to my family and friends, who are too many to name. In particular, I would like to thank Christopher Larkin for his ideas, continued collaboration, mentoring, and friendship; Jill Jones and Annette Willis for their ongoing encouragement for me and whatever I do, and for their help in retrieving the Adnyamathanha translation; my parents, stepfather, brother, and his wife for their support; my grandparents, for the memories that I will never forget; OOOOORRRR(T), Cameron Salvestro, and John Fazakerley for the valued downtime; and Alex Knoll and Jessica Archbold for their invaluable friendship. I would also like to thank the Woodville Concert Band for the continued distractions despite the looming deadlines.

Most of all, I would like to thank Camille Roulière. I am forever indebted to you for your continued ability to make me smile and believe in myself. And, of course, for your ability to push (rather, pull) me forward when I need it most.

I acknowledge the Kaurna people as the Traditional Custodians of the lands on which the University of Adelaide is located and on which I live; and I acknowledge the Adnyamathanha people as the Traditional Custodians of the lands on which part of my research took place. I recognise and respect their cultural heritage, beliefs and relationship with the land. I acknowledge that they are of continuing importance to the Kaurna and Adnyamathanha people living today. I pay my respect to Kaurna and Adnyamathanha Elders, past, present, and emerging.

1 | Introduction

Virtual Reality, Music, and Alternative Music Notation Systems

1.1 Introduction

My grandfather used to say that he was heralded as one of the best singers in the area; he had sung in front of the Duke of York and even the Prince of Wales. 'I've sung in front of all of them', he would say. He would then pause for a while, have a mouthful of beer and add: 'The finest establishments in Adelaide'.

I am often asked what started my interest in music. 'My grandfather was apparently a great singer', I reply. Truthfully, I still do not know, but I do have a suspicion. I grew up trying to find who I was and, as with many children, I found comfort with a console rather than a football. I preferred a little red plumber and a bear with a bird in his backpack. Each different game was a new world I could immerse myself in.

Still, to this day, every time I turn on *Banjo-Tooie* (Rare), I find myself with a childish grin on my face. Not just for the nostalgic feeling of pulling out my old dusty *Nintendo 64*, but for the auditory experience. There is something satisfying about that percussive *clunk* when inserting the cartridge, and the rhythmic *flick* of the power switch leading to *Banjo-Tooie*'s iconic opening slide whistle glissando. The anticipation of the main theme only grows more and more intense while that synthetic banjo tunes. I am projected back to my childhood: it was not just with the bear and bird that I found comfort. I found comfort with the music.

I realise now that it is video games which made classical music accessible to me. *Super Mario Galaxy* (Nintendo EAD Tokyo) in particular, composed by Koji Kondo and Mahito Yokota, brimmed with classical music references and expertly crafted orchestrations. The driving rhythmic ostinato of *Attack of the Airships* and *Airship Armada* led me to Gustav Holst's *The Planets*, which then led me to Igor Stravinsky.

Just as I had favourite video games, I had favourite pieces of music. While I could share the experience of replaying the same video game levels with others, I could not always share the experience of (re)performing my favourite music. I discovered the wonderful world of classical music through video games, and I wanted to share that world with my family and friends.

Fortunately, I could access the resources I needed to support this newly found passion. My parents sent me to a music school where I could and did improve (depending on who you ask) my performance and composition skills. I loved being able to use music notation to precisely recreate sounds I liked, and quickly learned different methods of notating musical material with words, pictures, and most often with traditional stave notation. I loved and still love being able to share the joy of listening to and reperforming music. But what about those without these resources? What could they do?

This accessibility problem has two components. The ability to recreate a musical performance requires a system of notation[1] which prescribes the performance material. While the purpose, history, and function of these notation systems vary quite significantly, traditional stave notation is used in most classical music performances. However, this stave notation is not universally understood, limiting performances of such musical materials to those able to read it. The second component to this problem is that performance often requires training on the musical instruments prescribed for the composition. A performer—even with an understanding of the language that is musical notation—usually cannot simply be handed a clarinet and have the ability to perform Mozart's *Clarinet Concerto in A major* unless they had training on that instrument and were proficient enough.

Some solutions to these problems have been in the form of alternative music notation

[1]The term notation is difficult to describe without limiting its use and purpose. In this thesis, the term notation generally refers to any method of describing a sonic idea.

systems. These include graphic notation and descriptive forms of communicating performance instructions through written word. An example of this is the piece that I wrote while on a research retreat in 2019 at Oratunga Station, South Australia. *Yurndu akanatha Oratunganga*[2] was inspired by *Teach Yourself to Fly* by Pauline Oliveros, and describes the musical performance of the work itself. It was performed by the musicians, sound artists, writers, and artists, trained and untrained, while at the retreat. They used nothing but voices and whatever lay close by.

> You start to wake peacefully to the sound of a single bird whistling.
> Or was it rain? Wind?
> You are not sure.
> The sun is peaking above the horizon.
>
> Slowly, after a time, there are more sounds.
> The birds are slowly waking, one by one.
> You can almost hear them sing.
> The sun has almost risen.
>
> They are all singing now.
> Harmonious, it is something of beauty.
> A glorious euphony.
> The sun has completely risen.
>
> The sound is dying down now.
> The trees are no longer filled with Babblers or even with Bushlarks.
> A single Bellbird still whistles in the distance.
> You are awake.

Such notation is descriptive: it narrates how a performance of the material *should*

[2]The title translates from the local Adnyamathanha language to *Sunrise at Oratunga* in English.

sound. Descriptive notation is inclusive and can usually be performed by anyone—given enough effort—often on any instrument, regardless of ability. The performance of such a descriptive work, however, often has limited ability for recognition on subsequent performances as their descriptive nature invites improvisation. In such cases, performances of works composed using descriptive forms of notation cannot be 'reperformed', so descriptive notation can be labelled as an improvised form of notation. The piece itself—being the concept of an organised collection of sound—would be performed again, but the precise musical content produced in the previous performance would not. That is often the purpose and beauty of pieces such as these: each performance is unique and is influenced by the performers and even the local surroundings. A performance of *Yurndu akanatha Oratunganga* for example, would have different bird whistles if performed away from Oratunga.

If a 'descriptive' form of music notation roughly describes how a composer would like the material to sound, a 'prescriptive' form would be one that attempts to produce a more precise sounding result in subsequent performances. Besides differences in interpretation and even with virtuosity and player skill, pieces composed with such a prescriptive notation have the ability for subsequent aurally recognisable 'reperformances'. That is, prescriptive forms of notation are non-improvised: the composer retains control over the produced sound to the degree they wish[3].

I have chosen to use the terms reperformance, reperformed, and reperform to emphasise the ability of notation to enable a performance of a piece that is recognisable with another performance of the same piece. Both performances retain the same basic musical characteristics that the composer intended. Traditional stave notation is an example of this type of notation. Figure 1.1 shows the first system of *Clarinet Concerto in A major*. It includes specific values for most elements required to produce an ordered

[3]This also includes pieces that do not specify instrumentation or contain allocated sections for the purposes of improvisation.

collection of musical sounds, including time, dynamics, pitch, timbre, and articulation. The specific indication of these characteristics becomes the criteria for the recognition of a reperformance of the piece, and form the identity of the work. If a group of musicians were to read and interpret the notation with the same intent—that is, the notation was interpreted in the manner that the composer intended and specified—their performance would be aurally recognisable as a reperformance of the piece. Differences between reperformances may be analysed, but they will likely be due to interpretation or situation. A composer may choose not to indicate certain characteristics, such as timbral quality by not specifying instrumentation, or time by the omission of a tempo marking. A piece that does not specify some characteristics retains its identity as while other characteristics have been improvised, those that were specified have remained the same. The omission of too many characteristics, however, increases the likelihood of the piece not retaining the ability of recognition, as there are less criteria which form the identity of the work. As Gimbel discusses, all *Exercises* by Christian Wolff forgo timbral specification by not listing any instrumentation. They can be performed 'by any number of unspecified instruments' (190), yet the identity of the works remain strong, and reperformances are recognisable. The term reperformance is distinguished from reinterpretation as the term reinterpretation implies a new or different approach to specified characteristics of the original work. For example, the alteration of timbral quality by adjusting the instrumentation, or a change of rhythmic character by adjusting the melody and harmony to fit a new time signature, would be a reinterpretation. As seen with Wolff's *Exercises*, the performance of a work that does not specify instrumentation would be a reperformance, as long as the specified characteristics have remained the same. Many of the 'time-brackets' of Cage's *Number Pieces* require interpretation, but the characteristics that are specified are not *reinterpreted* (Popoff 67). In this way, modern stave notation is able to be reperformed, and has the flexibility to blend descriptiveness and prescriptiveness.

The development of Digital Musical Instruments (DMIs) attempts to address the
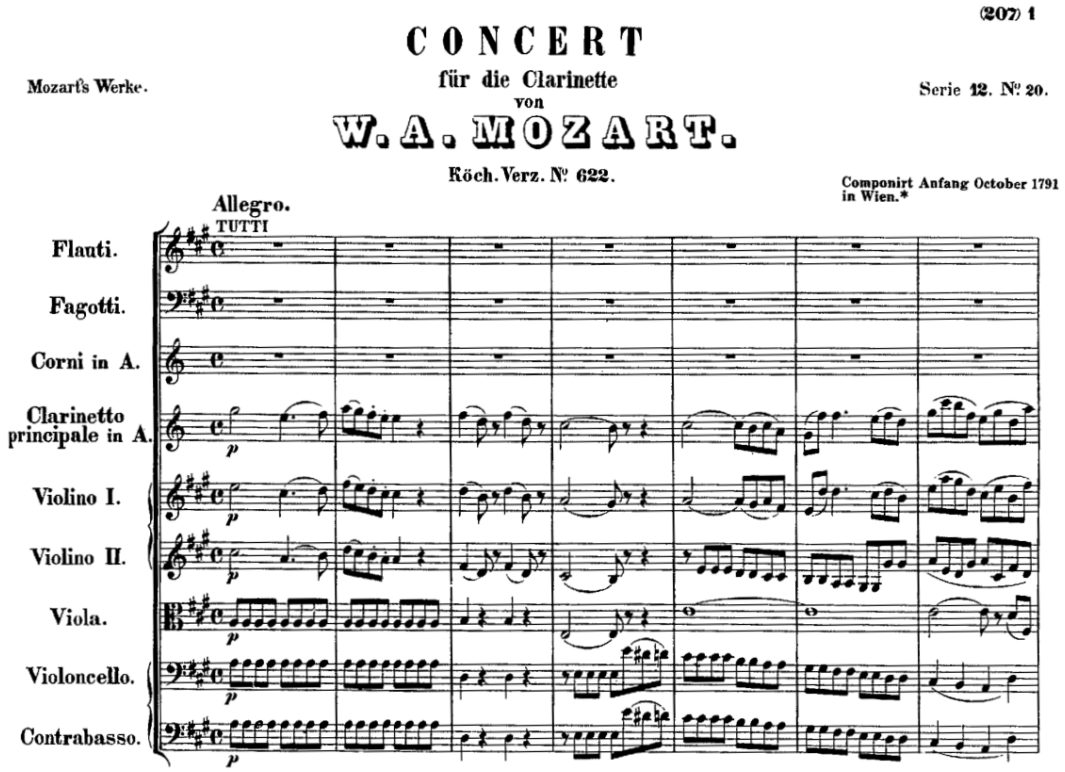


Figure 1.1: The first system of *Clarinet Concerto in A major*

second component of the accessibility problem. The New Interfaces for Musical Expression (NIME) community in particular has made considerable contributions in this area. For example, the 'hackable' *D-Box* was 'designed to appear simple and limited at first but allow wide-ranging creative exploration' (McPherson and Zappi 163). Such instruments allow new and innovative ways to approach music making, often through simple control mechanisms. The *reacTable*, for example, is a tabletop instrument 'designed for installations and casual users as well as for professionals in concert'. It presents an intuitive platform in which 'several musicians can share the control of the instrument by caressing, rotating, and moving physical artifacts' (Jordà et al. 4).

These instruments, however, are often criticised for being limited in expressiveness. They limit expressive performance actions, which, as Kirke and Miranda describe, are 'the strategies and changes which are not marked in a score but which performers apply to the music' (2), such as rubato or expressive intonation. Therefore, they are often not suitable for the evolution of musical personality, character, and 'the continued development of

musical expressiv[eness]' (Wessel and Wright 12). Additionally, they often do not include methods for the notation and persistence of settings or for the notation of performance instructions. This leads to an inconsistent experience when using multiple DMIs, and limits the ability of a performer to increase their skill with the instruments. As Wessel and Wright describe it: 'many of the simple-to-use computer interfaces proposed for musical control seem, after even a brief period of use, to have a toy-like character and do not invite continued musical evolution' (Wessel and Wright 11).

1.2 The Approach

This twofold accessibility problem needs to be addressed in two parts: first, through a method to communicate musical material so that it can be reperformed and recreated, but is intuitive and accessible even without extensive training; and second, via an instrument that is also intuitive and accessible without training. Like the notation, this instrument must allow 'the continued development of performance skill and personal style' by allowing expressive, interpretive, and creative freedom (Wessel and Wright 12).

With this research, I have addressed this problem by developing an accessible form of music that I call Virtual Music[4]: an innovative, and comprehensive system of music performance, notation, and composition for Virtual Reality (VR). Virtual Music uses the accessible and intuitive nature of VR technologies to allow the recreatable and aurally recognisable reperformance of music without training or

[4]I use this term to describe the software package described in this thesis, which combines Virtual Reality (VR) technologies with music creation. The term is also used by other authors including composer and scholar David Cope, who uses it to describe 'a broad category of machine-created composition which attempts to replicate the style but not the actual notes of existing music' (3). 'Internet composer' William Duckworth also uses the term as the title of his 2005 book, in which he discusses a history of interactive music and the opportunities that the internet provides music composition. Recently, musicologist Shara Rambarran also used the term to title their book discussing 'virtuality' and the interactive relationship of digital virtual music. My definition does not attempt to replace these existing definitions.

knowledge of traditional music theory and performance practices. It offers incentive for trained and untrained performers to develop their musicality through varied difficultly settings called Performance Modes, which allow a performer to hone their performance skills and personal styles. As a performer progresses through these Performance Modes, their ability to musically express themselves will improve, as will the quality of their performance. Virtual Music has three main aims. It should:

- be accessible
- allow musical expressiveness
- allow reperformance.

Virtual Music consists of two main components: the notation, called Virtual Music Notation, and the instrument, called the Virtual Music Instrument. Virtual Music Notation is comprised of shapes or 'notes' that approach the performer at the rate or 'tempo' defined by the composer. These notes are sized according to their musical attributes: their height corresponds to their pitch, their width to their dynamic, and their length to their rhythmic duration. In its default Performance Mode, the Virtual Music Instrument automatically translates the musical attributes of the notes into the production of sound. The performer must only touch the note with their hand or controllers in the virtual environment. As the performer becomes familiar with the Virtual Music Instrument and with Virtual Music Notation, they may progress through the Performance Modes to gain better control over the way that the Virtual Music Instrument produces the sound and the way that it translates each note's musical attributes. The performer may also use the 'Advanced' Performance Mode to produce tones manually. If the performer presses either of the synthesiser production buttons, a tone will be produced. The height of the Virtual Music Instrument corresponds to the pitch or frequency of the produced

tone and the rotation of the Virtual Music Instrument corresponds to the dynamic or amplitude.

Combined, these components form a prescriptive form of music notation and performance by ensuring that the performance of a composer's notated work is precisely recreatable. The Virtual Music Instrument precisely translates the prescriptive musical instructions of Virtual Music Notation into performance. However, their combination is not exclusive: the notation may be interpreted by a performer using a conventional instrument, such as a violin as seen in Figure 1.2, making Virtual Music Notation also a form of graphic notation.



Figure 1.2: A violinist interpreting Virtual Music Notation

Similarly, the Virtual Music Instrument may be used in combination with other

instruments and with traditional stave notation by a performer with sufficient skill. In order to ensure that these components met the aims of Virtual Music, the development process responded to four questions:

1. Can Virtual Music enable new performers to reperform and recreate music to some degree of recognition with minimal or no training, while allowing the development of their skill and musical expressiveness?
2. Can the Fundamental Elements of (Virtual) Music Notation[5] and other notation and performance conventions be adapted to VR?
3. What new opportunities and limitations does VR bring to music notation, composition, and performance?
4. Can Virtual Music Notation appropriately convey a composer's intention, while still allowing the performer musical expressiveness?

The design and functionality of these components were developed and extensively tested with these questions in mind. This development and testing process was also firmly rooted in accepted software development practices (including use of version control with *Git*, and issue tracking and release notes through *GitHub*), and as such, was recorded and is presented here through journalling documentation.

1.3 Why Virtual Reality?

Besides my keen interest in VR, having been a follower and consumer of it since the early days of the *Rift* by Oculus—one of the first commercially available Head-mounted Displays

[5]See subsection 1.4.1.

(HMDs)—I chose to use VR for this project for a variety of reasons. In particular, because it is, as computer scientist Ivan E. Sutherland describes, the 'ultimate display'. The ability of VR to immerse the user in a virtual environment with intuitive control systems gives VR inherent accessibility. It gives users access to exciting new worlds and unlimited ability to imagine and create. In ten minutes, a user can traverse the depths of the ocean in *Ocean Rift* (Llyr ap Cenydd), travel through space in *Elite Dangerous* (Frontier Developments), and finish by expressing their creativity with a 3D painting in *Tilt Brush* (Google). Such accessibility makes VR an excellent candidate for an accessible form of music creation. While *Tilt Brush*, for example, gives users access to methods of creating visual art, few applications exist that target music performance. Even fewer have methods that allow a performer unfamiliar with the act of music creation to recreate and understand their performances in the form of some sort of notation. Music applications can use the intuitive control systems and high availability of VR to bring music reperformance to a new and globally distributed audience via novel methods of playing and notating. Combined with these intuitive control systems, I hope to make Virtual Music the 'ultimate display' of music notation.



(a) *Ocean Rift* (b) *Elite Dangerous* (c) *Tilt Brush*

Figure 1.3: Three VR applications allowing users to experience new worlds or art forms

1.3.1 The Ultimate Display

> The ultimate display would, of course, be a room within which the computer can control the existence of matter. A chair displayed in such a room would be

> good enough to sit in. Handcuffs displayed in such a room would be confining, and a bullet displayed in such a room would be fatal. With appropriate programming such a display could literally be the Wonderland into which Alice walked. (Sutherland 508)

In 2020, only 55 years after Ivan E. Sutherland first described the 'ultimate display' in 1965, VR quite aptly fulfilled some of his predictions with *Curious Alice*. This VR experience, created by the *Victoria and Albert Museum* and *HTC Vive Arts* with original artwork by Kristjana S. Williams, can be seen in Figure 1.4 (McGreevy).



Figure 1.4: A screen-shot from *Curious Alice*

The field of VR has experienced significant growth in recent years. This can be largely attributed to the advancement in mobile display technologies and Palmer Luckey's *Oculus Rift* HMD prototype. This prototype gained significant attention after being demonstrated by John Carmack at *QuakeCon* in 2010, who received $2.4 million[6] of funding through the *Kickstarter* platform only two years later (Luckey). In 2014, social

[6]All currencies listed are the United States Dollar.

media giant *Facebook* purchased Luckey's company *Oculus VR* for $2.3 billion (Facebook). A highly competitive and rapidly advancing industry quickly developed, culminating in the release of the consumer version of *Oculus Rift* and the *HTC Vive* in 2016. Grand View Research state that the VR market was valued at $15.81 billion in 2020 and estimate a compound annual growth rate of 18 per cent from 2021 to 2028.

While financial evaluations of the industry are increasing, the cost to the consumer is decreasing and VR is becoming more accessible to consumers. The original *Rift* by Oculus cost $599 for the HMD only. The optional *Oculus Touch* hand-held controllers were available separately at a cost of $199. This version of the *Rift* also required additional sensors in order to support room-scale experiences at the cost of $79 each. It also required a computer that could support running the device. Such a computer could cost anywhere from a few hundred to a few thousand dollars, depending on the desired experience. By contrast, Oculus's latest release, *Quest 2*, now costs only $299 for the base model, which includes the hand-held controllers, and does not require any additional sensors. The *Quest 2* is also portable, which removes the need for a connected computer.

The accessible nature of VR makes it a good solution for many research projects or products that are designed with a global audience in mind. This is because modern interfaces for interaction in VR are becoming progressively *human*, contributing to 'virtuality' as Popper describes: the 'humanizing of technology through interactivity and neocommunicability as well as sensory immersion and multisensoriality' (16). The *Quest* and *Quest 2*, for example, may be interfaced by using the hands only, without any controllers. Their sensors track the user's movement, immersing them in an environment with which they can interact as expected. This ability to immerse the user in a virtual environment may trick the user into believing what they are seeing is true. As sound and media artist J. Camillo Sánchez Carranco states: '[o]nce this immersion is achieved, the user's perception is tricked into believing that what is happening in a virtual environment, it is actually happening in real life' (Carranco 8). In reference to 'virtual

art', Grau describes this immersion as 'undoubtedly key to any understanding of the development of the media' (13).

VR technologies are increasingly being incorporated into education and training settings with positive results. In a study of surgical residents, Neal E. Seymour et al. found that the dissection of a gallbladder was completed 29 per cent faster by VR-trained residents (458). Non-VR-trained residents were nine times more likely to fail to make progress and five times more likely to injure the gallbladder or surrounding tissue. Similarly, research into the use of VR in health and therapy has yielded positive results. Maryrose Gerardi et al. concluded that VR Exposure Therapy reduced the self-reported symptoms of post traumatic stress disorder sufferers by 56 per cent (212). Considering this research, it is clear that the immersive and intuitive nature of VR can be used as an effective tool in the health industry in particular. The wide variety of research involving VR indicates the field of medicine has acknowledged that the applications and benefits of VR are seemingly limitless. Research into the application of VR into other fields, however, is still in its infancy.

1.3.2 Virtual Reality, Music, and the Gamification of Music

While there is extensive research being conducted into the use of VR in education and medicine in particular, research into the use of VR in music creation is still in the early stages. Researchers have been experimenting with DMIs for quite some time within the NIME community. The advancement of VR technologies in recent years, in particular of hand-held controllers, has allowed for some interesting developments with Virtual Reality Musical Instruments (VRMIs). These VRMIs are distinguished from DMIs by the inclusion of a component involving VR, usually with an HMD, or other forms of

immersive visualisation systems surrounding a listener, such as a *CAVE* (Cruz-Neira et al. 135). Yet, VR is still underrepresented in comparison to DMIs. As Carranco writes:

> However, despite all the categories covered by NIME, only nine papers related to VR can be found in the NIME proceedings archives. The limited amount of VR NIME articles implies the poor adaption of a technology that's features can be used in benefit of music interaction. (Carranco 9)

This number has barely increased since, with a search of 'virtual reality' in the NIME proceedings archives resulting in eight papers, and 'vr' resulting in seven papers. There still seems to be a reluctance to incorporate VR into the creation of music. Part of this reluctance to embrace the technology may be attributed to the fact that VR technologies 'do not favour collective VR experiences' (Carranco 8). Because of this, most research projects involving VR and music focus on the individual experience. Stefania Serafin et al., for example, 'present a field overview of VRMIs from the viewpoint of a performer' (26). They developed a set of nine design principles for interactive music systems in VR that largely focus on the technical aspects of creating a single-performer VRMI, including considerations for button mappings and the reduction of latency. Serafin et al. also distinguish between VRMIs and Virtual Musical Instruments (VMIs), describing VRMIs as instruments that 'also include a simulated visual component delivered using either an HMD or other forms of immersive visualization systems such as a CAVE', and VMIs as 'software simulations or extensions of existing instruments' (22). Often, such instruments focus on sonic emulation by physical modelling synthesis, as Vesa Välimäki and Tapio Takala describe. Such physical modelling can be seen in the work of spatial audio researcher Lui Albæk Thomsen, which documents 'several implementations of real-time sound synthesis related to interaction with and realism of musical instruments' (1). It can also be seen in the work of interaction researcher Andrew Johnston, who developed

three 'interactive virtual musical instruments intended to augment acoustic instruments in live performance' (xiv): *Partial Reflections I*, *Partial Reflections II*, and *Spheres of Influence* (76).

However, not all research falls into these two streams. Some research combines both VRMI with VMI. 'SpectraScore VR' by Benedict Carey, for example places itself somewhere in between both of these, as a real-time collaborative composition and performance tool involving VR. Performers or 'real-time composers' are placed among shapes or 'objects' generated by harmonic analysis of an input audio stream which form the notation. These real-time composers then improvise music from this notation with theremin-like instruments, which in turn is re-fed back into the score generation system. Music scholars Martin Naef and Daniel Collicott also present an improvised musical performance tool that makes use of VR and tracked gloves to 'manipulate audio objects and stereoscopic project to display the virtual 3D sound stage' (57). Both of these systems, however, present no method for the notation and reperformance of compositions. Each performance, therefore, is improvised and not recreatable.

While music and VR are still underrepresented in music research, the combination has gained widespread traction in the gaming space. Such video games—called music rhythm video games—are often designed so that players can interact with existing music. Performances using these video games are recreatable in the sense that a player may choose to play the same level again and repeat the same movements or combination of buttons as indicated. The form of notation for these movements, however, often does not reflect the type of tone that will be produced. In some cases, the player's actions and button presses do not produce tones at all. Nonetheless, music rhythm video games are becoming increasingly popular. *Beat Saber*, a music rhythm video game by Beat Games is a good example of this phenomenon. A player of *Beat Saber* must simply hit boxes that approach them with the controller that corresponds to the colour of the box. Unlike with interactive music performance installations such as the *Very Nervous System*, which uses

'video cameras, image processors, computers, synthesizers and a sound system to create a space in which the movements of one's body create sound and/or music' (Rokeby), the precise actions of the player have no impact on the production of the music. Rather, the music informs the player what actions they must perform. It has achieved critical acclaim and won multiple awards such as the best VR/Augmented Reality (AR) game in the 2019 *Golden Joystick Awards* (GamesRadar+). Statistics site TwitchTracker reports that a total of 150,962 concurrent viewers were watching streamers play *Beat Saber* on 9 January 2021 alone (TwitchTracker). This is an audience number larger than many live concerts today, especially in light of the ongoing coronavirus restrictions and confinements worldwide.

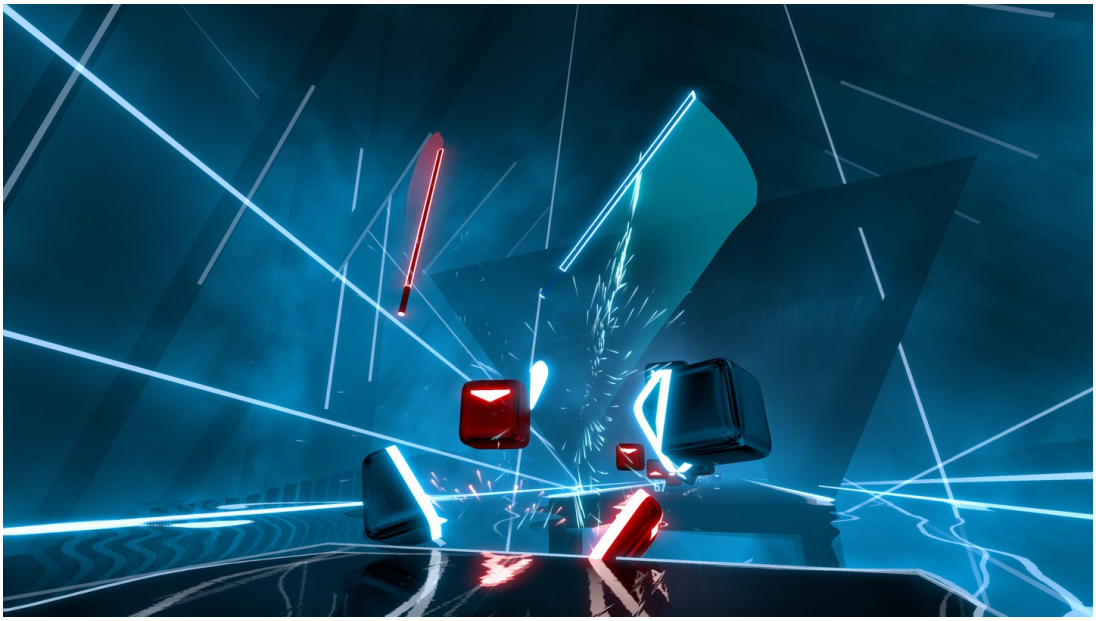


Figure 1.5: *Beat Saber*: a popular music rhythm video game

The music rhythm games space is not by any means a new one. It has enjoyed a large audience and a large catalogue of games, including the famous *Guitar Hero* (Harmonix). VR, however, offers a new immersive component to the experience that was not previously possible. VR has created new opportunities for the act of music creation to reach new audiences that were previously left to science fiction, or predictions such as those by

Sutherland. Discussing *Audioshield*, a VR music rhythm game by Dylan Fitterer predating *Beat Saber*, game designer Jeff Grubb wrote:

> [H]is rhythm game is a whole-body experience. Like *Guitar Hero* or *AurdioSurf*, notes come flying toward you, and you need to stay on beat with the music to successfully hit them. Unlike those games, the immersion of virtual reality makes it feel like the song is really overwhelming you physically. Tech adviser Digi-Capital is predicting that VR will generate $30 billion in spending by 2020, and it's software like *Audioshield*—which I think everyone could pick up and enjoy—that could prove that estimate accurate. (Grubb)

While the use of music in video games is the focus of more and more research projects today, research into the gamification of music itself would benefit from receiving closer scrutiny. It is a tool that could better connect the student to the teacher and provide motivation to learn in a manner that suits the student:

> I have found the phenomenology of flow[7] to be a useful lens for looking at ways to motivate students in their music learning. While digital technology and games will never replace the teacher-student relationship, this technology might help to connect student and teacher outside of lesson times and to provide feedback to students in their individual practice. Music teachers have a role in educating their students about how to use digital technology effectively to enhance their practice experience. (Wagner 121)

While gamification might not be a replacement for music learning, it is clear that

[7]The term 'flow' refers to a feeling of immersion, creativity, and involvement as discussed by Mihaly Csikszentmihalyi.

gamification can be used as a pathway to music creation by other means, or at least as a method to make music reperformance accessible to a larger audience. This is especially evident when considering the popularity of music rhythm video games. The recent resurgence in popularity of music rhythm games with *Beat Saber* also makes it evident that VR can and should be used as an effective tool to bring music reperformance to a wider and musically untrained audience.

1.4 Music Notation and Preservation

> How do we write music? What is musical notation? Why do we write music? These are simple questions with complex answers that relate to culture, media, syntax and semantics, communication, practice, education, and performer skills. (Magnusson 71)

Music was performed long before it could be written down. In many cultures, music is still performed and shared 'by ear': the communication of the music and performance instructions is done without any sort of written notation. As Magnusson describes, 'the notation of [early] musical works was not given much importance as music was primarily transmitted orally and by imitation' (72). Such music is difficult to preserve as subsequent performances will change ever-so subtly. Small elements, including the rhythm, dynamics, or pitch, may change over time. While the recording technology of today can assist with preserving this music, it relies on the ability of the technology to capture every component of the performance. It also relies on the performance being a true rendition of the original piece, without alteration. There is, however, some sort of stability in the transmission of this music. For example, as journalist Frances Mulraney writes, traditional Irish music has a longstanding tradition of oral exchange:

> Irish traditional music began as an oral tradition, passed on from generation to generation by listening, learning by ear, and without formally writing the tunes on paper. This is a practice that is still encouraged today and students of traditional music are encouraged to pick up tunes they hear from others or to learn as they listen. Many formal classes will provide music notes for students and books do print tunes on a traditional music stave, however. (Mulraney)

As Mulraney indicates, music written for the purposes of preservation and communication is most often done using traditional stave notation (Schmidt-Jones 3). By using a staff and other symbols accompanying it, a composer may precisely indicate how they wish a performance of the notation should sound to the degree that suits their intention. As Goehr discusses in *The Imaginary Museum of Musical Works*, the available level of precision in modern stave notation today 'largely resulted from a novel desire to preserve the identity of music exactly as determined by the composer' (29). Reperformances of works using stave notation produce similar results throughout the world, even after centuries:

> With ever more precise notation it was possible to produce numerous performances of the same sound and instrumental specifications; it became increasingly easy to hear and recognise exactly the same piece of music in different performances. (Goehr 29)

This can be seen with performances of the previously mentioned *Clarinet Concerto in A major*; a performance in Europe would sound similar to a performance in Australia, and those performances would sound relatively similar to how Mozart intended. This is largely due to the ability of modern stave notation to precisely indicate—or not indicate—performance elements that the composer considers necessary. These include elements

which I will call the 'Fundamental Elements of (Virtual) Music Notation'[8]: pitch (how high or low the note should sound), time (how long or short the note should be, additionally how fast or slow successive notes should occur), and dynamics (how quiet or loud the note should be). Other elements may also be notated, including articulation, ornamentation, and even timbre through instrumentation, style indications, playing techniques, or emotive instruction. Likewise, these elements can be omitted if the composer chooses. With some explanation, additional symbols or text can be added to more precisely notate a sound or performance action. This ability to precisely notate performance instructions allows music notated in stave notation to be preserved. Subsequently, its preservation allows reperformance.

1.4.1 The Fundamental Elements of (Virtual) Music Notation

I define the elements of pitch, time, and dynamics as *fundamental* elements for Virtual Music Notation because the definition of these elements at minimum give a composer using Virtual Music Notation the best ability convey their intention in most scenarios. A composer may choose not to supply a definition for these values, but this lack of definition would be intentional in order to invite improvisation. A composer may also choose to notate additional performance instructions through articulations, ornaments, and written playing techniques. While such performance instructions may be added by the composer or performer to add interest and character, they are often 'not part of the main melodic line' (Schmidt-Jones 73), so in most cases can considered 'subordinate' (Schoenberg 169), yet desired functionality. Therefore, in the early design stages of Virtual Music Notation as described in this thesis, limiting the

[8]This term refers to but does not attempt to redefine Schoenberg's *Fundamentals of musical composition*. In particular, it is referring to the discussion of the production of a theme for variation, in which Schoenberg discusses the need to 'recognize the essentials of the theme' (169).

development scope to these three elements best supports a focused development that covers most use cases. The ability to extend the functionality of VIRTUAL MUSIC gives the composer the flexibility to indicate other instructions more precisely and allowing increased expressiveness in the packaged notation system would then be part of future work.

In traditional stave notation, the combination of these fundamental elements can very precisely indicate the composer's original intent to the degree they wish. The omission of these, however, can quickly make these phrases unrecognisable. Take for example the short phrase seen in Figure 1.6, notated using traditional stave notation. It contains most of the standard elements of traditional stave notation, including optional articulation and ornaments.
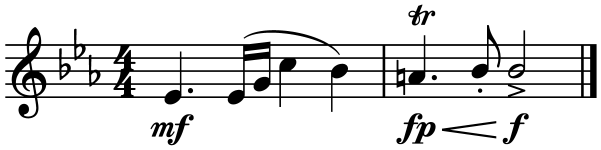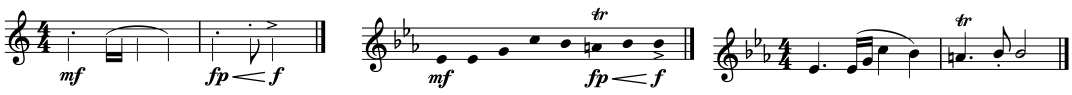


Figure 1.6: A short phrase notated using traditional stave notation

This phrase can also be seen in Figure 1.7. This figure, however, shows the phrase first without any indication of pitch, second without time, and third without dynamics.



(a) without pitch (b) without time (c) without dynamics

Figure 1.7: The same short phrase without the fundamental elements

Arguably, the indication of dynamics would be the least important of these definitions: the duration and pitch of the notes have greater impact on the ability of a person to recognise a phrase. However, without dynamics a piece may become dull. Take, for

example, the dynamics in *The Augurs of Spring*, the first dance of the ballet *The Rite of Spring*, composed by Igor Stravinsky. A performance of such a phrase at a uniform dynamic level quickly becomes uninteresting and loses the original intent. Additionally, as Holmes discusses in regards to composers Anton Webern, Pierre Boulez, and Karlheinz Stockhausen seeking the total 'serialization' of all tonal elements, those composers applied strict rules 'not only to pitches, timbres, and rhythms, but to dynamics, densities, and amplitude as well' (35). For these reasons, I consider the ability to indicate dynamics as a fundamental element of Virtual Music Notation.

1.4.2 Alternative Notation Systems

I have chosen to use stave notation as a basis of comparison due to its frequent and widespread use in many different cultures and performance practices. Importantly, I also refer to it frequently due to it forming the basis of my understanding of music notation. Music notation is of course not limited to stave notation and there are an infinite number of ways that sonic ideas can be described. Stave notation itself has evolved over time and will continue to evolve to suit new performance conventions. Likewise, stave notation can be morphed, or entirely new notation systems created for varied or specific use cases, such as guitar tablature, which is used to prescribe instrument fingering rather than musical pitches. As a prescriptive form of notation, Virtual Music Notation aims to support music preservation and must have the ability to clearly indicate at minimum the Fundamental Elements of (Virtual) Music Notation. However, some notation systems, such as graphic, animated, or text-based notations, offer different levels of preservation ability while often ignoring one or many of these elements. Such notation systems vary in purpose. They may simply be used for the easier transmission of an abstract concept, or for their ability to specify performance instructions in a way not possible or difficult with other notation systems.

Alternative notation systems may be used as a tool to assist performers without formal music training. The Music Notation Project lists seventeen criteria desirable for alternative systems of notation with a goal to 'make reading, writing, and playing music more enjoyable and easier to learn'. The criteria listed, however, largely assume an octave of twelve pitches. They derive from the '13 Commandments for a Music Notation' presented by the Music Notation Modernization Association, whose first commandment states '[t]here shall be twelve invariable note symbols in the octave' (Eggleston 661). Systems epitomising the listed criteria are usually created to act as a replacement for traditional stave notation. Additionally, although some compliant notation systems lessen the level of training required to correctly interpret the notation, an understanding of their symbols and indications is still necessary.

Other alternative systems of notation often exist because the composer wishes to indicate something that they could not with traditional stave notation. This may be because traditional stave notation is, in many ways, limited, and often not suitable for aleatoric textures. As Australian composer, musician, and academic Cat Hope describes:

> The creation of a score is an important aspect for the transference of ideas from composer to performer. It provides the template from which a performance can begin. Traditionally, music notation provides rather strict information for the performance realisation of rhythm, pitch, structure, dynamics, and instrumentation. However, not all composers—or performers—wish to have all of these elements so strictly described. (Hope, 'Wording New Paths' 36)

Composers wishing to better notate their ideas often use graphics, text, or a combination of both with traditional stave notation. Pieces notated using these elements can be difficult to classify as descriptive or prescriptive; pieces involving text may perfectly prescribe details of the performance, but their interpretation is left to the

performer so could be considered descriptive. Similarly, some graphic notation uses large portions of musical material notated with prescriptive traditional stave notation, but some elements of their indications are also subject to interpretation. This can be seen with many of the musical scores in *Notations* by John Cage, or with *Threnody to the Victims of Hiroshima* by Krzysztof Penderecki, seen in Figure 1.8. These scores are often precisely notated, but their performance is left to interpretation. The ability for these scores to be reperformed is mixed. *Threnody to the Victims of Hiroshima*, for example, is largely comprised of aleatoric textures, but is aurally recognisable on subsequent performances. This is largely because the main 'landmark' elements of the piece are precisely indicated with time, dynamics, and pitch. These landmark elements, however, would be difficult to recognise if removed and played separately from the work.

In 'Wording New Paths', Hope discusses the notation of three percussion works, describing them as epitomising the effectiveness of combining common notation practices with graphics and text, saying 'certain musical possibilities may only be expressed with words, supported by pictures and signs' (46). Animated notation is also discussed in 'Electronic Scores for Music', in which Hope describes it as 'a predominantly graphic notation that engages the dynamic characteristics of screen media' (21). She argues that music featuring electronics and aleatoric elements in particular benefit from animated notation. Understanding the notation of *Threnody to the Victims of Hiroshima*, for example, is made much simpler with the addition of animated computer graphics to create a 'digital score', as Vear describes. A video uploaded to *YouTube* titled *Penderecki - Threnody (Animated Score)* by user gerubach, for example, is an excellent example. The video places animated red bars across the notation and moves the view of the notation so that the viewer can more easily understand the notation in relation to the recording. Vear claims that musicians that use digital scores are 'liberated from any constraints that may be imposed by adhering to traditional methods of notation or any visual symbolism' (Chapter 2). Vear also mentions the importance of the the ability for digital scores to be reperformable and

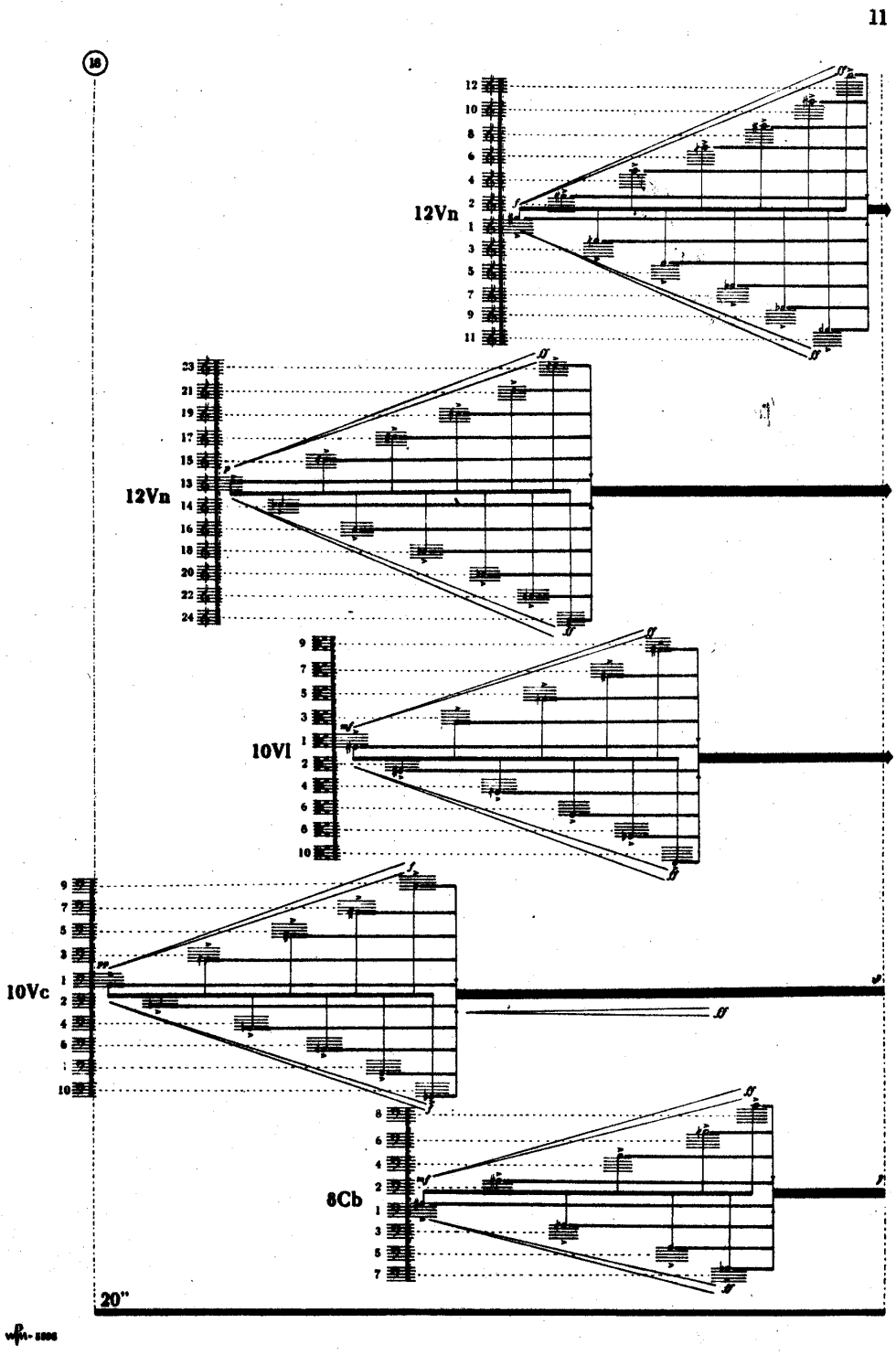


Figure 1.8: Page 11 of *Threnody to the Victims of Hiroshima* by Krzysztof Penderecki

places emphasis on their ability to be distributed, saying that one function of a digital score is to:

> support the distribution of the music idea remotely and to enable the enactment of the musical work in a situation disconnected from the immediacy of the composition of the idea. To function in such a way, the digital score must have a durable ability to facilitate an accurate reproduction of a singular musical idea over repeated performances, even without the prescence of the composer. (Vear Chapter 2)

While graphic and animated scores are an effective alternative form of notation, text-based scores are also effective methods of musical communication. Text-based and event scores such as *Make a Salad* by artist Alison Knowles (Walker Arts Center) gained popularity with the Fluxus score events in the early 1960s:

> Many scores by Fluxus artists contain instructions that could be performed by anyone, anytime, anywhere. Some could be realized easily, others are more abstract and remains acts executed only in the reader's imagination. Fluxus-scores can be perceived as the instruction to do the something but also, in its material quality, as the artwork itself. (Peterle 439)

Composers John Lely and James Saunders provide an in-depth commentary into a number of text-based scores or 'verbal notation', which they describe as 'an approach to scoring that uses the written word, as opposed to symbols, to convey information to whoever chooses to interpret it' (xix). Virginia Anderson outlines two categories for text-based scores: allusive and instructional. Anderson further describes allusive scores as 'conceptual or ambiguous' (133). My own piece, *Yurndu akanatha Oratunganga* (see



Figure 1.9: Alison Knowles performing *Make a Salad* (Walker Arts Center)

page 5), is a good example of allusive scores: the description of the performance is equivocal and left to the performers to interpret. The second category (instructional) is described as a method in which a composer prescribes the performance instructions. Often, such instructions include performance methods, stage layout directions, or even duration, of which *4'33"*, by John Cage, is a famous example.

The *Harmolodics* theory, initially coined by saxophonist and composer Ornette Coleman, finds itself somewhere in between instructional and allusive. It represents a compositional or improvisational philosophy, which Coleman describes as:

> the use of the physical and mental of one's own logic made into an expression of sound to bring about the musical sensation of unison executed by a single person or with a group. Harmony, melody, speed, rhythm, time, and phrases all have equal position in the results that come from the placing and spacing of ideas. (Coleman 54)

The theory can be likened to the philosophy of 'affinity dynamics' as developed by composer Anthony Braxton, which he describes as 'the vibrational attraction mechanism which determines how a given individual or person moves towards defining—and interpreting' (75). Both the *Harmolodics* theory and affinity dynamics posit 'the idea of self-realization through the tapping of one's life experiences and historical and cultural makeup' (Frederic 138). While Coleman references the Fundamental Elements of (Virtual) Music Notation in his description of the *Harmolodics* theory, which implies it is in some ways instructional, how a performer interprets these instructions is often allusive. This can also be said for graphics scores. Their instructions are often precise, but their interpretation is allusive and left to the performer. Latham places graphic scores into two new categories, describing graphic notation as:

> A system developed in the 1950s by which visual shapes or patterns are used instead of, or together with, conventional musical notation. Graphic scores tend to fall into one of two categories. First there are those which attempt to communicate particular compositional intentions. […] Second there are those in which visual, often aesthetically pleasing, symbols are presented so as to inspire the free play of the performer's imagination in unstipulated ways. (Latham 537)

The first category can be said to be instructional (as the notation attempts to communicate precise compositional intent) and the second allusive (as the notation is intended to inspire creative interpretation). *Treatise* by experimental music composer Cornelius Cardew, is cited as an example of the second category (see Figure 1.8). Cardew rebuts the common suspicion that graphical notation is driven by aesthetics over compositional intent, saying 'a musical notation that looks beautiful is not a beautiful notation, because it is not the function of notation to look beautiful' (29). I would argue that the beauty of notation derives from its functionality, not its appearance. In my

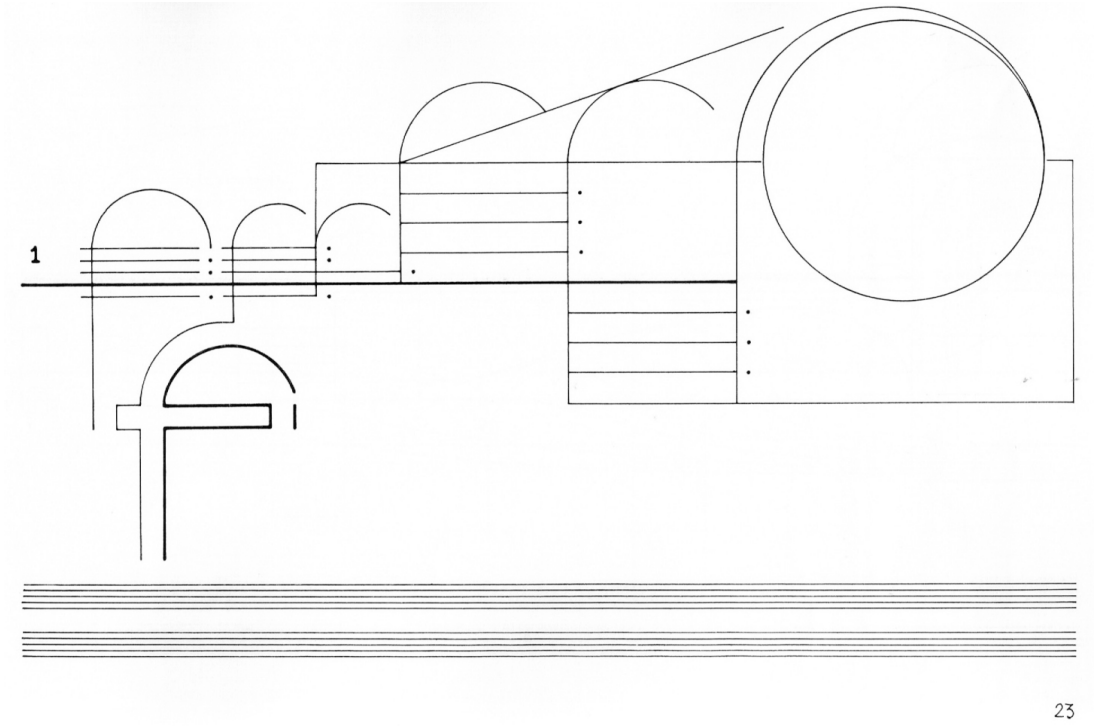


Figure 1.10: Page 23 of *Treatise* by Cornelius Cardew

own experience, there is something satisfying about notating a technique or phrase in such a way that its intent is easily understandable. While, aesthetically speaking, the same indication could have been written in a less beautiful way, its ability to translate musical instruction and intent into performance makes it beautiful. Its beauty comes from its ability to make the composer's intention accessible to the performer.

1.5 Supporting Accessibility

The primary aim of Virtual Music is accessibility (see section 1.2). The term 'accessibility' means different things for different people. In the field of public transport, researcher Katrin Lättman et al. defines the term as 'how easy it is to live a satisfactory life with the help of the transport system' (501). In reference to digital technology, computer technology historian Elizabeth R. Petrick writes, '[l]ike buildings, personal

computers had to be made accessible, with features built into them that allowed people with a variety of bodies and abilities to use them' (6). We can infer from this that accessibility is the quality of being easy to approach, use, or understand, with no restriction on ability. It can also be said to refer to the ability of a medium to be consumed by people either unfamiliar with the medium or of different backgrounds or abilities. Media scholar Aaron Scott Humphrey argues that, with regard to comics in an education and academic setting, '[c]omics can be used as sophisticated tools at a high level of education' (7). Humphrey describes the creation of the *Intern Comic*, an educational comic which was handed to new doctors at their inaugural day of training at the Mackay Base Hospital, and which they clutched 'like a security blanket':

> After the welcome packs were passed out, the interns took their seats in chairs arranged around the edge of a large classroom. While waiting for the education program to start, some interns began to flip through the materials they had been handed. Some overlooked the comic, but others immediately read every page. A few laughed or pointed out aspects to their new colleagues.
>
> Later that week, the interns had their first placements on the wards of the hospitals, and it was reported back to me that a few interns had brought their copies of the comic there, clutching them like a security blanket as they were guided through their new workplace as they began their professional practice as doctors. (Humphrey 314)

I relate very closely to this through the ever-funny and accessible *xkcd* web-comics by Rundall Munroe[9]. *Wikipedian Protester*—seen in Figure 1.11—shows a protester questioning the validity of a politician's claim in the same manner that editors of the online encyclopedia *Wikipedia* use to question the validity of an article's claim (Munroe, 'Wikipedian Protester'). Munroe's *Exploits of a Mom* brilliantly explains the importance

[9] Accessible at https://xkcd.com/

of sanitising data input for database queries. These allowed me to feel included in the computer science world, a field in which I had no previous experience. To assist with VIRTUAL MUSIC's aim of accessibility, it only makes sense that I should make this thesis accessible. This inspired the *xkcd*-styled comics and graphs throughout the thesis.

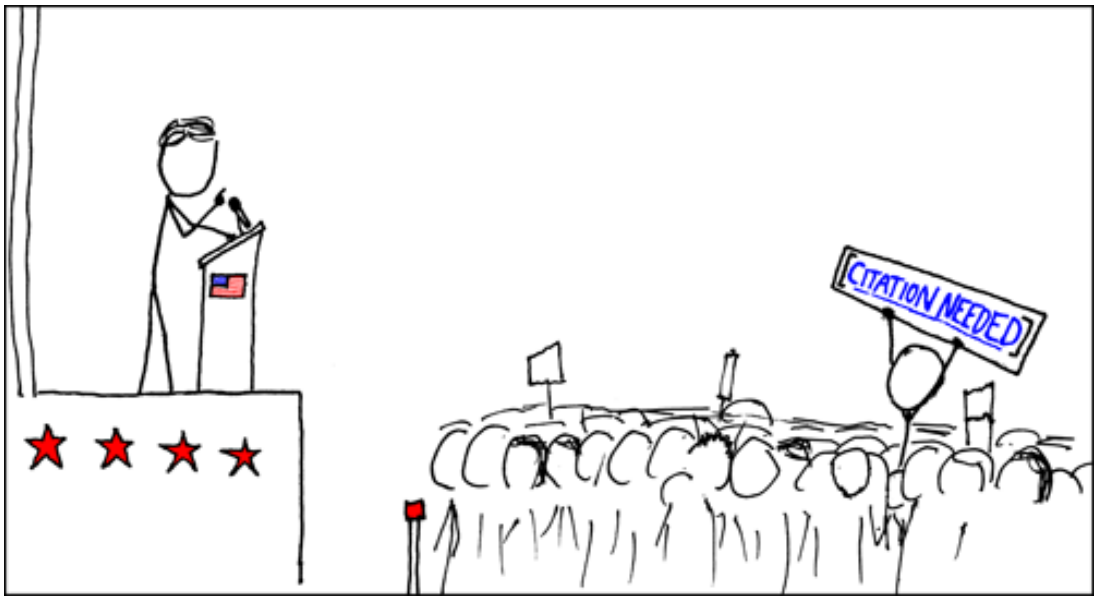


Figure 1.11: *Wikipedian Protester* by Rundall Munroe

Additionally, this aim of accessibility places importance on good documentation of the software. Such documentation is crucial and justifies the inclusion of the third chapter, *Developer Manual*. As software developer Rod Stephens states, 'the biggest purpose of documentation is to ensure that everyone is headed in the same direction' (19). Stephens later summarises the importance of documentation not just for others but for the developer:

> Without good documentation, including both design documents and comments inside the code, the poor fool assigned to fix your code will stand little or no chance. This is even more important when you realize that the poor fool may be you. The code you find so obvious and clever today may

> make no sense at all to you in a year or even a few months. (In some cases, it may not make sense a few hours later.) You owe it to posterity to ensure that your genius is properly understood throughout the ages. (Stephens 22)

I also use the term accessibility in reference to the language of this thesis. I am not disassociating myself from the language I use because it embodies the reflective process in which I am engaging to develop Virtual Music. The use of the 'I' voice places me—and the reader who can relate—within the research. Therefore, the 'I' voice is used throughout the thesis[10].

1.6 The Development of Software for Music

There is a number of existing theses and articles covering the development of music-related software. These vary significantly in style and purpose. Two of them are of particular interest to me, as they discuss the various issues surrounding the use and development of software intended for the creation of music. These sources assisted me in constructing my approach to this thesis and to the software itself.

Firstly, *jMusic*, a 'package of Java classes which provides an environment for non real time music composition', was first introduced in 'Introducing jMusic' (Sorenson and Brown 1). *Making Music with Java* by Brown (and the later *Making Music with Computers* by Manaris and Brown) discuss the *jMusic* library. These books (and the original paper) were an excellent resource in the initial design phases of this project, and when constructing the thesis. This was because the format of the books was friendly, accessible, and easy to understand. The inclusion of 'case studies' in *Making Music with*

[10]The choice to use the 'I' voice also represents an ethical choice aligned with decolonial practices in Humanities, Arts, and Social Sciences scholarship: it indicates the speaking position of the researcher. I am the sole author responsible (and accountable) for what is said within the pages of this thesis.

Computers explained how things could be achieved using the software in an easy and accessible way. This inspired the inclusion of *Case Studies* as the fourth chapter.

Secondly, 'Declarative Computer Music Programming: using Prolog to generate rule-based musical counterpoints' by Robert Paweł Wolf was an incredible resource and was on my desk for a significant time. It helped me decide on the formatting of code in the thesis, and the discussion of the 'construction of the contrapuntal expert system in Prolog' (39) informed some decisions in the initial design of VIRTUAL MUSIC. Namely, the pages following the discussion of the *LilyPond* compiling system of music notation (LilyPond) assisted in deciding the format of the VIRTUAL MUSIC SHORTHAND NOTATION.

I also found *Beginning software engineering* by Rod Stephens immensely helpful. As its title indicates, this was not specific to the development of music-related software, but nonetheless I found it an excellent encyclopedic resource and referred to it whenever I needed clarification. In particular, the analysis of Software Development Life Cycle (SDLC) methods informed the approach to the design of the software and the methodology.

1.7 Methodology

1.7.1 Software Development Methodology

The first steps of software development are simultaneously the most difficult and the most enjoyable. It is very enjoyable and even exciting to see your ideas come to life, even if only in the form of a simple prototype. The difficulty arises not from the tedious prerequisite task of planning and constructing a project brief and implementation plan,

but instead from having to withhold from writing the first lines of code and designing the first prototype.

In certain scenarios, especially when you do not care about your colleagues or your mental health, forgoing the aforementioned tedium is perfectly acceptable. I know at least one programmer that used to forgo these steps frequently. 'All I need is a script that will recursively delete all these files in this folder', this programmer said, as they happily typed up a simple script that did in fact eventually delete all the required files. This was, of course, after it had also deleted all the files in the parent directory, including the file that they had spent many hours on days prior.

If it is not obvious already, that programmer was me. I was the programmer who that day learned the importance of testing and reading documentation before haphazardly putting something together in a production environment. I also learned the importance of planning and reviewing functionality, if not just to prevent the accidental deletion of files (and sanity), but also to prevent the codebase from turning into spaghetti[11]. As Stephens writes, 'to produce applications that are effective, safe, and reliable, you can't just sit down and start typing. You need a plan' (xxi).

In the field of software engineering, an SDLC refers to the process for planning, creating, testing, and deploying new software. Due to the varied complexities of each software development project, including the initial requirements, aims, budget, and even team size, many different SDLC methods have been developed. In some cases, variations to these methods have been created to improve upon the weaknesses of particular methods in certain types of projects. Typical SDLC methods include the Waterfall[12], V-Model, Spiral, Evolutionary Prototyping, Iterative and Incremental, and Rapid Application Development models. These SDLC methods are often described as

[11]This refers to the pejorative term 'spaghetti code', which describes unstructured and difficult to maintain code.

[12]For the sake of simplicity, this includes variations such as Classic and Iterative.

'heavyweight' due to their documentation-driven nature. In response to this, 'Agile' methods such as Extreme Programming and Scrum were popularised with *Manifesto for Agile Software Development* as 'an alternative to documentation driven, heavyweight software development processes' (Beck et al.). Since choosing the correct SDLC is such a crucial step in ensuring the success of a software development project, many companies may develop an SDLC of their own. This can be seen with IBM's Rational Unified Process, specifically designed to '[d]irect the tasks of individual developers and the team as a whole' (Kruchten 15).

Before the development of Virtual Music could begin, it was necessary to reflect upon its primary aims in order to develop the SDLC that would best support the project in achieving these aims: accessibility, the ability to be a replacement for traditional stave notation, the capability for reperformance while still allowing emotive and creative interpretation, and finally extensibility.[13] In other words, Virtual Music must intuitively convey the Fundamental Elements of (Virtual) Music Notation, be easy to use, and performances using Virtual Music must be recognisable on subsequent performances. Virtual Music Notation also must not limit the performer's ability to interpret the notation's meaning nor the composer's ability to indicate their intentions in new ways. This notation must also retain uniformity, be understandable, and be comparable with traditional stave notation in functionality.

Considering these aims implied thinking about progression: Virtual Music Notation must convey the Fundamental Elements of (Virtual) Music Notation first and foremost before the development could address the next series of functionality. The development of Virtual Music therefore needed be incremental or sequential, segmented by grouped functionality so that appropriate testing could be done. This helped ensure that Virtual Music Notation still conveyed the Fundamental Elements of (Virtual) Music Notation in-line with the aims of Virtual Music. At this

[13]Referring to the quality of being designed to be 'extended' with new functionality and capabilities.

point, it was necessary to consider some of the various iterative, incremental, or sequential SDLC methods to see which worked best for this developmental approach. Of the many that exist, in particular I considered the Waterfall, V-Model, Iterative and Incremental, and Evolutionary Prototyping methods. These particular methods were selected for further investigation for their position and recognition as standards in the field (the Waterfall and V-Model methods), for their inherit incremental nature (the Iterative and Incremental model), and for their suitability to research and development projects (the Evolutionary Prototyping model).

1.7.2 Waterfall

Considered the traditional approach to software development, the Waterfall method describes linear and sequential development with distinct objectives for each phase. This method derives its name from the unidirectional movement of the development process which bears similarity to the movement of water in a waterfall: the project cascades in a flowing motion through different phases of development and cannot return back to a previous state or stage (Stephens 270). A typical phase structure as seen in Figure 1.12 segments the research and development stages (the top of the waterfall) from the design and coding stages (the falling water stream), before finishing with testing and finally deployment (the pool of water underneath).

Because of the linear and segmented nature of the life cycle, the Waterfall method is generally well understood. It is easy to manage and works well with large, geographically separated teams, as the project will progress sequentially through its phases at the same time for all teams. Yet, computer scientist Winston W. Royce describes this method as 'risky and invit[ing] failure' since the testing phase occurs at the end of development (329). A project cannot be classed as a success or failure until close to completion, because

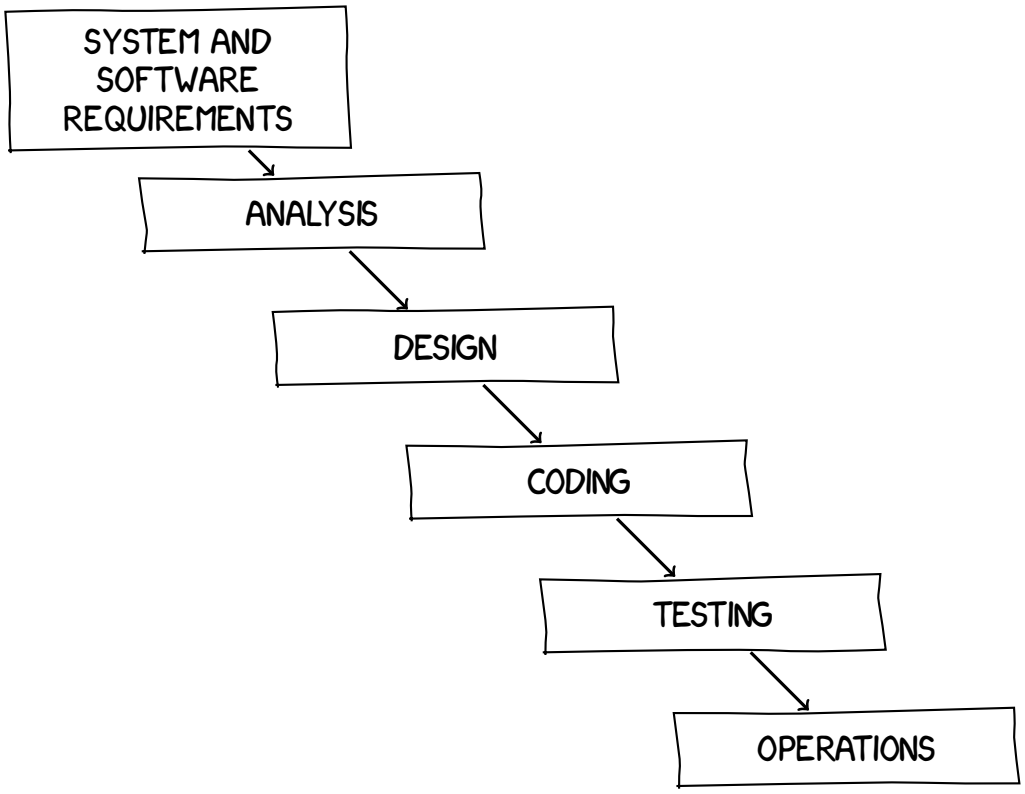


Figure 1.12: A typical Waterfall phase structure

testing to ensure that the project has met the requirements set in the first phase does not occur throughout the development. Similarly, changes to the initial requirements are also not well handled by the Waterfall method: changes require significant alteration to design, causing the entire team to move back to the appropriate stage. This backwards movement significantly increases cost and complexity. Waterfall is also not suitable for new or evolving technologies since it does not produce prototypes at the early stages of development. It is for these reasons that the Waterfall model is often replaced with an alternative model that uses it as its basis, but places focus on either testing, as seen with the V-Model method, or risk analysis and iterations, as seen with the Spiral method.

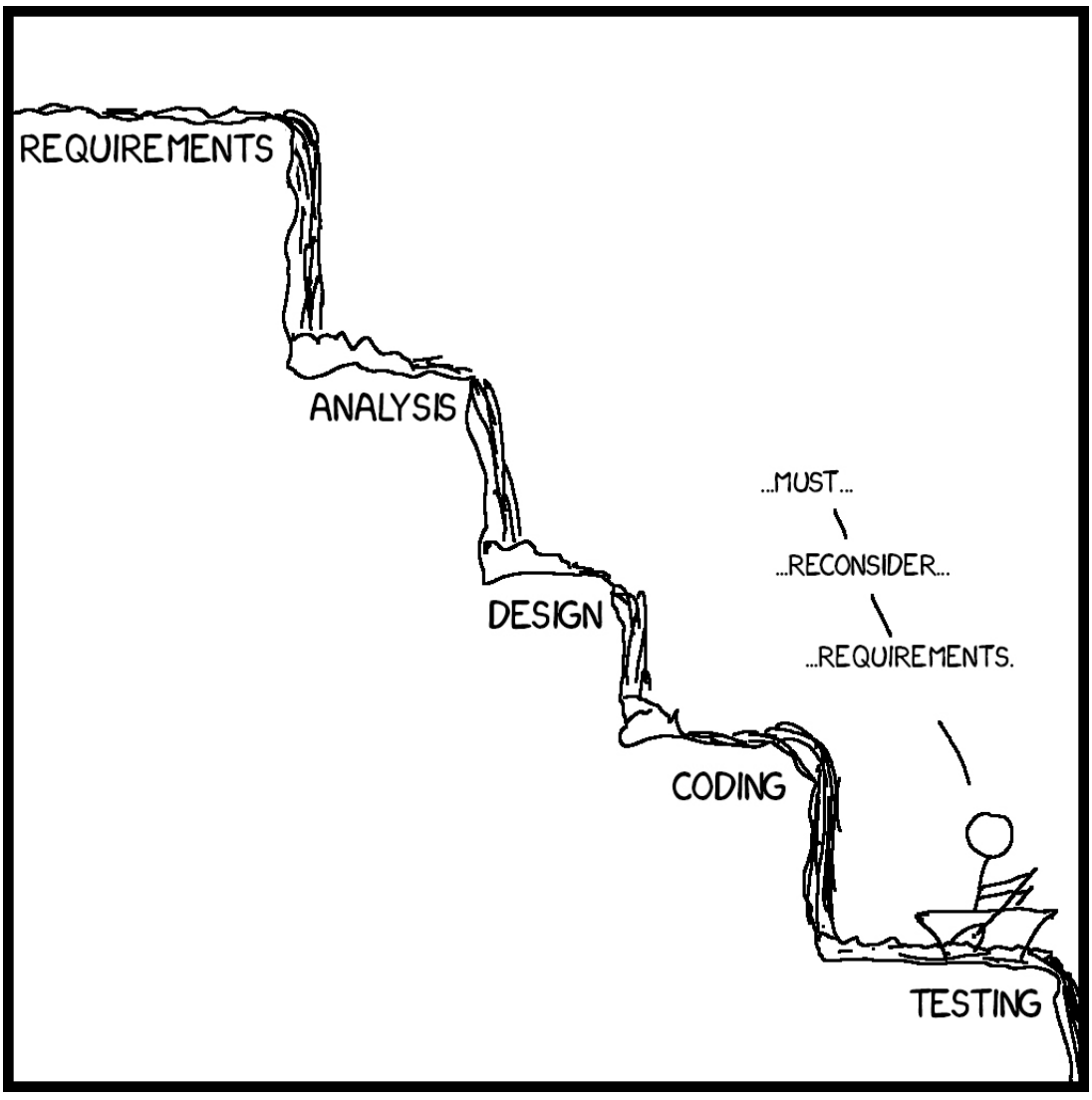


Figure 1.13: The problem with late testing

Strengths	**Weaknesses**
• Well understood • Easy to manage • Works with any type of team	• Late testing creates opportunity for the project to fail to meet the requirements • All requirements must be known from the start • Does not allow the repetition of phases or the return to a previous phase • Early prototypes are not produced

Table 1.1: The strengths and weaknesses of Waterfall

1.7.3 V-Model

The V-Model method is an extension of the Waterfall method. It places more emphasis on thorough testing by pairing a testing, verification, or validation component with each phase. After the coding stage, the phases bend upwards to form a V, demonstrating the relationship between each phase of the life cycle and its associated phase of testing, as illustrated in Figure 1.14

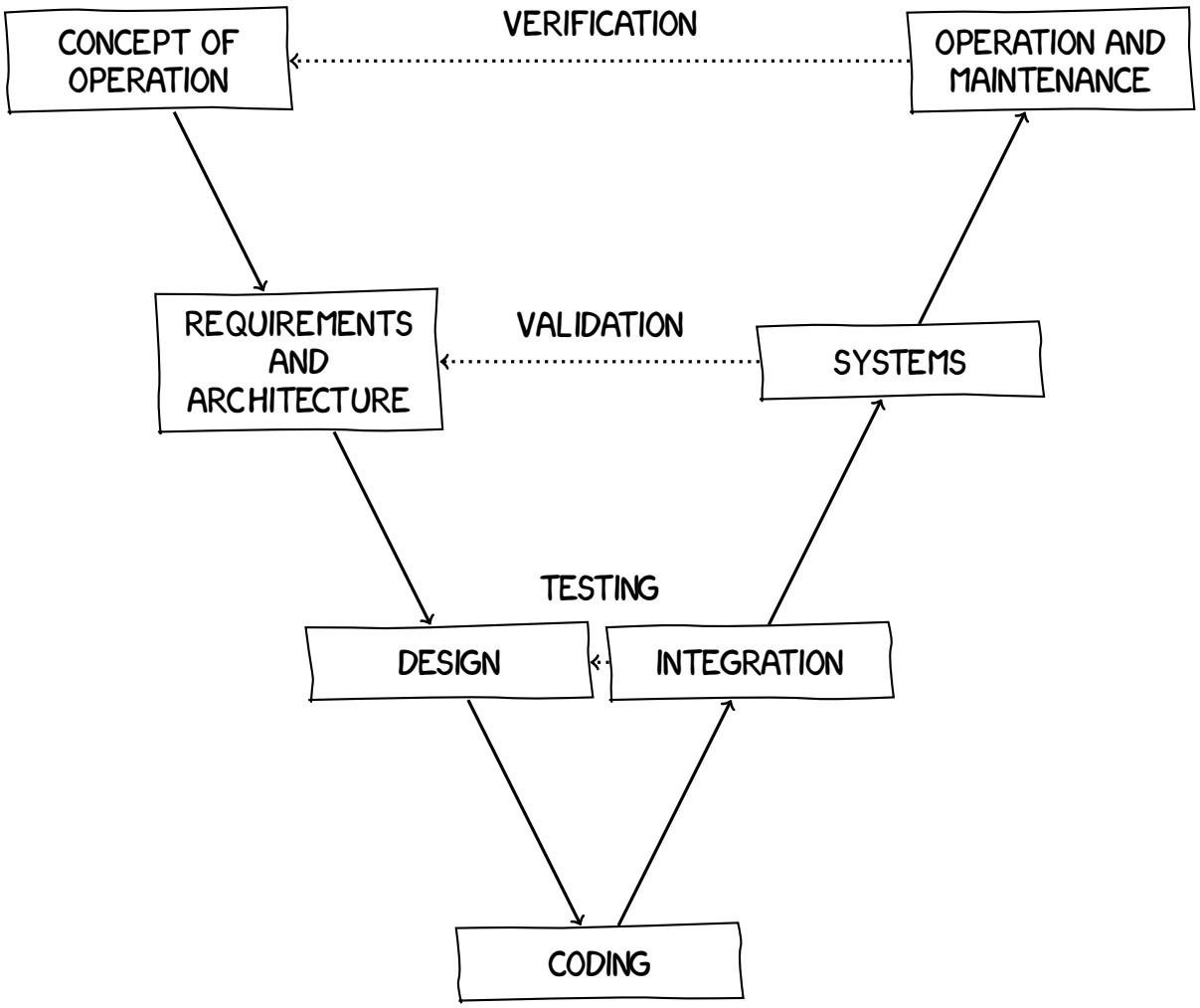


Figure 1.14: A typical V-Model phase structure

Because of its lineage, the V-Model method has similar advantages to Waterfall

method: it is well understood and its linear nature makes it easy to manage. It also has similar disadvantages: it does not cope well with changes to the initial requirements, and while the issue of late testing is somewhat resolved by the testing phases being more focused, testing still occurs after the coding stage and no prototypes are produced. These disadvantages make it unsuitable for new, untested, and evolving technologies.

Strengths	**Weaknesses**
• Well understood • Easy to manage • Has improved testing in comparison to Waterfall	• Late testing creates opportunity for the project to fail to meet the requirements • All requirements must be known from the start • Does not allow the repetition of phases or the return to a previous phase • Early prototypes are not produced

Table 1.2: The strengths and weaknesses of V-Model

1.7.4 Iterative and Incremental

Brewer and Dittman describe the Iterative and Incremental method as 'an intuitive approach to the waterfall model and is similar to the spiral model' (41). The entire development life cycle is divided into smaller, focused iterations sometimes called 'timeboxes'. Each timebox progresses through a standardised phase structure, commonly derived from the Waterfall model's phase structure. A working software version is produced at the end of each timebox, which is then used as the basis for subsequent versions (see Figure 1.15).

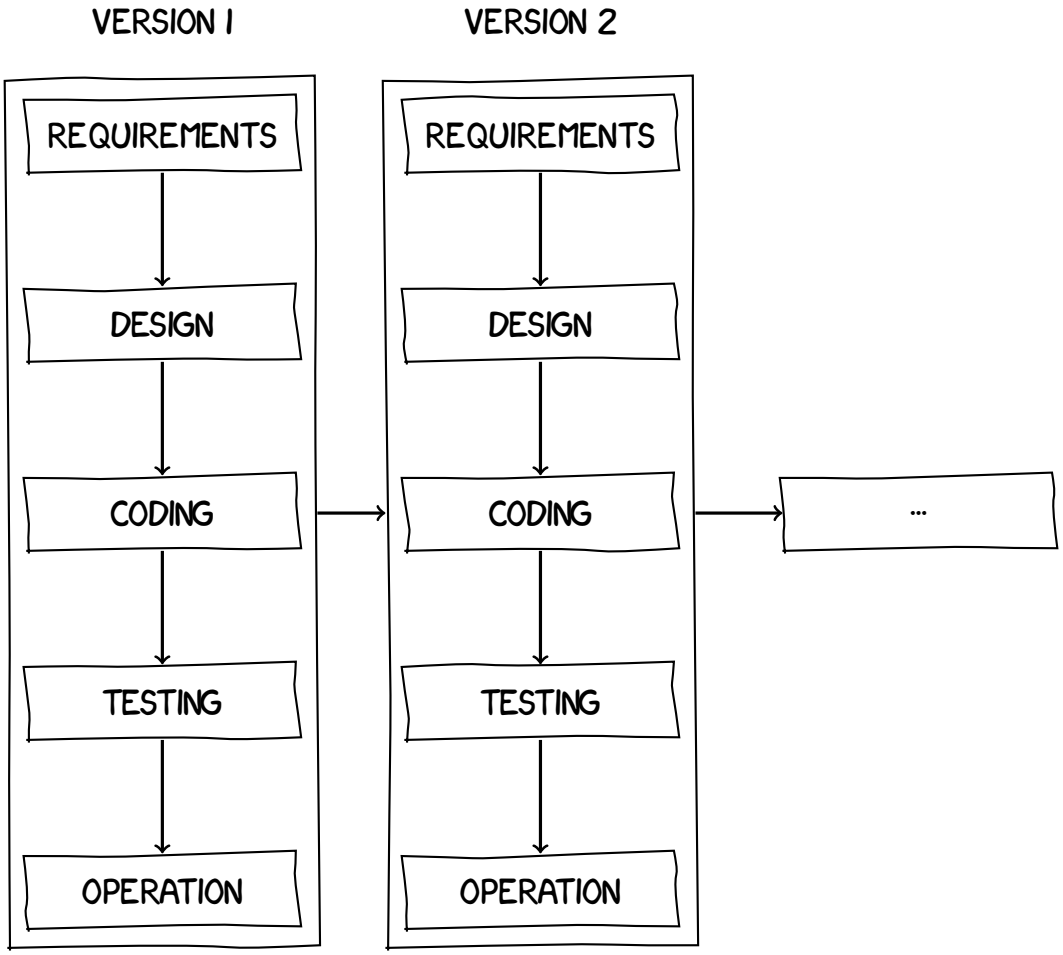


Figure 1.15: The Iterative and Incremental life cycle

The Iterative and Incremental method is generally considered difficult to manage in medium and large teams due to the complexity and frequency of the design and testing stages. These characteristics imply that the team must be experienced and communicate regularly. The time needed to produce the final working version of the software is also complicated to estimate. This makes determining costs and budget unmanageable due to the uncertainty in development time. However, since working software is produced so early, the final software produced is generally better tested and handles risk better since they can be identified and attended to early. The Iterative and Incremental method is also more flexible: changing requirements can be more easily incorporated into subsequent versions with targeted testing. It differs from the Evolutionary Prototyping method in that the Iterative and Incremental method generally iterates upon one approach to the software design, making refinements and improvements where necessary. By contrast, the Evolutionary Prototyping method discards previous working approaches in favour of new approaches that better target the project's requirements.

Strengths	**Weaknesses**
• Working software is produced early • Final software is well tested • Handles risk well • Flexible to changing requirements	• It is difficult to manage in medium or large teams • The team must be experienced • The team must communicate regularly • It is difficult to estimate costs

Table 1.3: The strengths and weaknesses of Iterative and Incremental

1.7.5 Evolutionary Prototyping

Evolutionary prototyping continuously builds working versions through a continuous stream of new prototypes or through a series of gradual refinements. Each prototype is evaluated, a plan for refinements is made and finally a new prototype is produced.

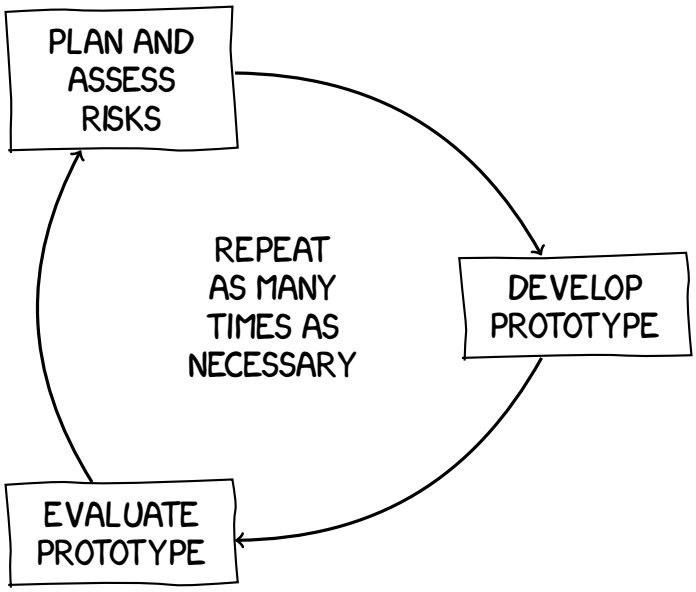


Figure 1.16: The Evolutionary Prototyping life cycle

The Evolutionary Prototyping method works well for research and development projects as requirements are discovered throughout the lifespan of the project. They are continually evolved and refined based on the discoveries of previous prototypes. Much like the nature of prototyping in other fields, some prototypes that had been built up through previous revisions may occasionally be discarded in favour of the project taking a different approach. However, the lessons learned with those prototypes can often be used for other projects or in other contexts. Project managers or clients may have a greater amount of input into projects using the Evolutionary Prototyping method, as it is easier to incorporate feedback into subsequent prototypes. They can also see steady progression between these prototypes. However, these characteristics make it impossible

to gather an accurate cost estimation due to the inability to know how many iterations will be required. These characteristics also make this method difficult to manage. The frequently changing requirements causes frequent design reorientations and a high level of communication between teams is required.

Strengths	**Weaknesses**
• Handles risk well	• It is difficult to manage
• Extremely flexible to changing requirements	• The team must be able to communicate well
• Customers or clients can see steady progress	• It is difficult to impossible estimate costs
• Well suited for research and development	• It is impossible to know how long it will take to produce a final version
• Allows easy integration of feedback	

Table 1.4: The strengths and weaknesses of Evolutionary Prototyping

1.7.6 Comparison and Analysis

Comparing these SDLC methods and collating their most common characteristics helped define a life cycle specific to Virtual Music (see Table 1.5). The strengths and weaknesses discussed for each method generally related to a handful of categories: the suitability for various team sizes, the point at which testing took place, the ease of team management, the time it took to produce working (albeit incomplete) software, the ability to manage risks, the ability to estimate costs, and the ability to handle changing or uncertain requirements. After collating these characteristics, it was possible to compare them to the known factors of Virtual Music and how they fit in regards to these characteristics (see Table 1.6).

	Waterfall	V-Model	Iterative and Incremental	Evolutionary Prototyping
Team size	Any	Any	Small	Small
Time of testing	Late	Late	Before every version	After every prototype
Ease of team management	Simple	Simple	Difficult in large teams	Difficult
Risk management	Risk of failing to meet the requirements	Risk of failing to meet the requirements	Well handled	Well handled
Cost estimation	Simple	Simple	Difficult	Impossible
Time until software	End of project lifetime	End of project lifetime	At every version	At every prototype
Requirement handling	None	None	Integration at start of next release	At planning and design stage of next prototype

Table 1.5: Comparing the characteristics of the investigated SDLC methods

Team size	One
Time of testing	Earlier and more frequent to ensure Virtual Music can meet its requirements
Ease of team management	Since the team size is only one, management problems related to team size and communication can be ignored
Risk management	The primary risk is that the aims of Virtual Music are not met
Cost estimation	Not required, but time cost management should be strict in order to submit in a timely fashion
Requirement handling	Requirements may change due to the research and development nature, but it would be best to build upon previous developments after an initial development stage rather than disregard previous versions as the reformation of the project at a late stage would likely cause the project to fail

Table 1.6: The identified common characteristics of the investigated SDLC methods in relation to Virtual Music

Since I was developing and managing Virtual Music alone, all considerations regarding team size could safely be ignored. These include weaknesses from the perspective of a team manager—such as the ability and ease of communication—and difficulties arising from the coordination of frequent testing and design phases (most notably with the Iterative and Incremental and Evolutionary Prototyping methods). Due to the research and development nature of the project, frequent testing and design changes were imperative to achieving its goals. This ruled out the Waterfall and V-Model methods, leaving the Iterative and Incremental and Evolutionary Prototyping methods. Finally, since a timely completion was crucial to the success of Virtual Music (and to the success of this thesis), a structure or timeline needed to be put into place. This was required in order to produce a viable prototype which could then be discussed and evaluated. This prototype could then be taken and refined to suit specific settings in future work. Because of this, strict adherence to the Evolutionary Prototyping method was impossible: the lack of ability to estimate time requirements meant that development cycles could not be structured.

1.7.7 Applying the SDLC Method as a Research Method

Of the four SDLC methods under scrutiny, I was thus left with only one: the Iterative and Incremental method. I favoured this method for its ability to construct a clear developmental timeline, which informed the structure of the project as a whole. I also preferred its focus on testing and refinement in comparison to other methods, which assisted Virtual Music's primary aim of accessibility by placing more importance on ensuring its features and functions were accessible. Furthermore, since this research was primarily concerned with the development of software, it made sense to embed the Iterative and Incremental method within the methodology too—or more precisely, to shape the methodology around this method. Crafting a space where the format of the thesis (including the composition of the music) could follow a pattern similar to the development of the software meant that software and thesis could grow together. They could mirror and respond to each other at every phase of the development, and as final products too. The resulting relation between software and thesis meant that each was enriched through ongoing dialogues and exchanges, rather than impoverished by the reinstatement of the supposed (and limiting) dichotomy between practice and theory—between creativity and scholarship. My use of the Iterative and Incremental method can thus be seen as aiming to bring creativity and scholarship together on an equal footing, in settings where each come to support and feed the other. Interestingly, this use of the Iterative and Incremental method as a research methodology is in many ways congruent with the use of a more traditional research methodology, the reflective practice model, where each phase of the research comes to inform, define and refine the next in a constant, self-actualising reflective loop made of three phases: design, develop and compose. My endeavour to bring software and thesis development together via the Iterative and Incremental SDLC method could thus easily be translated and

implemented within both academic and creative contexts.

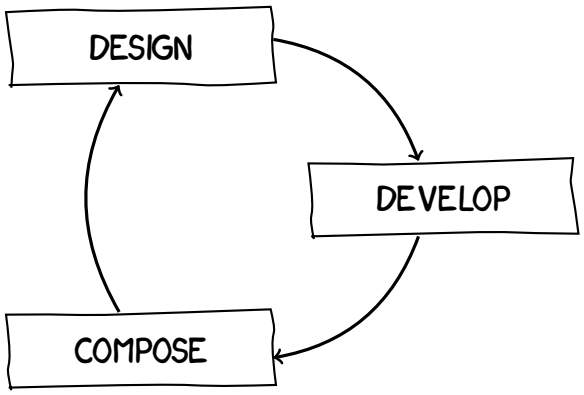


Figure 1.17: The reflective practice approach

The research and SDLC method is divided into separate major versions or 'releases', with at least one subsequent minor release refining the functionality. These releases are equivalent to the timeboxes of the Iterative and Incremental method, as seen in Figure 1.15. First, I considered the requirements and aims of the release which informed the design phase. The design requirements and aims of each release were usually tracked via 'issues' in the *GitHub* web interface. As GitHub describes: '[i]ssues are a great way to keep track of tasks, enhancements, and bugs for your projects'. All issues added to the Virtual Music repository can be found in Appendix B. After the design requirements had been set, I moved onto the development or 'coding' phase. The development phase of each release focused on implementing the planned features and 'closing' the issues created in the design and testing phases. In-depth and well documented testing then occurred after each release in the form of a standardised set of questions called 'Performance Tests'. These Performance Tests informed the requirements and design phases of subsequent releases.

The releases were numbered with a version number following a `vX.Y.Z` pattern, taking inspiration from *Semantic Versioning* (Preston-Werner). The first digit represents a major release. As Preston-Werner describes, these numbers increase when

'incompatible [Application Programming Interface] changes' are made. The major release segment for all releases for VIRTUAL MUSIC remains `0` to signify that the software is still in development. That number will only increase to `1` once I believe VIRTUAL MUSIC is complete and at a production standard. The second digit represents minor releases. In VIRTUAL MUSIC's development, they signify the targeted feature set. To better support the aim of being a replacement for traditional stave notation, the SDLC began with an extensive period focusing on the Fundamental Elements of (Virtual) Music Notation (see subsection 1.4.1). These were the `v0.1.X` releases. The `v0.2.X` releases then focused on the aim of allowing emotive and creative freedom by offering timbral alteration and polyphonic ability. Following this, the `v0.3.X` releases introduced articulations and new features supporting the aims of extensibility and accessibility. Improvements to the sythnesiser were made in the final `v0.4.X` releases. Finally, the last digit symbolises 'patches'. They signify changes to functionality related to the targeted feature set.

The version numbers of VIRTUAL MUSIC begin with `v0.1.0` as the first release, then `v0.1.1`, `v0.1.2`, and so on for subsequent patch releases, before `v0.2.0`, `v0.2.1`, and so on. The `v0.1.X` development phase was an extensive phase compared to subsequent minor releases (having four patch releases compared to two for others). This was because this research and development phase included elements from the Evolutionary Prototyping method: some elements were completely discarded in favour of solutions that worked better when considering the project's requirements and aims.

Each release was usually accompanied by a short composition called an étude, named after the French term, which simply means 'study'. There are eight études and ten software releases in total. These études, as the term suggests, were designed to assist in studying and testing the changes made and additional functionality introduced throughout the development of their accompanying software release. A performance of the accompanying étude was done and recorded for each software release to test the

functionality. My thoughts and considerations were then documented with a prepared set of questions designed to identify the focus for the next development phase. Thus, these performances were called 'Performance Tests' as they were tests or experiments in the form of a performance. Additionally, they tested the ability of Virtual Music to be used in the performance of music. They are intended to support the research by allowing focused testing which targeted changes made in their accompanying software release. This reflects how the Iterative and Incremental SDLC method was also used at times as a composition method. The études were incremented upon during the software development to include new features. Once the intended design and development stages were completed, two additional pieces were composed to demonstrate some of the capabilities of Virtual Music. Those pieces are longer in form and prove that Virtual Music is viable as a form of notation, performance, and composition.

It is important to note that Performance Tests during the development stages within this thesis were intentionally limited in user base. This was done so that widespread testing could be conducted once the project reached a stage where the basic functionality as described in this thesis was as I intended.

1.8 Format and Chapter Outline

This thesis discusses the development of music-related and virtual reality software. It is supported by a small series of compositions created specifically for the software and notation system as designed in the thesis. These compositions were designed to test the focused development phases discussed in the thesis, while proving that Virtual Music Notation is viable as a system of notation. Since these compositions are in a notation style that is not always translatable to traditional stave notation, video recordings of the compositions being performed accompany this thesis. Equivalent translations to

traditional stave notation have been provided where possible and convenient. The folio of compositions therefore supports the major work, the thesis itself, by showcasing and supporting the development of the software as discussed in the thesis.

The format of this thesis is reflective of the development process and of the SDLC. It transitions between chapters from initial design, to the first complete prototype, to the discussion of future work and potential design elements of future software iterations.

1.8.1 Chapter 2 Overview

The second chapter, *Developing Virtual Music*, discusses the development process of Virtual Music. It demonstrates the SDLC and its various increments and iterations through four focused development phases: Pitch, Rhythm, and Dynamics; Polyphonic Ability and Timbral Alteration; Articulations and Extensibility; and finally Improvements to the Synthesiser.

Each design phase has at least two software versions called 'releases' and their accompanying 'Release Notes'[14] (see Appendix C.1 for an example). These releases refer to the point at which the implemented features were tested with a Performance Test (see Appendix C.2 for an example), and the point where the SDLC method cycled to the beginning of its next phase, informed by the discussion in the Performance Test. Video recordings of all Performance Tests are provided at Appendix V.

Expanding on the methodology section (see section 1.7), this chapter further justifies and illustrates the format chosen for this thesis. Instead of the standard portfolio of compositions supported by an exegesis, the discussion and developments contained in

[14]The terms 'Release Notes', 'Change Logs', and 'Patch Notes' are usually synonymous and are commonly used to refer to a list of changes accompanying a new software version or release.

this thesis are here supported by the compositions. They prove that the software works as intended and is congruent with the project's aims.

1.8.2 Chapter 3 Overview

The third chapter, *Developer Manual* is a curated and descriptive guide on how a composer, performer, or developer may use Virtual Music to assist in achieving their goal or reaching their creative vision. However, the contents should not be considered simply as a set of instructions covering all available features of Virtual Music, but instead as a descriptive analysis of just some of what can be achieved with Virtual Music. The guide also frames a discussion of how Virtual Music has the potential to be used in a number of different ways in a number of different fields, some of which are discussed further in chapter 4. Additionally, the *Developer Manual* describes how some of the features of the pieces composed for this thesis were achieved and how a composer or developer may achieve similar effects and results with their own compositions.

The documentation discussed in the *Developer Manual* is a curated selection of what was created throughout the entire development of the software. The documentation was written and updated as necessary upon the change of functionality. As such, the content of this section was iterated and incremented upon, as dictated by the methodology. Compiling the documentation also provided a good opportunity to reflect upon the development process and draft ideas for the compositions (discussed in chapter 4), and for future work (discussed in chapter 5). Additionally, not including such documentation in this thesis could have led to the stagnation of Virtual Music, and, considering the ongoing nature of the project, risk its failure entirely. Detailed technical descriptions of all fields and properties of classes that are crucial to the functionality of Virtual Music are additionally provided as technical listings in the appendices. A

detailed technical listing of the `Piece` class, for example, is provided in Appendix P. These appendices are included to provide a developer a better understanding of how they may better use the classes of Virtual Music for specific functionality that is not covered in the *Developer Manual*.

1.8.3 Chapter 4 Overview

Case Studies discusses the application of Virtual Music in two case studies. These include the implementation of Virtual Music into a video game—as seen in section 4.1—and into an experiential exhibition-like experience, seen in section 4.2. Both cases are supported by a composition specifically targeted at their accompanying implementation. These compositions are longer in duration than the études discussed in the previous chapters and are intended to demonstrate the use of Virtual Music in these applications.

1.8.4 Chapter 5 Overview

The final chapter of this thesis summarises the development of Virtual Music throughout my candidacy. It also identifies additional features necessary for the implementation of Virtual Music in specific scenarios, and discusses other potential implementations and use cases of Virtual Music. As the nature of the research is ongoing, any conclusions made are relative to the final version of the software discussed in this thesis (`v0.4.1`), or are speculative. There is great potential for further application of Virtual Music. This means that the software version discussed in the final chapter should be considered the first viable prototype of Virtual Music which will be further refined to suit more specific settings.

2 | Developing Virtual Music

A Journey from Initial Design to Finished Product

2.1 Requirements and Specifications

As described in the project's SDLC (illustrated in Figure 1.15), it was necessary to first consider Virtual Music's aims. The project's requirements, specifications, and desired functionality also had to be considered. These considerations impacted technology use, such as which hardware, software packages, and programming language Virtual Music should work with and offer support for. Virtual Music's aim of accessibility extends to developers, and includes its easy implementation projects and systems. It also encompasses the idea that a performer should be able to use Virtual Music without requiring extensive training or hardware which is specific, proprietary, or difficult to obtain. Therefore, it was necessary for Virtual Music to offer support for hardware and software packages that are widely available and inexpensive.

2.1.1 Supported Hardware and Software

Writing this thesis in 2021, it is easy to forget just how far the VR industry has come in the last five to ten years. Currently, consumer VR offers a wide variety of options. These range from relatively inexpensive and portable options such as the *Quest 2* by Oculus and *PlayStation VR* (an accessory of the *PlayStation 4* and *PlayStation 5*), to systems targeted at the 'enthusiast' consumer, including the *Index* by *Valve*. Importantly, at the time of writing, most (if not all) of these options include trackable controllers.

When I started this research in 2016, consumer VR existed in very few commercially available forms that fit the project's accessibility requirements. VR was at a relatively early stage of development and did not have many traits or technologies considered as standard throughout all platforms. Trackable controllers, for example, were not as

prevalent in comparison to today's standards, and in some cases were unavailable for the system or only available as an optional accessory[1]. This made the decision of which VR system VIRTUAL MUSIC should support an important one, as it would directly impact accessibility.

Popular and commercially available VR systems included the *Rift* by Oculus released in early 2016, the *Vive* by HTC released shortly afterwards, and the *PlayStation VR* by Sony Interactive Entertainment released in late 2016. I also considered the *Hololens* by Microsoft (available from early 2016) as an augmented reality alternative, but quickly ruled it out as a viable option, because of its status as a development kit and its high price tag. Since *PlayStation VR* only supported the *PlayStation* platform, it was also removed. Its single-platform architecture would have limited ability for real-world implementation. These VR systems are laid out in Table 2.1, comparing their cost, availability, and feature set in late 2016.

Device	Reality type (XR)	Controllers	Release date	Price	Platform
Rift	VR	Announced but not yet available	March, 2016	$600	Personal Computer
Vive	VR	Included	April, 2016	$800	Personal Computer
PlayStation VR	VR	Included	October, 2016	$399	*PlayStation*
Hololense	AR	Not applicable	Not applicable	$3000	Personal Computer

Table 2.1: A comparison of considered VR systems[2]

[1]The *Rift* by Oculus, for example, initially shipped with an *Xbox One Wireless Controller* by Microsoft.

[2]To elaborate on the table headers: the second column, 'reality type' refers to the primary reality type that the device was designed for. The fourth, 'release date', refers to the public release date to consumers in the United States of America. Finally, the fifth column, 'price', refers to the price of the device at the time of release in the United States dollar.

Each VR system also supported different development platforms called 'game engines'[3] out-of-the-box through a Software Development Kit. Such engines include *Unity* by Unity Technologies and *Unreal Engine* by Epic Games. Other engines, such as *CRYENGINE* by Crytek, also included support for VR devices and the *OSVR* open-source software platform for VR, but were not officially supported by any VR manufacturer, including Oculus and HTC, through their Software Development Kit. However, each VR system also had resources available so that they could be implemented in custom 'native' engines built from scratch. Game developer Michael Kissner argues:

> Why write an Engine? The short answer is: don't, if you can avoid it. [...] The current selection of excellent Engines such as *Unity*, *Unreal* or *CRYENGINE* are as flexible as one could hope and can be used to make pretty much any game. For more specialised tasks, there are of course more specialised solutions like *Adventure Game Studio* or *RPG Maker*, just to name a few. Not even the cost of commercial grade Engines is an argument anymore.
>
> There are only a few niche reasons left to write your own engine: you want to learn how an engine works; you need certain functionalities that aren't available or the available solutions are unstable; you believe you can do it better/faster; you want to stay in control of development. (Kissner)

Creating a new engine specifically for use with Virtual Music was initially an attractive idea: it could allow better support for specific features as the engine would be built around the project requirements. However, it would greatly increase the complexity of the project by significantly reducing the accessibility of Virtual Music to developers: developers would need to learn code syntax and practices specific to Virtual Music and be limited to what was implemented in the engine, including

[3]While they are called 'game engines', or more simply 'engines', their intended use reaches far beyond gaming applications.

physics and graphics options. Ultimately, considering Kissner's four points negated the option of creating a custom engine. Did I want to learn how a game engine works? No, not really. Did I need certain functionalities that were not available in other engines? No, they each were compatible with VR, and sound synthesis could be patched in one way or another. Did I believe I could do it better or faster? Absolutely not. Did I want to stay in control of development? Yes, but I wanted to be able to focus on the features of Virtual Music and not have to worry about implementing features like rigid bodies and integrating a graphics system like the *OpenGL Framework*.

Once the option of developing a custom engine was removed, I was left with three other options: *Unity*, *Unreal Engine*, and *CRYENGINE*. As game developer Răzvan C. Rădulescu states in *Choosing the right game engine*, 'when it comes to picking the right tool for the job, there isn't a simple answer that works for everyone'. Rădulescu lists five different considerations a developer must reflect upon before choosing a game engine suitable for a project:

- proprietary[4] versus open-source
- documentation, support, and community
- game editor and customisation
- available programming languages
- performance.

These game engines differed in many aspects, from availability and cost through to extensibility, performance, and their native programming language. For the purpose of narrowing down these options, I ignored some advertised features when comparing them. My choices were linked to the fact that these features were often similar with

[4]The term 'proprietary' and 'closed-source' are synonymous in this case.

varying terminology, or were not relevant to the aims and scope of this project. This includes *CRYENGINE*'s 'real-time dynamic water caustics' and *Unity*'s 'splash screen customisation', for example, which were irrelevant to the project. Similarly, I decided to discard performance differences, which would have no impact on the scope of the project, or the usability of the engine in relation to the project's requirements. It is also important to note that, while additional programming languages or programming methods may be supported through third party extensions or modifications, only the languages advertised and supported by the developer out-of-the-box were considered.

Engine	Closed/Open source	Native programming language	Extensibility
Unity	Closed source	C#	Easily extensible through scripting and plugins
Unreal Engine	Source code available for viewing, but license remains closed	C++ or visual 'blueprint' scripting	Extensible through plugins
CRYENGINE	Open Source	C#, C++, LUA, and visual scripting	Somewhat extensible through plugins or via adjustments to the original source code

Table 2.2: A comparison of considered game engines

Engine	Pricing model
Unity	Free for individuals with less than $100,000 revenue or funding in the last twelve months, or monthly fees with no royalty payments
Unreal Engine	Free for creators publishing free projects, or 5 per cent royalty payments once exceeding $1,000,000 lifetime gross revenue
CRYENGINE	5 per cent royalty payments once exceeding $5000 annual revenue

Table 2.3: The pricing models for the considered game engines

As both VR systems had a similar level of engine support, deciding which VR system to use was largely influenced by availability and level of community support. The *Rift* was less expensive (albeit without controllers) than its HTC counterpart and readily available. The Oculus developer support forums also seemed more active than those of HTC. This meant that if I had any issues with the hardware, I would be able to search these forums and find answers quickly. Personal situation and preference also influenced this decision to some degree: having been a keen follower and VR enthusiast, I already owned a development version and consumer version of the *Rift*.

Similarly, the decision of which engine to support was influenced by anecdotal experiences within each engine's respective support community, and again by personal experience and preference. I had experience developing applications in *.NET* and C#, which meant that I was familiar with the syntax necessary to develop in *Unity* and *CRYENGINE*. This familiarity would enable me to spend most of my time designing Virtual Music rather than searching *Stack Overflow* for answers on how to set up a `for` loop, for example. C#, as a high-level language, also managed memory itself through garbage collection, of which I had become very accustomed (or rather, reliant), as it 'takes the burden of explicitly managing memory away from the programmer' (Frampton 6). While Puranam argues that learning a low(er)-level programming language like C++ would be beneficial to developers in the long-term, learning the proper management of memory would be a roadblock for many users new to the language:

> Learning C++ definitely takes a lot of effort. Even in comparison with C#, it is more effort centric. An in depth understanding of programming, memory allocation and pointers is a must to learn C++. (Puranam)

Using C++ would have impacted Virtual Music's aim of accessibility by increasing

the level of difficulty for users who wished to create something with VIRTUAL MUSIC but had no knowledge of code or programming. While the extensibility of *Unity*'s *Editor* could be used to create Graphical User Interfaces (GUIs) for these users, the usually limited ability of the GUI would be better suited for specific functionality[5].

Finally, to investigate Rădulescu's consideration of 'documentation, support, and community', I looked at each game engine's documentation, popularity, communities, and usage statistics. I also considered common problems and threads when scouring through each engine's support forums (an important resource, as I soon discovered). All three had extensive tutorials on most aspects of typical game creation, and expansive forums in which a user could post a question and get a response from the community. Both *Unity* and *Unreal Engine* had large communities within popular question and answer site *Stack Overflow*, though the *CRYENGINE* tag returned limited results. Even today, by simply counting the number of issues submitted to *Stack Overflow*, it is clear how large the *Unity* community in particular is, in comparison to the *Unreal Engine* and *CRYENGINE* communities. I found the documentation of *Unity* easiest to understand and often more informative, thanks to the inclusion of examples, thorough explanations, and sample code.

Engine	Tag	Number of questions
Unity	unity3d	60,224
Unreal Engine	unreal-engine4	1780
CRYENGINE	cryengine	35

Table 2.4: The number of Stack Overflow results for each game engine's 'tag'

My findings suggested that *Unity* was more popular, and data obtained by Clement supports this: 72 per cent of respondents to a survey titled *What game engines do you*

[5]The extensibility of the *Editor* was used in VIRTUAL MUSIC to allow the creation of synthesisers in `v0.4.1`, for example (see subsection 2.6.2).

currently use? responded with *Unity*, and 27 per cent responded with *Unreal Engine*. These statistics made the decision clear: to get Virtual Music to more people, the popular and accessible *Unity* engine would be used in conjunction with the *Rift*.

2.2 Initial Design

With the basic hardware and software packages chosen, the project could transition into the initial design phase. As Virtual Music is largely a form of notation which is supported by the Virtual Music Instrument, most design trials related to Virtual Music Notation. The design of Virtual Music Notation went through significant visual changes throughout development. In particular, various methods of indicating the three Fundamental Elements of (Virtual) Music Notation—pitch, dynamics, and time (see subsection 1.4.1)—were trialled in the initial development stage, before the release of the first version. Initially, the notes—then known as 'stimuli'—appeared as spherical objects which produced tones as long as they were touched, and moved or changed shape to indicate pitch and dynamics.

These stimuli portrayed the notational elements of pitch by their height, dynamics by their radius, and time by how long they appeared for. They could be interpreted as a descriptive notation by a performer with an instrument, or touched with the VR controllers which produced predefined sounds. However, they failed to allow the performer to interpret the notation without prior context and, in some cases, failed to indicate these elements clearly. They also did not allow the performer to anticipate any of these elements. This prohibited predictable performance behaviour. The stimuli would appear to signify the start of the note and its disappearance signified the end of the note. This meant that a performer could not anticipate when the note would start or the required rhythm. Once shown a stimuli, the performer often did not begin playing a tone until

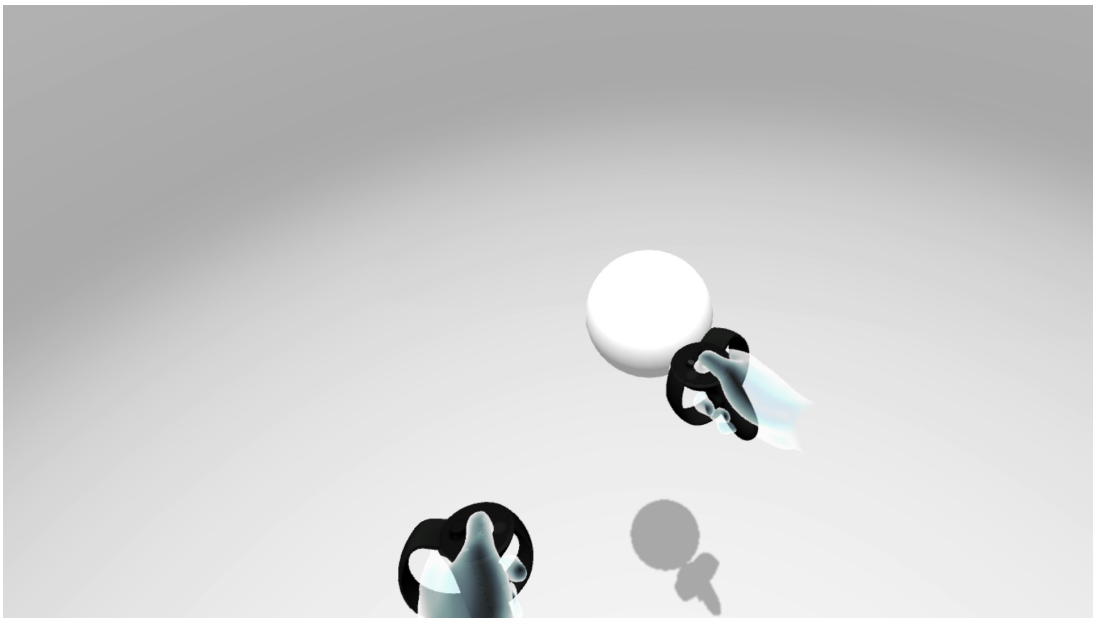


Figure 2.1: The initial spherical appearance of the 'stimuli'

a moment later, once they had had the time to process the information that they had been given. Additionally, as dynamics were indicated by the radius of the stimuli, the indication of pitch was often unclear since it was difficult to see where the centre of the stimuli was positioned vertically. This was especially evident for performers interpreting the notation on real instruments, each of which interpreted the notation differently and on different attempts. These issues additionally made it impossible to have more than one simultaneous performer.

'Stimuli' were soon abandoned in favour of a piano roll type of appearance, similar to what can be seen in most digital audio workstation software. Objects—now known more simply as 'notes'—would indicate time by their length, pitch by their height, and dynamics by their width. This allowed performers to anticipate the notation, and subsequent performances of the same notational instructions produced musical interpretations that were similar and more aurally recognisable on reperformance.

The control system also went through a number of different design iterations before I settled on a control system that met the main requirement of accessibility. Trialled control

systems initially used a second hand, but this was found to be prohibitive or confusing.

The first trialled scheme functioned similarly to a theremin: the dynamic level (rather, the amplitude) of the produced tone was controlled by the position of the left hand in relation to a predefined point. However, inversely to the theremin, tones of higher amplitude would be produced when the hand was further away from the defined position (Martín et al. 1). Another trialled control system involved setting the hand as the position at which the dynamics were based. In both cases, the control systems were found to be counter-intuitive and limiting. Additionally, requiring both hands to produce any tone would limit accessibility of Virtual Music to only those with dexterity in two hands, and would limit the ability of polytonality, a planned feature. Starting a pitch and moving the hand to the desired dynamic position was tried as a solution to this, but was found equally counter-intuitive and resulted in the production of small crescendos on each tone. Finally, using the horizontal position of the hand for dynamics was considered. However, this system was promptly discarded as it was anticipated that the hands would naturally fall into different positions and produce unexpected dynamic values: to produce the same dynamic level, both hands would also need to be at a similar position which would be impossible.

Learning from these trials, the initial control system was designed as follows:

- To produce a tone, a button would be pressed, and the duration of the note would depend on how long the button was held down.
- The pitch of the tone would be according to the height of the instrument.
- The dynamic level of the tone would be according to the rotation of the instrument.

This system formed the basis for initial software versions. With only small alterations, this method of control formed the control system of the 'Advanced' Performance Mode.

Other Performance Modes, introduced later in development, are variations of this initial control system. At this point, both the control and notation systems had gone through enough design iterations and the project was ready for further development with the first version, `v0.1.0`, and the first piece written for Virtual Music, *Étude 1*.

2.3 Pitch, Rhythm, and Dynamics

2.3.1 v0.1.0: Étude 1

The development phase for the `v0.1.0` release of Virtual Music was the longest development phase throughout the entire development lifetime. Despite this, the `v0.1.0` release was also the shortest in terms of the number of lines of code, excluding any dependencies: it only had 733 lines compared to 4259 in the final release, `v0.4.1`. This extended development phase was required to align with the project's SDLC. The basic functionality and principles formed in this release guided the direction of future iterations and increments.

This development phase focused on generating sound in the synthesiser using *NAudio* (Mark Heath) as a dependency, and developing a human-readable method to notate Virtual Music. A large portion of time was spent focused on the mathematics required to reliably and accurately convert the height data of the Virtual Music Instrument into frequency and amplitude data usable by the synthesiser.

Linear Measurements vs Music-related Attributes

The specification and measurement of space in *Unity* is done linearly on three axes: X, Y, and Z. While there is no specific term for these measurements other than simply 'unit' or 'unit in world space', it is designed so that 'one unit in world space corresponds to one metre [in the real world]' (Unity Technologies, 'Transforms')[6]. This similarity is important for the production of immersive and comfortable VR environments. It enables the developer to create objects and spaces equivalent to real-world counterparts more easily.

This linear type of measurement is not applicable to the measurement of frequency, especially in relation to the octave interval. In most musical systems, an octave is the interval between one musical pitch and another with double its frequency. Cooper describes this relationship as the 'basic miracle of music', a shared characteristic displayed in many musical systems (16). This means that the frequency between octaves increases exponentially and frequency may be calculated with an exponential function. For instance, consider the pitch A4, the first A above middle C, tuned to the modern tuning standard of 440 Hz (International Organization for Standardization). Raising A4 by one octave to A5 doubles the frequency to 880 Hz. Raising this tone an additional octave to A6 again doubles the frequency to 1760 Hz. Similarly, lowering A4 at 440 Hz by an octave halves the frequency to 220 Hz for A3, and lowering an additional octave to A2 would halve the frequency to 110 Hz. This pattern is seen throughout musical pitches.

[6]This default behaviour may be changed in the project's settings.

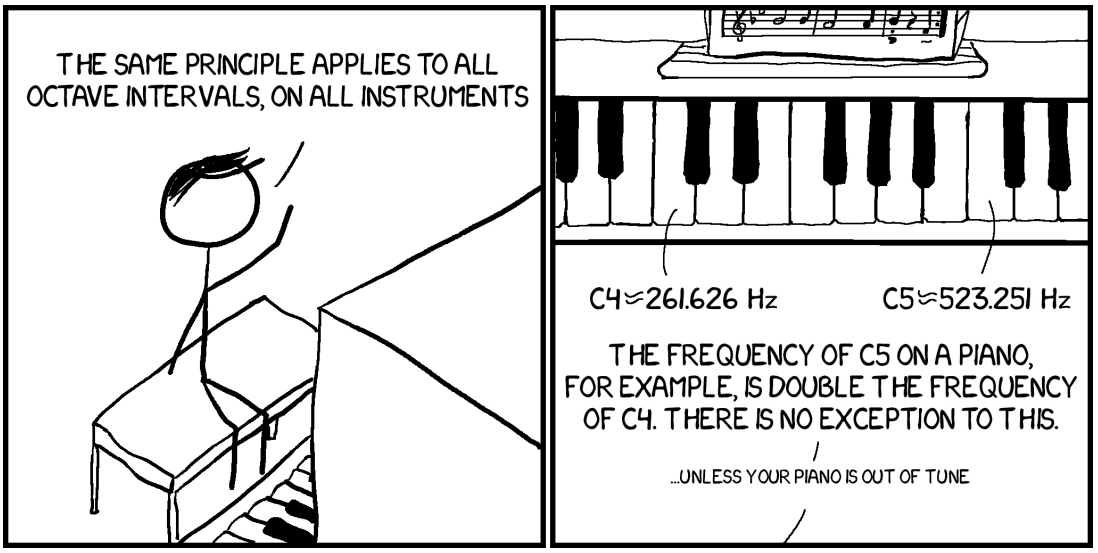


Figure 2.2: The only exception to the octave principle

Converting Measurements: First Steps

To create an exponential function to convert the linearly expressed distance measurements of *Unity* into frequency, it was necessary to first define a 'play area' in which the performer using VIRTUAL MUSIC was limited. The performer must first specify their play area by setting the lowest and highest world space points that they can comfortably reach using the handheld controllers. This also ensures that the performer is able to perform easily and is not restricted due to their physical attributes or ability. These world space points equate to the points above the ground in the real world in metres and are stored internally as a `float` at the `MinimumHeight` and `MaximumHeight` fields, respectively. `0.75f`[7] and `2.25f` were provided as default values for these fields in the `v0.1.0` release, giving the performer a play area of `1.5f` or 1.5 metres.

This play area is equally divided and stored into an array representing the number of pitches available to the composer and performer by the `GetPossibleTones` method. This method returns an array with a size of $(t \times o)+1$, ranging from zero to the maximum number of octaves, using a divisor of $\frac{1}{t}$. In these equations, t is equal to the number of

[7]The `f` signifies that the value is a `float`.

tones per octave and o to the number of octaves, as specified in the 'piece script'[8]. This formula can be seen implemented in Code Sample 2.1, an excerpt of the `Environment.cs` file. Positions outside of the play area are 'clamped' between zero and the number of available octaves set in the piece script: positions below the minimum point on the Y axis are read as `0f`, and as the number of octaves above the maximum point, `2` by default[9].

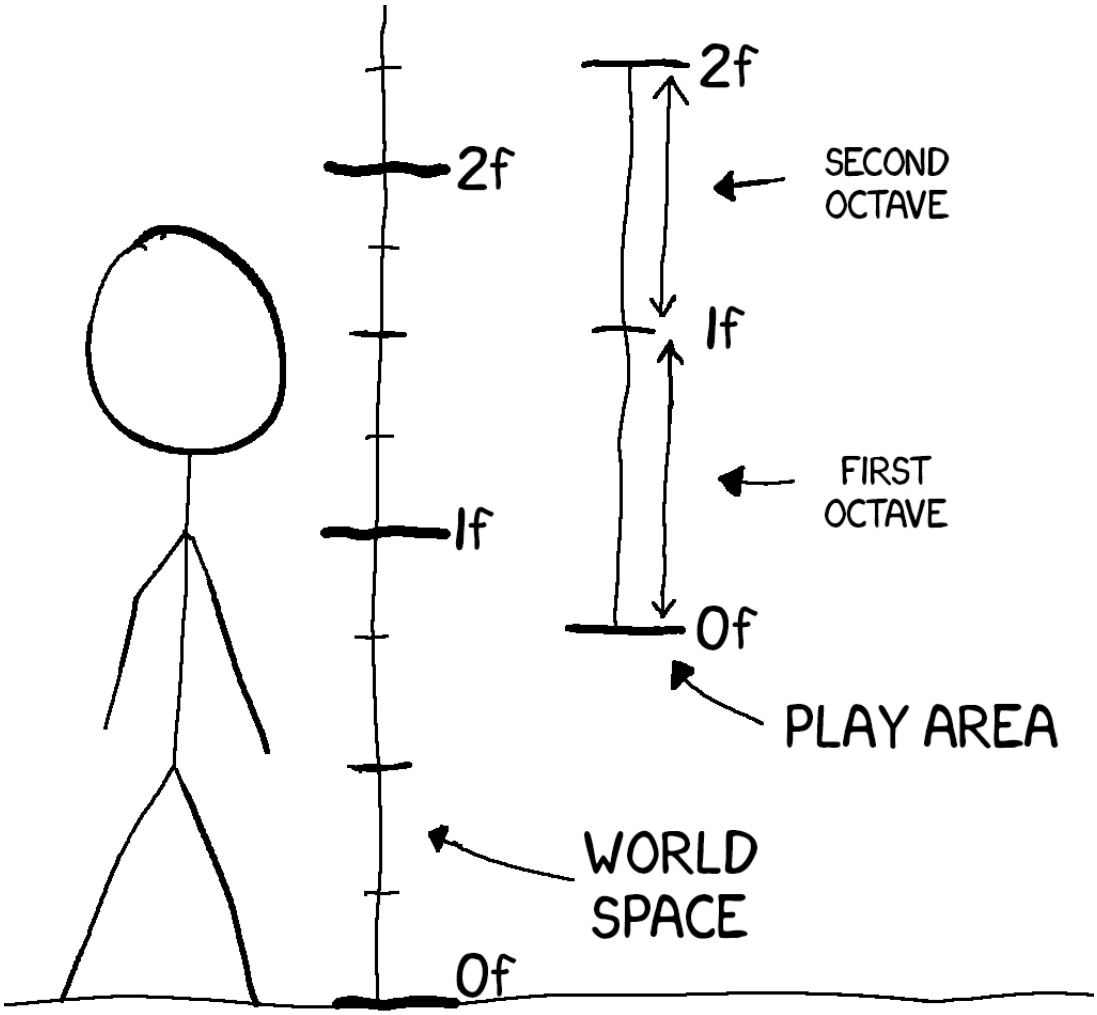


Figure 2.3: The play area

By using this method to translate the game world points to the play area, it was

[8]The piece script allows the composer to define the number of available octaves, the division of those octaves, the base frequency which is used as the lowest fundamental tone, and supply notation to the piece. It is also used to initialise a performance using Virtual Music.

[9]This value is stored as an `int` but is here converted to a `float`.

```
public float[] GetPossibleTones()
{
    float[] tones = new float[(Director.Instance.piece.TonesPerOctave *
        Director.Instance.piece.Octaves) + 1];
    float divisor = 1f / Director.Instance.piece.TonesPerOctave;
    for (int n = 0; n < (Director.Instance.piece.TonesPerOctave *
        Director.Instance.piece.Octaves) + 1; n++)
    {
        tones[n] = 0 + (divisor * n);
    }
    return tones;
}
```

Code Sample 2.1: Getting the heights for the possible tones

possible to retrieve the frequency of a pitch in two different ways. These will be called the 'wrong' way, and the 'simple' way.

Converting Measurements: The Wrong Way

The 'wrong' way is named so because it is an example of a poorly researched (but eagerly implemented) conversion system. It uses a `GetFrequency` method which contains the formula in Equation 2.1. In this formula, p is equal to the vertical position of the Virtual Music Instrument in the play area, rounded to the nearest height in the array returned by the `GetPossibleTones` method, and *base* is simply the fundamental frequency as defined in the piece script. The full implementation of this method may be found in the highly inefficient Code Sample 2.2.

$$f = \left(base \cdot 2^{\lfloor p \rfloor}\right) + \left(\left(base \cdot 2^{\lfloor p \rfloor}\right) \cdot (p + 1 - \lceil p \rceil)\right)$$

Equation 2.1: The formula to convert the linear height to frequency

The formula produces a frequency in two stages: first, it calculates the frequency of the base frequency risen to the closest octave, and second, it divides the final octave into equal parts according to the octave division. While this method of retrieving the

```
public float GetFrequency(float pitch)
{
    // limit the pitch to the confined octaves
    int pitchRoundedUp = (int)Mathf.Ceil(pitch);
    if (pitchRoundedUp > Director.Instance.piece.Octaves)
    {
    pitchRoundedUp = Director.Instance.piece.Octaves;
    }

    int pitchRoundedDown;
    // check if pitch is a whole number
    if (pitch == (int)pitch)
    {
        pitchRoundedDown = (int)pitch - 1;
    }
    else
    {
        // pitch isn't a whole number so round down
        pitchRoundedDown = (int)Mathf.Floor(pitch);
        if (pitchRoundedDown > Director.Instance.piece.Octaves)
        {
            pitchRoundedDown = Director.Instance.piece.Octaves;
        }
    }

    if (pitchRoundedDown < 0)
    {
        // cannot be less than 0
        pitchRoundedDown = 0;
    }

    float frequency;
    if (pitchRoundedDown == Director.Instance.piece.Octaves) // if maximumHeight
    {
        frequency = (Director.Instance.synthesiser.BaseFrequency * Mathf.Pow(2,
            pitchRoundedDown));
    }
    else if (pitch == 0) // if minimumHeight
    {
        frequency = Director.Instance.synthesiser.BaseFrequency;
    }
    else
    {
    frequency = (Director.Instance.synthesiser.BaseFrequency * Mathf.Pow(2,
        pitchRoundedDown)) + ((Director.Instance.synthesiser.BaseFrequency * Mathf.Pow(2,
        pitchRoundedDown)) * (pitch + 1 - pitchRoundedUp));
    }

    return frequency;
}
```

Code Sample 2.2: Implementing the linear height to frequency formula

frequency indeed divides the octave into twelve equal parts, it does not take temperament into consideration and the resulting intervals are not equally tempered. They are instead based on the pure harmonic series (Schmidt-Jones 231)[10].

This issue was initially solved with an altered array returned by the `GetPossibleTones` method adjusted to account for the approximate decimal value of each equally tempered interval. However, I desired a more optimised implementation, which led to 'the simple way'.

Converting Measurements: The Simple Way

Twelve-tone equal temperament is a tuning system where 'the "equal" ratios of its half steps are the twelfth root of two' (Schmidt-Jones 223). Since Virtual Music allows a composer to equally divide the octave into any number of intervals, equal temperament in Virtual Music consists of intervals with a ratio equal to the x root of two, where x is equal to the octave division. By default, if the number of tones per octave is not changed (or if the composer specifies an octave division of twelve), this temperament system creates twelve equally tempered tones.

The implementation of equal temperament into Virtual Music was significantly simpler and more efficient than with previous attempts to calculate frequency. Instead of using the position of the Virtual Music Instrument in the play area as a component of the formula, the pitch degree was ascertained by finding the nearest height in the array returned by the `GetPossibleTones` method and using the array index of that height. The formula as seen in Equation 2.1 was replaced with the much more concise formula seen

[10]It may be argued that this temperament system could have been used to give Virtual Music a unique sound. However, as Virtual Music was intended to be usable and accessible by musicians and non-musicians alike with the possibility of accompaniment, it was important to instead implement an equally-tempered intonation system.

```
public float GetFrequency(float pitchDegree)
{
    return Director.Instance.synthesiser.BaseFrequency * Mathf.Pow(2, (pitchDegree - 1) /
        Director.Instance.piece.TonesPerOctave);
}
```

Code Sample 2.3: The greatly simplified GetFrequency method

in Equation 2.2, where p is equal to the pitch degree and t to the octave division. This allowed the `GetFrequency` method to be greatly simplified, as demonstrated in Code Sample 2.3. Additionally, this formula is able to calculate the frequency over multiple octaves, including for frequencies below the defined fundamental frequency.

$$f = base \times 2^{\frac{p}{t}}$$

Equation 2.2: Calculating the frequency with a root of two

Audio Synthesis

Once calculated, the frequency is passed to the synthesiser component in order to generate sound. So I could better focus on the basic implementation of Virtual Music Notation and the Virtual Music Instrument, I chose to implement the *NAudio* library into the project for audio synthesis. This was implemented with the intention of replacing *NAudio* with *Unity*'s audio system so that *Unity* would be responsible for all aspects of Virtual Music. I used the library's `WaveProvider32` and `WaveOut` classes as properties of an overall `Synthesiser` class that only had basic 'stop' and 'start' type functions added. I experimented with external applications, such as *Max* by developer Cycling '74, but favoured the *NAudio* library so that it could simply be included seamlessly as a dependency and the installation of external applications would not be required. All audio generation was therefore handled directly on the sound card by *NAudio* and *Unity* was not responsible for any sound production.

Devising a Human Readable Shorthand Notation

Now that Virtual Music Notation had basic shape and notes of Virtual Music Notation could be used to retrieve data necessary for the production of audio such as frequency and amplitude, it was necessary to consider how a composer could create musical works using Virtual Music. At this point, a composer could only place the notation manually and give all notation the required components. While possible, the process was cumbersome and time-consuming. Additionally, notes had a high potential of improper representation of their musical properties, as the size and shape of the notation were not automatically set and the composer had to place and shape them manually.

The solution was Virtual Music Shorthand Notation: a human-readable string of text that could be parsed by software to create Virtual Music Notation. Virtual Music Shorthand Notation took inspiration from *LilyPond*, a compiled form of music notation similar in ideology to that of LaTeX, in which 'you write music by typing text' (LilyPond). For the `v0.1.0` release, Virtual Music Shorthand Notation had three segments that indicated the three Fundamental Elements of (Virtual) Music Notation ordered accordingly: pitch, time, and dynamics.

```
8c5 1q 3q 5q 6q 8c 1c 1c
```

Piece Excerpt 2.1: A short phrase written with Virtual Music Shorthand Notation

Pitch The definition of pitch in Virtual Music Shorthand Notation is by a number. The number, beginning from one, indicates the pitch degree. With the default fundamental frequency and octave division settings, one is equal to C3, two C♯3, three D3, and so on. Rests may be indicated with a zero. This is in direct contrast to *LilyPond*, which defines pitch with a letter. This numbered definition was chosen

instead of a letter for one reason. As a composer may provide VIRTUAL MUSIC any fundamental frequency, the note names would no longer reference their equivalent frequencies. Additionally, any alterations to the octave division would require new letter definitions. The definition of pitch by letter makes sense for *LilyPond*, as its functionality is reliant on traditional stave notation. It does not make sense for VIRTUAL MUSIC, considering these added abilities. The definition of pitch by number solves this issue.

Time The time or duration of a note is defined by a letter corresponding to a predefined rhythmic value. Like with traditional stave notation, these values are determined in reference to the length of a beat. With the exception of the semibreve, the available characters for the base rhythmic values are the same as the first letter of their equivalent note values in British music terminology, ranging from a semiquaver (a sixteenth note in American music terminology), to a semibreve (a whole note). The duration of the note can be extended with the addition of a dash to represent a tie followed by any time character. The characters are:

- `s` (semiquaver, 𝅘𝅥𝅯)
- `q` (quaver, ♪)
- `c` (crotchet, ♩)
- `m` (minum, 𝅗𝅥)
- `b` (semibreve, 𝅝)
- `-` (a tie)

Dynamics Finally, dynamics are indicated by a number ranging from one to nine. One is the lowest dynamic value and can be considered as equivalent to *ppp*, and nine is the

highest, approximately equivalent to $ffff$. If a dynamic indication is not specified, the dynamic value of the previous note is used.

2.3.2 v0.1.1: Étude 1

The first piece composed for Virtual Music, *Étude 1*, was intentionally limited in rhythmic variation in order to test the basic representation of pitch and dynamics by Virtual Music Notation. It was followed by a piece testing different rhythms specifically in later releases. The test performance of *Étude 1* proved that Virtual Music Notation had the tendency to obscure later notes, as I noted in the `v0.1.0` Performance Test:

> The effectiveness of the current implementation is mixed. Initial expectations were accurate: dynamics are represented flawlessly, but some issues with pitch, and some lesser issues with time are apparent.
>
> [The] pitch of subsequent notes is occasionally occluded by notes prior to them. This is most evident with short notes that are higher pitched than the longer notes preceding them, when both notes are above the performer's eye level. When short notes are lower pitched than the longer notes preceding them, and both notes are below eye level, the latter note is similarly occluded. (Appendix C.2)

The Performance Test also revealed that the dual requirement of touching a note *and* pressing a button on the Virtual Music Instrument to produce a tone with the synthesiser was cumbersome and could cause the performer to miss producing a tone entirely. This, in particular, caused dissatisfaction with the interface: it meant running

the risk of performers becoming frustrated and losing interest in Virtual Music altogether. This can be likened to the description that M. Gutierrez Miranda offers, regarding computer software:

> The importance of good User Interface Design can be the difference between product acceptance or rejection. If end-users feel it is not easy to learn, not easy to use, or too cumbersome, an otherwise excellent product could fail. Good User Interface Design can make a product easy to understand and use, which results in greater user acceptance. (Gutierrez Miranda 7137)

Additionally, problems with the method used to track which notes were touched caused increased dissatisfaction with the control scheme. Notes would not activate correctly and correct button presses were not processed accurately.

Based on this Performance Test, the `v0.1.1` release focused on three development points: removing the need to press buttons to create a tone with the addition of changeable 'Performance Modes', improving the clarity of the Virtual Music Notation with transparency and by approaching the performer from a different direction, and improving the touch registration.

Adding Performance Modes

The previous control system of `v0.1.0` was entirely replaced in `v0.1.1` with two 'Performance Modes': Simple and Advanced. The two Performance Modes separate the functionality of the Virtual Music Instrument and in turn Virtual Music Notation into a prescriptive method of performance and notation in the Simple

Performance Mode, and into an interpretive descriptive method of performance and notation in the Advanced Performance Mode.

In the Simple Performance Mode, the tone generation is entirely automatic and activation-based. When a note is touched, the note is activated and a tone is produced. Likewise, when a note is no longer being touched, the note is deactivated and the produced tone stops. No button presses are required, but the performer cannot produce tones other than prescribed by the notation. In contrast, in the Advanced Performance Mode, tone generation is not automatic and not dependent on the activation and deactivation of notes. Notes are activated and deactivated if the instrument is touching or not touching the note accordingly, but tone generation only occurs when a button is pressed. The notation only serves as a guide, and the performer may produce tones as they wish. In code, the Advanced Performance Mode is activated via a `FreeMode` `bool`, which is switched between `true` and `false` when the thumbstick on either hand is pressed. A value of `true` indicate that the instrument is in the Advanced Performance Mode. These Performance Modes assist in giving the Virtual Music Instrument a 'low entry fee with no ceiling on virtuosity' as Wessel and Wright describe:

> Getting started with a computer-based instrument should be relatively easy but this early stage ease-of-use should not stand in the way of the continued development of musical expressiv[eness]. (Wessel and Wright 11)

The Simple Performance Mode requires little to no training in order to produce sound. This allows new users to easily get started with Virtual Music and the Virtual Music Instrument. However, musical expressiveness is still possible to some degree by using the Advanced Performance Mode. The Advanced Performance Mode allows free musical interpretation and increased musical expressiveness through the ability to create sounds

that are not indicated. It also allows the production of sounds not possible in the Simple Performance Mode, such as glissandi.

Improving Clarity of the Notation

The clarity of the notation was low for one significant reason: notes could obscure later notes. This was addressed in two ways: with the addition of transparency, and by switching the axes of time and dynamics. This required some alterations to the way that the notation was generated, particularly for dynamics.

Adding Transparency In the development phase for `v0.1.0`, I was unfamiliar with how materials could be applied to objects and, in turn, unfamiliar with the options for materials. After I discovered the transparency options for materials (or rather, after I created a material and realised that the rendering mode could be changed to 'transparent'), I decided to implement transparency in addition to the change to the time and dynamics axes. When a note is in a deactivated state, it appears with a 40 per cent transparency level; in an activated state, it appears with an 80 per cent transparency level. This also gives the performer visual feedback of which note they are activating. Additionally, this made it simpler for me to debug touch registration issues later, as I could see exactly which notes were being activated and when those activations did not match what I expected.

Switching the Time and Dynamics Axes Switching the time and dynamics axes meant that the notation would move horizontally across the performer instead of towards the performer. By default, notation in the `v0.1.1` release approached the performer from the left. As dynamics were indicated by the width of the note and this width was

mirrored on the centre of the stave, switching the time and dynamics axes meant that notes had to instead indicate their dynamic value with their depth. As the dynamic level of a note was variable, keeping the stave at a playable distance meant that the notes could occasionally become too large and appear to go through the performer, causing the performer themselves to become the obstruction. To address this, the dynamic indication of all notes were no longer mirrored and the closest point of a note was made always consistent. This allowed the notation to still correctly indicate their dynamic level while staying at a consistent distance from the performer, as seen in Figure 2.4.

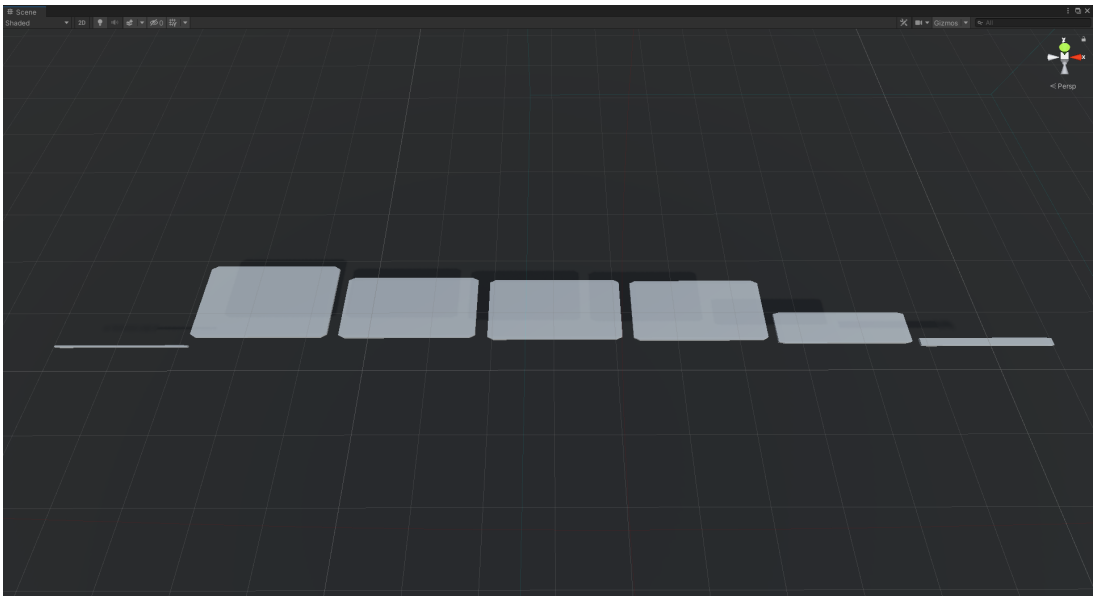


Figure 2.4: Notes of a stave that approaches from the side remain at a constant distance from the performer

Improving Touch Registration

The activation of notes was done by checking if the instrument was within a certain radius of a note every frame by using the `SphereCastAll` method from the `UnityEngine.Physics` class, which returned an array of `RaycastHit` objects on the `MusicLayer` layer. If this array contained more than one object, this meant that there

was more than one note in this radius. In this case, the instrument would select the note that was the closest to it by checking the `distance` property of each `RaycastHit` and selecting the object with the lowest distance.

This counter-intuitively had the opposite effect to what was desired, due to the nature of the stave: as the stave progressed, notes which were later in the stave became closer to the performer. This meant that notes further away from the *performer* would be expected to be activated instead, and distance checks from the Virtual Music Instrument had no relevance. When two notes were within the 'touch radius' of the Virtual Music Instrument, it was usually because the stave was progressing and subsequent notes moved either in a step-wise manner or were repeated. In these cases, the Virtual Music Instrument was often still within the previous note that was to be deactivated, and that note naturally was returned as the one closest to the instrument. This, combined with a potentially unlimited number of notes that could be touched simultaneously, created an unreliable touch registration system.

This issue was addressed in three ways. First, the `MusicLayer` was removed in favour of the `OnTriggerEnter` and `OnTriggerExit` methods (Unity Technologies, 'Collider.OnTriggerEnter'; 'Collider.OnTriggerExit'). Utilising `Collider`s rather than a `MusicLayer` proved more reliable and removed the necessity of specifying a layer. Second, the instrument now keeps track of the amount of notes being touched with an `amountTouched` `int`. If this field returns a value greater than one, which means that more than one note is being touched, the instrument runs the distance check and ensures that a `lastTouchedNote` field is kept updated with the note returned by the distance check. The instrument also deactivates other notes not returned by the distance check. This means that the instrument can now only touch a single note. Additionally, the notes store what instrument activates them in an `ActiveController` field. This prohibits notes being activated by more than one instrument. Third, a new distance check was implemented which is invoked in the `OnTriggerEnter` method (when a note

is touched). This new distance check first checks if the `amountTouched` field returns a value greater than one. If so, the distance check compares the `lastTouchedNote` with the note that has just been touched and favours the note furthest from a point just behind the position of the performer. This means that the instrument now prefers notes that are *furthest* from the performer. This matches the nature of the stave, which always has notes that are later further away from the performer.

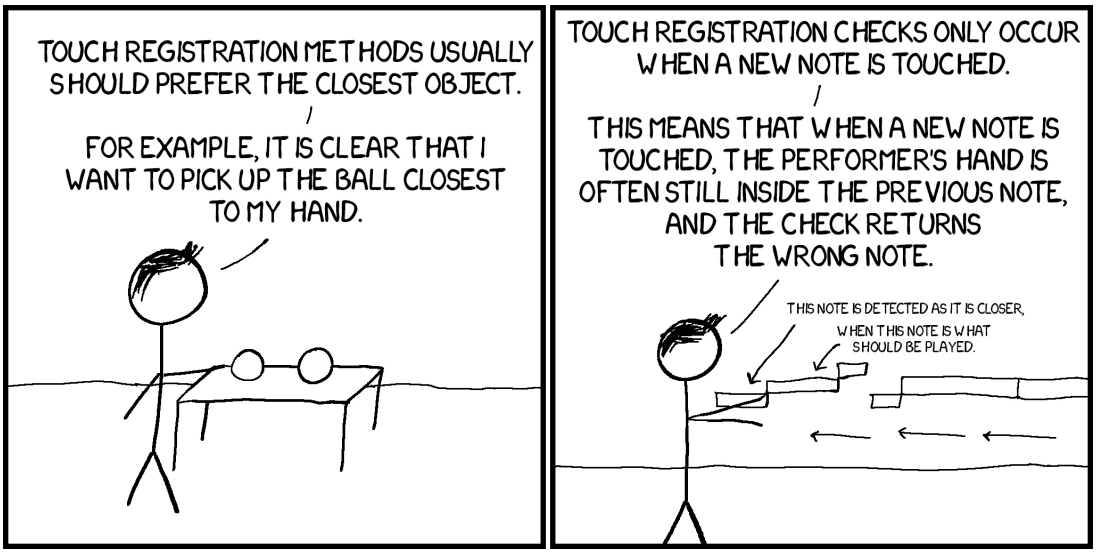


Figure 2.5: Demonstrating the touch registration problem

2.3.3 v0.1.2: Étude 1

The first alterations to the `v0.1.2` release directly addressed feedback from the `v0.1.1` Performance Test. Most notably, the `FromDirection` was reverted back to `Straight`, as the addition of transparency notably improved the clarity of the notation, and as the horizontal approach had adverse effects on the control scheme and the nature of the performance. In the `v0.1.1` Performance Test, I noted:

> The change of axis did not achieve the expected result. The performer would

> naturally rotate their head to face the direction of the note. The performer would then move their hands in a 'dog paddle' fashion, cycling their hands up and down in order to get 'out of the way' of later notes. If the performer strictly maintained a forward-looking position, the pitch of each note appeared more clearly and was more understandable, but this usually was not the case. The addition of transparency, however, was very effective: upcoming notes were clear and this implementation feels better.
>
> Dynamics were less clear. When notes approach the performer from the left, they must move in front of the performer with a 'buffer' of a preconfigured value. This buffer must change in order to ensure that all notes are within range of the performer's reach, but also not too close to them so that the performer themselves is the obstruction. Currently, the side of the note closest to the performer is set to remain the same distance away from the performer, therefore notes with a higher dynamic value will be deeper than ones with a lower dynamic value. The depth of the note can be obstructed, and is also not as intuitive as the width of the note as a means for representing dynamics.
>
> Overall, this implementation is awkward and does not achieve the expected results: performances using this implementation feel worse to perform and are less intuitive. (Appendix D.2)

It was also noted that, since the height of the note was hard-coded, the pitch of a note was not always clear as they could either be too thick (for higher octave divisions), or too small (for lower octave divisions). This led to the implementation of the `GetObjectNoteHeight` method in the `Environment` class (see subsection 3.9.2), and additional methods which gave the composer the ability to override this height if desired. Additionally, it was recorded that the notes moved slowly and were short, which felt 'unusual and unpolished' (Appendix D.2). This led to the implementation of the `TimeScale` field, also in the `Environment` class, which allowed the length of the

notation to be changed while ensuring that the tempo remained the same (see subsection 3.9.3).

The `v0.1.1` Performance Test clearly highlighted that notes often became stuck as active, a result of a persistent and unresolved issue with touch registration. This issue was addressed by improving the use of the `OnTriggerEnter` method, which was made responsible for ensuring that the `CurrentTouchedNotes` `List` remained at a size of `1`, and by changing the point used for the distance check to be the `OVRCameraRig`, which could be used to ascertain the position of the performer's head (see subsection 3.10.4 for further discussion of the final use of the `OVRCameraRig` in Virtual Music). The `v0.1.1` Performance Test also documented that there was no way to pause a performance, especially when a user removed their headset. This led to the use of the `HMDMounted` and `HMDUnmounted` events of the `OVRManager` class—part of the *Oculus Integration* package—to handle the removal of the HMD, and the first basic implementation of the pause mechanism (see subsection 3.6.4).

An ongoing issue documented after the `v0.1.0` release was that the instrument could not be played freely without a piece script that defines the available pitch set. This was because, at this stage, much of the functionality of Virtual Music relied on the use of a `Notation` class, which formed the basis of the piece script. To remedy this, it was necessary to completely refactor the code, since the desired functionality had already outgrown the existing codebase. Learning from the problems of the previous releases allowed me to implement further changes, including refactoring the `Synthesiser` class to allow the procedural generation of audio within *Unity* and the removal of *NAudio*. The `Director` component, which was a singleton using an external dependency, was also changed to the `VMManager` with this release. This was the first step towards the proper segmentation of functionality and the removal of external dependencies in favour of completely internal implementations, which gave me greater flexibility and control.

Allowing Free Play

Before the `v0.1.2` release, Virtual Music could only be used to perform a piece with the use of a piece script. This meant that it was not possible to simply practice or 'free play' with the Virtual Music Instrument without having to interpret the notation. At the time, I had found that the structure of the code was limiting, not very extensible, and required a refactor.

> Refactoring is used to improve the quality of the design whilst retaining the functionality rather that to extend the functionality. It therefore supplements the normal programming activity. (Rumpe 23)

I quickly learned, as computer scientist Bernhard Rumpe states, that 'refactoring and the evolution of functionality are complementary activities that can quickly alternate in the development process' (23). As the methodology dictated that I iterate and increment previous code by adding new features and adjusting where necessary, the codebase quickly and often become unmanageable. This refactor was just the first of many to come.

Through 637 line deletions, 405 additions, and the alteration of 11 files, I used this refactor to segment functionally better, and in doing so, was able to separate functions of the piece script into new or different classes. Most notably, before this release, the `GetFrequency` method of the `Synthesiser` required a piece to be loaded, as the piece would stipulate the current base frequency, amount of octaves, and octave division. This method was moved appropriately to the `Instrument` class and instead used the `Notation` and `Environment` classes to retrieve this information. I also renamed the `Director` class to `VMManager` —short for 'Virtual Music Manager'—to better reflect its purpose, as it was now the class that managed how Virtual Music functioned.

Functions responsible for 'ready' checks, initial setup, and the amalgamation of data—previously a part of the `Notation` class—were moved more appropriately to this class so that the `VMManager` class could be used as a central point of information and control. This allowed the `Notation` class to only give these classes instructions to change these properties when necessary. Changes made during the refactor also allowed the implementation of the pause mechanism, in response to the lack of ability to pause a performance as previously described.

Procedural Audio Generation in Unity

The `OnAudioFilterRead` method of the `MonoBehaviour` class of the `UnityEngine` namespace allows the modification (or replacement) of audio data. To enable the `Synthesiser` to procedurally generate audio, the `Synthesiser` class simply had to inherit the `MonoBehaviour` class, so that it 'implicitly gain[ed] all the members of the base class' (Microsoft, 'Inheritance in C# and .NET').

The structure of the data in the `OnAudioFilterRead` method follows the standard digital waveform data structure: it is a collection (buffer) of samples that are numbers that range from negative one to positive one (Unity Technologies, 'MonoBehaviour.OnAudioFilterRead'). This meant that the existing data could be replaced with data that followed standard computer synthesis mathematics with little or no modification. With this data structure, a sine waveform, for example, could be created by using the inbuilt sine function of many maths libraries, including the `Mathf` class in the `UnityEngine` namespace, which effectively created procedural audio generation within *Unity*. Code Sample 2.4 shows a first unoptimised attempt at using *Unity*'s `OnAudioFilterRead` method to implement procedural audio generation. Notably, this method used a `timeIndex` variable to keep track of the progress of the

sine wave, so it produced artefacts when changing frequency, as noted in the 'were any new issues discovered?' section of the `v0.1.2` Performance Test (Appendix E.2). This is an issue that plagued the synthesiser component until the final stages of development, where particular focus was placed on improving the synthesiser.

```
void OnAudioFilterRead(float[] data, int channels)
{
  for (int i = 0; i < data.Length; i += channels)
  {
    float sample = Amplitude * Mathf.Sin(2 * Mathf.PI * timeIndex * frequency / sampleRate)
    data[i] = sample;
    if (channels == 2) data[i + 1] = sample;

    timeIndex++;

    // if timeIndex gets too big, reset it to 0
    if (timeIndex >= (SampleRate * waveLengthInSeconds))
    {
      timeIndex = 0;
    }
  }
}
```

Code Sample 2.4: A first attempt at procedural audio generation

2.3.4 v0.1.3: Étude 2

As noted in the `v0.1.2` Performance Test (see Appendix E.2), the changes made with the `v0.1.2` release had very positive effects and each change achieved its expected result. This allowed the `v0.1.3` release to focus on quality of life improvements and features to improve extensibility, such as the events system. Focus was also placed on the functionality of notational elements (including pitch, rhythm, and time), rather than just the representation of these elements. This included the addition of dotted notes to extend a note's rhythmic value by half its lettered value, similar in function to rhythmic dots in traditional stave notation. Notably, the first iteration of the events system and the ability to redefine performance elements (such as tempo, fundamental frequency, and the division of the octave) during a performance was introduced via 'Settings Blocks'.

With this release also came the second piece composed for Virtual Music, *Étude 2 (Variation 1)*. In contrast to *Étude 1*, this piece focused on testing the representation of rhythms in Virtual Music Notation. For this purpose, the piece used a short ostinato-like repeating motif that changed rhythmically on each repetition.

Quality of Life Improvements (or not?) and Extensibility

Quality of life is used here to describe improvements made to either increase the ability for developers to use features of Virtual Music, or fix niggling issues that, albeit small, had big impacts on how Virtual Music felt to use. Some quality of life improvements in this release included some further improvements to the touch registration by making the trigger box[11] of the Virtual Music Instrument the same height as the note. This improved touch registration as the vertical space that the instrument used to check for collisions was subsequently limited to a maximum of two different pitches. Other improvements related to extensibility, such as the ability for a developer to implement their own pause mechanism by using the `ControlPause` field of the `VMManager`. This could be used, for example, by a developer so that they could implement their own pause screens, or simply prohibit pausing when the headset was removed—a feature which would be undesirable in ensemble playing (see subsection 3.6.4 for further discussion).

With this release, I became more familiar with C#, *.NET*, and the *Unity* framework. This helped simplify the set-up process. For instance, for the `Note` class, the `RigidBody` and `MeshRenderer` components became required with the `RequireComponent` attribute, which 'automatically add[ed] required components as dependencies' (Unity Technologies, 'RequireComponent'). I also started to use XML comments so that 'Visual Studio and other IDEs can use IntelliSense to show quick

[11] A 'trigger box' refers to an invisible border around an object. It is the point at which other objects will collide with the containing object. It may also be called a 'collision box', or more simply a 'hit box'.

information about types or members' (Microsoft, 'Document your C# code with XML comments'). This improved the accessibility of Virtual Music to developers by describing the purpose of classes and their fields, properties, and methods.

The Events System

In order to improve extensibility, the `v0.1.3` release implemented the first variation of the events system. This system allows developers and composers to subscribe to events related to the notation (such as when a note is activated or deactivated and when a stave is finished), and to 'general' events related to Virtual Music (such as when the application is paused and unpaused). The available events at this time were:

- `General`
  - Paused
  - Unpaused
- `Notes`
  - Activated
  - Reactivated
  - Deactivated
  - Redeactivated
  - Failed
- `Staves`
  - Finished

This events system used multiple 'event manager' classes based on the implementation discussed in the *Create a Simple Messaging System* live-streamed session by *Unity* on 13 April 2015 (Unity). However, this system was adjusted in two main ways. First, multiple instances of the `EventManager` class were created for each type of event, `General`, `Notes`, and `Staves`, which were combined into a container class on the `VMManager`. For this reason, the `EventManager` class was not made a singleton (as recommended in the live-streamed session), as multiple instances of the class needed to exist concurrently. Second, the event manager also included an `EventParameters` `struct`. This `struct` is passed as a parameter to each event, allowing the developer to retrieve information related to the event. These classes and the `EventParameters` `struct` can be seen in Code Sample 2.5.

The same `EventParameter` `struct` was used for each event manager despite the different parameter requirements, as each event would supply values only for the fields it required. This was done to simplify the implementation when testing, and to enable the reuse of the `EventManager` class for each type of event. Ultimately, after multiple trial and error revisions throughout the project's development period, this event system was replaced in the `v0.3.0` release with an entirely different system that used the inbuilt event specification of C# (see subsection 2.5.1).

Improvements to Notational Elements

With the `v0.1.3` release, a composer or developer could provide their own rhythmic length definitions by supplying the `Notation.Generate` method with a `Dictionary` object. This `Dictionary` can be provided with five different elements corresponding to the rhythmic character options in the Virtual Music Shorthand Notation, `s`, `q`, `c`, `m`, and `b` as keys (see section 3.2). For each key, the corresponding rhythmic

```
public class EventsContainer
{
  public EventManager General = new EventManager();
  public EventManager Notes = new EventManager();
  public EventManager Staves = new EventManager();
}

public class EventManager
{
  private Dictionary<string, System.Action<EventParameters>> eventDictionary = new
      Dictionary<string, System.Action<EventParameters>>();

  public void StartListening(string eventName, System.Action<EventParameters> listener)
  {
    System.Action<EventParameters> thisEvent = null;
    if (eventDictionary.TryGetValue(eventName, out thisEvent))
    {
      thisEvent += listener;
      eventDictionary[eventName] = thisEvent;
    }
    else
    {
      thisEvent += listener;
      eventDictionary.Add(eventName, thisEvent);
    }
  }

  public void StopListening(string eventName, System.Action<EventParameters> listener)
  {
    System.Action<EventParameters> thisEvent = null;
    if (eventDictionary.TryGetValue(eventName, out thisEvent))
    {
      thisEvent -= listener;
      eventDictionary[eventName] = thisEvent;
    }
  }

  public void TriggerEvent(string eventName, EventParameters eventParameters = new
      EventParameters())
  {
    System.Action<EventParameters> thisEvent = null;
    if (eventDictionary.TryGetValue(eventName, out thisEvent))
    {
      thisEvent.Invoke(eventParameters);
    }
  }

  public struct EventParameters
  {
    public VirtualMusic.Notation.Note Note;
    public GameObject GameObject;
    public GameObject Stave;
  }
}
```

Code Sample 2.5: The EventManager class and its container

value is provided as a `float` which reflects its value in comparison to a full beat. Similarly, dynamic values can be provided in a similar manner by supplying the `Notation.Generate` method with a `Dictionary` object containing the available dynamic indications and the value that they would be interpreted as. Default values are supplied in the `GenerationDefaults` class, as seen in Code Sample 2.6. This feature was added in an attempt to make the differences in dynamic values more prominent, particularly in the synthesiser (an ongoing issue that had not yet been closed), but was implemented in an extensible way so that they could be overridden if desired. This system formed the basis of the functionality in the final release, `v0.4.1`, which allows a composer or developer to supply rhythmic duration and dynamic level overrides in a similar manner (see subsection 3.3.7 and subsection 3.3.6, respectively).

```
/// <summary>
/// Defaults for using <see cref="Notation.Generate()."/>
/// </summary>
public static class GenerationDefaults
{
  public static Dictionary<char, float> NoteLengths = new Dictionary<char, float>
  {
    { 's', 0.25f },
    { 'q', 0.5f },
    { 'c', 1f },
    { 'm', 2f },
    { 'b', 4f }
  };

  public static Dictionary<int, float> NoteDynamics = new Dictionary<int, float>
  {
    { 1, 0.03125f },
    { 2, 0.0625f },
    { 3, 0.125f },
    { 4, 0.25f },
    { 5, 0.5f },
    { 6, 0.75f },
    { 7, 1f },
    { 8, 1.1f },
    { 9, 1.2f }
  };
}
```

Code Sample 2.6: The default note lengths and dynamics

The stave was also made to follow the position of the performer in this release. If a stave was approaching the performer frontally, it could move horizontally to remain within reach of the performer even if they moved. Interestingly, this change brought attention to

issues with a 'rhythmic zero point', as discussed in the Performance Test for this release (see Appendix F.2). This is discussed further in subsection 2.3.5 and subsection 3.10.2.

Settings Blocks

Before the `v0.1.3` release, the tempo, octave division, amount of octaves, and fundamental frequency of a piece could only be defined before the piece was generated. Additionally, it was not possible to ascertain the tonality of the piece by looking at the Virtual Music Shorthand Notation alone, and similarly, it was not possible to define these elements by using Virtual Music Shorthand Notation. The 'Settings Block' was introduced with this release, which allowed for the definition of these features to happen at the beginning of the piece. This Settings Block could be defined in the Virtual Music Shorthand Notation with the inclusion of a key-value pair surrounded by square brackets. While the functionality of Settings Blocks improved throughout the project's lifetime (for example with the addition of the ability to notate a Settings Block at multiple points throughout a piece in `v0.1.4`), they barely differed in the final release. Discussion of Settings Blocks and their use can be found in subsection 3.2.2.

2.3.5 v0.1.4: Étude 2

Most changes of the `v0.1.3` release achieved positive results: touch registration was again improved, the events system functioned well enough to serve its purpose, and the quality of life improvements had me more and more excited to use and work on Virtual Music each day. The addition of rhythmic dots also increased the ability for composers to notate more naturally, and I found myself thinking of new ideas each day,

just as I would for composition traditionally. Now, however, I was also considering how Virtual Music could be used in order to create new effects. As I started likening composition in Virtual Music to traditional compositional practices, I found that there were some key functionalities missing. Most notably, it was not possible to notate crescendos and diminuendos, triplets (see subsubsection 3.2.3), and tempo changes during a piece (because allowing Settings Blocks could not be notated at any point in a piece). Therefore, one focus point of the `v0.1.4` release was to add these features to further liken the notational ability of Virtual Music Notation to that of traditional stave notation.

This release also showed the first example of the Iterative and Incremental SDLC method (see subsection 1.7.4) used on a composition. With the addition of crescendos and diminuendos (called 'hairpins') and triplets, *Étude 2 (Variation 2)* varied the original piece by including the newly available hairpins and triplets.

Hairpins and Procedurally Generating Note Meshes

The addition of crescendos and diminuendos required a completely new system of note generation which did not use a supplied `GameObject` and instead used inbuilt mesh generation. This new generation system was required so that the width of a note could be different at either end (indicating a gradual change in dynamic level), as the existing method could only scale objects uniformly. The option to supply a `GameObject` was nonetheless retained to increase customisation options, at the cost of this reduced functionality[12] (see subsubsection 3.3.8).

This was a significant change that took a very large portion of the development time for

[12] A warning is displayed when the developer provides a custom `GameObject` so that they are aware of this trade-off.

this release, as the mesh generation system involves knowledge of how *Unity* constructs the mesh and interprets normal vectors, of which I had neither. However, the implementation of this generation system meant that I and other developers and composers could then have full control over the appearance of the notation. It also allowed the notation to more clearly show additional features in future releases, such as octave differences in `v0.2.0` (see subsection 2.4.1) and articulations in `v0.3.0` (see subsection 2.5.1). Additionally, generating meshes internally fixed the errors related to the world unit size problem. This problem affects the scale of objects created with external applications such as *Blender* (Blender Foundation), which would appear larger or smaller than intended as the units of measurement between applications are not equal by default. The addition of the internal mesh generation system also meant that the need to supply a default object was thus removed, as noted in the `v0.1.3` Performance Test:

> It was also noted that the practice of supplying an object to be scaled according to the dynamic level and rhythmic duration of the note could cause issues, including incorrect world unit size, increased risk of obscuring the intention of the notation, and unwanted space between notes. The latter especially was noted during this Performance Test, in which the supplied object caused a visual separation between each note which was interpreted as intended separation similar to a mezzo staccato. Although difficult, one solution to this would be to generate the mesh for the notes in-code. This has the added benefit that custom note shapes may be added in the future, perhaps for articulations or similar. (Appendix F.2)

To generate meshes in *Unity*, a developer must supply a `Mesh` to the `MeshFilter` component of a `GameObject`. But what is a `Mesh`?

A mesh consists of triangles arranged in 3D space to create the impression of

> a solid object. A triangle is defined by its three corner points or vertices. In the Mesh class, the vertices are all stored in a single array and each triangle is specified using three integers that correspond to indices of the vertex array. (Unity Technologies, 'Anatomy of a Mesh')

While a shape can be defined only by these triangles, additional information is required in order for *Unity* to correctly shade and light the `Mesh`. Fortunately, this can mostly be achieved with the `RecalculateNormals` method of the `Mesh` class, but care must be taken to ensure that the three integers are specified in the correct order: 'the ordering is important because you must order the corners clockwise' (Unity Technologies, 'Example - Creating a quad').

A new `ShapeType` class was created to support the shape being overridden (note that this class name was not final. See subsubsection 3.3.8 for more information). This class could then be passed parameters on creation which indicated the desired height, length, and importantly, width at either side.

To better understand the way that the triangles must be formed, I took the closest square object on my desk near me (a portable Bluetooth speaker) and simply looked at each face. Each corner of the square could be assigned a number by following a 'Z' shaped pattern on each face, starting from the front face, then the back, left, right, top, and bottom. These points were stored in a `Dictionary`, which was then used to retrieve them by a string which described the position of that point on the cube. Finally, an array of integers was created that refer to the indexes of the `Vertices` array in groups of three, as seen in Code Sample 2.7.

```
1 float front = 0f;
2 float back = length;
3 float backLeft = -(endingWidth / 2);
4 float backRight = endingWidth / 2;
5 float frontLeft = -(startingWidth / 2);
6 float frontRight = startingWidth / 2;
7 float top = height / 2;
```

```
float bottom = -(height / 2);

Dictionary<string, Vector3> VertexPositions = new Dictionary<string, Vector3>() {
  { "left top back", new Vector3(backLeft, top, back) },
  { "left top front", new Vector3(frontLeft, top, front) },
  { "left bottom back", new Vector3(backLeft, bottom, back) },
  { "left bottom front", new Vector3(frontLeft, bottom, front) },
  { "right top back", new Vector3(backRight, top, back) },
  { "right top front", new Vector3(frontRight, top, front) },
  { "right bottom back", new Vector3(backRight, bottom, back) },
  { "right bottom front", new Vector3(frontRight, bottom, front) },
};

Vertices = new Vector3[] {
  // front face
  VertexPositions["left bottom front"],
  VertexPositions["left top front"],
  VertexPositions["right bottom front"],
  VertexPositions["right top front"],

  // back face
  VertexPositions["right bottom back"],
  VertexPositions["right top back"],
  VertexPositions["left bottom back"],
  VertexPositions["left top back"],

  // left face
  VertexPositions["left bottom back"],
  VertexPositions["left top back"],
  VertexPositions["left bottom front"],
  VertexPositions["left top front"],

  // right face
  VertexPositions["right bottom front"],
  VertexPositions["right top front"],
  VertexPositions["right bottom back"],
  VertexPositions["right top back"],

  // top face
  VertexPositions["left top front"],
  VertexPositions["left top back"],
  VertexPositions["right top front"],
  VertexPositions["right top back"],

  // bottom face
  VertexPositions["left bottom back"],
  VertexPositions["left bottom front"],
  VertexPositions["right bottom back"],
  VertexPositions["right bottom front"]
};

Triangles = new int[] {
  // front face
  0, 1, 2,
  1, 3, 2,

  // back face
  4, 5, 6,
  5, 7, 6,

  // left face
  8, 9, 10,
  9, 11, 10,

  // right face
  12, 13, 14,
  13, 15, 14,

  // top face
  16, 17, 18,
  17, 19, 18,

```

```
  // bottom face
  20, 21, 22,
  21, 23, 22
};
```

Code Sample 2.7: An excerpt of the ShapeType class showing the VertexPositions Dictionary referencing all corners of a square

This `ShapeType` class is then used during the `Start` method of the `Note` class to supply the `MeshFilter` with its required `vertices` and `triangles` parameters, generating the `Mesh`. The `ShapeType` `Vertices` and `Triangles` properties are also used to create a new `MeshCollider`, ensuring that the note's collision detection is precise. The construction of a note using these classes can be seen in Code Sample 2.8.

```
if (!VMManager.Notation.SupplyCustomNoteObject)
{
  MeshFilter meshFilter = gameObject.AddComponent<MeshFilter>();
  meshFilter.mesh.Clear();
  meshFilter.mesh.vertices = Shape.Vertices;
  meshFilter.mesh.triangles = Shape.Triangles;
  meshFilter.mesh.RecalculateNormals();

  Mesh mesh = new Mesh
  {
    vertices = Shape.Vertices,
    triangles = Shape.Triangles
  };

  MeshCollider meshCollider = gameObject.AddComponent<MeshCollider>();
  meshCollider.convex = false;
  meshCollider.sharedMesh = mesh;
  meshCollider.sharedMesh.RecalculateNormals();
}
```

Code Sample 2.8: Constructing a note using the ShapeType class

Further Improving the Notation's Clarity

The clarity of the notation was also a focus of this release, due to my suggestions in the `v0.1.3` Performance Test the current transparency levels needed to be further adjusted. These suggestions led to the reduction of the default note opacity to 20 per cent when not

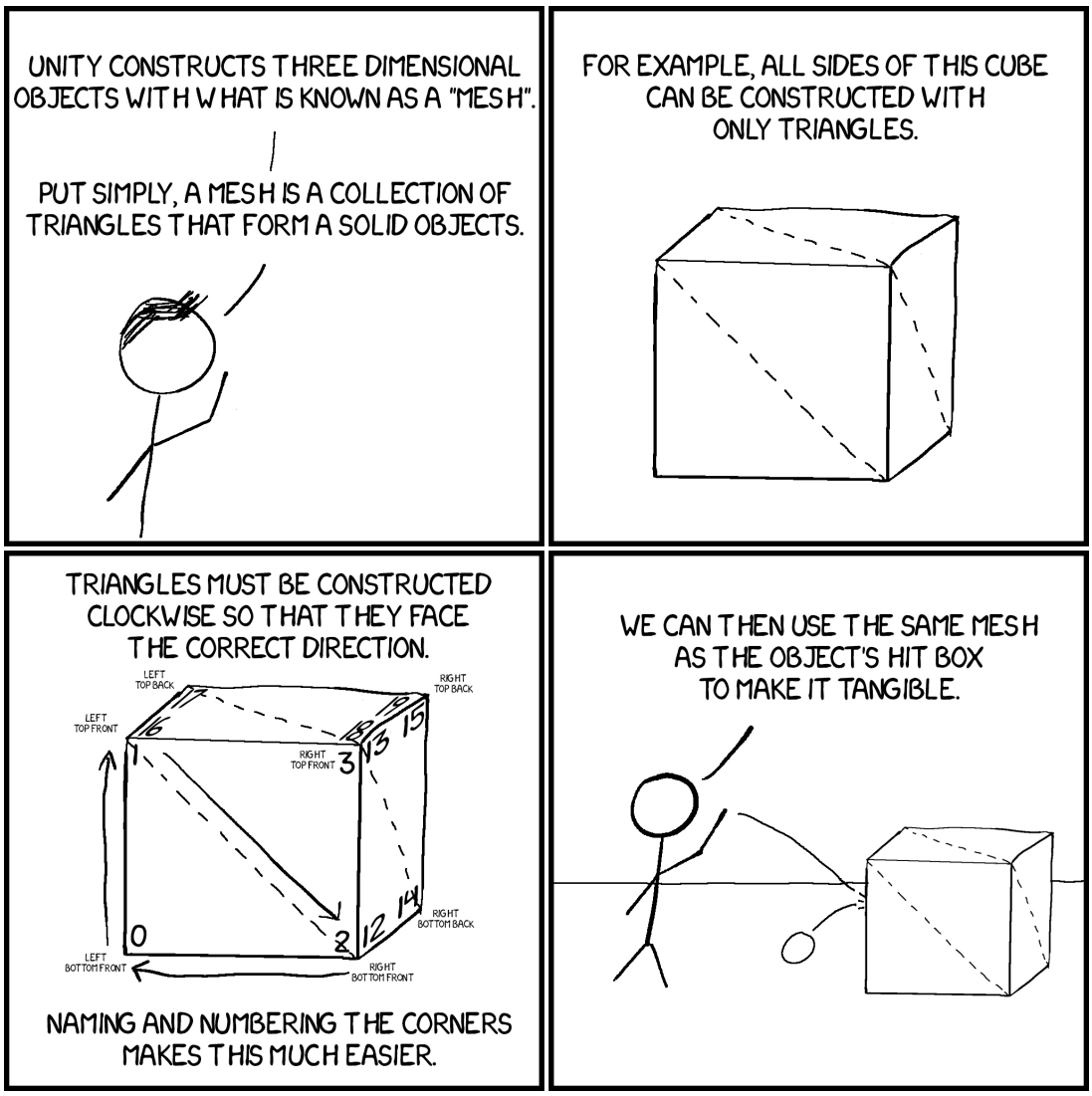


Figure 2.6: Constructing shapes in Unity

activated and 50 per cent when activated. Light emission was also added so that notes could be seen in dark environments and did not rely on other in-engine light sources such as spot lights or global illumination.

The ability to add particle effects to activated notes was also added via custom note properties. This increased customisation options, making it possible to create unique visual flair (see subsubsection 3.3.8 and subsection 3.3.9). The Performance Test also noted that while the note height was dynamic and calculated according to the current environment, the dynamic value was still hard-coded. This comment led to the introduction of the `NoteWidthModifier` field, a multiplier set to `0.8f` by default. This reduced the visual indication of dynamic values so that they appeared more natural. When combined with the ability to supply custom dynamic values to a piece (now correctly limited to `0f` and `1f` to prevent clipping in the synthesiser), this gave the composer greater customisation options for the appearance of dynamics.

Object-oriented Programming Principles

Finally, in this release the codebase for Virtual Music started to move towards more standard object-oriented programming principles. This was seen with the separation of classes and the introduction of inheritance, allowing developers to create new classes which automatically inherit 'any properties and methods defined by the [original] class' (Stephens 109). For example, the `Synthesiser` class was rewritten so that it could be inherited. This greatly simplified the process to create new synthesisers, as only the oscillator had to be overridden. Classes that inherited the `Synthesiser` class could override the `GetWaveform` method to allow the combination of multiple oscillators. This meant that a developer with knowledge of synthesiser programming could implement any type of synthesis. Three examples are given in Code Sample 2.9: a basic sine wave, a

synthesiser that combines a sine wave and a triangle wave, and finally an example of frequency modulation synthesis.

Other examples of the separation of functionality included the newly introduced `Performer` class (which handled the performer and their preferences), the `Stave`, and the `StaveContainer` classes. The latter was created to directly address an issue from the `v0.1.3` Performance Test:

> The change made to mimic the horizontal position of the the performer highlighted that it is possible for the performer to simply walk forward, changing the point where the rhythmic beat lies. To combat this, it is necessary to define a point which is considered the rhythmic 'zero' point. This point should then follow the performer.
>
> However, it is important to note that when performing with `Virtual Music`, the performer will generally hold their hands out in front of them. This means that making the performer themselves the rhythmic zero point makes little sense, as the performer is continually 'anticipating' the beat. It would be best to use the position of the performer's hand (likely their preferred hand) as the rhythmic zero point. This will still allow them to use their opposite hand to anticipate the beat if they wish to do so. (Appendix F.2)

The performer's dominant or preferred hand can be set via the `DominantHand` field of the newly created `Performer` class. This field, of the `InstrumentType` type, simply defines which hand the performer prefers and which hand therefore is used as the position where the beat will lie. This means that, if the performer quickly moves their hand forward and backwards, the stave appears to move forward and back in time. However, the stave is actually moving to ensure that time is not altered by the performer anticipating the note when they do not intend to.

```
/// <summary>
/// A basic sine wave synthesiser.
/// </summary>
public class Sine : Synthesiser
{
  public override float Oscillator(float frequency, float amplitude, int time, float
      sampleRate)
  {
    return Mathf.Sin(2 * Mathf.PI * frequency * time / sampleRate) * amplitude;
  }
}

/// <summary>
/// A synthesiser that combines a sine wave and a triangle wave.
/// </summary>
public class SineAndTriangle : Synthesiser
{
  public override float Oscillator(float frequency, float amplitude, int time, float
      sampleRate)
  {
    return Mathf.Sin(2 * Mathf.PI * frequency * time / sampleRate) * amplitude;
  }

  public override float Oscillator2(float frequency, float amplitude, int time, float
      sampleRate)
  {
    return (1f - 4f * Mathf.Abs(Mathf.Round(frequency * time / sampleRate - 0.25f) -
        (frequency * time / sampleRate - 0.25f))) * amplitude;
  }

  public override float GetWaveform()
  {
    // level range -1 to 1
    // eg. -1 = first oscillator 100%, second 0%
    float level = -0.75f;
    (Oscillator(Frequency, Mathf.Sqrt(0.5f * (1f - level)), timeIndex, SampleRate) +
        Oscillator2(Frequency, Mathf.Sqrt(0.5f * (1f + level)), timeIndex, SampleRate)) *
        Amplitude;
  }
}

/// <summary>
/// An example of frequency modulation synthesis using a sine wave.
/// </summary>
public class FMSine : Synthesiser
{
  public override float GetWaveform()
  {
    float fmFrequency = Frequency + Oscillator(2f, 1f, timeIndex, SampleRate);
    return Oscillator(fmFrequency, Amplitude, timeIndex, SampleRate);
  }
}
```

Code Sample 2.9: Overriding the new Synthesiser class with object-oriented principles

2.4 Polyphonic Ability and Timbral Alteration

2.4.1 v0.2.0: Étude 3

Besides another comment about not being able to perform hairpins in the Simple Performance Mode, all changes from the `v0.1.4` again resulted in their expected outcomes. The clarity and representation of Virtual Music Notation had reached a point at which Virtual Music was ready to move onto the next development stage: implementing polyphony and timbral alteration. Some development time could also be spent on improving the feature-set of Virtual Music, improving how it felt to use and perform with Virtual Music.

The `v0.2.0` release saw the first iteration of polyphony, where polyphony is the ability to notate more than one note concurrently without using a second stave. Additionally, a number of features were also introduced in the interest of music preservation, customisation, to give both the composer and performer better creative options, and finally ease of use. For the latter especially, the further implementation of object-oriented programming principles and a newly introduced `Piece` class allowed the use and composition of a piece without the use of any code.

The `v0.2.0` release also saw another instance of the Iterative and Incremental SDLC (see subsection 1.7.4) being applied to the compositional method: *Étude 1* had been re-imagined to make use of new features such as hairpins, and the polyphonic features

introduced as part of the `v0.2.0` release including octave alterations[13]. The incremental additions to *Étude 1* altered the original piece to an extent that the new version felt like a new piece entirely. *Étude 3* is therefore named accordingly.

Polyphony

The first iteration of polyphony in VIRTUAL MUSIC in the `v0.2.0` release was rhythmically locked to the note that it accompanied. The `v0.2.0` Release Notes give a good summary of how this first iteration of polyphony functioned:

> - Polyphony! 🎵 (#25) (`c74d2c5`)
>   - Polyphony works in a similar fashion to 'double stops' on violin.
>   - Second notes are currently limited to the same rhythm as the original note it is accompanying.
>   - Second notes can be in altered octaves and will appear differently when their octave has been altered (#26) (`87709e8` , `9f32099`)
>
> (Appendix H.1)

In order to support polyphonic notes, support for their indication was needed in VIRTUAL MUSIC SHORTHAND NOTATION. In code, VIRTUAL MUSIC SHORTHAND NOTATION was parsed by the `Regex` class—the *.NET Framework*'s regular expression engine—using a regular expression pattern, a 'pattern that the regular expression engine attempts to match in input text' (Microsoft, 'Regular Expression Language - Quick Reference'). The `Regex` class parses the incoming VIRTUAL MUSIC SHORTHAND

[13]Note that, at this time, octave alterations were only available on second, polyphonic notes, so was considered a polyphonic feature. This would be changed in the `v0.2.1` release (see subsection 2.4.2).

Notation staves (the 'input text') and generates new notes and Settings Blocks accordingly.

In order to support polyphony in Virtual Music Shorthand Notation, the existing pattern had to be extended to include a new polyphonic identification pattern. The extended pattern must clearly separate the additional pitch so that both the composer and the computer may process them accordingly while not interfering with existing methods to notate other notational elements. For example, the addition of the polyphonic note with a dash character (`-`) may interfere with the existing method to notate ties, which are also notated with a dash.

The final pattern to notate polyphony took inspiration from software development; programming languages such as C# often denote additional information that should be processed differently (parameters, array indexes, and additional code within a function, for example) through three different brackets: curved brackets (`()`), square brackets (`[]`), and and curly brackets (`{}`). Using brackets would also conveniently encompass the information so that the composer could look at the Virtual Music Shorthand Notation and see clearly when polyphonic tones are used. Since square brackets were already used to indicate Settings Blocks, curly brackets were chosen.

Since tone creation is done with the hands and most performers would have a maximum of two, it would only be necessary to support the indication of one additional concurrent pitch. This fortunately reduced the complexity significantly, and meant that the notation of an additional pitch could be done as simply as an additional pitch degree within the curly bracket. For example, `{8}` , would indicate an additional polyphonic note at the eighth pitch degree. Limiting additional pitches to the same rhythmic length at this stage would also help test practice of notating polyphony with curly brackets. Additionally, it simplified the `Regex` pattern, which was highly unoptimised and was quickly becoming complicated and confusing. As the development time of this release was already longer

than expected, this simpler method was preferred. This pattern as of the `v0.2.0` release can be seen in Code Sample 2.10[14].

```
var match = Regex.Match(notes,
    "([0-9]+)([a-z](\\.)?(\\[3\\])?)(-[a-z](\\.)?(\\[3\\])?)?([0-9])?([<>][0-9])?" +
"({[0-9]+(v+|\\^+)?})?");
```

Code Sample 2.10: The Regex pattern to generate notes as of v0.2.0

While refining *Étude 1*, I often wanted to include additional pitches in different octaves so that these additional pitches could function as a sort of harmony or bass. This was achieved with the addition of the octave alteration characters (see subsubsection 3.2.3) and the new `OctaveDifference` field on the `ShorthandNote` class. The characters `^`and `v` were chosen simply for their appearance being similar to arrows, visually indicating that the note should be risen or lowered an octave, respectively. The number of times the octave alteration character was notated indicated the number of octaves that pitch would be raised or lowered.

The `OctaveDifference` field stored the number of octaves altered as a whole number. If the octave was unaltered, it would equal `0`. Similarly, for notes raised a single octave, it would equal `1`, and for notes lowered two octaves, it would equal `-2`. The 'true' frequency of any note, including those with octave differences, could then be calculated simply by multiplying the written pitch degree's frequency by 2^d, where d is equal to the difference. This can be seen in Equation 2.3, where f is equal to the original frequency, d is equal to the octave difference, and t is equal to the true frequency. An example of this equation is provided in Equation 2.4, which shows the working for `1vv`. For this polyphonic note, the first pitch degree, which when using the default fundamental frequency setting is equal to 261.6256, would be lowered by two octaves.

[14]Note that, amusingly, the line break separating the polyphony section was only added in this document so that the code could fit on the page

$$f \times 2^d = t$$

Equation 2.3: Calculating the note's true frequency including any octave differences

$$261.6256 \times 2^{-2} = 65.4064$$

Equation 2.4: Lowering the first pitch degree by two octaves

To show that generated notes had an altered octave when played, two additional note shapes (now called 'note forms') were introduced: a triangle with an upward facing point to show that the octave of the note had been raised, and a triangle with a downward facing point to show that the octave of the note had been lowered. These used the same mesh generation mechanism as implemented in `v0.1.4`, however each placed their vertices differently with different values in the `VertexPositions` `Dictionary`. These shapes were intended to show the performer that the octave of the note had changed from what was indicated by its height, but did not indicate the number of octaves of which the note had changed.

The Piece Class

Following the developmental focus of previous versions refactoring current code to follow object-oriented programming principles, the `v0.2.0` release replaced the confusingly titled `Notation` class—which served as a monolithic container for information related to a piece—with multiple different classes in the `Notation` namespace. The function of each class within the `Notation` namespace was related to the representation or functionality of Virtual Music Notation, and this functionality was supported by the refactored `Environment` class, which now simply served as a method of ascertaining and maintaining the current performance environment. This included `Stave`, `Note`, `NoteForm` (previously `ShapeType`), `SettingsBlock`, and finally, `Piece`.

While dividing these notational elements into separate classes made it easier to further iterate on their functionality, the introduction of the `Piece` class also enabled the preservation of music written for VIRTUAL MUSIC, as instances of the `Piece` class could now be saved to a file. The `Piece` class also stored music preservation data such as the title, composer, and date of composition in the `Title`, `Composer`, and `DateOfComposition` fields, respectively. These fields would then be displayed at the start of each piece on a newly created 'Information Slide' (see subsection 3.3.3).

The separation of functionality also helped remove the requirement of code knowledge to set up a scene in *Unity* using VIRTUAL MUSIC, as to implement a piece within a scene it was now only necessary to add a `Piece` component to a `GameObject` with the friendlier GUI. The *Inspector* panel could then be used to insert the VIRTUAL MUSIC SHORTHAND NOTATION as well as set any materials or other basic customisation options.

This functionality was only possible with two additional fields added to the `Piece` class: `GenerateOnStart`, and `AutoPlayOnSinglePiece`. These fields were added so that when starting the scene, VIRTUAL MUSIC could generate the piece and begin the performance automatically, removing the necessity of calling the `Play` method in a script and reducing the risk of the `Play` method being called incorrectly. However, if `GenerateOnStart` was set to `true` the ability to add advanced custom functionality that must be added via a script was removed, as the piece was generated before such functionality could be added. This included the addition of custom components to the note via the `CustomNoteComponents` field, and custom note dynamics and lengths via the `GenerationSettings` field. Advanced customisation options such as these could still be added by setting the `GenerateOnStart` field `false` and then calling the `Generate` method manually.

Increasing Customisation and Creative Options

A number of features were added in this release in the interest of increasing customisation and creative options to the developer, composer, and performer. These included features to improve how it felt to use and perform with VIRTUAL MUSIC, which also allowed composers, developers, and performers to compose and perform more naturally. Examples of these features include the ability to adjust the tempo with the thumbstick of the controller (see subsubsection 3.7.1), and the addition of vibration on note activation to provide haptic feedback to the performer (see subsubsection 3.7.1 and subsubsection 3.7.1 for a discussion on how and when the controller will vibrate). New events were also added to improve the ability for developers and composers to implement custom functionality, including:

- `Piece`
  - `Generated` (`1721ad7`)
  - `Finished` (`1721ad7`)
  - `Removed` (`f98fb34`)
- `Stave`
  - `Generated` (`38e6b29`)
- `Note`
  - `Generated` (`38e6b29`)

The concept of a Performance Mode was improved in this release, building upon the iterations to the control schemes of previous releases with the addition of an Intermediate Performance Mode. The Performance Mode was the direct answer to the ‘virtuosity

ceiling' as described by Wessel and Wright, who state that the 'early stage ease-of-use [of a computer-based instrument] should not stand in the way of the continued development of musical express[eness]' (11).

The early stage ease-of-use for the Virtual Music Instrument comes in the form of the Simple Performance Mode: the instrument is easy to pick up for new users while still being capable of performing the notated music exactly as indicated. However, once a performer becomes familiar with this simpler mode of operation, they may chose to advance to a mode with higher degree of potential musical expressiveness. A performer's pathway to the Advanced Performance Mode—being a completely free mode of operation not limited by the notation—would be through the Intermediate Performance Mode. This mode offers greater musical expressiveness through freedom of interpretation for dynamics, while limiting freedom of interpretation of pitch, which is the most difficult aspect of tone creation with the Virtual Music Instrument as the hands must stay perfectly balanced.

In previous releases, the Advanced and Simple Performance Modes were representative of the `FreeMode` `bool` being `true` or `false`, respectively. As the `FreeMode` was simply augmenting the functionality of the original control scheme, there was no functionality in place to create additional modes outside of this, and the terminology was confusing. The `v0.2.0` release built upon the foundation laid in the `v0.1.4` release to refactor the adjustable control schemes into the three different performance modes. The Simple Performance Mode functioned as previously: tone generation occurred only when a note was touched and the pitch and dynamic level of the tone was automatic with no required button presses. The Advanced Performance Mode also functioned as previously: tone generation only occurred on button presses and all aspects of the tone generation are completely free, including pitch, dynamics, and rhythm, which are not impacted by the notation. Finally, the newly introduced Intermediate Performance Mode was an in-between: tone generation only occurred when

a note was touched and when the tone generation button was also being pressed. Pitch was taken from the note, but dynamics were freely controlled by the performer with the same control scheme as with the Advanced Performance Mode.

The last notable improvements in the `v0.2.0` release were related to Settings Blocks, which were further improved in three ways. First, numeric tempo indications were improved by adding the text 'BPM' after the number to indicate the tempo was in beats per minute. Second, text could be supplied to a tempo marking without a numeric tempo indication, allowing the composer to write short messages or indicate style. Finally, such text could also be supplied with underscores (`_`) to indicate spaces. These underscores would be converted to spaces when the Settings Block was generated.

2.4.2 v0.2.1: Étude 4

The changes implemented in the `v0.2.0` release again had mostly positive results. By this point of development, I had found that I was able to more effectively preempt any bugs during implementation, and since the development was now more focused on the addition or improvement of specific features, more testing could occur before the release. This was not always true, of course, and oftentimes the adjustments or the implementation of new functionality did not have any major code bugs but instead had unexpected issues in their functionality. This is evident in the newly introduced note shapes, as described in the `v0.2.0` Performance Test (see Appendix H.2):

> I felt, however, that the shape of notes that had their octave altered did not fit very well with how everything else looks. While everything is square edged, I felt that the 'points' on the outside edges of the notes, where the inside angle was less than 90 degrees, were too sharp and felt out of place. Additionally,

> as the mesh would become very thin on the outer edges, it also contributed to the dynamic indication and therefore the note being unclear. I would prefer that octave changes had a pentagonal or 'house-like' shape, where the triangle is added to the upper or lower half of the note's usual rectangular shape. (Appendix H.2)

Improving the Note Shapes

The improved 'house-like' octave difference shapes were constructed (somewhat) simply by making the standard note shape half the height, and then replacing either the top or bottom face with four additional faces forming a point: a triangle on either side of the note, and then four triangles making two rectangles, bridging the gap between them. This new shape still showed the direction that the octave had been altered in, but removed the sharp points at the edges of the triangle which were less than 45 degrees and could obscure details of the note by causing the edges to become thin and unclear. Ensuring a 90 degree angle was always present on each side of the note made the edges of the note considerably clearer. Figure 2.7 shows the improved house-like shape for a note that has its frequency altered up an octave.

While adjusting these shapes, the classes that inherited the `NoteForm` class were finally changed to classes implementing the new `INoteShape interface`. These classes would be then stored in a class derived from the `INoteShapes interface`. This change was the final change in class name and structure of the note shape, and was done as the note shapes were a perfect example of `interface`s: classes implementing the `interface` must provide an implementation of the members defined in the interface, but the manner in which these were implemented were not important to how Virtual Music functioned (Microsoft, 'interface'). This was done because while the method I was using to construct

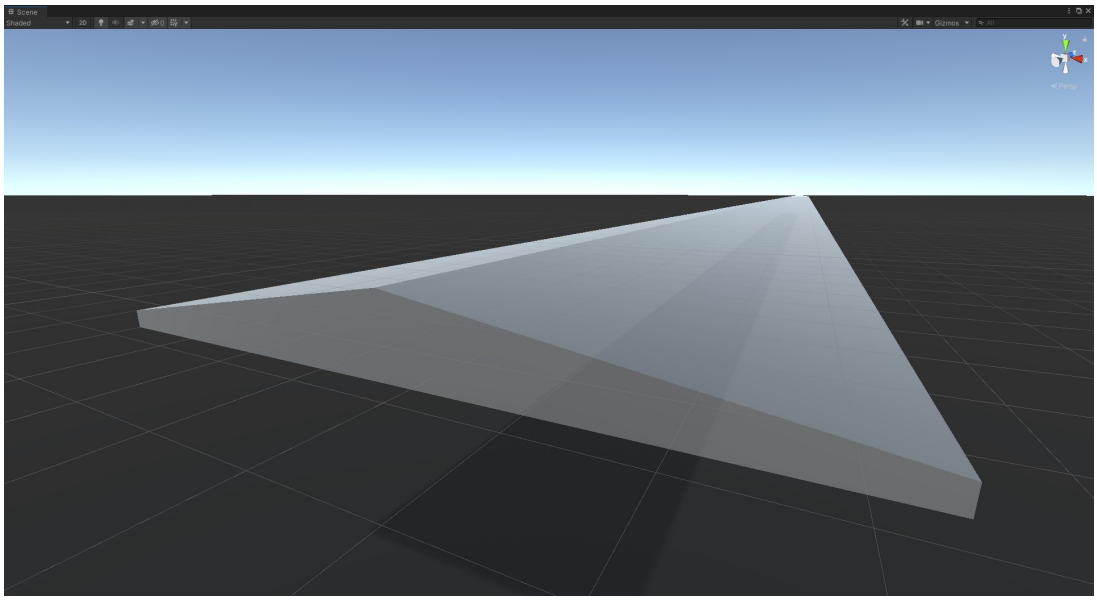


Figure 2.7: The house-like octave up shape

the note shapes suited the way I thought about them, it would not be suitable for every user, especially those with a better understanding of mesh generation. Developers with a better understanding of these concepts would only have to supply values to the `Triangles` and `Vertices` arrays.

Improving Polyphony

Octave alterations were made available to standard single notes and additionally polyphonic notes were made to be completely independent. This was done by the refactor of the `ShortHandNote` object, which now had `static` methods to create a new object of the type. Additionally, the previously mentioned `Regex` pattern as seen in Code Sample 2.10 was refactored to include named matched subexpressions. These new patterns, seen in Code Sample 2.11 were considerably more readable and meant that the subexpressions could then be accessed through the `Group` field of the `Regex.Match` object. For example, the pitch degree could be retrieved simply by the string 'pitch_degree': `match.Groups["pitch_degree"].Value`.

```
private const string FirstNoteRegexPattern =
  "^(?<issecondary>!)?" +
  "(?<pitch>" +
  "(?<pitch_degree>[0-9]+)" +
  "(?<pitch_octavealteration>v+|\\^+)?" +
  ")" +

  "(?<time>" +
  "(?<time_length>[sqcmb])" +
  "(?<time_rhythmicdot>\\.)?" +
  "(?<time_tuplet>\\[3\\])?" +
  "(?<time_tie>-" +
  "(?<time_tie_length>[a-z])" +
  "(?<time_tie_rhythmicdot>\\.)?" +
  "(?<time_tie_tuplet>\\[3\\])?" +
  ")?" +
  ")" +

  "(?<dynamic_degree>[0-9])?" +

  "(?<dynamic_hairpin>[<>]" +
  "(?<dynamic_hairpin_degree>[0-9])" +
  ")?$";

private const string SecondNoteRegexPattern =
  "^(?<issecondary>!)?" +
  "(?<pitch>" +
  "(?<pitch_degree>[0-9]+)" +
  "(?<pitch_octavealteration>v+|\\^+)?" +
  ")" +

  "(?<time>" +
  "(?<time_length>[sqcmb])" +
  "(?<time_rhythmicdot>\\.)?" +
  "(?<time_tuplet>\\[3\\])?" +
  "(?<time_tie>-" +
  "(?<time_tie_length>[a-z])" +
  "(?<time_tie_rhythmicdot>\\.)?" +
  "(?<time_tie_tuplet>\\[3\\])?" +
  ")?" +
  ")?" +

  "(?<dynamic_degree>[0-9])?" +

  "(?<dynamic_hairpin>[<>]" +
  "(?<dynamic_hairpin_degree>[0-9])" +
  ")?$";
```

Code Sample 2.11: The new Regex pattern for the first and second notes

This newly found independence between singular and polyphonic notes allowed interesting functionality; by indicating a rest as the primary note (the note that the polyphonic note was accompanying), a composer could notate arpeggiations, and the rest indicated the rhythmic duration before the next note of the arpeggiation. This additionally inspired the ability to sustain notes with the introduction of the `SustainSynthesiserButton` and the `SustainedNoteOrTone` class, a rather problematic class replaced in `v0.4.1` (see subsection 2.6.2) that maintained the status of the note or tone that was being sustained but limited the number of tones able to be sustained to two per synthesiser type.

The Secondary Note and Synthesiser Types

Now that the 'polyphonic ability' part of 'Polyphonic Ability and Timbral Alteration' had been implemented to some successful degree, it was necessary to tackle the 'timbral alteration' part. This was done with the addition of a secondary synthesiser type, giving the composer and performer access to two different types of sound: the 'Primary Synthesiser' and the 'Secondary Synthesiser'.

So that a composer could indicate the use of the secondary synthesiser, it was first necessary to define a method to notate the secondary note type in Virtual Music Shorthand Notation. Thanks to the recent refactor of the `Regex` pattern, this was achieved quite simply with the addition of a `issecondary` named group at the start of the pattern Code Sample 2.11. This named group checked for the inclusion of a single exclamation mark (`!`) at the beginning of a note's definition and would override the note's type as secondary accordingly. If a note's definition did not have this indication, it would be primary.

It was then necessary to add fields to the `Note`, `Synthesiser`, and `ShortHandNote` class which reflected which synthesiser type that note or synthesiser was or would activate. A second activatable synthesiser was also added to the `Instrument` class along with the definition of a `SecondarySynthesiserButton` for the Advanced Performance Mode. In the Simple or Intermediate Performance Modes, the Virtual Music Instrument would automatically activate the indicated synthesiser type, however in the Advanced Performance Mode, the performer must press the `SecondarySynthesiserButton` in order to activate the secondary synthesiser. This meant that the notation also had to visually distinguish the different types. To achieve this, a new material option was added to the `Piece` class, so that the composer and developer could assign a different material to the secondary note type. This material could be entirely different than the primary and was not limited in any way by the appearance of the first, so could be as easily (or not) distinguishable as the composer or developer wished.

Arpeggiation, the new ability to sustain notes, and the secondary sythesiser type were tested in *Étude 4*, which arpeggiated frequently, sustained up to four notes concurrently, and used the secondary voice to separate the melody from the accompaniment.

Tracking the Rhythmic Position of the Instrument and Notes

The long-standing issue #21 was finally addressed in this release (see Appendix G.1)[15]. This had been delayed for a number of releases as there was no method to ascertain the rhythmic position of the note or instrument in relation to the progress of the note. In order to support hairpins, it was necessary for the note to be able to report how far the instrument was through it as a value between zero and one, where zero is at the beginning of the note, and one is at the end of the note. This could then be used to calculate the

[15] As of the `v0.2.1` release, there were 47 different releases. Issue #21 had been added in the `v0.1.4` release.

dynamic of the note which included crescendos simply by using the `Lerp` function of the `Mathf` class in the `UnityEngine` namespace, which returns a linearly interpolated result between two values, based on this percentage (Unity Technologies, 'Mathf.Lerp'). For example, the current dynamic of a note with a crescendo could be ascertained by returning a value between the dynamic value at the start of the note and the dynamic value at the end, based on this percentage. This method of linear interpolation was implemented with the `CurrentDynamic`, `CurrentAmplitude`, and `CurrentFrequency` methods, seen along with a segment of the `CurrentProgressThroughNote` method, in Code Sample 2.12. The latter `CurrentProgressMethod` method uses the `InverseLerp` method, also in the `Mathf` class, to calculate the progress through the note as a value between zero and one. The `CurrentFrequency` method was added preemptively to support glissandi, though was never used throughout the project's lifetime due to difficulties with time restrictions and implementing new note shapes.

```
1  public float CurrentFrequency(Instrument instrument) => Mathf.Lerp(Frequency,
       SecondFrequency, CurrentProgressThroughNote(instrument));
2
3  public int CurrentDynamic(Instrument instrument) => (int)Mathf.Lerp(Dynamic, SecondDynamic,
       CurrentProgressThroughNote(instrument));
4
5  public float CurrentAmplitude(Instrument instrument) => Mathf.Lerp(Amplitude,
       SecondAmplitude, CurrentProgressThroughNote(instrument));
6
7  public float CurrentProgressThroughNote(Instrument instrument)
8  {
9    if (Piece.Direction == PieceDirection.Left)
10   {
11     return Mathf.InverseLerp(0.001f, Length, instrument.transform.position.x -
           transform.position.x + (Length / 2));
12   }
13   else if (Piece.Direction == PieceDirection.Right)
14   {
15     return Mathf.InverseLerp(0.001f, Length, instrument.transform.position.x -
           transform.position.x - (Length / 2));
16   }
17
18   return Mathf.InverseLerp(0.001f, Length, instrument.transform.position.z -
         transform.position.z + (Length / 2));
19 }
```

Code Sample 2.12: Methods returning the current musical attributes or progress through a note

Additionally, as the rhythmic zero point was now more clearly defined, it was possible

to ascertain if the note was before or after this position and finalise the note accordingly. This prohibited the note from being played if it had passed the point.

Lazy Loading

The `v0.2.1` further simplified the process of setting up a scene in *Unity* by implementing the lazy singleton pattern, which automatically loaded and created required components of the `VMManager` class (Skeet). This removed the requirement to create a `GameObject` and manually place the `VMManager` onto that object as Virtual Music would now create one the first time it was referenced. This can be seen in Code Sample 2.13 and Code Sample 2.14, which provide excerpts from the `VMManager` class providing lazy loading, and the `Init` method which provides the singleton pattern.

```
private static VMManager _Instance;
public static VMManager Instance
{
  get
  {
    // Check first if there is another VMManager in the scene
    if (_Instance == null)
    {
      _Instance = FindObjectOfType<VMManager>();
    }

    // If there is still no VMManager, create one
    if (_Instance == null)
    {
      GameObject managerObject = new GameObject("Virtual Music");
      _Instance = managerObject.AddComponent<VMManager>();
    }

    return _Instance;
  }
}
```

Code Sample 2.13: An excerpt from the VMManager source code providing lazy loading

```
private void Awake()
{
  Init();
}

private void Init()
{
  if (Instance != null && Instance != this)
  {
    Destroy(gameObject);
    return;
  }

  if (RetainInstance)
  {
    DontDestroyOnLoad(gameObject);
  }

  // the rest of the Init() method loads required
  // components such as the Environment

  ...

}
```

Code Sample 2.14: An excerpt from the VMManager source code which provides the singleton pattern

2.5 Articulations and Extensibility

2.5.1 v0.3.0: Étude 5 and v0.3.1: Étude 6

The `v0.3.0` release was the first release targeting articulations and specifically targeting extensibility. It was also the largest release throughout the project's development (see Appendix J.1). Conversely, the `v0.3.1` release was comparatively much shorter (see Appendix K.1), and mostly focused on bug fixes or small improvements to the features introduced in the `v0.3.0` release. Because of this, discussion of the `v0.3.0` and `v0.3.1` releases has been combined.

The addition of articulations was chosen as a focus of this development period as their inclusion would increase creative options from the composer's perspective, making Virtual Music more attractive and natural as a method to express musical ideas.

These articulations came in the form of staccatos and accents, though groundwork was laid to support the easy implementation of additional articulations in the future via the `Note.Articulation` `enum` and the `Note.Articulations` `HashSet`[16].

After articulations, improvements to extensibility consumed much of the focus of this development period. This was because supporting the aim of extensibility also improved the accessibility of Virtual Music to developers, as having more possibilities for implementation increased the potential use cases of Virtual Music.

Both releases had a significant impact on how Virtual Music felt to use from the composer's, developer's, and performer's perspectives, because this focus on extensibility was also, to some extent, a focus on usability. A number of features which significantly impacted the usability of Virtual Music were also added outside of the focus of the development period. Most notably, the Pitch Guide and the Touch Buffer were introduced in these releases. These are significant features that are important to mention because of the effect that they had on the usability and extensibility of Virtual Music; however, they are explained elsewhere or are not relevant to the development focus and discussion in this section. The `v0.3.0` and `v0.3.1` Release Notes provide a detailed description of all changes in these release (see Appendix J.1 and Appendix K.1).

The first step to implement articulations was to consider what I as a composer sought, in terms of additional notational capabilities. These came in two categories: articulation, and ornamentation.

[16] A `HashSet` is a collection that prohibits duplicate elements (Microsoft, 'HashSet<T> Class')

To Ornament or Not To Ornament

Schmidt-Jones describes ornaments as 'extra notes, such as trills and slides, that are not part of the main melodic line but are added to the melody either by the composer or the performer to make the melody more complex and interesting' (73). Ornaments are usually related to pitch and may significantly alter the character of a phrase or melody. They are also a chance for a performer to express their musicality. They are often up to individual interpretation, however, as the notation of an ornament is often simply a convenient method to notate an understood playing technique or embellishment that can be notated manually otherwise. For example, trills, mordents, turns, appoggiatura, acciaccatura, and, to some extent, glissandi can be approximated with additional written notes both in traditional stave notation and with Virtual Music Notation. Performance practices surrounding these ornaments often change depending on context and style. This can be seen in the trill, which can start on a different note, alter between the notes at vastly different speeds, or end with another embellishment in the form of an additional turn, depending on context and even expressiveness. Such differences can be seen by comparing the description of the trill by authors in different historical periods. For example, Baroque flautist Johann Joachim Quantz states that each trill 'begins with the appoggiatura that precedes its note […] [that] may be taken from above or below'. Quantz also describes the 'termination' of the trill as 'two little notes which follow the notes of the shake', starting one diatonic tone below the originating trill. Conversely, in the 1893 publication titled *Musical Ornamentation*, pianist Dannreuther states that '*every shake* should begin *with the note itself*'[17] and does not describe any termination of the trill (145). Engraving standards may also impact the use and understanding of such ornamentation, though this is often due to the incorrect usage of the notation standard[18].

[17]Original emphasis.

[18]This can be seen especially with appoggiatura, which are often used in place of acciaccatura, making their indication often synonymous in modern compositions, simply due to most music notation programs placing them close together in the GUI.

Additionally, since performance in Virtual Music in the Simple Performance Mode especially is prescriptive (the performer is usually incapable of performing anything outside of what is notated), ornamentation—being an expressive form of embellishment—would also be forced to be prescriptive and sound exactly as defined in the settings of Virtual Music. If a performer wishes to ornament or embellish, they may use the Advanced Performance Mode.

For these reasons, I chose not to implement ornamentation during this focused development phase. Though, as seen with the implementation of the `CurrentFrequency` methods in `v0.2.1` (see subsection 2.4.2), Virtual Music is capable of producing such ornamentation if a developer should choose to inherit any class in the `Notation` namespace and implement their own ornaments and furthermore articulations.

Introducing Articulations

Schmidt-Jones describes the articulation as 'what happens in between the notes' (58). Schmidt-Jones goes on to say that 'the attack—the beginning of a note—and the amount of space in between the notes are particularly important'. Considering this, two articulations match these criteria specifically: the accent, and the staccato. Other articulations, such as marcato accents, mezzo staccatos, and staccatissimos, could be considered as variations of these, so, for this reason, the accent and staccato accents were designed to be variable through the `AccentPoint`, `AccentStrength`, and `StaccatoLength` respectively.

To notate an articulation, a composer may include an articulation character at the beginning of a note, after the specification of note or synthesiser type (see

subsubsection 3.2.3). A staccato may be indicated by a period character (`.`), and an accent may be indicated by a greater-than symbol (`>`). These characters were chosen due to their appearance being similar to their equivalent in traditional stave notation. A note may have more than one type of articulation.

The staccato effects the duration of the note, which is differentiated from the length. In Virtual Music, the length of a note is defined as the space between the start of the note and the start of the next. A note's duration represents how long it will sound in comparison to its length. The duration of a note with a staccato is by default set to 50 per cent of its length[19]. The shape of a note is set according to it's duration. This means that a note with a staccato will appear 50 per cent as long as a note with the same length that does not have a staccato. They will be spaced the same distance from the note that proceeds them, but the note with a staccato will add space to fill the rest of its length. Figure 2.8 shows eight crotchets at the same pitch. The first four crotchets do not have any articulations so their length matches their duration. The last four have staccatos, shortening their duration, while having the same length.

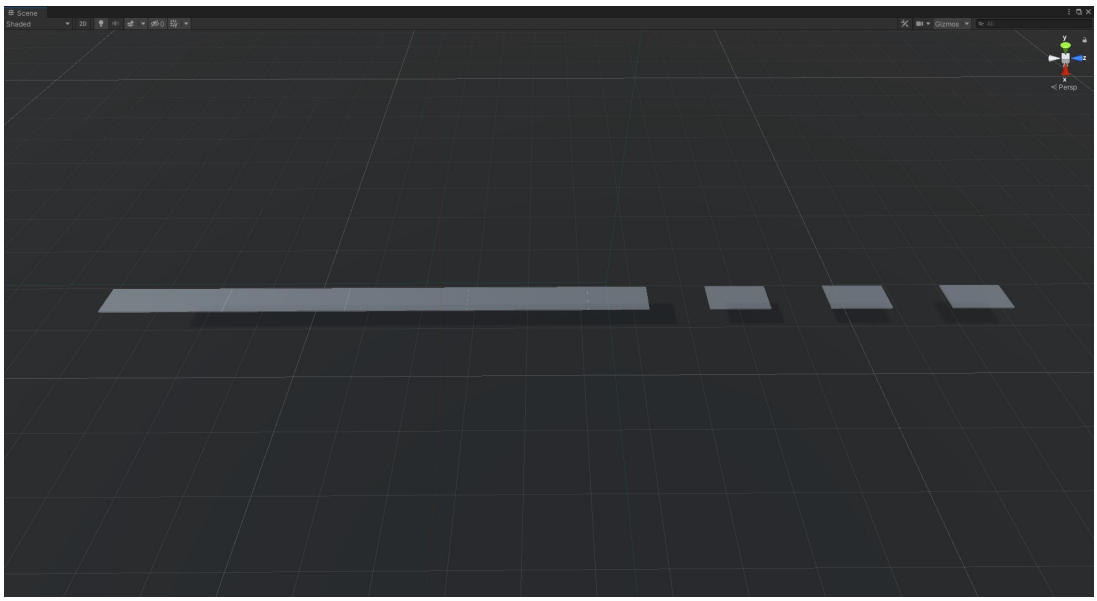


Figure 2.8: Four crotchets followed by four crotchets with staccatos

[19]This may be changed via the `StaccatoLength` field of the `Note`, and may be set per note.

An accent effects the dynamic value and amplitude at the start of the note. The strength of the accent, that is, how much the accent is multiplied by, is controlled by the `AccentStrength` field. This field by default equals `2f - Amplitude` so that quieter dynamics are more effected by accents. The strength of the accent reduces linearly until the accent point, set via the `AccentPoint` field. This field is considered a multiplier of the duration of the note and set to `0.2f` by default, which equates to 20 per cent of the note's duration. The appearance of the note with an accent is adjusted dynamically to reflect the accent, and the note shapes were adjusted so that notes with an accent increase in size and decrease linearly until the accent point, as expected. Figure 2.9 shows six crotchets at the same pitch. The first two have no articulations. They are followed by two accented crotchets, and then by two crotchets with accents and staccatos.

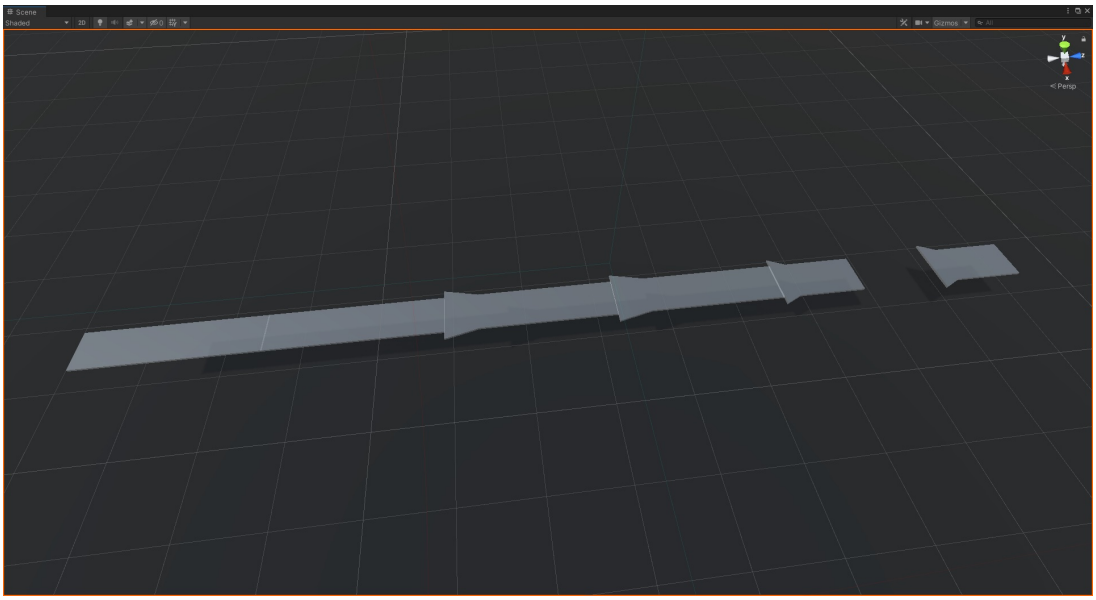


Figure 2.9: Six crotchets, four of which are accented

Extensibility and Activatables

The term 'activatables' in Virtual Music refers to objects which activate functionality that is custom or external to the code of Virtual Music. This can be seen in the form

of the events system, which could be used as a general purpose method to control and maintain Virtual Music. However, the ability to activate very specific functionality at specific times or with specific notes was missing or convoluted to implement. For example, by using the events system, it was only possible to activate a particular method on certain notes by comparing the properties of the note. Doing so meant that those properties, such as the note's number in the stave, must be known.

In order to be able to more easily specify which notes should activate certain methods, it was necessary to be able to 'flag' those notes and supply methods which should be run on triggering events related to those specific notes. This lead to the implementation of the note flag and flagged actions. The use of these functions is described in subsubsection 3.2.3 and subsubsection 3.3.14, respectively. *Étude 5* demonstrates the use of note flags to activate coloured lights around the player when notes with the appropriate note flag are touched, as seen in Figure 2.10. The intensity of the lights is increased if the note also has an accent. *Étude 6* demonstrates the use of note flags to activate particle effects that simulate fireworks, seen in Figure 2.11.

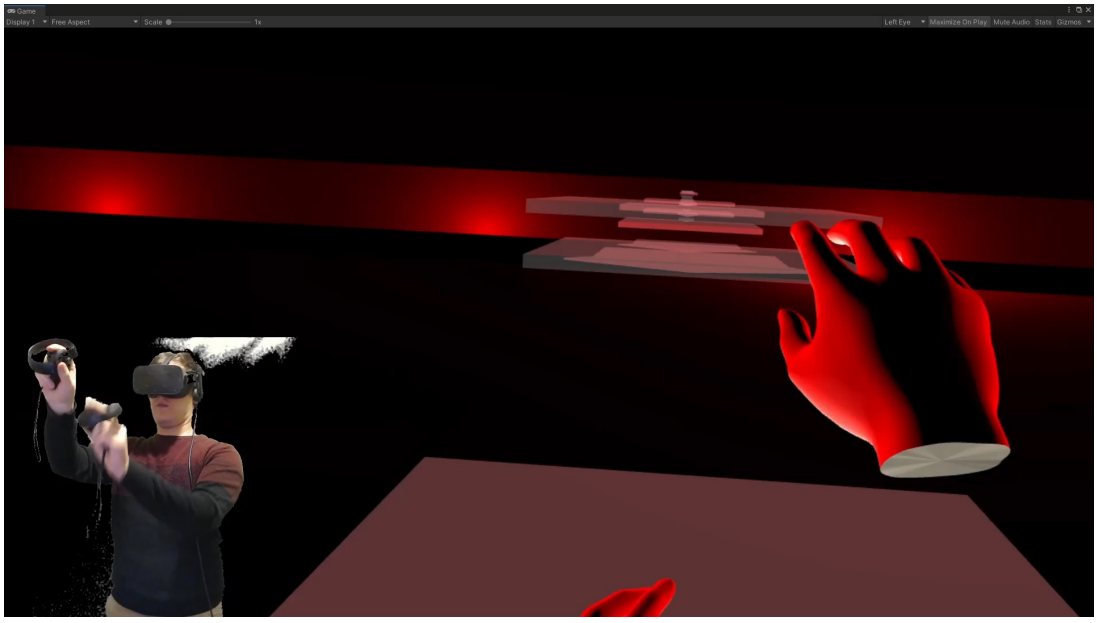


Figure 2.10: A screenshot from the *Étude 5* Performance Test

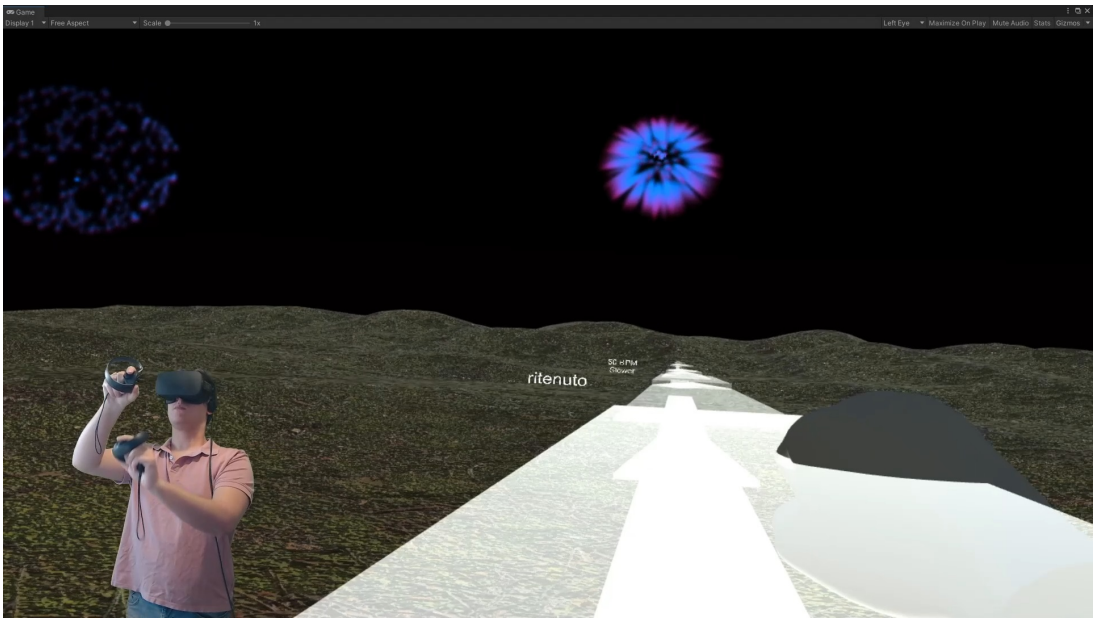


Figure 2.11: A screenshot from the *Étude 6* Performance Test

The term 'activatables' also in ways refers to extensibility. Before the `v0.3.0` release it was also only possible to use the note specification as defined by Virtual Music. For this reason, the `v0.3.0` release introduced the concept of the `StaveObject`, an inheritable class that represents an object on a stave. The `StaveObject` class maintains the functionality of a basic object on a stave, including its position, length, and which stave and therefore piece it is in. This class is designed to be inherited and is therefore `abstract`, prohibiting its direct use (Microsoft, 'Abstract'). It is used in the Virtual Music code in two places: in the `SettingsBlock` class, and in the `ActivatableStaveObject` class, both of which inherit the `StaveObject` class.

The `ActivatableStaveObject` class is an `abstract` class that inherits the `StaveObject` class but adds extra functionality by allowing the activation and deactivation of that object by the Virtual Music Instrument. It is finally inherited by the `Note` class, which adds functionality related to tone generation. The `ActivatableStaveObject` class allows a developer to easily implement their own stave objects that can be activated. This may be done to create additional objects that will make a percussive sound when touched, for example, that will still appear on a stave

and act as though they are a native part of VIRTUAL MUSIC.

Each of these classes, including the `Note` class, are designed for inheritance and include various 'pre' and 'post' methods that can be overridden by the inheriting class. For example, the `Note` class includes `PreStart` and `PostStart` `virtual` methods which are called at the start and end of the of the `Start` method, respectively. These methods, added in the `v0.3.1` release (see Appendix K.1), allow a developer to implement their own note with similar characteristics to the original note specification, but with new functionality specific to their use case.

Replacing the Events System

The focus on activatables brought attention to the problems with the current implementation of the events system. The existing event system had a number of issues that, while were addressable and in some cases seemingly desirable, caused confusion and ended in messy code. For example, the event system as of `v0.1.3` used the same `EventParameter` `struct` for each event manager (see subsection 2.3.4). This `struct` had predefined fields, including a `Note`, `GameObject` and `Stave` field, that each event manager supplied for as necessary, ignoring those that were not applicable to the event. However, this meant that the use of any events would need to appropriately check for `null`, and could cause confusion as it would not be clear what parameters were actually passed for that event without checking each field for `null` values. This was addressed in later releases by creating new event manager classes for each type of event, each with their own `EventParameters` `struct`, but resulted in multiple classes which inefficiently copied the same code and would need to be maintained individually.

Secondly, the custom event system implementation allowed events to be fired more

easily outside of the class they were related to. The `Note` class, for example, invoked methods related to staves, as a `Stave` class did not exist at this time. However, this could also be used incorrectly to emit events such as the 'finished' event for the stave, which could be invoked at any point, even if a stave was not finished, by any script. According to how the project was structured, there was no convenient fix to this.

Finally, as events were referenced to by a string, it was unclear which events existed, and events could be subscribed to incorrectly as the strings were case-sensitive. While this issue was addressed with the creation of an `enum` that listed the available event types as its members, it only created a new object that must be maintained and referenced.

The entire events system was replaced with the inbuilt specification as part of C# (Microsoft, 'Events'). It directly addressed these issues while retaining the same ease of use from the perspective of a developer and composer, as events could be subscribed to just as easily as they were static.

When events of this new events system implementation were triggered, they would call all methods supplied to that event. Specifically, the event allowed `System.Action` methods to be supplied to them. `System.Action` has seventeen different variants which are reflective of how many parameters they may take. These range from `System.Action`, which takes no parameters, to `System.Action<T>`, which takes a single parameter, to `System.Action<T1,T2,...,T15,T16>`, which takes sixteen parameters. The `T` here refers to 'Type' and can be replaced with any class used throughout Virtual Music (Microsoft, 'Action<T> Delegate'). This means that events may specify their own parameters individually, though this level of distinction was not used in Virtual Music as the events did not require it, as seen in Code Sample 2.15.

Secondly, events that use the inbuilt specification can only be triggered by methods in

```
public static event System.Action<Note> Activated;
public static event System.Action<Note> Deactivated;
public static event System.Action<Note> Reactivated;
public static event System.Action<Note> Redeactivated;
public static event System.Action<Note> Finalised;
public static event System.Action<Note> Generated;
public static event System.Action<Note> Failed;
```

Code Sample 2.15: The events in the Note class as of v0.3.0

their own class. This is done for method security, so that malicious or poorly implemented code cannot incorrectly force a class to emit an event. However, this required some refactoring as the `Stave` class, for example, relied on other classes to trigger its events. New methods were created in such classes that would trigger these events while ensuring that the event that they triggered was done so correctly or was relevant. This can be seen with the `Stave.FinaliseGeneration` method which is called by the `Piece` class in order to finalise the generation of the stave. This method first checks if the stave has in fact finished generating and if it has already triggered this event, prohibiting the trigger if so.

Finally, the available events could be clearly seen by either inspecting the class, or with an Integrated Development Environment that supports it. The default Integrated Development Environment recommended by and installed with *Unity*, *Microsoft Visual Studio* by Microsoft, supports this.

2.6 Improvements to the Synthesiser

2.6.1 v0.4.0: Étude 7

The `v0.4.0` release finally addressed long standing issues related to the reliability, sound, and usability of the synthesiser (see Appendix L.1). Notably missing from the synthesiser was the ability to customise the sound through amplitude envelopes and Low Frequency Oscillators (LFOs). All tones created by the synthesiser were therefore unfiltered (and uninteresting). The synthesiser also created artefacts when changing frequencies in the form of audible clicks. Finally, the concept of the synthesiser in Virtual Music was confusing and the interface to create new sounds was not simple or accessible. This was because the synthesiser was responsible for maintaining its oscillators and the specification of the oscillators must be done within the synthesiser class by overriding the `virtual Oscillator`, `Oscillator2`, and `GetWaveform` methods. This also limited the number of oscillators a synthesiser could have, unless a developer chose to inherit the synthesiser class and implement their own synthesiser with custom code.

The `v0.4.0` release targeted these issues with a complete refactor of the synthesiser class and with the separation of the oscillator from the synthesiser into a new `Oscillator` class. This `Oscillator` class, being a class that maintained the functionality of an oscillator, was then inherited by the `LFO` class, which extended this basic functionality with the ability to define an overriding frequency and amplitude so that it could serve as a Low Frequency Oscillator. Finally, amplitude envelopes were introduced so that a tone could shape its volume more naturally, and not simply produce a tone with constant amplitude.

Enveloping the Amplitude

Before the `v0.4.0` release, tones produced by the synthesiser generally remained at the same amplitude unless the note contained a hairpin or the performer varied the dynamics of the Virtual Music Instrument during performance. This, combined with limitation of simple oscillators with no filtering, made the Virtual Music Instrument sound 'digital' and lack character.

In digital synthesis, a synthesiser can shape the amplitude of a tone through use of an envelope or 'envelope generator', which contours the amplitude of the generated tone through various phases. Music journalist Mark Vail describes the ADSR (attack, decay, sustain, release) envelope as 'perhaps the most common type' of envelope (152). This type of envelope shapes the amplitude of a tone over four stages: the attack phase (when the tone is first started), the decay phase (the phase immediately following the attack phase), the sustain phase (when the tone is being held, after the attack and decay phases), and the release phase (when the tone is no longer being played and the amplitude is returning to zero). The application of an ADSR envelope adds significant character to a tone produced by a synthesiser and can be used to create any number of timbres. For example, a 'plucky', pizzicato-like sound can be created with low values for the attack and decay phases (implying that they are completed quickly), and a low or entirely zero value for the sustain and release phases, resulting in a tone that is short and does not sustain.

The `v0.4.0` release implemented envelopes through an `Envelope` class. The `Envelope` class is an envelope that can be applied to the amplitude of any type of oscillator and also the the overall amplitude of a synthesiser. In Virtual Music, they are in the form of D+ADSR, an alteration of the standard ADSR envelope with the addition of a 'delay' parameter, which adjusts how long the amplitude will remain at a

zero value before transitioning to 'attack' phase. The `Envelope` class adjusts the amplitude through its `Apply` method, which takes the current 'maximum' amplitude reported by the instrument, the time in seconds (calculated by dividing the sample index by the sample rate), and a true or false value reporting if the tone is being sustained or if it is the release phase. This method can be seen in Code Sample 2.16, which uses various methods in the `Mathf` class of the `UnityEngine` namespace.

```
public float Apply(float amplitude, int time, int sampleRate, bool sustaining)
{
  float timeInSeconds = (float)time / (float)sampleRate;
  if (!sustaining)
  {
    return Mathf.Lerp(amplitude, 0f, Mathf.InverseLerp(0, ReleaseTime, timeInSeconds));
  }

  if (timeInSeconds < DelayTime)
  {
    return 0f;
  }

  if (timeInSeconds < DelayTime + AttackTime)
  {
    return Mathf.Lerp(0f, Mathf.Clamp01(amplitude * AttackStrength),
        Mathf.InverseLerp(DelayTime, DelayTime + AttackTime, timeInSeconds));
  }

  if (timeInSeconds < DelayTime + AttackTime + DecayTime)
  {
    return Mathf.Lerp(Mathf.Clamp01(amplitude * AttackStrength), Mathf.Clamp01(amplitude *
        SustainStrength), Mathf.InverseLerp(DelayTime + AttackTime, DelayTime + AttackTime +
        DecayTime, timeInSeconds));
  }

  return Mathf.Clamp01(amplitude * SustainStrength);
}
```

Code Sample 2.16: The Apply method of the Envelope class

Though the envelope went through a rewrite in the `v0.4.1` in order to support the implementation of a GUI, the functionality of the envelope remained the same for the remainder of Virtual Music's development. For further discussion of the envelope, see subsubsection 3.8.3.

The Oscillator Classes

Throughout the development of Virtual Music I naturally became much more familiar with features of *.NET* and C#. In particular, towards the end of the development lifetime, I became familiar with and started using the unique access modifiers such as `internal` and `protected` to make Virtual Music more accessible to new users by limiting or hiding functionality of Virtual Music that was necessary for Virtual Music to function, but would not be needed for developers (Microsoft, 'Access Modifiers'). Importantly, as seen with the `StaveObject` class (see subsection 2.5.1), I also became familiar with the concept of `abstract` classes. Such classes could be used to create a simple method for developers to extend the features of Virtual Music, but also provide the basic functionality for these new features to work and integrate into Virtual Music effectively ('Abstract').

The `v0.4.0` release introduced the `Oscillator` class—a perfect example of an `abstract` class. The `Oscillator` class segmented and simplified the functionality of the synthesiser by being solely responsible for the maintenance of the waveforms it produced with methods such as `GetSample`, which itself called the `GetAmplitude` and `GetFrequency` methods. These methods took the amplitude and frequency specified by the synthesiser in the `GetSample` method and automatically calculated the amplitude and frequency, respectively. The methods also accounted for any modifications to these properties, including any specified in the newly introduced `FrequencyAdjustment`, `FrequencyMultiplier`, `OctaveDifference`, and `AmplitudeMultiplier` fields. Additionally, modifications to the amplitude and frequency by the newly introduced envelopes and LFOs supplied in the `Envelope` and `LFOs` fields respectively were applied automatically if applicable, as seen in Code Sample 2.17. These fields allowed the easier customisation of the sound.

```
public float GetAmplitude(float amplitude, int time, int sampleRate, bool sustaining)
{
  float adjustedAmplitude = Envelope?.Apply(GetBasicAmplitude(amplitude), time, sampleRate,
      sustaining) ?? GetBasicAmplitude(amplitude);

  return Mathf.Clamp01(adjustedAmplitude + adjustedAmplitude * GetLFOAmplitudeModifier(time,
      sampleRate, sustaining));
}

public float GetFrequency(float frequency, int time, int sampleRate, bool sustaining)
{
  float adjustedFrequency = GetBasicFrequency(frequency);

  return adjustedFrequency + frequency * GetLFOFrequencyModifier(time, sampleRate,
      sustaining);
}

public float GetBasicAmplitude(float amplitude)
{
  return Mathf.Clamp01(amplitude * AmplitudeMultiplier);
}

public float GetBasicFrequency(float frequency)
{
  return (frequency + FrequencyAdjustment) * FrequencyMultiplier * Mathf.Pow(2,
      OctaveDifference);
}
```

Code Sample 2.17: Methods of the Oscillator class calculating the amplitude and frequency as of v0.4.0

A developer or composer could now simply inherit the `Oscillator` class to create new oscillators by overriding `virtual` methods with methods relevant to the type of waveform desired. This is discussed further in the next section. This separation of functionality into a new `Oscillator` class allowed the synthesiser to behave more like a traditional synthesiser, as it was now simply responsible for the generation of audio signals by using any oscillators specified in its `Oscillators` field when the `Play` and `Stop` methods were called by the Virtual Music Instrument.

The `Oscillator` class, as an `abstract` class, was also an example of object-oriented programming principles and allowed the efficient code reuse for the `LFO` class (Stephens 122). The `LFO` class was introduced to increase options for timbre by acting as 'a signal generator that produces waveforms at sub-audible rates for controlling parameters such as the pitch of audio-frequency oscillators to produce vibrato using a triangle or sine wave or trills using a pulse or square wave' (Vail 138). The `LFO` class, like the `Oscillator` class, was an oscillator, but specified its own frequency and amplitude and did not consider

the frequency and amplitude requested by the synthesiser and instrument (though this functionality could be overridden if desired). It also specified what parameter it was controlling via the `Route` field, which could be set to either `Frequency`, or `Amplitude`, which adjusted the frequency or amplitude of the oscillator the LFO was attached to, respectively. These routes could be used for various effects, such as a vibrato effect with a sine or triangle waveform routed to frequency, or tremolo effect via a sine or square waveform routed to amplitude. Any number of LFOs could be added to the `Oscillator`. Interestingly, the `LFO` class, being an extension of the `Oscillator` class, could also have LFOs applied to it through the same `LFOs` field, vastly increasing potential options for sound and timbre. They may also use amplitude envelopes to adjust their effect over time. For example, by using the delay of the envelope, a vibrato effect could be made that only comes into effect after a short period of time. This effect was used in the piece composed for this release, *Étude 7*.

Eliminating Audible Artefacts

In digital synthesis, a waveform can be produced by taking multiple amplitude 'samples' per second, based on a given function, where amplitude here refers to a value between the numbers negative one and positive one. The amount of samples taken is dependent on the sample rate, often 44,100 or 48,000 samples per second. Throughout releases `v0.1.0` to `v0.3.1` the synthesiser produced samples for a waveform with a function based on a given time variable, which was simply the sample number (index) requested. A sine wave, for example, could be formed with the code provided in Code Sample 2.18, taken from the 'Sine.cs' file in `v0.3.1`, which returns the amplitude at the requested time[20]. This method uses the formula as seen in Equation 2.5, where y is equal to the returned sample amplitude, t is equal to time (`int time`, the sample index, divided by `int sampleRate`), A is equal to the maximum amplitude (`float amplitude`), and f is

[20]Note that the 'amplitude' parameter in this method refers to the maximum amplitude value.

equal to the requested frequency (`float frequency`). The `sampleRate` parameter in the method is simply the sample rate of the host system, often 44,100 or 48,000.

```
public override float Oscillator(float frequency, float amplitude, int time, float sampleRate)
{
  return Mathf.Sin(2 * Mathf.PI * frequency * time / sampleRate) * amplitude;
}
```

Code Sample 2.18: Retrieving a sample of a sine wave at a given time

$$y(t) = Asin(2\pi ft)$$

Equation 2.5: Retrieving a sample of a sine wave as a function of time

While effective in producing the waveform efficiently, if the requested frequency were to change, this method would no longer produce a reliable waveform. As this method does not consider the previous sample in any way, the next sample returned after a frequency change could be a considerably different value than what is expected, resulting in a quick and harsh jump from one amplitude to the other. These jumps, as illustrated in Figure 2.12, produce 'clicks' in speakers known as artefacts. Additionally, retrieving the sample in such a way prohibits any sort of modification to frequency with an LFO.

To solve this, it was necessary to track the current 'phase' of the oscillator, which is the sampled position in relation to one complete oscillation of the oscillator. The phase of a sine wave, for example, refers to a value between 0π and 2π. 0 refers to the starting point of the oscillation, 1π where the amplitude of the waveform has returned back to 0 after completing half a cycle, and 2π where the amplitude has again returned to 0 after completing a full cycle. This is illustrated in Figure 2.13.

By storing the current phase of an oscillator, a waveform may be created by returning the amplitude at that phase point as a sample with some sort of method. The `Sin`

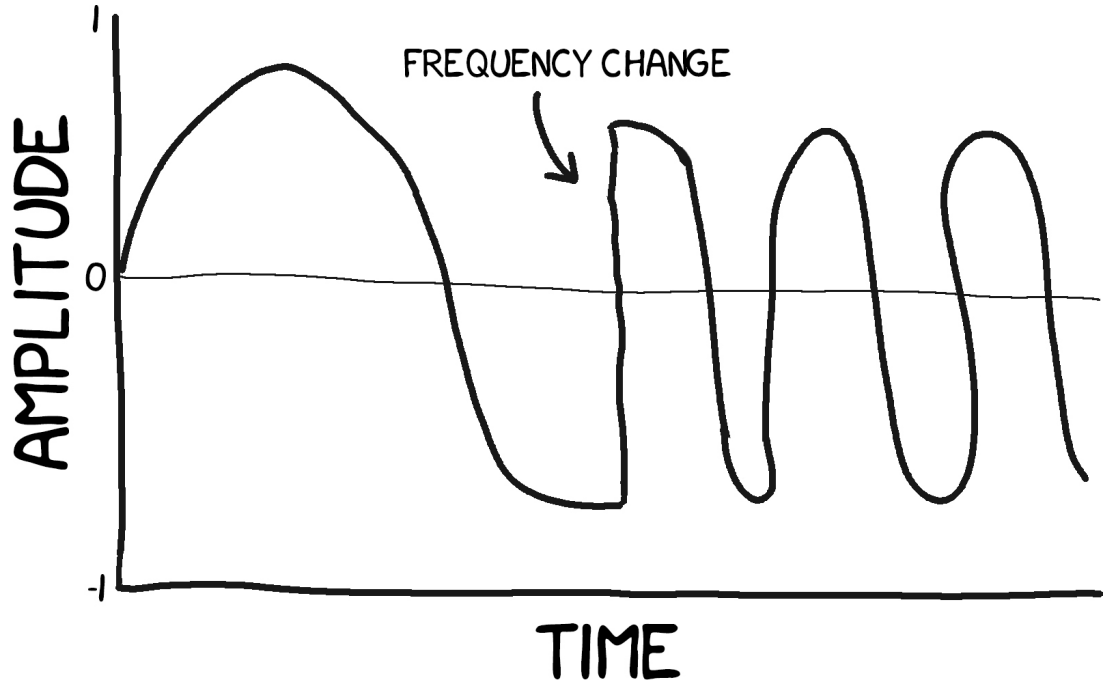


Figure 2.12: A sudden change in amplitude when retrieving a sample as a function of time

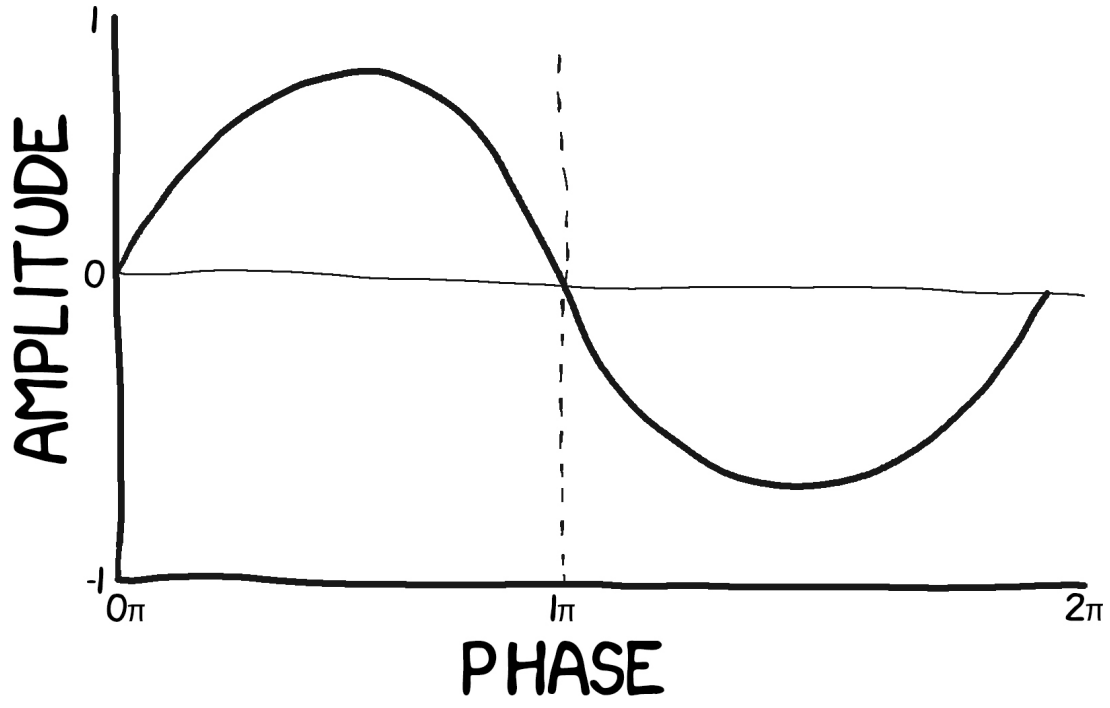


Figure 2.13: The phase of a sine wave

function of the `Mathf` class in the `UnityEngine` namespace is an example of such a function, which takes a phase parameter and returns the amplitude according to that value. Once the sample has been taken, the phase should be increased for the next sample, and so on.

Increasing the phase can be done with a method that takes current requested frequency. For a sine wave, the phase increase is simply calculated by converting the frequency to angular frequency. The formula for this can be seen in Equation 2.6, where ω is the angular frequency and f is the requested frequency. An application of this can be seen in Code Sample 2.19, which also shows the `Waveform` method and considers the sample rate. Note the use of the modulo operator (`%`), which ensures the phase stays between 0 and 2π.

$$\omega = 2\pi f$$

Equation 2.6: Converting frequency to angular frequency

```
protected virtual float IncreasePhase(float currentPhase, float frequency, float sampleRate)
{
  currentPhase += frequency * Mathf.PI * 2f / sampleRate;
  return currentPhase % Mathf.PI * 2f;
}

public virtual float Waveform(float phase)
{
  return Mathf.Sin(phase);
}
```

Code Sample 2.19: Increasing the phase of a sine wave according to the requested frequency

As the current phase of an oscillator was used as the basis for retrieving the sample and the frequency was used in order to calculate how quickly the sample increased the phase of the oscillator, the synthesiser could freely alter the requested frequency with no audible artefacts. This was because increasing the phase ensured uniformity in the waveform

and ensured that there were no harsh jumps in amplitude (unless intended as with the new 'white noise' oscillator, which returns samples of random values between negative one and positive one). This is illustrated in Figure 2.14, which shows the waveform of a phase-tracking oscillator over time, changing frequencies.

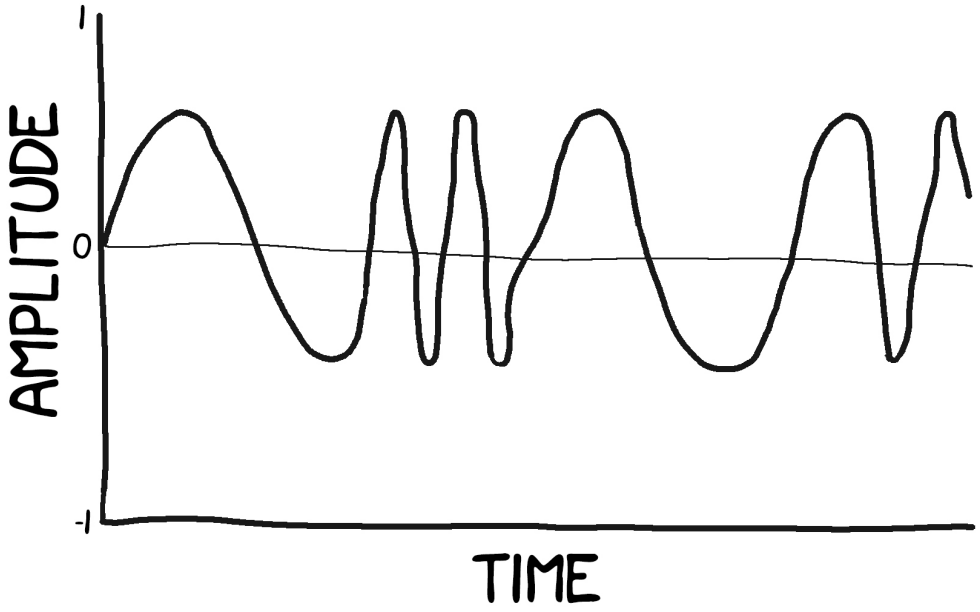


Figure 2.14: The waveform of phase-tracking oscillator with changing frequencies

This manner of phase tracking can be used for all basic waveform types, and a number of defaults are included as part of VIRTUAL MUSIC in separate classes. These classes each inherit the original `Oscillator` class, but override necessary methods and provide their own fields or methods where necessary. This can be seen in the `Square` class, which implements a square wave oscillator by means of a 'duty cycle'. The `DutyCycle` field is equivalent to the period of the oscillator's phase cycle as a percentage. In this period, the oscillator will return negative one and the remainder the oscillator's phase cycle will return positive one. For example, if the `DutyCycle` field is set to `0.5f`, the oscillator will return negative one for the first 50 per cent of its phase. The code for the final version

of this class is provided in Code Sample 2.20[21].

```
using UnityEngine;

[CreateAssetMenu(fileName = "Square", menuName = "Virtual Music/Oscillator/Square", order =
    2)]
public class Square : Oscillator
{
  [SerializeField]
  [Range(0.01f, 0.99f)]
  private float _DutyCycle = 0.5f;
  public float DutyCycle
  {
    get => _DutyCycle;
    set => _DutyCycle = Mathf.Clamp(value, 0.01f, 0.99f);
  }

  public override float Waveform(float phase)
  {
    return Mathf.Approximately(phase, 0f) ? 0f : phase < DutyCycle * TwoPI ? -1f : 1f;
  }
}
```

Code Sample 2.20: A square wave oscillator that tracks phase

2.6.2 v0.4.1: Étude 8

As noted in the `v0.4.0` Performance Test (see Appendix L.2), while the modifications to the synthesiser and related classes improved the timbral characteristics of the synthesiser, synthesisers, oscillators, LFOs, and envelopes could still only be created in code. Additionally, it was not possible to override the synthesiser without using a script or modifying the code of Virtual Music, as I noted in the Performance Test:

> All changes to oscillators are proving good. It is much easier to create new synthesisers and tonal qualities without the necessity to have a deep understanding of Virtual Music, and both synthesis and code. The implementation of a new Synthesiser still requires some code

[21] The keen-eyed of you will notice use of 'CreateAssetMenu'. See subsection 2.6.2 for more information.

> implementation, however, which would detract some users from creating new synthesisers. It may be good to look into some sort of user interface or external application which can help users develop new Synthesisers.
>
> [...]
>
> The simplification of Oscillator and Synthesiser has made it easier to create new Synthesisers, but made it clear that it is still not trivial to override which Synthesiser is loaded. It will be finally necessary to simplify the way to override the default synthesiser, including adding the option to specify a Synthesiser on a per Piece basis. (Appendix L.2)

While some timbral character could be added to the synthesiser by using LFOs, it was not possible to filter or add any sort of audio effect to the synthesisers or Virtual Music Instrument. This, and the comments in the `v0.4.0` Performance Test led to the complete redesign of the classes related to the synthesiser, consuming much of the development focus for this phase. The redesign added two new classes of particular interest: the `Tone` class, and the `Output` class. The combination of these two classes finally allowed an unlimited number of tones to be played simultaneously[22] and the addition of audio effects and filters to the Virtual Music Instrument.

The 'Basic' Performance Mode was also introduced in this release. This Performance Mode would sustain all notes once they were touched, regardless if they were sustained using the sustain button or if they were deactivated. This Performance Mode was introduced as a 'special' Performance Mode to increase accessibility to users with mobility issues and to increase applications of Virtual Music in therapeutic scenarios. See subsubsection 3.7.2 for more information about the Basic Performance Mode. Other usability improvements were also included in the `v0.4.1` release, such as improved handling of the deletion of notational objects and the ability to prohibit changing a

[22]Prohibited only by computer performance and limitations of live performance.

piece's `FromDirection` once it had been generated. See Appendix M.1 for a complete listing of new features and modifications in the `v0.4.1` release.

Redesigning The Synthesiser

The `Oscillator` class introduced in `v0.4.0` made it clear that the `Synthesiser` class had become confusing and limiting. The `Oscillator` class, in many ways, was 'stateless'; it provided a means to calculate the state of tones produced using that oscillator, but stored no information about the tones it produced. This was not true for the `Synthesiser` class, however, which stored information about what frequency and amplitude it was currently producing. It was therefore 'stateful' and could only produce one tone at a time. The Virtual Music Instrument also did not store any information about the tones it produced. Instead, it monitored its attached synthesisers and controlled them by providing frequency and amplitude information. The Virtual Music Instrument overcame this limitation of a single tone somewhat, by including two synthesisers of each type, allowing a total of four tones in total, per instrument. The code to maintain these tones, however, was becoming unmanageable as it was merely iterated upon on when polytonality was introduced (see subsection 2.4.1). It had been iterated upon frequently, but never refactored. As synthesisers were attached to the instrument, a composer could not specify nor supply a synthesiser for use without modifying the code of Virtual Music. There was also no means to change the synthesisers once they were loaded, and loading multiple synthesisers was inefficient and impacted computer performance.

To solve these problems, I needed to proverbially 'go back to the drawing board' and rethink what the synthesiser was and what it should achieve. Initially, the synthesiser was a self-contained (and somewhat poorly implemented) means to produce a single tone,

as there was initially no concept of polytonality nor of timbral alteration. The Virtual Music Instrument, conversely, merely instructed the synthesiser that it should produce a tone at a certain frequency and amplitude, but had no responsibility to maintain those tones. If the instrument component was removed, for example, the synthesiser would continue to happily produce a tone indefinitely[23].

With the introduction of oscillators and envelopes especially, the purpose of a synthesiser was changed to a means of describing how a tone should sound. The Virtual Music Instrument therefore needed be responsible for the tones it produced, and should pass data to the synthesiser in some sort of data collection that maintained the tone, including its duration, current frequency, and amplitude. This could be likened to a synthesiser controlled by a piano keyboard: a synthesiser defines how the keyboard should sound when its keys are pressed, and, if the synthesiser is polyphonic, many keys of the keyboard can be pressed simultaneously and the synthesiser will produce similar sounding tones.

This lead to three changes; first, the `Tone` class was introduced, a class responsible for maintaining the status of the tone it represented, including storing the phase of the oscillators it used. Second, a new `Output` class was introduced to handle tones and route the produced audio. Finally, the `Instrument` and `Synthesiser` classes were refactored to suit this format. I also took advantage of this refactor to implement a GUI for the creation of all classes related to the synthesiser by using the `ScriptableObject` class of the `UnityEngine` namespace and its `CreateAssetMenu` attribute (see subsection 3.8.3). This refactor allowed the `Synthesiser` class to become stateless and, furthermore, allowed a developer or composer to specify a synthesiser on a per-piece basis. This was because many synthesisers could now be loaded concurrently with little impact to computer performance, and tones could be passed freely between them (see

[23]In some instances, these tones would continue to be produced even after the application was closed. When debugging this issue, I quickly learned what 261.6256 Hz sounded like.

subsection 3.3.15 and subsection 3.6.6).

The Tone and Output Classes

The `Tone` and the `Output` classes were the most difficult features to implement throughout the entire development of Virtual Music. This was because implementing these features required an understanding of working within *Unity*'s multi-threaded environment[24].

In *Unity*, audio processing is done within the 'audio' thread. Conversely, most functionality involving scripting, including most functionality of Virtual Music, is executed in the 'main' thread. The `OnAudioFilterRead` method, in particular, is used by Virtual Music to generate audio procedurally, which means that 'calling into many *Unity* functions from this function [was] not allowed' (Unity Technologies, 'MonoBehaviour.OnAudioFilterRead'). Before the `v0.4.1` release, the synthesiser merely read data in its own class set by the instrument in a non-destructive manner. It was not responsible for maintaining the data which it read, and therefore handled the multi-threaded environment effectively[25].

A 'tone' in Virtual Music is considered as the planned creation, continuation, and finalisation of a musical sound by the Virtual Music Instrument. To support a structure in which a tone is contained within its own class, the instrument would need to create the class for the tone. The synthesiser would then need to read the data from

[24]'Multi-threaded environment' refers to *Unity* executing functionality on multiple 'threads' concurrently. These threads operate simultaneously and care must be taken to ensure that collections of data are not modified in one thread while another is handling that collection. Some features and and functions are either limited or prohibited in threads they are not designed for. The *Console* window in *Unity* shows warnings and errors in these cases (Unity Technologies, 'MonoBehaviour.OnAudioFilterRead').

[25]Incidentally, this meant that I had never considered needing to support multi-threadedness, as no issues related to multi-threadedness had ever occurred.

that created class and set the phase and related data accordingly. To finish a tone while supporting the release phase of the envelope (which helps create more natural timbral characteristics), the instrument would need to mark the tone as 'released' while the synthesiser continued to produce the audio for that tone until the release phase is finished.

Two issues surround this process. Firstly, the frequency and amplitude information of tones are calculated using the position and rotation of the instrument. This data cannot be accessed from the audio thread. The position and rotation of the instrument would therefore need to be cached regularly and that cached data should be used instead for such calculations. This was a relatively simple issue, and was solved by storing the current positional and rotation data into the `CachedPosition`, `LocalPosition`, and `CachedRotation` fields of the `Instrument` class, respectively. The `CurrentPitchDegree` field, `GetFrequency` method, and `GetAmplitude` methods of the `Instrument` class were then altered slightly to use these fields, accordingly.

Secondly, both the instrument, which operates in the main thread, and synthesiser, which operates in the audio thread, would need to access the same collection of tones. This issue was solved by carefully structuring the tone collections so that the instrument would not modify the size of them directly. This was done with the addition of three collections in the `Output` class: the `Tones`, `TonesAwaitingAdd`, and `TonesAwaitingRemoval` collections, each of a `HashSet` type to ensure that there were no duplicate entries.

The organisation of tones in these collections allowed the instrument to request the generation of sound by adding newly created instances of the `Tone` class into the `TonesAwaitingAdd` collection, if they did not already exist there or in any of the other collections. The `Output` class could then sort through this collection and move the tones within it to the `Tones` collection. Once in the `Tones` collection, the instrument

could update the frequency and amplitude information freely by using the adjusted `GetFrequency` and `GetAmplitude` methods as mentioned previously. Finally, to trigger the transition of a tone into the envelope's release phase, the instrument would simply alter the state of the tone to show that it should be releasing. The `Envelope` class would then process the amplitude of the tone accordingly, and once it had finished the release phase, it would copy the tone and place it into the `TonesAwaitingRemoval` collection. The `Output` class would then traverse through this collection upon the execution of the `HandleTones` method in the audio thread, and remove that tone accordingly.

This method allows the instrument to request an unlimited number of tones which may be passed to any synthesiser freely, a significant improvement over the maximum of four tones in previous releases. However, issues relating to multi-threading may occur infrequently; it is possible that the frequency or amplitude information of a tone may be updated in the same computation cycle as the audio thread is retrieving this data. This causes no errors, however, and will only show a warning in the *Console* window.

The Addition of Audio Effects and Filters

The newly introduced `Tone` and `Output` classes allowed routing the audio produced by the instrument and individual synthesisers to mixers. This meant that *Unity*'s inbuilt audio effects and filters could now be applied to the instrument, or to individual synthesiser types. These effects and filters included reverb, chorus, distortion, compressors, and high and low passes. *Étude 8* demonstrates the effect of a reverb effect (see subsection 3.7.8 and subsection 3.6.8 for more information). The piece also demonstrates the use of a second oscillator on the secondary synthesiser in order to add a fifth interval above secondary note types. This was achieved by supplying the

`FrequencyMultiplier` field of the `Oscillator` class a value of `1.498307f`, the decimal value of a fifth interval in a twelve-tone equally tempered octave. Note flags were used to switch the secondary synthesiser with an alternate synthesiser in order to produce a delayed vibrato like effect on certain tones. This alternate synthesiser made use of an LFO and an envelope with a delay, the same technique demonstrated in the piece composed for `v0.4.0`, Étude 7 (see subsection 2.6.1).

While I considered implementing my own filters by making use of the Fast Fourier Transform, I chose to 'keep it simple, stupid', and instead use these inbuilt effects and filters. This was also so that composers, developers, and I could make use of the wealth of knowledge and packages available in *Unity*'s *Asset Store*. Additionally, the implementation of the Fast Fourier Transform was complicated, would limit effects and filters to those that I implemented, and was also simply a little out of my breadth of knowledge.

3 | Developer Manual

A Developer's Guide to Using Virtual Music

3.1 Get Started

3.1.1 How to use this Get Started Guide

This *Get Started* section contains ordered topics to help a new user get started with VIRTUAL MUSIC in *Unity* with an Oculus device. The structure supports a user to quickly and easily install and use VIRTUAL MUSIC and familiarises them with *Unity*, Oculus, and VIRTUAL MUSIC fundamentals, if they are not already familiar with these technologies. Depending on their needs and level of experience with the technologies, readers may choose to follow this *Get Started* section from start to finish or use individual parts as a standalone reference.

3.1.2 Prerequisites

A development environment with an Oculus device and an appropriate version of *Unity* must be correctly configured and installed before getting started with VIRTUAL MUSIC. This process and more information on how to create a new 3D *Unity* project is covered in great detail in *Get Started with Oculus in Unity* (Facebook Technologies, 'Get Started with Oculus in Unity'). For the convenience of any reader who wishes to get started quickly, the process to install *Unity* and set up a new 3D project for all versions of *Unity* at the time of writing is summarised in Appendix N. This summarised process assumes that the Oculus device is already correctly configured with the required settings for development and testing.

3.1.3 Preparation of Required Components

Before importing the VIRTUAL MUSIC package, both the *OVRCameraRig* and *LocalAvatar* need to be installed and configured in the scene. These components are part of the *Oculus Integration* package and are required for crucial functionality, including tracking the performer's position and hands. Appendix O describes the recommended process to allow the use of all components of VIRTUAL MUSIC. Other methods may be acceptable and produce a minimal set of functionality, but are not explicitly described or supported.

3.1.4 Install Virtual Music

First, download and extract the contents of the VIRTUAL MUSIC repository into a directory called 'VirtualMusic'. This directory will serve as the root folder for all classes and resources required to use VIRTUAL MUSIC, and should contain subdirectories including 'Environment' and 'Instrument' as seen in Figure 3.1. Other files and directories including '.gitignore', 'LICENSE', and '.github' may also exist, but can be ignored.

Second, to install the VIRTUAL MUSIC package in a *Unity* project, drag the 'VirtualMusic' folder into the project's 'Assets' directory. VIRTUAL MUSIC will automatically load required components for most situations, thanks to the implementation of the lazy loading pattern (Skeet). If the *Console* reports any errors, refer to *Prerequisites*, because the *Oculus Integration* package might not have been imported correctly.

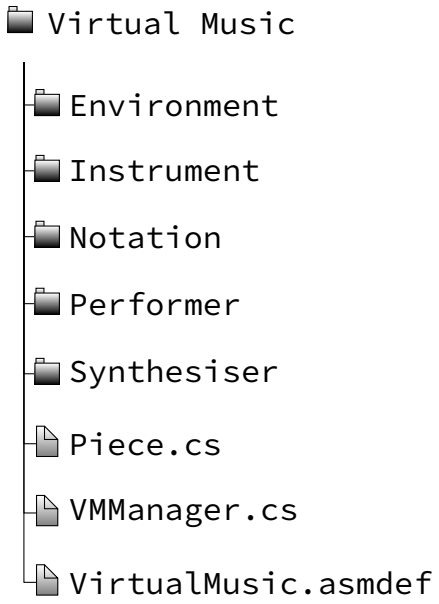


Figure 3.1: The contents of the 'Virtual Music' repository and folder

3.1.5 Next Steps

Virtual Music and *Unity* are now set up with the minimum required components to *Load and Play a Piece* or *Free-play with the Virtual Music Instrument*. However, it may be necessary to *Adjust the Playable Area* if the default values are not suitable.

Load and Play a Piece

Music composed for Virtual Music is human-readable computer-parsable strings of characters called Virtual Music Shorthand Notation. Blocks of characters usually written by hand are arranged in a particular format to represent musical notes, chords, or instructions, and are separated by spaces. These notes are then sequentially interpreted by Virtual Music to create lines of music notation called staves which form the visual notation—Virtual Music Notation. Virtual Music Notation is suitable for live interpretation as a form of graphic notation, or for prescriptive performance using the

Virtual Music Instrument. Virtual Music Shorthand Notation, like modern music notation, is capable of representing anything from simple one-dimensional melodies to complex contrapuntal sets of musical instructions. For more information, see *Shorthand Notation*.

Being simply plaintext for later interpretation by Virtual Music, Virtual Music Shorthand Notation is easily preserved and can be distributed amongst composers and performers through any communication or sharing platform. These platforms include email and message or file sharing services, or theoretical distribution platforms developed specifically for Virtual Music such as discussed in subsection 5.2.5. Once a stave or staves of Virtual Music Shorthand Notation have been obtained either by composition or by a method as previously discussed, it can be performed with one change to the scene, as described in *Adding a Piece to a Scene*. Synthesisers or visual elements including the performance environment are not possible to capture with Virtual Music Shorthand Notation due to their complexity and size. This likens the distribution of Virtual Music Shorthand Notation to the distribution of sheets of modern music notation; while some music is interchangeable or can be played on different instruments, such as a violin piece being performed on a clarinet or a violoncello piece being performed on a viola, it will not necessarily sound or appear as the composer intended.

Free-play with the Virtual Music Instrument

By default, the Virtual Music Instrument is configured for simplified usage with automatic tone generation and control: it produces tones with its synthesisers automatically when a note is touched, and it is not possible to produce a tone otherwise. In this situation, a piece must be loaded in the scene as described in *Adding a Piece to a Scene*.

However, it is possible to free-play with the VIRTUAL MUSIC INSTRUMENT by switching its 'Performance Mode' to 'Advanced' as described in *Changing the Performance Mode*. In the Advanced Performance Mode, the performer may play freely and produce tones with or without a piece with the appropriate control scheme (see *The Control Scheme*). This mode may also be desirable for improvisation or practice. If a piece is loaded, a performer may prefer this mode to have more creative and interpretive freedom. See Appendix S for more information about the available Performance Modes and their respective functionalities.

Adjust the Playable Area

The playable area is measured from the floor height. The minimum and maximum heights can be obtained through `VMPerformer`, at `VMPerformer.Preferences.MinimumHeight` and `VMPerformer.Preferences.MaximumHeight` respectively. By default, the minimum height is set to one metre from the floor, and the maximum is set to one meter and sixty centimetres. To change these values, see *Setting the Performance Area*.

3.2 Shorthand Notation

Music composed for VIRTUAL MUSIC is notated with human-readable computer-parsable strings of characters called VIRTUAL MUSIC SHORTHAND NOTATION, inspired by the *LilyPond* scheme of music notation (LilyPond). Blocks of characters called symbols represent musical notes, chords, or instructions. Usually written by hand, a symbol's characters, called segments, must be arranged in a particular format and order. Symbols are separated by a space to create lines of music notation called stave strings, which are sequentially interpreted to create staves of the

interpreted visual notation, Virtual Music Notation. Bracketed symbols called Settings Blocks allow the composer to indicate tempo or tonal changes such as changes to the fundamental frequency and the division and amount of octaves.

Virtual Music Shorthand Notation, like modern music notation, is capable of representing anything from simple one-dimensional melodies to complex contrapuntal sets of musical instructions. In order to make full use of Virtual Music, especially as a composer, a good understanding of Virtual Music Shorthand Notation is recommended. This requirement is not necessary in order to perform using Virtual Music, though a basic understanding will help the performer understand how a particular musical instruction was notated and would improve their understanding of the composer's intention.

The first stave of a piece will often start with a Settings Block so that the produced notation accurately represents the composer's intentions in terms of pitch and tempo. Because of this, it is necessary to begin with the Settings Block as it may impact the use of any of the following music symbols and segments. The following sections describe the use of the available symbols, segments, and their character options.

3.2.1 Multi-stave Limitations

Due to the ability to notate *Polytonality*, it is usually only necessary to notate one stave string in order to produce a contrapuntal composition. However, an unlimited number of stave strings may be notated with some limitations. All staves begin at the same time and progress at the same tempo, and each stave may have the same types of symbols. However, by default, Settings Blocks may only be provided on the first stave unless `Piece.Generate()` is called with an optional `true` parameter, which sets

`allowMultiStaveSettingsBlocks` to true. This limitation limits unintentional and confusing behaviour, as both staves are impacted by the resulting effects of any Settings Block. For more information, see *Giving a Piece Notation*.

Providing multiple stave strings may be desirable to simplify contrapuntal music, for example when a complicated and rhythmically independent bass line accompanies a separate melody. This is seen in *Peril (But Not Quite Doom)*, which has two staves separated by function, the first serving melodic and harmonic functions, and the second serving as a bass line. As each stave is interpreted individually, a composer must take care to calculate and provide the appropriate amount of rests, as well as ensure that the performance remains possible and is not visually confusing.

3.2.2 Settings Blocks

Much like tempo markings and key signature information in traditional stave notation, Settings Blocks are usually the first symbol of a stave string. They control various aspects of the performance and notation, including the tempo, the fundamental frequency that each pitch degree is based on, and how many pitches are available. Another similarity to tempo markings in traditional stave notation is that Settings Blocks may also be notated at any point in a piece, any number of times. This includes changes to tempo and the fundamental frequency, and to the amount of pitches available per octave, including changes to how the octave is divided and the number of octaves available.

When a Settings Block approaches the playable area, the Settings Block and its settings are invoked immediately. Changes are invoked silently with no specific feedback to the performer. Tempo markings are shown either to the side if the notation is approaching the performer from in front of them, or behind the notation if the notation

is approaching from the left or right (see *Tempo, B*).

```
[B=120,March_tempo,t;O=2;T=12;F=261.63;]
```

Piece Excerpt 3.1: An example of a Settings Block

Settings Blocks are notated by square brackets with any number of the available settings separated by a semicolon. Settings are indicated with key-value pairs delimited by an equals sign, similar to INI file style configuration systems commonly seen in server configuration files such as *Windows Server 2012* (Microsoft, 'Configure an Ini File Item'). Additional parameters for tempo settings are delimited by a comma. Piece Excerpt 3.1 shows an example Settings Block with a tempo adjustment signified by the `B` key, an octave number change signified by the `O` key, an octave division change signified by the `T` key, and a frequency change signified by the `F` key.

Tempo, B

Tempo changes adjust the tempo and are accessible via the `B` key, short for Beats Per Minute (BPM). It can be seen as the first key-value pair in Piece Excerpt 3.1. By default, changes to tempo are shown to the performer with a small block of text listing the tempo marking in Beats Per Minute (BPM) and any accompanying text below it. If a piece has `Piece.FromDirection` set to `Straight`, tempo change indications will appear to the left hand side of the generated notation. If `Piece.FromDirection` is set to either `Left` or `Right`, tempo change indications will instead appear behind the notation.

Tempo changes have three optional parameters: first, the numeric tempo indication, second, optional text to accompany the tempo change, and finally, a display option. Additional parameters are added by a comma as a delimiter, and must be ordered

correctly.

The numeric tempo indication should immediately follow the equals sign and is simply the desired tempo in the number of BPM, where a beat is defined as one non-dotted crotchet. This tempo change must be a positive whole number (a positive integer) above zero. Second, a composer may provide a string of text to be included below the numeric indication. The provided text may only consist of alphanumeric characters of any case, and underscore characters (`_`) which will be converted into spaces. Finally, the tempo change may be hidden entirely with `n` for 'none', only show the numeric tempo marking with `b` for 'BPM', only show the accompanying text with `t` for 'text', or verbosely state that both the numeric indication and the accompanying text should be shown with `a` for 'all'. This final parameter may only exist if the first two are also included, however may be added with a blank second parameter by notating two commas after the numeric tempo indication. This may be desirable when the composer wishes to hide the tempo change entirely and does not wish to provide text to the second parameter, since it is to be hidden anyway.

A tempo change may also be simply the second parameter. This would result in a tempo change that only shows the desired text but does not change tempo, without the need to restate the tempo value and hiding the tempo in the Settings Block. However, this text cannot only be numeric, as those numbers would be interpreted as the numeric tempo indication. If a composer wishes to include only numbers as the tempo text, they must also include a numeric tempo indication and optionally hide them with the third parameter.

The definition of beats is currently limited to simple, non-compound beats. That is, a single beat is a single crotchet (two quavers). To compose a piece in compound time, a composer must either increase the BPM by one half so that rhythmic lengths remain relative to what would be expected of their equivalent counterpart in traditional

stave notation, or add a tuplet to every rhythmic indication. The latter case would be recommended for brief and rhythmically simple uses of a compound time, whereas the former would be better suited if the compound time was for an extended period of time. In the case of the former, a composer may also wish to hide the text showing the numeric value of the BPM and instead show a custom set of text. Another option would be to use the tempo text to show the equivalent tempo marking in dotted crotchets per minute.

Fundamental Frequency, F

The fundamental frequency is the 'base frequency' from which each pitch degree ascertains its pitch. By default, the fundamental frequency is set equal to C4 or 'middle C' of an equal-tempered scale where A4 is tuned to 440 Hz, `261.63`, seen also as the second key-value pair in Piece Excerpt 3.1. The fundamental frequency can be changed via the `F` key, and following the equals sign delimiter, a number indicating the frequency which should be used. This number is saved in memory as a `float` and is subject to floating point accuracy limitations, so a maximum of six decimal places is recommended, though no more than three should ever be necessary (Microsoft, 'Floating-point numeric types').

Number of Octaves, O

The `O` key indicates a change to the number of complete octaves available to the performer and in the notation. The value represents the minimum number of octave intervals for any given degree of an octave. This excludes the tone that is the fundamental frequency, which has an additional tone to complete the final octave. The value must be a positive whole number (a positive integer) above zero. The default value of two can be seen as the third key-value pair in Piece Excerpt 3.1.

While there is no limit to the value of this key, increasing the value far beyond the default will cause the notation to become unclear, as the spacing between each pitch degree will become increasingly smaller as the value increases. The default value of two allows a good amount of range and a composer should use octave alterations or changes to the fundamental frequency if they wish to briefly notate outside of that range.

Octave Division, T

The 'Octave Division' (also called the 'Number of Tones Per Octave') is accessible via the `T` key, short for 'tones per octave'. Adjusting the value for this key adjusts how many tones there are per octave and excludes the repeated top octave, the octave interval. The tone division is tuned according to the equal temperament tuning system. That is, each tone of the octave is spaced on a logarithmic scale by a ratio equal to the t root of 2, where t is equal to the octave division. By default, the octave division is set to `12`, matching a semitone scale of modern Western musical convention, but the octave division could, for example, be set to `24` to create a microtonal composition. This means that the frequency of any pitch degree (x) can be calculated by multiplying the base frequency (f) by two to the power of the pitch degree (p) divided by the octave division (t), seen in Equation 3.1.

$$x = f \times 2^{\frac{p}{t}}$$

Equation 3.1: Calculating the frequency of a pitch degree

3.2.3 Music Symbols

Segments of music symbols control various aspects of any produced notation, ranging from the note or synthesiser type to the duration and pitch of the note. They must be carefully constructed to ensure that they will be correctly interpreted by VIRTUAL MUSIC, and each symbol must have its segments arranged in a particular order. There are eight possible segments. In their appropriate order, the available segments are:

1. (optional) note type
2. (optional) articulation
3. pitches and rests
4. (optional) octave alteration
5. time
6. (optional) dynamics
7. (optional) note flags
8. (optional) polytonality

Not all symbol segments are compulsory; only pitch and time are compulsory segments. Segments such as articulation indications are optional indications, matching their traditional stave notation counterpart, and other segments such as dynamics may be either remembered or set contextually if none are supplied, such as rhythms in polytonal symbols.

For the purposes of this *Developer Manual* and to make the necessary order understandable, the numbered position of each segment is defined by which position

that segment would be in if all possible note segments were included. For example, the articulation segment is defined as the second segment, even if a music symbol does not have a note type segment in its definition. Similarly, the polytonality segment is the eighth segment; even in the absence of all other optional segments, this segment would be considered the eighth segment and not the third.

Piece Excerpt 3.2 shows a single complicated symbol that contains most available segments and their available characters. In this example, the produced note would be of the secondary note type due to the `!` character, and would have a staccato articulation as indicated by the `.` character. For its pitch, it would be at the seventh pitch degree, indicated by the `7`, though the produced sound would be raised by an octave due to the octave alteration character, the `^`. The produced note would be equivalent in length to a dotted minum (`m.`) tied (`-`) to a dotted crotchet (`c.`). It would start at the fourth dynamic level (`4`) with a crescendo (`<`) to the sixth (`6`). It would also contain a single 'bonus' note flag, as seen in the `("bonus")` segment. Finally, the produced note would also produce an additional polytonal note, which is simply a minum at the third pitch degree (`3m`), as seen with the `{3m}` segment.

```
!.7^m.-c.4<6("bonus"){3m}
```

Piece Excerpt 3.2: A single symbol that contains most available segments

Note Type

The note type segment is a binary type and has only two values: primary or secondary. By default, if the note type segment is not included on a symbol, the produced note will have its `Type` set to `Primary`. If a music symbol is preceded by a single exclamation mark (`!`), its `Type` will be set to `Secondary`. The ability to alter the note type allows for

greater flexibility through the visual and audible distinction between groups of notes. If a custom note object has not been supplied (see *Supplying a Custom Note Object*), setting the note's `Type` to `Secondary` will make the note use the `SecondaryNoteMaterial` when it is generated (see *Altering Colour, Transparency, and Light Emission*). It will also activate the `SecondarySynthesiser` (see *The Synthesiser Types*), allowing the activation of a secondary timbre.

Articulations

An articulation alters the visual and audible characteristics of a note. They are the second segment of a music symbol's definition with two characters: a staccato represented by the period character (`.`), and an accent by the greater-than character (`>`). A note may have either one or both of these articulations with the inclusion of the articulations' characters as part of its symbol definition. These characters may appear in any order, but only one of each character will be considered; any repeated characters are considered superfluous and are ignored.

Staccato The addition of a staccato to a note halves its musical duration while retaining the note's full notated length. To notate a staccato, add a period character (`.`) to the articulations segment. A note with a staccato would have its time definition (see *Time*) and therefore its length halved, and the remaining length would be padded with rests. This behaviour positions the proceeding note as though it was notated without a staccato, matching the expectation of composers and performers familiar with traditional stave notation.

Notating a staccato could be considered a verbose indication since a composer could simply half the musical duration of the note and notate the appropriate amount of rest

manually. However, especially when multiple notes are staccato, the addition of a staccato greatly decreases visual complexity when reading and writing Virtual Music Shorthand Notation. This can be seen in Piece Excerpt 3.3 and Piece Excerpt 3.4, where the former is notated with staccato indications, and the latter without. The equivalent phrase in traditional stave notation is shown in Figure 3.2.

```
.3q5 .5q .6q .8q .5c .1q .3q
```

Piece Excerpt 3.3: A short phrase with staccato indications

```
3s5 0s 5s 0s 6s 0s 8s 0s 5q 0q 1s 0s 3s 0s
```

Piece Excerpt 3.4: A short phrase without staccato indications

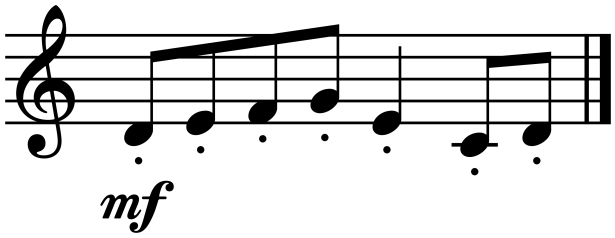


Figure 3.2: A short staccato phrase in traditional stave notation

Accent An accent increases the dynamic value of a note for a short period at the beginning of the note. It is notated by adding the greater-than character (`>`) to the articulations segment. The accent effects the note both visually and audibly: the note's shape is altered to reflect the accent, and if the Virtual Music Instrument is in `Simple` mode (see *The Performance Modes*), the amplitude will be increased.

The characteristics of an accent are effected by two fields of the `Note` class: `AccentPoint`, and `AccentStrength`. An accented note will begin accented and the effects of the accent will linearly decrease until it reaches a predefined point—the `AccentPoint`—at which the effects of the accent are removed. By default, `AccentPoint` is equal to `0.2`—20 per cent of the musical duration of the note. The `AccentStrength` is the multiplier which effects the maximum width of the accent and the maximum amplitude accordingly. By default, `AccentStrength` multiplies the amplitude by two minus the amplitude, making the effects of an accent greater at lower dynamic levels.

Pitch and Rests

The third segment, pitches and rests, is the first compulsory segment of a music symbol. The segment consists of a whole number or integer indicating the desired pitch degree beginning from `1`. A `0` may be used to indicate no pitch (a rest)—a gap between notes. Pitch degrees are indicated by a number rather than a letter (as a composer or musician familiar with Western music theory may expect) to support changes of Octave Division or Fundamental Frequency; a change of either from their default value would result in letter definitions not matching their counterpart in traditional stave notation.

The number of pitches available is equivalent to the Number of Octaves multiplied by the Octave Division, plus an additional pitch to make the final octave complete. This is demonstrated in Equation 3.2 which calculates the number of pitches as p where o is equal to the Number of Octaves and t is equal to the Octave Division.

$$p = (o \times t) + 1$$

Equation 3.2: Calculating the number of available pitches

The default values of `2` and `12` for the Number of Octave and Octave Division keys respectively produces twenty-five available tones: two complete semitone scales with the inclusion of the final octave interval. If the Fundamental Frequency has the default value of `261.63`, the first pitch degree, `1`, would be equal to C3, `2` would be equal C♯4, `3` D4, and so on, until reaching `25`, C5.

Once the Virtual Music Shorthand Notation notation has been converted to Virtual Music Notation for performance, notes with a higher pitch degree will produce a higher frequency and will be placed higher in the performance area than notes with a lower pitch degree. Notes with a pitch degree of `1` will be placed at the performer's minimum performance height (`VMPerformer.Preferences.MinimumHeight`). Similarly, the highest pitch degree of the current available pitch collection, by default `25`, will be placed at the performer's maximum performance height (`VMPerformer.Preferences.MaximumHeight`). Rests, indicated by a pitch degree equalling `0`, will not generate any visible notation and will appear instead as a gap between the surrounding notes.

Octave Alterations

An octave alteration either raises or lowers the octave of a note audibly. The octave can be raised with the caret character (`^`) and lowered with a lowercase v. Either of these characters can be repeated as many times as desired and the effect is compounded: one `^` would raise the octave of a note by one, and the notation of `vvv` would lower the note by three octaves. While either character may be repeated, only one of the two characters may be present in the octave alteration segment. The inclusion of an octave alteration also effects the reported frequency of a note at `Note.Frequency` accordingly.

```
12q4 12q 10q 8q 10m>2 3vvvm
```

Piece Excerpt 3.5: A phrase that contains an octave alteration

Piece Excerpt 3.5 demonstrates a piece that contains an octave alteration on its final note. If all fundamental frequency, number of octaves, and octave division settings are set to their defaults (see *Settings Blocks*), the third pitch degree as indicated in the pitch and rests segment of the final note would be equal to D4, `587.33f`. This frequency would then be multiplied by two to the power of the octave difference, negative three, as seen in Equation 3.3.

$$587.33 \times 2^{-3} = 73.41625$$

Equation 3.3: Calculating the frequency of a note with an octave difference

Visually, the note is altered to reflect the octave difference, but the note remains at the same height as though it was unaltered. If a piece's `NoteShapes` have not been altered at `Piece.GenerationSettings.NoteShapes`, notes altered up any number of octaves will appear to point upwards, indicating that the frequency that they will produce is higher than they indicate. Similarly, notes altered down any number of octaves will appear to point downwards, indicating that the frequency that they will produce is lower than they indicate.

Time

The time segment is the fifth segment of a music symbol. It is also the second compulsory segment and each symbol must have a time segment in its definition. The time segment consists of at least one single-notehead rhythmic value, where a single-notehead rhythmic value is rhythmic definition that is fully defined by a base rhythmic character and the

optional inclusion of a rhythmic dot and/or tuplet in precisely that order. This rhythmic value may then be joined to one other single-notehead rhythmic value by using a tie. This is similar to a singular notehead in traditional stave notation being lengthened by using a tie; if the tie is removed in either VIRTUAL MUSIC SHORTHAND NOTATION or traditional stave notation, the rhythmic value would still be fully defined and understandable.

A single-notehead rhythmic value must be defined first by the compulsory inclusion of a base rhythmic character. With the exception of the semibreve, the available characters for the base rhythmic values are the same as the first letter of their equivalent note values in British music terminology, ranging from a semiquaver (a sixteenth note in American music terminology), to a semibreve (a whole note). The available characters are:

- `s` (semiquaver, 𝅘𝅥𝅯)
- `q` (quaver, ♪)
- `c` (crotchet, ♩)
- `m` (minum, 𝅗𝅥)
- `b` (semibreve, 𝅝)

A period character (`.`) may be added after the base rhythmic value to increase its length by half, matching the convention of the rhythmic dot in Western musical notation. Finally, a tuplet may be added with square brackets surrounding the subdivision, where the subdivision is considered as the amount of times the single-notehead rhythm would fit where two of the same single-notehead rhythm would normally. Tuplets are currently limited to triplets only, leaving `[3]` as the only character option for a tuplet. This limitation will be removed in a later version, and the terminology surrounding triplets reflects this.

Finally, a tie character (`-`) may be used to join one single-notehead rhythmic value to the next. Currently, no more than one tie may be notated, but the tied value may include any or all of the characters possible for a single-notehead rhythmic value.

Dynamics

The dynamics segment consists of a single whole number (an integer) from `1` to `9` and an optional gradual dynamic level change. If a dynamic level is not specified, the most recent indication will be used, making the dynamics segment an optional segment. Notes with a higher dynamic value will be performed louder than notes with a lower dynamic value. The available dynamic values are listed with their equivalent markings in traditional stave notation in Table 3.1.

1	*ppp*
2	*pp*
3	*p*
4	*mp*
5	*mf*
6	*f*
7	*ff*
8	*fff*
9	*ffff*

Table 3.1: The dynamic values and their equivalent markings in traditional stave notation

Virtual Music Notation, once generated, will reflect their dynamic value with their size. If the piece's `FromDirection` is set to `Straight`, produced Virtual Music Notation will reflect the dynamic value in their width, with higher dynamic values

producing wider notes. For pieces with a `FromDirection` set to either `Left` or `Right`, how far the note reaches in front of the performer reflects their dynamic value instead.

The dynamics segment may also include a gradual dynamic level change similar to a crescendo or diminuendo in traditional stave notation called a 'hairpin'. Unless they have their own dynamics segment, notes following a hairpin will have the dynamic value that the hairpin finishes at. A hairpin may be notated with a `<` or `>` character respectively in the dynamics segment following the initial dynamic value and followed by the proceeding dynamic value. As hairpins last for one note only, if a hairpin character is included, the proceeding dynamic value must also be included. Hairpins begin immediately as the note starts with their first or initial dynamic value and end at their second dynamic value at the end of the note. The note will be adjusted visually and will either grow or shrink accordingly. As the initial dynamic value of a note is remembered from previous notes if none is supplied, hairpins do not require a preceding dynamic value. The dynamics segment may therefore begin with a hairpin character and the hairpin once generated will begin at the dynamic value of the notes before it.

Note Flags

A note flag is an identifier used to trigger functionality or to allow the later collection and identification of the note or multiple notes with a particular note flag. An unlimited number of note flags may be added in the note flag segment by surrounding a comma-delimited string of note flags with brackets. Note flags must also be individually surrounded by quotation mark characters (`"`). Note flags may be any case-sensitive alphanumeric string of characters. Importantly, as Virtual Music delimits Virtual Music Shorthand Notation by the space character, a note flag cannot contain a space and only a comma can be used as a delimiter for multiple note flags. A space

must not be inserted after the comma character.

Note flags give VIRTUAL MUSIC extensibility by allowing the implementation of custom functionality through the use of custom methods, typically run when notes with a note flag are activated, deactivated, finalised, or failed. The process to make use of note flags in such a manner is described in more detail, along with a complete example, in *Flagged Actions*. The process to collect any notes with a note flag is described in *Accessing Notes by their Note Flag*.

Polytonality

The final segment of a music symbol is the polytonality segment. The polytonality segment is the most complex segment of all segments of a music symbol as it contains an entirely new music symbol with entirely different segments. A polytonality segment is notated by a new music symbol surrounded by curly brackets (`{` and `}`). Music symbols defined within a polytonality segment have different restrictions; most importantly, only the time segment is compulsory, and the polytonality segment can not be defined again within a polytonality segment. The following ordered segments are available for music symbols within a polytonality segment:

1. (optional) note type
2. (optional) articulation
3. pitches and rests
4. (optional) octave alteration
5. (optional) time

6. (optional) dynamics

7. (optional) note flags

The music symbol in the polytonality segment and the note generated by that music symbol is considered the child music symbol and note respectively. Therefore, the music symbol that has the polytonality segment and the note generated by that music symbol are considered the parent music symbol and note. Child notes begin at the same rhythmic position as the parent note, and the child is numbered following the parent. For example if the parent note was the twelfth note in a stave, the child note would be the thirteenth, and following notes would increase from that number. This means that a composer must be careful to ensure that the time segment of the parent note does not equal a smaller value, as it may cause unintended behaviour at the finalisation of the stave and piece.

If a child music symbol contains no time segment, it will use the time segment from the parent music symbol. Similarly, if a child music symbol has no dynamics segment, it will use the dynamics segment of the parent, including any hairpins if they are defined. Dynamics segments of the parent music symbol are remembered for later notes, and if they are defined, dynamic values of child music symbols are isolated to that child music symbol.

While being the most complex segment due to the varied restrictions and necessary considerations, it allows the definition of a polytonality segment to be simply the definition of an additional pitch degree. `1m{13}`, for example, is a valid music symbol that states that an additional note at the thirteenth pitch degree should be generated with the same rhythmic and dynamic value as its parent. This behaviour makes such a music symbol similar to the notation of a double stop in violin and string music.

Defining a parent note as a rest with a smaller value in its time segment creates an

arpeggiation effect. Music symbols following the parent note will begin after the length defined in the parent note's time segment, and the child note will continue independently for the length in its time segment. This arpeggiation effect can then be repeated as many times as desired to create chords as seen in Piece Excerpt 3.6, which creates a four note chord arpeggiated by quavers, lasting a semibreve in total.

```
0q3{8vb} 0q{3c.-m} 0q{10c-m} 12q-m
```

Piece Excerpt 3.6: A phrase creates a chord via an arpeggiation effect

3.3 Piece

In Virtual Music, the term 'piece' is used to refer to the organised collection of musical instructions for the intention of performance. The term may also refer directly to the `Piece` class and, furthermore, to the concept of a composition using Virtual Music Notation. A piece during performance is an instance of and organisation of data within the `Piece` class. In this document, a piece may be differentiated from the class of the same name through the use of inline code blocks surrounding the term: `Piece`[1].

As with other sections within this *Developer Manual*, this section ranges from basic concepts to some advanced functionality of the `Piece` class. It describes common desired behaviours as well as some advanced potential use cases. In some instances, this section explains how the `Piece` class was used to create the compositions included with this thesis or theorises potential ways in which certain functions could be used.

The potential use and functionality of the `Piece` is not limited to what is described

[1]That is, when the `Piece` class is referred to and not the concept of a musical composition for Virtual Music, it will be presented in the same manner as other objects and items of code throughout this thesis.

here; with an in-depth understanding of the `Piece` class, a composer or developer may create or devise new functionality outside of what is described. See Appendix P for a detailed listing of all fields, properties, and methods for the `Piece` class.

3.3.1 Adding a Piece to a Scene

To add a piece to a scene, add the `Piece` component to a `GameObject`. This component is referred to as the piece and manages all functionality related to that piece and its performance. Adding a piece to a scene can be done in a number of ways, including dynamically through scripting, or by creating and adding all `GameObject`s and components manually. For basic usage or for applications only requiring a single piece, it is recommended to manually create and add all components, but this remains the developer's discretion.

To manually create and add a `GameObject` and `Piece` component to a scene:

1. Right-click in the *Hierarchy*.
2. Select *Create Empty*.
3. (Optional) Rename the newly created `GameObject`:
   (a) Right-click on the newly created `GameObject`.
   (b) Select *Rename*.
4. In the *Inspector* window, click the *Add Component* button.
5. Type 'Piece' and select the result.

This described method is the simplest way to create a new piece. Any process that results in the 'Piece.cs' script being added to a `GameObject` as a component is acceptable but will not be explicitly described. For example, using the `AddComponent` method of the `GameObject` class in the `UnityEngine` namespace would be acceptable. The process to do this will not be described, however, as such an implementation would require a specific knowledge and understanding of the `AddComponent` method.

3.3.2 Giving a Piece Notation

VIRTUAL MUSIC constructs its notation by parsing a string or multiple strings of VIRTUAL MUSIC SHORTHAND NOTATION known as 'shorthand staves' or more simply 'staves'[2]. These staves may be added to a piece by adjusting the `ShorthandStaves` field:

1. Find the piece and make sure it is visible in the *Inspector* window.
2. Find 'Shorthand Staves' and expand the item if necessary by clicking on the arrow to the left of it so that it points down.
3. Change the 'Size' to the amount of staves required for the piece.
4. Insert the staves into their respective elements, beginning from 'Element 0'[3].

Once a stave has been added to `ShorthandStaves`, it can be altered as necessary, but any changes will only be reflected once the piece has been (re)generated and (re)started. Consequently, changes to staves should not be made while a piece is being performed.

[2]The term 'stave' or 'staves' also applies to VIRTUAL MUSIC NOTATION generated from parsed VIRTUAL MUSIC SHORTHAND NOTATION into a new `Stave` class, but this distinction is clear in context and within the code. In the `Piece` class, shorthand staves are accessible through the `ShorthandStaves` field, and the `Stave`s are accessible through the `Staves` field.

[3]Arrays in C# are 0 indexed. This means that the first element of an array is indexed by the number 0. An array with `x` elements indexes from `0` to `x-1` (Microsoft, 'Arrays').

The appearance of staves in the *Unity Inspector* is currently limited to the default appearance of `string` arrays and can be cumbersome to use for fine adjustments to the Virtual Music Shorthand Notation. Because of this, if adjustments are required, it is recommended to do so in an external application that can handle plaintext. If adjustments to the stave should be required for the purposes of dynamic or procedural music, it is possible to achieve this by supplying methods to events of the *The Events System*. Events such as `Note.Finalised` or `Stave.Finalised` could be used to replace the stave of a piece and then trigger (re)generation via the `Generate` method.

Virtual Music Shorthand Notation has the ability to adjust tempo and other such performance-related data through Settings Blocks (see *Settings Blocks*). This means that in many cases, it is only necessary to add the shorthand staves to the piece to be ready for performance. No further adjustments are required as Virtual Music will provide adequate default `Material`s and `Synthesiser`s, but it is recommended to improve visual clarity of the notation by adjusting the default appearance (see *Changing the Appearance of Generated Notation*). Virtual Music will provide a warning during generation if it detects that its generation system is being used, but a `NoteMaterialPrimary` or `NoteMaterialSecondary` has not been provided. See *Altering Colour, Transparency, and Light Emission* for more information.

For the purposes of preservation, it is also possible to serialise and save the `Piece` class. This enables the preservation of music by allowing these saved files to be distributed through means of a file sharing platform or a theoretical Virtual Music composition sharing application and network, in which composers and performers could compose, share, and perform their compositions. See subsection 5.2.5 for further discussion and exploration of this topic.

3.3.3 Setting the Title Information

The piece's title, composer, or date of composition can be set by supplying values to the `Title`, `Composer`, and `DateOfComposition` fields, respectively. This information can be shown on an 'Information Slide' in the first stave of the piece's generated notation, seen in Figure 3.3.



Figure 3.3: The Information Slide

Each value can be shown or hidden on the Information Slide by altering the values in `ShowTitle`, `ShowComposer`, and `ShowDateOfComposition`. The entire information slide can also be hidden entirely by setting `ShowInformation` to `false`. Note that if `ShowInformation` is set to `false`, the `Title`, `Composer`, and `DateOfComposition` will not be shown, despite the values in `ShowTitle`, `ShowComposer`, and `ShowDateOfComposition` respectively, as the entire Information Slide is hidden. The location of the information slide may also be adjusted away from its default. See *Overriding the Location of the Information Slide*.

Most of these values are serialisable and therefore able to be altered either in code or through the *Inspector*, but some values cannot be shown due to limitations of the *Inspector* or by design (Unity Technologies, 'Script Serialization'). For example, the *Inspector* is not able to display `DateTime` values, which makes `Piece.DateOfComposition`, a `DateTime` field, only alterable by script. Code Sample 3.1 provides a script that uses *Unity*'s `Start()` method to set the date of composition to 22 August 2020. This limitation to `DateTime` explains the reasoning for `DateOfComposition`'s default value of `false`.

```
using System;
using UnityEngine;
using VirtualMusic.Notation;

public class SetTheDateOfComposition : MonoBehaviour
{
  [SerializeField] private Piece piece;

  private void Start()
  {
    piece.DateOfComposition = new DateTime(year: 2020, month: 8, day: 22);
  }
}
```

Code Sample 3.1: Setting the date of composition

The ability to show and store the title information of a piece was added in `v0.2.0` in order to facilitate the preservation of compositions created using Virtual Music (see subsection 2.4.1). It also facilitates the sharing and distribution of compositions by allowing referential information to be embedded into a file which can then be shared or stored in an external database.

3.3.4 Changing the Piece's Direction

All objects in a piece's staves, including notation, the Information Slide, or any other `StaveObject`, can approach the performer from three different directions. All such

objects approach the performer from the same direction. The direction that these objects approach the performer from must be set before the piece is generated, and this direction cannot be changed once the piece has been generated. To change the direction, alter the value in `FromDirection` to reflect from which direction the stave and its objects should approach the performer.

The ability to change the direction of the stave was added in `v0.1.1` (see subsection 2.3.2). While the default `Straight` should be suitable for most situations and is recommended, pieces with frequent arpeggiation may benefit from approaching the performer from a sideways direction as interval leaps and rhythmic values are more clearly visible. In these cases, `Right` is recommended, emulating a piano roll as seen in most digital audio workstation software.

3.3.5 Altering the Location of the First Note

The location of the first note of any stave is measured by the number of beats or crotchets in the first tempo of a piece. The number of beats is stored in the `BeatsForCountIn` field and can be edited by the *Inspector* or by a script. The default setting of `8` means that the first note will reach the performer and be playable eight beats after the piece has been started. For example, if the beginning of a piece has a tempo indication of sixty beats per minute, the first note will be playable after eight beats, equivalent to eight seconds.

It may be desirable to change the location of the first note by changing the value of `BeatsForCountIn` away from its default. For example, if a performance environment contains interesting visual elements, increasing `BeatsForCountIn` to a higher value may assist in immersing the performer in the environment. Alternatively, to start a piece quickly, the value may be decreased or changed to `0`. The latter case may be desirable

in combination with *The Events System* to quickly change to a different piece, creating a pseudo-generative or game-like 'choose your adventure' type piece.

`BeatsForCountIn` should, in most situations, be a positive number, but it is possible to set it to a negative number. In this case, the piece will appear to the performer as though it has already started. In the previously mentioned 'choose your adventure' piece, this may be desirable to create the illusion that the piece is altering itself as, if the performer were to look at the notation that has already passed, it will appear similar to what had just been performed.

3.3.6 Altering the Dynamics

By supplying `GenerationSettings` with a new set of dynamic values, it is possible to adjust the dynamic level for a piece both visually and audibly. This could be used for a variety of reasons, ranging from practical purposes such as 'flattening' all dynamic values to make the piece entirely the same, to creating nonsensical variations to dynamics for an interesting effect. Flattening the dynamic values could be desired in a pedagogical situation to allow the performer to focus on producing the correct pitches and movements related to pitch without becoming overwhelmed, distracted, or confused by dynamic variation.

To adjust the dynamic values, a new `Dictionary` must be created with keys from `1` to `9` matching the available shorthand dynamics options (see *Dynamics*), and with values not lower than `0` or higher than `1`. The `GenerationSettings.Dynamics` field may then be replaced with the newly created `Dictionary` containing the dynamic values, as demonstrated in Code Sample 3.2.

```
Dictionary<int, float> newDynamics = new Dictionary<int, float>
{
  { 1, 0.10f },
  { 2, 0.20f },
  { 3, 0.30f },
  { 4, 0.40f },
  { 5, 0.50f },
  { 6, 0.60f },
  { 7, 0.75f },
  { 8, 0.90f },
  { 9, 1.00f }
};

piece.GenerationSettings.Dynamics = newDynamics;
```

Code Sample 3.2: Changing a piece's dynamics

These dynamic values will then be used when the piece is next generated. They will affect the audio produced as the visual representation of the dynamics. For further adjustment of the visual representation of dynamics, see *Altering the Note Widths and Heights*. Importantly, like other changes that effect generation, changes to these values will not take place while a piece is being performed and will only be reflected in the notation once it has been (re)generated and (re)started.

3.3.7 Altering the Note Lengths

By supplying `GenerationSettings` with a new set of rhythmic values, it is possible to adjust the duration of all notes generated as part of a piece. Doing so may be desirable in a pedagogical situation, for example, so that a performer can isolate difficult intervallic passages and have all notes represented with the same rhythm. Adjusting the rhythmic lengths may also be used to halve or adjust the tempo of an entire piece without altering the piece's shorthand notation, for example, if a piece has multiple complicated tempo changes. As with *Altering the Dynamics*, to adjust a piece's default rhythmic duration values, replace `GenerationSettings.Lengths` with a new `Dictionary` containing rhythmic values sorted by their respective shorthand as keys (see *Time*).

```
Dictionary<char, float> newLengths = new Dictionary<char, float>
{
  { 's', 0.25f },
  { 'q', 0.5f },
  { 'c', 1f },
  { 'm', 2f },
  { 'b', 4f }
};

piece.GenerationSettings.Lengths = newLengths;
```

Code Sample 3.3: Changing a piece's default rhythmic values

By default, these values are reflective of their relationship to a single beat. This means that a semiquaver—referred to by `s`—is equal to 0.25 crotchets, and a semibreve (`b`) is equal to 4 crotchets. Changes to these values will come into effect once the piece has been (re)generated and (re)started.

3.3.8 Changing the Appearance of Generated Notation

Altering Colour, Transparency, and Light Emission

To improve visual clarity, it is recommended to enable note transparency and light emission. Similarly, if using multiple synthesiser types, it is recommended to distinguish between the two voices by supplying a `Material` in `NoteMaterialPrimary` and an altered or entirely new `Material` to use for the second type in `NoteMaterialSecondary` (see *The Note Types* for more information on other differences between the note types). Since transparency and emission is recommended, `NoteTransparency` and `NoteEmission` are set to `true` by default; however, the rendering mode of any supplied `Material` must be changed from the default `Standard` to `Transparent` and `Emission` must be enabled and configured with appropriate

settings. The colour of notes will remain the colour set in the `Material` for each type and can therefore be adjusted by altering the colour values in these materials.

There are four states or situations that will impact transparency and emission levels: when a note is inactive (not being touched), when a note is active (being touched), when a note is at the same height as the Virtual Music Instrument, and when a note is finalised (past the playable area or unplayable). The amount of transparency for these states or situations can be adjusted by altering the values in `NoteTransparencyWhenInactive`, `NoteTransparencyWhenActive`, `NoteTransparencyWhenInstrumentAtSameHeight`, and `NoteTransparencyWhenFinalised`, respectively. Similarly, the amount of emission for these states or situations can be adjusted by altering the values in `NoteEmissionWhenInactive`, `NoteEmissionWhenActive`, `NoteEmissionWhenInstrumentAtSameHeight`, and `NoteEmissionWhenFinalised`, respectively. These situations offer visual feedback to the performer, so that they may see when a note is inactive or active, when it is no longer playable, and when it is at the same height as the Virtual Music Instrument and so will be correctly activated when it reaches the playable area.

In situations where notes should be transparent or emit light but no such visual feedback is desired, simply set the desired transparency and emission settings in the supplied `Material` and set `NoteTransparency` or `NoteEmission` to `false`. Generated notes will then uniformly retain transparency levels or emit the same level of light in all states and Virtual Music will not control or adjust these levels.

Once generated, any of these values cannot be changed on a piece-wide basis and can only be changed for each note individually. This is because each note individually calculates the appropriate values when it is generated, saving valuable computing resources during performances.

Supplying Custom NoteShapes

Notes generated using the inbuilt generation system are procedurally generated based on values defined in the `INoteShape` interface. Specifically, a `Calculate` method takes in the required height, the starting and ending width, and an optional accent width parameter to calculate `Vertices` based on the shape's `Triangles`. Currently, these shapes are used to visually reflect alterations to a note's pitch and are thus `Standard`, `OctaveUp`, and `OctaveDown`, but these shapes could also be used to indicate advanced techniques such as vibrato, glissandi, or other indications inviting live interpretation.

To alter one of these shapes, adjust the `NoteShapes` in `GenerationSettings`. A default set is provided under `VirtualMusic.Notation.Note.DefaultNoteShapes`, with the default `Standard`, `OctaveUp`, and `OctaveDown` note shapes set to their provided defaults found at `Note.DefaultShape.Standard`, `Note.DefaultShape.OctaveUp`, and `Note.DefaultShape.OctaveDown`, respectively. Altering these shapes requires an understanding of mesh generation within *Unity*. See *Procedural Mesh Generation* in the *Unity* documentation for more information (Unity Technologies, 'Procedural Mesh Generation').

Supplying a Custom Note Object

It is possible to make a piece's notation instantiate a supplied `GameObject` in place of the generated meshes. This could be implemented in advanced scenarios to create complex visual notation, for example to make use of animations or a mesh created with an external application such as *Blender*. If a custom object is supplied, some features such as crescendo and diminuendo will not be visually represented and only the first dynamic value will be retained.

To supply a custom note object to be used in place of the inbuilt generation system, set `NoteObject` to a `GameObject` that contains a `RigidBody` component. This can be achieved in code or via the *Inspector*. The `isKinematic` and `useGravity` fields of the `RigidBody` component will be set to `true` and `false` respectively as they are required by VIRTUAL MUSIC to operate correctly. The supplied `NoteObject` will then be used during generation and, as such, if a `NoteObject` is supplied after a piece has already been generated, the piece must be regenerated and restarted.

This feature was the original method through which the notation would be generated and was superseded by the mesh generation system in `v0.1.4` in order to allow hairpins (see subsection 2.3.5). The functionality was retained with a limited feature set in order to allow for custom and complex interaction and visual notation. If a `NoteObject` is provided, inbuilt mesh generation does not take place and any custom note shapes (see *Supplying Custom NoteShapes*) are ignored.

Transparency, emission, and similar visual distinction features still function correctly when supplying a custom note object. See *Altering Colour, Transparency, and Light Emission* for more information.

Altering the Note Widths and Heights

The width and height of generated notes can be adjusted by altering the values in `NoteWidthModifier` and `NoteHeightOverride` respectively. These adjustments are purely visual and do not affect any audio produced—they are designed to be used for greater visual distinction or to create a particular effect.

The value in `NoteWidthModifier` is used to modify the values provided in

`GenerationSettings.Dynamics`. This value is set to `0.8f` by default. This feature was introduced in `v0.1.4` because notes generated at extreme dynamic values were too wide and in some cases were unintelligible since they could stretch beyond the field of view or could obscure other notes (see subsection 2.3.5).

The generated notation will have a note height according to the value returned by the `GetNoteObjectHeight` method in the `Environment` class. This method returns the correct note height at any point in a piece and also considers any overrides (see *Retrieve the Note Object Height*). This means that if `NoteHeightOverride` is supplied, that value will be used regardless of the situation. Therefore, considerations must be made regarding any possible occlusion or the likelihood of spacing being too great. In these situations, the chances of the notation being unintelligible, misinterpreted, or missed entirely, is increased significantly.

Adding a Particle Effect

A particle effect may be added to Virtual Music Notation to add a unique visual flair by creating and expelling particles of light or any other type of particle from notes. Virtual Music has compatibility with *Unity*'s particle system and can create particles when a note is being activated, and stops such particle creation when the note is no longer being activated. For more information on what particle types are available and how to create them, consult *Particle systems* in the *Unity* documentation (Unity Technologies, 'Particle systems').

If such an effect is desired, it is first necessary to add a particle system to each note by *Giving All Notes Custom Components* or by *Supplying a Custom Note Object* that has a particle system component already on it. Only one particle system per note is

controllable; if more than one particle system is found, only the first will be used and others will be ignored. During the generation phase, VIRTUAL MUSIC will detect if a note has a particle system component added to it and will stop it automatically before the piece begins, ensuring that the particle system is only started upon note activation.

Setting `ControlParticleSystem` to `true` will then enable VIRTUAL MUSIC to control the included particle systems. Particle effects will start on note activation and will stop on note deactivation. The value of `ControlParticleSystem` can be altered either in the *Inspector* window or through custom scripts and can be altered at any time, including during performance. `ControlParticleSystem` is set to `true` by default to simplify this process, as it is assumed that if a particle system has been added to the note it is intended for use as a general visual effect.

However, it may be desirable to set `ControlParticleSystem` to `false` before the piece generation has occurred or the piece has been started. For example, the value of `ControlParticleSystem` could be altered during performance in a gaming scenario to create a special 'bonus' mode similar to the *Star Power* feature included in many games of the *Guitar Hero* series 'to provide a little more depth to the game—some replay value, some interest for people as they were playing beyond just hitting the notes' (Iain). In many of the games in the *Guitar Hero* series, when the *Star Power* feature is activated, a score multiplier is added, among other positive effects, and the player is informed of the activation of the feature visually by means of material and colour changes as well as particle and sprite effects (Harmonix, Guitar Hero). To create a similar effect using VIRTUAL MUSIC, upon the activation of this bonus mode set `ControlParticleSystem` to `true`. This could be combined with any number of other alterations to the environment, for example raising the volume of an audience sound effect, and gives feedback to the performer, informing them that the mode has been correctly enabled and that its effects are in play.

3.3.9 Giving All Notes Custom Components

Since Virtual Music Notation is usually generated using an internal mesh generation system with no input required by the developer or composer, it may be desirable to add additional components to every note without the limitations when supplying a custom note object (see *Supplying a Custom Note Object* for more information). For example, to provide a particle system to each note (see *Adding a Particle Effect*), a developer must either supply a custom note object or use the `CustomNoteComponents` field, which is used by the `AddCustomComponents` method during the generation of a piece's notation. The latter method retains the use of the internal mesh generation system while still allowing the developer to adjust the appearance and behaviour of the note.

Items of the `CustomNoteComponents List` are intended to be simple methods which add components to the note's `GameObject`. These methods should accept a `GameObject` as a parameter and add the new component to the passed `GameObject`, most often done using the `AddComponent` method of the `GameObject` class (Unity Technologies, 'GameObject.AddComponent'). The passed `GameObject` is the `GameObject` which contains the `Note` component, and should be considered as a direct representation of the note in question, as any adjustments to the `GameObject` will directly effect the note. During the piece's generation, `AddCustomNoteComponents` is called on each note, which iterates through each item in the `CustomNoteComponents List`. It should be noted that using these methods incorrectly could easily break functionality of Virtual Music, as access to a note's `GameObject` gives access to all properties of the note such as its name and its position. Adding custom components should be done with care to ensure that the methods invoked only supply custom components. Code Sample 3.4 shows one way that `CustomNoteComponents` can be used to add a basic particle system to the `GameObject` or note in question.

```
using System;
using UnityEngine;
using VirtualMusic.Notation;

public class AddParticleSystem : MonoBehaviour
{
  [SerializeField] private Piece piece;

  private void AddParticleSystem(GameObject other)
  {
    ParticleSystem particleSystem = other.AddComponent<ParticleSystem>();

    // set up the new particle system
    particleSystem.main.startSize = 0.02f;
    particleSystem.main.startSpeed = 0.85f;
    // etc...
  }

  private void Start()
  {
    piece.CustomNoteComponents.Add(AddParticleSystem);
  }
}
```

Code Sample 3.4: Adding a basic particle system by using CustomNoteComponents

In advanced scenarios or for specific requirements, the `AddCustomNoteComponents` method of a different piece may be used. This can be done by taking advantage of the fact that an item of the `CustomNoteComponents` field is simply a method that passes `GameObject` containing the note as a parameter. A method therefore can be added to the `CustomNoteComponents` field which refers to the `AddCustomNoteComponents` method of a different piece, as seen in Code Sample 3.5. This may be desirable in a pseudo-procedural generation or 'choose your adventure' type of piece to ensure that all notes throughout the performance are consistent in their included components while reducing code complexity.

3.3.10 Overriding the Location of the Information Slide

If it is being shown, the information slide, by default, positions itself at the halfway point between the performer and the start of the first note. If the position of the first note has not been changed, this equates to four beats or crotchets compared to the eight beat

```
using System;
using UnityEngine;
using VirtualMusic.Notation;

public class AddCustomComponentsFromSuppliedPiece : MonoBehaviour
{
  // (optional) this piece will be compared to the finalised piece
  [SerializeField] private Piece pieceToUseMethodFrom;
  [SerializeField] private Piece originalPiece;

  private void AddComponents(GameObject other)
  {
    // note that this method forgoes any error checking, so
    // it is recommended to first check if "other" has a Note
    // component on it first
    pieceToUseMethodFrom.AddCustomNoteComponents(other.GetComponent<Note>());
  }

  private void Start()
  {
    originalPiece.CustomNoteComponents.Add(AddComponents);
  }
}
```

Code Sample 3.5: Using the AddCustomNoteComponents method of a different piece

position of the first note. The default position of the information slide may be overridden by a position measured in beats or crotchets. To override the position of the information slide, provide `InformationSlideBeatPositionOverride` with a value that is not `null`. To remove any override, change the value of `InformationSlideBeatPositionOverride` to `null`. This value should, in most situations, be positive, but may also be a negative number for similar reasons as to those discussed in *Altering the Location of the First Note*.

If the position of the information slide is further than the first note, that is, if `InformationSlideBeatPositionOverride` is a value higher than `BeatsForCountIn`, the information slide will position itself so that it does not disrupt the visibility of the notation. If `FromDirection` is set to `Straight`, it will position itself on the opposite side to any tempo markings. Otherwise, if `FromDirection` is set to either `Left` or `Right`, it will position itself 1.2 game-world units—equivalent to 1.2 metres in the real world—above the performer's minimum reachable height.

3.3.11 Prohibiting Automatic Generation

By default, VIRTUAL MUSIC will generate any piece in the scene automatically via *Unity*'s `Start()` method. This is useful in most situations: if a piece is in a scene, it is assumed that it is intended for immediate performance with the values provided in the *Inspector*. Importantly, this helps remove the requirement of any knowledge of code in order to perform, compose, or develop using VIRTUAL MUSIC.

To adjust fields not visible in *Unity*'s inspector, such as date of composition (see *Setting the Title Information*), or to make use of any advanced functionality that requires initial configuration before a piece begins (see *Triggering Custom Functionality*), the automatic generation can be prohibited by setting `GenerateOnStart` to false. This may also be used in other situations as well, for example in a booth at a convention that will have multiple and frequent changes of performer. In this situation, prohibiting automatic generation would be preferred so that the frequently changing performers can first set their minimum and maximum height. It would also allow the performer to trigger the generation and start of a piece once they are ready to start, or allow them to choose a piece from multiple provided to them in some sort of GUI.

3.3.12 Detecting When a Piece Has Finished

VIRTUAL MUSIC includes an event emission system as one method to trigger custom functionality (see *The Events System*). To detect when a piece is finished (when the final note of all staves is finalised), the `Finalised` event may be used. For example, a performer may be taken back to a menu so that they may choose another piece to perform, or they may be shown statistics on their performance such as their accuracy and

how many notes that they missed. To achieve this, a developer could supply a method to the `Finalised` event, as seen in Code Sample 3.6.

```
using System;
using UnityEngine;
using VirtualMusic.Notation;

public class DetectFinalisation : MonoBehaviour
{
  // (optional) this piece will be compared to the finalised piece
  [SerializeField] private Piece pieceToCheckFor;

  private void CheckFinalisedPiece(Piece finalisedPiece) {
    // (optional) if the finalised piece is not the one we want, exit the method
    if (finalisedPiece != pieceToCheckFor)
    {
      return;
    }

    // destroy the generated notation and then display statistics about that piece
    //
    // GameManager is used here as an example singleton game manager that
    // has the ability to display menus about a piece
    piece.TriggerDestroy();
    GameManager.DisplayStatisticsAbout(finalisedPiece);

    // the GameManager can also take the performer back to the selection screen
    GameManager.DisplaySelectionScreen();
  }

  private void OnEnable()
  {
    // provide a method to the static event
    Piece.Finalised += CheckFinalisedPiece;
  }

  private void OnDisable()
  {
    // remove the method when this GameObject is disabled
    Piece.Finalised -= CheckFinalisedPiece;
  }
}
```

Code Sample 3.6: Supplying a method to a piece's finalised event

3.3.13 Destroying a Piece When Finished

By default, the notation generated for a piece will remain in the scene indefinitely after the piece has been finished; Virtual Music provides methods to detect when a piece has been finished (see *Detecting When a Piece Has Finished*), but the implementation of

these methods is the responsibility of the developer. A `DestroyOnFinalise` field has been provided which will simply destroy the `PieceObject` when a piece has been finalised or finished. That is, if `DestroyOnFinalise` is set to `true`, once a piece has triggered its `Finalised` event, the notation, Settings Blocks, and musical instructions for the piece generated from the shorthand notation will automatically be removed. `DestroyOnFinalise` is set to `false` by default.

This feature was added in `v0.3.0` in order to simplify the process to remove a piece from a scene (see subsection 2.5.1). The addition of this feature assists in making VIRTUAL MUSIC accessible to developers without any knowledge of code.

3.3.14 Triggering Custom Functionality

A developer or composer may implement custom functionality in a piece through a variety of different methods. In particular, VIRTUAL MUSIC supplies three main methods which allow such implementation: pre/post actions, flagged actions, and the events system (see *The Events System*).

Pre/post Actions

The `Piece` class contains various 'pre' or 'post' time-based actions for common and repeated triggers relating to notes. These triggers include the generation, activation, deactivation, and finalisation of notes. Activation, deactivation, and finalisation each have a respective 'pre' and 'post' field, whereas generation only has a 'post' field as the `Note` component is created during the generation process and does not yet exist. These fields are titled with the convention of `(time)(trigger)Actions`. Each bracketed section

is replaced with the desired time (either 'pre' or 'post') and trigger (generation, activation, deactivation, or finalisation). The time marking is indicative of when the methods in the `List` will be invoked in respect to VIRTUAL MUSIC's note activation code. Each of these actions calls any methods in the appropriate field and supplies the note that triggered that action to those methods. These triggers are non-selective and methods in these fields will be invoked on every note in a piece.

For example, when a note is first touched, it will first trigger the `PreActivationActions` methods and will call each method in the respective `List`, supplying itself as the parameter. When the methods in `PreActivationActions` are invoked, the note has not yet been marked as `Active` as VIRTUAL MUSIC's activation code has not yet been run. It will then proceed through the activation process as normal, marking the note as `Active` before triggering the `PostActivationActions` methods once the activation process has been completed. Time-based triggers are related only to their respective trigger; `PreActivationActions` and `PostActivationActions` both occur during the activation of a note. `PostActivationActions` must not be confused with `PostDeactivationActions`, for example, which are triggered upon the *deactivation* of a note.

The available time-based actions are therefore:

- `PreActivationActions`
- `PostActivationActions`
- `PreDeactivationActions`
- `PostDeactivationActions`
- `PreFinalisationActions`
- `PostFinalisationActions`

- `PostGenerationActions`

Implementing functionality using these time-based actions is similar to using *Supplying a Custom Note Object*: a method is added to the respective field which accepts a `Note` as a parameter. For example, the `PostActivationActions` field could be used to change the colour of the note to a random colour when it is activated, overriding any transparency and colour values set as described in *Altering Colour, Transparency, and Light Emission*, seen in Code Sample 3.7[4]. The note will then return to its default colour upon deactivation, but the `SetRandomColour` method could also be supplied to `PostDeactivationActions` as well, overriding any transparency and colour settings for deactivation.

```
using System;
using UnityEngine;
using VirtualMusic.Notation;

public class ApplyARandomColour : MonoBehaviour
{
  [SerializeField] private Piece piece;

  private void SetRandomColour(Note note)
  {
    note.GetComponent<Renderer>().material.color = new Color(Random.Range(0f, 1f),
        Random.Range(0f, 1f), Random.Range(0f, 1f), Random.Range(0.25f, 1f));
  }

  private void Start()
  {
    piece.PostActivationActions.Add(SetRandomColour);
  }
}
```

Code Sample 3.7: Setting a note to a random colour upon activation by using PostActivationActions

[4]The incorrect and inconsistent spelling of 'colour' is due to the fact that classes in *Unity* are named in American English.

Flagged Actions

Flagged actions are actions or methods invoked on a note that has an exactly matching note flag[5]. Flagged actions can be invoked on four different triggering events: the activation, deactivation, finalisation, or failure of a note that has a note flag. Methods for these four triggering events are stored in `Dictionary` objects at `FlaggedActivationActions`, `FlaggedDeactivationActions`, `FlaggedFinalisationActions`, and `FlaggedFailActions`, respectively.

Each triggering event retrieves a single method to invoke from its relevant collection of methods by its note flag and sends the triggering note as a parameter to the retrieved method. That is, the note flag is used as the key to retrieve the respective value or method from the `Dictionary` of the appropriate trigger event, which stores methods as its values retrievable by the note flag as the key. This allows different and unique methods to be invoked on each of these triggering events, and also allows option of not invoking any methods on some triggering events if the developer or composer chooses to do so. However, this approach also limits the number of methods per note flag and triggering event combined to one only. This limitation is due to intentional functionality of the `Dictionary` class—an inbuilt class of the `System.Collections.Generic` *.NET* namespace—that prohibits multiple identical keys (Microsoft, 'Dictionary<TKey,TValue> Class').

To store a flagged action method, add a new entry to the relevant `Dictionary` that has the exact note flag as the key, with the value of a method accepting a `Note` object as the only parameter. In a gaming scenario, for example, it may be desirable to activate a special bonus mode at certain points during the performance if the performer successfully activates a particular note. This can be achieved by adding a 'bonus' note flag to the

[5]Any number of note flags can be added to a note with *Shorthand Notation*. For more information on note flags and how to include flags on a note, see *Note Flags*.

note which will trigger the flagged event, as seen in Piece Excerpt 3.7.

```
8b2<5 20s 20s 20s 20s 20q 20s 20s 20s 20q. 20s("bonus")
```

Piece Excerpt 3.7: An excerpt of a piece that contains a note with a 'bonus' note flag

Then, a method must be supplied to the desired flagged action collection. In this case, since the 'bonus' mode should be activated upon the successful activation of the note, a method will be added to the `FlaggedActivationActions` collection, with 'bonus' as the key. Adding a different method to the `FlaggedFailActions` collection with the same 'bonus' key may also produce an interesting result, for example to give the performer feedback of their failure with an audio cue. An example of the implementation of these two flagged actions can be seen in Code Sample 3.8.

```
using System;
using UnityEngine;
using VirtualMusic.Notation;

public class BonusModeActivator : MonoBehaviour
{
  [SerializeField] private Piece piece;

  private void Activate(Note note)
  {
    Debug.Log($"{note.name} was activated! Activating bonus mode...");

    // GameManager is used here as an example singleton that can activate the bonus mode
    GameManager.bonus.Activate();
  }

  private void Failure(Note note)
  {
    Debug.Log($"The performer failed to activate {note.name}. Playing the failure sound...");

    // AudioManager is another example singleton with the ability to play sounds
    AudioManager.PlaySound("bonusmode_failure");
  }

  private void Start()
  {
    piece.FlaggedActivationActions.Add("bonus", Activate);
    piece.FlaggedFailActions.Add("bonus", Failure);
  }
}
```

Code Sample 3.8: Utilising the 'bonus' note flag

3.3.15 Setting the Piece's Synthesisers

The sound of the Virtual Music Instrument is defined by the available synthesisers. These synthesisers can be set on a per-piece or global basis by changing the `PrimarySynthesiser` or `SecondarySynthesiser` fields on the `Piece` class, or the similarly named fields on the `VMManager`, to an alternative either created (see *Creating a New Synthesiser*) or supplied. This may be done in the *Inspector* or in code. For more information on supplying a `Synthesiser` for use globally, see *Overriding the Default Synthesisers*. If a `Synthesiser` component is supplied to either of these fields, that `Synthesiser` component will be used instead of any default or globally overridden component.

The `PrimarySynthesiser` and `SecondarySynthesiser` components may also be changed during performance by using the *The Events System* or *Flagged Actions*, theoretically removing the limit of only two timbres per piece. For example, *Étude 8* utilised flagged actions in order to change the secondary synthesiser between one that added a fifth interval above every note, and one that included a vibrato effect through the use of an LFO. This approach could also be used in a piece to allow the performer the choice of how they want the piece to sound. The developer or composer must only create a script that changes the `PrimarySynthesiser` or `SecondarySynthesiser` when a particular note is touched through use of a Note Flag or other event. See *Triggering Custom Functionality* and *The Events System* for more information.

3.3.16 Accessing Notes by their Note Flag

Notes with a note flag are collected and stored in a `Dictionary` collection at `FlaggedNotes`. A collection of notes that have a particular note flag may be obtained by using the precise note flag as the key for the `Dictionary`, which will then return a `List` containing all notes with that note flag. The list may then be used in any manner to implement custom functionality, for example by enumerating through the list and marking them as unplayable as demonstrated in Code Sample 3.9. See *Prohibiting a Note from Being Played* for more information.

```
if (VMManager.Piece != null)
{
  string theNoteFlag = "aNoteFlag";
  var notesWithTheNoteFlag = VMManager.Piece.FlaggedNotes[theNoteFlag];

  foreach (var note in notesWithTheFlag)
  {
    note.OverrideAsUnplayable = true;
  }
}
```

Code Sample 3.9: Accessing notes by their note flag and enumerating through them

3.3.17 Accessing Notes or Staves Directly

Once generated, the staves of a piece can be accessed through the `Staves` field, which provides a `List` of `Stave`s. This allows for a variety of functionality and can, for example, be used to obtain the length of a stave in game-world units, the total number of objects loaded in that stave, or importantly, the notes within that stave. For more information, refer to *Retrieving Stave Objects from a Stave*.

3.4 Note

In Virtual Music, a note is an instance of the `Note` class and is the representation of a musical sound. It represents the definition of pitch and duration and can be interpreted precisely by the Virtual Music Instrument, or in an improvised fashion by a performer translating its meaning to a real-world or other digital instrument. It is equivalent to the musical note or notated musical symbol representing duration and pitch in traditional stave notation, and similarly can represent other elements of traditional music notation such as articulation and ornamentation.

In most cases, once generated, a note usually maintains required functionality itself and no developer input is required. In fact, changing most fields or properties on a note once it has been generated is discouraged as it can easily cause unintentional or breaking changes. The `Note` class—which represents an object on a stave that can be activated by the Virtual Music Instrument—is an extension of the `Activatable Stave Object` class. It is also extensible and contains fields, methods, and properties that may be replaced by classes that inherit it. A detailed description of all fields, methods, and properties of the `Note` class, including those that are inherited, can be found at Appendix Q.

3.4.1 The Note Types

The note type, or rather more accurately, the note's synthesiser type, indicates the material and synthesiser it will use. The term 'note type' and 'synthesiser type' may be used interchangeably, as they both refer to the `Synthesiser.Type` `enum` object, but are applied differently in different classes. For more information on `Synthesiser.Type`,

refer to *The Synthesiser Types*. The note's current type may be accessed by the `Type` field, and this field is set during generation according to the type segment in the note's *Shorthand Notation*.

The default type, `Primary`, produces a note with the default appearance as it has used the `PrimaryNoteMaterial` during its generation. The `Secondary` note type similarly uses the `SecondaryNoteMaterial` during its generation. This difference may be used to create a visual distinction between the different note types. Likewise, notes with a `Primary` type will activate or indicate the use of the `Primary` synthesiser, and the `Secondary` type will activate or indicate the use of the `Secondary` synthesiser, accordingly.

The appearance of both types may be adjusted by supplying the piece with an appropriate `Material` in `Piece.NoteMaterialPrimary` and `Piece.NoteMaterialSecondary`. This process is described in *Altering Colour, Transparency, and Light Emission*. Similarly, the process to alter the activated or indicated synthesiser for each type is described in *Setting the Piece's Synthesisers* on a per-piece basis and *Overriding the Default Synthesisers* on a global basis.

3.4.2 Retrieving Information about the Note

Information about the note, including its frequency, pitch degree, dynamic value, and even the amount of times the note has been activated, can be accessed through various fields, properties, and methods of the `Note` class. See Appendix Q for a detailed list of all members of the `Note` class.

3.4.3 Implementing Custom Functionality

Custom functionality for notes may be implemented via three methods: by creating a new class that inherits the `Note` class and makes use of the included `virtual` methods intended for overriding[6], with custom supplied components supplied by *Giving All Notes Custom Components* or by *Supplying a Custom Note Object* that contains scripts that trigger custom functionality, or finally by using *The Events System*.

3.4.4 Prohibiting a Note from Being Played

A developer may prohibit a note from being played by setting `OverrideAsUnplayable` to `true`. Upon doing so, the note will be deactivated if it is currently activated, and will not be able to be (re)activated until `OverrideAsUnplayable` is returned to false. The note will continue to operate as normal, ignoring any touch inputs, and will continue to emit events as normal (see *The Events System*. Overriding a note as unplayable may be desired in a game-like scenario in which a group of notes would be overridden as unplayable in order to prevent the player's progression. This scenario is demonstrated in Code Sample 3.9, which shows how a developer may enumerate through a group of notes flagged with a particular note flag and overrides them as unplayable.

It should be noted that this option only effects performance modes other than `Advanced`. This is as in the `Advanced` performance mode, in which a user interprets the notation with the Virtual Music Instrument and the Virtual Music Instrument does not rely on the activation and deactivation of notes in order to produce any tones. For more information on the performance modes, see *The*

[6]These include the `PreStart`, `PostStart`, `PreUpdate`, `PostUpdate`, `PreOnDestroy`, `PostOnDestroy`, `PreOnDisable`, and `PostOnDisable` methods.

Performance Modes.

3.4.5 Retrieving Notes from a Stave

As the `Note` class inherits from the `Stave Object` class, all instances of the `Note` class are grouped into a containing list on the `Stave` class. This allows the retrieval of all notes in a stave through the `GetObjectsOfType<T>` method on the `Stave` class, or `ContainingList` field on the `Note` class. Refer to *Retrieving Stave Objects from a Stave* for more information.

3.5 Stave

A Virtual Music stave is an instance of the `Stave` class. A stave in Virtual Music is equivalent to a stave in traditional stave notation. They contain stave objects, where a stave object is an object that progresses along with the stave and is usually of a notational or instructional nature. Because of this, the function of a stave in Virtual Music Notation is mostly limited to the containerisation and maintenance of stave objects.

Staves are the basis of Virtual Music Notation, without which the notation would have no form or structure, as the size and shape of the generated notation on a stave indicates the characteristics of the notation. The size of the stave in terms of physical dimension is equivalent to the performance area. For more information, see *Setting the Performance Area.*

3.5.1 Retrieving Stave Objects from a Stave

Stave objects must be classes that inherit the `Stave Object` class. Usually, these are created with VIRTUAL MUSIC SHORTHAND NOTATION through a string of music symbols called a stave string. Examples of stave objects include Settings Blocks and notes. The latter is an example of an activatable stave object, which inherits from the `Activatable Stave Object` class. The `Activatable Stave Object` class itself inherits from the `Stave Object` class but provides required functionality to allow the activation and deactivation of the stave object. All stave objects are grouped into `List`s of like objects on a per-stave basis. These lists can be retrieved with the `GetObjectsOfType<T>` method at any point, where `T` is replaced by the type of stave object that is to be retrieved.

Any objects that inherit from the `Stave Object` class either directly or through a class that inherits from it may use the object's `ContainingList` property. The `ContainingList` property is a convenience property that simply refers to `Stave.GetObjectsOfType<T>`, where `T` is replaced with the object in question and `Stave` refers to the stave that the object is included on. For example, referring to `ContainingList` on a note would return a `List` of `Note`s on the same stave. Similarly, referring to `ContainingList` on a Settings Block would return a `List` of `Settings Block`s.

The index of a stave object within the list of its type should equal the value of its `Number` field, minus one. This can be used to retrieve the next stave object of the same type. For example, in a pseudo-generative 'choose your adventure' type of piece, a composer or developer may wish to mark certain notes following a note as unplayable (see *Prohibiting a Note from Being Played*. While this may be done with flagged notes (see *Accessing Notes by their Note Flag*) to prohibit an organised collection of notes

from being played, using `ContainingList` and `Note.Number - 1` may create a random, haphazard effect by randomly selecting notes to be marked as unplayable, as seen in Code Sample 3.10.

```
// for this example, a note has already been obtained
// via a flagged event or any other method and it is
// in the "note" variable

// what is the maximum number of notes to select?
int maximumNumberOfNotesToSelect = 20;

// using "Random" from the .NET "System" namespace
// as a pseudo-random number generator
Random random = new Random();

// this list will contain all the randomly selected notes
List<Note> selectedNotes = new List<Note>();

for (int i = 0; i < maximumNumberOfNotesToSelect; i++)
{
  // pick a random number between the obtained note and
  // the final note in the stave
  Note potentialNote = random.Next(note.Number - 1, note.ContainingList.Count - 1);

  // random.Next does not always guarantee a unique
  // number, so check if the potential note has
  // already been selected
  if (!selectedNotes.Contains(potentialNote))
  {
    selectedNotes.Add(potentialNote);
  }
}

// finally, mark eack of the selected notes as unplayable
foreach (var noteToMarkAsUnplayable in selectedNotes)
{
  noteToMarkAsUnplayable.OverrideAsUnplayable = true;
}
```

Code Sample 3.10: Prohibiting a random selection of notes from activation

3.5.2 Retrieving Information about a Stave

Various fields and properties of the `Stave` class return values for particular bits of information that a developer or composer may use to create particular functionality. For an exhaustive list of fields, properties, and methods of the `Stave` class, refer to Appendix R.

3.5.3 Implementing Custom Functionality

Custom functionality relating to staves may be implemented by the `Stave` class's two events, `Generated` and `Finalised`, which are triggered after the generation and finalisation of a stave, respectively. For more information, see *The Events System*.

3.6 VMManager

The `VMManager` class, referred to as the 'VMManager' or more simply as the 'manager', short for Virtual Music Manager, is the centralised manager and maintainer of the current instance of Virtual Music. It maintains the current state of Virtual Music and allows access to the current performed piece and all data associated with the current performance. In most cases, the manager is self-maintaining and it is not necessary to alter the manager in any way. However, a good understanding of the manager and its functionality will allow a developer to produce more immersive applications by, for example, providing audio effects (see *Set Global Audio Effects*) or implementing a custom pause screen (see *Prohibiting Automatic Pause Control*).

The `VMManager` implements the singleton pattern, which allows the simple access in a single instance of the `VMManager` within a scene through the `VMManager` class. It also implements the lazy implementation pattern, which allows the `VMManager` to instantiate itself when required if it has not already been instantiated (Skeet). This lazy implementation significantly simplifies the process to start using Virtual Music, and completely removes any requirement of any knowledge of code to use Virtual Music for basic functionality. Excerpts of the `VMManager` source code have been provided at Code Sample 2.13 which demonstrates the lazy loading implementation, and Code Sample 2.14

which illustrates how *Unity*'s `Awake` method was used to create a singleton.

Unlike the common pattern of accessing instanced data through an `Instance` or similarly named field, `static` fields have been provided which internally reference the `Instance` field. Data may be accessed through fields on the `VMManager` class, such as the current pause state, accessed through `VMManager.IsPaused` instead of `VMManager.Instance.IsPaused`. The `Instance` field has been retained for developers who are used to the singleton pattern, but these convenience fields allows simplification, particularly for developers unfamiliar with these types of patterns.

3.6.1 Accessing The Current Piece

The current piece being performed may be accessed by the `Piece` field. It is simply a reference to `VMManager.Enviroment.Piece` as the `Environment` is responsible for maintaining functionality of the performance (see *Environment*), but has been included for convenience. For more information on the `Piece` class, see *Piece*.

3.6.2 Accessing the Current Environment

The current performance environment may be accessed through the `Environment` field. For more information on the `Environment` class, see *Environment*.

3.6.3 Prohibiting Automatic Play

By default, the `VMManager` will play the current loaded piece automatically. This allows the simplification of the process in order to have a piece proceed as expected, and provides the recommended method to perform a piece. This behaviour may be overridden by setting `AutoPlay` to `false`, which simply prohibits the manager from controlling the progression of the piece.

If manual control of the piece is desired, a developer must simply call `VMManager.Play` during *Unity*'s `FixedUpdate` method within a class that inherits from the `MonoBehaviour` class in the `UnityEngine` namespace. Importantly, the developer must ensure that they invoke `VMManager.Play` only once in *Unity*'s `FixedUpdate` method[7].

3.6.4 Prohibiting Automatic Pause Control

The `VMManager` will, by default, control the pausing and playing of any loaded piece via the removal and replacement of any HMD worn by the performer. If the performer should remove their HMD during a performance, the piece will pause and any tones currently being produced will stop. The piece will resume as expected when the HMD is replaced.

This automatic control is considered a basic implementation suitable for most self-containing single-piece applications. In some advanced scenarios, a developer may wish to implement their own pause mechanism. For example, a developer may prefer a manual control so that they can implement a countdown timer before a piece is resumed, allowing

[7]Internally, after an initial warning, the `Environment` will silently prohibit multiple calls per `FixedUpdate` to its `Advance` method.

a performer sufficient time to prepare themselves.

To prohibit the automatic control of pausing by the `VMManager`, set `ControlPause` to `false`. Then, to manually pause a piece, set `IsPaused` to `true`. Doing so will correctly pause the progression of the piece as expected. To resume a piece, return `IsPaused` to `false`. In both cases, the `Paused` and `Unpaused` event will be triggered as expected, and any functionality that uses these events will still function correctly (see *The Events System* for more information).

If `ControlPause` is set to false, a developer must also ensure that they appropriately handle the HMD. This may be achieved through the use of the `HMDMounted` and `HMDUnmounted` events of the `OVRManager`, provided as part of the *Oculus Integration Package* (Facebook Technologies, 'OVRManager Class Reference').

3.6.5 Turning on Debug Mode

The VMManager will report only necessary information through *Unity*'s *Console* window. This includes known limitations or warnings such as when a custom note object is provided, which prohibits gradual changes to dynamics (see subsubsection 3.3.8). The verbosity of this reporting can be increased by setting `Debug` to true. This is recommended when a developer is encountering unintended or confusing behaviour so that they may debug their code.

3.6.6 Overriding the Default Synthesisers

The `VMManager` provides a set of default synthesisers for both the primary and secondary synthesisers. These synthesisers are used when a piece does not supply any equivalent synthesisers. They may be overridden by supplying a new synthesiser at either `DefaultPrimarySynthesiser` or `DefaultSecondarySynthesiser`, which overrides the primary and secondary synthesiser, respectively. This may be done via code or through the *Inspector*, which serializes the private `_DefaultPrimarySynthesiser` and `_DefaultSecondarySynthesiser` fields. See *Synthesiser* for more information about the functionality of a synthesiser, and *Creating a New Synthesiser* for how to create a new synthesiser.

3.6.7 Set an Overall Mixer

All audio produced by Virtual Music and the Virtual Music Instrument may be routed through a `Mixer` component for the purposes of applying audio effects (see *Set Global Audio Effects*) or adjusting the volume of any produced audio. A mixer may be supplied to the `Mixer` field through code or directly to the `_Mixer` field in the *Inspector*. Importantly, the ability to define a mixer within code allows the mixer or any component of the mixer to be changed at any point, including during performance. Refer to *Audio Mixer* in the *Unity* documentation for more information on the implementation and abilities of audio mixers within *Unity* (Unity Technologies, 'Audio Mixer').

3.6.8 Set Global Audio Effects

Unity allows the modification of the output of `Mixer` components with the application of audio effects. These may be applied to any mixer, including any configured for use with Virtual Music (see *Set an Overall Mixer*).

Unity includes many pre-built effects which are highly optimised and functional. They include, but are not limited to: high and low pass filters, echo and reverb, chorus and compression effects, and equalisation effects. The *Audio Effects* section of the *Unity* documentation provides instructions on how to apply audio effects to a mixer. It also contains a detailed list of the audio effect components included with *Unity*, and provides reference for how each effect's parameters may be used (Unity Technologies, 'Audio Effects').

3.7 Instrument

The Virtual Music Instrument or the 'instrument' is the term used to represent the manner in which Virtual Music Notation is translated to the production of sound within Virtual Music. Usually, the term Virtual Music Instrument refers to the use of the *Oculus Touch* controllers, but may also, when using recent HMDs which support hand tracking, simply refer to the hands. In both of these cases, the production of sound is handled by an instance of the `Instrument` class.

An instrument designed for use outside of Virtual Music played by the performer may also be referred to as an instrument in this *Developer Manual*. In this case, the performer interprets Virtual Music Notation and translates its indications to the

production of sounds suitable for that instrument. However, only the Virtual Music Instrument allows Virtual Music Notation to become a prescriptive form of music notation which is audibly recognisable on reperformance, rather than a descriptive form of notation as with an acoustic instrument.

The Virtual Music Instrument passes any instruction to produce audio to the appropriate synthesiser according to its type. This means that the Virtual Music Instrument itself does not produce any sound, but is instead the manner in which Virtual Music Notation or any performance direction is translated to the production of sound by the synthesiser. The manner in which it translates changes according to *The Performance Modes*.

In most situations, the Virtual Music Instrument is self-maintaining and requires no adjustments or monitoring. However, various functions of the instrument may be used or overridden to create custom functionality. For a full reference of the available fields, properties, and methods of the `Instrument` class, refer to Appendix S.

3.7.1 The Control Scheme

The Virtual Music Instrument may be controlled in a variety of different ways, and the required controls may change for each performance mode (see *The Performance Modes*). Each instrument operates individually and the current status of one instrument has no impact on the operation of the other. The following sections describe the default method to control each aspect of tone production if they are required or available for the current performance mode, including how to control pitch, rhythm, and dynamics, and how to adjust the tempo, sustain notes, or adjust the tempo.

Controlling Pitch and Time

Tones may be produced by pressing the `PrimarySynthesiserButton` or the `SecondarySynthesiserButton` which will activate the primary synthesiser or the secondary synthesiser, respectively. These buttons are, by default, the 'A' and 'B' buttons on the right-hand controller, and the 'X' and 'Y' buttons on the left-hand controller, respectively named `OVRInput.Button.One` and `OVRInput.Button.Two` in the `OVRInput` namespace of the *Oculus Integration* package. The produced pitch of any tone is directly related to the height of the instrument in three-dimensional space: the higher the instrument is in the performance area, the higher the frequency, and likewise the lower the instrument is, the lower the frequency. The instrument will also vibrate to give haptic feedback to the performer, and will vibrate at higher frequencies for tones of a higher pitch, and at lower frequencies for tones at a lower pitch.

For tones currently being controlled—that is, tones that the performer is controlling the pitch of and which are not under the influence of the sustain button (see *Sustaining Notes*)—the pitch of the tone may be adjusted by moving the instrument up or down. This produced pitch is by default tonally locked to the temperament and tonality system of the current environment, but this pitch correction behaviour may be 'unlocked' with the `UnlockIntervallicModeButton`, which is by default the index finger trigger.

Controlling Dynamics

The dynamic level of any produced tone is controlled by the rotation of the instrument. If the instrument is rotated anti-clockwise[8], tones produced will be of a lower dynamic

[8]For the left hand, the performer's palm will be facing up, and for the right hand, their palm will be facing down.

value. Similarly, if the instrument is rotated clockwise[9], tones produced will be of a higher dynamic value. Any point between these two values will produce a dynamic value appropriate to the degree to which the instrument is rotated. The vibration of the instrument will adjust according to the dynamic value of the instrument, and will vibrate with a higher intensity or amplitude for higher dynamic values.

Sustaining Notes

By pressing the `SustainSynthesiserButton`, any produced tone may be sustained. By default, this button is the primary hand trigger, `PrimaryHandTrigger` in the `OVRInput` namespace of the *Oculus Integration* package: the trigger activated by any of the last three fingers of the hand or by squeezing the controller. For tones in performance modes that take their pitch and rhythm from the activation and deactivation of notes, their notes remain in an active state even if the instrument is no longer within their activation area, therefore allowing an unlimited number of tones to be produced simultaneously by a single instrument. Such tones will continue to ascertain the pitch or dynamic value from the note according to the performance mode. For example, in `Simple` mode, a sustained tone will continue retrieving its pitch and dynamic values from the note, and in `Intermediate` mode, the tone will only retrieve the pitch from the note and will continue retrieving the dynamic value from the instrument. For performance modes that require button presses to produce tones, such as the `Intermediate` and `Advanced` modes, the pitch of any sustained note remains at the value when the relevant button was released. Tones that are sustained will cease sustaining when the `SustainSynthesiserButton` is released, or for tones that require the activation of a note, when the note is finalised.

Retrieving the dynamic value from the instrument directly in Performance Modes such

[9]For higher dynamics, the palm of the performer's left hand will face down, and the palm of the right hand will be facing upwards.

as `Intermediate` and `Advanced` allows the performer to create chords and control the dynamic value of all notes in the chord simultaneously. If fine control over the dynamic value of individual tones of a chord is desired, a performer may use a different instrument for those tones, making use of their independent control. The vibration of the instrument is reflective of the first tone produced of any sustained notes.

Changing the Performance Mode

The performance mode (see *The Performance Modes*) may be changed by supplying the `PerformanceMode` with a new `PerformanceModeOption`, or by the performer with the `IteratePerformanceModeButton`. By default, this is done by pressing the thumbstick on either controller. Pressing the `IteratePerformanceModeButton` iterates through the available options, starting from the basic options and ending with the special modes before looping back to the beginning. They are, in order:

1. `Simple`
2. `Intermediate`
3. `Advanced`
4. Special Modes:
    (a) `Basic`

Adjusting the Tempo

The tempo of a piece may be adjusted during performance by moving the thumbstick of either hand up to increase the tempo, or down to reduce it. This can be done to

create an accelerando or ritardando effect, either due to notated instructions or due to the performer's creative interpretation.

The current tempo adjustment can be retrieved via the `BPMFineAdjustment` field, which reflects the multiplier used on the current tempo. When the thumbstick is in its neutral position, `BPMFineAdjustment` returns `1f`. In the thumbstick's lowest position, the field returns `0.1f`, equalling one tenth of the original tempo. In the thumbstick's highest position, the field returns a value of `2f`, doubling the tempo. Each instrument impacts the tempo independently, but the effect is compounded by multiplying the effect of both together. For example, if the tempo of a piece was 120 and the `BPMFineAdjustment` field of both instruments returned a value of `2f`, the tempo would equal $2 \times 2 \times 120 = 480$. The minimum value of `0.1f` for each instrument prohibits the performance stopping completely, though very low tempos may still be achieved: if both instruments had their thumbsticks in their lowest position, the tempo would be multiplied by 0.1×0.1, $1/100$ of the original tempo.

Adjusting the Control Scheme

The control scheme of an instrument may be adjusted by changing the `OVRInput.Button` field relevant to the control to be changed. These fields may be changed to any button or trigger of the `OVRInput.Button` `enum`. For more information, see *OVRInput Class Reference* (Facebook Technologies, 'OVRInput Class Reference'). Fields exposing changeable controls include:

- `PrimarySynthesiserButton`
- `SecondarySynthesiserButton`
- `SustainSynthesiserButton`

- `IteratePerformanceModeButton`
- `UnlockIntervallicModeButton`

3.7.2 The Performance Modes

The Virtual Music Instrument has four performance modes. These performance modes reflect the level of freedom the performer has in regard to the interpretation of the Virtual Music Notation. This interpretive freedom is gained through the increased control over various aspects of the tone production, including the dynamics, pitch, and rhythm. The modes start from Simple, where the Virtual Music Notation is prescriptive and the Virtual Music Instrument will produce tones exactly as indicated, to Advanced, where the performer is free to interpret the notation as they wish. The default performance mode is Simple but this mode may be changed at any time. See *Changing the Performance Mode* for more information.

Simple

The Simple Performance Mode is the default Performance Mode. Tone generation begins when a note is touched and continues until the note is no longer touched, if it finalises, or if it is forced to be unplayable (see *Prohibiting a Note from Being Played*). The Simple Performance Mode allows for some control of rhythm by allowing the performer to choose when or when not to touch a note, but limits the performer's control over the interpretation of pitch and dynamics, which are taken directly from any touched note. This includes any sustained tones, of which the pitch and dynamics are according to the note that began them.

The Simple Performance Mode also includes a 'touch buffer'. This touch buffer allows a performer to fluctuate between touching and not touching a note, and the note will remain as though it was touched. The touch buffer can be adjusted via the `TouchBufferInSeconds` field, which defaults to `1.5f`. This value is used by VIRTUAL MUSIC in an internal calculation which uses this value and the amount of time that has passed since the note activation area of the note was exited, and returns a point between the centre of the note being touched and the current position of the VIRTUAL MUSIC INSTRUMENT. The point returned is the point an object would be on the path between the two positions if it would take the amount of time indicated in `TouchBufferInSeconds` to travel between them. If the point returned is outside of the activation area of the note, it is deactivated. Therefore, the value of `TouchBufferInSeconds` equals the maximum amount of time before a note will be deactivated. This increases the perceived reliability of the VIRTUAL MUSIC INSTRUMENT's tone production as small, unintentional note deactivations are ignored, but importantly, intentional deactivations by larger movements correctly deactivate the note.

Intermediate

The Intermediate Performance Mode allows the control of dynamics and greater control of time but does not allow any control over what pitch is produced, as the pitch of tones produced is taken from any touched note. Dynamics of any tones produced, including any sustained tones, are according to the current dynamic value of the instrument, equivalent to the rotation of the instrument (see *Controlling Dynamics*). Appropriate tone generation button presses for the note's indicated synthesiser type are required for this mode. See *The Control Scheme* for more information on the tone generation control scheme. Intermediate mode also removes the touch buffer from Simple mode and so gives the performer more

responsibility to ensure that their movements are intentional and controlled.

Advanced

The Advanced Performance Mode does not take any instruction from the notation, and the pitch, dynamic value, or rhythm of any tone produced is at the discretion of the performer. In this Performance Mode, the notation is simply for the purposes of free expression; the performer may, if they wish, produce a tone exactly as indicated by the note, or they may interpret that note in any way they wish. This mode requires input for all aspects of tone production including pitch and rhythm (see *Controlling Pitch and Time*), and dynamics (*Controlling Dynamics*). The dynamic value of any sustained tone is according to the current dynamic value of the instrument.

By default, a 'pitch guide' is shown when in the Advanced Performance Mode. The pitch guide assists the performer in producing the desired pitch by allowing them to see the division between each pitch. For more information, see *The Pitch Guide*.

Special Performance Modes

Basic The Virtual Music Instrument has one 'Special' Performance Mode, included to increase accessibility especially for users with low physical mobility. The Basic Performance Mode functions similarly to Simple mode: the instrument takes all tone generation information from the notation and allows no interpretive freedom over dynamics and pitch, but allows some freedom over the aspect of time due to tone production beginning when a note is touched. The Basic mode differs from Simple in one key aspect: once tone production has begun due to a note being touched, the tone

will continue to be produced even if the note is no longer being touched. Tone production behaves as though the sustain button is always pressed.

This Performance Mode is considered 'special' as its reduced functionality is not typical of usual performance methods and can be considered as less functional than the simple mode. However, this mode may be useful in situations where consistently touching a note may be problematic for a user, such as for those with low mobility or in therapy. Further discussion of the use of the Basic mode in therapy may be found in subsection 5.2.1.

3.7.3 The Pitch Guide

The pitch guide is a series of lines representing where the change between each pitch lies, as seen in Figure 3.4. The pitch guide shows where the octave changes with a slightly thicker line. In addition, the minimum and maximum playable points are shown with a line thicker than the octave division line. The pitch guide appears in a mask surrounding the instrument so that the pitch guide does not occlude the performer's view too greatly. The appearance of the pitch guide, including the size of the mask may be adjusted by the `PitchGuideLineThickness`, `PitchGuideWidth`, and `PitchGuideMaskSize` fields.

The pitch guide allows a performer to more clearly see where they must place their hands in order to produce a pitch. This also improves the ability for the Virtual Music Instrument to be used in conjunction with an alternate notation system, for example traditional stave notation, as the performer may more easily produce tones as they intend. With enough skill, the pitch guide would not be necessary in such a situation, however its inclusion allows the performer to progress in their ability to express their musicality.

The pitch guide is automatically shown if the performance mode recommends it.

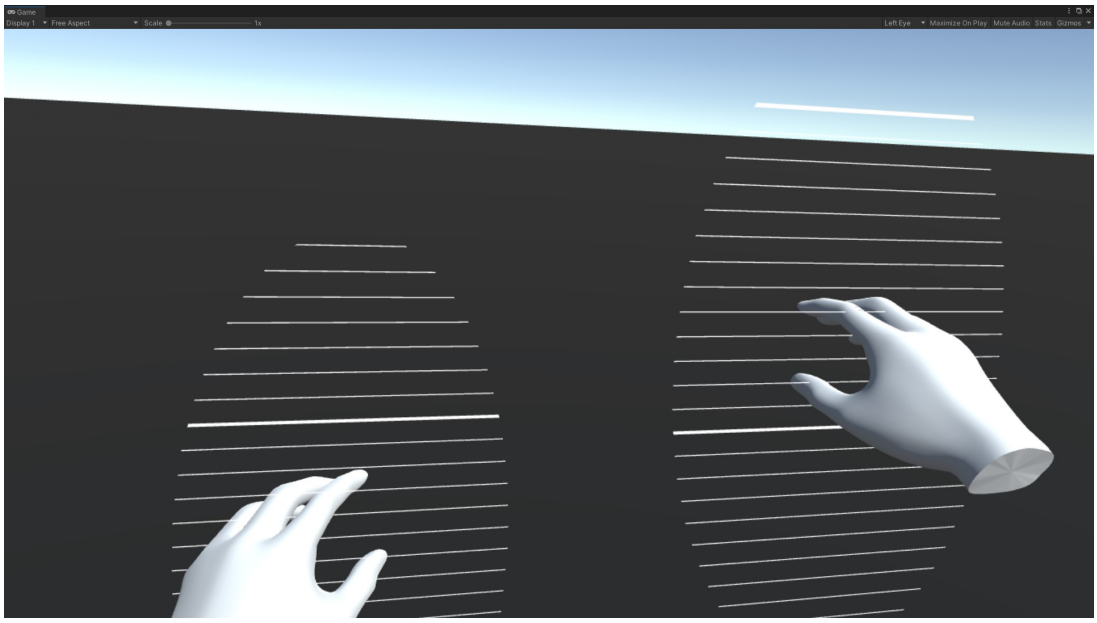


Figure 3.4: A pitch guide being displayed for both instruments

Currently, the pitch guide is only recommended for the Advanced performance mode, since it is the only mode in which a performer may alter the pitch of a tone freely. This behaviour may be overridden by providing a non-`null` value to the `OverridePitchGuide` `bool?`[10]. A value of `true` ensures that the pitch guide is shown in all performance modes, and a value of `false` prohibits the pitch guide from being shown in any mode. The default behaviour may be restored by restoring `OverridePitchGuide` to `null`.

3.7.4 The Instrument Types

The `Type` field of the `Instrument` class simply refers to whether the instrument is for the left or right hand. The field is a value of the `InstrumentType` `enum` which has possible options of `Left` or `Right`. These values are equal to the controller types in the `OVRInput` namespace of the *Oculus Integration* package and are reflective of which

[10]The `?` character after the term `bool` indicates that `bool` has been made a nullable value type (Microsoft, 'Nullable value type').

controller they will follow.

3.7.5 Retrieving the Current Pitch Degree

The current pitch degree may be retrieved at any point using the `CurrentPitchDegree` field. It returns a positive whole number (an `int`) equivalent to the pitch degree that would be produced at the instrument's current height in the current performance environment. This field is used internally by the `Note` class to highlight when a note is at the same height as the instrument, but it may also be used to create a GUI or for educative purposes. For example, a developer may retrieve the pitch degree of both instruments and calculate when the performer is able to perform the desired interval. If the octave division settings are set to their default values (see *Octave Division, T*), for an interval of a major third, a developer would only have to subtract the `CurrentPitchDegree` of one instrument from the other and return if that value is equal to four, as seen in Code Sample 3.11. This may be used in a tutorial familiarising a performer with Virtual Music.

```
Instrument left = Instrument.Get(InstrumentType.Left);
Instrument right = Instrument.Get(InstrumentType.Right);

// check if the interval between both instruments is four
// semitones, a major third, return true if so
//
// note the use of the "Abs" method of the "Mathf" namespace,
// which always returns the "absolute" positive value
return Mathf.Abs(left.CurrentPitchDegree - right.CurrentPitchDegree) == 4;
```

Code Sample 3.11: Calculating the interval between two instruments

3.7.6 Retrieving an Instrument

An instrument may be retrieved at any time with the `static` `Get` methods. These methods allow the retrieval of the instrument matching the performer's dominant hand or the non-dominant hand by passing through a parameter of `true` or `false` respectively. The left or right instrument may be retrieved instead by specifying an `InstrumentType` parameter. The `Get` method may also be used without any parameter to retrieve the performer's dominant hand. As these methods are `static` methods, they are callable from scripts and allow the developer to create custom functionality, such as checking the interval between two instruments, as seen in Code Sample 3.11.

Retrieving the instruments could allow a performance to be recorded for example, by tracking the location of the instruments; the location data would be recorded and used to make a `GameObject` follow the exact movements of the instrument in the recorded performance at a later time. This could also be combined with the `GameObjectToFollow` field to create a recorded performance that could be played back at any time (see *Following the Location of Another Object*).

3.7.7 Assigning the Instrument to a Mixer

The audio for the instrument may be routed through a mixer which combines the output of both instruments together (a 'parent' mixer), or individual mixers which separate the audio for both instruments ('child' mixers). In order to use mixers for the instrument, a separate mixer must first be supplied to the `VMManager.Mixer` field (see *Set an Overall Mixer*). This mixer will then contain mixer groups with the appropriate naming scheme for a parent or child mixer configuration. The mixer group for a synthesiser type may

be retrieved in code at any time via the `static` `GetMixer` method, which accepts a `Synthesiser.Type` parameter.

Parent Mixer Configurations

A parent mixer configuration allows the simple addition of audio effects to all synthesisers of an instrument simultaneously by routing all audio through a single parent mixer group. This can be achieved simply with the creation of a mixer group within the global mixer called simply 'Instrument'. This mixer group is subsequently used for both `Primary` and `Secondary` synthesiser types. Audio effects added to this mixer effect both types of synthesiser simultaneously.

Child Mixer Configurations

A child mixer configuration allows greater control over the sound of each synthesiser as audio effects may be applied separately. This configuration can be achieved with the creation of mixer groups called 'Primary Synthesiser' or 'Secondary Synthesiser' as children of the global mixer. Audio of the respective synthesiser types is then routed accordingly through their mixer.

A child mixer configuration may also be used in conjunction with a parent configuration by creating synthesisers with the appropriate child naming scheme as mixer groups of the parent configuration. A complete configuration in this manner would result in a parent 'Instrument' mixer group with 'Primary Synthesiser' and 'Secondary Synthesiser' children. This allows audio effects such as reverb to be applied to both synthesiser types and effects, such as a distortion or chorus effect, to be applied

separately. If a child mixer has not been created for a particular synthesiser type, the audio for that synthesiser type is routed directly through the parent mixer group.

3.7.8 Adding an Audio Effect to an Instrument

Virtual Music offers significant customisation of sound quality through available audio effects for mixers at a global, instrument, or per-synthesiser level. See *Assigning the Instrument to a Mixer* for further discussion on how an instrument or synthesisers of an instrument may be routed through mixers, and *Set an Overall Mixer* for directions on how to apply a global mixer. Any of these mixers may have audio effects added to them in the same manner as discussed in *Set Global Audio Effects*.

3.7.9 Following the Location of Another Object

By supplying a `GameObject` to the `GameObjectToFollow` field, the position of an instrument may be overridden. The field could be used for adaptive controllers or control mechanisms, or even to allow the use of Virtual Music without VR. For example, a performer with limited mobility could use an adaptive controller such as the *Xbox Adaptive Controller* (Microsoft, 'Xbox Adaptive Controller') to control the location of a `GameObject` on the screen. If this `GameObject` was referenced in the `GameObjectToFollow` field, the instrument would follow the position of that `GameObject`, enabling that performance. The location may then be further adjusted by using the `PositionOffset` and `RotationOffset` fields. For further discussion, see subsection 5.2.1.

3.8 Synthesiser

The synthesiser refers to an instance of the `Synthesiser` class. The synthesiser may be thought of as a timbre preset rather than an instrument in itself, similar to the various presets included on many electronic piano keyboards. This is because it does not maintain the status of the sound it produces, but instead informs how 'tones'—an instatiation of the `Tone` class—should construct their waveform. Besides the input required in creating them, a synthesiser usually requires no continuous input or monitoring.

3.8.1 The Synthesiser Types

The term 'synthesiser type' simply refers to whether the synthesiser is primary or secondary. In code, it refers to the `Synthesiser.Type` `enum`, which has two members: `Primary` and `Secondary`. A primary synthesiser is used by default and for note types which indicate they should sound with the primary synthesiser, whereas a secondary synthesiser is used for notes which have their type set to secondary (see *The Note Types*) or when the performer specifically activates the secondary synthesiser.

Components of the `Synthesiser` class should be considered equivalent to objects or concepts named similarly for electronic musical synthesisers. This includes oscillators (referring to objects that produce a periodic, oscillating signal), envelopes (referring to mechanisms that shape a signal, usually amplitude), and LFOs (oscillators usually at a very low frequency, often used to alter the produced frequency or amplitude of an oscillator).

A synthesiser does not explicitly specify its type. Instead, a synthesiser may be set as a

primary or secondary synthesiser either globally (see *Overriding the Default Synthesisers*) or for a particular piece (see *Setting the Piece's Synthesisers*).

3.8.2 Filtering the Output of a Synthesiser

The output of a synthesiser may be filtered with the addition of audio effects to mixers. See *Adding an Audio Effect to an Instrument* for more information.

3.8.3 Creating a New Synthesiser

The sound of a synthesiser is defined by its oscillators and envelopes. A number of basic oscillators have been provided, including a sine wave, sawtooth wave, triangle wave, square wave, and white noise oscillator. Any number of oscillators, envelopes, and LFOs may be combined to produce a unique timbre, limited only by the processing capabilities of the device used to run the application.

The process to create a new synthesiser is a complex process but is greatly simplified thanks to the implementation of a GUI. Synthesisers may also be created in code via the `Synthesiser` class. To create a new synthesiser using the GUI, right-click in the *Assets* directory in the *Unity* window, navigate to *Create > Virtual Music > Synthesiser*, and select *Synthesiser*, as seen in Figure 3.5. This creates a new, blank synthesiser in the project's *Assets* folder. Selecting the synthesiser file will activate the GUI in the *Inspector* window, which allows the customisation of the synthesiser's timbre.

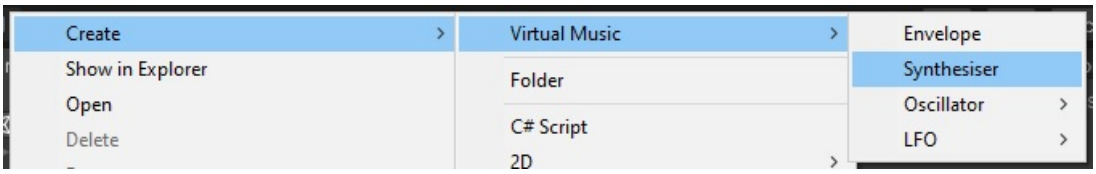


Figure 3.5: Creating a new synthesiser

Creating an Oscillator

An oscillator may be created by right-clicking in the *Assets* directory, navigating to *Create* > *Virtual Music* > *Oscillator*, and selecting the desired basic waveform type. Selecting the newly created oscillator in the project's *Assets* folder activates the GUI in the *Inspector*. This GUI may be used to modify the timbre of the oscillator, such as by the addition of an envelope, LFOs, or by modification to the input frequency and amplitude through the `OctaveDifference`, `FrequencyAdjustment`, `FrequencyMultiplier`, and `AmplitudeMultiplier` fields, as seen in Figure 3.6.

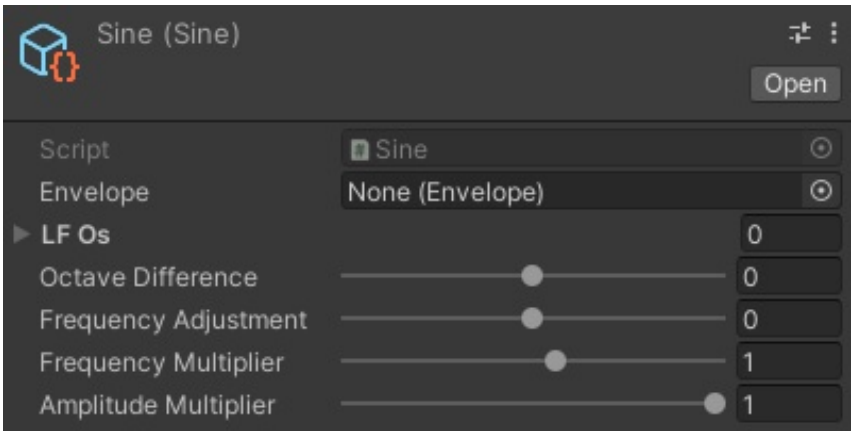


Figure 3.6: The oscillator GUI

Custom oscillators may also be created and used in synthesisers by inheriting the `Oscillator` class to create new waveform types. Since the `Tone` class creates sound through adjustment to their `Phase` field, to create a new oscillator, a developer should only need to override the `virtual` `Waveform` method. The `IncreasePhase` and

`GetSample` methods may also need to be overridden, depending on the complexity of the desired waveform.

Applying an Envelope

An envelope or 'envelope generator' is a method of contouring the amplitude of a sound (Vail 152). Envelopes in VIRTUAL MUSIC are *D+ASDR* envelopes: a modification to the standard *ASDR* type as Vail describes with the addition of an optional delay time. They offer fine control over the shape of amplitude through five parameters, seen in Figure 3.7: delay (D^1), attack (A), decay (D^2), sustain (S), and release (R). While delay, attack, decay, and release refer to the time in seconds for the entire phase to complete, sustain is a zero to one value used as a multiplier on the maximum amplitude. They may be applied to the amplitude of a synthesiser directly, or to the input amplitude to an oscillator, including LFOs.

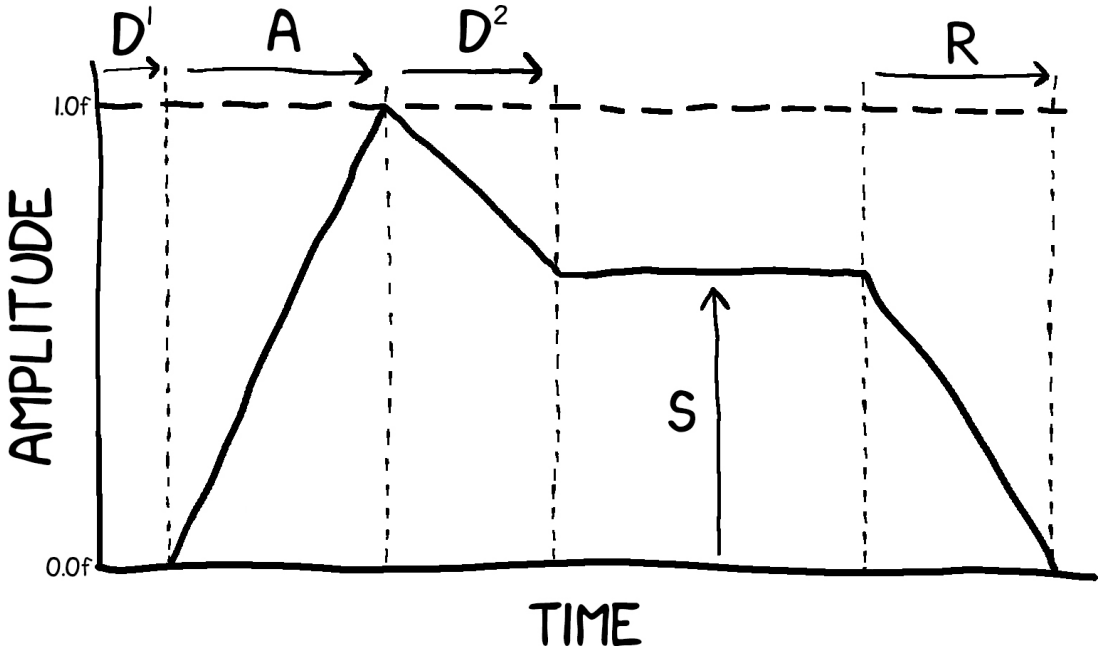


Figure 3.7: The schematic of an envelope

The delay time of an envelope is the amount of time that a tone should remain silent before moving to the attack phase, where the attack time refers to the amount of time until the amplitude of a tone reaches its maximum value after the decay time. Once the amplitude has reached its maximum value (once the attack phase is complete), for the amount of time defined by the `Decay` field, the amplitude will linearly reduce to the maximum amplitude value multiplied by the sustain strength. The amplitude of the tone will remain multiplied by the sustain strength until it is released, where the amplitude will linearly return to zero after the amount of time defined by the `Release` field.

Envelopes may be created in a similar manner to the oscillator: right-click in the *Assets* directory and navigate to *Create > Virtual Music > Envelope*. After selecting the newly created envelope in the *Assets* folder, the *Inspector* will show an interface to adjust the above parameters. Once an envelope has been created, it can be applied to a synthesiser as an 'overall amplitude envelope' through the field in the *Inspector* described as such, or to an oscillator or LFO via the *Envelope* field. Envelopes may also be applied in code via the `OverallAmplitudeEnvelope` and the `Envelope` fields, respectively.

Adding a Low Frequency Oscillator

An LFO is used to modulate the frequency or amplitude of the oscillator to which it is attached. LFOs can create a vibrato effect by modulating frequency, or can create a tremolo effect by modulating amplitude. Any number of LFOs can be applied to an oscillator.

The `LFO` class inherits from the `Oscillator` class, which means that an LFO has much of the same functionality of an oscillator, including the ability to have envelopes and even LFOs of its own. However, an LFO has a constant frequency and amplitude,

which must be predefined. These parameters may be modulated by envelopes and LFOs. For example, an envelope with a half second decay was used to create a delayed vibrato effect for *Étude 7* and *Étude 8*. As the `LFO` class inherits the `Oscillator` class, it can be created and modified in the same way as an oscillator (see *Creating an Oscillator*), though by instead navigating to *Create > Virtual Music > LFO*.

Once created, selecting the newly created LFO will activate the interface in the *Inspector*. This interface allows the modification to all parameters of the LFO, including which parameter it applies to. LFOs may then be applied to oscillators in the *Inspector* or via code through the `LFOs` field.

The LFO adjusts the routed parameter by multiplying the input value. This can be considered as $y = x + x \times LFO$, where x is the input value and LFO is the waveform of the LFO, ranging from negative one to positive one. The amplitude of the LFO is the strength the LFO has on the routed parameter as a percentage. For example, if an LFO with an amplitude of one was routed to modify frequency and the input frequency was 220 Hz, the returned frequency once modified by the LFO could range anywhere from 0 Hz to 440 Hz. Similarly, if the amplitude of the LFO was 0.5, the returned frequency would be 220 Hz plus or minus 0.5×220: 110 Hz to 330 Hz.

Testing the Sound

The sound of a synthesiser may be tested thanks to the implementation of a custom editor, which alters the appearance of the *Inspector* when viewing a synthesiser file to allow sample tones to be produced called 'test tones'. To generate a test tone, simply click the 'Generate test tone' button, seen in Figure 3.8. The frequency, amplitude, and length of the test tone may be adjusted with the sliders or by inputting values.

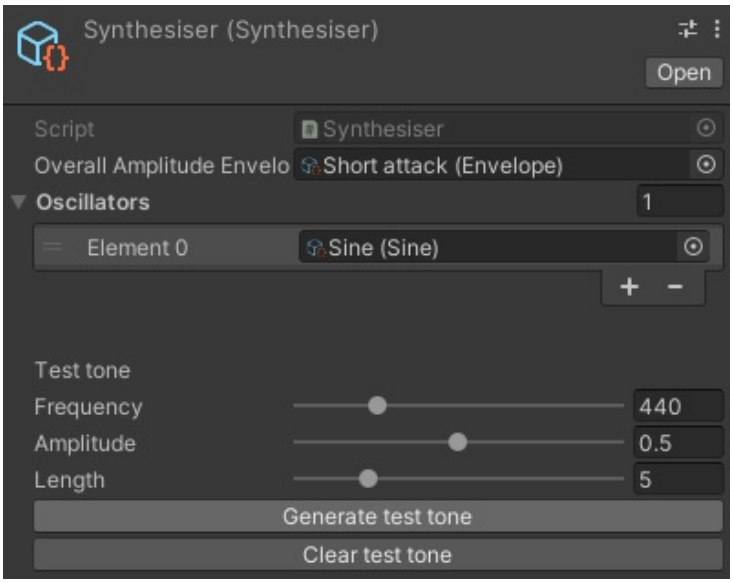


Figure 3.8: The synthesiser GUI

Due to limitations of the mixer, it is not possible to audition any audio effects applied to the synthesiser's respective mixer and the tone produced will be a pure representation of the synthesiser. The timbre of a synthesiser with audio effects may only be auditioned by playing a scene which has the synthesiser set up for use.

3.8.4 Retrieving the Current Synthesisers

The current synthesiser for each type may be retrieved at any time via the `static` `Get` method. This method must be called with a `Syntheser.Type` parameter which informs which type of synthesiser is to be returned. Optionally, a `bool` may be supplied after the `Synthesiser.Type` parameter in the `Get` method, which informs whether the method should return the default synthesiser of the requested type.

3.9 Environment

The `Environment` class is responsible for maintaining the current performance environment. This includes maintaining and controlling the current performed piece and its related data, and how that piece is generated or functions. Usually, a developer or performer does not need to interface with the environment directly. Instead, the environment is referred to internally to ascertain necessary elements of generation or performance, such as the note object height (see *Retrieve the Note Object Height*) or the time scale, including any fine adjustments to tempo by the instrument (see *Adjusting the Tempo*) via the `RealTimeScale` property. However, adjusting some fields of the `Environment` class may assist with altering the overall appearance and feel of VIRTUAL MUSIC, or assist with creating desired functionality, such as *Changing the Time Scale* to create a fast-paced feel.

3.9.1 Retrieving Current Performance Information

Members of the `Environment` class provide access to various aspects of the current performance, including the current piece (`Piece`), the BPM (`BPM`), the octave division (`OctaveDivision`), the fundamental frequency (`BaseFrequency`), the time scale (`TimeScale` and `RealTimeScale`), and the location of the tempo change indications (`BPMChangeLocation`). See Appendix T for a detailed list of the members of the `Environment` class.

3.9.2 Retrieve the Note Object Height

By default, generated notation will divide the playable area into the number of available pitches and calculate the required note height automatically. This is done with the `GetNoteObjectHeight` methods, which use the equation seen in Equation 3.4 to produce a note height that will be even and distribute notes evenly, leaving no vertical space between neighbouring pitches.

$$\text{note height} = \frac{\text{maximum performance height} - \text{minimum performance height}}{\text{octaves} \times \text{tones per octave}}$$

Equation 3.4: Calculating an even and distributed note height

These helper methods supply the correct note height in all situations at all points in a piece, including when the number of tones per octave or the number of octaves changes. By default, with no parameters, the `GetNoteObjectHeight` method will return the note height, considering any override if one is in place (see *Altering the Note Widths and Heights*). Variations of this method may be called by passing different parameters to forcibly retrieve the recommended note height instead, or the recommended note height for a different performer in the current environment.

3.9.3 Changing the Time Scale

The time scale effects the perceived speed of a piece without altering the tempo by adjusting the length and speed of the notation simultaneously. By default, the time scale is equal to `2f` and is accessible by the `TimeScale` field. This field reflects a multiplier used on both the shape of generated notation and the speed at which the

notation progresses. Higher values make the notation appear longer, but the staves move more quickly, making their musical duration the same. Similarly, lower values shorten the notation and lowers the speed of the staves. Adjusting the `TimeScale` field can create a fast-paced feel by increasing the time scale, or can allow the performer to see more notation concurrently by reducing the time scale. This field may only be changed before a piece has been generated, as it directly effects how the notation is generated.

3.9.4 Converting Height to Pitch Degree or Frequency

The `Environment` class allows the conversion of height or pitch degree data to precise pitch degrees or frequencies. Height data in the environment is measurable in two ways: in game-world units (the 'precise height'), or relative to the current performance environment, which is equal to zero at the performer's minimum performance height, and the current maximum number of octaves at the performer's maximum performance height (the 'relative height'). These heights can be input into a variety of different methods, for example to translate the precise height to a relative height, and then convert that data to a pitch degree or frequency. These methods include:

- `TranslatePreciseHeightToRelativeHeight`
- `GetFrequencyFromPreciseHeight`
- `GetPitchDegreeFromPreciseHeight`
- `GetPitchDegreeFromRelativeHeight`
- `GetFrequencyFromRelativeHeight`.

Some methods are simply convenience methods which first convert the input data in the required way. For example, the `GetPitchDegreeFromPreciseHeight` method first converts the given precise height to a relative height with the `TranslatePreciseHeightToRelativeHeight` method, and then converts that relative height to a pitch degree with the `GetPitchDegreeFromRelativeHeight` method.

While there are methods to allow the conversion of height to pitch degree or height directly to frequency, there is no method to convert a pitch degree to frequency. This is because a pitch degree may be used to retrieve the frequency of that pitch degree directly by using the pitch degree (minus one) as an index for the `Frequencies` array.

Importantly, the relative height returned by these methods does not consider intervallic structure, so may be higher or lower in pitch than their closest equal-tempered pitch degree as a developer may expect. These heights may be 'pitch corrected' by the `PitchCorrectFromRelativeHeight` method, which returns the pitch degree which the relative height is closest to but not below.

3.10 VMPerformer

The `VMPerformer` class is a `static` class that allows the maintenance and monitoring of a performer using Virtual Music through any script. It also allows the retrieval of their preferences through the `Preferences` field, used for the storage and retrieval of the performance area via `MinimumHeight` and `MaximumHeight`, and to set performer's preferred hand via `DominantHand`.

3.10.1 Setting the Performance Area

The performance area may be adjusted by setting the performer's minimum and maximum performance height in the `Preferences.MinimumPerformanceHeight` and `Preferences.MaximumPerformanceHeight` fields. These fields represent the lowest and highest point that a performer can comfortably and repeatedly reach, respectively. Either field may only be adjusted when a piece is not loaded. It is recommended to set the performance area before a piece, during a tutorial, or in a settings menu. An example of a method to achieve this is seen in Code Sample 3.12, which sets the minimum height to the position of the left hand and the maximum height to the position of the right hand.

These points are measured as the distance in game-units from the floor. This means that a developer must ensure that the performer is at zero on the Y axis when taking these measurements. They must also ensure that it is clear that these minimum and maximum points should be comfortable to reach repeatedly, without strain, as these points may be used frequently during a performance. Incorrect measurements may cause the performer to become uncomfortable, or limit their ability to perform.

3.10.2 Setting the Performer's Dominant Hand

The performer's dominant hand is the position that Virtual Music Notation uses as its rhythmic origin. The dominant hand can move in any direction without impacting the rhythmic interpretation or tempo of a piece, as the stave will match these movements accordingly. If a performer wishes to anticipate or delay a note, they may do so with their non-dominant hand. This ensures that the tempo of a piece remains clear and predictable,

```
using UnityEngine;
using VirtualMusic;

public class SetPerformanceArea : MonoBehaviour
{
    void Update()
    {
        // check if any button is pressed on any controller
        if (OVRInput.GetDown(OVRInput.Button.Any, OVRInput.Controller.LTouch) ||
            OVRInput.GetDown(OVRInput.Button.Any, OVRInput.Controller.RTouch))
        {
            // set the minimum height to the height of the left controller
            VMPerformer.Preferences.MinimumHeight =
                Instrument.Get(Instrument.InstrumentType.Left).transform.position.y;

            // set the maximum height to the height of the right controller
            VMPerformer.Preferences.MaximumHeight =
                Instrument.Get(Instrument.InstrumentType.Right).transform.position.y;

            // save and reload the preferences
            VMPerformer.SavePerformerPreferences();
            VMPerformer.ReloadPerformerPreferences();
        }
    }
}
```

Code Sample 3.12: Setting the performance area

in turn allowing live accompaniment by other performers also performing with Virtual Music or with real instruments, or by recording.

The performer's dominant hand may be changed by supplying the relevant `Left` or `Right` member of the `Instrument.InstrumentType` `enum` to the `Preferences.DominantHand` field. Though this may be done at any time, like setting the performance area, it is recommended to set this value during a tutorial or in a settings menu.

3.10.3 Saving and Loading Preferences

Preferences for a performer are stored only in memory through an instance of the serializable `Preferences` class (Unity Technologies, 'Script Serialization'). They may be saved to a file for persistent storage and reuse by calling the `SavePerformerPreferences` method. This method stores the performer's preferences in

a binary formatted file called 'performer.dat' in the `Application.persistentDataPath`, which is the recommended directory for the storage of persistent data ('Application.persistentDataPath'). Preferences may be (re)loaded by invoking the `ReloadPerformerPreferences` method. As the method's name implies, a performer's preferences are automatically loaded when Virtual Music is initialised, but this method may be used to reload the preferences file if needed. Both the `SavePerformerPreferences` and `ReloadPerformerPreferences` methods can be seen in the last few lines of Code Sample 3.12.

3.10.4 Retrieving the Location of the Performer

The position of the performer may be retrieved via the `transform` field. The `transform` field exposes the position of the `centerEyeAnchor` property of the `OVRCameraRig`, a class from the *Oculus Integration* package. The `centerEyeAnchor` is representative of the point between both eyes of the performer's head and is therefore best suited to ascertain the performer's position. The `transform` field could be used, for example, to create a camera system that follows the performer, allowing an audience to watch a Virtual Music performance.

3.11 The Events System

The events system in Virtual Music is a system of event emission that allows the implementation of custom functionality through the invocation of supplied methods. It uses the event specification as part of the C# language which has native language support regardless of framework (Microsoft, 'Events'). The native event specification was used

instead of the `UnityEvent` class or a custom event implementation for three reasons: to improve standardisation, to allow greater accessibility to developers already familiar with the concept of events in C#, and finally so that a class which represents an object that may be instantiated multiple times—such as the `Note` class—may trigger events and be subscribed to simply. Additionally, the *Oculus Integration* package also uses the native event emission system, allowing greater interface between Virtual Music and the *Oculus Integration* package.

Virtual Music includes a large variety of events that may be used to create limitless functionality. The events in Virtual Music are:

- `Environment`
  - `BaseFrequencyChanged`
  - `OctavesChanged`
  - `OctaveDivisionChanged`
  - `RelativeHeightsChanged`
  - `PreciseHeightsChanged`
  - `FrequenciesChanged`
- `Note`
  - `Activated`
  - `Deactivated`
  - `Reactivated`
  - `Redeactivated`
  - `Finalised`
  - `Generated`

  - `Failed`

- `Piece`
  - `Generated`
  - `Finalised`
  - `Destroyed`
- `SettingsBlock`
  - `Generated`
  - `Invoked`
- `Stave`
  - `Generated`
  - `Finalised`
- `VMManager`
  - `Paused`
  - `Unpaused`
  - `Destroyed`

These events are `static` events, which allows a developer to provide methods to these events through the event's class without requiring instancing or specific object references. For example, note events are easily subscribed to through the `Note` class, and similarly, events related to pieces may be subscribed to through the `Piece` class.

Events of classes representing notational objects or concepts will pass a parameter of the object that emitted the event. These include `Note`, `Piece`, `SettingsBlock`, and `Stave`, each of which pass the object that emitted the event as a parameter. These

parameters can be used to check for equivalence, be used to modify the object that is passed, adjust functionality of the method depending on the status of the object, or be simply ignored.

Subscribing to an event—rather, supplying a method to an event which will be invoked when that event is emitted—is done by adding a reference to a method which is to be invoked to the event. Unsubscribing can be achieved by subtracting the previously supplied method. This can be seen in Code Sample 3.13, which supplies an `AddToScore` method to the `Note.Activated` event. The signature of the method must match the event, so events which emit a parameter must be supplied methods which take the expected parameter. In this case, the `AddToScore` method takes a `Note` parameter.

```
using UnityEngine;
using VirtualMusic.Notation;

public class SubscribeToEvent : MonoBehaviour
{
    public int Score = 0;

    private void AddToScore(Note note)
    {
        if (note.TimesActivated > 1)
        {
            return;
        }

        Score++;
    }

    private void OnEnable()
    {
        Note.Activated += AddToScore;
    }

    private void OnDisable()
    {
        Note.Activated -= AddToScore;
    }
}
```

Code Sample 3.13: Subscribing to an event

The event names are descriptive and the name of each event is indicative of when that event is emitted. For example, the `Environment.FrequenciesChanged` event is emitted when the current set of frequencies is changed, likely after the base frequency has been

changed, which will similarly emit a `BaseFrequencyChanged` event (see *Fundamental Frequency, F*). Likewise, `Note.Activated` events are emitted when a note is activated, and `Finalised` events for notes, staves, and pieces are emitted when their respective object has been finalised[11]. For the latter, if a piece has multiple staves, the piece's `Finalised` event will only be emitted once all staves have emitted a `Finalised` event.

The reactivation or re-deactivation of a note may be detected by using the `Activated` or `Deactivated` event and running a comparison on `TimesActivated` field. However, the `Reactivated` and `Redeactivated` events are provided for convenience.

[11]When an object has passed the performance area it is deemed to have been finalised.

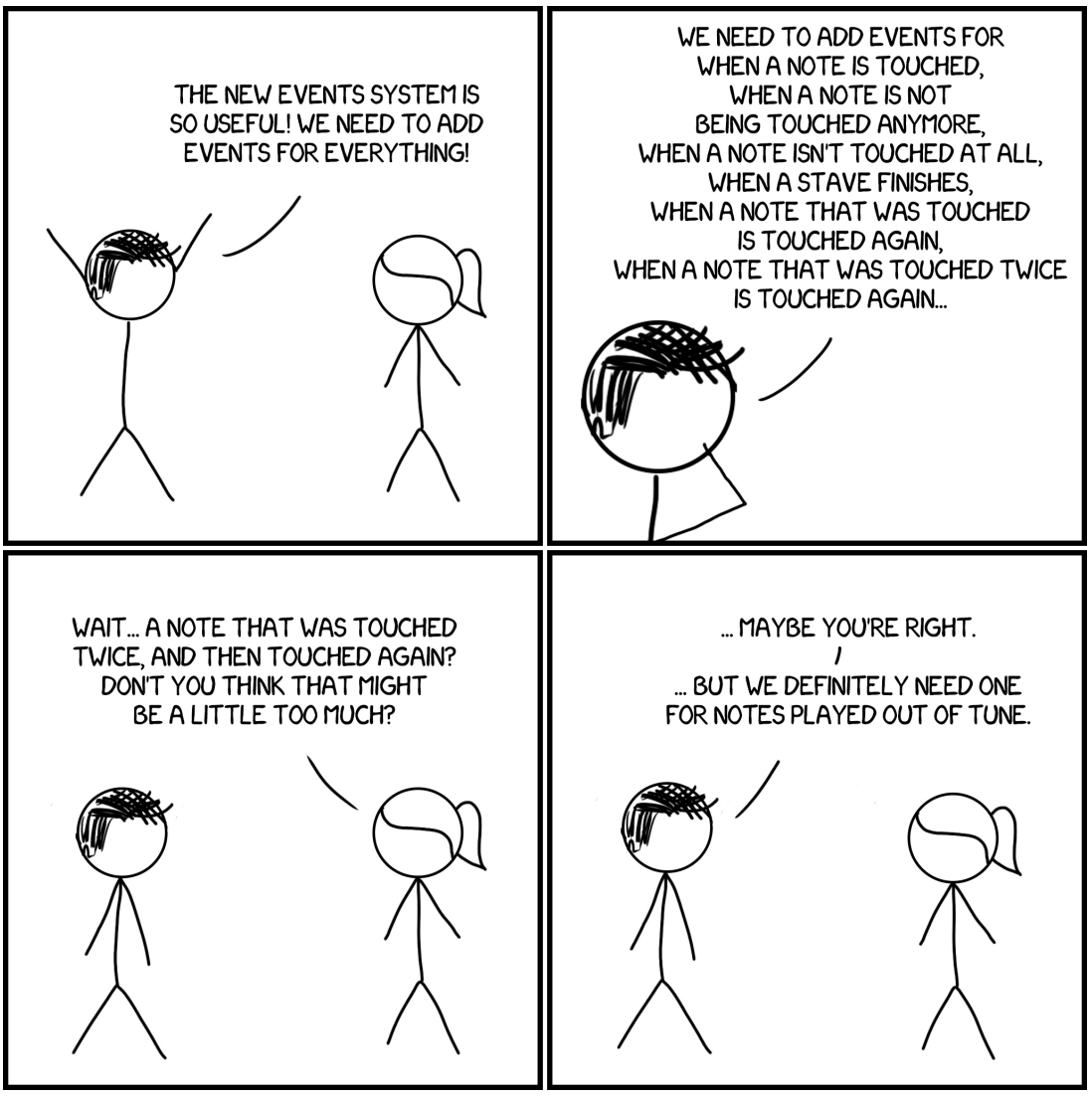


Figure 3.9: How many events is too many events?

4 | Case Studies

Two Example Implementations

4.1 Case Study 1: Gaming

When I spoke with colleagues about how our projects were progressing, VIRTUAL MUSIC was often compared with two things: a theremin, and a music rhythm video game. Perhaps coincidentally, these colleagues often fell into two distinct categories: those who knew what a theremin was, and those who knew what a music rhythm video game was.

Inspired by these comments, I decided to implement VIRTUAL MUSIC into a music rhythm video game. In order to do this, I first needed to consider common features of these games and how I could best implement them with VIRTUAL MUSIC.

4.1.1 Background

Players of music rhythm video games currently have the choice between many different forms of game. For example, some video games implement rhythm as a form of player control. These range from *Donkey Kong Jungle Beat* (Nintendo EAD Tokyo) (a video game where the player controls the protagonist with correct hits on a bongo peripheral), to *Crypt of the NecroDancer* (Brace Yourself Games) (a video game which increases the protagonist's abilities if the player correctly times their button presses with the music). Other video games, such as *Guitar Hero* (Harmonix) and *Rock Band* (Harmonix), give the player instrument peripherals so that they can recreate music performances. *Guitar Hero* is generally considered the first commercially successful music rhythm video game (Francisco). Learning from their previous success with *Karaoke Revolution* (Harmonix), Harmonix breathed new life into and epitomised the genre with *Guitar Hero* (Taylor). Its success can largely be attributed to it being an accessible form of music playing that

rewarded and immersed players with a set of unique features. These features inspired many future *Guitar Hero*-esque video games and were common throughout the *Guitar Hero* series. Features common in many games of the *Guitar Hero* series include a score counter, multiplier (effected by the 'combo meter', or 'combo counter'[1]). Many video games of the *Guitar Hero* series also include a special *Star Power* feature that allows the player to quickly increase their score. These features encourage the player to correctly perform the music in order to achieve the highest score. The scoring system, in particular, was instrumental[2] to *Guitar Hero*'s success, spawning multiple competitions, *YouTube* videos, and even its own *Guinness World Record* category (Guinness World Records Ltd.). Speaking of the formation of Harmonix and of *Guitar Hero*, journalist Eric Francisco writes:

> Over a decade later, they would crack the code with *Guitar Hero*, released on the PlayStation 2. The *very*[3] accessible pick-up-and-play nature skyrocketed *Guitar Hero* to success.
>
> It's not hard to see the appeal. It was air guitar given a physical form, a suburban past-time with tangible scores that can be measured. And, like *Donkey Kong* and its Atari forefathers, *Guitar Hero* lived up to the essence of gaming: achievement without achieving. You didn't sell out Madison Square Garden, but you broke your own high score on Black Sabbath's 'Iron Man'. (Francisco)

Features enabling immersion and realism are also common throughout the series. In particular, four features targeting realism are prevalent: first, most levels of *Guitar Hero* will place the player on a virtual stage so that the player can better feel a part of the performance; second, if a player does not play a note, the prerecorded audio that

[1]The term 'combo' refers to the number of consecutive notes played correctly.
[2]Pun intended.
[3]Original emphasis.

represents the player's instrument does not play; third, a guitar-like sound effect is played to inform players that they pressed a button incorrectly; fourth, dynamic crowd noises change according to the player's performance. If they are playing well, the crowd will cheer and shout. If they are playing badly, the crowd will show their displeasure by 'booing' and calling out phrases such as 'you suck'. If the player performs *very* badly, the performance will be forced to stop. How the crowd will react to the performance is reflected in a 'Rock Meter', seen in the bottom right corner of Figure 4.1.



Figure 4.1: The *Guitar Hero* performance interface

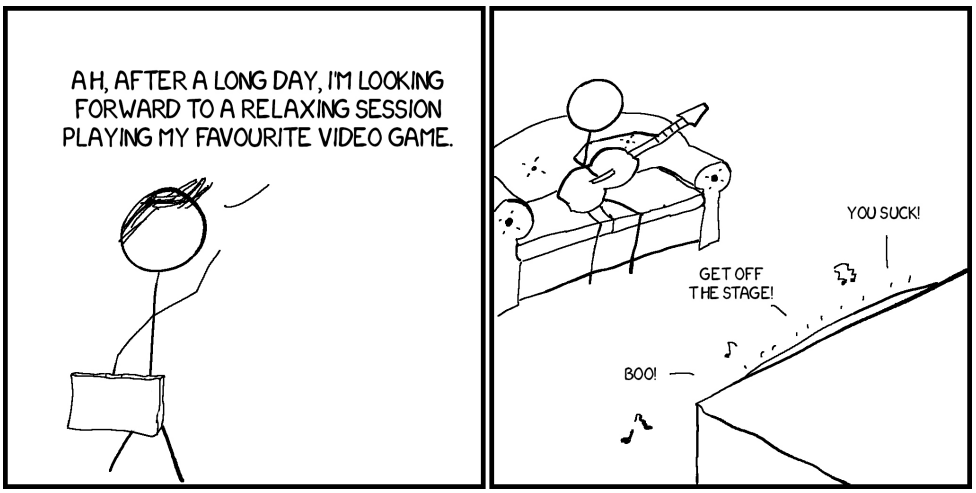


Figure 4.2: *Guitar Hero*

4.1.2 Summary of Desired Features

From this analysis, features common in the *Guitar Hero* series can be placed into two categories: scoring, and realism. They are summarised in Table 4.1. While these features are in two distinct categories, their functionality overlaps. For example, the sound effects—such as the reactions of the crowd—are dependent on the combo counter.

Scoring	Realism
• The score counter • The score multiplier • The combo counter (the 'Rock Meter')	• The performer is placed in a replica stage in front of a replica crowd • The crowd reacts to how well the performer is playing • Sound effects are played if the performer presses the wrong buttons • Sounds are not played when the performer misses a note

Table 4.1: A summary of features common in the *Guitar Hero* series

All of these features are easy to implement in Virtual Music with the events system (see section 3.11). In particular, the `Note.Activated` and `Note.Failed` events can be used to replicate the scoring system, sound effects, and features related to the crowd. The second realism feature (preventing the prerecorded audio from playing when the player misses a note) is similar to how the Virtual Music Instrument functions in the Simple and Intermediate Performance Modes without modification: the instrument will only produce a tone when a note is touched. Other features related to realism could be implemented directly in *Unity*. The replica stage, for example, can be created in *Unity*.

4.1.3 Creating the Video Game

Inspired by the rock and metal music from *Guitar Hero*, I decided that I would compose a metal piece for this case study. I was heavily inspired by the music of *Doom* (id Software). This inspired the play on and reference to the word 'doom' in the title of the work, *Peril (But Not Quite Doom)*. To support this choice of music genre, a music festival stage setting seemed appropriate. In this environment, I would be able to add a crowd and crowd noises. This meant that I first needed to set up the scene in *Unity*.

Creating the Environment

Rather than designing a stage in a 3D modelling application, I chose to create the environment relatively simply with *Unity*'s inbuilt geometric shapes. I used a cube for the floor, ceiling, and back wall, two cylinders for poles holding the stage up, four cubes to represent speakers on the poles, and a few point lights in order to provide basic lighting. I then created a plane to represent the ground and three additional cubes to represent a fence enclosing the space. The finished stage is seen in Figure 4.3.

Next, I wanted to create audience members. Audience members would improve realism and give the performer a better sense of immersion by making it seem as though they were performing in a live environment. I created a simple audience member and saved it as a `Prefab`, seen in Figure 4.4. I will not be receiving any sort of design award for the model.

I made the audience member a `Prefab` as `Prefab`s can be easily reused. As Unity Technologies describes:

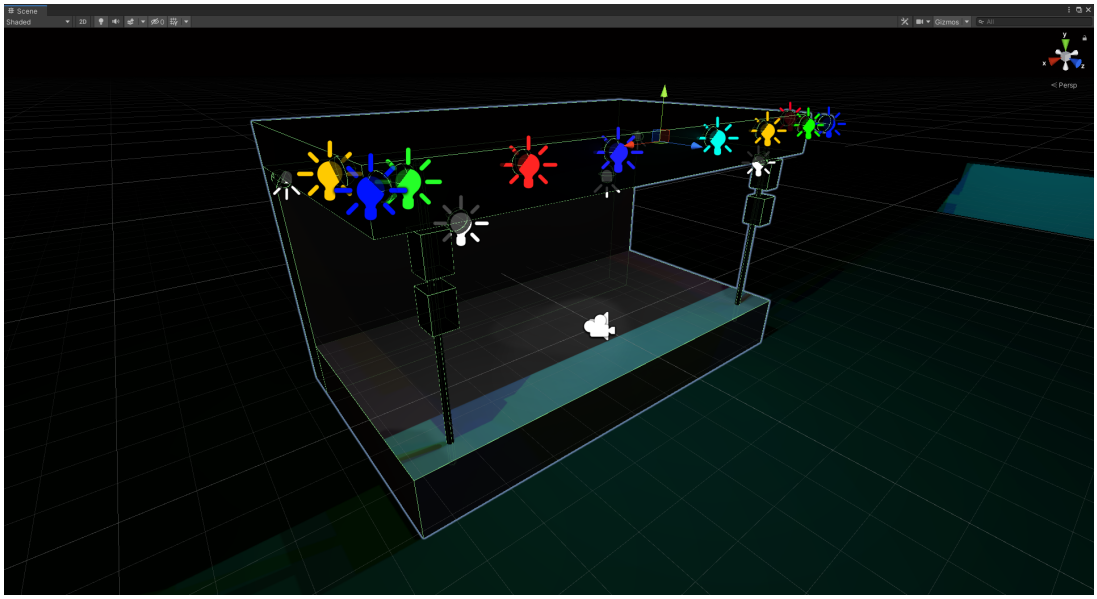


Figure 4.3: The finished stage

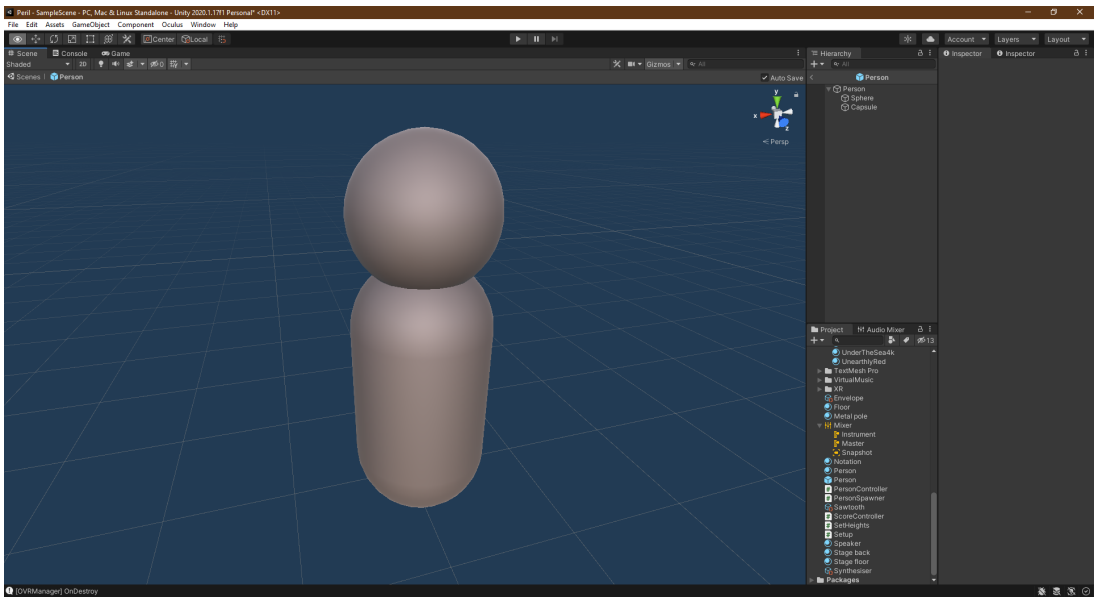


Figure 4.4: The audience member prefab

> Unity's Prefab system allows you to create, configure, and store a GameObject complete with all its components, property values, and child GameObjects as a reusable Asset. The Prefab Asset acts as a template from which you can create new Prefab instances in the Scene. (Unity Technologies, 'Prefabs')

I used a `Prefab` for two reasons: first, I could reference the `Prefab` in a `PersonSpawner` script[4] to simplify the process of placing audience members; second, `Prefab`s allow the addition of components and scripts to them. This allowed me to create a script that would be on every audience member. I created a `PersonController` script that served two purposes: first, the script randomises the colours of the object's body and head to add a bit of variety; second, it makes the object jump in time with the music. The latter functionality was relatively simple to implement with *Unity*'s `Rigidbody` component (Unity Technologies, 'Rigidbody'). Specifically, by using a `Rigidbody` component, I was able to use the `AddForce` method ('Rigidbody.AddForce'). If a force is applied to an object on its Y axis, the `AddForce` method can make it appear as though the object is jumping. The `AddForce` method can be seen used in conjunction with the `Note.Activated` event of VIRTUAL MUSIC in Code Sample 4.1, which is a fragment of the `PersonController` script. `AddForce` was used in conjunction with the `Note.Activated` event as the audience could be made to jump when notes were touched, giving the impression that the audience was jumping in time with and excited by the music. They would also not jump if the performer was playing badly, giving the performer feedback about their feelings about the performance.

[4]This script can be seen at Appendix U.

```
using UnityEngine;
using VirtualMusic.Notation;
using Random = UnityEngine.Random;

[RequireComponent(typeof(Rigidbody))]
public class PersonController : MonoBehaviour
{
  // the first portion of the script stores many of the below variables
  ...
  private void Jump(Note note)
  {
    // first check if the object is touching the ground and also check if the
    // object has already been told to jump this frame
    if (isGrounded && Time.time > lastCalled)
    {
      // set the jump force, varying it if necessary
      var force = shouldVaryJumpForce
        ? jumpForce + Random.Range(jumpForceVariance.x, jumpForceVariance.y)
        : jumpForce;

      rb.AddForce(new Vector3(0f, force, 0f), ForceMode.Impulse);

      // set the lastCalled time to the current frame time so that touching multiple
      // notes does not make the object jump too high
      lastCalled = Time.time;

      // the object is jumping, so it is no longer grounded
      isGrounded = false;
    }
  }

  // the OnCollisionEnter method is invoked when the
  // object touches another object with a Collider component
  void OnCollisionEnter(Collision collision) => isGrounded =
      collision.gameObject.CompareTag("Ground");

  private void Start()
  {
    // add the jump function to the Note.Activated event
    Note.Activated += Jump;

    // store the RigidBody component in a variable
    rb = GetComponent<Rigidbody>();

    // add a bit of randomness to the initial jump force
    // to add a little more realism
    if (shouldRandomiseInitialJumpForce) jumpForce = Random.Range(initialJumpForceRange.x,
        initialJumpForceRange.y);
    ...
  }
  ...
}
```

Code Sample 4.1: Using the AddForce method of RigidBody with Note.Activate to make an object jump

The Scoring System

Once the environment and audience was created, I could focus on the scoring system. The scoring system of *Guitar Hero* encourages the player to correctly play the specified notes through the use of a combo counter and score multiplier. If a player correctly activates consecutive notes, the combo counter increases, which, in turn, increases their score multiplier. If the player fails to activate a note, the combo counter and multiplier will reset to zero and one respectively. The scoring system is also used by the system that controls the audience's reactions. For example, if the combo counter is high, the audience will react positively. If a player fails multiple notes in a row, the audience will react negatively.

These features were implemented via the `Note.Activated` and `Note.Failed` events in a `ScoreController` script. The `ScoreController` script maintains the current score and updates a 'score screen' accordingly, as seen in Figure 4.5. It does so via two methods: the `AddScore` and the `FailNote` methods. These methods add to the player's current score and maintain the combo counters. The `FailNote` method has additional functionality: it activates a random sound effect when the player has missed a note. Both methods can be seen in Code Sample 4.2.

The `ScoreController` is also responsible for other functionality related to the audience. For example, a `FirstNote` method is called when the first note is touched. The `FirstNote` method decreases the volume of an ambient audience sound file, and increases the volume of a sound file of a cheering audience. The script also controls other reactions of the crowd via the `excitement` field. This field controls the volume of a looping sound file of a cheering audience. The field also activates certain sound effects depending on its value. For example, if the `excitement` field reaches its maximum value, the `ScoreController` script will trigger positive crowd sound effects. These



Figure 4.5: The score screen

```
private void AddScore(Note note)
{
  // if the note has already been activated, the score should not be increased
  if (note.TimesActivated > 1) return;

  success++;
  combo++;
  failedCombo = 0;
  excitement += excitementModifier;
  score += baseScore * multiplier;
}

private void FailNote(Note note)
{
  failed++;
  combo = 0;
  failedCombo++;
  excitement -= excitementModifier;

  // play a random sound effect to give the performer feedback about their bad playing
  sfxSource.pitch = Random.Range(0.95f, 1.05f);
  sfxSource.PlayOneShot(failedSfx[Random.Range(0, failedSfx.Count - 1)]);
}
```

Code Sample 4.2: The AddScore and FailNote methods

positive sound effects will continue playing while the `excitement` field is at its maximum value (shown to the player as 100 per cent). Similarly, if the `excitement` field goes below its neutral value (shown as 50 per cent), the audience will stop cheering. Furthermore, if the `excitement` field reaches its minimum value (shown as zero per cent), the audience will start booing, and, if the field stays at its minimum value for a predetermined amount of time (set via the `timeUntilStop` field), the performance will be forced to stop. This dynamic audience functionality is seen in the video recording provided at Appendix X.

Other Virtual Music Features

Once the functionality of the scoring system was finalised, I considered performance elements that could better immerse the performer or assist with making the piece fit within the metal genre. First, I assigned the Virtual Music Instrument to a mixer in order to provide reverb (see subsection 3.7.8). This was simply done to make the instrument sound better and as though it was placed in the environment. Second, I considered instrumentation. Additional instruments could make the piece more interesting and similar to the soundtrack of *Doom*. The instrumentation of the *Doom* soundtrack commonly includes guitar, bass guitar, and percussion, usually in the form of a drum kit. The Virtual Music Instrument can be made to sound like a guitar through a carefully created synthesiser (see subsection 3.8.3). I used a sawtooth wave oscillator that had an envelope with a high attack value applied to it to replicate the sound of a guitar from an early 1990s video game. Adding a bassist to the piece was simple through the use of an additional stave that acted as a bass line. While I could have simply created a drum track that would be played underneath the Virtual Music Instrument and in time with the music, it is possible to create a simulated drum kit with flagged actions (see subsubsection 3.3.14). Doing so meant that the drum

kit would only play when the notes were activated, and if the performer was not playing well, it would also sound as though the drummer was playing at a similar level. I used four flags: 'b', 's', 'h', and 'c'. These flags activate audio samples of a bass drum, snare drum, hi-hat, and crash cymbal, respectively. To improve immersion, I created an additional `AudioSource` component for the drum kit. This made it possible to place the virtual drum kit on the stage, so that it sounded as though it was physically behind the performer (Unity Technologies, 'AudioSource'). Additionally, the `AudioSource` was then routed through the same mixer as the Virtual Music Instrument so that the reverb effect was also applied to the drum kit. A complete video recording of the performance of *Peril (But Not Quite Doom)* in provided at Appendix W.

4.2 Case Study 2: Art and Experiential Music

VR applications—especially those designed to showcase an art form—are often called 'experiences'. The store to purchase applications for the *Rift* by *Oculus*, for example, uses the term to headline its applications (Facebook Technologies, 'Rift Store: VR Games, Apps and More'). The use of the term in relation to VR can be traced to the 'Experience Theater' created by cinematographer Morton L. Heilig in 1969. The 'Experience Theater' allowed users to 'experience realistically the full effect or illusion of being a part of or physically responding to the environment depicted by the motion picture' (1) through visual displays and systems of air ventilation that immersed the viewer as seen in Figure 4.6.

Like the 'Experience Theater', experiential applications for VR are often simple to control and allow the user to immerse themselves in the virtual environment. This ability

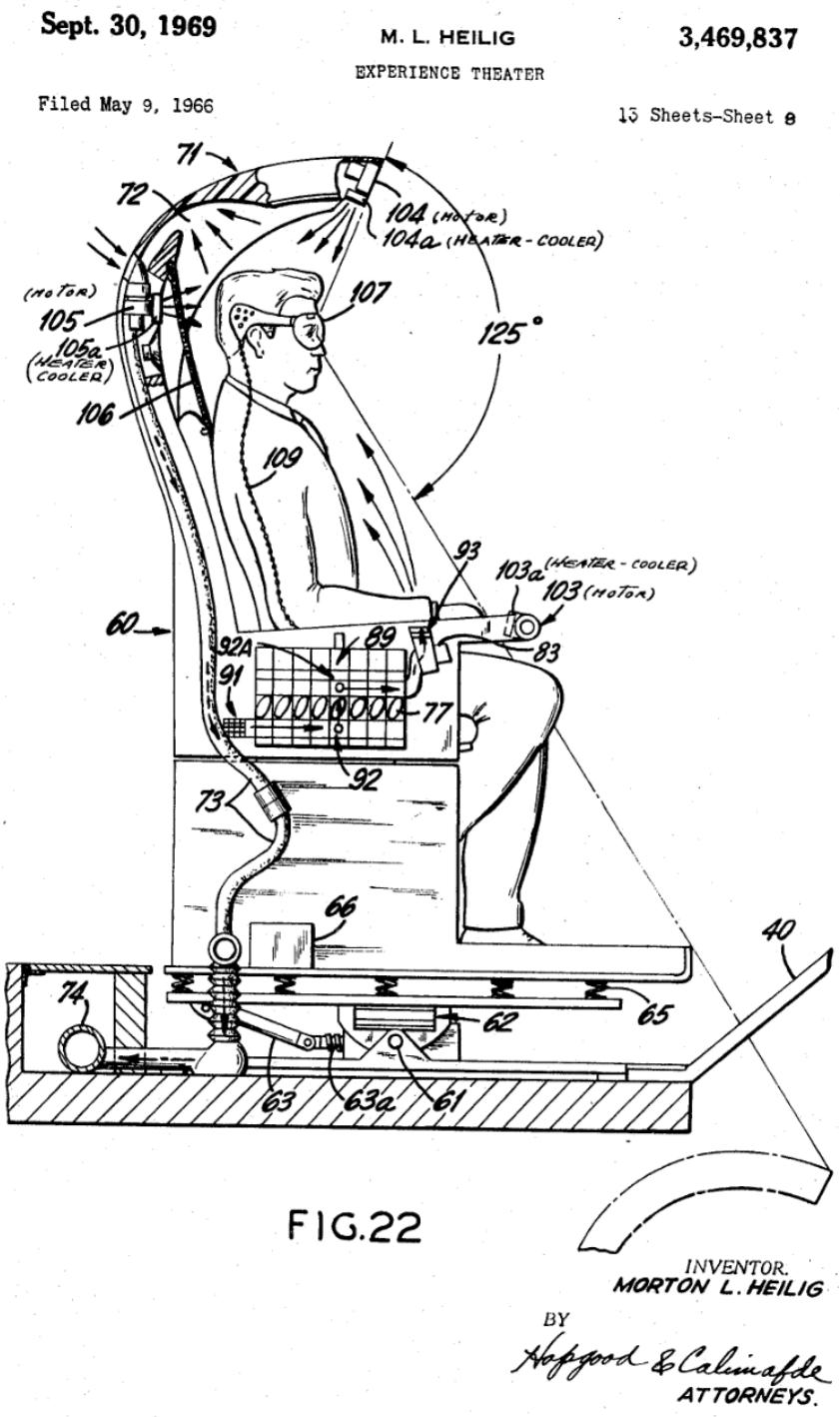


Figure 4.6: The 'Experience Theater' (Heilig 8)

to immerse the viewer gives VR a unique ability to better portray the narrative of the work. A famous example is *Henry*, produced by Oculus Story Studio in 2016. The application is an experiential application showcasing the use of VR in narrative film. A hedgehog—narrated by Elijah Woods and pictured in Figure 4.7—learns on his birthday that his love of hugging is not one that everyone shares with him. The simplicity of controls is evident in the fact that the application recommends that the user sit on the floor and only gives instructions on how to navigate the menus. Once the film has started, the controls are no longer necessary. This simplicity better supports the viewer in becoming immersed in the virtual environment as they can simply *watch* the film and not have to remember any buttons or control systems. This is a trait shared by many narrative experiences in *Oculus*'s application store, including *Lost*, another VR film by Oculus Story Studio in 2016, which also has no method of control.



Figure 4.7: Henry and his birthday cake

These simplified control systems make experiential applications excellent for use in art exhibitions. This is because frequently changing users—who may not be familiar with VR—have limited time to become familiar with complicated control systems. Indeed, VR experiences are already being included in exhibitions in galleries and museums. Visitors

to the *Musée du Louvre* (the Louvre Museum) in late 2019, for example, could have experienced *Mona Lisa: Beyond the Glass*, which offered 'visitors a chance to have a heightened personal encounter with the [Mona Lisa] artwork as was never before possible' (VIVE Arts). Jake Cigainero discusses many examples of the use of VR in museums, including at *Kansallis Museo* (the National Museum of Finland), that created a VR experience of *The Opening of the Diet 1863 by Alexander II* (Ekman) allowing visitors to 'speak with the Russian emperor and other depicted characters' (Cigainero). The VR experience was accessible to visitors of all ages:

> Hanna Forssell, head curator of education, expected it to appeal mostly to the under-18 set that makes up one-third of the museum's entries. Instead, she was surprised by the interest from all ages.
>
> When the feature started recently, she said, 'There was a long line of mainly older people over 60'. (Cigainero)

VR experiences that showcase music, however, are less prominent and tend to fall into three different categories: first, rhythm games such as the famous *Beat Saber* and *Audioshield* (Beat Games; Fitterer); second, applications that encourage improvised performance on musical instruments such as *Instrument Studio VR* and *Percussive VR* (SkyWatcherVR; Jamhack Games); and third, experiences that encourage listening and immersion, such as *Siren Song* (which features original music by French composer and director Denis Volte) (Mysteo), and recorded concert experiences such as *Encore in VR* by the Adelaide Symphony Orchestra (Niscioli). While applications in the second category especially would be capable of producing performances—and, indeed, *Percussive VR* includes a music stand with sheet music—their intent is not for performance, but for experience.

The intuitive controls of Virtual Music give excellent potential for use in such VR

experiences. A piece could very easily be composed for a VR experience which focuses on narrative and immersion. Importantly, it would be accessible for performers of all ages and to developers: most—if not all—tools required in order to produce a VR experience are free and easy enough to use. To prove this, I chose to translate *Yurndu akanatha Oratunganga* (see page 5) into an exhibition-like experiential piece for VIRTUAL MUSIC using only free and accessible resources in *Unity*.

4.2.1 Planning and Creating Key Functionality

Before beginning composition, I first planned what I wanted the user to experience. *Yurndu akanatha Oratunganga* depicts a person waking at sunrise to the sounds of birds and other creatures. The sounds build up into a 'glorious euphony' before reducing back down to the single sound again once the sun has risen. This implied four components: the bird calls, the sunrise, the environment, and the narration.

The Bird Calls

I decided to use a single bird call as the main motif. The bird call I composed was inspired by the call of a Singing Bushlark and can be seen in Figure 4.8a. Additional calls representing other animals are used later in the piece. Most notably, the call of a Crow is introduced in the second stanza and can be seen in Figure 4.8b. The Bushlark motif is developed especially in the middle section by extending it to form a melody. I wanted the musical material to be relatively simple so that the performer could better immerse themselves in the virtual environment without becoming distracted by the music. These short motifs could be developed easily and repeated in short segments to achieve this immersion.



(a) the Bushlark motif

(b) the Crow motif

Figure 4.8: The main musical motifs for *Yurndu akanatha*

The Sunrise

Next was the sunrise. A sunrise can be created through the use of *Unity*'s directional lights. Every new project in *Unity* starts with a directional light in order to provide basic lighting. They are designed to behave similarly to a sun:

> Directional lights are very useful for creating effects such as sunlight in your scenes. Behaving in many ways like the sun, directional lights can be thought of as distant light sources which exist infinitely far away. […] Directional lights represent large, distant sources that come from a position outside the range of the game world. In a realistic scene, they can be used to simulate the sun or moon. (Unity Technologies, 'Types of light')

In *Unity*, a skybox is used to 'give the impression of complex scenery at the horizon' (Unity Technologies, 'Skybox'). Usually, the scene and cameras of a *Unity* project are placed within the skybox so that the surrounding view is made visually interesting. The skybox is usually textured with a material or shader that represents a sky. However, the texture may also be set to any carefully designed texture representing a different environment, to give the impression that the scene is within that environment. The texture of the skybox, for example, could be set to represent a furnished room to give the

impression that the scene is within the room. It could also be set to a cosmic pattern to give the impression that the scene is in space, as seen in Figure 4.9.

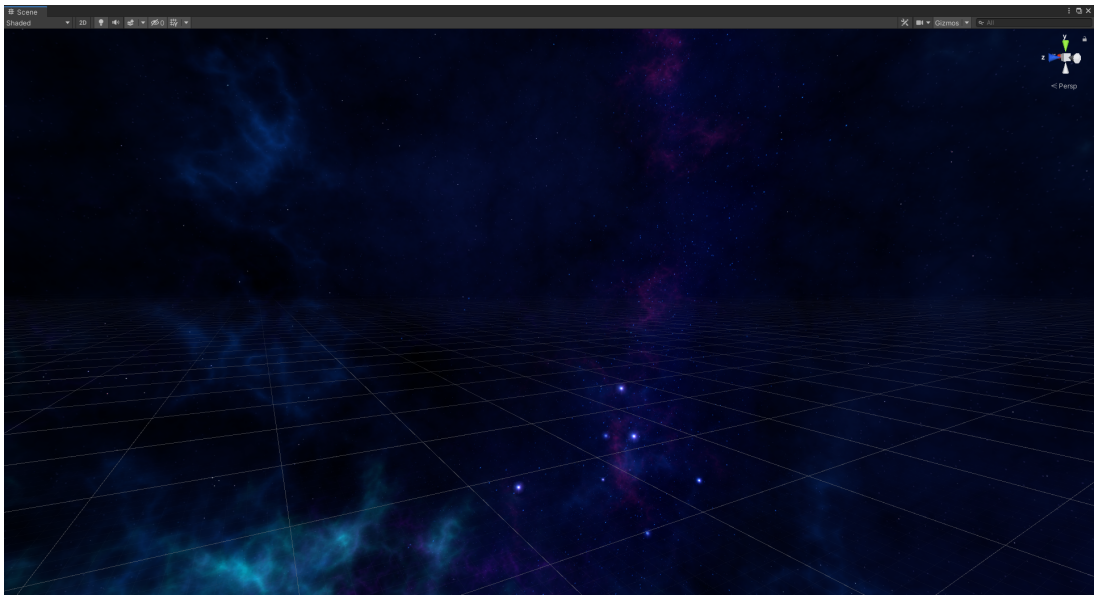
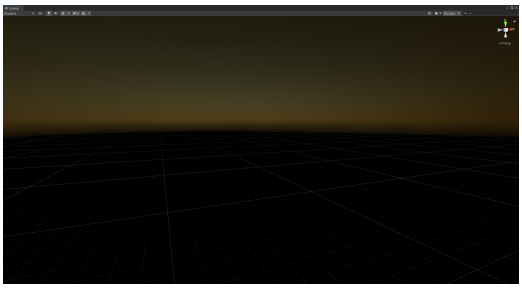
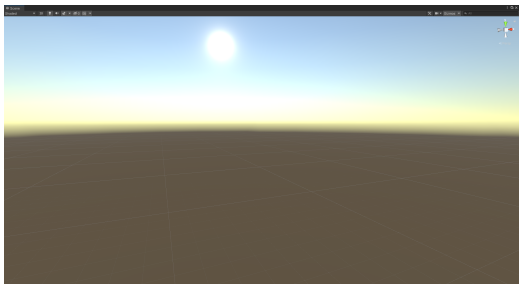


Figure 4.9: A skybox with a cosmic texture

Recent versions of *Unity* come with a 'procedural skybox' `Shader`. This `Shader` 'does not require any input Textures and instead generates a skybox' (Unity Technologies, 'Procedural skybox'). It is designed to be used with a directional light. If a directional light is used in conjunction with a procedural skybox, it will appear as a sun via a 'sun disk'. The colours of the procedural skybox and the lighting of the scene changes depending on the rotation of the directional light. If the rotation of the directional light makes the light point upward, the skybox and scene will be dark. If the rotation makes the directional light point downward, the skybox will resemble a sky and the scene will be light. The procedural skybox scatters the light of the sun depending on the `Atmosphere Thickness` property of the `Shader`. This means that when the directional light points sideways, the light has to travel through more atmosphere and the procedural skybox will remove the shorter wavelengths of the visible light spectrum, leaving predominantly red or orange coloured light. This creates an effect that mirrors how the light of the sun interacts with the atmosphere of the Earth and can be seen in Figure 4.10.



(a) a directional light rotated so that it points upwards



(b) a directional light rotated so that it points downwards

Figure 4.10: Demonstrating the interaction of a directional light with a procedural skybox Shader

Importantly, the rotation of the directional light may be adjusted dynamically. This ability to be dynamically rotated can be used with the procedural skybox `Shader` to create a day-night cycle:

> If you choose to render a sun disk in your skybox[…], this Shader uses the rotation of the active Light to position the sun in the skybox. […] You can use this behaviour to create a simple day-night cycle. To do this, continuously rotate your main Directional Light around a particular axis. (Unity Technologies, 'Procedural skybox')

The directional light can be dynamically rotated by using the `Quaternion.Lerp` method of the `UnityEngine` namespace, which is used to smoothly transition the rotation of an object between two values (Unity Technologies, 'Quaternion.Lerp'). I could use this method in a simple script to create a convincing sunrise by rotating the sun according to the progress of the piece and defined duration and rotation values. A simple implementation of the `Quarternion.Lerp` method to create a sunrise can be seen in Code Sample 4.3.

```
using UnityEngine;
public class RotateSun : MonoBehaviour
{
    public GameObject sun;
    public float durationInSeconds = 300f;
    public Transform sunStartPosition;
    public Transform sunFinishPosition;

    private void Start()
    {
        sun.transform.rotation = sunStartPosition.rotation;
    }

    private void Update()
    {
        sun.transform.rotation = Quaternion.Lerp(sunStartPosition.rotation,
            sunFinishPosition.rotation, Mathf.InverseLerp(0, durationInSeconds,
            Time.fixedTime));
    }
}
```

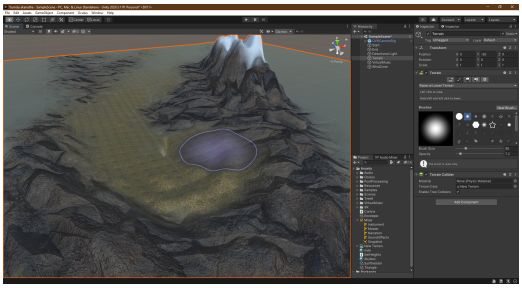
Code Sample 4.3: A script that smoothly rotates the sun between two positions

The Environment

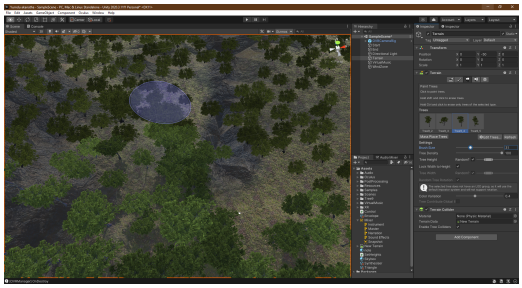
Now that I had a method to create a convincing sunrise, I needed an environment that the sun could rise in. Recent versions of *Unity* include a set of tools to create landscapes. They include methods to adjust the height of terrain, change the texture of parts of the terrain, and create trees, grass, and other decorative elements. These tools combined can make convincing and realistic terrain of any type, including mountains, plains, oceans, and forests (Unity Technologies, 'Terrain').

While new projects can be configured to start with some basic provided materials and objects, additional materials can be downloaded for free from the *Unity Asset Store* (Unity Technologies, 'Unity Asset Store'). Notably, to create an interesting environment, textures for grass, rock, and other surfaces are needed so that the terrain can look more natural and interesting. Fortunately, Unity Technologies has created the *Terrain Tools Sample Asset Pack* as a free asset in the *Unity Asset Store*. It includes a collection of samples and brushes to assist in creating realistic and convincing terrain. These tools and textures can be seen in Figure 4.11a. Trees and rocks are also available for free in the

Unity Asset Store. I chose the *Realistic Tree 9 [Rainbow Tree]* asset by Pixel Games, which included five different types of trees. These assets can be added to *Unity*'s terrain tools and 'painted' on in a similar manner to how the terrain's textures can be added, as seen in Figure 4.11b. Unfortunately, I could not find trees that looked similar to those that can be found in Oratunga. For this reason, I decided to remove reference to Oratunga in the title of the work. It was possible for me to design a tree that looked similar to the trees in Oratunga with a free 3D modelling application such as *Blender*. For the purposes of this work—which was to demonstrate the capabilities of VIRTUAL MUSIC in a compelling VR experience only using free resources—I decided that this was not necessary.



(a) setting the height of and adding textures to the terrain



(b) adding trees to the environment

Figure 4.11: Creating the environment

To increase immersion, I also included a wind zone and included ambient bird and animal noises. Wind zones effect trees and some decorative objects by bending them 'in a realistic animated fashion'. Additionally, the wind itself 'pulses' to create a natural movement pattern by varying its speed in gusts (Unity Technologies, 'Wind Zones'). To include the bird noises, I attached an `AudioSource` to the `OVRCameraRig` so that the sound would encompass the performer. I found a free-to-use recording of birds and simply added it to the `AudioSource` component and ensured that `Loop` was enabled, so that the bird sounds looped continuously.

The Narration

The final component of the experience was the narration. I considered whether to include the original poem as a recording or as on-screen text. I decided to include the narration as prerecorded audio that would be played between phrases to accompany the music. The recorded phrases were set to pan randomly, making better use of the virtual environment. That is, the audio would sound as though it was coming from someone speaking near the performer, and the location of the speaker would change. This adds an additional unique layer of immersion and interest to the experience, only made possible because of Virtual Music the capability of VR technologies to produce binaural audio. I used the application *Cubase* by Steinberg to record the audio as I am familiar with its functions. However, the audio could easily have been recorded to the same degree of quality with free software such as the feature-rich *Audacity* (The Audacity Team).

4.2.2 Implementing Virtual Music

After the piece had been planned and the basic functionality of the experience created, implementing Virtual Music was quite simple. First, I chose to split the recorded audio into 18 different segments. These segments would be played by a method invoked by a note flag (see subsubsection 3.2.3). Second, I created a material and enabled emission on that material so that the notes could be seen in the dark, when the sun had not yet risen (see subsubsection 3.3.8). Third, I altered the location of the stave's first note to a position that was further away (see subsection 3.3.5). This gives the performer more time to look around at the beginning of the piece, allowing them to immerse themselves in their surroundings before the piece starts. Fourth, to support this, I also changed the location of the information slide so that it appeared earlier than the default (see subsection 3.3.10). This makes it more obvious that the immersive qualities of the experience are part of the

piece and music itself. Fifth, I implemented a mixer and assigned the VIRTUAL MUSIC INSTRUMENT to it so that I could add a reverb effect (see subsection 3.7.7). This better supported the environment by making the instrument feel like it was a part of it.

The required script to control the performance was very simple to create. It rotates the sun via the `Quaternion.Lerp` method as seen in Code Sample 4.3. Additionally, it adds the note flag methods to the piece by iterating through the collection of audio files and adding a `FlaggedFinalisationAction` per file. This could have been done manually, but I chose this method so that the script did not have to be altered if I chose to further split the audio files. The script in its entirety can be seen in Code Sample 4.4. It is heavily documented with comments and XML Documentation in order to provide commentary of its functionality. A complete video recording of the performance of *Yurndu akanatha* is provided at Appendix W.

```
using System.Collections.Generic;
using UnityEngine;
using VirtualMusic.Notation;

/// <summary>
/// This class controls the sun and narration for Yurndu akanatha
/// </summary>
public class Control : MonoBehaviour
{
    /// <summary>
    /// The <see cref="AudioSource"/> that will play the narration
    /// </summary>
    [SerializeField] private AudioSource narrationSource;

    /// <summary>
    /// The <see cref="VirtualMusic.Notation.Piece"/> being performed
    /// </summary>
    [SerializeField] private Piece piece;

    /// <summary>
    /// The <see cref="GameObject"/> that the sun (the DirectionalLight) is attached to
    /// </summary>
    [SerializeField] private GameObject sun;

    /// <summary>
    /// How long it will take for the sun to rise in seconds
    /// </summary>
    [SerializeField] private float durationInSeconds = 300f;

    /// <summary>
    /// The position and rotation of the sun at the start of the performance
    /// </summary>
    [SerializeField] private Transform sunStartPosition;

    /// <summary>
    /// The position and rotation of the sun after the time specified in <see
        cref="durationInSeconds"/>
    /// </summary>
    [SerializeField] private Transform sunFinishPosition;
```

```
39
40      /// <summary>
41      /// All of the audio files for the narration
42      /// </summary>
43      [SerializeField] private List<AudioClip> narration;
44
45      /// <summary>
46      /// Play the narration for the note's first note flag
47      /// </summary>
48      /// <param name="note">The note that has the note flag</param>
49      private void PlayNarration(Note note)
50      {
51          // check to see if the note flag is a valid integer,
52          // exit the method if not
53          if (!int.TryParse(note.Flags[0], out var flag)) return;
54
55          // check to see if the note flag has an equivalent
56          // audio clip and that the AudioSource for the
57          // narration exists
58          if (flag <= narration.Count && narration[flag - 1] && narrationSource != null)
59          {
60              narrationSource.PlayOneShot(narration[flag - 1]);
61          }
62      }
63
64      private void Start()
65      {
66          // check if the "sun" field has been set or not
67          if (sun != null)
68          {
69              sun.transform.rotation = sunStartPosition.rotation;
70          }
71
72          // check if the "piece" field has been set,
73          // exit the method if not
74          if (ReferenceEquals(piece, null)) return;
75
76          // add a finalisation action for each narration clip
77          //
78          // finalisation is used so the clip is triggered even
79          // if the performer has not touched the note
80          for (var i = 0; i < narration.Count; i++)
81          {
82              piece.FlaggedFinalisationActions.Add((i + 1).ToString(), PlayNarration);
83          }
84
85          // increase the count in time to increase immersion
86          piece.BeatsForCountIn = 16;
87
88          // set AutoPlay to true just in case its been overridden
89          VMManager.AutoPlay = true;
90
91          // generate the piece
92          piece.Generate();
93      }
94
95      private void Update()
96      {
97          // check first if the sun exists,
98          // exit the method if not
99          if (ReferenceEquals(sun, null)) return;
100
101         // make the sun rise!
102         sun.transform.rotation = Quaternion.Lerp(sunStartPosition.rotation,
                sunFinishPosition.rotation, Mathf.InverseLerp(0, durationInSeconds,
                Time.fixedTime));
103     }
104 }
```

Code Sample 4.4: The first section of the Control.cs file

5 | Conclusion

Summary and Future Work

5.1 Summary

I embarked on this project with the aim of creating an accessible form of music so that I could share the joy that is music making. I hoped to create something that was also intuitive, allowing new performers to pick up Virtual Music and start performing without training. Throughout the development of Virtual Music, I asked myself four questions. These questions have been answered throughout the thesis but are also summarised in the following sections. Since completing the planned development, a new question has been added: what is next?

Can Virtual Music enable new performers to reperform and recreate music to some degree of recognition with minimal or no training, while allowing the development of their skill and musical expressiveness?

While developing Virtual Music, I, of course, had discussions with colleagues about our respective projects. As Virtual Music combines computer sciences with music, these colleagues were a good mix of musical and non-musical. The innovative nature of Virtual Music—and the fact that VR was still new to many people—meant that some of these colleagues were interested in trying Virtual Music for themselves at various stages of development. This provided me with many opportunities to test how intuitive the control mechanisms were since I never gave them any instructions on how to perform, besides merely saying: 'touch the blocks coming at you, they are the notes'. Each was able to perform the études I had composed to some degree. Importantly, each found it easier on reperformance and those reperformances were recognisably similar to their source material and to each other. I found these colleagues an excellent source of

enthusiasm and opinion. Each gave their own take on how Virtual Music functioned and offered suggestions on how they thought it could be improved. The touch buffer (see subsubsection 3.7.2), introduced in `v0.3.0` (see subsection 2.5.1) was a good example of this, suggested by a good musician friend with a background in computer-based music production.

Ideally, these anecdotal tests with colleagues would have been a more formalised process throughout the development phase of Virtual Music, but managing play-testing and obtaining ethics clearances for such tests was tricky (at best). Therefore, I tested Virtual Music at many points throughout the development via the Performance Test process, with an aim for a more widespread testing phase once the project reached its next stages. These tests were not limited to only the Performance Tests, of course. I cannot put a number on the amount of times I strapped the HMD on my face, until I eventually found a method of holding it to my face without having to put the straps around my head. My own journey with Virtual Music also demonstrates that the system allows the development of a performer's skill, musical expressiveness, and the quality of subsequent performances. When reperforming the same étude straight after a colleague, for example, they would often say 'oh, that's what it is meant to sound like'. Indeed, the development of my ability can be seen in the progression of my playing in the recorded performances. It can also be seen by the development of the additional Performance Modes when I felt I needed them, as I expect others will, too. While I am still not at the level I wish to be when performing with Virtual Music (I have to admit that I often find myself using the Simple Performance Mode), I find myself getting better every time I use it.

As such, to say that Virtual Music can be used by new performers without training is true. This is supported by my colleagues who so kindly and eagerly gave me feedback throughout the development. Similarly, to say that Virtual Music allows the development of a performer's skill is also true. Virtual Music, with its multiple

levels of difficulty through its Performance Modes, gives a performer a 'low entry fee with no ceiling on virtuosity' (Wessel and Wright 11).

In fact, progression to a higher level can take a high amount of practice as the pitch of a produced tone can be hard to control. In this way, the Virtual Music Instrument is similar to a theremin, which is notoriously difficult to master due to the lack of any spacial reference (Martín et al. 2). This is eased by the suggested progression from the Simple Performance Mode (see subsubsection 3.7.2), though to the Intermediate Performance Mode (which does not allow complete control over the production of pitch) and finally to the Advanced Performance Mode (which does allow complete control of pitch production). The Pitch Guide (see subsection 3.7.3), introduced in `v0.3.0` (see subsection 2.5.1), also supports this progression by giving the performer the ability to see exactly where the different pitches lie. Additionally, the ability for the Virtual Music Instrument to lock the production of pitches to only those defined also supports the performer, by automatically correcting intonation when required[1].

Can the Fundamental Elements of (Virtual) Music Notation and other notation and performances conventions be adapted to VR?

This question has been answered in chapters two and three in particular. After considerable experimentation, the Fundamental Elements of (Virtual) Music Notation as described in subsection 1.4.1 were successfully adapted to VR. However, the clarity of these elements may occasionally be obscured by other elements. This was addressed by offering the composer different options for how the stave approaches the performer, as different directions were found to suit different compositions (see subsection 2.3.2 and subsection 2.3.3). The introduction of transparency also addressed this issue by allowing

[1]This functionality is enabled by default, but can be overridden with `UnlockIntervallicModeButton` (see subsubsection 3.7.1).

a performer to see later notes more clearly (see subsection 2.3.3). Features such as the ability to alter the material, and therefore appearance, also address this issue, while also giving Virtual Music extensibility. This is also seen with the ability to include and control particle effects and the ability to modify the note's heights and widths.

Other notation and performance conventions are either supported or somewhat supported. Some articulations, for example, are supported directly in the notation. Ornamentation, however, is not supported by the notation for reasons discussed in subsection 2.5.1. A performer may still ornament and express their creativity by switching Performance Modes where necessary. Other performance conventions such as the ability to notate and perform gradual or immediate tempo changes are also supported with the Settings Block (see subsection 3.2.2) and with the joystick (see subsubsection 3.7.1).

From the perspective of a composer, common desirable features, including the ability to change key, tempo and even tonal intervals (and thus allowing microtonality, for instance), are supported via Settings Blocks. As Virtual Music Notation has no concept of a bar besides how a composer groups successive notes, the ability to change time signatures is also inherently supported. The ability to change and control timbre is also supported directly through the considerable options and functionality of the synthesiser. This includes the ability to create and combine oscillators (see subsubsection 3.8.3), modify the output of an oscillator through an LFO (see subsubsection 3.8.3), envelop the amplitude (see subsubsection 3.8.3), as well as the ability to add effects to individual synthesisers (see subsection 3.7.8) and to the output of the instrument entirely (see subsection 3.6.8).

What new opportunities and limitations does VR bring to music notation, composition, and performance?

This question has two components. The opportunities and limitations brought by VR directly impact the corresponding opportunities and limitations of Virtual Music. These are reflective of the current state of the technology and will improve over time. Virtual Music, however, has opportunities and limitations of its own. To answer this question thoroughly, a distinction is made between the effects that VR has on these elements, compared to the effect of Virtual Music.

The benefits gained by the digitisation of music creation into a VR medium is, in ways, immeasurable. As Carranco describes: 'since this multi-sensorial interaction happens in a virtual environment, it has the possibility of not following the natural laws of physics' (Carranco 11). The new opportunities that VR has brought to music notation, composition, and performance are therefore solely restricted by the imagination of the composer and the abilities of the technology. This technology will only improve over time, as will our ability to use it. Most importantly, VR is inherently immersive: a user of VR, and therefore of Virtual Music, is completely engulfed in the performance of the work. This immersion, however, can limit human interaction. This limits Virtual Music in one major way: it cannot easily be used for ensemble performance, especially with other performers using Virtual Music. At this stage, there are only two ways in which it can be used in conjunction with other instruments and performers.

First, other performers may follow the direction of a performer using Virtual Music Notation. In this case, the performer becomes the conductor. The other performers must either improvise to accompany the Virtual Music performer, or they may follow the progression of the piece with performance instructions notated otherwise. In most

situations, however, the performer using Virtual Music may not see or interact with the other performers. This may be solved with the addition of a video feed in the Virtual Music application or through the addition of some sort of tracking system. This could be an interesting method to accompany a performance that makes use of specific features of Virtual Music that are not easily translatable to non-digital performance methods. For example, performers on other instruments may accompany a performance of *Yurndu akanatha* through improvisation or through notated instructions, creating an entirely new experience.

Second, the Virtual Music Instrument may be used without Virtual Music Notation. In this case, the Virtual Music Instrument becomes similar in function to a theremin. The performer using the Virtual Music Instrument must either improvise or interpret notation presented to them in VR. This could be done with descriptive improvised notations or with prescriptive non-improvised notations presented to the performer in some way. For example, a pass-through camera would allow the performer to see notation in front of them while wearing the HMD. Other notation systems may also be presented to the performer in VR, including traditional stave notation or any alternative. Since Virtual Music is already in a digital form, this method of performance could also use animated notation systems as discussed by Cat Hope (21). This could be an excellent method to allow musicians with a fear of performance to perform in an environment in which they are comfortable. Virtual Music could even assist in therapy for such a performer by changing the performance environment slowly so that the performer may become used to performing to an audience. Additionally, since the Virtual Music Instrument uses synthesisers for its sound production, it has unlimited ability for timbral quality. This makes the Virtual Music Instrument an interface suitable for the production of any type of sound, and allows a performer using the Virtual Music Instrument to fit within any number of ensembles

As Virtual Music is able to be distributed as a package for inclusion in *Unity* (see subsection 3.1.4 for installation instructions), the opportunities it offers—like those offered by VR—are only limited by the imagination of the user. Virtual Music provides a new medium in which to notate and perform music that simply did not exist before. It may be implemented in any number of scenarios and in any number of applications. I have proved the application of Virtual Music in two applications in particular: first, gaming applications, as seen with the application developed for *Peril (But Not Quite Doom)* (see section 4.1); second, immersive experiences, as seen with *Yurndu akanatha* (see section 4.2). Virtual Music has potential for application outside of these two provided examples, including in music therapy (as discussed in subsection 5.2.1), music education (see subsection 5.2.2), and the widespread distribution of music performance (see subsection 5.2.5), to name a few. Nothing remotely similar to Virtual Music exists in these fields, so it is not possible to evaluate Virtual Music against comparable systems. It would also not be possible to test Virtual Music in these fields at this stage without further ethics clearances and methodologies, which would entail an entirely new project. I can, however, offer some comparisons in the gaming field. I can compare existing music rhythm video games directly with the application of Virtual Music developed for *Peril (But Not Quite Doom)*, which could be considered as a prototype for future work in the field of gaming.

In gaming applications, Virtual Music is most comparable to *Guitar Hero* (Harmonix). And unsurprisingly, when I spoke of Virtual Music, the first response of most people was 'oh, so like *Guitar Hero*?'. I can see why this comparison is made: *Guitar Hero* is a good example of an accessible form of alternative notation. A clear distinction between Virtual Music and *Guitar Hero*, however, is that the notation of *Guitar Hero* is descriptive and requires a specific controller in order to perform. This is because the notation that accompanies *Guitar Hero* refers to specific buttons on a required controller. Additionally, the notation of *Guitar Hero* does not clearly represent all Fundamental Elements of (Virtual) Music Notation (see subsection 1.4.1). It is

descriptive and not prescriptive, as its notation involves five different 'pitch degrees', representing the relevant buttons on the controller, which approximate frets on a guitar. These pitch degrees can be seen in Figure 5.1. Its representation of time is precise: the speed at which the 'notes' progress and the position and length of them is equivalent to their real length. However, the notation does not represent dynamics in any way, and the pitch of tones is controlled by the application and not always representative of the notation's pitch degree.



Figure 5.1: The five 'pitch degrees' of the *Guitar Hero* notation

All of those characteristics are in direct contrast to Virtual Music, which when used in combination with the Virtual Music Instrument, is prescriptive. Its notation clearly defines the pitch degree, the time, and the dynamic value. Without the Virtual Music Instrument, its notation is descriptive, however it still precisely represents all Fundamental Elements of (Virtual) Music Notation in some way.

Pitch The five pitch degrees of the *Guitar Hero* notation are merely representations of which button presses are required to produce a prerecorded tone. Therefore, they are laid out horizontally so that they correspond to the buttons on the controller. This means that the pitch degrees of *Guitar Hero* are reused for different pitches, and the same pitch may be represented by a different pitch degree, depending on context. For example, if a note's pitch was the lowest point in a phrase, it would likely be placed on one of the lowest pitch degrees. If the same pitch was then the highest pitch in a subsequent phrase, it is likely that the pitch would be placed on one of the highest pitch degrees. In contrast, the pitch degrees of VIRTUAL MUSIC are precise and unlimited. They indicate pitch by their height and are directly correlated to the pitch of the tone that will be produced. Pitch degrees are not reused, and they do not change according to context. If the frequency of a pitch degree is altered by changing its octave, its shape is altered to reflect this. Because of this, the representation of pitch in VIRTUAL MUSIC is precise, whereas in *Guitar Hero*, it is descriptive. In VIRTUAL MUSIC, the tone produced is also produced live in response to the notation, where as the notation of *Guitar Hero* is a representation of prerecorded audio, pressing the correct button as indicated by the pitch degree merely plays the respective audio filee.

Time The representation of time has two components: where the note is placed rhythmically in comparison to other notes, and the rhythmic duration of the note. Time is represented in the notation of *Guitar Hero* by the placement of the pitch degree on a scrolling platform resembling the neck of a guitar. With few exceptions, the notes usually only show their rhythmic placement in comparison to other notes and do not indicate duration. This is as where the notation of *Guitar Hero* is placed indicates when the player should move the button on the controller. This button actuates vertically to represent a strum of pluck of a guitar string. VIRTUAL MUSIC, however, represents both rhythm and duration clearly. The rhythmic placement of a note in comparison to other notes is represented by its place on the scrolling stave. The duration of a note is

shown by its length.

Dynamics The notation of *Guitar Hero* does not represent the dynamic value of a note in any way. While the volume of the prerecorded audio changes, the volume of this audio is in no way reliant on or represented by the notation. This is in direct contrast to Virtual Music, which precisely indicates the dynamic value of a note by the note's width. The wider the note, the louder it is. Virtual Music is also capable of showing gradual dynamic level changes such as crescendo and diminuendo in the form of hairpins. As the notation of *Guitar Hero* does not represent dynamics in any way, no equivalent convention exists.

Having listed the capacities of Virtual Music in comparison to *Guitar Hero*, some limitations should be noted. Virtual Music Shorthand Notation is somewhat limited by not having a GUI in which to compose. The pieces presented in this thesis do prove that such a GUI is not a prerequisite in order to compose using Virtual Music Shorthand Notation. However, the time taken to carefully construct these compositions was considerably higher than that of other notation styles. Additionally, Virtual Music Shorthand Notation offers no audible feedback or feedback for correct syntax while composing. These issues impact Virtual Music's primary aim of accessibility, as composing using Virtual Music Shorthand Notation is difficult. This will be addressed in future work, such as via the development of a GUI to compose. This is discussed further in subsection 5.2.5. It would also be possible to create some sort of mode in which a performer could improvise either with the Virtual Music Instrument or with an external instrument, and Virtual Music would record and save the performance's respective Virtual Music Shorthand Notation. This Virtual Music Shorthand Notation could then be adjusted further as necessary and implemented in other projects, if desired. Audible feedback could also be addressed by a special mode in which all notes are activated during a performance with no user

input. Such a mode would allow the composer to hear the composition in its most basic form. These issues were not directly addressed and the additional modes not produced in the lifetime of this project simply because for this project I hoped to develop the core functionality of Virtual Music to a point where the Virtual Music Instrument and Virtual Music Notation were viable as a means of performance and composition. Virtual Music has proven itself as a viable system of notation and performance, and I believe that these issues can be addressed in work following this thesis.

Can Virtual Music Notation appropriately convey a composer's intention, while still allowing the performer musical expressiveness?

While composing pieces for Virtual Music, I never found myself limited. In fact, I found myself liberated. My intentions were clear, and I had found new ways to express them with timbral characteristics that I had never experimented with. In particular, I was able to notate ideas that combined my interests. This is seen especially with *Peril (But Not Quite Doom)*, which combines my interest in gaming with my passion for music. I was also able to rework old ideas, as seen with the translation of *Yurndu akanatha Oratunganga* into Virtual Music or even with the reworking of *Étude 1* into *Étude 3*, which allowed performers to experience these piece in a brand new and exciting way. I also enjoyed regularly improvising while developing the application and still find myself starting a project without a piece loaded just so that I can improvise and explore the creative and expressive options that the Virtual Music Instrument offers.

This proves that Virtual Music Notation and the Virtual Music Instrument offer great ability for musical expression. Importantly, it proves this while addressing

Wessel and Wright's requirement of a 'low entry fee with no ceiling on virtuosity' (11). This is done via the progression through the various Performance Modes. Similar to performance using existing instruments with existing notation styles, a composer and a performer retain as much control as they wish. It is possible, in fact, that a performer is able to express themselves more easily than a composer: they are able to explore the sound and options that the composer and developer has provided for them at any time, and can freely make use of the applications and synthesisers developed by other composers and developers.

5.2 Future Work

The four preceding questions informed the decision-making and features that were implemented during the development of Virtual Music for this thesis. But the nature of this project is ongoing. Virtual Music will only continue to improve over time, aided by the future development of VR technologies, and by these questions. This thesis allowed Virtual Music to develop from an idea to a fully functional system of composition and performance. This thesis provides one answer to all of these questions, but has asked many more. Can Virtual Music be used to aid music therapy? Can Virtual Music be used in music education or as a pathway to music performance with a different instrument? How can Virtual Music be adapted to enable ensemble performance? Can Virtual Music be used in conjunction with machine learning or other methods to generate music compositions procedurally? How can pieces composed for Virtual Music be preserved and distributed widely?

5.2.1 Therapy and Accessibility

The field of music therapy is quickly evolving and diversifying. It has only become recognised as a profession in modern times, though the connection between music and healing likely existed in early societies (West 51). What music therapy entails can be difficult to summarise, and there are therefore many different definitions of it, some more broad than others. Music therapists Rachel Darnley-Smith and Helen M. Patey describe two applications of music therapy: first, the use of music for its inherent restorative or healing qualities; second, the use of music as a means of interaction and self-expression within a therapeutic relationship (17).

After speaking with colleagues of mine who are music therapists, I have found that music therapy has a significant range of uses and purposes. These can range from addressing social disorders by finding, discussing, and performing music in group or individual sessions (Darnley-Smith and Patey 19), to using music performance as a method to communicate with patients with dementia (102). Physical illnesses can also be targeted through the use of recorded or performed music which, in many cases, results in less need for pain-relieving drugs (18).

Based on the literature, I see four applications for Virtual Music in the space of music therapy: first, targeting social disorders, fears, and anxieties; second, targeting physical illnesses and rehabilitation; third, targeting pain relief or distraction; finally, as a means of self-expression. The benefits of such applications are supported by recent studies surrounding the use of VR in therapy. For example, Gerardi et al. concluded that VR Exposure Therapy assisted in reducing symptoms of post traumatic stress disorders sufferers by 56 per cent (212). In a study of the rehabilitation of three patients following stroke, after using VR-based therapy, 'the patients' strength, range of motion, fractionation, and speed improved' (Jack et al. 913). Another medical study involving

military personnel concluded that soldiers using VR when being treated for combat-related burn injuries reported large reductions in pain (Hoffman et al. 256). Finally, a 2020 study that placed users in a virtual environment in which they could paint found that their self-reported levels of anxiety and stress were reduced. Additionally, the users expressed 'a sense of play and experimentation due to the novelty of virtual media' (Kaimal et al. 7).

The physical performative nature of Virtual Music adds unique functionality and combines the benefits of music therapy with the benefits of VR in other forms of therapy. Programs supporting these uses can already be developed fairly easily with no modifications to Virtual Music.

For example, an application could be developed that targets physical rehabilitation. A piece could be designed to make the user stretch by reaching from lower pitches to higher pitches. For this sort of application, the user would first set their performance area to a limited area that they are comfortable with. In later performances, they could increase this area, making their movements greater in each subsequent performance. Such a piece could also be used for patients with trembling issues by encouraging them to keep steady during the performance, ensuring that the notes stay active. In this case, the touch buffer could be adjusted to allow more or less lenience (see subsubsection 3.7.2). Alternatively, the Basic Performance Mode could be used so that notes remain active once they are touched for the first time (see subsubsection 3.7.2). Such functionality is already supported by Virtual Music, and the development of an application to support this would be simple. Similarly, the composition of the work would also be easy.

Virtual Music could also be used to target anxiety or fears. For example, a fear of public performance or crowds could be targeted by an application that slowly increases the number of members in the audience or crowd, and their level of interaction and engagement with the performer. Since the performer would be focused on performing the

work in controlled surroundings, they should feel comfortable and would then be able to progress at a rate that suits them. This progression could be controlled by a therapist who adjusts the number of audience members manually at certain times, or by values that are predefined by the application. An application that has predefined values has the added benefit of being usable without a therapist. The environment itself can also be adjusted. For example, a virtual stage or smaller performance space could be created and chosen depending on the personality of the performer. Similarly, an environment that the user is familiar with could make the performer more at ease, for example, recreating a well-known, safe space or even their home or bedroom.



Figure 5.2: The Xbox Adaptive Controller

Virtual Music's aim of accessibility gives it inherent capacity to be used by performers with mobility issues. This can be achieved through a variety of methods, but most notably through custom or accessible controllers such as the *Xbox Adaptive Controller*, seen in Figure 5.2. Such adaptive controllers are designed to be extensible and accessible; that is, any number of mechanisms can be used to act as buttons, triggers, or joysticks. These include buttons placed on wheelchairs, or soft cushions that can be placed near and activated by the face, a feature which is especially suitable for

performers with motor issues impairing the movement of their limbs. The control scheme of the instrument can be overridden to match the output of such controllers by supplying values to the appropriate fields on the `Instrument` class (see subsubsection 3.7.1). The position of the instrument may also be overridden by making the Virtual Music Instrument follow the position of another object (see subsection 3.7.9). This other object could be controlled by another user such as the therapist or caretaker, or even by the adaptive controller. These adaptive controllers are often designed to be usable by more than one person, also allowing the therapist or caretaker to assist where necessary.

5.2.2 Beginning Music Education

There is considerable research into the effects of music education in the development of cognitive and communicative skills, especially in children and teenagers. It is unclear to what extent music education has in these areas and 'neuromyths' such as the 'Mozart effect'—which implied that students that listened to Mozart temporarily showed increased levels of spatial reasoning—conflate these studies (Odendaal et al. 5). However, music education undeniably has *some* effect. Wu and Lu summarise the effect of music education on the development of social behaviours, saying 'group music education could be considered as one of the main approaches for the development of empathy and prosocial behaviours' (4). Similarly, there is evidence to suggest that music education supports skills related to engineering and technical design:

> Significantly, one of the most striking facts of the life in Silicon Valley industry is that the very best engineers and technical designers are, nearly without exception, practising *musicians*. (Venerable 64)

Instruments designed for learners and children are unfortunately often difficult to use and are therefore not conducive of further learning. This is because such instruments are often 'either toys or [...] merely scaled down version of adult instruments' (Smith 14). This often results in learners who are not able to grow musically, an issue that impacts many DMI (Wessel and Wright 11).

It is possible that Virtual Music and the Virtual Music Instrument have educational uses, most notably in educational games. Such games could promote musical education with new and fun methods for music learning, tailored for the age level and purpose. For example, Virtual Music could be used in early education in conjunction with media designed for children. This could include an application in which the main character of a children's show plays the show's theme song with them.

Such an application could also be used as a pathway to musical performance on an acoustic instrument. For example, with some modifications, Virtual Music Notation could show corresponding note letters or even fingerings for an instrument. It could also be aligned with a piano in a manner similar to 'piano tutorials' created by applications such as *Synthesia* (Synethesia LLC).

5.2.3 Ensemble Performance

As Serafin et al. state, 'one of the fundamental aspects of music is its ability to create shared social experiences' (29). The Virtual Music Instrument may be used in conjunction with performers on other instruments through the methods as described in subsubsection 5.1. However, at this stage, Virtual Music does not directly support multiple concurrent performers in its code. It may be possible to support multiple concurrent performers with some modifications, such as the implementation of code

supporting performances over a network or the internet. There are a number of considerations that the developer must take into account before any modifications. The following considerations assume that all performers are using different machines.

- The piece must start at the same time for all performers.
- Computer performance must not impact the progression of the piece, and all performers must remain in sync.
- Any modifications to the tempo must affect all performers simultaneously (see subsubsection 3.7.1).
- It is preferred that all performers hear the audio produced by others and themselves.
- It is preferred that all performers can communicate at minimum, or see each other and their staves at maximum.

In its current state, it is possible to add support for ensemble performance with these considerations in mind. However, the implementation would be unique to the application and up to the developer to design. For example, ensemble performance could be supported through the use of network code and a centralised server or a leader/followers relationship. In the following modifications, the centralised server is referred to as the leader. All other machines are referred to as the followers. The host machine is responsible for coordinating the follower machines. The leader could either be a dedicated server, or a machine taking part in the performance that has been designated as the leader.

The piece could be started at the same time for all machines by the leader after checking that all followers are ready. Each follower would require some sort of indicator, likely on `VMManager`, that states whether the piece has finished loading and is ready for performance. Once the leader has detected that each follower is ready, it could start the

performance for all followers simultaneously. Ensuring that the progression of the stave is uniform between followers could be achieved by limiting the control of the stave to the leader. This can already be achieved with the current codebase by setting `VMManager.AutoPlay` to `false` on each machine. The host could then invoke the `Play` method on the `VMManager` instance of all followers directly. The host would also need to track the progression of each follower in order to ensure that they remain in sync, should there be any sort of disconnection. Alternatively, the followers could keep in sync with the leader by calling `VMManager.Play` as normal, but setting the position of the stave to what is reported by the leader. Real-time tempo modifications could be supported by combining the effect of all followers together, or by limiting modifications to only the leader. Finally, giving all performers audible and visual feedback on the current state of the piece as well as the status of the other performers needs network code implementation. This also requires some sort of video feed, or else some sort of method to represent the performers in the virtual environment.

Latency is a significant problem for each of these modifications and considerations. Serafin et al. list 'Reduce Latency' as the second design principle for VRMIs:

> All interactions should be smooth and have minimum latency. Because VRMIs are inherently multisensory, reduction of latency with respect to both sound and visuals is paramount. […] [I]t is crucial that VRMIs represent timely and synchronized audiovisual feedback in response to the user's actions. (Serafin et al. 27)

Though this principle refers to reducing the latency of a VRMI, the discussion is especially relevant for network and internet performance. Any delay caused by network, code, or Virtual Music itself causes desynchronisation, making ensemble performance impossible.

5.2.4 The (Procedural) Generation of Music Compositions

Aided by the rapid advancement of computer technologies—and more recently by significant advancements in machine learning—composers have been able use computers for the generation of music. The term 'generative music' was popularised by self-described 'non-musician' Brian Eno, and encompasses the concept of 'real-time algorithmic composition' (Collins and Brown 1). Creative technologies researcher Oliver Bown provides an overview of advanced generative technologies or what he calls 'machines that make art', and their impact on music in the twenty-first century (I.1). Virtual Music offers a platform for the development of generative music systems, enabling VR display and interaction.

Generative music systems are relatively simple to construct in Virtual Music for two reasons: first, Virtual Music Shorthand Notation is purely text that can be parsed, greatly increasing the speed and simplicity of any necessary calculations; second, Virtual Music Shorthand Notation has finite and simple rules and constraints, understood easily by both human and computer. Generative compositions or compositions with randomised elements may be created in Virtual Music in its current state. This can be done with the creation of a program or script that parses and randomises elements of the Virtual Music Shorthand Notation or constructs the stave using models aided by artificial intelligence.

It is also currently possible to create compositions that alter themselves either randomly or morph depending on the performance. For example, it is possible to use the events system (see section 3.11) with events such as `Note.Activated` in conjunction with such methods of generative composition or randomisation.

While some of these examples are currently possible using external programs or scripts, better support for such use cases is needed. Better support would increase the accessibility of Virtual Music and offer greater opportunities for implementation. It would be quite simple to alter the generation code to include options for randomisation. Such code has not yet been implemented, however, and the use of external programs or scripts is necessary. Additionally, changes to a stave or to the properties of a note are not reflected until the piece has been regenerated. Once the note has been generated, it is not possible for its properties to change. This includes properties such as its representation of pitch and dynamics. The aforementioned functionality could be implemented by making the note more responsible for its representation of the Fundamental Elements of (Virtual) Music Notation. For example, *Unity*'s `Update` method could be used to check if any value has changed and update the properties of the note accordingly (although this would come at the cost of valuable processing time). However, the implementation of such checks for a note's representation of time would be difficult. Each note is positioned sequentially, which means that each note has no reference point to other notes. If the rhythmic position or length of a note were to change, other notes would not be aware of this change, and the total length of the stave would remain the same. These issues are not impossible to solve, but require significant development time and testing.

5.2.5 Preservation and Widespread Distribution

A composed piece may currently be distributed in three ways. Each way has its own set of benefits and limitations. Their suitability also changes according to intention.

First, the Virtual Music Shorthand Notation may be directly inserted into any application that uses Virtual Music. The Virtual Music Shorthand Notation itself may be distributed by any means that can store and share text or files.

This is similar to how the pieces composed and included with this thesis are preserved: any person with Virtual Music could input the provided Virtual Music Shorthand Notation into their application. The piece could then be performed and sound exactly as I intended it. This method assumes that the user has installed and set up the development environment (see section 3.1). This is an interesting method because while musical content remains the same, the appearance of the Virtual Music Notation and the performance environment including the synthesisers would change, as they are controlled by the developer. The musical content may also be easily changed simply by altering the Virtual Music Shorthand Notation before inserting it into the application.

Second, instances of the `Piece` class may be serialized and saved to file (Unity Technologies, 'Script Serialization'). They may then be loaded into *Unity* through any means, once it has been configured for use with Virtual Music (see subsection 3.1.4). This method has the added capacity to store other information, including transparency levels and preservation information such as the title. Therefore, pieces distributed in this manner will be similar in appearance to what the composer and developer first created. Synthesisers and their related components may also be distributed in this manner, so the original composer may choose to also supply a synthesiser. However, all fields of the loaded `Piece` class the musical content and even the preservation information may easily be altered once loaded into *Unity*. This includes the note materials, synthesisers, and the performance environment.

Finally, pieces may be preserved and distributed by creating applications specific to a piece or collection of pieces. For example, all pieces composed as part of this thesis were packaged as separated applications. They could also have been packaged as one application in which the performer could choose the piece that they would like to perform. If another performer were to load that application on their system, they could perform any included works with great ease, since no configuration or the installation of any

application is required. However, this method does not allow the easy replication of the piece(s), as the application is considered 'self-contained'. As such, pieces distributed in this way cannot be altered and appearance of the notation and performance environment remains as the composer and developer created. This method assumes that the composer and developer are capable of producing an application that is suitable for widespread distribution.

Considering the possibilities of Virtual Music, I am already planning to create an internet-connected application that Virtual Music composers and performers throughout the world can use to perform. This application could be considered a 'score library', and would allow composers to submit pieces that could then be selected and performed by any other user of the application. Pieces submitted to the score library would instantly have a large pool of potential performers. Additionally, such an application could have an 'improvise' mode in which a performer could simply improvise with the Virtual Music Instrument and experiment with new synthesisers and timbral qualities. The score library could also include an 'audience' mode so that other users could watch a performance or improvisation session. A 'record' mode would also allow performers to record their performances for later distribution or viewing.

For the purposes of distributing and preserving compositions, such an application currently relies on composers being capable of producing compositions with Virtual Music Shorthand Notation (see section 3.2). Currently, pieces would need to be composed externally and submitted to the score library. As discussed in subsubsection 5.1, a new GUI could be developed or additional modes created that allow a composer to create new musical works. Currently, though, such a mode does not exist. It is not, however, impossible to create. When creating such an application or GUI, the events system (see section 3.11) and note flags (see subsubsection 3.2.3) must also be considered, as they may be limited. This is because applications that use these features are often specifically developed. It may be possible to include a standardised set of functions that may be

activated or frameworks that composers may use in order to develop their own methods in a standardised way.

5.3 Reflecting on the SDLC Method as a Methodology

My decision to use the Iterative and Incremental Software Development Life Cycle (SDLC) method as a methodology provided me with great flexibility to experiment with my scholarly and creative practice. While, in the past, other methodological approaches had hindered my ability to play with (and bend) conventions to craft theoretically anchored and yet innovative content, this SDLC method provided me with a generative framework to support the experimental nature of my project. It enabled me to roam (relatively) freely within both academic and creative realms, and more importantly, to elegantly bring the two together so that each could support and nurture the other. The Iterative and Incremental SDLC method indeed enabled me to craft a productive flow between theory and practice, as it generated enriching exchanges between the scholarship and my trial-and-error software development process: my trials and errors fuelled further research into the literature to help me overcome the issues that I was facing, and the scholarship provided me with ongoing material to reflect upon and improve the aims, requirements and functionality of Virtual Music. Additionally, the fact that this methodology called for the inclusion of a curated selection from the documentation into the body of the thesis strongly supported my aim of accessibility, which could then not only apply to Virtual Music, but also to the written work which accompanies and narrates its development into being. Interestingly, this SDLC method also proved quite representative of my compositional approach, as illustrated in particular by the études. A 'prototype' of each piece was composed, and then refined

through systematic increments designed to improve specific aspects (such as melody development, orchestration, phrasing, and dynamics), or target certain features (such as polytonality, extensibility, or multiple timbres). As such, this methodology has contributed to crafting a strong sense of unity and complementarity between all products developed throughout my candidature (the thesis, the software, and the pieces). It helped me mitigate and overcome the risks of fragmentation inherent to interdisciplinary ventures such as mine, and hold musical composition, software development and theoretical inquiries together on the same plane. It is through the unifying use of SDLC as a methodology that cross-pollination could occur, and it is this cross-pollination which led to creation and innovation—to Virtual Music.

5.4 Conclusion

In this thesis, I have developed Virtual Music, a new and innovative form of music notation, composition, and performance that makes use of the accessible nature of VR technologies. Virtual Music uses these technologies to provide a method of musical performance to musicians and non-musicians of varied skills and abilities. The Virtual Music Instrument does this through its Performance Modes, which engages 'the continued development of performance skill and personal style' (Wessel and Wright 12) by providing a pathway to increased musical expressiveness. The addition of new expressive capabilities with later Performance Modes adds incentive for a performer to increase their skill at a rate that suits them through gamification. This can be likened to a video game that gives the player increased abilities as they progress. *World of Warcraft* (Blizzard Entertainment), for example, starts players with a limited set of skills, but slowly gives the player additional abilities as their skill increases.

Virtual Music contributes towards the greater research interests of the New

Interfaces for Musical Expression community. In particular, it explores the new opportunities that VR and the digitisation, and furthermore the gamification, of music notation brings to the act of music composition and performance. The thesis has shown the capabilities of Virtual Music and some of its potential for further implementations and research projects. It has also shown its current limitations and provided potential methods to work around these limitations.

In a broader context, this thesis provides a resource for future creators of alternative forms of music notation. It also provides a resource for the implementation and creation of music software, particularly for applications targeting the *Unity* game engine. This is achieved by detailed documentation and analyses which show the workings of a composer trying their best and trialling different methods, iterating and incrementing upon the design of the software to find the solution that best achieves their goals.

Importantly, this thesis has provided an outlet to pursue a long-standing passion and goal of mine: to share the joy of music reperformance with my family and friends. This thesis is merely one step towards the broader goal of making music accessible to non-musically trained composers, performers, and developers. It represents only the beginning of Virtual Music.


VM

6 | References

Anderson, Virginia. 'The Beginning of Happiness: Approaching Scores in Graphic and Text Notation'. *Sound and Score: Essays on Sound, Score and Notation*, edited by Paulo de Assis, Leuven UP, 2013.

Ap Cenydd, Llyr. 'Ocean Rift'. 2016. www.oculus.com/experiences/rift/1253785157981619/. Accessed 13 June 2021.

Beat Games. Beat Saber. *Beat Games*, 2019. Video game.

Beck, Kent, et al. 'Manifesto for Agile Software Development'. 2001. agilemanifesto.org/.

Blender Foundation. Blender. *Blonder Foundation*, 2021. Software.

Blizzard Entertainment. World of Warcraft. *Blizzard Entertainment*, 2004. Video game.

Bown, Oliver. *Beyond the Creative Species: Making Machines That Make Art and Music*. MIT P, 2021.

Brace Yourself Games. Crypt of the NecroDancer. *Brace Yourself Games*, 2015. Video game.

Braxton, Anthony. *Tri-Axium Writings*. Vol. 2, Frog Peak Music, 1985.

Brewer, Jeffrey L. and Kevin C. Dittman. *Methods of IT Project Management*. Purdue UP, 2013.

Brown, Andrew R. *Making Music with Java: an introduction to computer music, Java programming, and the jMusic library*. Lulu, 2005.

Cage, John. 4'33". *Edition Peters*, 1953.

———. Notations. *Something Else Press*, 1969.

Cardew, Cornelius. 'Notation: Interpretation, etc.' *Tempo*, vol. 58, 1961, pp. 21–33.

———. Treatise. *The Gallery Upstairs Press*, 1967.

Carey, Benedict. 'SpectraScore VR: A networkable virtual reality real-time spectral music composition and performance package'. *Proceedings of the International Conference on New Interfaces for Musical Expression*, 2016, pp. 3–4.

Carranco, J. Camillo Sánchez. 'Vibrating Instruments In Virtual Reality: A cohesive approach to the design of Virtual Reality Musical Instruments'. 2018. Aalto University, MA thesis.

Cigainero, Jake. ‘European Museums Get Adventurous With Virtual Reality’. 12 Mar. 2018. www.nytimes.com/2018/03/12/arts/european-museums-get-adventurous-with-virtual-reality.html. Accessed 9 June 2021.

Clement, J. ‘Leading game engines used by video game developers in the United Kingdom (UK) 2019’. 2021. www.statista.com/statistics/321059/game-engines-used-by-video-game-developers-uk/.

Coleman, Ornette. ‘Prime Time For Harmolodics’. *Down Beat*, vol. July, 1983, pp. 54–55.

Collins, Nick and Andrew R. Brown. ‘Generative Music Editorial’. *Contemporary Music Review*, vol. 28, no. 1, 2009, pp. 1–4. doi:10.1080/07494460802663967.

Cooper, Paul. *Perspectives in music theory: An historical-analytical approach*. Dodd, Mead, 1973.

Cope, David. *Virtual Music: Computer Synthesiser of Musical Style*. MIT P, 2001.

Cruz-Neira, Carolina, et al. ‘Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE’. *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH ’93, Association for Computing Machinery, 1993, pp. 135–142.

Crytek. CRYENGINE. *Crytek*, 2020. Software.

Csikszentmihalyi, Mihaly. *Flow: the psychology of optimal experience*. 1st ed., Harper Collins, 1990.

Cycling ’74. Max. *Ableton*, 2021. Software.

Dannreuther, Edward. *Musical Ornamentation*. Vol. 2, E. Kalmus, 1893.

Darnley-Smith, Rachel and Helen M. Patey. *Music therapy*. Creative therapies in practice, SAGE Publications, 2003.

Duckworth, William. *Virtual Music: How the Web Got Wired for Sound*. Routledge, 2005.

Eggleston, Suzanne. ‘New Periodicals: ”Music Notation News: The Journal of the Music Notation Modernization Association”’. *Notes - Quarterly Journal of the Music Library Association*, vol. 51, no. 2, Dec. 1994, pp. 660–61.

Ekman, Robert Wilhelm. The Opening of the Diet 1863 by Alexander II. 1865.

Eno, Brian. ‘Genative Music’. 1996. inmotionmagazine.com/eno1.html. Accessed 15 June 2021.

Epic Games. Unreal Engine. *Epic Games*, 2021. Software.

Facebook. ‘Facebook to Acquire Oculus’. 2014. newsroom.fb.com/news/2014/03/facebook-to-acquire-oculus/.

Facebook Technologies. ‘Configure Unity Settings’. 2021. developer.oculus.com/documentation/unity/unity-conf-settings/.

———. ‘Get Started with Oculus in Unity’. 2021. developer.oculus.com/documentation/unity/unity-gs-overview/.

———. ‘OVRInput Class Reference’. 2021. developer.oculus.com/reference/unity/v27/class_o_v_r_input.

———. ‘OVRManager Class Reference’. 2021. developer.oculus.com/reference/unity/v27/class_o_v_r_manager.

———. ‘Rift Store: VR Games, Apps and More’. 2021. www.oculus.com/experiences/rift/. Accessed 9 June 2021.

Fitterer, Dylan. Audioshield. *Dylan Fitterer*, 2016. Video game.

Frampton, Daniel. ‘Garbage Collection and the Case for High-level Low-level Programming’. 2010. The Australian National University, Thesis.

Francisco, Eric. ‘After ’Guitar Hero’ and ’Rock Band,’ Games Lost Their Rhythm: And how we’ll get our groove back.’ 18 Aug. 2015. www.inverse.com/article/5283-after-guitar-hero-and-rock-band-games-lost-their-rhythm. Accessed 11 June 2021.

Frederic, Gerald J. ‘The African Aesthetic in World Creativity: Anthony Braxton’s Philosophy of Vibrational Affinity Dynamics’. *Journal of Black Studies*, vol. 38, no. 2, 2007, pp. 130–141.

Frontier Developments. Elite Dangerous. *Frontier Developments*, 2021. Video game.

GamesRadar+. ‘Here’s every winner from this year’s Golden Joystick Awards, including the Ultimate Game of the Year’. 2019. www.gamesradar.com/Golden-Joystick-awards-2019-winners/.

Gerardi, Maryrose, et al. 'Virtual reality exposure therapy using a virtual Iraq: Case report'. *Journal of traumatic stress*, vol. 21, 2 2008, pp. 209–213. doi:10.1002/jts.20331.

gerubach. 'Penderecki - Threnody (Animated Score)'. 2011. www.youtube.com/watch?v=HilGthRhwP8. Accessed 13 June 2021.

Gimbel, Allen. 'WOLFF: Exercises: The Record Connoisseur's Magazine'. *American Record Guide*, vol. 70, no. 1, Jan. 2007, p. 190.

GitHub. 'Mastering Issues'. 2020. guides.github.com/features/issues/. Accessed 6 June 2021.

Goehr, Lydia. *The Imaginary Museum of Musical Works*. Oxford UP, 1994.

Google. 'Tilt Brush'. 2021. www.tiltbrush.com/. Accessed 13 June 2021.

Grand View Research. 'Virtual Reality Market Size, Share & Trends Analysis Report By Technology (Semi & Fully Immersive, Non-immersive), By Device (HMD, GTD), By Component (Hardware, Software), By Application, And Segment Forecasts, 2021 - 2028'. San Francisco, CA, 2021. www.grandviewresearch.com/industry-analysis/virtual-reality-vr-market.

Grau, Oliver. *Virtual art: from illusion to immersion*. Leonardo, MIT P, 2003.

Grubb, Jeff. 'Audioshield is the best virtual reality game I've played yet'. 2016. venturebeat.com/2016/03/29/audioshield-is-the-best-virtual-reality-game-ive-played-yet/.

Guinness World Records Ltd. 'Highest score on Guitar Hero III (single track)'. 2009. www.guinnessworldrecords.com/world-records/highest-score-for-a-single-track-on-guitar-hero-iii. Accessed 11 June 2021.

Gutierrez Miranda, M. 'The Importance of Graphic Users Interface, Analysis of Graphical User Interface Design in the Context of Human-Computer Interaction'. *EDULEARN11 Proceedings*, Third International Conference on Education and New Learning Technologies, IATED, July 2011, pp. 7137–7144.

Harmonix. Guitar Hero. *Red Octane*, 2005. Video game.

———. Karaoke Revolution. *Konami*, 2003. Video game.

———. Rock Band. *MTV Games*, 2007. Video game.

Heath, Mark. NAudio. *Mark Heath*, 2021. Software.

Heilig, Morton L. 'Experience Theater'. US3469837A, 1969.

Hoffman, Hunter G., et al. 'Virtual reality hand therapy: A new tool for nonopioid analgesia for acute procedural pain, hand rehabilitation, and VR embodiment therapy for phantom limb pain'. *Journal of hand therapy*, vol. 33, no. 2, 2020, pp. 254–262.

Holmes, Thom. *Electronic and experimental music*. 2nd ed., Routledge, 2002.

Holst, Gustav. The Planets. *Goodwin & Tabb, Ltd.*, 1914–1916.

Hope, Cat. 'Electronic Scores for Music: The Possibilities of Animated Notation'. *Computer Music Journal*, vol. 41, no. 3, Sept. 2017, pp. 21–35. doi:10.1162/comj_a_00427.

———. 'Wording New Paths: Text-Based Notation in New Solo Percussion Works by Natasha Anderson, Erik Griswold, and Vanessa Tomlinson'. *Contemporary Music Review*, vol. 36, no. 1-2, 2017, pp. 36–47. doi:10.1080/07494467.2017.1370793.

Humphrey, Aaron Scott. 'Comics and pedagogy: authorship, authority and literacy'. 2016. University of Adelaide, PhD dissertation.

Iain, Simons. *Inside game design*. Laurence King Publishing, 2007.

id Software. Doom. *id Software*, 1993. Video game.

International Organization for Standardization. *Acoustics – Standard tuning frequency (Standard musical pitch)*. ISO 16:1975, International Organization for Standardization, 1975.

Jack, Alma S. Meriansand David, et al. 'Virtual Reality - Augmented Rehabilitation for Patients Following Stroke'. *Physical Therapy*, vol. 82, no. 9, Sept. 2002, pp. 898–915. doi:10.1093/ptj/82.9.898.

Jamhack Games. 'Percussive VR'. 2018. www.oculus.com/experiences/rift/1594093733950080/. Accessed 9 June 2021.

Johnston, Andrew. 'Interfaces for Musical Expression Based on Simulated Physical Models'. 2009. University of Technology, PhD dissertation.

Jordà, Sergi, et al. 'The reacTable: Exploring the Synergy between Live Music Performance and Tabletop Tangible Interfaces'. *International Conference on Tangible and Embedded Interaction*, 2007.

Kaimal, Girija, et al. 'Outcomes of Visual Self-Expression in Virtual Reality on Psychosocial Well-Being With the Inclusion of a Fragrance Stimulus: A Pilot Mixed-Methods Study'. *Frontiers in Psychology*, vol. 11, 2020, pp. 1–13. doi:10.3389/fpsyg.2020.589461.

Kirke, Alexis and Eduardo R. Miranda. 'An Overview of Computer Systems for Expressive Music Performance'. *Guide to Computing for Expressive Music Performance*, edited by Alexis Kirke and Eduardo R. Miranda, Springer, 2013, pp. 1–47.

Kissner, Michael. 'Writing a Game Engine from Scratch: Part 1: Messaging'. 2015. www.gamasutra.com/blogs/MichaelKissner/20151027/257369/Writing_a_Game_Engine_from_Scratch___Part_1_Messaging.php.

Kruchten, Philippe. *The Rational Unified Process: An Introduction*. Addison-Wesley object technology series, Addison-Wesley, 2004.

Latham, Alison, ed. 'Graphic Notation'. *The Oxford Companion to Music*, Oxford UP, 2011.

Lättman, Katrin, et al. 'A new approach to accessibility – Examining perceived accessibility in contrast to objectively measured accessibility in daily travel'. *Research in Transportation Economics*, vol. 69, 2018, pp. 501–511. doi:10.1016/j.retrec.2018.06.002.

Lely, John and James Saunders. *Word Events: perspectives on verbal notation*. Continuum, 2012.

LilyPond. '"Compiling" Music'. 2021. lilypond.org/text-input.html.

Luckey, Palmer. 'Oculus Rift: Step Into the Game'. 2012. www.kickstarter.com/projects/1523379957/oculus-rift-step-into-the-game.

Magnusson, Thor. *Sonic writing: technologies of material, symbolic and signal inscriptions*. 1st ed., Bloomsbury Academic, 2019.

Manaris, Bill and Andrew R. Brown. *Making Music with Computers: Creative Programming in Python*. CRC P, 2014.

Martín, Carmen Bachiller, et al. 'Study of the Interference Affecting the Performance of the Theremin'. *International Journal of Antennas and Propagation*, vol. 2012, no. 348151, Feb. 2012, pp. 1–9. doi:10.1155/2012/348151.

McGreevy, Nora. 'Venture Down a VR Rabbit Hole With This Free 'Alice in Wonderland' Tour: The Victoria and Albert Museum will host a free, Lewis Carroll-inspired virtual reality experience on October 22'. 2020. www.smithsonianmag.com/smart-news/alice-wonderland-vr-exhibition-180975952/.

McPherson, Andrew P. and Victor Zappi. 'Exposing the Scaffolding of Digital Instruments with Hardware-Software Feedback Loops'. *Proceedings of the International Conference on New Interfaces for Musical Expression*, 2015, pp. 162–167.

Microsoft. 'Abstract'. 2015. docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/abstract.

———. 'Access Modifiers (C# Programming Guide)'. 2020. docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers.

———. 'Action<T> Delegate'. 2020. docs.microsoft.com/en-us/dotnet/api/system.action-1.

———. 'Arrays (C# Programming Guide)'. 2021. docs.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/.

———. 'Configure an Ini File Item'. 2013. docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2008-R2-and-2008/cc731332(v=ws.11).

———. 'Dictionary<TKey,TValue> Class'. 2021. docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2.

———. 'Document your C# code with XML comments'. 2020. docs.microsoft.com/en-us/dotnet/csharp/codedoc.

———. 'Events (C# Programming Guide)'. 2015. docs.microsoft.com/en-us/dotnet/csharp/programming-guide/events/.

———. ‘Floating-point numeric types’. 2021. docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types.

———. ‘HashSet<T> Class’. 2020. docs.microsoft.com/en-us/dotnet/api/system.collections.generic.hashset-1.

———. ‘Inheritance in C# and .NET’. 2018. docs.microsoft.com/en-us/dotnet/csharp/tutorials/inheritance.

———. ‘interface’. 2020. docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/interface.

———. Microsoft Visual Studio. *Microsoft*, 2021. Software.

———. ‘Nullable value type’. 2019. docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/nullable-value-types.

———. ‘Regular Expression Language - Quick Reference’. 2021. docs.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference.

———. ‘Xbox Adaptive Controller’. 2021. www.xbox.com/en-AU/accessories/controllers/xbox-adaptive-controller.

Mozart, Wolfgang Amadeus. Clarinet Concerto in A major. *Breitkopf und Härtel*, 1791.

Mulraney, Frances. ‘The history and origins of traditional Irish music’. 2020. www.irishcentral.com/roots/history/history-origins-traditional-irish-music. Accessed 13 June 2021.

Munroe, Rundall. ‘Exploits of a Mom’. 2007. xkcd.com/327/. Accessed 4 June 2021.

———. ‘Wikipedian Protester’. 2007. xkcd.com/285/. Accessed 4 June 2021.

Mysteo. ‘Siren Song’. 2017. www.oculus.com/experiences/rift/1413791848642701/. Accessed 9 June 2021.

Naef, Martin and Daniel Collicott. ‘A VR Interface for Collaborative 3D Audio Performance’. *Proceedings of the International Conference on New Interfaces for Musical Expression*, 2006, pp. 57–60.

Nintendo EAD Tokyo. Donkey Kong Jungle Beat. *Nintendo*, 2004. Video game.

———. Super Mario Galaxy. *Nintendo*, 2007. Video game.

Niscioli, Paola. ‘ASO’s groundbreaking Virtual Reality Orchestra’. 2019. www.aso.com.au/2019/01/01/adelaide-symphony-orchestras-groundbreaking-virtual-reality-orchestra/. Accessed 9 June 2021.

Oculus Story Studio. ‘Henry’. 2016. www.oculus.com/experiences/rift/1043217179077622/. Accessed 8 June 2021.

———. ‘Lost’. 2016. www.oculus.com/experiences/rift/1016967501697367/. Accessed 9 June 2021.

Odendaal, Albi, et al. ‘Lost in translation? Neuroscientific research, advocacy, and the claimed transfer benefits of musical practice’. *Music education research*, vol. 21, no. 1, 2019, pp. 4–19.

Oliveros, Pauline. Teach Yourself to Fly. *Smith Publications*, 1974.

Penderecki, Krzysztof. Threnody to the Victims of Hiroshima. *Polskie Wydawnictwo Muzyczne*, 1961.

Peterle, Astrid. ‘Fluxus scores and instructions: the transformative years, ”Make a salad”’. *Women & performance*, vol. 19, no. 3, 2009, pp. 439–441.

Petrick, Elizabeth R. *Making Computers Accessible: Disability Rights and Digital Technology*. Johns Hopkins UP, 2015.

Pixel Games. ‘Realistic Tree 9 [Rainbow Tree]’. 2016. assetstore.unity.com/packages/3d/vegetation/trees/realistic-tree-9-rainbow-tree-54622. Accessed 10 June 2021.

Popoff, Alexandre. ‘John Cage’s Number Pieces: The Meta-Structure of Time-Brackets and the Notion of Time’. *Perspectives of New Music*, vol. 48, no. 1, 2010, pp. 65–82.

Popper, Frank. *From Technological to Virtual Art*. MIT P, 2007.

Preston-Werner, Tom. ‘Semantic Versioning’. 2013. semver.org/.

Puranam, Amarja. ‘Which Is Better to Learn First - C++ or C#?’ 2017. www.greycampus.com/blog/programming/which-is-better-to-learn-first-c-plus-plus-or-c-hash.

Quantz, Johann Joachim. *On playing the flute*. Schirmer Books, 1975.

Rădulescu, Răzvan C. ‘Choosing the right game engine’. 2020. www.gdquest.com/tutorial/getting-started/learn-to/choosing-a-game-engine/.

Rambarran, Shara. *Virtual Music: Sound, Music, and Image in the Digital Era*. Bloomsbury Publishing PLC, 2021.

Rare. Banjo-Tooie. *Nintendo*, 2000. Video game.

Rokeby, David. ‘Very Nervous System’. 2010. Accessed 4 Dec. 2021.

Royce, Winston W. ‘Managing the development of large software systems’. *Proceedings of IEEE WESCON*, 1970, pp. 328–388.

Rumpe, Bernhard. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International Publishing AG, 2017, doi:10.1007/978-3-319-58862-9.

Schmidt-Jones, Catherine. *Understanding Basic Music Theory*. OpenStax CNX, 2013.

Schoenberg, Arnold. *Fundamentals of musical composition*. Faber, 1970.

Serafin, Stefania, et al. ‘Virtual Reality Musical Instruments: State of the Art, Design Principles, and Future Directions’. *Computer Music Journal*, vol. 40, 3 2016, pp. 22–40. doi:10.1162/COMJ_a_00372.

Seymour, Neal E., et al. ‘Virtual Reality Training Improves Operating Room Performance: Results of a Randomized, Double-Blinded Study’. *Annals of surgery*, vol. 236, 4 2002, pp. 458–464. doi:10.1097/00000658-200210000-00008.

Skeet, Jon. ‘Implementing the Singleton Pattern in C#’. 2011. csharpindepth.com/Articles/Singleton. Accessed 12 June 2021.

SkyWatcherVR. ‘Instrument Studio VR’. 2019. www.oculus.com/experiences/rift/1746672708729216/. Accessed 9 June 2021.

Smith, Derek. ‘A Guide To Musical Instrument Design’. 2005. Auburn University, MA thesis.

Sorenson, Andrew and Andrew R. Brown. ‘Introducing jMusic’. *Proceedings of The Australasian Computer Music Conference*, 2000, pp. 68–76.

Steinberg. Cubase. *Steinberg*, 2021. Software.

Stephens, Rod. *Beginning software engineering*. Wrox, 2015.

Stravinsky, Igor. The Rite of Spring. *Muzgiz*, 1913.

Sutherland, Ivan E. 'The Ultimate Display'. *Proceedings of the IFIP Congress*, 1965, pp. 506–508.

Synethesia LLC. Synethesia. *Synethesia LLC*, 2021. Software.

Taylor, Chris. 'Guitar Hero Gone: What Went Wrong?' 10 Feb. 2011. mashable.com/2011/02/09/guitar-hero-dead/. Accessed 11 June 2021.

The Audacity Team. Audacity. *The Audacity Team*, 2021. Software.

The Music Notation Project. 'The Music Notation Project - Alternative music notation systems'. 2021. musicnotation.org/.

Thomsen, Lui Albæk. 'Physics-Based Real-Time Sound Synthesis for Virtual Reality Musical Instruments (VRMI)'. 2017. Aalborg University, MA thesis.

TwitchTracker. 'Beat Saber'. 2021. twitchtracker.com/games/503116.

Unity. 'Live Training April 13th, 2015: Creating a Simple Messaging System'. 2015. www.youtube.com/watch?v=0AqG1fDhPT8.

Unity Technologies. 'Anatomy of a Mesh'. 2020. docs.unity3d.com/Manual/AnatomyofaMesh.html.

———. 'Application.persistentDataPath'. 2021. docs.unity3d.com/Documentation/ScriptReference/Application-persistentDataPath.html.

———. 'Audio Effects'. 2021. docs.unity3d.com/Manual/class-AudioEffectMixer.html.

———. 'Audio Mixer'. 2021. docs.unity3d.com/Manual/AudioMixer.html.

———. 'AudioSource'. 2020. docs.unity3d.com/ScriptReference/AudioSource.html. Accessed 12 June 2021.

———. 'Collider.OnTriggerEnter'. 2020. docs.unity3d.com/ScriptReference/Collider.OnTriggerEnter.html.

———. 'Collider.OnTriggerExit'. 2020. docs.unity3d.com/ScriptReference/Collider.OnTriggerExit.html.

———. 'Example - Creating a quad'. 2020. docs.unity3d.com/Manual/Example-CreatingaBillboardPlane.html.

———. ‘GameObject.AddComponent’. 2020. docs.unity3d.com/ScriptReference/GameObject.AddComponent.html.

———. ‘Mathf.Lerp’. 2020. docs.unity3d.com/ScriptReference/Mathf.Lerp.html.

———. ‘MonoBehaviour.OnAudioFilterRead’. 2020. docs.unity3d.com/ScriptReference/MonoBehaviour.OnAudioFilterRead.html.

———. ‘Particle systems’. 2021. docs.unity3d.com/Manual/ParticleSystems.html.

———. ‘Prefabs’. 2020. docs.unity3d.com/Manual/Prefabs.html. Accessed 12 June 2021.

———. ‘Procedural Mesh Generation’. 2021. docs.unity3d.com/Manual/GeneratingMeshGeometryProcedurally.html.

———. ‘Procedural skybox’. 2020. docs.unity3d.com/Manual/shader-skybox-procedural.html. Accessed 10 June 2021.

———. ‘Quaternion.Lerp’. 2020. docs.unity3d.com/ScriptReference/Quaternion.Lerp.html. Accessed 10 June 2021.

———. ‘RequireComponent’. 2020. docs.unity3d.com/ScriptReference/RequireComponent.html.

———. ‘Rigidbody’. 2020. docs.unity3d.com/ScriptReference/Rigidbody.html. Accessed 12 June 2021.

———. ‘Rigidbody.AddForce’. 2020. docs.unity3d.com/ScriptReference/Rigidbody.AddForce.html. Accessed 12 June 2021.

———. ‘Script Serialization’. 2021. docs.unity3d.com/Manual/script-Serialization.html.

———. ‘Skybox’. 2019. docs.unity3d.com/2019.2/Documentation/Manual/class-Skybox.html. Accessed 9 June 2021.

———. ‘Terrain’. 2020. docs.unity3d.com/Manual/script-Terrain.html. Accessed 10 June 2021.

———. ‘Terrain Tools Sample Asset Pack’. 2019. assetstore.unity.com/packages/2d/textures-materials/nature/terrain-tools-sample-asset-pack-145808. Accessed 10 June 2021.

———. ‘Transforms’. 2021. docs.unity3d.com/Manual/class-Transform.html.

———. ‘Types of light’. 2020. docs.unity3d.com/Manual/Lighting.html. Accessed 9 June 2021.

———. Unity. *Unity Technologies*, 2021. Software.

———. ‘Unity Asset Store’. 2021. assetstore.unity.com/. Accessed 10 June 2021.

———. ‘Wind Zones’. 2020. docs.unity3d.com/Manual/class-WindZone.html. Accessed 10 June 2021.

Vail, Mark. *The Synthesizer: A Comprehensive Guide to Understanding, Programming, Playing, and Recording the Ultimate Electronic Music Instrument*. OUP USA, 2014.

Välimäki, Vesa and Tapio Takala. ‘Virtual musical instruments — natural sound using physical models’. *Organised sound: an international journal of music technology*, vol. 1, no. 2, 1996, pp. 75–86.

Vear, Craig. *The Digital Score: usicianship, Creativity and Innovation*. Taylor & Francis, 2019.

Venerable, Grant. *The paradox of the silicon savior: charting the reformation of the high-tech super-state*. MVM Productions, 1987.

VIVE Arts. ‘Mona Lisa: Beyond the Glass’. 2019. arts.vive.com/us/articles/projects/art-photography/mona_lisa_beyond_the_glass/. Accessed 9 June 2021.

Wagner, Carolyn. ‘Digital Gamification in Private Music Education’. *Antistasis*, vol. 7, no. 1, 2017, pp. 115–122.

Walker Arts Center. ‘Alison Knowles: Make a Salad’. 2014. Accessed 5 Dec. 2021.

Wessel, David and Andrew Wright. ‘Problems and Prospects for Intimate Musical Control of Computers’. *Proceedings of the CHI’01 Workshop on New Interfaces for Musical Expression*, 2001, pp. 11–14.

West, Martin. ‘Music Therapy in Antiquity: The History of Music Therapy Since Antiquity’. *Music as Medicine*, edited by Peregrine Horden, Routledge, 2000.

Wolf, Robert Paweł. 'Declarative Computer Music Programming: using Prolog to generate rule-based musical counterpoints'. 2015. University of Adelaide, PhD dissertation.

Wu, Xiao and Xuejing Lu. 'Musical Training in the Development of Empathy and Prosocial Behaviors'. *Frontiers in Psychology*, vol. 12, 2021, p. 1662. doi:10.3389/fpsyg.2021.661769.

7 | Appendices

A The complete commit history of the Virtual Music repository

`413f8b7` Initial commit

`d92443b` Added Major Review

`fe00421` More changes to Major Review

`ceb62bc` Test both repos

`1411905` Remove test.txt

`4137341` Testing auto-pull on remote

`0b336b4` Remove test.txt

`f02d04e` More work on Major Review including addition of dot points

`b2dc07b` More additions to section 3.3

`52bf537` Almost finish first major review draft

`c852b90` Remove '. '

`635663c` Finalise first draft without some figures

`f117ea8` PhD Proposal ready

`73a99d8` Add VirtualMusic

`f43d4cd` Add VirtualMusic

`9cf9c55` Fix .meta

`da96551` Force push .gitignore

`c64dc48` Base at major review

`d6bc18f` Change baseFrequency to global

`aaeabaf` Change to just build structure

`a582f7f` Add missing NAudio dependency

`eb447e4` Staging unknown changes before merging with Unity

`a04d936` Staging unknown changes to bring as of current state

`d2617d1` Update issue templates

`4209f40` Add ”Ideas for how to fix this issue?”

`af546a0` Add ”Was this issue discovered during a Performance Test?”

`ffb1ec0` Update bug_report.md

`2b2731c` Add ”Did a Performance Test inspire this enhancement?”

`1a38442` Add some readme

`89ee193` Forgot to update with logo

`4315faf` Add some basic instructions to begin using Virtual Music (unfinished)

`783768a` Complete basic instructions

`d84a097` Comment and uncomment various Instrument lines to reflect compatibility

`3868126` Add basic functionality to show currently touched note

`d8d9409` Add framework for simple and advanced modes

`c82a10c` Improve touch detection

`9ad5ea9` Tweak note transparencies

`2a5db1b` Add basis for other wave types

`9908c3f` Limit stimuli activation to one instrument (and some code tidy)

`a3fdcef` Add ”FromDirection” setting to allow changing direction of performance

`2cc3e5b` Make FromDirection left as default

`0f20429` Set the position to compare the furthest note from to the OVRCameraRig

`5f91028` Set the furthest to default to the newest if something weird happens and its not set

`09eae68` Improve stimuli becoming stuck and improve instrument responsiveness

`a8f9ba2` Remove NAudio dependency

`f229f7d` Fix SetInactive to not need OVRController, remove getters

`f78cdac` Remove NAudio, first steps at new synthesis, fix some bugs with touching

`5569db9` Remove MusicLayer requirement in favour of Stave list

`a8d8eca` Clean code reuse when generating notes

`e9f22a9` Add TimeScale defaulted to 2

`3c64f6c` Note height defaults to the distance between each note (divided by 2)

`7dc0329` Handle VR Focus, stop if not focused

`c693274` Restore FreeMode functionality to a point I'm happy with

`6fde9dd` Complete refactor to support #5 and #10

`d88d25d` Add the ability to regenerate (and remove old and add new) staves

`ddbef3f` Add missing boolean for note transparency

`fff01f8` Move pause checking to VMManager.Play to allow forcing play without HMD

`39ee104` Add RequireComponent to VMManager to allow persistent settings

`9932f93` TIdy initialisation of VMManager

`e7c56c2` Simplify VMManager.Version

`ef45720` Add boolean which allows control of paused state

`7778938` Add an EventEmitter to VMManager to trigger third party methods

`72c8ff1` Create a "paused" and "unpaused" event

`ce4461c` Add "HasBeenActivated" to the Note class which allows tracking total activated notes

`d3c61d3` Change bool Note.HasBeenActivated to int TImesActivated to allow emitting (re)activations

`ea789f4` Add emitter for failure to activate a note

`de33960` Add Notation.RemoveAnyExistingStaves method

`4339a17` Change EventEmitter to accept EventParameters

`5fb1b22` Add Note.PositionType

`85e7e71` Add stave.finished emitter, pass note object to note acitvation emitters

`486ca18` Move ready checks as per #14

`6ddb86a` Fix note.failure emission on other directions as per #15

`c43cd5a` Split event emitter into different categories, add Stave to parameters

`df5f0a8` Fix spelling of "position"

`3e9814a` Add "MimicNoteHeight" bool to Instrument

`d220742` Handle note position in single note staves

`a3f3a68` Note objects require RigidBody and set appropriate settings on Start()

`d8f32f9` Add NoteStartingPosition to Notation

`8155838` Fix detection of note finishing being too far

`2f190f2` Staves will follow the performer's position, and add depth buffer

`9aadc9e` Bump version number in preparation of next release

`a484882` Add definition for the stave's starting position. Start documentation.

`39c05d8` Change to Notation.RemoveStaveFromEnvironment with a few overloads

`9bc0bee` Change struct Settings.FromDirection to enum Direction

`dec575b` Add Notation.GenerationDefaults. Will allow custom Notation.Generate()

`e409d64` Add bool Environment.CenterStaveToPerformerOnGenerate

`7471434` Remove frequencies and heights from Generate parameters

`0df96db` Add basis of defining settings in shorthand notation

`1728502` Add warning about setting the stave position on Awake() or Start()

`b550145` Add the ability to notate dotted notes

`7feb7d3` Fix note number being wrong when settings string present

`e3c08c0` Remove new lines and whitespace in stave string

`ec478e2` Notes will always emit "Activated" or "Deactivated"

`f107f2d` Bump version number for next release

`8e7e4ce` Adjust note transparencies

`1740d2f` Correctly require BoxCollider

`e18ea73` isKinematic is better suited over useGravity

`ed87965` Activate the note's particle system when active if there is one

`3cf1d61` Add Boolean to control a note's particle system

`94e7fc2` Add the option to clear all particles on stop

`c54139b` Fix indentation

`8ba2f97` Stimuli are now generate in-code with a `Mesh`

`113d92c` Add `ShapeTypeDefaults.EightVertices` as default

`0bafaf1` Add option to supply note object

`b9b1937` Can now add custom properties (or components) to all notes

`e903b8e` Fix `PositionInStave` being wrong for first notes in the stave

`af36a15` Add missing checks for `Note.Playable` in `Instrument`

`4f31c69` Remove superfluous `Note.FailedToActivate`

`668372c` Add `Stave` class, simplify and tidy some `Notation.Generate()` code

`c7c235b` Controlling the particle system is better suited as a global setting

`18d69c3` Add total duration (in game unit size) to Stave class

`e202b81` Complete mesh rendering, including support for cresc. and dim.

`5f7fa2a` Fix stimuli being oblong in shape

`4b8a9f5` Adjust dynamics to range from 0f-1f

`2bfbee3` Add `Note.Length` and change `Stave.Duration` to `Stave.Length`

`2879204` Add support for hairpins in the shorthand notation

`56abe25` Add further checks to see if `dynamics` is suitable

`d649c16` `Note` needs to be added to `Stave` earlier

`0018dd9` Add support for triplets (not other tuplets)

`56c7353` Fix stimuli staying less transparent after being activated

`de4bdf1` Add rotation when `FromDirection` is not `Direction.Straight`

`249f119` Remove redundant auto-implemented properties, simplify and improve

`7b984ec` Follow OVR's event convention

`4455d5f` Remove unused "music" array

`94babf4` Add option to destroy notes when past the performer

`1bf35bd` Change method name to more clearly describe its use

`76332af` Fix note events being emitted too early

`12c3825` Add missing null checks to event emitters

`41f7312` Rewrite `Synthesiser` as a base class to inherit, include some examples

`131b166` Take `SquaredTriangle` out of the stratosphere

`a3845f2` Add the ability to add extra components to the synth (such as filters)

`261e5a0` Add some extra synths, change to equal power amplitude

`8673c1c` Change decimal for 12-TET from done by ear to actual maths

`07337d5` Add the ability to change settings of the piece during a performance

`fd919cf` Add `Notation.NoteWidthModifier`

`2a9f2de` Add BPM change notices

`16f573f` Fix settings blocks when `Notation.FromDirection` != `Straight`

`07b783b` Add the option to add emission to notes

`546b9a0` Update issue templates to include myself as the assignee

`e947d77` `StaveFirstNoteBuffer` should be a number of beats

`b729dcb` The `StaveContainer` will follow the position of the `Right` instrument

`4ab4717` Create `PerformanceModeOption` and `IntervallicModeOption` enums

`b675187` Create `Performer` class and use it for global settings such as heights

`4fae089` Setting the local position of the `staveObject` seems to fix things

`dae7758` Clarify the warning about `CenterStaveToPerformerOnGenerate`

`43c72ae` Adjust emission to use the initial emission colour as the basis

`abd2cba` Destroying objects isn't necessary, but nice as an option

`6f12d03` Change references to GameObjects to appropriate classes, tidy some clode

`9296b37` Code tidy for `Instrument`, simplify setting `Synthesiser` variables

`973e107` Change `OVRController` to `Instrument.InstrumentType`

`2eee4a9` Further code tidy, add documentation for public attributes

`6b41415` Stimuli are now generated in-code with a `Mesh`

`4b7fbd0` Add `ShapeTypeDefaults.EightVertices` as default

`9d68bbf` Add option to supply note object

`93150f1` Can now add custom properties (or components) to all notes

`0d60b35` Fix `PositionInStave` being wrong for first notes in the stave

`212f171` Add missing checks for `Note.Playable` in `Instrument`

`829be16` Remove superfluous `Note.FailedToActivate`

`0fc5846` Add `Stave` class, simplify and tidy some `Notation.Generate()` code

`ea300b1` Controlling the particle system is better suited as a global setting

`ddb932b` Add total duration (in game unit size) to Stave class

`b4c5d1d` Complete mesh rendering, including support for cresc. and dim.

`e92c795` Fix stimuli being oblong in shape

`cc9908a` Adjust dynamics to range from 0f-1f

`10c1d89` Add `Note.Length` and change `Stave.Duration` to `Stave.Length`

`da3246c` Add support for hairpins in the shorthand notation

`9184ea3` Add further checks to see if `dynamics` is suitable

`1cd85b4` `Note` needs to be added to `Stave` earlier

`3e46b19` Add support for triplets (not other tuplets)

`cd5ff00` Fix stimuli staying less transparent after being activated

`5eb9740` Add rotation when `FromDirection` is not `Direction.Straight`

`fdbd871` Remove redundant auto-implemented properties, simplify and improve

`8325980` Follow OVR's event convention

`58afb88` Remove unused "music" array

`48431a9` Add option to destroy notes when past the performer

`6b36257` Change method name to more clearly describe its use

`18e10d7` Fix note events being emitted too early

`ff77b02` Add missing null checks to event emitters

`bd66158` Rewrite `Synthesiser` as a base class to inherit, include some examples

`dc7cea2` Take `SquaredTriangle` out of the stratosphere

`efd2c10` Add the ability to add extra components to the synth (such as filters)

`6e5d773` Add some extra synths, change to equal power amplitude

`d56ebc6` Change decimal for 12-TET from done by ear to actual maths

`324250b` Add the ability to change settings of the piece during a performance

`98d156c` Add `Notation.NoteWidthModifier`

`61fd357` Add BPM change notices

`9cb802f` Fix settings blocks when `Notation.FromDirection` != `Straight`

`67f0df1` Add the option to add emission to notes

`17bbfc4` Update issue templates to include myself as the assignee

`30c15c0` `StaveFirstNoteBuffer` should be a number of beats

`cba25e3` The `StaveContainer` will follow the position of the `Right` instrument

`6493e8f` Create `PerformanceModeOption` and `IntervallicModeOption` enums

`a6a08fc` Create `Performer` class and use it for global settings such as heights

`96a9b77` Setting the local position of the `staveObject` seems to fix things

`8e391e0` Clarify the warning about `CenterStaveToPerformerOnGenerate`

`661db46` Adjust emission to use the initial emission colour as the basis

`a6462ab` Destroying objects isn't necessary, but nice as an option

`bddc3b1` Change references to GameObjects to appropriate classes, tidy some clode

`cedb48c` Code tidy for `Instrument`, simplify setting `Synthesiser` variables

`1b4ad78` Change `OVRController` to `Instrument.InstrumentType`

`9d8a439` Further code tidy, add documentation for public attributes

`7355acb` Merge remote-tracking branch 'origin/master'

`725a46c` Add the dynamics the stave was generated with to the `Stave` class

`52b456d` Introduce `Synthesiser` namespace

`c74d2c5` Add intervals with same rhythm, optionally in different octave

`87709e8` Add different shapes to show notes will sound in a different octave

`358f087` Add title, composer, and date of composition to `Notation`

`f92ff63` Add the ability to show composition information

`9f32099` Allow raising the sounding octave

`1714c7a` Revert to old event system so events are centralised, begin separation of `Notation` and `Piece`

`77b090a` Add " BPM" to BPM markers

`3c6408e` Fix the title card location according to #29

`c308473` Add `Events` namespace to `Notation`

`8c3130c` Virtual Music will now work without `Notation.Piece`. Probably buggy

`a5e20c4` Add `GenerateOnStart` to ease piece generation

`fb938cd` Bump version number

`84ec6e6` Make `Environment` basis for current playing conditions

`fddbdbd` Add `ShortHand` to `Note`

`d8dbd59` Add vibration for `Simple` mode

`3f4c4a9` Move `NoteFinishedBuffer` and `BPMChangeLocation` to `Environment`

`8ddd373` Remove `Piece` requirement from `VMManager`

`19c405f` `Piece` will be added to the `Environment`

`ee2afda` Give variables clearer names for their purpose, improve code efficiency

`6961377` Create helpers for Piece Information and Settings Blocks

`96a737c` Create helpers for ShortHandNotes

`6c7031a` Fix local variable naming convention

`90f1be7` Simplify StaveStartingPosition

`1c1ac84` Add helper for creating a `Note` from a `ShortHandNote`.

`d7c4622` Fix `RemoveGeneratedPieces` removing the Piece Container

`a82d3f6` Once a generated `Piece` is removed, `Ready` will be set to false

`a6dcbe6` Add `InformationSlidePosition`, fall back to inbuilt generation

`e964b60` Move generation warnings to `Generation()` to not spam. Add warning for no `NoteMaterial`

`1e75122` Clarify note transparency field names

`3dcce3d` Separate note emission settings from transparency settings

`180d2b0` Clarify `PieceObject`

`2ee047f` Add documentation to `Piece`

`731999b` Clarify `FromDirection` documentation

`bdce2b3` Dynamics are freely controllable in `PerformanceModeOption.Intermediate`

`a62861e` Add documentation to `Piece` itself

`5950a96` Add more documentation to `Instrument`

`7550c3b` Move warnings and fallbacks to be together

`413efd9` Remove comments explaining the shorthand notation, add safety

`de8d9ba` Add ability to supply note but override amplitude in `Synthesiser`

`610b224` Fix `Intermediate` performance mode and touch detection

`89926ca` Vibration only occurs in `Simple` mode for now

`9a31acf` Make `Synthesiser.AudioSource` publicly accessible

`837c0f9` Don't advance the `Stave` if the `Piece` is not `Ready`

`19d29a6` `Stave.SettingBlock` should be `Stave.SettingsBlocks`

`7ebad85` Add `TotalLoadedObjects` and `EstimatedTotalObjects` to `Stave`

`253d4ae` Make `Piece.Generate` an `IEnumerator` to allow asynchronous loading

`3db1f96` Add load status for `Stave`s in `Piece.Generate`

`0931ac6` Change `WaitForSeconds` to `WaitForSecondsRealTime` for `Time.timeScale`

`455bf1b` Better handle stopping vibration

`724a759` Add live BPM adjustment to the `Instrument`

`2940158` `Environment` doesn't need to be a `MonoBehaviour`

`13ce507` Add the ability to supply tempo markings without a BPM, with spaces too!

`2b0e652` Add `Piece.Play()` => `VMManager.Play` => `Environment.Advance`

`4024cff` Remove calls to `GameObject.Find("CenterEyeAnchor")`

`c5924ef` Add basis of ID generation for `Piece`, allowing proper calling by ID

`38e6b29` Add `Generated` event to `Stave` and `Note` events

`1721ad7` Add events `Finished` and `Generated` for `Piece`s

`f98fb34` Add `Removed` event for `Piece`

`141ebbb` Add `AutoPlayOnSinglePiece` which allows performing without any code

`8cbe84e` Remove `Environment.Squash` (same as `Mathf.Clamp`)

`4d86e76` Optimise `Environment.GetFrequency...`. Add `GetFrequencyFromPrecise...`

`3b81752` Refactor into separate files per class

`de63daf` Move `Synthesiser` examples to separate files

`1fef297` Add `GetPositionOf[NonDominant]Hand`, change to switch case

`8ba3738` Add the ability to get the current frequency, dynamic, and amplitude

`4456c7d` Fix `Simple` touch detection, allow hairpins, etc., in simple mode

`67636ff` Convert to expression bodies

`f91aa4f` Set note colors at start rather than calculating every time

`63341df` Mesh generation properly reflects vertex positions (I kinda understand)

`1e4ff40` Add locals for key shape positions, improve octave shapes

`c1c6abf` Differentiate point heights

`34fd716` Restore sanity checks during `Piece.Generate()`

`e696f24` Add `Note.AddCustomProperty`

`eefe92a` Add static methods to `Note`, `Note.ActiveInstrument` is the instrument

`ea5fb68` Remove repeated `GetComponent` calls

`ba3d6d7` Remove unnecessary `Stave.`

`c3bb8e8` Clean `Instrument`, remove repeated `GetComponent`, `GetInstanceID()`

`10ba28f` Simplify `new Mesh()`

`9843e14` Move default note shapes to `Note`

`0540160` Add `Accent` note shape

`eb42914` Add a `Instrument.MinimumSize` to help when the note size is very small

`885de2b` Add the ability to notate accents (though they currently do not sound)

`8902419` Override GetHashCode instead of making new property

`dd0ad88` Making Generate an `IEnumerable` isn't worth it at this time

`7edb1d3` Adjust execution orders

`a534465` Different time, dynamic on second notes, octave changes on single notes

`c15a774` Fix appearance of `OctaveUp`

`fadb810` Use better method to check for null or empty string

`d892949` Regex patterns should be const, not readonly static

`43ed962` Can specify `InformationSlidePosition`, deal with < `FirstNotePosition`

`00d92c1` `InformationSlidePosition` override should be beats, not a precise value

`1768d97` Incorrectly only passing "time_length" to `GetLength`

`1d8e8fe` Fix looking for second char in tie (carried over from previous versions)

`248d0bd` Fix rests

`63bfd78` Forgot to pass the full time block to `GetLength` for the second note

`81c5977` Be more precise with what characters are accepted for time

`1d6c797` Fix for height of rests now applies to all situations

`6d5e57c` Change `NoteFinishedBuffer` to `0f` to accommodate future sustains

`39b9bc6` Add `Finalised` event for `Note`

`07a58c1` Change `Note.InRangeBuffer` to `0f` by default

`b2b969b` `Current*` methods now require an `Instrument` argument

`a0e5fdf` Adjust order of arguments in `Note.Activate`

`ce9c072` `null`ify `note.ActiveInstrument` and set `Active` false in `Finalise`

`2c5c74a` Add `Synthesiser.Types` and `Synthesiser.Type`

`3c31f11` Set default `Note.SynthesiserType` to `Primary`

`999c0d8` Deactivate the `Note` in `Note.Finalise()`

`97a0dc5` `null`ify `Environment.Piece`

`1a0d701` Add comparative methods for `Note`, `Stave`, and `Synthesiser`

`56ec83e` Reorder properties of `Note`, add `NoteNumberInStave` for comparisons

`ba2d4e2` Support sustaining `Note`s and tones in all modes, add documentation

`a801332` Add ability to specify secondary timbre in shorthand, more articulations

`0a480af` Remove support for articulations until next major version

`dbf1d10` Secondary notes now use different textures

`f2fd3ba` Implement Lazy Loading, better handle Singleton pattern in VMManager

`ad90386` Remove unused private methods in `Synthesiser`

`627f297` Stop associated synthesisers in `ListenForDeactivated`, fixes sustains

`b2df28f` Split overcrowded `Instrument.Update()` into understandable ”handlers”

`9db80b1` Performer no longer has to be at unity transform position

`7492f30` Can play notes even once they’re passed (usually by sustaining)

`1ed93d2` `Synthesiser.Coroutine` no longer needed

`0298f55` No longer need to divide by 2 since equal power was implemented

`5d81def` Add .vscode folder to ignore list

`372db37` Make regexs a bit more robust

`cb275b8` Add option to retain the `Virtual Music` gameObject between scenes

`fc0f58f` `IsPaused` should change through the setter

`01f1198` `InitialSettingsContainer` should obtain valid defaults

`3720f41` Add instruments automatically if they don’t exist

`5b98f25` Retain instance by default, supporting scene changes

`254eaa9` Introduce `GameObjectToFollow`, add rotation and position offsets

`f186de6` Rename `CustomNoteProperties` to `CustomNoteComponents` for clarity

`8a7f080` Add the ability for `Notes` to activate `Flagged*Actions`

`565da29` Missed a few non-specific time length regexs

`97ba5c7` Handle any character case in shorthand notation (including BPM text!)

`760f20f` More succinctly handle character cases in Settings Blocks

`ef0cf4b` Support multiple instruments of the same type

`fc58285` Simplify `Note.HasHairpin`

`a1fa8c1` Better handle pauses, notes when finalised

`33df92c` Add ShortHand notation for Flags, appropriately add them to `Note`s

`8cb3227` Simplify `Note`, remove unnecessary assignments

`1403346` Fix frequency reporting as 0f

`e27567a` Change state indicators to `Note.State`, allow overriding

`bdf2c6d` Prefer `Note.State` over `Note.Playable`

`626f80f` Hairpins got lost at some point

`7f5bf84` Current implementation of polytonality requires diving waveforms by 4

`1cd97f2` Add `Instrument.CleanCurrentTouchedNotes()`

`0ba1b8f` `Instrument` will maintain a vertical rotation by default

`edc5271` Finalising should be handled according to the dominant hand

`5cdf6c3` Add `Touch Buffer` concept, add helper functions to tidy

`2b446f7` Add events for `Environment` when frequencies or heights change

`a503768` Adjust `IteratePerformanceMode` for legibility

`8a445c0` `Instrument` should set `IsPrimary`

`83fe43b` Better handle `NoteBuffer`, ensure deactivation in all situations

`4dad294` Reorganise `HandleTouchBuffer` while loop

`a100736` Add documentation for `IsThe SustainedSynthesiser`

`a5c2dd0` Add a ”Pitch Guide”, clear touched notes on iteration, handle `Colldier`

`e4fb293` Replace multiple `Shader.Find` calls with single call

`ea9429c` Fix `GetFurthest`

`b546ae5` Invert `TonalOffset` to be inline with offset convention elsewhere

`361b6a0` Centralize `GetFurthest` with `GetCurrentTouchedNote`

`cc963cb` Remove unnecessary `_Clean`

`e796f1d` Remove unnecessary active check

`3c7396f` Change `CurrentTouchedNotes` to `HashSet`

`5f45aa0` Add equatable modifiers for `ShortHand`, `ToString` for `Note`

`775297b` Implement equality overrides for `SettingsBlock`

`5be135e` `SettingsBlock` should invoke based on position of dominant hand

`cd9a3c1` `NoteState` should be in `Notation` namespace

`54029be` Remove unnecessary state check

`f8685cd` Add `NoteArticulation` and `Note.Articulations`

`90c82c0` Add articulation support in shorthand

`343874b` The distance of the `Note` should be based on the closest point

`4c6d0ef` Add "TODO" in relation to furthest check

`4f48a48` Disable `useGravity`

`ae9ed50` Distinguish between `Note.Length` and `.Duration`, `transform` at start

`b61f090` Reduce accent width

`abf84e1` Add Accent shapes for octave differences

`f24a64d` Invert trigger convention: `Instrument` is trigger, collider not convex

`16b58d7` Furthest check replaced with check for which is later in the piece

`ebe61ad` Clarify documentation

`1031711` Greatly simplify overrides with nullables and nullable coalescing

`2838ef8` Add a collection that contains all flagged `Note`s

`162f27d` Move `Instrument.cs` to `Instrument/`, reflecting recent automation

`313d8f8` Add a mask to the Pitch Guides to lessen clutter

`96e011a` Rename Pitch Guide shaders to better reflect their use

`6701098` Add methods to ascertain the pitch degree of a height

`7266d03` Add `CurrentPitchDegree` to `Instrument`

`d37489e` `Note`s can now show when they are at the same height as an `Instrument`

`33877ee` Add various customisable pre and post `Note` actions

`5248aff` Remove superfluous `Note.TimesActivated`

`32fc7ed` Introduce constants for accents

`100e84e` `CurrentProgressThroughNote` was always returning `1f`

`958d029` Adjust accent point positions

`91bdd72` Accents effect amplitude and dynamic, more audible on lower dynamics

`09dfc40` Merge accent shapes into standards, allowing visual difference as prev.

`341d864` Rename `NoteArticulations.Accented` to `NoteArticulations.Accent`

`55b3cd6` `NoteArticulation` should be in the `VirtualMusic.Notation` namespace

`9ff8aef` Introduce inheritance with Stave Objects, adhering more to OOP

`822fd28` Correct rests appearing

`3c0446b` Trim whitespace characters

`c2f83ee` Introduce multiple set check

`e625153` Instrument should refer to `note.Number`

`59300f0` Add `Index` and `Number` to Settings Blocks

`f200eb8` Move `Index` and `Number` to `Stave Object`

`60a1439` `RetainInstance` should be static

`1238042` Change back to static events, add some triggering methods

`76c7b26` Remove `DestroyNotesWhenPastPerformer` as it has potential for bugs

`a697323` Bump version number in preparation of release

`aea5441` Add some documentation, especially for events

`ded3dde` Convert block bodies to expression bodies

`83a8b4c` Prevent `Play()` and `Update()` from being called more than once

`7e7904e` Rename `VMManager.VMDebug` to `.Debug`

`3c495e6` Convert to expression bodies

`25fcbb4` Add `Generated` and `Invoked` events for Settings Blocks

`85fe13f` Centralize Stave Objects, handle naming and collections automatically

`2ba46ec` Fix `StaveObject.Index`

`44b84b2` Simplify `StaveObject.Stave.Set`

`c071e3e` Add ability to add `StaveObject`s to `Stave`, `Stave.GetObjectsOfType`

`1553a12` Move `PieceDirection` to `Piece.Direction`

`909627f` Move `NoteArticulation` to `Note.Articulation`

`ad71191` Remove leftover empty ”Piece Direction.cs”

`388d408` Move position getters of hands to static methods on `Instrument`

`fe79acd` `TimeAdvancedLastCalled` was never being set

`71f8a81` Correct spelling of ”Frequencies”

`3853fa0` Change `ClearParticlesOnStop` to false by default #53

`81a2d80` Remove multiple pieces#55, start major refactor to assembly, PieceState

`5c8f056` Move Performer to it’s own static class with inherit error checking

`a4f2621` Greatly simplify getting instruments with `Instrument.Get`

`873708f` Change property access modifiers

`4d19539` Move state change to before `TriggerDestroy`, fixing garbage

`dbfe7ac` `OnDestroy` should be private

`70402ca` `TriggerDestroy()` should still set state

`e98b20b` Move stave finalisation to the stave, deal with OnDestroy

`b7e1a43` Deal with ActivatableState overrides and OnDestroy

`e13f157` Disabled objects are treated as though they were destroyed

`7aa8be8` Offer virtuals for extending Note while prohibiting overriding required

`bafd3bf` Index can be set when the Piece is set, add access modifiers

`ea9b6d5` Remove `Note`s and `Stave`s `OnDestroy`

`9f2f313` Should finalise regardless of state

`34a3d5e` `OnDestroy` should also finalise `Note`s

`837e38c` `FinaliseGeneration` should be `protected internal`

`9261041` Change to `protected internal`

`b14ef21` Remove `Stave.Index`

`e9e514a` Add `Stave` to `Piece` automatically, name as with `StaveObject`

`a870ada` Handle `ShowInformation` toggle

`6991925` Limit `SettingsBlock`s to the first `Stave`, but allow forcing

`40bc812` Revamp `SettingsBlock` with nullables, allow displaying text only

`af68ee8` Match string convention

`f5d037e` Properly allow forcing `SettingsBlock`s in multiple staves

`0907b5d` Better handle `VMManager.OnDestroy` with `Piece` and `VMPerformer`

`3f2b04c` Better handle disabling, application pausing, fix a bug on disabling

`a0c7059` Better show when a `Note` is finalised

`f958807` Pieces will continue advancing even when finalised

`3e95b8b` Since the `Instrument` no longer rotates, we can no longer use rotation

`ebb8a36` Begin rewrite of synthesiser, CurrentState, Envelope and ”current amp.”

`14d565d` Invert if to reduce nesting

`d2bb179` Introduce `Oscillator`, `LFO`, `Envelope.Apply()`, abstract, allow many

`d30036b` Introduce easy way to adjust the octave of an oscillator

`d7d8ad0` Add convenient properties allowing fine adjustment of the frequency

`2a3c1ce` Add an overall amplitude multiplier

`3b5a65e` Clamp (or rather ”clip”) a sample between -1 and 1

`d8b10ad` Adjust default envelope a little to reduce the ”wah”

`31f801a` Add ”Delay” to the Envelope making it D+ADSR, fix decay envelope jumping

`800e14f` Add a simple white noise generator

`21df028` LFOs can specify a route (still to be implemented in each basic)

`cd32431` Should not divide the frequency modulation by the amount of LFOs

`59fa297` Add methods to get modified amplitudes and frequencies

`e514822` Add phase tracking oscillators for each common waveform

`4342d86` Oscillator class should be abstract

`787d5f1` Modulo is more succinct

`78aa8a2` Make phase assignment a float just in case there is any weirdness

`e16a983` Add TwoPI constant

`60b5a70` Remove dynamic keyword for .NET 2.0, remove `TotalLoaded` and estimated

`24d2225` Bump version (forgot to last time)

`32484d3` Remove the ability to override State, handle changes in setter

`f0e10f5` Add checks for unplayable, add finalised check in `ActivatableStave...`

`b6c0ede` Add a method to override a `Note`’s state as `Unplayable`

`104983e` Overcomplicated override caused the Instrument to get stuck as active

`2d17c2a` Default values were ignored by the constructor, adjust defaults

`1574b29` Add some average quality default synthesisers

`d143b6a` Restore `TotalLoadedObjects`, `Estimated*` using Linq

`f0a0e4e` Make the divisor after `GetSample()` more clear

`d0b06ec` `TimePlayLastCalled` was replaced with `Environment.TimeAdvance...`

`810010e` Bump version number

`090c0f2` Turn `TotalLoadedObjects` and `GetObjectsOfType` into one liners

`3349062` `OnDisable` to expression body

`6b7a37f` Clarify `MaximumTones` documentation

`87d3b65` The `StavesContainer` has been removed in favor of the `PieceObject`

`260bc07` Remove references to the stave container

`c7a1d2b` Handle the deletion of piece objects with `Piece Container`

`6d68514` Revert back to `NotGenerated` once destroyed

`1fbdf50` Prohibit changing a `Piece`'s `FromDirection`

`68d575e` Move `OnAudio...` to `Instrument`, now only one `AudioSource`, less GObj

`39cc5e1` Add the ability to assign the Instrument an overall mixer ON START ONLY

`1519ac9` Clarify some namespaces (automatically done with Rider)

`0bef34e` Move `InstrumentMixerGroup` to `Instrument`, deal when no "Instrument"

`91072a2` Remove redundant `AudioSource`

`6b91cd6` MAJOR refactor to `Tone` structure

`89b547b` Allow overriding the default synthesisers on a piece by piece basis

`ac04249` Convert all aspects of the synthesiser to scriptable objects

`a8b89c4` Serialize various `VMManager` fields, allowing editing in the Inspector

`35fd4de` Introduce `Instrument.Output`, allowing effects on mixers per each synth

`f8911ad` Rename `Synthesiser.Synthesiser.Type` to `Synthesiser.Type`

`9acf98b` Simplify getting the appropriate `Synthesiser` with `Synthesiser.Get`

`17bb9c7` Change to string interpolation

`42b155b` Simplify `Get` and `GetDefault` to `Get` with a `getDefault` parameter

`1e8b2f9` Remove redundant `Synthesiser.Type`

`395030e` Add the ability to create test tones for synthesisers in the editor

`219ea67` Handle error when there is no LFOs

`546199b` Handle settings in the setter so that `PerformanceMode` can be changed

B Issue Log

#1: [E] ”Simple” and ”Advanced” play mode options

#2: [BUG] Touch registration not entirely accurate

#3: [E] Add a visual indicator to show which stimuli is being touched

#4: [E] Change the default axis of time from Z to X.

#5: [BUG] Shouldn’t have to rely on a Piece Script for the instrument to work

#6: [BUG] Dynamics are not very present in the synthesiser

#7: [BUG] Some stimuli will get stuck on active

#8: [E] Add the ability to define settings of the piece in the shorthand notation

#9: [E] Stop a performance if the headset is removed

#10: [E] Remove/replace the Director component with references to the GameObject

#11: [E] Add a messaging or event emitting system

#12: [E] Set a note as ”failed to activate” if the performer misses the note while performing

#13: [E] Add a ”stave finished” emitter

#14: [E] Move the checks for `Environment.Ready` and `Notation.Ready` to `VMManager.Play`

#15: [BUG] ”failed to activate” and ”stave finished” emitters only work on Direction.Straight

#16: [E] Allow the user to specify the stave’s starting position

#17: [E] Add triplets (and other tuplets?)

#18: [E] Add dotted note lengths

#19: [E] Add crescendos and diminuendos

#20: [BUG] First notes in a stave are detected as ”Middle”

#21: [BUG] Hairpins are not possible in ”Simple Mode”

#22: [BUG] Notes emit too early

#23: [E] Be able to supply settings blocks during the piece

#24: [E] Notation should use the performer's preferred hand as the starting point

#25: [E] Add basic polyphony

#26: [E] Allow the second note in a chord to sound in a different octave

#27: [E] Add a fields to include the title, composer, and date of composition

#28: [E] Add the ability to show information about the composition in a "title slide"

#29: [BUG] The title slide appears too close when `FromDirection` != `Direction.Straight`

#30: [E] Tempo adjustment such as accelerando and ritenuto while performing

#31: [E] Add "BPM" to BPM text

#32: [E] Remove `Notation` from the `VMManager`

#33: [E] Method names in `Environment` are unclear

#34: [BUG] Events can't be triggered outside of their class, which makes it difficult to trigger some events and centralise them

#35: [E] Simplify the concept of a "piece" by splitting it into a separate file which can be dragged onto a `GameObject`

#36: [E] Add `Piece`s to the `Environment` rather than `Stave`s

#37: [E] `Piece.Generate()` is confusing. Split the generation of `Note`s, etc., into their own helper functions

#38: [E] Add haptic feedback to the `Instrument` in the form of vibrations

#39: [E] Separate `Note` transparency from emission

#40: [E] Freely control dynamics with `PerformanceModeOption.Intermediate`

#41: [E] Make `Piece.Generate()` asynchronous

#42: [E] Introduce a way to notate another note which starts at the same time as another but has a different length

#43: [E] Add a way to hold/sustain a note currently being touched

#44: [E] Add the ability to define the rhythmic location of the information slide rather than dividing the `FirstNotePosition` by two

#45: [E] Allow octave adjustments on single notes

#46: [E] Include and allow the activation of a secondary timbre (synthesiser)

#47: [BUG] Rests are broken

#48: [E] Allow developers to specify which GameObject the Instrument should follow with an optional offset

#49: [E] Add ”Flagged Actions”

#50: [E] Extend flags to also add flagged Notes to a collection based on that flag

#51: [E] Add an option to hide just the BPM number value but still show the text

#52: [BUG] When `DestroyOnFinalised == false`, when a piece is finished, it just sticks around in the same place

#53: [BUG] `ClearParticlesOnStop` should default to `false`

#54: [E] Limit settings blocks to the first stave only

#55: [E] Remove the ability to have multiple pieces being loaded

#56: [BUG] Virtual Music gets buggy when a ”Container” is destroyed or disabled

#57: [BUG] `Note`s continue when overriding their state once they’ve started

#58: [E] Add synthesiser envelopes

#59: [E] Separate oscillators from the synthesiser

#60: [E] Add Low Frequency Oscillators (LFOs)

#61: [E] Track phase of the oscillator

#62: [E] Simplify way to override the default synthesiser

#63: [E] Implement time domain synthesiser effects such as delay

#64: [E] Implement frequency domain synthesiser effects and filters such as low pass filters

#65: [E] Make it easier to attach Unity audio effects to the synthesiser

C v0.1.0

C.1 Release Notes

v0.1.0 eb447e4 490 commits to master since this release

First version!

C.2 Performance Test (Étude 1)

What changes have been made since the last performance test, and what is the focus and expected result of these changes?

This is the first performance test, so there is no previous performance test to compare to. However, as the first performance test, this test will focus on the strengths and weaknesses of the current implementation and representation of the three fundamental elements of music notation: pitch, dynamic, and time.

The current implementation has notes moving on the Z axis towards the performer. Time is in two parts: tempo is correlative to the speed of which the notes move towards the performer, and rhythm is to the length of the note. Pitch is correlative to the height of the note, and dynamics correlative to width. I expect this implementation will work well for dynamics, but will have some issues with pitch, and some minor issues with time.

Did the changes made achieve the expected result? If not, what could be done differently to achieve the expected result? What worked well, and what didn't work well?

The effectiveness of the current implementation is mixed. Initial expectations were accurate: dynamics are represented flawlessly, but some issues with pitch, and some lesser issues with time are apparent.

Pitch of subsequent notes is occasionally occluded by notes prior to them. This is most evident with short notes that are higher pitched than the longer notes preceding them, when both notes are above the performer's eye level. When short notes are lower pitched than the longer notes preceding them, and both notes are below eye level, the latter note is similarly occluded.

A fix for this could be to change the time axis from Z to increasing on the X axis. This will make pieces progress from left to right rather than towards the performer. All fundamental elements of music notation will be represented in the same manner, however

the left to right progression may solve the issues as highlighted as all notes will be visible with no chance of occlusion. A new issue may appear, however: since width would need to be changed to depth, dynamics may have new issues and may also need a new approach as it may be difficult to see the depth of some notes around eye level, or if the pitches of a series of notes are jumpy.

Eventually, another consideration could be to fade in notes, or increase their transparency. Though this would make it possible to see notes affected by these issues, it also has the potential to cause confusion when looking through multiple transparent notes.

Were there any adverse or otherwise unexpected effects of the changes?

As this is the first performance test, there is no changes to report on.

Were any new issues discovered? How can these new issues be dealt with?

The ideology of pressing a button as well as touching a note to produce a tone is a little confusing, and a little cumbersome. This method of tone production makes it difficult to perform smoothly as it is easy to miss the note, or press the mapped button at the wrong time. A solution to this may be to include two mapping options: a 'simple', and an 'advanced' option, with simple being the default. The simple mapping style would allow the performer to produce a tone simply by touching the note, and producing a tone when not touching a note would be impossible. Advanced would allow the performer to use the note as a guide, like traditional notation, and produce tones outside of what the notes prescribe, allowing for freedom of interpretation.

Touch registration is occasionally not accurate: the tonal output of the touched note is not as expected due to a touch being registered on a note that has already passed. There is a need to include a method to favour the furthest from (but still ahead of) the performer. Should also consider a visual indicator to show which note is being activated.

Notes for moving forward

- Change the time axis from Z to increasing on X.
- Based on the outcome of the next performance test, it might be necessary to include both options depending on the piece.
  - For example, the composer may choose to have the Virtual Music Notation coming towards the performer to increase immersion, though the default will be to progress the piece from left to right.
- Consider other options for the representation of dynamics.
- Dynamics are not very apparent in the synthesiser.

D v0.1.1

D.1 Release Notes

v0.1.1 `2cc3e5b` 471 commits to master since this release

Issues carried from last version

N/A.

✚ Issues added since last version

- ~~#1 [E] VI: `Simple` and `Advanced` play mode options~~
- ~~#2 [BUG] VI: Touch registration not entirely accurate~~
- ~~#3 [E] VI: Add a visual indicator to show which note is being touched~~
- ~~#4 [E] VE: Change the default axis of time from Z to X~~
- #5 [BUG] VI: Shouldn't have to rely on a Piece Script for the instrument to work
- #6 [BUG] VI: Dynamics are not very present in the synthesiser

New features

- Added a framework for `Simple` and `Advanced` play mode options (#1) (`d8d9409`)
- Added an installation readme (`1a38442` , `89ee193` , `4315faf` , `783768a`)

⚠ Major changes

- All notes are transparent by default, and are more opaque when being touched or activated (#3, #4) (`3868126` , `9ad5ea9`)
- Added `FromDirection` to `Virtual Music Notation` , with `Left` , `Right` , and `Straight` as available options (#4) (`a3fdcef`)
- Changed the default axis for time from decreasing on Z to increasing on X, making notes approach the performer from the left rather than from straight by setting the default for `FromDirection` to `Left` (#4) (`2cc3e5b`)

Improvements and bug fixes

- Improved touch registration (#2)
  - Only allow a note to be activated by one Virtual Music Instrument at a time (`9908c3f`)
  - Prefer the furthest note from the performer (`c82a10c`)
- Added the basis for more wave types in for the synthesiser (`2a5db1b`)

D.2 Performance Test (Étude 1)

What changes have been made since the last performance test, and what is the focus and expected result of these changes?

The most significant change is that notes now approach the performer from the left, increasing on the X axis instead of decreasing on the Z. This change was made as in *Performance Test 1*, it was occasionally difficult to see the upcoming notes. Alternatively, notes have also been made transparent, and are more opaque when being touched (activated) to provide visual feedback to the performer.

Touch registration - noted as being occasionally inaccurate in the last performance test - has been improved by only allowing one Instrument to touch a note at a time. Additionally, the Instrument prefers the note furthest from the performer, and can only touch one note at any time.

'Simple' and 'Advanced' play mode options have been developed. This test (and all future tests) will be based off 'Simple' mode, as that will be the recommended mode for all new performers, so it is important to test that mode most thoroughly.

Did the changes made achieve the expected result? If not, what could be done differently to achieve the expected result? What worked well, and what didn't work well?

The change of axis did not achieve the expected result. The performer would naturally rotate their head to face the direction of the note. The performer would then move their hands in a 'dog paddle' fashion, cycling their hands up and down in order to get 'out of the way' of later notes. If the performer strictly maintained a forward-looking position, the pitch of each note appeared more clearly and was more understandable, but this usually was not the case. The addition of transparency, however, was very effective: upcoming notes were clear and this implementation feels better.

Dynamics were less clear. When notes approach the performer from the left, they must move in front of the performer with a 'buffer' of a preconfigured value. This buffer must change in order to ensure that all notes are within range of the performer's reach, but also

not too close to them so that the performer themselves is the obstruction. Currently, the side of the note closest to the performer is set to remain the same distance away from the performer, therefore notes with a higher dynamic value will be deeper than ones with a lower dynamic value. The depth of the note can be obstructed, and is also not as intuitive as the width of the note as a means for representing dynamics.

Overall, this implementation is awkward and does not achieve the expected results: performances using this implementation feel worse to perform and are less intuitive

Were there any adverse or otherwise unexpected effects of the changes?

As mentioned above, when notes approach from the left or right, the performer would simply rotate their head in order to see the approaching notes more clearly. This was unexpected but an important revelation.

Were any new issues discovered? How can these new issues be dealt with?

Notes will occasionally get stuck as 'active'. This is especially evident when notes approach from the left or right and not straight.

The height of the note is also currently a static value. This means that the pitch of a note is not always clear, as notes will be too thick when there is a higher number of tones per octave, and too small when there is a lesser number of tones per octave.

Notes for moving forward

- Change the time axis back to decreasing on Z.
- Investigate notes becoming stuck.
- Notes move slowly and are short. Though this shows the tempo and time value clearly, it feels unusual and unpolished. Investigate a 'time multiplier' of sorts which increases the length of the notes as well as the tempo at the same rate, so that they are equal.
- Performances will continue if the headset is removed. For solo performances, this is not ideal.
- There is currently no way to pause a performance.
- Set the height of the note equal to the space between each pitch.

E v0.1.2

E.1 Release Notes

v0.1.2 `6fde9dd` 458 commits to master since this release

Issues carried from last version

- ~~#5 [BUG] VI: Shouldn't have to rely on a Piece Script for the instrument to work~~
- ~~#6 [BUG] VI: Dynamics are not very present in the synthesiser~~

Issues added since last version

- ~~#7 [BUG] VI: Some notes will get stuck on active~~
- #8 [E] VN: Add the ability to define settings of the piece in the shorthand notation
- ~~#9 [E] VE: Stop a performance if the headset is removed~~
- ~~#10 [E] Remove/replace the Director component with references to the GameObject~~

New features

- Removed NAudio dependency in favour of in code synthesis (#6) (`a8f9ba2` , `f78cdac`)
- Added `Environment.TimeScale` which increases the size of notes in conjunction with the speed in which they travel to increase overall comfort and enjoyment (`a8d8eca`)
- Handle VR focus, and pause if the headset is not on (#9) (`7dc0329` , `6fde9dd`)
- Restore `FreeMode` ! (`c693274`)

Major changes

- Complete refactor (`6fde9dd`)
  - Changed `Director` component to `VMManager` which is now the point of reference for all shared properties, which can now be access more simply with `VMManager.Notation.FromDirection` rather than `Director.instance.notation.FromDirection` (#10)

  - Most classes are now true extensions of `MonoBehaviour` and can be serialized and interpreted by the `Inspector` accordingly
  - Similarly, many classes have appropriate `RequireComponent` attributes
  - Again similarly, the `Synthesiser` component is now an extension of `MonoBehaviour`, and can be accessed with `GetComponent<Synthesiser>()`
  - Removed some unnecessary calls to shared properties in favour of standalone methods that work with arguments, for example `Environment.Squash` now accepts arguments for the minimum and maximum values rather than getting them from shared properties.
    * `Environment.GetAllFrequencies()` is included in this change. This allows the ability to change various factors of the performance throughout the piece, including the number of tones in the octave, rather than being limited to properties set on `Awake()` or in `Notation.Generate()`, and also allows Virtual Music to work without a Piece Script (#5)
  - Clarified many method names and properties, for example `Environment.SquashHeight` to `Environment.GetFrequencyFromHeightInRange`
  - `VMManager` now has more helpful methods make using Virtual Music easier, including `VMManager.Ready` which checks if `Environment.Ready` and `Notation.Ready` are both true, and can be further extended in the future which more checks if necessary
  - Moved various properties around to classes that make more sense, such as `Notation.BPM` to `Environment.BPM`, as it is the `Environment` that controls the speed of the `Notation`
  - Improved logging of errors and logging if `VMManager.VMDebug` is set to true
  - Removed extra dependencies including `Singleton` and `MonoBehaviourExtended` in favour of a Unity specific implementation that I understand better therefore can utilise better (#10)
  - Removed note material resources as they weren't used
  - Removed `Note` prefab in favour of users supplying their own. May choose to include one as an option later once I have made one that I am happier with

- `Notation.FromDirection` reverted back to `Straight` by default as transparency was a better solution (`6fde9dd`)
- Height of notes now defaults to the distance between each note (as below)
- Added an option to override the note height with a set value (`3c64f6c`)
- Improve presence of dynamics in the synthesiser and instrument (#6) (`c693274`)

Improvements and bug fixes

- Furthest note is now obtained from the `OVRCameraRig`'s `CenterEyeAnchor` (#7) (`0f20429`)
- Improve notes becoming stuck (as below)
- Improve instrument responsiveness (#7) (`0f20429` , `09eae68` , `f229f7d` , `f78cdac`)
- Code clean up
  - Remove `MusicLayer` in favour of list of `Stave`s (`5569db9`)
  - Clean code reuse when in `Notation.Generate()` (`a8d8eca`)

E.2 Performance Test (Étude 1)

What changes have been made since the last performance test, and what is the focus and expected result of these changes?

The axis for time has been changed back to decreasing on the Z as in v0.1.0. This change is not to be tracked further as is change is the reverse of the proposed change as tested in the previous performance test.

The height of each note is now equal to the distance between each note. This change allows a clearer distinction between each pitch, and pitches cannot overlap. This change should improve visual clarity and the performer's understanding of the piece, as it allows them to look ahead more easily.

A `TimeScale` has been introduced, with a default of `2f`. The `TimeScale` multiplies the length of all notes, while multiplying the speed at which they approach the performer. Combined, this increases the length of the notes but as they move more quickly, the are only in playable range for the same amount of time. The `TimeScale` should improve the performer's comfort, the notation's overall visibility and therefore its ability to show its function.

Synthesis is now in-code rather than relying on `NAudio`. This should improve the synthesiser's ability to represent the notation; as the synthesiser is my own code, I understand it better, and can therefore make use of it better. The first improvement made utilising my greater understanding is that dynamics should now be more present in the instrument.

Other changes have been made such as the fact that Virtual Music can now handle VR focus. These are not the applicable for testing during performance tests as though their changes directly impact the performer's experience, the improvements they make to the notation specifically are unquantifiable and not the focus of these tests.

Did the changes made achieve the expected result? If not, what could be done differently to achieve the expected result? What worked well, and what didn't work well?

The standardisation of height between each note made it easier to differentiate each pitch, and therefore perform the piece in a manner closer to the intention of the composer. It also has the benefit of being adjustable according to the performer's preferences, or the piece's amount of octaves and the amount of tones per each octave. For pieces with a lesser amount of total pitches (thanks to a lesser amount of tones per octave and total octaves), the height of the note may be too large as the distance between each pitch may be very large. Outside of allowing the composer to specify the note's height, it may be necessary to limit the automatic height specification. Doing so, however, may make the differentiation between pitches more difficult.

The implementation of the `TimeScale` achieved its expected result satisfactorily. The notation was easier to understand as they were more apparent and easier to see. The default value of `2f` is good, but higher values may be better for faster pieces. It may be interesting to include a method to change the `TimeScale` according to the tempo.

In-code synthesis also indeed improved the instrument's ability to portray dynamics. It would be good to investigate the ability to apply filters to the synthesiser. Additionally, it would be good to investigate more options for 'release' and enveloping the synthesiser.

Were there any adverse or otherwise unexpected effects of the changes?

Occasionally, touch generation isn't accurate. I did not account for the instrument's trigger box when standardising the height of the note. It would be good to match the instrument's trigger box to the height of the note.

Were any new issues discovered? How can these new issues be dealt with?

Clicking and artifacting occurs in the synthesis, especially when stopping the tone generation or when changing pitch. I think that this is occurring due to the synthesiser not returning to a '0 point' in its wave and therefore jumping to a different value too quickly.

Notes for moving forward

- Investigate options for dynamic (amplitude) scale, such a logarithmic volume scale rather than a linear scale.
- Investigate scaling the `TimeScale` according to the tempo of the piece.
- Investigate different options for the size of the instrument's trigger box.
- Look at making the synthesiser properly return to a '0 point'. This could be left till the appropriate point in development if necessary.

- Fairly happy with how things are looking in terms of representing the fundamental elements of music notation. It would be good to now improve the functionality of Virtual Music related to the goals of v0.1.X rather than purely focusing on the basic representation of the fundamental elements.

F v0.1.3

F.1 Release Notes

v0.1.3 `ec478e2` 419 commits to master since this release

Issues carried from last version

- ~~#8 [E] VN: Add the ability to define settings of the piece in the shorthand notation~~

Issues added since last version

- ~~#11 [E] Add a messaging or event emitting system~~
- ~~#12 [E] N: Set a note as 'failed to activate' if the performer misses the note while performing~~
- ~~#13 [E] N: Add a 'stave finished emitter'~~
- ~~#14 [E] Move the checks for `Environment.Ready` and `Notation.Ready` to `VMManager.Play`~~
- ~~#15 [BUG] 'failed to activate' and 'stave finished' emitters only work on Direction.Straight~~
- ~~#16 [E] N: Allow the user to specify the stave's starting position~~
- #17 [E] Add triplets (and other tuplets?)
- ~~#18 [E] Add dotted note lengths~~

New features

- Added an event emitting system (#11) (`7778938` , `4339a17` , `c43cd5a`)

  An event emitting system has been added, which will allow better and easier third party implementations of Virtual Music into applications such as games. Emitters can be subscribed to, and will pass various objects depending on the emitter. For example, `Notes.Activated` or `Notes.Failed` will pass which note was activated, or similarly, failed. A basic example of Virtual Music being implemented into a game would be to subscribe to the `Notes.Failed` emitter, and if too many notes are failed, then the performance will stop. The following emitters have been added:

  - `General`

        * `Paused` (`72c8ff1`)
        * `Unpaused` (`72c8ff1`)
    - `Notes`
        * `Activated` (`d3c61d3`, `ec478e2`)
        * `Reactivated` (`d3c61d3`, `ec478e2`)
        * `Deactivated` (`d3c61d3`, `ec478e2`)
        * `Redeactivated` (`d3c61d3`, `ec478e2`)
        * `Failed` (#12, #15) (`ea789f4`, `6ddb86a`)
    - `Staves`
        * `Finished` (#13, #15) (`85e7e71`, `6ddb86a`, `5fb1b22`)

- Shorthand notation can now specify settings with a "Settings String" at the start of the piece! Current settings include `BPM`, `TonesPerOctave`, and `BaseFrequency`. For example `[B=120;T=12;F=220;] 8q6 10q ...` sets the `BPM` to 120, the `TonesPerOctave` to 12, and the `BaseFrequency` to 220 (#8) (`0df96db`, `7feb7d3`)
- Can now notate dotted note values! (#18) (`b550145`)
- `VMManager.ControlPause` has been added to allow third party custom pause implementation (`ef45720`)
- The starting position of the stave can now be specified, and defaults to the position of the performer (#16) (`e409d64`, `a484882`, `1728502`)
- Add `Notation.RemoveStaveFromEnvironment()` and multiple overrides, which allows removing specific staves or all staves from the `Environment` (`de33960`, `39c05d8`)

⚠ Major changes

- Notes now check their position and can set properties of themselves accordingly (including if they were activated or not) (#12) (`ea789f4`, `ce4461c`, `d3c61d3`, `8155838`)
- `Notation.Generate()` can now remove existing staves from the `Environment` if the `RemoveExistingStaves` argument is set to `true` (as default) (`d88d25d`)
- `Notation.Generate()` now requires `lengths[]`, `dynamics[]`, and `RemoveExistingStaves` arguments, and will get these from `Notation.GenerationDefaults` by default (`dec575b`, `7471434`)
- The stave now follows the position of the performer by default (`e409d64`, `2f190f2`)

Improvements and bug fixes

- Restored missing option to enable/disable note transparency (`ddbef3f`)
- Pause checking has been moved to `VMManager.Play` to allow forcing play without HMD (#14) (`fff01f8`)
- Ready checks have been simplified to `VMManager.Ready` (`486ca18`)
- A stave can be forced to advance with `Environment.Advance()` instead of using `VMManager.Play()` , which is the recommended usage as it checks if everything is ready (#14) (`486ca18`)
- Can now specify the starting position of the notes within the stave (`d8f32f9`)
- `Environment` and `Notation` are now "required components" of `VMManager` to allow persistent configuration (`39ee104`)
- Notes now have a property which states their position in the stave (`5fb1b22` , `d220742`)
- Notes now require a `RigidBody` and set the required properties on `Start()` , which simplifies setup (`a3f3a68`)
- The `Virtual Music Instrument` now mimics the height of the notes by default. This behaviour can be disabled (`3e9814a`)
- Code tidy and general bug fixes (`9932f93` , `e7c56c2` , `df5f0a8` , `9bc0bee` , `7feb7d3` , `e3c08c0`)

F.2 Performance Test (Étude 2)

What changes have been made since the last performance test, and what is the focus and expected result of these changes?

The ability to notate dotted note values has been added. This gives the composer more options for musical expression. As a rhythmic 'dot' simply extend the time value of the note which has the dot applied to it, the effect of this change is unquantifiable in terms of its legibility and clarity. As the clarity of the dotted values is directly related to the clarity of the representation of time in `Virtual Music Notation` , any observation which is indicative of the time value of a rhythmic dot being unclear should be considered as the time value `Virtual Music Notation` being unclear. If it is found that it is unclear if a dotted value is indeed dotted, a potential fix may be to more clearly segment the `Virtual Music Notation` into a rhythmic subdivision, whether by its texture, or by its shape. For example, adding lines to the texture for the note, placed where the rhythmic

subdivisions lay, allows the performer to more clearly understand where the rhythmic subdivisions lay. Similarly, adding small 'notches' into the sides of the note will clearly show where the rhythmic subdivisions lay, though this approach is both more difficult to implement, and may potentially be obscured both by the note itself depending on the viewing point of the performer, and by other notes surrounding it.

The `Virtual Music Instrument`'s trigger box now mimics the height of the note by default. This change should help some touch detection issues as highlighted in the previous performance test.

Additionally, when approaching the performer with the default 'straight' setting, the stave now follows the horizontal position of the performer. This change will ensure that the `Virtual Music Notation` is always in 'range' of the performer.

Finally, many additional features have been added to `Virtual Music`, most notably the event emission system. These changes, while directly impacting how `Virtual Music` feels and operates, are not necessarily 'testable' and not the focus of these performance tests. Changes have also been made which increases the ease and also opportunities for creative expression for compositions and composers using `Virtual Music`, such as the option to include 'Settings Strings' at the beginning of the piece. Similarly, these changes are not the focus of these performance tests.

Did the changes made achieve the expected result? If not, what could be done differently to achieve the expected result? What worked well, and what didn't work well?

As expected, it was not completely clear that notes with a rhythmic dot were 'dotted' (therefore extended by half the length of the original value) rather than just 'longer'. However, the audible effect of the rhythmic dot is especially clear; while performing and listening to a performance of a piece which has rhythmic dots, it is perceptible that these notes were dotted and not just arbitrarily longer. Considering this, I do not believe that it is currently necessary to investigate the 'solution' to this as mentioned previously, as the intended effect of the rhythmic dot, which is to extend the time value of the note by half, is in practice achieved. Instead, it may be good to implement at some later point in development.

The trigger box of the `Virtual Music Instrument` mimicking the height of the note achieved the intended result of improving touch detection. Similarly, following the horizontal position of the performer achieved the intended result of keeping the `Virtual Music Notation` always in range of the performer.

Were there any adverse or otherwise unexpected effects of the changes?

There were no adverse effects of any of the changes mentioned above. Previous changes to the transparency of the notes may need to be further investigated, however, as notes can still be more obscured than I would like.

Standardising note height to be according to values obtained in the current `Virtual Music Environment` made it clear that the width (dynamic) of the note is not according to the current performance environment as it is hard-coded. It would be good to multiply the width of the note according to the dynamic, and a 'note width modifier'.

Were any new issues discovered? How can these new issues be dealt with?

The change made to mimic the horizontal position of the the performer highlighted that it is possible for the performer to simply walk forward, changing the point where the rhythmic beat lies. To combat this, it is necessary to define a point which is considered the rhythmic 'zero' point. This point should then follow the performer.

However, it is important to note that when performing with `Virtual Music`, the performer will generally hold their hands out in front of them. This means that making the performer themselves the rhythmic zero point makes little sense, as the performer is continually 'anticipating' the beat. It would be best to use the position the performer's hand (likely their preferred hand) as the rhythmic zero point. This will still allow them to use their opposite hand to 'anticipate' the beat if they wish to do so.

It was also noted that the practice of supplying an object to be scaled according to the dynamic level and rhythmic duration of the note could cause issues, including incorrect world unit size, increased risk of obscuring the intention of the notation, and unwanted space between notes. The latter especially was noted during this performance test, in which the supplied object caused a visual separation between each note which was interpreted as intended separation similar to a mezzo staccato. Although difficult, one solution to this would be to generate the mesh for the notes in-code. This has the added benefit that custom note shapes may be added in the future, perhaps for articulations or similar. An important consideration is that this approach may limit extensibility by not offering a means to alter the appearance of the note. It should be imperative to at least allow the supply of a custom texture, if not also the the means to supply a custom mesh if a developer may choose to do so.

Notes for moving forward

- Further investigate note transparency
- Standardise note width and remove hard coded values
- Investigate moving note generation to a mesh system
- Add a rhythmic zero point and set it to the performer's preferred hand
- Now that rhythmic dots are added, add tuplets!
- Settings blocks work excellently, but it would be good to be able to change settings during the performance.
- Look into notating and representing crescendos and diminuendos.

G v0.1.4

G.1 Release Notes

v0.1.4 `725a46c` 309 commits to master since this release

Issues carried from last version

- ~~#17 [E] Add triplets (and other tuplets?)~~

✚ Issues added since last version

- ~~#19 [E] Add crescendos and diminuendos~~
- ~~#20 [BUG] First notes in a stave are detected as `Middle`~~
- #21 [BUG] Hairpins are not possible in `Simple` mode
- ~~#22 [BUG] Notes emit too early~~
- ~~#23 [E] Be able to supply settings blocks during the piece~~
- ~~#24 [E] Notation should use the performer's preferred hand as the starting point~~

New features

- Activating a note will activate the note's particle effects, if present (`ed87965` , `3cf1d61` , `94e7fc2` , `ea300b1`)
- Can now add custom note properties or components during `Notation.Generate()` (`93150f1`)
- Can now notate crescendos and diminuendos (#19) (`b4c5d1d` , `e92c795` , `da3246c`)
- Can now notate triplets (#17) (`3e46b19`)
- Can now destroy notes when past the performer. This behaviour is disabled by default (`48431a9` , `a6462ab`)
- Can now supply settings blocks during the piece (#23) (`324250b` , `9cb802f`)
- Tempo and style markings can now be shown during a performance as a visible notice (`61fd357`)

- Notes can now emit light, so that they can be seen in the dark (`67f0df1` , `661db46`)
- Added a `Stave` class, which contains all information about a stave, including the notes it contains and the dynamics it was initiated with (`0fc5846` , `ddb932b` , `10c1d89` , `1cd85b4` , `725a46c`)
- Created a `Performer` class, which can save the performer's settings (such as their height) and preferences (such as their preferred hand) to a file (`a6a08fc`)
- The `StaveContainer` (which contains all the stave `GameObject` s) will now use the position of the performer's preferred hand as the point where the beat lands (#24) (`cba25e3`)

⚠ Major changes

- Notes are now generated in-code with a `Mesh` (`6b41415` , `5eb9740`)
  - Notes will be generated with the default `ShapeTypeDefaults.EightVertices` object (`4b7fbd0`)
  - The option to supply a custom note object still exists (`9d68bbf`)
- Event emission now follows the convention of `OVR` (`8325980` , `ff77b02`)
- The width of a note is now a multiple of the dynamic and the `NoteWidthModifier` (`98d156c`)
- `Synthesiser` has been completely rewritten as a base class to inherit (`bd66158` , `dc7cea2` , `efd2c10` , `6e5d773`)
- `StaveFirstNoteBuffer` has been changed to `BeatsForCountIn` (`30c15c0`)
- `Instrument.PerformanceMode` and `Instrument.IntervallicMode` is now more clear (`cba25e3`)

🐞 Improvements and bug fixes

- Fix `PositionInStave` being incorrect for first notes in a stave (#20) (`0d60b35`)
- Note transparencies have been altered to make note more visible when looking through an active note (`8e7e4ce` , `cd5ff00`)
- Dynamics can only range from `0f` to `1f` (`cc9908a` , `9184ea3`)
- Fix notes emitting too early (#22) (`18e10d7`)
- Fix decimal values for pitch correction in a twelve-tone octave from done by ear to actual math (`d56ebc6`)
- Fix the notation being in random positions sometimes during builds (`96a9b77`)

- Code quality, readability, clarity, and usability improvements, including some documentation (`f107f2d` , `1740d2f` , `e18ea73` , `c54139b` , `212f171` , `829be16` , `10c1d89` , `fdbd871` , `58afb88` , `6b36257` , `17bbfc4` , `8e391e0` , `bddc3b1` , `cedb48c` , `1b4ad78` , `9d8a439`)

G.2 Performance Test (Étude 2)

What changes have been made since the last performance test, and what is the focus and expected result of these changes?

Triplets have been added. Similarly to the addition of rhythmic dots in the previous version, increases options for musical expression. As the addition of triplets impacts `Virtual Music Notation` 's representation of the essential element of time by increasing the options for rhythmic values and lengths, the addition of triplets will have the same impact as the addition of rhythmic dots, in that their clarity may need to be improved.

Crescendos and diminuendos have also been added, as well as the ability to define 'Settings Blocks' during a piece instead of just at the beginning. For the latter, tempo and style marking notices have been added so that the performer has a visual cue which informs them of the change and when the change will occur. For the former, the width of the notes will either increase or decrease in width according to the dynamic change. There is no expected negative impact to legibility or the ability of performance, but this change again allows better creative expression in composition, and brings the capabilities of `Virtual Music Notation` as a legitimately form of music notation more closely in-line with traditional notation.

The `StaveContainer` , which contains all staves (and therefore notes) now uses the position of the performer's preferred hand as the point where the beat lands. This change supports changes recommend in the previous performance test, which described the need for the `Virtual Music Notation` to use a standardised position in order to retain correct tempo.

Finally, changes have been made which effect the appearance and therefore legibility of the notes. These changes include the ability to activate a note's particle system, if present, and the addition of emitting light, which means that `Virtual Music Notation` can be seen in dimly lit environments. Most importantly, notes are now generated in code with a `Mesh` . This change allows better control over the shape and position of the notes, and corrects the issue described in the previous performance test where notes could be interpreted as though they were separated.

Did the changes made achieve the expected result? If not, what could be done differently to achieve the expected result? What worked well, and what didn't work well?

All changes worked well. Triplets were clear, noting the same clarity issue as described when rhythmic dots were added. Crescendos and diminuendos were clear, and the change to make the width of a note a multiple of the dynamic and `NoteWidthModifier` helps this clarity.

The stave following the position of the performer's preferred hand makes `Virtual Music` as a whole feel significantly more 'polished' and well thought out. While when the performer first raises their hand, it may be confusing at first as the `Virtual Music Notation` will seem to 'freeze' momentarily, it is not something that I expect will cause any issues in terms of legibility.

Tempo and style marking notices were a good addition, and are one that I would like to explore further with the addition of 'title cards' which show information such as the title, and composer's name.

Were there any adverse or otherwise unexpected effects of the changes?

In `Simple` mode, crescendos and diminuendos are not performable. This is because the `Virtual Music Instrument` will gather its dynamic and pitch from the note being touched, and there is no method to obtain the point of the note which is being touched in terms of the current position during the note's rhythmic duration. A method to obtain the rhythmic position of the `Virtual Music Instrument` in a note will need to be created, which I expect will not be a particularly easy task.

Were any new issues discovered? How can these new issues be dealt with?

Outside of new feature additions as described above, no new issues related to specifically to the representation of the three fundamental elements of music notation were discovered.

Notes for moving forward

I feel it is a good point to move to the next development stage: polyphony!

H v0.2.0

H.1 Release Notes

v0.2.0 `141ebbb` 249 commits to master since this release

Issues carried from last version

- #21 [BUG] Hairpins are not possible in `Simple` mode

✚ Issues added since last version

- ~~#25 [E] Add basic polyphony~~
- ~~#26 [E] Allow the second note in a chord to sound in a different octave~~
- ~~#27 [E] Add a fields to include the title, composer, and date of composition~~
- ~~#28 [E] Add the ability to show information about the composition in a 'title slide'~~
- ~~#29 [BUG] The title slide appears too close when `FromDirection` != `Direction.Straight`~~
- ~~#30 [E] Tempo adjustment such as accelerando and ritenuto while performing~~
- ~~#31 [E] Add 'BPM' to BPM text~~
- ~~#32 [E] Remove `Notation` from the `VMManager`~~
- ~~#33 [E] Method names in `Environment` are unclear~~
- ~~#34 [BUG] Events can't be triggered outside of their class, which makes it difficult to trigger some events and centralise them~~
- ~~#35 [E] Simplify the concept of a 'piece' by splitting it into a separate file which can be dragged onto a `GameObject`~~
- ~~#36 [E] Add `Piece`s to the `Environment` rather than `Stave`s~~
- ~~#37 [E] `Piece.Generate()` is confusing. Split the generation of `Note`s, etc., into their own helper functions~~
- ~~#38 [E] Add haptic feedback to the `Instrument` in the form of vibrations~~
- ~~#39 [E] Separate `Note` transparency from emission~~

- ~~#40 [E] Freely control dynamics with `PerformanceModeOption.Intermediate`~~
- ~~#41 [E] Make `Piece.Generate()` asynchronous~~

New features

- Virtual Music can now be used without any knowledge of code thanks to major simplifications related to ideology, and improvements to `Piece`, including `Piece.GenerateOnStart` which, when enabled, automatically generates the `Piece`, and `Piece.AutoPlayOnSinglePiece` which, when enabled, automatically plays the `Piece` if there is only one generated (#35, #36, #37) (`a5e20c4`, `1714c7a`, `8ddd373`, `19c405f`, `d7c4622`, `a82d3f6`, `6961377`, `96a737c`, `1c1ac84`, `837c0f9`, `141ebbb`)
  - `Piece.Play()` has also been added. which is a shortcut to tell the `Environment` to play that specific `Piece` (`2b0e652`)
- Polyphony! 🎵 (#25) (`c74d2c5`)
  - Polyphony works in a similar fashion to ”double stops” on violin.
  - Second notes are currently limited to the same rhythm as the original note it is accompanying.
  - Second notes can be in altered octaves and will appear differently when their octave has been altered (#26) (`87709e8`, `9f32099`)
- Title slides which can show the title, composer, and date of composition (#27, #28, #29) (`358f087`, `f92ff63`, `a6dcbe6`, `3c6408e`, `a6dcbe6`)
- The instrument and performer can now adjust the tempo while performing such as with accelerando and ritenuto (#30) (`724a759`, `13ce507`)
- `Piece` now generates a unique ID which can be used to find the specific `Piece` in the `Environment` (#36) (`c5924ef`)
- New events!
  - `Piece`
    - `Generated` (`1721ad7`)
    - `Finished` (`1721ad7`)
    - `Removed` (`f98fb34`)
  - `Stave`
    - `Generated` (`38e6b29`)
  - `Note`
    - `Generated` (`38e6b29`)
- `Piece.Generate()` is now asynchronous! (#41) (`253d4ae`)

  - The current load status can be obtained through the new `Piece.TotalLoadedObjects` and `Piece.EstimatedTotalObjects` fields (`7ebad85`)

- The `Instrument` now vibrates to provide haptic feedback (#38) (`d8dbd59`, `89926ca`, `455bf1b`)

⚠ Major changes

- BPM text (called "tempo markings") in `Settings Block`s can now be supplied without a BPM value itself (`13ce507`)
- Tempo markings can have spaces by using underscores, as underscores get automatically converted to spaces (`13ce507`)
- `Notation` is no longer necessary as part of the `VMManager` and is now a namespace that contains helper methods, structs, enums, and classes related to notation (#32) (`8c3130c`, `84ec6e6`, `3f4c4a9`, `ee2afda`)
- `Environment` is now purely a container for all current settings and attributes and includes methods related to the current performance environment (#36) (`19c405f`, `d7c4622`, `a82d3f6`)
- The events system has been reverted back to the old system so that events can be centralised (#34) (`1714c7a`, `c308473`)
- `Piece.cs` is now the central point for all `Piece`s and can be dragged onto a `GameObject` (#35) (`1714c7a`, `8ddd373`)
- Dynamics are now freely controllable in `Intermediate` performance mode (#40) (`bdce2b3`, `de8d9ba`, `610b224`)
- `Note` transparency values are now separated from emission values (#39) (`3dcce3d`)

🐞 Improvements and bug fixes

- Lots of new documentation inline documentation! (`2ee047f`, `731999b`, `a62861e`, `5950a96`)
- Removed calls to `GameObject.Find("CenterEyeAnchor")` to centralise calculation of the performer's general position, and to improve efficiency (`4024cff`)
- BPM indicators also show the words "BPM" so that their function is clear (#31) (`77b090a`)
- Method names now show their function and expected return value much more clearly (#33) (`84ec6e6`)

- Add fallbacks for `Piece.Generation()` in case generation settings are not acceptable (`a6dcbe6` , `e964b60` , `7550c3b`)
- `Synthesiser.AudioSource` is now publicly accessible (`9a31acf`)
- Add current load status messages to `Stave` if `VMManager.VMDebug` is equal to `true` (`3db1f96`)
- General code improvements and small bug fixes (`52b456d` , `8c3130c` , `fb938cd` , `ee2afda` , `6c7031a` , `90f1be7` , `1e75122` , `180d2b0` , `413efd9` , `19d29a6` , `0931ac6` , `2940158`)

H.2 Performance Test (Étude 3)

What changes have been made since the last performance test, and what is the focus and expected result of these changes?

Polyphony (outside of supplying multiple staves) is now possible! The octave of the second note may be altered either up or down any number of octaves. Notes which have an altered octave will remain at the same height as their original note, but will change shape to reflect their octave change; Notes with an upward octave change are triangles with their points facing upward, and conversely, notes with a downward octave change are triangles with their points facing downward.

Title slides have been added. These title slides will by default show the title and composer, but can also include the date of composition. Each element can be enabled or disabled at will.

Live tempo adjustments, such as accelerando and ritenuto have been added. These can be done by tilting the control sticks forward or backward, respectively. It is now possible to indicate relative tempo adjustments with the previously added tempo markings.

Haptic feedback has been added to the `Virtual Music Instrument` when in `Simple` mode. This gives the performer vibrational feedback, letting them know that they are activating a note.

Dynamics are now freely controllable when in `Intermediate` mode. This further distinguishes `Simple` and `Intermediate` mode, and feels appropriate as it is something more for the performer to do and pay attention to (in-line with the concept of 'virtuosity'), but is still not 'advanced'. It is also a good stop-gap solution to issue #21, which states that it is not possible to playback hairpins in `Simple` mode.

Did the changes made achieve the expected result? If not, what could be done differently to achieve the expected result? What worked well, and what didn't work well?

Polyphony works well, and notes that have their octave altered are especially clear. There was little impact to the visibility of notes further in the piece thanks to previous changes in transparencies. I felt, however, that the shape of notes that had their octave altered did not fit very well with how everything else looks. While everything is square edged, I felt that the 'points' on the outside edges of the notes, where the inside angle was less than 90 degrees, were too sharp and felt out of place. Additionally, as the mesh would become very thin on the outer edges, it also contributed to the dynamic indication and therefore the note being unclear. I would prefer that octave changes had a pentagonal or 'house-like' shape, where the triangle is added to the upper or lower half of the note's usual rectangular shape. This should remove any angles that are less than 90 degrees in most circumstances, though there is a possibility that the 'point' of the triangle may have an angle less than 90 degrees when a note has an extremely low dynamic. It would be interesting to be able to adjust octaves on single notes, as it is currently limited to the second notes in chords only.

Title slides worked well, and again added significantly to the `Virtual Music`'s feeling of 'completeness'. It would be best to be able to define the location, rather than it being hard-coded as half the position of the first note.

Live tempo adjustments worked well, though I found them a little tricky to use. Currently, the `RealTimeScale`, or the speed of which the `Virtual Music Notation` approaches the performer, is a multiple of the `TimeScale` and both sticks together. I found it easy to adjust the tempo too far, too quickly. At this time, I do not see another button mapping which would improve the usability of these tempo adjustments, but I think that adjusting how the sticks impact the `RealTimeScale` may make the feature easier to use. This could be done by fine-tuning the amount of adjustment each stick makes, altering the functionality of the second stick when both sticks are used simultaneously, or by increasing the amount of adjustment the sticks make the longer they are used. The latter could perhaps be an option, or a feature added to the second stick.

The addition of haptic feedback is a good change and again contributes positively to how `Virtual Music` feels when performing. Similarly, making dynamics freely controllable when in `Intermediate` mode is a good change and is in-line with the `Virtual Music`'s goal of retaining the concept of virtuosity.

Were there any adverse or otherwise unexpected effects of the changes?

It should be noted that now that polyphony has been introduced, repeated notes happen more regularly as when progressing through a piece, what was the upper note of a chord may become the lower note of the next chord. For example, on a piano, when playing a C and E chord, followed by an E and G chord, the E is repeated. I did not see an issue here, as notes will still generate a start and end texture, so it is clear visually that the note is repeated at the same pitch and is not just one long note, but it is important to note in case this comes up in later versions.

Were any new issues discovered? How can these new issues be dealt with?

No new issues or bugs were discovered in this performance test.

Notes for moving forward

- Adjust the shape of notes with octave alterations.
- Allow octave alterations on all notes, not just the second notes of chords.
- Allow defining the `InformationSlidePosition`.
- Fine-tune the live tempo adjustment feature, or;
- Investigate other implementations of live tempo adjustment.
- Keep in mind repeated notes. If the fact that they are repeated immediately after each other is not clear in future versions, it will perhaps be necessary to investigate their clarity.
- Investigate a true solution to issue #21.

I v0.2.1

I.1 Release Notes

v0.2.1 `372db37` 185 commits to master since this release

Issues carried from last version

- ~~#21 [BUG] Hairpins are not possible in `Simple` mode~~

✚ Issues added since last version

- ~~#42 [E] Introduce a way to notate another note which starts at the same time as another but has a different length~~
- ~~#43 [E] Add a way to hold/sustain a note currently being touched~~
- ~~#44 [E] Add the ability to define the rhythmic location of the information slide rather than dividing the `FirstNotePosition` by two~~
- ~~#45 [E] Allow octave adjustments on single notes~~
- ~~#46 [E] Include and allow the activation of a secondary timbre (synthesiser)~~
- ~~#47 [BUG] Rests are broken~~

New features

- The `Virtual Music Instrument` now has two timbres which can be sounded and notated, differentiated both by sound and appearance (#46, `ce9c072`, `2c5c74a`, `3c31f11`, `a801332`, `dbf1d10`, `372db37`)
- Can now sustain a single note of each timbre, and the sustained note will act as though it is being touched (#43, `6d5e57c`, `39b9bc6`, `07a58c1`, `b2b969b`, `ce9c072`, `2c5c74a`, `3c31f11`, `999c0d8`, `1a0d701`, `56ec83e`, `ba2d4e2`, `627f297`)
- Added `Note.Current*` which returns the current frequency, amplitude, or dynamic according to position of the `Instrument` in relation to its progress through the note, adding support for hairpins and other live changes such as glissandi (#21, `8ba3738`, `4456c7d`, `67636ff`)

⚠ Major changes

- Polyphony has been improved from being limited to the same dynamic and length to being completely independent, though always starting at the same time as the other note in which it shares its rhythmic starting position (#42, `a534465`)
  - This can be used in conjunction with rests to arpeggiate!
- Octave adjustments are no longer limited to second notes in a chord (#45, `a534465`)
- The performer no longer has to be at unity transform position! (`9db80b1`)

🐞 Improvements and bug fixes

- Can now specify the rhythmic location of the information slide and appropriately change position when its position is less than `FirstNotePosition` (#44, `43ed962`, `00d92c1`)
- Fixed rests which broke with some recent refactoring (#47, `248d0bd`, `1d6c797`)
- Added the ability to get the position of any hand/instrument (`1fef297`)
- Fix mesh generation as meshes were generated flipped according to the addresses expected (`63341df`)
- Make mesh generation easier to understand, improving extensibility (`c1c6abf`, `10ba28f`)
- Improve mesh shapes, most importantly, make octave alterations more obvious by making the top edge thicker and not pointed (`1e4ff40`, `c15a774`)
- Improve `Instrument` responsiveness and touch registration when note size is very small (`eb42914`)
- Notes can no longer sound once they've passed (most notable when sustaining) by appropriately checking for `IsFinalised` and `Playable` (`7492f30`)
- Add `Note.AddCustomProperty` (`e696f24`)
- Restore sanity checks during `Piece.Generate()` (`34fd716`)
- Preemptively add `Accent` note shape, add (and later remove) ability to notate accents (`0540160`, `885de2b`, `0a480af`)
- Make `Piece.Generate` synchronous again as the calculation is currently very quick so is not necessary to make asynchronous (`dd0ad88`)
- Adjust execution orders to make `VirtualMusic` load at an appropriate time (`7edb1d3`)

- Implement lazy loading and better handle singleton pattern in `VMManager`, removing the requirement of adding a `VirtualMusic` `GameObject` (`f2fd3ba`)
- Add comparative methods (`1a0d701`, `56ec83e`)
- Code tidy, small bug fixes, optimisation, and refactoring to improve usability, comprehensibility, and extensibility (`8cbe84e`, `4d86e76`, `3b81752`, `de63daf`, `67636ff`, `f91aa4f`, `1e4ff40`, `eefe92a`, `ea5fb68`, `ba3d6d7`, `c3bb8e8`, `9843e14`, `8902419`, `fadb810`, `d892949`, `1768d97`, `1d8e8fe`, `63bfd78`, `81c5977`, `a0e5fdf`, `97a0dc5`, `ad90386`, `b2df28f`, `1ed93d2`, `0298f55`, `5d81def`, `372db37`)

I.2 Performance Test (Étude 4)

What changes have been made since the last performance test, and what is the focus and expected result of these changes?

The ability to notate to different timbres has been added. The secondary timbre can be notated with the addition of a `!` character before the `Note`'s `Shorthand`. This secondary timbre is differentiated both visually by using a different material and audibly by activating a different `Synthesiser`. The addition of this secondary timbre creates new possibilities for tonal quality by allowing the composer to specify different sounds for the `Virtual Music Instrument`.

A single `Note` of both the primary and secondary timbres can now be sustained. Additionally, polyphony is no longer limited to `Note`s having the same dynamic and rhythmic duration. Though this will still happen by default if only a pitch degree is given, polyphonic `Note`s (or rather 'chords') can now be completely independent of each other. Interestingly, this can allow arpeggiations when used in conjunction with rests. The restriction limiting octave adjustments to the second `Note` in a chord has also been removed. These changes were made to allow for greater creative opportunity.

Various methods have been added to `Note` which return the current frequency, amplitude, or dynamic. Importantly, these methods return their respective values in relation to how far the supplied `Instrument` has progressed through the `Note`, adding support for hairpins and other such live changes. This fixes long standing issue #21, which describes the fact that hairpins are not possible in `Simple` mode.

The shape of `Note`s altered with an octave difference has been improved by making the wider edge thicker and no longer pointed. This change was implemented due to comments made in the previous performance test, which explained that the pointed edges on `Note`s with an altered octave were unclear and additionally out of place.

`Note`s are no longer playable once past the performer. This change was made to prevent any timing issues caused by the non-dominant hand being rhythmically later than the dominant hand, especially notable when `Direction` is set to anything other than `Straight`. It also makes it more clear where the performance is up to, as it is now

more clear that once the `Note` is past the dominant hand, it is finalised and no longer playable.

Some changes have been made to improve the ease of use and to give the composer and the developer more options. First, the developer no longer has to place the performer at `Vector3(0, 0, 0)` position. This change allows greater freedom in creating performance environments, such as the beach environment as created in this performance test. Second, the composer can now specify the location of the information slide. If the specified location is later than the position of the first `Note`, the slide will move to the side so as not to get in the way of the performance. Finally, the setup requirement of adding `VMManager` to a `GameObject` has been removed thanks to the implementation of 'lazy loading'. This change should make it easier to set up a project for use with `Virtual Music`, therefore allowing greater accessibility.

Did the changes made achieve the expected result? If not, what could be done differently to achieve the expected result? What worked well, and what didn't work well?

Secondary timbres as a whole worked well. How to notate them is clear, and they function as expected. Differentiating secondary timbres with a different material worked well, but it is clear that some thought must be made in order to differentiate the different timbre more clearly, as the material used for this piece (a white material and a light yellow material for the primary and secondary timbres respectively) did not work well and the difference between the materials was not clear. This may be desirable in some instances, say for example if the second timbre is intentionally very similar, of if the use of a secondary timbre is intentionally being hidden when interpreting as a performer playing an acoustic instrument. However, in this case, that was not the intention. The performance environment, being a beach scene with warm colours and bright lighting, perhaps contributed to the poor visual difference as well. The materials chosen would have worked better in the nebula-like environment as used previously. It should also be noted that the default `Synthesiser` for the secondary timbre is - for the lack of a better term - terrible.

Removing the notational restrictions from `Note`s in chords is good and gives the composer more creative freedom by allowing them to more easily notate counter melodies and such things without resorting to a second stave. However, it is important to note that what dynamic is remembered as the last dynamic is not necessarily clear intuitively and must be made clear in the documentation. Additionally, since second `Note`s no longer have these restrictions, it is also possible and in fact easy to compose something that is simply not playable. It is easy to notate subsequent `Note`s that will run into each other or notate more notes than is possible to perform. The latter is especially true considering that it is currently only possible to sustain one `Note` on each `Instrument`, equalling four `Note`s in total. While this could be desirable in the some instances, for example if the composer is giving the performer a choice of `Note` to perform, unless that fact is made clear to the performer (which in my opinion is the responsibility of the composer to do so), it will just appear as though it is an error and will easily cause frustration.

Also noted is that arpeggiations using rests work well, but should there be an easier way to do this? Currently to notate an arpeggiation, the sustained `Note` is notated as a secondary `Note`, with the first being a rest for the duration until the next `Note`. For

example, `0q{1b} 5m.-m` indicates that a breve on the first pitch degree is to sustain, and a dotted minim tied to another minim at the fifth pitch degree is to begin one quaver after the first `Note`. It is clear that this notation is not intuitive; though it is easy to get used to, it feels like a workaround rather than intended use. It also makes the shorthand notation more cluttered and harder to understand.

The new octave difference shapes work well. Their shape is less pointed and feels better. They are also now much easier to see, as the pointed edges from previous versions would occasionally appear invisible to the performer thanks to the viewing angle, and this no longer occurs.

The ease of use improvements including the ability to change the position of the information slide and the ability to specify a location for the performer that is not `Vector3(0, 0, 0)`, like other changes, works well and makes `Virtual Music` feel more like a complete, workable, thought out product. The ability to adjust the location for the performer most notably makes it considerably easier to create more interesting performance environments. Interestingly, it should also allow the performer themselves to move around in the environment. This could be used in game-like scenarios, where the performer has control over how they move within the scene, or an 'on rails' scenic environment, in which the developer or composer is in control of the performer's movement. In either scenario, care must be taken to ensure that the performer does not suffer from virtual reality sickness, particularly by forcing the performer to move in ways that the performer is not prepared for.

Making `Note`s unplayable once past the performer correctly prevents their accidental activation. One may argue that this change limits the ability to play with a sense of rubato or prohibits the performer playing either before or after the actual beat as a creative choice, but I would argue that this still can be achieved with creative use of the non-dominant hand and live tempo adjustments. In fact, I would argue that this method is more preferable as it allows the use of accompaniment tracks and potentially allows for a multi-performer `Piece`s in the future, as live tempo adjustments would be applied to all performers simultaneously. I would also argue that the benefits of this change, especially that time is now consistent and therefore followable by other performers (both real and not real) and the prevention of accidental activations of `Note`s in some wild gesture celebrating the end of a `Piece` and therefore spoiling the performance (outside of the control of the performer), outweigh the need to facilitate the use case where a performer would like to use the dominant hand for such a purpose. This would perhaps instead raise the importance of improving the live tempo adjustment controls to make these possibilities more easily accessible.

Were there any adverse or otherwise unexpected effects of the changes?

The new methods to ascertain the current progression of an `Instrument` through a `Note` and therefore also the current amplitude and dynamic inclusive of any hairpins work well. They finally make it possible to hear crescendos in `Simple` mode. However, the effect of the crescendo is perhaps a little too subtle, but this may be a reflection of the 9 different possible dynamic levels rather than the effect of the crescendo itself. Currently, the `DefaultDynamics` object has 9 different dynamic levels ranging from 0 to 1. These dynamic levels are split evenly by 0.1 other than between 6 to 7 and 7 to 8, which are

split by 0.15f. It may be beneficial to add some sort of curvature to the dynamic levels, meaning that the extreme ends of the dynamic levels have a smaller difference between surrounding levels than a dynamics that are mid range. This is also reflective of how a lot of audio software works. It should be noted that overriding these dynamic levels is already possible by overriding the `Piece.GenerationSettings` object, but improving the default configuration would be better.

Different voices brought attention to issues related to notes stopping and starting as touching for prolonged periods while paying attention to other notes is a little difficult. May be interesting to investigate a 'touch buffer'.

Implementation of lazy loading worked well but brought attention to the fact that setup is still confusing, in particular with the `Instrument`. The process of adding `Instrument`s is a little confusing and goes against the ideology of Virtual Music being accessible, simple to use, and intuitive.

It is sometimes not clear how multiple notes should be sustained and the performer will try and play three notes with one hand. Is this a problem with the notation, or is this a problem of practice and rehearsal?

Were any new issues discovered? How can these new issues be dealt with?

In advanced mode, it is difficult to tell what pitch the instrument is at. Investigate a 'pitch guide' and methods to ascertain which pitch the instrument is at (consider for showing when a note is at the same height as the `Instrument`, as well). There are also still some issues with detecting the furthest note.

Writing a piece in an external application and pasting it into the `Piece` script can cause errors. Investigate trimming whitespace.

Notes for moving forward

- Articulations are next
- Improve the default secondary `Synthesiser`
- Consider other methods to notate arpeggiations.
- Add some sort of curvature or fall off to the dynamic levels.
- Trim whitespace from `Stave` strings
- Still need to improve furthest note detection
- Implement methods to ascertain the note degree value of the `Instrument`
- Use these methods to make the notes show when they are the same height as the `Instrument`
- Consider ways to improve setup, in particular with the `Instrument`

J v0.3.0

J.1 Release Notes

v0.3.0 `71f8a81` 86 commits to master since this release

Issues carried from last version

N/A.

✚ Issues added since last version

- ~~#48 [E] Allow developers to specify which GameObject the Instrument should follow with an optional offset~~
- ~~#49 [E] Add Flagged Actions~~
- ~~#50 [E] Extend flags to also add flagged Notes to a collection based on that flag~~

New features

- Articulations have been properly added!
  - Articulations can be specified in the Shorthand (`90c82c0`)
  - The available articulations are listed in the `Note.Articulation` `enum`, currently including `Staccato` and `Accent` (`f8685cd`, `341d864`, `55b3cd6`, `909627f`)
  - Notes store their articulations in their `.Articulations` `HashSet` (`341d864`)
  - Accents affect the amplitude of the Note, audible when played with "Simple" mode (`32fc7ed`, `958d029`, `91bdd72`)
  - The effect of an accent is dependent on the Note's dynamic, where an accent on a Note with a lower dynamic has a greater effect than an accent on a Note with with a greater dynamic (`b61f090`, `958d029`, `09dfc40`)
  - Accents also affect the shape of a Note, and Note shapes incorporate accented varieties with a nullable `float? accentWidth` (`abf84e1`, `958d029`, `09dfc40`)
  - Staccatos shorten the playable length of a Note by half, and `Note.Length` and `Note.Duration` now distinguish the length of the Note without articulation and the length of the Note including the effect of any articulation respectively (`ae9ed50`)

- When in "Simple" mode, the Instrument will now have an elastic "touch buffer" (`5cdf6c3` , `83fe43b` , `4dad294`)
  - The touch buffer allows some lenience with touch registration, improving usability by leaving a Note as "Activated" for a short amount of time even if the Note is no longer being touched but the Instrument is nearby
- A "Pitch Guide" now shows the performer exactly where the pitches are (`a5c2dd0` , `e4fb293` , `313d8f8` , `96e011a`)
  - The "Pitch Guide" is on by default in "Advanced" mode
- Notes can now have "Flags": a custom string or strings assigned to them with Shorthand that can be used to more easily identify the Note (`33df92c`)
- Notes with "Flags" are added to a collection attached to the Piece, accessed by the Flag as a `string` (`2838ef8`)
- Notes can now activate a Piece's "Flagged Actions" (#49, `8a7f080`)
- Instruments can now follow a supplied `GameObject` (#48, `254eaa9`)
  - This can be used for various custom functionality, from locking the position of the Instrument in a game-like scenario, to accessibility implementations where the location and rotation of a `GameObject` is controlled by an accessible controller
- Virtual Music now has persistence between scene changes and persists by default (`cb275b8` , `5b98f25` , `60a1439`)
- Instruments are now added automatically if they don't already exist (`3720f41` , `162f27d`)
- Can now have multiple Instruments of the same type, with one set as the "Primary Instrument" (`ef0cf4b`)
  - The "Primary Instrument" will be set automatically if one is not supplied (`8a445c0`)
- Added methods to ascertain the pitch degree of a supplied height (`6701098`)
- Added `Instrument.CurrentPitchDegree` (`7266d03`)
- Notes now show when they are the same height as an Instrument to improve usability by offering the performer visual assurance that they are at the correct height to activate a Note (`d37489e`)
- Added various customisable "pre" and "post" `Note` actions (`33877ee`)
- Add events to `Environment` , including events for when note heights and frequencies change (`2b446f7`)
- Added `Generated` and `Invoked` events for Settings Blocks (`25fcbb4`)

⚠ Major changes

- Inheritance has been introduced for `Stave Object`s, including `Activatable Stave Object`s, which `Note` inherits from (`9ff8aef`, `59300f0`, `f200eb8`, `2ba46ec`, `44b84b2`)
  - This allows the easier creation and inclusion of new and custom objects to be included in a Stave
- A `Stave`'s `Stave Object`s are added to a collection automatically and can be obtained using `GetObjectsOfType` (`85fe13f`, `c071e3e`)
- The Events System has been reverted back to static events to make the classes which the events relate to a centralized point (`1238042`, `aea5441`)
- Shorthand notation can now handle any character case, most notably in BPM changes (`97ba5c7`, `760f20f`)
- The state or play-ability of a Note can now be ascertained with `Note.State` (`e27567a`, `bdf2c6d`, `cd9a3c1`)
  - `Note.State` (or `ActivatableObject.State`) can be overridden to create custom functionality
- The Instrument will retain a vertical rotation by default (`0ba1b8f`)
- "Trigger" convention has been inverted: the Instrument is now the trigger (`f24a64d`)
- The `Collider` for Notes is no longer convex to better reflect the Note's shape, as Notes do not have enough polygons for this to have much of an impact (`f24a64d`)
- `DestroyNotesWhenPastPerformer` has been removed (`76c7b26`)

🐞 Improvements and bug fixes

- `Play()` and `Update()` can now only be called once (`83a8b4c`)
- Finalised notes are now better handled by the Instrument (`a1fa8c1`)
- The instrument now adjusts the size of the `Collider` rather than the scale of the `GameObject` (`a5c2dd0`)
- After various iterations, the Instrument should now handle the "furthest" Note by it's position in the Stave. Later positions will be preferred over earlier (`ea9429c`, `343874b`, `16b58d7`, `e625153`)
- Position getters for hands and Instruments has been moved to static methods on `Instrument` (`388d408`)

- Overrides are now greatly simplified thanks to nullables and nullable coalescing (`1031711`)
- `Instrument.CurrentTouchedNotes` is now a `HashSet` to better reflect its purpose (`3c7396f`)
- Add some checks to prevent multiple assignments (`c2f83ee`)
- Trim whitespace characters from Shorthand, fixing unhelpful errors (`3c0446b`)
- Move `PieceDirection` to `Piece.Direction` (`1553a12` , `ad71191`)
- `VMManager.VMDebug` has been renamed `VMManager.Debug` (`7e7904e`)
- Begin to add equatable modifiers (`5f45aa0` , `775297b`)
- Various bug fixes, performance improvements, new methods, documentation, and refactoring to improve usability (`fc0f58f` , `01f1198` , `f186de6` , `565da29` , `fc58285` , `8cb3227` , `1403346` , `626f80f` , `7f5bf84` , `1cd97f2` , `edc5271` , `a503768` , `a100736` , `b546ae5` , `361b6a0` , `cc963cb` , `e796f1d` , `5be135e` , `54029be` , `4c6d0ef` , `4f48a48` , `ebe61ad` , `5248aff` , `100e84e` , `822fd28` , `a697323` , `aea5441` , `ded3dde` , `3c495e6` , `fe79acd` , `71f8a81`)

J.2 Performance Test (Étude 5)

What changes have been made since the last performance test, and what is the focus and expected result of these changes?

Articulations have been added! They include staccatos and accents. The width and effect of accents is impacted by the initial dynamic (see release notes) Staccato shorten the rhythmic duration of a note by half by default.

The touch buffer has been added. The touch buffer allows a performer to briefly stop touching a note, and the note will remain active. This should help touch registration and make Virtual Music friendlier and more forgiving to use, as it is often difficult to keep the hands absolutely steady. This should also assist in therapeutic scenarios, and makes it easier for performers with tremors to perform.

The pitch guide has been added. It is enabled by default in the Advanced Performance Mode, and shows the performer where the pitches are. This allows the performer to more easily produce the pitches that they intend. They utilise a custom shader so that the lines of the pitch guide only show around the instrument, reducing user interface clutter.

A number of extensibility features have been added allowing significant custom functionality. They include note flags, flag collections, and flagged actions. They allow

the activation of certain functionality for particular notes, as previously it was only possible to invoke methods with the event system. The ability to make an `Instrument` follow a `GameObject` has been added but this won't be commented on as it's use is limited to custom functionality, so debugging should happen elsewhere. This creates interesting options for implementation, such as overriding the position of the instrument in a special mode, and, more importantly, allows the use of accessible controllers for performers with limited mobility. `Instrument`s now also add themselves, making setup significantly easier. They may still be added manually in advanced situations, such as when overriding their position with an accesible controller. It is also possible to add multiple `Instrument`s. One is automatically set as the primary instrument. The position of any or all of these instruments can be offset. This can be used for game like situations (for example making the instrument turn into multiple or something similar). Various 'pre' and 'post' `Note` actions have also been added. Inheritance for `Stave Object` has been introduced, including `Activatable Stave Objects`, which `Note` inherits from. These `Stave Object`s are automatically added to a `Collection` and these `Collection`s can be accessed through the `Stave`s `GetObjectsOfType` method, so extensibility is inherently supported. Excess whitespace characters are now trimmed from the shorthand (preventing unhelpful issues when adding staves, increasing ease of use).

Methods to ascertain the `CurrentPitchDegree` have been added. Using these methods, `Note`s now show when they are at the same height as the `Instrument`.

Touch registration has been improved in a number of ways. The `Instrument` maintains a vertical rotation by default, which should assist in touch registration. It automatically faces the correct way, according to `Piece.FromDirection` The `Instrument` adjusts the `Collider` rather than the scale of the `GameObject`, which should assist touch registration. I also realised I was being too clever with furthest detection and opted to get the 'furthest' `Note` by the `Note`'s position in the `Stave`. This was much simpler to implement and makes much more sense than the previous mechanisms. It also better supports all directions.

Some changes to `SettingsBlock`s have also been made. They are now invoked based on the position of the dominant hand. This means that they are now invoked at their precise rhythmic position. Shorthand notation now supports any character case, most notably in BPM changes.

Did the changes made achieve the expected result? If not, what could be done differently to achieve the expected result? What worked well, and what didn't work well?

Articulations are good. The effect of an accent being dependant on the initial dynamic is good and seems more natural. Staccatos are currently a percentage of their initial rhythmic value. I wonder if they should they all go to a predefined length (and have a minimum/maximum/whatever so as to prevent any problems) instead. Additionally, the point of accents is straight. While this is fine, it is not particularly pleasing aesthetically, and it may look better to make them more curved. This also matches how they sound.

The touch buffer excellent and needs no changes. This is the same for the pitch guide. There is currently a method to override if the pitch guide is shown or not, but it may be

useful to also add a preference to the performer class (i.e. yes, no, recommended). It also uses the default texture. It would perhaps be good to be able to change that, eventually, but this is not required.

Extensibility improvements to `Note` are nice, but it would also be good to add extensibility in code via inheritance and virtual methods that can be overridden. `Instrument.CurrentPitchDegree` and furthermore works well, exactly as intended, and has negligible performance overhead. Using `Instrument.CurrentPitchDegree` to change the transparency of `Note`s when at same height is good. They could perhaps be a bit more opaque depending on the scenario, but their current values are a good default. It is easy to change on a per-piece basis. A lower transparency level for finalised `Note`s would be good, too.

Getting `Instruments` is much easier, but it's a little confusing to get the current position of `Instrument`s. Should the method to get the position be changed? Should it only return the `Primary Instrument`? Should it return the `Instrument` itself, or the position? Both maintaining a vertical rotation and changing the size of the `Collider` rather than the scale of the `Instrument` works well, and when combined with the touch buffer and the simpler approach to detecting the furthest (or rather, latest) `Note`, I no longer notice any issues with touch registration. Making the `Instrument`s load by themselves works great, but does make it less obvious that custom functionality can still be had if the `Instrument` is added manually.

`SettingsBlock`s invoking based on the position of the dominant hand works much better as it happens when expected. There could perhaps be a bit of a buffer, but this may cause issues on quick tempo changes, such as double times or anything like that. Supporting any character case in tempo changes is excellent and feels much better and more polished. It felt strange to see 'ritenuto' written with a capital 'R'. The ability to write different cases is nice but indications like 'ritenuto' or 'a tempo' doesn't work as you can't hide the number. Trimming whitespace characters automatically also makes it much more friendly to get started and removed unhelpful errors that even I was getting annoyed with.

Were there any adverse or otherwise unexpected effects of the changes?

Using the 'pre' and 'post' actions to delete or disable objects, including containers or staves, makes Virtual Music get a little buggy. It's even possible to prevent pieces being finished if the last note is deleted. This needs to be addressed by making use of the `OnDestroy` and `OnDisable` methods, and possibly the creation of a class maintains the `GameObject` that stores the staves and/or pieces.

Were any new issues discovered? How can these new issues be dealt with?

All extensibility improvements work well, however the ability to add any function means that it is easier to break things, so there needs to be significant focus on preventing things from breaking (such as altering/deleting `Note`s, etc.). For example, if the last note of a stave is deleted or disabled, the `Piece.Finalised` event will never trigger.

It's currently possible for different staves to specify settings blocks, which can cause at best confusion, and at worst bugs. It also contributes to confusion surrounding the difference between a stave and a piece. Settings blocks should be limited to the first stave only, either silently or with an error when `VMManager.Debug` is equal to `true`. This also brings attention to the fact that the ability to have multiple pieces loaded concurrently should be removed as it's getting too difficult to continue supporting it for such a limited use case. It is currently possible to have multiple pieces loaded, and this feature is only partially supported as it is difficult to ensure all functionality continues to work correctly. In some ways, continuing to vaguely support this feature is prohibitive and limits the ability to add new features, while significantly increasing difficulty in supporting existing features for a very limited use case.

`ClearParticlesOnStop` should default to `false`. Most particle effects would likely have some sort of trail and it's a bit strange when they are cut short.

If `DestroyOnFinalised == false`, `Stave`s will just stick around in place. This was particularly jarring as the final note of the piece stayed in range but wasn't playable. Staves should continue once they're finalised.

Notes for moving forward

- Implement virtual methods in `Stave Object`s, in particular, `Note`, which allow greater extensibility by supporting inheritance of the `Note` class, and in fact encouraging a developer to do so
- Change the `Note` transparency level when finalised
- Consider a custom method call when a `Note` is at the same level as the `Instrument`
- Simplify the way to get the `Instrument` in code
- Limit `SettingsBlock`s to the first `Stave` only
- Remove the ability to include multiple concurrent pieces
- Consider options for staccatos (will have to mention later that there was no time, could be good to talk about time management or something)
- Pitch guide in preferences (less priority)
- Pitch guide material change (less priority)

K v0.3.1

K.1 Release Notes

v0.3.1 `f958807` 57 commits to master since this release

Issues carried from last version

N/A.

✚ Issues added since last version

- ~~#51 [E] Add an option to hide just the BPM number value but still show the text~~
- ~~#52 [BUG] When `DestroyOnFinalised == false`, when a piece is finished, it just sticks around in the same place~~
- ~~#53 [BUG] `ClearParticlesOnStop` should default to `false`~~
- ~~#54 [E] Limit settings blocks to the first stave only~~
- ~~#55 [E] Remove the ability to have multiple pieces being loaded~~
- #56 [BUG] Virtual Music gets buggy when a ”Container” is destroyed or disabled

New features

- Various virtual methods have been added to `Note` to allow easy extensibility on `Start()` and `Update()` (`7aa8be8`)
- `ShowInformation` can be toggled on or off during a performance and will handle the information slide accordingly. Other elements may use similar functionality in the future using this at its model (`a870ada`)
- `Note`s will better show when they are finalised with a different transparency level (`a0c7059`)

⚠ Major changes

- The ability to have multiple pieces playing concurrently has been removed as supporting the feature inhibits implementation of new features, while having a very limited use case (#55) (`81a2d80`)

- Started refactor to an assembly to make use of access modifiers such as `internal`. This is very much still in progress and will take a few releases to be complete (`81a2d80`, `873708f`, `bafd3bf`, `837e38c`, `9261041`)
- `Performer` has been moved to its own static class (`VMPerformer`) (`5c8f056`)
- Getting `Instrument`s has been greatly simplified with the static `Instrument.Get()` method (`a4f2621`)
- `Piece.IsReady` and other similar properties have been removed in favour of `PieceState` (`81a2d80`, `4d19539`, `70402ca`)
- `SettingsBlock`s can now only be on the first `Stave` (#54) (`6991925`)
  - Can still force old behaviour by passing `true` to `Piece.Generate` (`f5d037e`)
- `SettingsBlock` has been revamped, removing the nested `BPMChange` class and now allowing displaying the tempo text only (#51) (`40bc812`)
- `Piece`s will continue advancing even when finalised (#52) (`f958807`)

Improvements and bug fixes

- Start supporting (or rather, dealing with) destroying or disabling notational elements (`e98b20b`, `ea9b6d5`, `b7e1a43`, `e13f157`, `9f2f313`, `34a3d5e`, `0907b5d`, `3f2b04c`)
- Naming notational elements is handled automatically and cannot be set more than once (`bafd3bf`, `b14ef21`, `e9e514a`)
- Setting the `Stave`'s `Piece` automatically adds the `Stave` to the `Piece` (`e9e514a`)
- Particles are no longer cleared on stop by default (#53) (`3853fa0`)
- Various bug fixes and refactoring (`dbfe7ac`, `af68ee8`)

K.2 Performance Test (Étude 6)

What changes have been made since the last performance test, and what is the focus and expected result of these changes?

`Note`s will show when they are finalised with a different transparency level. Expected result is that finalised `Note`s will be more clearly finalised.

The ability to have multiple `Piece`s loaded concurrently has been removed. This allows easier progression from a development perspective, as some functionality is dependant on which single piece is currently loaded and being performed and these functions would require special and sometimes convoluted exceptions (or just pick the first `Piece` anyway) in order to function at least somewhat correctly.

Various virtual methods have been added to `Note` giving the class increased extensibility. These won't be commented on as it's for the purpose of custom functionality.

`ShowInformation` can be toggled on and off and will respond to that. This means that if `ShowInformation` is toggled, the information slide will appear or disappear accordingly. Other elements may use similar functionality in the future so this change in particular sets a good model.

`Performer` has been moved to its own static class, `VMPerformer`. This will allow increased extensibility by making it easier to access the current `Perfomer`.

Getting the `Instrument` has been greatly simplified with static `Instrument.Get()` methods. Similarly, this will allow increased extensibility.

`SettingsBlock`s are now limited to the first `Stave` to make their purpose more clear and to simplify their use. This can be overridden by passing `true` to `Piece.Generate()`. `SettingsBlocks` have also been revamped in code, making them easier to understand and extend. It is now possible to show tempo text only, allowing indications such as 'a tempo' and 'ritenuto'. `Piece`s will continue even when finalised, as it seems more natural and means that they won't stick in place as with the previous version. `PieceState` has been introduced to more clearly ascertain the current state of a `Piece`.

This release has started implementing support for destroying and disabling notational elements, which allows doing things like destroying a `Note` or even a `Stave` within events or flagged actions. It also started the major move to an assembly structure to make use of access modifiers such as `internal`.

Particles are no longer cleared on stop by default. The option to enable this still remains.

Did the changes made achieve the expected result? If not, what could be done differently to achieve the expected result? What worked well, and what didn't work well?

Finalised `Note`s are indeed more clear, however despite the fact that the transparency level is set to a very low value by default, it's still not particularly clear when they are finalised rather than inactive. This is a problem more with Unity and the materials chosen rather than with Virtual Music. It could be interesting to directly support changing the material once finalised. This can already be done by utilising events, finalisation actions, or even extending the `Note` class and overriding a `PreUpdate()` or `PostUpdate()` method, newly introduced with this version. Though supporting this inherently, utilising a method as mentioned supports greater functionality (such as changing the material on activation, as well), and is already quite easy to implement. Supporting this functionality would in some ways limit or make the functionality of Virtual Music confusing by offering too many limited use cases rather than supporting greater extensibility as a whole.

Removing multiple `Piece`s makes `Environment.Piece` more reliable and makes it easier to implement custom functionality. I see no negatives of this change.

It is a little strange to toggle `ShowInformation` on and off. This is because it would be very rare to hide and show the information slide. However, the information slide actually following the Boolean feels more complete.

The move to a assembly structure is proving beneficial. It allows making static methods and accessing properties on instances of `Environment` or what have you more understandable, as there is now considerably less of them and they are more clearly related to what a user/developer will actually need.

Getting the `Instrument` and `VMPerformer` in code is now much easier and makes it easier to implement custom functionality. For example, it is now simple to create a custom camera that follows the position of the performer with an offset so that audiences can watch with more interesting camera work and angles.

Being able to show just 'a tempo' and other such indications is great! It also allows for 'fake-outs' and things like that. From the perspective of a performer, that makes it not entirely clear, but that's a decision for the composer rather than me, and allowing that functionality is better than just prohibiting it.

`Piece`s continuing once finalised feels much better, however since the `Finalised` event happens as soon as the `Piece` is finalised, any custom functionality happens instantly afterwards. Things like pulling up a finished screen with statistics on the performance, such as number of `Note`s missed and other such statistics, will feel better with a small delay. This, however, is more something that a developer using Virtual Music will need to consider rather than baking the functionality into Virtual Music, but would benefit from a mention within the documentation.

Particles not clearing automatically feels more natural. It also better supports immersion as existing particles will end as expected rather than disappearing.

Were there any adverse or otherwise unexpected effects of the changes?

While removing the ability to have multiple pieces loaded concurrently makes much more sense, a handful of features are still bolstered as they expect multiple pieces and have checks accordingly. Before release, it will be necessary to go through and remove any existing limitations on such features so that they can operate properly. Otherwise, the changes made did not have many, if any, unexpected adverse effects.

Were any new issues discovered? How can these new issues be dealt with?

Destroying notational elements works relatively well, but there is still work to be done here. Notably, overriding the `Note`'s `State` can cause problems, as seen when overriding a `Note`'s `State` as `Unplayable` when it was `Active` at the time. This made the `Note` act as though it was `Active` but stopped responding to any input, causing the `Instrument` to get stuck. The only time overriding a `Note`'s `State` would really be used properly would be to override a `Note` as `Unplayable`, so it may be better to add an option for that specifically, rather than a confusing implementation with exceptions than rules.

Notes for moving forward

- Fix the note state override situation, likely by removing the ability to override with any state and replacing it with a `OverrideAsUnplayable bool`.
- Check for remaining limitations that accounted for multiple concurrent pieces loaded simultaneously.
- Check for more places where `internal` or other access modifiers could be used.
- The `dynamic` keyword used to more easily retrieve all stave objects requires a .NET version greater than 2.0, Unity's default. It would be best to replace it with something else.
- Not many changes are necessary for features implemented with this development phase. It is definitely time to move on to improving the synthesis. Most notably, I would consider removing clicks in the synthesiser as a higher priority than any other issues.

L v0.4.0

L.1 Release Notes

v0.4.0 `1574b29` 30 commits to master since this release

Issues carried from last version

- #56 [BUG] Virtual Music gets buggy when a ”Container” is destroyed or disabled

Issues added since last version

- ~~#57 [BUG] `Note`s continue when overriding their state once they've started~~
- ~~#58 [E] Add synthesiser envelopes~~
- ~~#59 [E] Separate oscillators from the synthesiser~~
- ~~#60 [E] Add Low Frequency Oscillators (LFOs)~~
- ~~#61 [E] Track phase of the oscillator~~

New features

- Oscillators now track phase, eliminating artifacting and clicking! (#61) (`e514822`)
- Oscillators are now separate from the Synthesiser and are combined by the synthesiser to form additive synthesis by default (#59) (`d2bb179` , `4342d86`)
- `Oscillator` contains convenience fields and properties which allow easy modification of frequency and amplitude (`d30036b` , `d7d8ad0` , `2a3c1ce` , `e16a983`)
- LFOs have been introduced and can be added to oscillators, routed to either the oscillator's amplitude, or it's frequency (#60) (`d2bb179` , `21df028`)
  - As the LFO is an extension of the `Oscillator` class, and LFO can be added to the `LFO` class recursively (yo dawg I heard you liked LFOs, so I added an LFO to your LFO so you can oscillate while you oscillate your oscillators)
  - The modification of the routed parameter should be considered as `x = x + x * LFO` where `x` is the input value, and `LFO` is the waveform of the LFO, ranging from -1 to 1. The amplitude of the LFO should be

thought of as the strength that the LFO's output has on the routed parameter as a percent.

  * This means that if an LFO with an amplitude of 1 was routed to modify frequency, and the input frequency value was 220hz, the returned frequency would be 220hz plus or minus 1 * 220hz (220hz) (ie. 0hz to 440hz). If the amplitude of the LFO was 0.5, the returned frequency would be 220hz plus or minus 0.5 * 220hz (110hz) (ie. 110hz to 330hz).

- Amplitude envelopes have been introduced with a D+ADSR (Delay, Attack, Decay, Sustain, Release) envelope! (#58) (`ebb8a36` , `d8b10ad` , `2d17c2a` , `31f801a` , `1574b29`)
  - A single envelope can be applied to the Synthesiser as an "overall amplitude envelope", and can also be added to oscillators themselves
- Convenience methods have been added to the `Oscillator` class to get the amplitude and frequency once modified by the envelope and any LFO.
- Added a "white noise" oscillator (`800e14f`)

⚠ Major changes

- Synthesiser has gone through a major rewrite and is now much easier to extend (`ebb8a36`)
- Removed the ability to override the state of a `Note` as it caused more bugs than worth the limited use case (#57) (`32484d3`)
  - `Note` s can still be overridden as `Unplayable` with `Note.OverrideAsUnplayable` (`b6c0ede`)
- The `dynamic` keyword has been removed for .NET 2.0 compatibility (`60b5a70`)
  - This meant that fields returning the total loaded and estimated total objects for a `Stave` have been removed

🐞 Improvements and bug fixes

- Fixed the `Instrument` being stuck as `Active` in very rare situations (`104983e`)
- `Oscillator` clamps (or "clips") a sample between -1 and 1 (`3b5a65e`)
- Minor bug fixes and improvements (`3e95b8b` , `14d565d` , `f0e10f5` , `cd32431` , `787d5f1` , `78aa8a2` , `24d2225`)

L.2 Performance Test (Étude 7)

What changes have been made since the last performance test, and what is the focus and expected result of these changes?

A number of changes to synthesisers and oscillators has been made. Oscillators are now separated from the `Synthesiser`. They are combined within the `Synthesiser` class to form additive synthesis by default. This change makes it much easier to create new synthesisers, and allows oscillators to be independent, easily transferable between projects, and makes the `Synthesiser` class easier to understand by separating functionality into appropriately separated classes. Most importantly, their newly allowed independence allows oscillators to track phase, eliminating artifacting and clicking. It also allows portamenti and glissandi, among other live changes to the production of the tone. Oscillators now have convenience fields and properties that allow easy modification of frequency and amplitude, allowing better and easier customisation of sound quality. Low Frequency Oscillators (LFOs) have been introduced and can be added to oscillators, routed to either the oscillator's amplitude or its frequency. This allows for much better customisation of tone quality, including the ability to add vibrato or tremolo. Amplitude envelopes have been introduced either as an 'overall amplitude envelope' added to the `Synthesiser`, or on oscillators individually. This change allows much better customisation of tone quality, such as a decay time and speed, which can be used to create wildly different sounds. A white noise oscillator has also been added. While limited in use, it can be layered into a `Synthesiser` to create different sounds and allows for greater extensibility. Taking advantage of the separation from `Oscillator`s, the `Synthesiser` has gone through a major rewrite. This makes it much easier to extend and improves extensibility.

Other changes include limiting the overriding the state of `Note` to `Unplayable` only, fixing bugs noted in the previous performance test. The `dynamic` keyword has been removed for .NET 2.0 compatibility. This change was made to ease the requirements in getting Virtual Music running.

Did the changes made achieve the expected result? If not, what could be done differently to achieve the expected result? What worked well, and what didn't work well?

All changes to oscillators are proving good. It is much easier to create new synthesisers and tonal qualities without the necessity to have a deep understanding of Virtual Music, and both synthesis and code. The implementation of a new `Synthesiser` still requires some code implementation, however, which would detract some users from creating new synthesisers. It may be good to look into some sort of user interface or external application which can help users develop new `Synthesiser`s.

Phase tracking works extremely well and I am very happy with the result. It feels polished and makes Virtual Music feel almost complete and I am starting to see the finish line.

While glissandi and portamento are available in free mode and no longer produce artefacts, there is currently no way to notate them. The creation of `Oscillator`s would also benefit from some sort of user interface as with `Synthesiser`.

The convenience fields and properties that allow convenient alteration of an `Oscillator`'s tone quality are in fact convenient. Changing these during performance may be interesting and to support this it may be necessary to limit alterations to sane values as with `AmplitudeMultiplier`.

LFOs work well and open many doors. Much like with previous changes, the implementation of LFOs makes Virtual Music feel much more complete and considered, and makes the Virtual Music Instrument viable as a true synthesiser. In the future, it would be interesting to investigate the ability to route LFOs to other parameters, but for now amplitude and frequency will cover most bases and therefore is low priority. I would also like to look at making LFOs multiply the output by 0-2 rather than -1-1 (or at least having an option that does that) as I initially hoped, but I don't know how feasible or even desirable that is.

Amplitude envelopes work well and do not need any changes. However, these envelopes may be beneficial to further performance improvements, as if `SustainStrength` is equal to `0f` and the input time has reached `SustainTime`, or in other words, if an envelope implies that a tone has a finite life span, then once the tone has reached the life span it should be considered as finished and should therefor finalise or terminate. The white noise oscillator works fine and needs no further comment. Its use cases may need clarification in final documentation.

Were there any adverse or otherwise unexpected effects of the changes?

The simplification of `Oscillator` and `Synthesiser` has made it easier to create new `Synthesiser`s, but made it clear that it is still not trivial to override which `Synthesiser` is loaded. It will be finally necessary to simplify the way to override the default synthesiser, including adding the option to specify a `Synthesiser` on a per `Piece` basis.

The removal of the `dynamic` keyword did improve compatibility but came at the cost of removing the fields that return the total number of loaded `Stave Object`s. It may be possible to reinstate this functionality by cycling through all `List`s or something similar.

Were any new issues discovered? How can these new issues be dealt with?

It is currently possible to change the `FromDirection` of a `Piece` during performance, which will make the `Stave` move in a direction that no longer makes any sense, since generation is impacted by `FromDirection`. `FromDirection` should be locked once the `Piece` has been generated. `Piece`s, once destroyed, will remain in the `State` that they were before they were destroyed. This can cause an issue as if a custom function checks the `Piece`'s `State`, it will report incorrectly. It should revert back to `NotGenerated` when destroyed.

It is easy to severely impact performance by using too many LFOs and `Oscillator`s. It will be necessary to consider performance improvements before final release.

Notes for moving forward

- Investigate ways to make the `Synthesiser` stateless, so that `Note`s or some other data structure can be passed to it, and it will calculate the waveform for that object alone.
- If the above is implemented, it may be possible to have more than two tones of each synthesiser type, as the `Synthesiser` will not have to maintain its frequency, amplitude, and other tone generation related information.
- Simplify the way to override the default synthesiser, including adding the ability to supply or specify a `Synthesiser` on a per `Piece` basis.
- Create some sort of interface that allows or helps users create `Synthesiser`s.
- Consider any further limitations (or 'clamps') to the convenience fields and properties as seen with `AmplitudeMultiplier`.
- Consider performance improvements.
- Investigate a way to terminate or finalise a tone once it has reached its life span, if the `SustainStrength` implies one.
- Reinstate methods to ascertain the total number of loaded `Stave Object`s, if possible.
- Implement filters and audio as custom in code methods utilising Fourier transforms, via Unity's audio effects, or both.

M v0.4.1

M.1 Release Notes

v0.4.1 `546199b` 0 commits to master since this release

Issues carried from last version

- ~~#56 [BUG] Virtual Music gets buggy when a Container is destroyed or disabled~~

✚ Issues added since last version

- ~~#62 [E] Simplify way to override the default synthesiser~~
- ~~#63 [E] Implement time domain synthesiser effects such as delay~~
- ~~#64 [E] Implement frequency domain synthesiser effects and filters such as low pass filters~~
- ~~#65 [E] Make it easier to attach Unity audio effects to the synthesiser~~

New features

- Tone generation has been majorly refactored to a new `Tone` class structure, which handles its own state including tracking phase! (`6b91cd6`)
  - `Synthesiser` s and `Oscillator` s are now stateless and operate by handling `Tone` s supplied to them, which means...
  - There is now no limit to the number of `Tone` s that can be produced concurrently (up from 2!), outside of performance restrictions!
  - `Tone` s can be passed from one `Synthesiser` to another
  - Multiple `Synthesiser` s can exist concurrently with negligible performance impact
  - Less `GameObject` s are now produced which improves simplicity and performance
- Thanks to the new `Tone` structure, and more importantly the new ability to have multiple concurrent `Synthesiser` s, a `Piece` can now specify a `Synthesiser` ! (#62) (`89b547b`)
  - The default `Virtual Music Synthesiser` s can be attained using `Synthesiser.Get` , and these defaults can be overridden by changing the `Synthesiser` s on `VMManager` (`9acf98b` , `42b155b`)

- `Instrument.Output` has been introduced, which is now the central point for audio generation (`35fd4de`)
- Audio generation can now be piped to `Mixer` s on an overall basis, individual `Instrument` basis, and on a `Primary` and `Secondary` `Synthesiser` basis, which means... (`68d575e` , `39cc5e1` , `0bef34e` , `35fd4de`)
  - Time domain effects such as delay and reverb can now be implemented by adding Unity's inbuilt `AudioEffect` s to the `Mixer` s! (#63, #65)
  - Frequency domain effects such as high and low pass filters can be added similarly (#64, #65)
  - Please note that, especially notable for frequency domain effects, currently `AudioEffect` s can only be added to the `Synthesiser` at the lowest level, and not individual `Oscillator` s
- All aspects of the `Synthesiser` have been converted to `ScriptableObject` s (`ac04249`), which means...
- `Synthesiser` s and `Oscillator` s can now be created with Unity's Inspector!
  - The GUI is rather rudimentary for now, but is perfectly usable for those familiar with how the Inspector works
  - A custom Editor has been implemented to allow the creation of `TestTone` s to demo how a `Synthesiser` will sound (`395030e`)
- Added `Basic` `PerformanceModeOption` (`6b91cd6`)
  - Tone generation will occur when a `Note` is touched, and will automatically control its `Frequency` and `Amplitude`
  - `Note` s and `Tone` s will remain sustained without any button presses, which means that once a `Note` has started, it will continue until it is `Finalised`
  - `Basic` is considered as "simpler" than `Simple`
    - `Simple` remains the default `PerformanceModeOption`
    - `Basic` is recommended when button presses are too difficult or not possible, either due to physical ability or available controllers (what if the controllers don't exist? *foreshadowing*)

⚠ Major changes

- `StavesContainer` has been removed in favour of `PieceContainer` , which is a more simplified and less buggy approach to handling "containers", especially when `GameObject` s are destroyed (#56) (`87d3b65` , `260bc07` , `c7a1d2b`)
- Various fields on `VMManager` have been serialized, which allows editing in the Inspector (`a8b89c4`)

🐞 Improvements and bug fixes

- Added `Instrument.MaximumTones` const which is indicative of how many `Tone`s an `Instrument` can produce before causing clipping (`f0a0e4e`, `6b7a37f`)
- Prohibit changing a `Piece`'s `FromDirection` once the `Piece` has been generated (`1fbdf50`)
- Restored `TotalLoadedObjects` and `EstimatedTotalObjects` by utilising a LINQ method (`d143b6a`, `090c0f2`)
- Added easier method to attain the `EstimatedPercentLoaded` (`d143b6a`)
- `Piece`s revert back to `NotGenerated` once destroyed (`6d68514`)
- Settings per `PerformanceModeOption` are now set in the setter, which means that `PerformanceModeOption` can be changed directly and appropriate settings will be applied (`546199b`)
- General bug fixes and improvements (`d0b06ec`, `810010e`, `090c0f2`, `3349062`, `1519ac9`, `91072a2`, `f8911ad`, `17bb9c7`, `1e8b2f9`, `219ea67`)

M.2 Performance Test (Étude 8)

What changes have been made since the last performance test, and what is the focus and expected result of these changes?

Tone generation has been majorly refactored to a new `Tone` class structure, which means that `Synthesiser`s and `Oscillator`s are now stateless and just inform the `Tone` how its phase should increase. This means that there is now no limit on the number of `Tone`s being produced concurrently, outside of performance restrictions. `Tone`s can also be passed from one `Synthesiser` to another, which means that changing the `Synthesiser`s during performance is now possible, and as `Synthesiser`s are now stateless, many can exist concurrently with negligible performance impact. Importantly, this means that `Piece`s can now easily specify a `Synthesiser`! This is largely thanks to the introduction of `Instrument.Output`, which allows piping audio generation to mixers, in turn allowing the addition of filters and effects including any of Unity's inbuilt audio effects such as low pass filters and reverbs (frequency domain and time domain).

All aspects of the `Synthesiser`, including `Oscillator` and `LFO` have been converted to `ScriptableObject`s. This change has allowed the creation of a rudimentary user interface. The ability to test the `Synthesiser` in the editor has been added with the introduction of the `TestTone` class.

A new `Basic` `PerformanceModeOption` has been added. This mode is more simple to use than `Simple` mode and is aimed at users that have trouble sustaining `Note`s, as once a `Note` has been touched it will remain `Active` until it is `Finalised`, even if the performer is no longer touching it.

Various fields on `VMManager` have been serialized, which allows editing in the Inspector. Notably, this allows changing the default synthesiser, used if a piece does not specify a synthesiser. Other quality of life improvements have also been made, including making `Piece`'s `FromDirection` no longer changeable once the `Piece` has been generated, fixing the issue mentioned in the previous performance test. A `Piece` will now correctly revert back to `NotGenerated` once destroyed. Settings per `PerformanceModeOption` are now set in the setter, which means that `PerformanceModeOption` can be changed directly and appropriate settings will be applied. Methods to ascertain the `*LoadedObjects` have been restored, which will allow a progress bar in the future when and if asynchronous loading is implemented. A convenient `EstimatedPercentLoaded` property has been added accordingly. `PieceContainer` replaces `StavesContainer` and fixes long standing issues surrounding destroying or disabling Virtual Music related `GameObject`s.

Did the changes made achieve the expected result? If not, what could be done differently to achieve the expected result? What worked well, and what didn't work well?

Effects can only be added to the `Synthesiser` and not to individual `Oscillator`s, which somewhat limits options. There is also no clear way to adjust parameters of filters and effects. However, in my opinion, if a composer/ `Synthesiser` designer wants to have control of the tonal quality at that level, it is up to them to consider the best method of implementation. There are enough existing points of information, all easily accessible, of which a composer or designer can utilise in such a scenario. The user interface is indeed rudimentary, but works well enough to allow the easier creation of `Synthesiser`s. More polish to this may be good, such as considering custom list/array fields and buttons to add `LFO`s and `Oscillator`s. `TestTone`s only sustain, so it is hard to judge the release tail without trying it in engine. It would be good to be able to produce a `TestTone` that lasts for the specified time and that also has a release tail. They are also currently not impacted by mixers and therefore audio effects.

`Basic` mode works well. It has considerable uses in therapeutic or similar situations where a performer cannot sustain or has trouble sustaining `Note`s for an elongated period. `Simple` will remain the default, but `Basic` will be recommended for such scenarios and also when controls don't exist as with hand tracking.

`PieceContainer` seems to work well, but some refactoring will need to be done to ensure it encompasses all scenarios. For example, more investigation into when `VMManager.PieceContainer` is made `null` is needed, and what happens when something refers to `PieceContainer` when it is wrongfully `null`.

Were there any adverse or otherwise unexpected effects of the changes?

The default synthesisers set themselves up in `VMManager.Init()`. While this is required, it causes problems when overriding them, as changes by the user can be overwritten. A solution to this would be to still set up the default synthesisers as currently being done, but then store those in a private variable. Another field would be created to override the default, and then an additional field would be created which allows accessing the default synthesiser by first checking if an override exists, and if not, reverting back to the default.

Were any new issues discovered? How can these new issues be dealt with?

To get to `Basic` mode, a performer must first cycle through all other `PerformanceModeOption`s. It is finally necessary to extend the `Preferences` class with more options, as most shippable functionality now exists. `VMPerformer.OnVMDestroyed()` is always added to the `VMManager.Destroyed` event in `VMPerformer.Init()`. This causes `OnVMDestroyed()` to be called multiple times.

Notes for moving forward

- Polish the `Synthesiser` user interface.
- Add release tails to `TestTone`s.
- Investigate support for hand tracking with Oculus Quest.
- Further test `PieceContainer` as described above.
- Add a better method to override the default `Synthesiser`s that won't be overwritten with `VMManager.Init()`.
- Extend `Preferences` with more options.
- Investigate events being added multiple times, as seen with `VMPerformer.OnVMDestroyed()`.

N Preparing Unity for Virtual Reality Development

1. Download the *Unity Hub* from the *Unity* website[1].
2. Install and run the *Unity Hub*.
3. In the *Installs* tab, click *Add* and then select the latest Long Term Service version of *Unity*[2].
4. To support the *Quest*, when given the option, check the *Android Build Support* checkbox and ensure any submodules are also checked.
5. Create a new 3D project from the *Unity Hub*.
6. In the newly created project, import the from the *Asset Store*:
   (a) Navigate to *Window > Asset Store* in the *Editor*.
   (b) Search for *Oculus Integration* and click *Import*.
   (c) In the *Import Unity Package* window, ensure all files and folders are selected by pressing *Select all* and then click *Import*.
   (d) If prompted to update the *Oculus Utilies* or *Spatializer* plugin, click *Yes* or *Upgrade* and restart the *Editor* as required.
7. Enable VR support:
   (a) Navigate to *Edit > Project Settings* in the *Editor*.
   (b) In the *Project Settings* window, select *XR Plugin Management* and then click *Install XR Plugin Management* if it is not already installed.

[1] https://unity3d.com/get-unity/download
[2] Alternatively, any other recent version of *Unity* that is supported by the *Oculus Integration* package is also suitable

(c) For *Microsoft Windows* support, in the *Windows* tab[3], check the *Oculus Plug-in Providers* option.

(d) For *Android* support, in the *Android* tab[4], check the *Oculus Plug-in Providers* option.

The *Unity* project is now configured for use with an *Oculus* device. The *Configure Unity Settings* documentation in *Oculus for developers* describes additional recommended settings for optimal rendering, quality, and user comfort (Facebook Technologies, 'Configure Unity Settings'). While these settings are not required for the purposes of development, testing, and basic usage of VIRTUAL MUSIC, they are recommended by *Oculus* for performance and policy reasons before publishing or building an application for an *Oculus* device, and may change depending on the specifications of the development environment and targeted device ('Configure Unity Settings').

[3]Represented by an icon depicting a computer monitor
[4]Represented by the Android logo icon

O Preparing the Unity Project for Virtual Music

Import the OVRCameraRig

1. Navigate to *Assets* > *Oculus* > *VR* > *Prefabs* in the *Project* window.
2. Drag the *OVRCameraRig* prefab into the scene.
3. In the *Inspector*, verify that the *Position* is set to origin (0, 0, 0). If not, set it to origin by selecting the gear icon pull-down in the upper-right of the *Transform* section of the *Inspector* and clicking *Reset*.
4. Under *OVR Manager (Script)* in the *Inspector*, change *Tracker Origin Type* to *Floor Level*.
5. Delete the *Main Camera* object if it exists.

Show the Performers Hands in the Game-world

1. Navigate to *Assets* > *Oculus* > *Avatar* > *Content* > *Prefabs* in the *Project* window.
2. Drag the *LocalAvatar* to the *Hierarchy* window.

P Piece

The following is a detailed technical listing of public members of the `Piece` class, ordered by their first appearance in the relevant files.

type, Name, "default value" Description.

---

string, Title, "My Virtual Music Piece" The title of the Piece. This can be displayed on the "Information Slide".

string, Composer, "A Virtual Music Composer" The composer of the Piece. This can be displayed on the "Information Slide".

DateTime, DateOfComposition, "DateTime.Now" The date the Piece was composed. This can be displayed on the "Information Slide".

bool, ShowInformation, "true" If a slide showing the information of the composition should be show during the performance. If set to `false`, the `Title`, `Composer`, and `DateOfComposition` will not be shown despite the values in `ShowTitle`, `ShowComposer`, and `ShowDateOfComposition` respectively.

bool, ShowTitle, "true" If `Title` should be displayed on the "Information Slide".

bool, ShowComposer, "true" If `Composer` should be displayed on the "Information Slide".

bool, ShowDateOfComposition, "false" If `DateOfComposition` should be displayed on the "Information Slide".

(readonly) PieceState, State, "NotGenerated" The current `PieceState`. This can be used to ascertain if the Piece is currently ready, being generated, or any other `PieceState`. The value of `State` cannot be changed manually and is only meant for reference, for example with events.

enum, PieceState Possible values are `NotGenerated`, `Generating`, `Ready`, `Finalised`, and `Destroyed`.

bool, DestroyOnFinalise, "false" If the `Piece` should be destroyed once the `State` is set to `Finalised`. Destroying the `Piece` may be desirable in a situation where another `Piece` is loaded instantly when one is finished.

List<string>, ShorthandStaves A `List<string>` of stave strings that adhere to the Virtual Music Shorthand Notation standard. They will be used to generate the notation in `Generate`.

Direction, FromDirection, "Straight" The direction that the notation approaches the performer from.

enum, Direction The available directions that the `Stave`s can approach the performer from. Possible values are `Left`, `Right`, and `Straight`.

Material, NoteMaterialPrimary Which `Material` should be used for the generated `Note`s that have a `Note.Type` of `Primary`.

Material, NoteMaterialSecondary Which `Material` should be used for the generated `Note`s that have a `Note.Type` of `Secondary`.

GameObject, NoteObject, "null" If set, the mesh generation system will be ignored in favour of instantiating copies of this `GameObject`. Gradual dynamics are not supported when supplying a custom `NoteObject`.

bool, ControlParticleSystem, "true" If enabled an a `Note` has a particle system, the particle system will be activated when `Note.Activated` is set to `true`.

bool, ClearParticlesOnStop, "false" If set to `true`, and if `ControlParticleSystem` is also set to `true`, the particle system will clear all existing particles when `Note.Active` is set to `false`.

float?, NoteHeightOverride, "null" If set, the height of the `Note` will be overridden by this value. The recommended range is between `0.01f` and `0.1f`.

float, NoteWidthModifier, "0.8f" The modifer for the width of generated `Note`s.

int, BeatsForCountIn, "8" The number of beats before the stave begins.

bool, NoteTransparency, "true" If set to `true`, the `Material` of the `Note` will be transparent. The `Material` supplied must be compatible with transparency.

float, NoteTransparencyWhenActive, "0.5f" The transparency value of the `Note` when `Note.Active` is equal to `true`.

float, NoteTransparencyWhenInactive, "0.2f" The transparency value of the `Note` when `Note.Active` is equal to `false`.

float, NoteTransparencynWhenInstrumentAtSameHeight, "0.3f" The transparency value of the `Note` when the `Instrument` is at the same height as the `Note`.

float, NoteTransparencyWhenFinalised, "0.001f" The transparency value of the `Note` when `State` is equal to `Finalised`.

bool, NoteEmission, "true" If set to `true`, the `Material` of the `Note` will be emissive. The `Material` supplied must be compatible with emission.

float, NoteEmissionWhenActive, "0.5f" The emission value of the `Note` when `Note.Active` is equal to `true`.

float, NoteEmissionWhenInactive, "0.2f" The emission value of the `Note` when `Note.Active` is equal to `false`.

float, NoteEmissionWhenInstrumentAtSameHeight, "0.3f" The emission value of the `Note` when the `Instrument` is at the same height as the `Note`.

float, NoteEmissionWhenFinalised, "0.001f" The emission value of the `Note` when `State` is equal to `Finalised`.

bool, GenerateOnStart, "true" If set to `true`, `Generate` will run automatically on `Start`.

List<Stave>, Staves All of the generated `Stave`s.

GameObject, PieceObject The generated `GameObject` which contains the generated `Piece`.

IGenerationSettings, GenerationSettings, "DefaultGenerationSettings" The settings which will be used during `Generate`, including the dynamics and lengths of the `Note`s generated.

(readonly) float, FirstNotePosition The position of the first `Note` in the `Stave`.

int?, InformationSlidePositionOverride, "null" The rhythmic location for the Information Slide will be overridden by this value, in number of beats.

(readonly) float, InforamtionSlidePosition The location of the Information Slide in the `Stave`.

List<System.Action<GameObject>>, CustomNoteComponents Methods that add `Component`s to a `GameObject` or modify existing ones. Use to supply custom components to `Note`s via the `AddCustomNoteComponents` method.

void, AddCustomNoteComponents(Note note) Used to manually add custom components in `CustomNoteComponents` to the given `Note`. This method is invoked automatically in `Generate`.

Synthesiser, PrimarySynthesiser If set, the `Instrument` will use this `Synthesiser` for `Primary` `Notes`.

Synthesiser, SecondarySynthesiser If set, the `Instrument` will use this `Synthesiser` for `Secondary` `Notes`.

Dictionary<string, System.Action<Note>>, FlaggedActivationActions Methods within this `Dictionary` will be run when a `Note` has been activated and has the appropriate note flag.

Dictionary<string, System.Action<Note>>, FlaggedDeactivationActions
Methods within this `Dictionary` will be run when a `Note` has been deactivated and has the appropriate note flag.

Dictionary<string, System.Action<Note>>, FlaggedFinalisationActions
Methods within this `Dictionary` will be run when a `Note` has been finalised and has the appropriate note flag.

Dictionary<string, System.Action<Note>>, FlaggedFailActions Methods within this `Dictionary` will be run when a `Note` has failed and has the appropriate note flag.

Dictionary<string, List<Note>>, FlaggedNotes Contains `Dictionary<string, List<Note>>`s with all `Note`s that have note flags. The `Dictionary<string, List<Note>>`s are accessed and grouped by the note flag.

List<System.Action<Note>>, PostGenerationActions Methods which will run after a `Note` is generated.

List<System.Action<Note>>, PreActivationActions Methods which will run before a `Note` is activation.

List<System.Action<Note>>, PostActivationActions Methods which will run after a `Note` is activated.

List<System.Action<Note>>, PreDeactivationActions Methods which will run before a `Note` is deactivated.

List<System.Action<Note>>, PostDeactivationActions Methods which will run after a `Note` is deactivated.

List<System.Action<Note>>, PreFinalisationActions Methods which will run before a `Note` is finalised.

List<System.Action<Note>>, PostFinalisationActions Methods which will run after a `Note` is finalised.

(static) event System.Action<Piece>, Generated Will be triggered once the `Piece` has been generated.

(static) event System.Action<Piece>, Finalised Will be triggered when the `Piece` has been finalised.

(static) event System.Action<Piece>, Destroyed Will be triggered when the `Piece` is destroyed.

void, TriggerFinalised() Finalise the `Piece` and trigger the `Finalised` event.

void, TriggerDestroy() Remove the `Piece` from the `Environment`, destroy the `PieceObject`, and trigger the `Removed` event.

void, Generate(bool allowMultiStaveSettingsBlocks = false) Generate `Stave`s from shorthand in `Staves`. This method will replace the existing `Piece` in the `Environment`, if one exists.

float, TimeAdvanceLastCalled The time that the `Advance` method was last called. Used to prevent `Advance` being called multiple times per fixed update.

(static) Dictionary<char, float>, DefaultLengths A `Dictionary` containing the default rhythmic values. Their values are:

```
public static Dictionary<char, float> DefaultLengths = new Dictionary<char, float>
{
  { 's', 0.25f },
  { 'q', 0.5f },
  { 'c', 1f },
  { 'm', 2f },
  { 'b', 4f }
};
```

(static) Dictionary<int, float>, DefaultDynamics A `Dictionary` containing the default dynamic values. Their values are:

```
public static Dictionary<int, float> DefaultDynamics = new Dictionary<int, float>
{
  { 1, 0.10f },
  { 2, 0.20f },
  { 3, 0.30f },
  { 4, 0.40f },
  { 5, 0.50f },
  { 6, 0.60f },
  { 7, 0.75f },
  { 8, 0.90f },
  { 9, 1.00f }
};
```

class, DefaultNoteShapes The default noteshapes, including `Standard`, `OctaveUp`, and `OctaveDown`.

(static) GameObject, CreatePieceInformation(Piece piece) Used to create manually create the Information Slide.

Q Note

The following is a detailed technical listing of public members of the `Note` class, ordered by their first appearance in the relevant files.

type, Name, "default value" Description.

Q.1 Native members

enum, Articulation The available articulations. Available values include `Staccato` and `Accent`.

(readonly) Shorthand.Note, Shorthand The shorthand string that was used to generate the note.

(readonly) int, PitchDegree The pitch degree for the note.

(readonly) float, Frequency The frequency at the start of the note according to the current frequency set, including any octave differences.

(readonly) float, SecondFrequency The frequency at the end of the note according to the current frequency set, including any octave differences. Note that this field

currently returns `Frequency` with no alteration.

(readonly) int, OctaveDifference, ”0” The number of octaves that the note is altered in. Positive numbers mean that the note's octave has been raised. Negative numbers mean that the note's octave has been lowered.

(readonly) int, Dynamic The dynamic value at the start of the note.

(readonly) int, SecondDynamic The dynamic value at the end of the note.

(readonly) float, Amplitude The amplitude at the start of the note.

(readonly) float, SecondAmplitude The amplitude at the end of the note.

(readonly) bool, HasHairpin Returns `true` if the first dynamic is not equal to the second dynamic.

(readonly) float, Length The space between the start of this note and the start of the next.

(readonly) float, Duration The rhythmic duration of the note, including any articulations.

(readonly) Synthesiser.Type, Type The note's type, either `Primary` or `Secondary`.

List<string>, Flags A `List` containing all of the note flags for the note.

HashSet<Articulations>, Articulations A `HashSet` that contains all of the articulations for the note.

(const) float, AccentStrength The effect that an accent would have on the note's dynamic and amplitude values, equal to $2 - AccentPoint$.

(const) float, AccentPoint The position on the note where the accent will finish.

(static) event, System.Action<Note> Activated This event is invoked every time that a note has been activated.

(static) event, System.Action<Note> Deactivated This event is invoked every time that a note has been deactivated.

(static) event, System.Action<Note> Reactivated This event is invoked when a note has been reactivated (when it has been activated and `TimesActivated` is greater than one).

(static) event, System.Action<Note> Redectivated This event is invoked when a note has been re-deactivated (when it has been deactivated and `TimesActivated` is greater than one).

(static) event, System.Action<Note> Finalised This event is invoked when a note has been finalised.

(static) event, System.Action<Note> Generated This event is invoked when a note has been Generated.

(static) event, System.Action<Note> Failed This event is invoked when a note has been failed (when it has been finalised and `TimesActivated` is equal to zero).

void, Activate(Instrument instrument) Trigger the activation of the note by the given instrument. This method is called automatically during the performance mode handlers in the `Instrument` class.

void, Deactivate() Trigger the deactivation of the note. This method is called automatically during the performance mode handlers in the `Instrument` class.

void, Finalise() Trigger the finalisation of the note. This method is called automatically in the `Update()` method.

bool, OverrideAsUnplayable "false" If this field is equal to `true`, the stave of this note will be overridden as `Unplayble`, prohibiting its activation.

float, CurrentFrequency(Instrument instrument The frequency of the note according to the current position of the given instrument.

int, CurrentDynamic(Instrument instrument The dynamic of the note according to the current position of the given instrument.

float, CurrentAmplitude(Instrument instrument The amplitude of the note according to the current position of the given instrument.

(protected virtual) void, PreStart() If the `Note` class is inherited, this method will be run automatically before the `Start()` method.

(protected virtual) void, PostStart() If the `Note` class is inherited, this method will be run automatically after the `Start()` method.

(protected sealed override) void, Start() This method is an extension of *Unity*'s `Start()` method. It should not be overridden. See `PreStart()` and `PostStart()` for methods to inject code into the `Start()` process.

(protected virtual) void, PreUpdate() If the `Note` class is inherited, this method will be run automatically before the `Update()` method.

(protected virtual) void, PostUpdate() If the `Note` class is inherited, this method will be run automatically after the `Update()` method.

(protected sealed override) void, Update() This method is an extension of *Unity*'s `Update()` method. It should not be overridden. See `PreUpdate()` and `PostUpdate()` for methods to inject code into the `Update()` process.

(protected virtual) void, PreOnDestroy() If the `Note` class is inherited, this method will be run automatically before the `OnDestroy()` method.

(protected virtual) void, PostOnDestroy() If the `Note` class is inherited, this method will be run automatically after the `OnDestroy()` method.

(protected sealed override) void, OnDestroy() This method is an extension of *Unity*'s `OnDestroy()` method. It should not be overridden. See `PreOnDestroy()` and `PostOnDestroy()` for methods to inject code into the `OnDestroy()` process.

(protected virtual) void, PreOnDisable() If the `Note` class is inherited, this method will be run automatically before the `OnDisable()` method.

(protected virtual) void, PostOnDisable() If the `Note` class is inherited, this method will be run automatically after the `OnDisable()` method.

(protected sealed override) void, OnDisable() This method is an extension of *Unity*'s `OnDisable()` method. It should not be overridden. See `PreOnDisable()` and `PostOnDisable()` for methods to inject code into the `OnDisable()` process.

(static) Note, Create(Shorhand.Note shortHand, Stave stave) Create a new note with the given shorthand and add it to the given stave. This method is called automatically during the piece generation process.

(static) void, AddCustomProperty(System.Action<GameObject> property, Note note) Invoke the supplied method which adds a custom note property to the supplied note. This method is called automatically during the piece generation process.

(static) class, DefaultShape The default note shapes. Includes `Standard`, `OctaveUp`, and `OctaveDown`, each of which are of the `INoteShape` type.

Q.2 Inherited Members

ActivatableStaveObject

(readonly) ActivatableState, State The current state of the object. The available options are `Inactive`, `Active`, `Finalised`, and `Unplayable`. The value of this field may be overridden with the `OverrideAsUnplayable` field.

(virtual) bool, OverrideAsUnplayable If set to true, the `State` of this object will be overridden as `Unplayable`.

(readonly) int, TimesActivated, "0" The number of times this object has been activated.

(readonly) Instrument, ActiveInstrument The instrument that is currently activating this object, if any.

(readonly) Dictionary<Instrument.InstrumentType, float>, TimeStoppedTouch The time that the `Left` and `Right` instrument types last stopped touching this object.

StaveObject

(readonly) Stave, Stave The stave that this object is attached to.

(readonly) Piece, Piece The piece that this object is a part of.

(readonly) List<T>, ContainingList A `List` which contains objects of the same type and on the same stave.

(readonly) int, Number What number this object is on its stave.

(readonly) Vector3, CachedPosition The position that this object was last in. This is updated automatically in `Update()`.

(readonly) Vector3, CachedLocalPosition The local-space position that this object was last in. This is updated automatically in `Update()`.

(readonly) bool, InRange Returns `true` if the object is in the playable range.

float, GetProgressThroughNote(Instrument instrument) Returns the position of the given instrument in relation to the start and end of the object as a value between zero and one.

R Stave

The following is a detailed technical listing of public members of the `Stave` class, ordered by their first appearance in the relevant files.

type, Name, "default value" Description.

(readonly) Piece, Piece The `Piece` that the stave is a part of.

(readonly) string, Shorthand The Virtual Music Shorthand Notation string used to generate the `Stave`.

(readonly) int, Number The `Stave`s number.

(readonly) float, Length The length of the `Stave` in game-world units.

List<T>, GetObjectsOfType<T>() A method which returns a `List` of the requested `StaveObject`s in the `Stave`.

void, AddToObjects<T>(object obj) A method which adds the supplied object to the appropriate `List`. This method is called automatically when the `Stave` is generated. It is usually not required to be invoked manually.

(readonly) int, TotalLoadedObjects The number of `StaveObjects` loaded in the stave.

(readonly) int, EstimatedTotalObjects The estimated number of `StaveObjects` in the stave, based on its `Shorthand`.

(readonly) float, EstimatedPercentLoaded If the stave is generated asynchronously, this field is equal to the estimated percent of total `StaveObject`s that have finished loading.

(readonly) InitialSettingsContainer, InitialSettings The settings that the `Stave` started with, including the tempo, tones per octave, number of octaves, and fundamental frequency.

(readonly) string[], ProcessedShorthand An array of each individual element, including `Note`s and `SettingsBlock`s, of the `Stave`'s `Shorthand`.

(static) event System.Action<Stave>, Generated This event is invoked when the `Stave` has finished generation.

(static) event System.Action<Stave>, Finalised This event is invoked when the `Stave` has been finalised.

(readonly) bool, IsFinalised, "false" Returns `true` if the `Stave` has been finalised.

void, Finalise() Manually triggers the finalisation of the `Stave`.

void, Advance() Manually advances the `Stave`. It is not recommended to manually invoke this method.

S Instrument

The following is a detailed technical listing of public members of the `Instrument` class, ordered by their first appearance in the relevant files.

type, Name, "default value" Description.

---

enum, InstrumentType Possible values are `Left` and `Right`, which are equivalent to `OVRInput.Controller.LTouch` and `OVRInput.Controller.RTouch`, respectively.

InstrumentType, Type, "Right" The instrument type, reflective of which hand this instrument is attached to, and which `OVRInput.Controller` it will follow.

enum, PerformanceModeOption The available Performance Modes. Possible values are `Basic`, `Simple`, `Intermediate`, and `Advanced`.

PerformanceModeOption, PerformanceMode, "Simple" The current Performance Mode. Setting the Performance Mode directly using this field will change any relevant settings accordingly.

enum, IntervallicModeOption The available intervallic modes. Options are `Locked`, in which tone generation will be intervallic and automatically tuned (pitch

correct) to the closest tone according to the current environment, and `Unlocked`, in which tone generation will not automatically pitch correct.

IntervallicModeOption, IntervallicMode The current intervallic mode. The returned value is dependent on the value of `UnlockIntervallicModeButtonPressed`.

bool, MimicRecommendedNoteHeight, "true" If the Virtual Music Instrument should mimic the recommended note object height. This value is used to size the collider and effects the touch registration.

(readonly) float MinimumSizeOnMimic, "0.05f" The minimum size for instrument's collider, if `MimicRecommendedNoteHeight` equals `true`.

bool, Vibration, "true" Enables the Virtual Music Instrument to vibrate on tone generation, if set to true.

float, BPMFineAdjustment, "1f" The effect this instrument has on the current tempo, as a multiplier.

GameObject, GameObjectToFollow, "null" If set, the instrument will follow the position of this `GameObject`.

bool, MaintainVerticalCollider, "true" If set to `true`, the collider will maintain a vertical rotation, assisting touch registration.

Vector3, PositionOffset, "new Vector3(0f, 0f, 0f)" The position of the instrument is offset by this amount.

Vector3, RotationOffset, "new Vector3(0f, 0f, 0f)" The rotation of the instrument is offset by this amount.

bool, IsPrimary, "false" If set to true, this instrument is the primary instrument of its type. Only one primary instrument of each type can exist concurrently.

float, TouchBufferInSeconds, "1.5f" The amount of time in seconds that this instrument allows for an activated note not to be touched before it is deactivated.

OVRInput.Button, PrimarySynthesiserButton, "OVRInput.Button.One" The button used to activate the primary synthesiser.

(readonly) float?, PrimarySynthesiserButtonPressed, "null" If not `null`, this field equals the time that the `PrimarySynthesiserButton` was pressed. A `null` value indicates that the button is not currently pressed.

OVRInput.Button, SecondarySynthesiserButton, "OVRInput.Button.Two" The button used to activate the secondary synthesiser.

(readonly) float?, SecondarySynthesiserButtonPressed, "null" If not `null`, this field equals the time that the `SecondarySynthesiserButton` was pressed A `null`

value indicates that the button is not currently pressed.

OVRInput.Button, SustainSynthesiserButton, "OVRInput.Button.PrimaryHandTrigger" The button used to activate the sustain.

(readonly) float?, SustainSynthesiserButtonPressed, "null" If not `null`, this field equals the time that the `SustainSynthesiserButton` was pressed A `null` value indicates that the button is not currently pressed.

OVRInput.Button, UnlockIntervallicModeButton, "OVRInput.Button.PrimaryIndexTrigger" The button used to switch the value of `IntervallicMode` between `Locked` and `Unlocked`. If pressed, `IntervallicMode` is equal to `Unlocked`, and if released, `IntervallicMode` is equal to `Locked`. The default button simulates the effect of a performer pinching their fingers, indicating that they want finer control.

(readonly) float?, UnlockIntervallicModeButtonPressed, "null" If not `null`, this field equals the time that the `UnlockIntervallicModeButton` was pressed A `null` value indicates that the button is not currently pressed.

OVRInput.Button, IteratePerformanceModeButton, "OVRInput.Button.PrimaryThumbstick" The button used to iterate the `PerformanceMode`.

(readonly) int, SampleRate The sample rate of this instrument.

(const) int, MaximumTones, "4" The maximum number of tones that the instrument can produce simultaneously before clipping may occur.

bool?, OverridePitchGuide, "null" By default, the pitch guide will appear when it is recommended. Currently, the pitch guide is recommended for the Advanced Performance Mode. The default behaviour of the pitch guide will be overridden by the value in this field if it is not set to null.

float, PitchGuideLineThickness, "0.0008f" The thickness of the pitch guide's lines.

float, PitchGuideWidth, "0.25f" How far left and right the pitch guide will extend.

Vector3, PitchGuideMaskSize, "new Vector3(0.2f, 0.5f, 0.2f)" The size of the pitch guide mask. The pitch guide mask is an ellipse sized according to what is in this field.

(readonly) int, CurrentPitchDegree The current pitch degree of this instrument, relative to its current height and the conditions of the environment.

(static readonly) AudioMixerGroup, MixerGroup The current mixer this instrument is routed through, if any.

(static readonly) AudioMixerGroup, PrimaryMixerGroup The current mixer the primary synthesiser of this instrument is routed through, if any.

(static readonly) AudioMixerGroup, SecondaryMixerGroup The current mixer the secondary synthesiser of this instrument is routed through, if any.

(static) AudioMixerGroup, GetMixer(Synthesiser.Type type) A static method which returns the mixer of the provided synthesiser type, if any exists.

class, Output An `Output` is responsible for handling `Tones` generated by an instrument. It maintains the attached `AudioSource` components and data related to the production of audio.

Output, GetOutput(Synthesiser.Type type) A method which returns the current `Output` of the provided synthesiser type.

(readonly) bool, IsSustaining Returns `true` if the sustain is activated. Always returns `true` if `PerformanceMode` equals `Basic`.

(readonly) Vector3, CachedPosition The position that this instrument was last in. This is updated automatically in `Update()`.

(readonly) Vector3, CachedLocalPosition The local-space position that this instrument was last in. This is updated automatically in `Update()`.

(readonly) Quaternion, CachedRotation The last rotation of this instrument. This is updated automatically in `Update()`.

(static) Instrument, Get(InstrumentType type) A static method which returns the primary instrument of the specified type.

(static) Instrument, Get(bool dominant = true) A static method which returns the instrument relative to the performer's dominant hand by default. If a boolean is provided as a parameter and is `true`, the returned instrument is relative to the dominant hand. If a boolean is provided as a parameter and is `false`, the returned instrument is relative to the non-dominant hand.

T Environment

The following is a detailed technical listing of public members of the `Environment` class, ordered by their first appearance in the relevant files.

type, Name, "default value" Description.

(readonly) Piece, Piece, "null" The piece currently being generated or performed. Will return `null` if no piece is currently loaded, which allows for appropriate checks for `null` before retrieving any data from or modifying any piece.

float, TimeScale, "2f" The current base timescale, which impacts the length of generated notes, and the speed in which they move. Not reflective of live adjustments to tempo.

float, RealTimeScale The real time scale, including live BPM adjustments.

int, BPM, "100" The current tempo in beats per minute.

(readonly) float, BaseFrequency, "261.63f" The current fundamental frequency. This field is updated automatically according to the values set by the piece.

(readonly) int, Octaves, "2" The current number of complete octaves. This field is updated automatically according to the values set by the piece.

(readonly) int, OctaveDivision, "12" The current octave division or number of tones per octave. This field is updated automatically according to the values set by the piece.

(readonly) float[], RelativeHeights An array that contains the heights for all tones in the current range, relative to the performance area. This field is updated automatically according to the values set by the piece and performance area.

(readonly) float[], PreciseHeights An array that contains the precise local space heights in game-world units. This field is updated automatically according to the values set by the piece and performance area.

(readonly) float[], Frequencies The frequencies for all current pitch degrees. This field is updated automatically according to the values set by the piece.

(static) event System.Action, BaseFrequencyChanged This event is emitted when the value in `BaseFrequency` has changed.

(static) event System.Action, OctavesChanged This event is emitted when the value in `Octaves` has changed.

(static) event System.Action, OctaveDivision This event is emitted when the value in `OctaveDivision` has changed.

(static) event System.Action, RelativeHeightsChanged This event is emitted when the values in `RelativeHeights` have changed.

(static) event System.Action, PreciseHeightsChanged This event is emitted when the values in `PreciseHeights` have changed.

(static) event System.Action, FrequenciesChanged This event is emitted when the values in `Frequencies` have changed.

(static readonly) float[], DecimalFrequencyForTwelveToneOctave An array that contains the decimal frequencies for a equally tempered twelve-tone octave. It is provided for convenience. The values are:

```
public static readonly float[] DecimalFrequencyForTwelveToneOctave = new float[12] {
  0.000000f, // C (from C base)
  0.059463f, // C#
  0.122462f, // D
  0.189207f, // D#
  0.259921f, // E
  0.334840f, // F
  0.414214f, // F#
  0.498307f, // G
  0.587401f, // G#
  0.681793f, // A
  0.781797f, // A#
  0.887749f  // B
};
```

float, GetNoteObjectHeight Returns the current recommended height for the note objects. This method has a number of options to accommodate different pieces, performers, preferences, and situations.

Vector3, PieceStartingPosition, "new Vector3(0f, 0f, 0f)" The local-space position where pieces will start their generation.

bool, FollowPositionOfPerformer, "true" If set to `true`, staves will follow the non-rhythmic position of the performer (the axis not equivalent to time) so that they remain in playable range. If `Piece.FromDirection` is equal to `Straight`, the stave will follow the horizontal position of the performer, ensuring the performer is always in the centre. If `FromDirection` is set to either `Left` or `Right`, the distance between the performer and the stave is maintained as equal to the `HorizontalDirectionBuffer`. This behaviour can be disabled by setting `FollowPositionOfPerformer`, which defaults to `true`, to `false`.

bool, CenterPieceToPerformerOnGenerate, "true" If set to `true`, staves will move so that they are centred with the performer when they generate. This ensures that they will be in playable range when the piece begins.

float, HorizontalPositionBuffer, "0.15f" The distance between the stave and the performer when `Piece.FromDirection` is set to a horizontal direction (`Left` or `Right`).

float, NoteFinishedBuffer, "0f" The buffer before a note will trigger its finalisation procedure.

float, BPMChangeLocation, "-2f" The location of the tempo change indications relative to the centre of the stave. By default, `BPMChangeLocation` is equal to `-2f`:

two game-world units away from the centre of the stave, approximately two metres. If a piece's `FromDirection` is set to `Straight`, the indications will appear to the left of the stave. When set to `Left` or `Right`, the indications will instead by two game-world units behind the stave.

float, TranslatePreciseHeightToRelativeHeight(float preciseHeight) A method which translates the given local-space height to the equivalent relative height.

float, GetFrequencyFromPreciseHeight(float preciseHeight) A method which translates the given local-space height to the equivalent frequency.

int, GetPitchDegreeFromPreciseHeight(float preciseHeight) A method which translates the given local-space height to the equivalent pitch degree.

int, GetPitchDegreeFromRelativeHeight(float relativeHeight) A method which translates the given height relative to the performance area to the equivalent pitch degree.

int, GetFrequencyFromRelativeHeight(float relativeHeight) A method which translates the given height relative to the performance area to the equivalent frequency.

float, PitchCorrectFromRelativeHeight(float relativeHeight) A method which returns the closest height in the `RelativeHeights` array, effectively applying pitch correction.

U The PersonSpawner Script

```
using UnityEngine;

public class PersonSpawner : MonoBehaviour
{
  [SerializeField] private GameObject person;
  [SerializeField] private int amount = 300;
  [SerializeField] private Vector2 xRange = new Vector2(-100f, 100f);
  [SerializeField] private Vector2 zRange = new Vector2(3f, 50f);
  [SerializeField] private Vector2 rotationRange = new Vector2(-25f, 25f);

  void Start()
  {
    for (var i = 0; i < amount; i++)
    {
      var newPerson = Instantiate(person);
      newPerson.transform.position = new Vector3(Random.Range(xRange.x, xRange.y), -2f,
          Random.Range(zRange.x, zRange.y));
      newPerson.transform.Rotate(0, Random.Range(rotationRange.x, rotationRange.y), 0);
    }
  }
}
```

V Video Recordings of All Etudes Through Performance Tests

1. Etude 1 (v0.1.0)
2. Etude 1 (v0.1.1)
3. Etude 1 (v0.1.2)
4. Etude 2 - Variation 1 (v0.1.3)
5. Etude 2 - Variation 2 (v0.1.4)
6. Etude 3 (v0.2.0)
7. Etude 4 (v0.2.1)
8. Etude 5 (v0.3.0)
9. Etude 6 (v0.3.1)
10. Etude 7 (v0.4.0)
11. Etude 8 (v0.4.1)

W Creative Works

1. Peril (But Not Quite Doom)

2. Yurndu akanatha

X Demonstrating the Dynamic Audience Noises

8 | Pieces Composed for Virtual Music

A Étude 1

A.1 Virtual Music Shorthand Notation
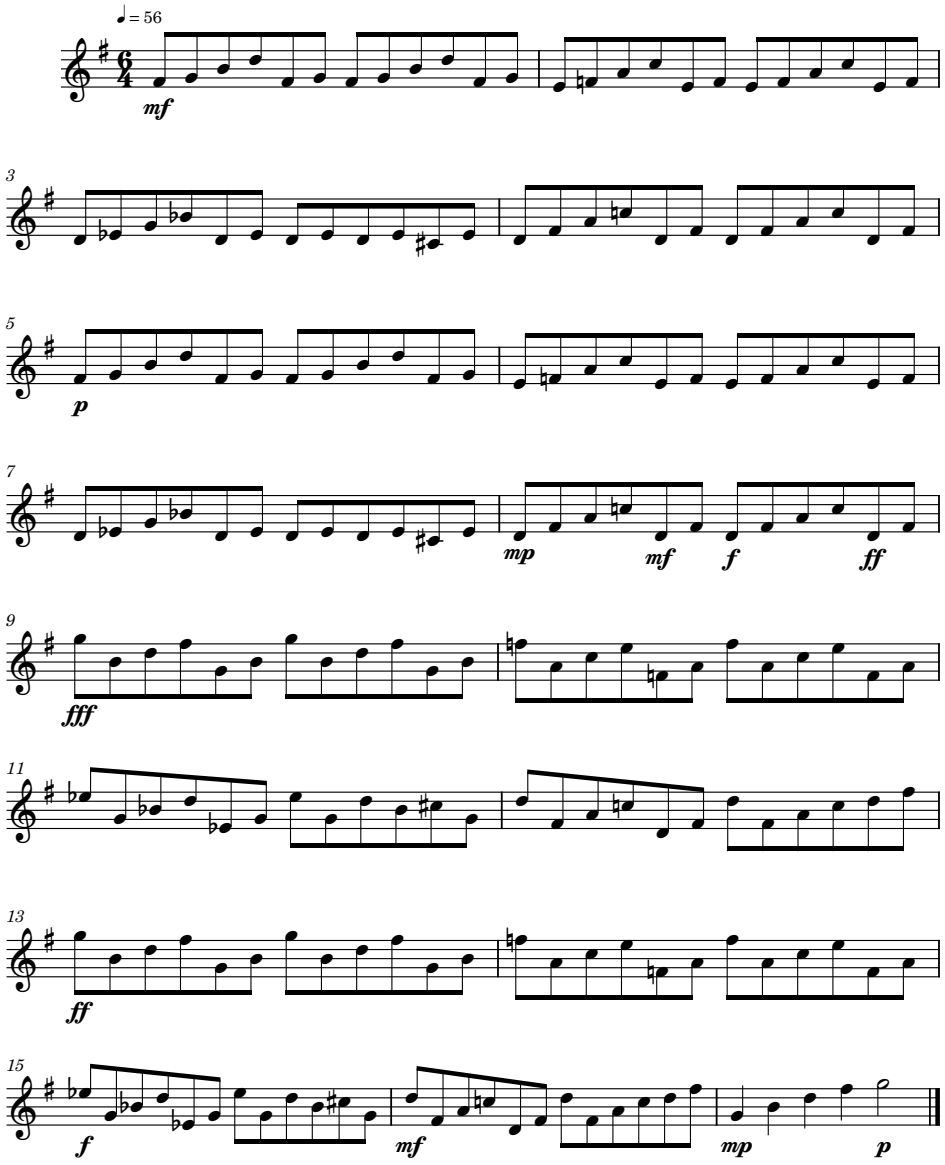
```
7q5 8q 12q 15q 7q 8q 7q 8q 12q 15q 7q 8q
5q 6q 10q 13q 5q 6q 5q 6q 10q 13q 5q 6q
3q 4q 8q 11q 3q 4q 3q 4q 3q 4q 2q 4q
3q 7q 10q 13q 3q 7q 3q 7q 10q 13q 3q 7q
7q3 8q 12q 15q 7q 8q 7q 8q 12q 15q 7q 8q
5q 6q 10q 13q 5q 6q 5q 6q 10q 13q 5q 6q
3q 4q 8q 11q 3q 4q 3q 4q 3q 4q 2q 4q
3q4 7q 10q 13q 3q5 7q 3q6 7q 10q 13q 3q7 7q
20q8 12q 15q 19q 8q 12q 20q 12q 15q 19q 8q 12q
18q 10q 13q 17q 6q 10q 18q 10q 13q 17q 6q 10q
16q 8q 11q 15q 4q 8q 16q 8q 15q 11q 14q 8q
15q 7q 10q 13q 3q 7q 15q 7q 10q 13q 15q 19q
20q7 12q 15q 19q 8q 12q 20q 12q 15q 19q 8q 12q
18q 10q 13q 17q 6q 10q 18q 10q 13q 17q 6q 10q
16q6 8q 11q 15q 4q 8q 16q 8q 15q 11q 14q 8q
15q5 7q 10q 13q 3q 7q 15q 7q 10q 13q 15q 19q
8c4 12c 15c 19c 20m3
```

A.2 Equivalent Stave Notation

Virtual Music Instrument

Étude 1

Nathan Cummins



B Étude 2 (Variation 1)

B.1 Virtual Music Shorthand Notation

```
[B=92;] 8q6 12q 15q 19q 20q 19q 15q 12q 7q 10q 15q 19q 15m
3q 7q 10q 13q 15q 13q 10q 7q 3q 8q 12q 15q 12m
4q 8q 11q 15q 16q 11q 8q 4q 6q 10q 13q 18q 13m
8q 12q 15q 19q 20q 19q 15q 12q 8q 12q 15q 19q 20c 8c

8q. 12s 15q. 19s 20q. 19s 15q. 12s 7q. 10s 15q. 19s 15m
3q. 7s 10q. 13s 15q. 13s 10q. 7s 3q. 8s 12q. 15s 12m
4q. 8s 11q. 15s 16q. 11s 8q. 4s 6q. 10s 13q. 18s 13m
8q. 12s 15q. 19s 20q. 19s 15q. 12s 8q. 12s 15q. 19s 20c 8c

8q 12c 15s 19s 20q 19c 15s 12s 7q 10c 15s 19s 15m
3q 7c 10s 13s 15q 13c 10s 7s 3q 8c 12s 15s 12m
4q 8c 11s 15s 16q 11c 8s 4s 6q 10c 13s 18s 13m
8q 12c 15s 19s 20q 19c 15s 12s 8q 12c 15s 19s 20c 8c

8s 12q 15s 19c 20s 19q 15s 12c 7s 10q 15s 19c 15m
3s 7q 10s 13c 15s 13q 10s 7c 3s 8q 12s 15c 12m
4s 8q 11s 15c 16s 11q 8s 4c 6s 10q 13s 18c 13m
8s 12q 15s 19c 20s 19q 15s 12c 8s 12q 15s 19c 20c 8c
```
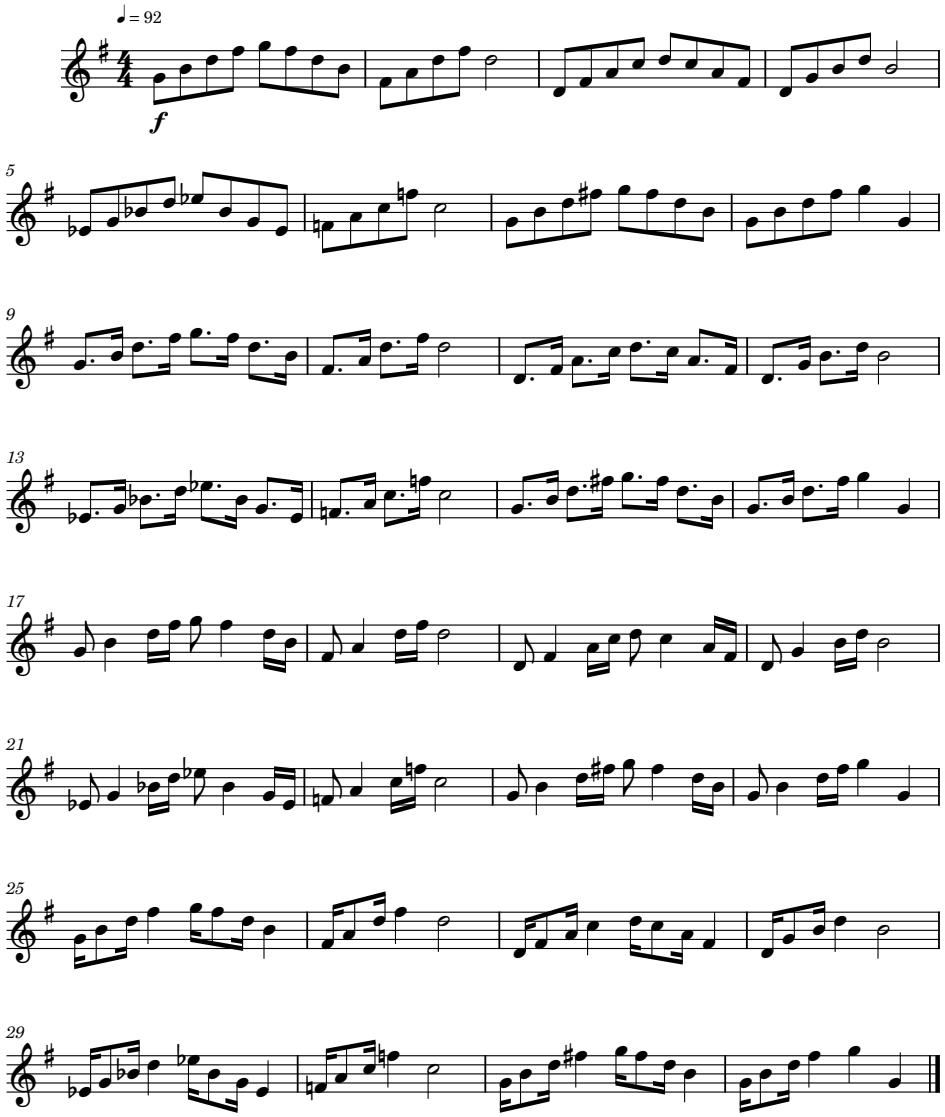
B.2 Equivalent Stave Notation

Virtual Music Instrument

Étude 2

Nathan Cummins

Variation 1



C Étude 2 (Variation 2)

C.1 Virtual Music Shorthand Notation

```
[B=92;] 8q6 12q 15q 19q 20q 19q 15q 12q 7q 10q 15q 19q 15m>2
3q 7q 10q 13q 15q 13q 10q 7q 3q 8q 12q 15q 12m<4
4q 8q 11q 15q 16q 11q 8q 4q 6q 10q 13q 18q 13m<8
8q 12q 15q 19q 20q 19q 15q 12q 8q 12q 15q 19q 20c 8c>6

8q. 12s 15q. 19s 20q. 19s 15q. 12s 7q. 10s 15q. 19s 15m>2
3q. 7s 10q. 13s 15q. 13s 10q. 7s 3q. 8s 12q. 15s 12m<4
4q. 8s 11q. 15s 16q. 11s 8q. 4s 6q. 10s 13q. 18s 13m<8
8q. 12s 15q. 19s 20q. 19s 15q. 12s 8q. 12s 15q. 19s 20c 8c>6

8q 12c 15s 19s 20q 19c 15s 12s 7q 10c 15s 19s 15m>2
3q 7c 10s 13s 15q 13c 10s 7s 3q 8c 12s 15s 12m<4
4q 8c 11s 15s 16q 11c 8s 4s 6q 10c 13s 18s 13m<8
8q 12c 15s 19s 20q 19c 15s 12s 8q 12c 15s 19s 20c 8c>6

8s 12q 15s 19c 20s 19q 15s 12c 7s 10q 15s 19c 15m<2
3s 7q 10s 13c 15s 13q 10s 7c 3s 8q 12s 15c 12m<4
4s 8q 11s 15c 16s 11q 8s 4c 6s 10q 13s 18c 13m<8
8s 12q 15s 19c 20s 19q 15s 12c 8s 12q 15s 19c 20c 8c>6

8q[3] 12q[3] 15q[3] 19c 20q[3] 19q[3] 15q[3] 12c 7q[3] 10q[3] 15q[3]
   19c 15m>2
3q[3] 7q[3] 10q[3] 13qc 15q[3] 13q[3] 10q[3] 7c 3q[3] 8q[3] 12q[3]
   15c 12m<4
4q[3] 8q[3] 11q[3] 15c 16q[3] 11q[3] 8q[3] 4c 6q[3] 10q[3] 13q[3]
   18c 13m<8
8q[3] 12q[3] 15q[3] 19c 20q[3] 19q[3] 15q[3] 12c 8q[3] 12q[3] 15q[3]
   19c 20c>6 8c
```
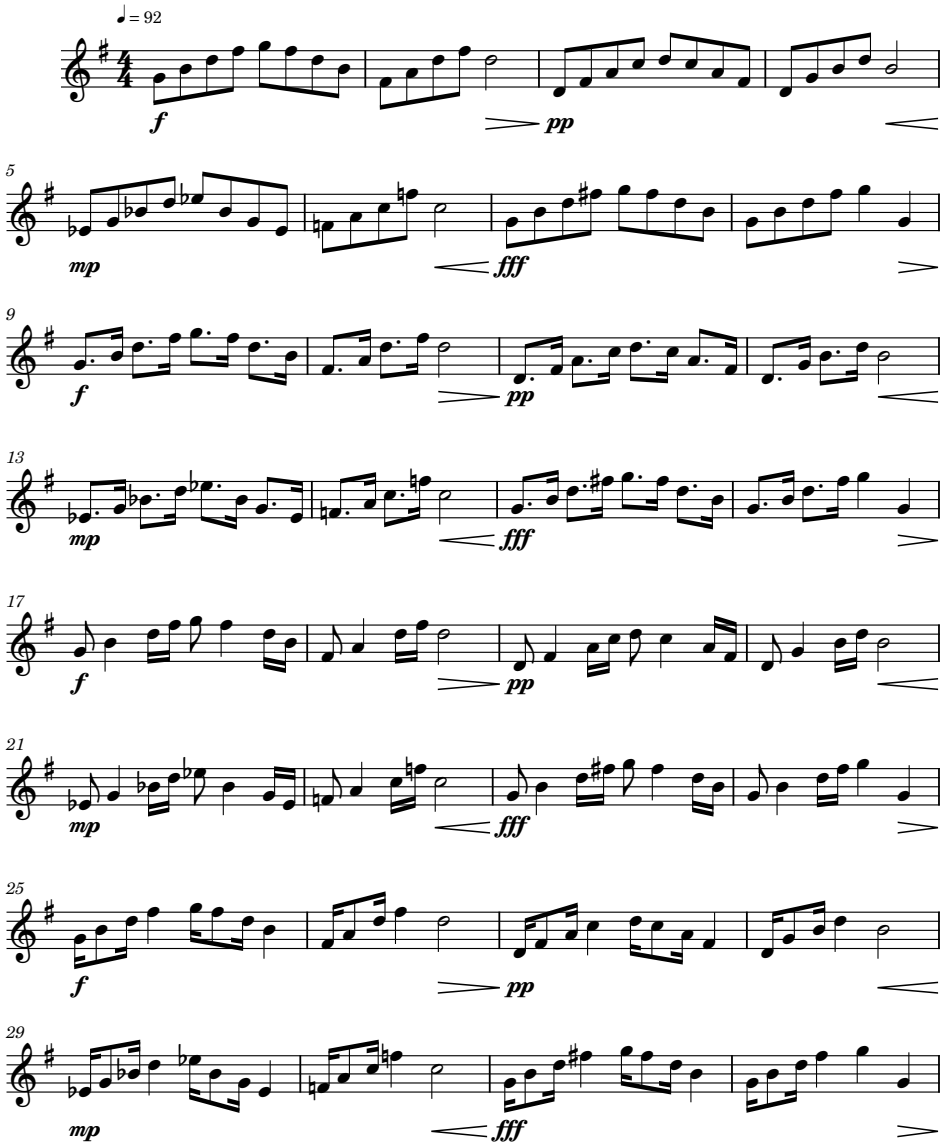
C.2 Equivalent Stave Notation

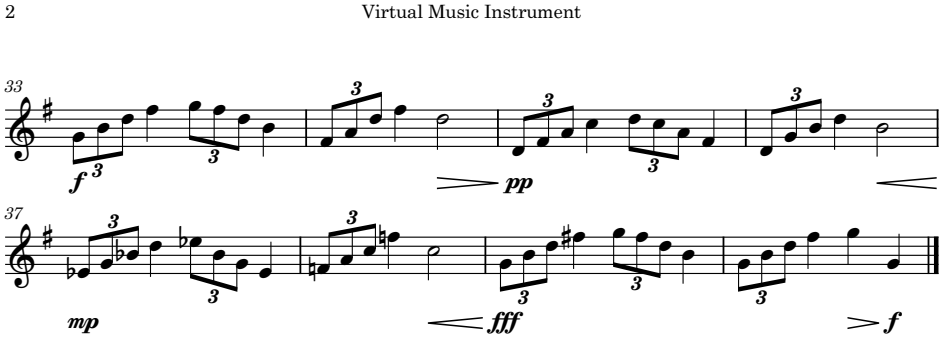
Virtual Music Instrument

Étude 2

Nathan Cummins

Variation 2




2
Virtual Music Instrument
33
f
pp
37
mp
fff
f

D Étude 3

D.1 Virtual Music Shorthand Notation

```
[B=52,Sweetly;] 7q5{20vv} 8q 12q 15q 7q 8q{15} 7q{20} 8q 12q 15q 7q
   8q{13vv}
5q{18vv} 6q 10q 13q 5q 6q{13} 5q{18} 6q 10q 13q 5q 6q{11vv}
3q{16vv} 4q 8q 11q 3q 4q{11} 3q{16} 4q 3q 4q 2q 4q{10vv}
3q{15vv} 7q 10q 13q 3q 7q{10} 3q{15} 7q 10q 13q 3q 7q>3{15vv}

7q3{20vv} 8q 12q 15q 7q 8q{15} 7q{20} 8q 12q 15q 7q 8q{13vv}
5q{18vv} 6q 10q 13q 5q 6q{13} 5q{18} 6q 10q 13q 5q 6q{11vv}
3q{16vv} 4q 8q 11q 3q 4q{11} 3q{16} 4q 3q 4q 2q 4q<4{10vv}
3q4{15vv} 7q 10q 13q<5 3q5 7q<6{10} 3q6{15} 7q 10q 13q<7 3q7 7q<8{15}

20q8{12} 12q 15q 19q{15} 8q 12q 20q{12} 12q 15q 19q{15} 8q 12q
18q{10} 10q 13q 17q{13} 6q 10q 18q{10} 10q 13q 17q{13} 6q 10q
16q{8} 8q 11q 15q{11} 4q 8q 16q{8} 8q 15q 11q 14q 8q
15q{7} 7q 10q 13q{10} 3q 7q 15q{7} 7q 10q 13q 15q 19q>7{15}

20q7{12} 12q 15q 19q{15} 8q 12q 20q{12} 12q 15q 19q{15} 8q 12q
18q{10} 10q 13q 17q{13} 6q 10q 18q{10} 10q 13q 17q{13} 6q 10q>6
16q6{8} 8q 11q 15q{11} 4q 8q 16q{8} 8q 15q 11q 14q 8q>5
15q5{7} 7q 10q 13q{10} 3q 7q 15q{7} 7q 10q 13q 15q 19q>4{15}
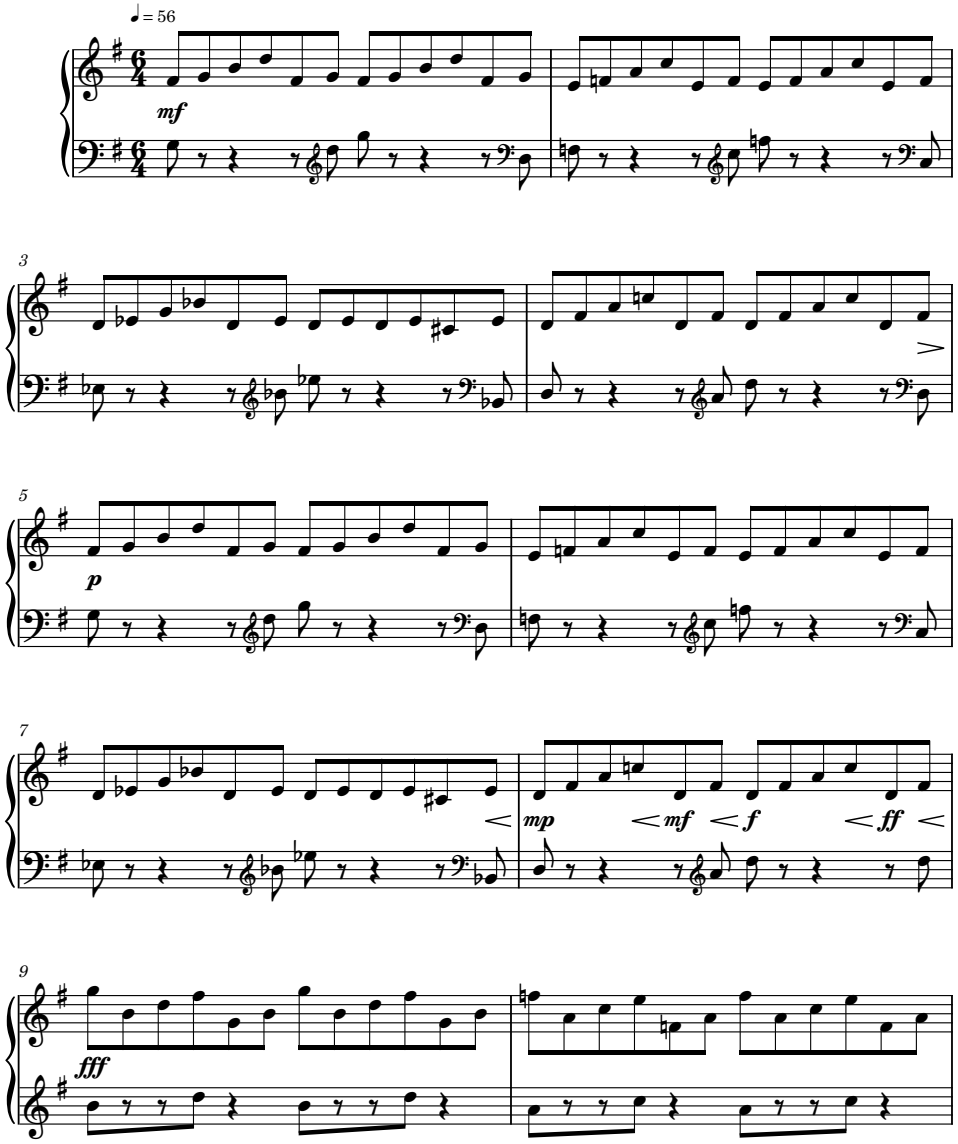
[B=ritenuto;] 8c4{3} 12c{3} 15c{8} 19c>3{13} 20m3{12}
```
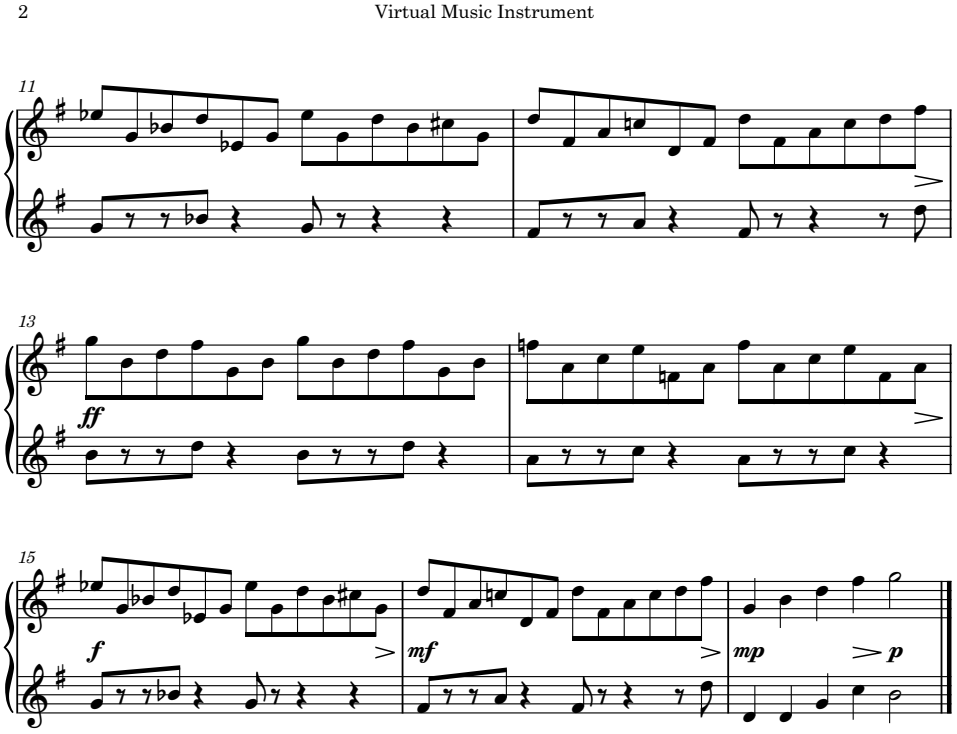
D.2 Equivalent Stave Notation

Virtual Music Instrument

Étude 3

Nathan Cummins




2
Virtual Music Instrument
11
13
ff
15
f
mf
mp
p

E Étude 4

E.1 Virtual Music Shorthand Notation

```
[F=130.81;B=68;]
0q3{8vb} 0q{3c.-m} 0q{10c-m} 12q-m
0q{6vb} 0q{1c.-m} 0q{10c-m} 12q-m
0q{8vb} 0q{3c.-m} 0q{10c-m} 12q-m
0q{6vb} 0q{1c.-m} 0q{10c-m} 12q-m
0q{4vb} 0q{11c.-m} 0q{8c-m} 10q-m
0q{3vb} 0q{10c.-m} 8q 7q-m

0b

[B=60;] 0q2{8vb} 0q{3c.-m} 0q{10c-m} 12q-m
0q{6vb} 0q{1c.-m} 0q{10c-m} 12q-m
0q{8vb} 0q{3c.-m} 0q{10c-m} 12q-m
0q{6vb} 0q{1c.-m} 0q{10c-m} 12q-m

0q4{8vb2} 0q{3c.-m2} !12q !13q !15m
0q{6vb2} 0q{1c.-m2} !12q !20q !18m
0q{8vb2} 0q{3c.-m2} !12q !13q !15q !12q !13q !20q
6vq2{!18b4} 0q{1c.-m} 0q{10c-m} 12q-m
0q2{!15m-c.4} 0q{11c.-m} 0q{8c-m} 0m{10q-m} !16s4 !15s
0q2{!13b4} 0q{3c.-m} 8q{10c-m} 7q-m

0b

[B=68;] 0q3{8vb} 0q{3c.-m} 0q{10c-m} 12q-m
0q{6vb} 0q{1c.-m} 0q{10c-m} 12q-m
0q{8vb} 0q{3c.-m} 0q{10c-m} 12q-m
0q{6vb} 0q{1c.-m} 0q{10c-m} 12q-m
0q{4vb} 0q{11c.-m} 0q{8c-m} 10q-m
0q{3vb} 0q{10c.-m} 8q 7q-m

0b

[B=50;] 0q3{8vb} 0q{3c.-m} 0q{10c-m} [B=30;] 12q-m
```

F Étude 5

F.1 Virtual Music Shorthand Notation

```
[B=80;] .6vvc1("w") 0c 0m 0m 0c .1vvc .6vvc("w") .9vvc("r"){.13vv} 0m 0m 0c >1vvc
.6vvc("w") .9vvc("r"){.13vv} .1vvc("w") >.9vvc("r"){>.13vv} 0m 0c >1vvc .6vvc("w") .9vvc("r"){.13vv} .1vvc("w") >.9vvc("r"){>.13vv} 0b
.6vvc("w") .9vvc("r"){.13vv} .1vvc("w") .9vvc("r"){.13vv} [B=88,accel;] .6vvc("w") .9vvc("r"){.13vv} .1vvc("w") .9vvc("r"){.13vv} [B=96,,n;] .6vvc("w") .9vvc("r"){.13vv} .1vvc("w") .9vvc("r"){.13vv} [B=102,,n;] .6vvc2("w") .9vvc3("r"){.13vv} .1vvc4("w") .9vvc5("r"){.13vv}

[B=112,Fast;] .6q6("w") .9q("r"){.13q} .1q("w") .9q("r"){.13q} .6q("w") .9q("r"){.13q} .1q("w") .9q("r"){.13q} .6q("w") .9q("r"){.13q} .1q("w") .9q("r"){.13q} >.5q("w") .8q("r"){.11q} >.1q("w") .8q("r"){.11q}
.6q3("w") .9q("r"){.13q} .1q("w") .9q("r"){.13q} .6q("w") .9q("r"){.13q} .1q("w") .9q("r"){.13q} .6q("w") .9q("r"){.13q} .1q("w") .9q("r"){.13q} >.5q("w") .8q("r"){.11q} >.1q("w") .8q("r"){.11q}

.6q("w") 0s .8s .9q .6q .13q >14c .13q .6q("r") 0s .8s .9q .6q .5q .8q >.1c
.6q("w") 0s .8s .9q .6q .13q >14c .13q .13s .13s .14q .15q .12q .13q .8q >.13s("r") .11s .9s .8s
.6q("w") 0s .8s .9q .6q .13q >14c .13q .6q("r") 0s .8s .9q .6q .5q .8q >.1c
.6q("w") 0s .8s .9q .6q .13q >14c .13q 13b<7

.6vq("ws") .9q("rs"){.13q} .1vq("ws") .9q("rs"){.13q} .6vq("ws") .9q("rs"){.13q} .1vq("ws") .9q("rs"){.13q} .6vq("ws") .9q("rs"){.13q} .1vq("ws") .9q("rs"){.13q} >.5vq("ws") .8q("rs"){.11q} >.1vq("ws") .8q("rs"){.11q}
.6vq2("w") .9q("r"){.13q} .1vq("w") .9q("r"){.13q} .6vq("w") .9q("r"){.13q} .1vq("w") .9q("r"){.13q} .6vq3("w") .9q("r"){.13q} .1vq("w") .9q("r"){.13q} >.5vq3("w") .8q("r"){.11q} >.1vq4("w") .8q("r"){.11q}

.6vvq6("w") 0s .8s .9q .6q .13q >14c .1vvq("r") .6vvq("w") 0s .8s .9q .6q .5q .8q >.1q >.1vvq("r")
>.6vvq("w") 0s .8s .9q .6q .13q >14c .13q .13s .13s .14q .15q .12q
```

```
    .13q .8q >.13vvs("r") .11vvs .9vvs .8vvs
.6vvq("w") 0s .8s .9q .6q .13q >14c .1vvq("r") .6vvq("w") 0s .8s .9q
    .6q .5q .8q >.1q >.1vvq("r")
>.6vvq("w") 0s .8s .9q .6q .13q >14c .13q 17b3<6

.14s3("w") .13s .14s .13s .14q("r") .11q .14s("w") .13s .14s .13s
    .14q("r") .11q
.13s("w") .11s .13s .11s .13q("r") .9q .13s("w") .11s .13s .11s
    .13q("r") .9q
.13s("w") .11s .13s .11s .13q("r") .8q .13s("w") .11s .13s .11s
    .13q("r") .8q
.13s("w") .11s .13s .11s .13q("r") .9q .13s("w") .11s .13s .11s
    .13q("r") .9q
.14s("w") .13s .14s .13s .14q("r") .11q .14s("w") .13s .14s .13s
    .14q("r") .11q
.13s("w") .11s .13s .11s .13q("r") .9q .13s("w") .11s .13s .11s
    .13q("r") .9q
.13s("w") .11s .13s .11s .13q("r") .8q .13s("w") .11s .13s .11s
    .13q("r") .8q
.13s("w") .11s .13s .11s .13q("r") .8q .1vvs7("w") .3vvs .5vvs .6vvs
    .8vvq .1vvq("r")
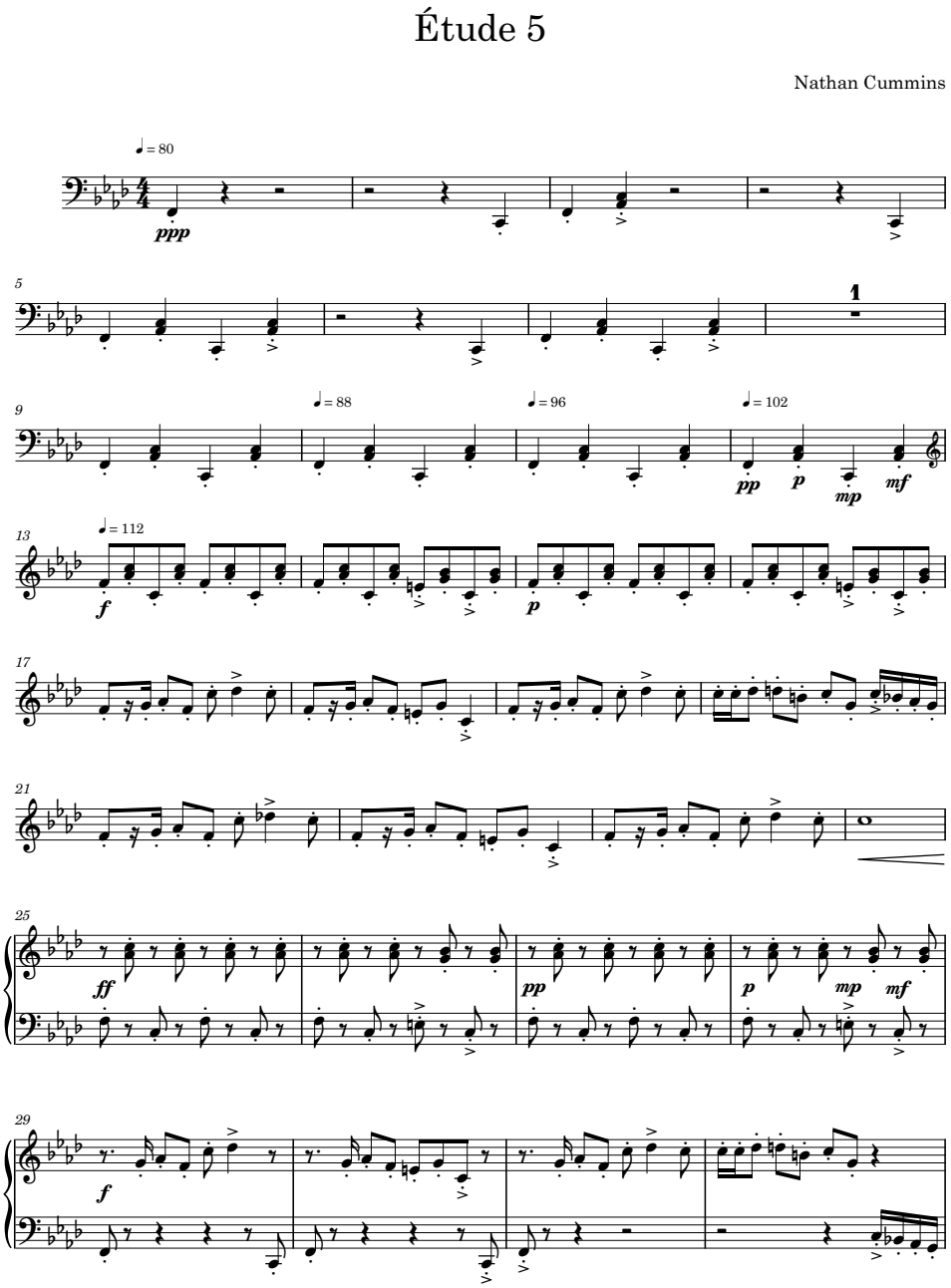
.6vq("ws") .9q("rs"){.13q} .1vq("ws") .9q("rs"){.13q} .6vq("ws")
    .9q("rs"){.13q} .1vq("ws") .9q("rs"){.13q} .6vq("ws")
    .9q("rs"){.13q} .1vq("ws") .9q("rs"){.13q} >.5vq("ws")
    .8q("rs"){.11q} >.1vq("ws") .8q("rs"){.11q}
.6vq2("w") .9q("r"){.13q} .1vq("w") .9q("r"){.13q} .6vq("w")
    .9q("r"){.13q} .1vq("w") .9q("r"){.13q} .6vq3("w") .9q("r"){.13q}
    .1vq("w") .9q("r"){.13q} >.5vq3("w") .8q("r"){.11q} >.1vq4("w")
    .8q("r"){.11q}
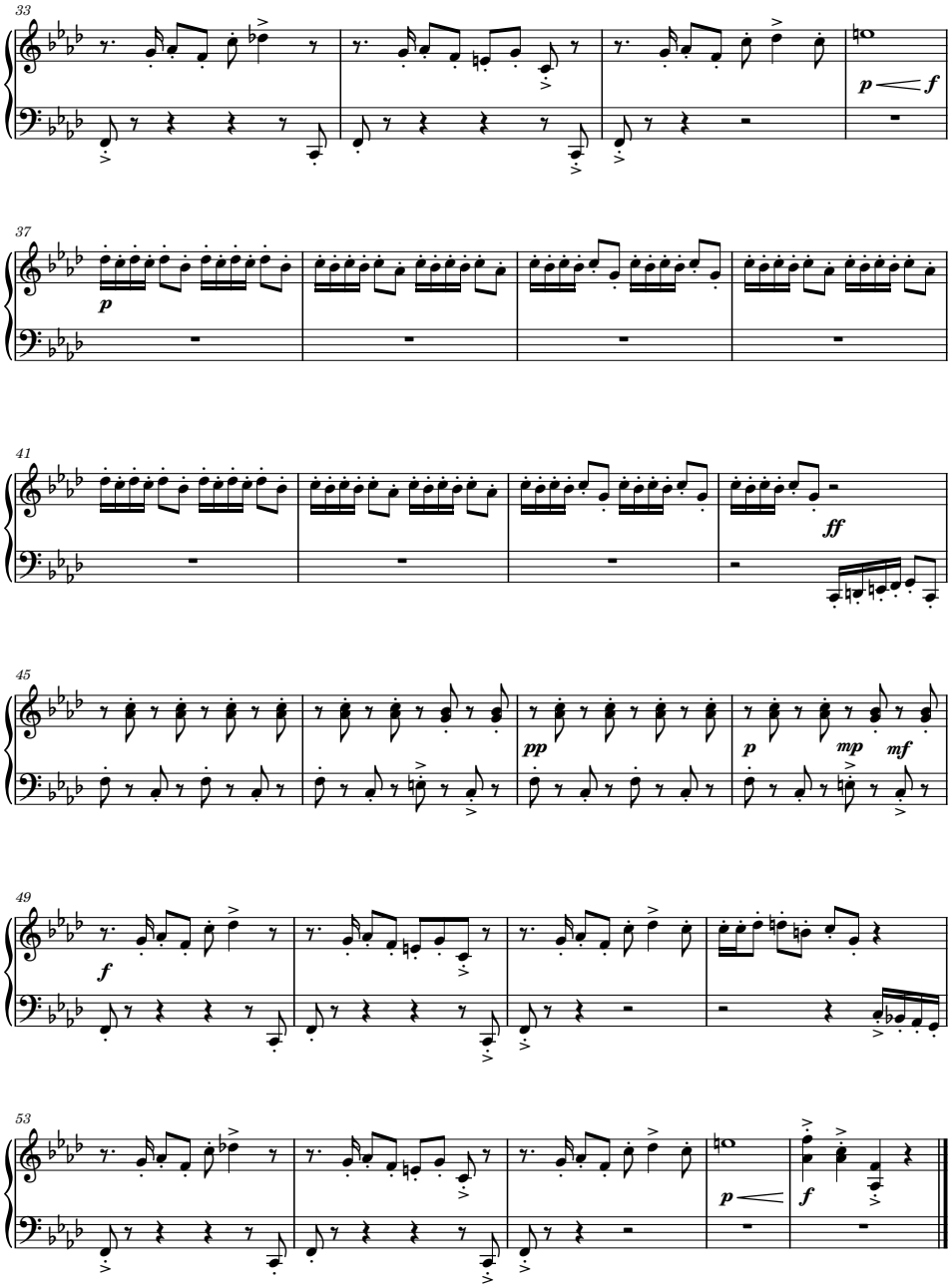
.6vvq6("w") 0s .8s .9q .6q .13q >14c .1vvq("r") .6vvq("w") 0s .8s
    .9q .6q .5q .8q >.1q >.1vvq("r")
>.6vvq("w") 0s .8s .9q .6q .13q >14c .13q .13s .13s .14q .15q .12q
    .13q .8q >.13vvs("r") .11vvs .9vvs .8vvs
.6vvq("w") 0s .8s .9q .6q .13q >14c .1vvq("r") .6vvq("w") 0s .8s .9q
    .6q .5q .8q >.1q >.1vvq("r")
>.6vvq("w") 0s .8s .9q .6q .13q >14c .13q 17b3<6

>.18c("w"){>.9} >.13c("r"){>.9} >.6c("w","r"){>.9}
```

F.2 Equivalent Stave Notation

Virtual Music Instrument




33
p
f
37
p
41
ff
45
pp
p
mp
mf
49
f
53
p
f

G Étude 6

G.1 Virtual Music Shorthand Notation

```
[B=78;] 13q3 13^q 13q 13^q 13q 13^q 13q 13^q
13q 13^q 13q 13^q 13q 13^q 13q 13^q

13q{5b("left")} 13^q 13q 13^q 13q 13^q 13q 13^q
13q{6b("right")} 13^q 13q 13^q 13q 13^q 13q 13^q
13q{5b("forward")} 13^q 13q 13^q 13q 13^q 13q 13^q
13q{3b("left","right","forward")} 13^q 13q 13^q [B=ritenuto;] 12q
   [B=72,,n;] 12^q [B=66,,n;] 12q [B=62,,n;] 12^q

[B=60,,n;] 8vb>1("change") 0b

[B=78,A_tempo;F=130.81;] 0q4{1b} 0q{8c.-m} 15q 13q 17m
0q{9vb} 0q{4c.-m} 15q 13q 16m
0q{6vb} 0q{1c.-m} 15q 13q 16m
0q{8vb} 0q{3c.-m} 15q 13q 12m

[B=86,Faster;] 0q5{1b} 0q{8c.-m} 15q 13q 17m<6
0q{9vb} 0q{4c.-m} 15q 13q 16m<7
0q{6vb} 0q{1c.-m} 15q 13q 16m<8
[B=ritenuto;] 0q{8vb>1} 0q{3c.-m>1} [B=72,,n;] 15q [B=66,,n;] 13q
   [B=62,,n;] 12m>1

[B=78,A_tempo;F=130.81;] 13q2{1vc("left")} 13^q 13q{8vc("right")}
   13^q 13q{5m("forward")} 13^q 13q 13^q
13q2{>9vvc("left")} 13^q 13q{>4vc("right")} 13^q 13q{>1m("forward")}
   13^q 13q 13^q
13q2{>6vvc("left")} 13^q 13q{>1vc("right")} 13^q
   13q{>10vm("forward")} 13^q 13q 13^q
13q2{>8vvc("left")} 13^q 13q{>3vc("right")} 13^q
   15q{>12vm("forward")} 15^q 15q 15^q

17q2{>1vc("left")} 17^q 17q{>8vc("right")} 17^q 17q{>5m("forward")}
   17^q 17q 17^q
16q2{>9vvc("left")} 16^q 16q{>4vc("right")} 16^q 16q{>1m("forward")}
   16^q 16q 16^q
18q2{>6vvc("left")} 18^q 18q{>1vc("right")} 18^q
   18q{>10vm("forward")} 18^q 18q 18^q
[B=ritenuto;] 20q2{>8vvc("left")} [B=72,,n;] 20^q 20q{>3vc("right")}
   [B=66,,n;] 20^q 20q{>12vm("left","right","forward")} [B=62,,n;]
   20^q [B=58,,n;] 20q [B=54,,n;] 20^q
```

```
30
31 0b
32
33 [B=60,Slower;] 0q2{1b-b("left")} 0q{8c.-b.} 15q("right") 13q
      [B=54,,n;] 0m{17m-b("forward")} 25b("left","right","forward")
```

H Étude 7

H.1 Virtual Music Shorthand Notation

```
[B=128;F=65.41;] .6q2 .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q
.6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .3q .3q .4q .4q .5q .5q

.6q{!22b-b} .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q
.6q{!23b-b} .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q
.6q{!20b-b} .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q
.6q{!22b-b} .6q .6q .6q .6q .6q .6q .6q .6q .6q .3q .3q .4q .4q .5q .5q
.6q{!22^b-b} .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q
.6q{!23^b-b} .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q
.6q{!20^b} .6q .6q .6q .6q .6q .6q .6q .6q{!23^b} .6q .6q .6q .6q .6q .6q .6q
.6q{!22^b-b} .6q .6q .6q .6q .6q .6q .6q .6q .6q .3q .3q .4q .4q .5q .5q

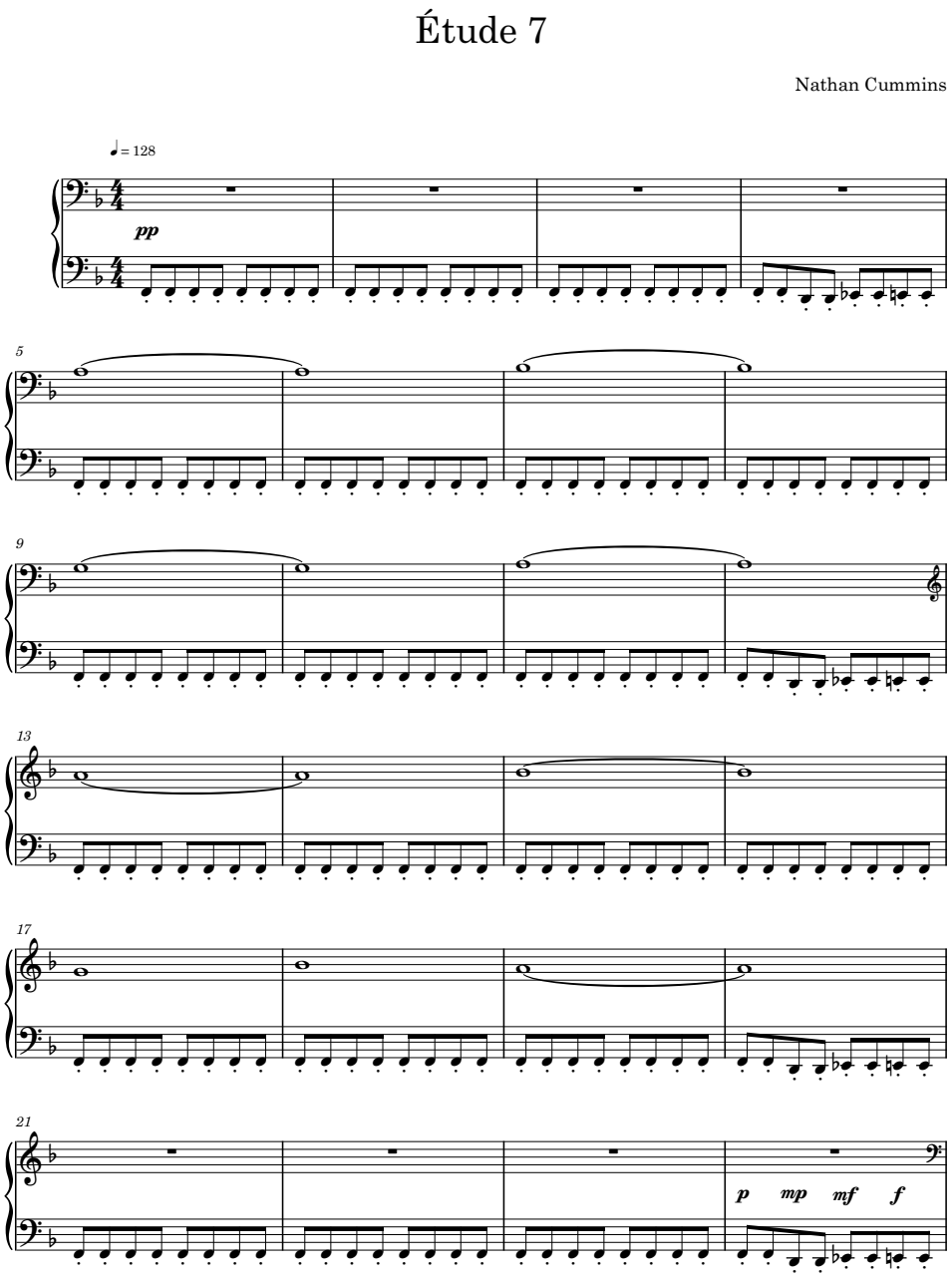
.6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q
.6q .6q .6q .6q .6q .6q .6q .6q .6q3 .6q .3q4 .3q .4q5 .4q .5q6 .5q
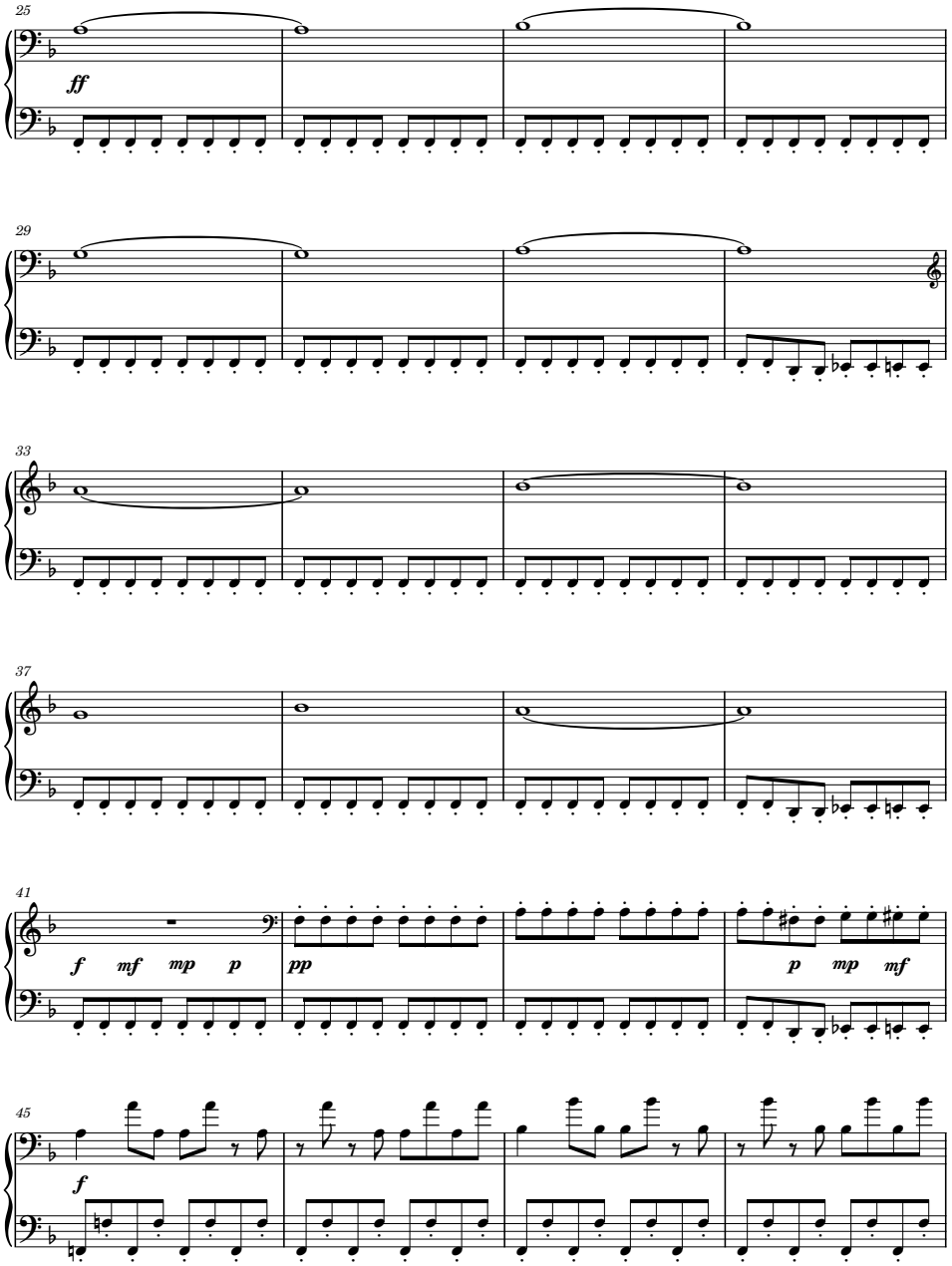
.6q7{!22b-b} .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q
.6q{!23b-b} .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q
.6q{!20b-b} .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q
.6q{!22b-b} .6q .6q .6q .6q .6q .6q .6q .6q .6q .3q .3q .4q .4q .5q .5q
.6q{!22^b-b} .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q
.6q{!23^b-b} .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q .6q
.6q{!20^b} .6q .6q .6q .6q .6q .6q .6q .6q{!23^b} .6q .6q .6q .6q .6q .6q .6q
.6q{!22^b-b} .6q .6q .6q .6q .6q .6q .6q .6q .6q .3q .3q .4q .4q .5q .5q
```
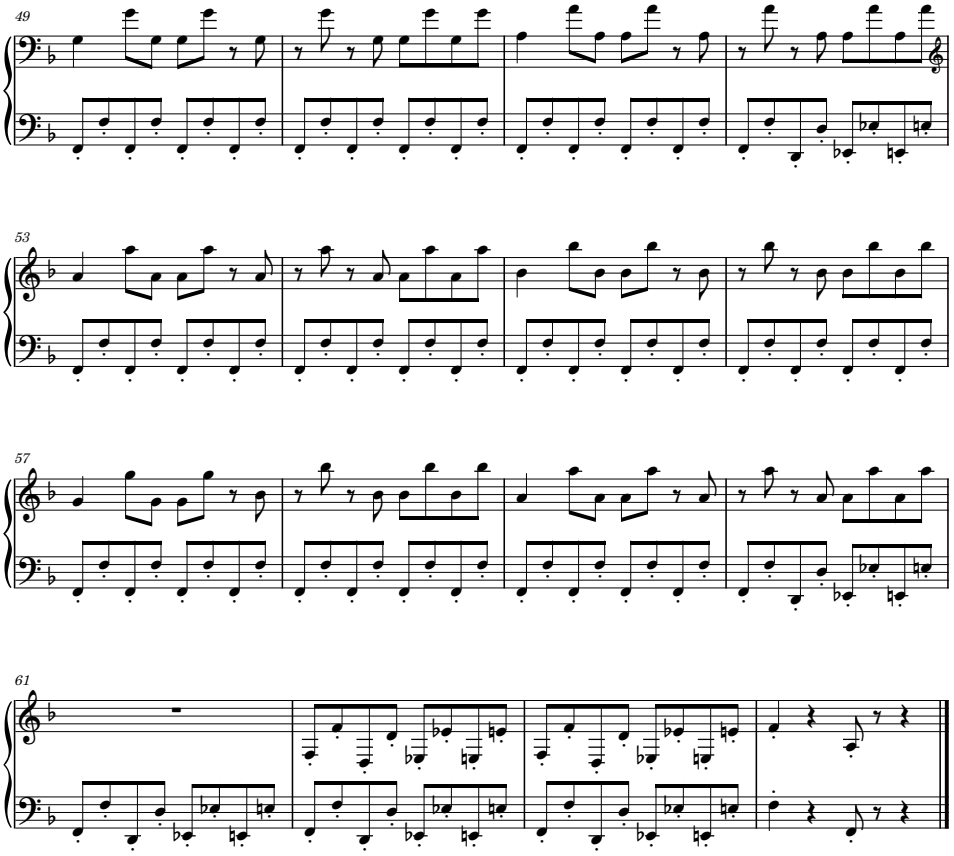
```
24
25 .6q6 .6q .6q5 .6q .6q4 .6q .6q3 .6q .6q2{.18} .6q{.18} .6q{.18}
      .6q{.18} .6q{.18} .6q{.18} .6q{.18} .6q{.18}
26 .6q{.22} .6q{.22} .6q{.22} .6q{.22} .6q{.22} .6q{.22} .6q{.22}
      .6q{.22} .6q{.22} .6q{.22} .3q3{.19} .3q{.19} .4q4{.20} .4q{.20}
      .5q5{.21} .5q{.21}
27
28 .6q6{!22c} .6^q .6q{!22^} .6^q{!22} .6q{!22} .6^q{!22^} .6q
      .6^q{!22} .6q .6^q{!22^} .6q .6^q{!22} .6q{!22} .6^q{!22^}
      .6q{!22} .6^q{!22^}
29 .6q{!23c} .6^q .6q{!23^} .6^q{!23} .6q{!23} .6^q{23^} .6q .6^q{!23}
      .6q .6^q{!23^} .6q .6^q{!23} .6q{!23} .6^q{!23^} .6q{!23}
      .6^q{!23^}
30 .6q{!20c} .6^q .6q{!20^} .6^q{!20} .6q{!20} .6^q{20^} .6q .6^q{!20}
      .6q .6^q{!20^} .6q .6^q{!20} .6q{!20} .6^q{!20^} .6q{!20}
      .6^q{!20^}
31 .6q{!22c} .6^q .6q{!22^} .6^q{!22} .6q{!22} .6^q{22^} .6q .6^q{!22}
      .6q .6^q{!22^} .3q .3^q{!22} .4q{!22} .4^q{!22^} .5q{!22}
      .5^q{!22^}
32 .6q{!22c} .6^q .6q{!22^} .6^q{!22} .6q{!22} .6^q{22^} .6q .6^q{!22}
      .6q .6^q{!22^} .6q .6^q{!22} .6q{!22} .6^q{!22^} .6q{!22}
      .6^q{!22^}
33 .6q{!23c} .6^q .6q{!23^} .6^q{!23} .6q{!23} .6^q{23^} .6q .6^q{!23}
      .6q .6^q{!23^} .6q .6^q{!23} .6q{!23} .6^q{!23^} .6q{!23}
      .6^q{!23^}
34 .6q{!20c} .6^q .6q{!20^} .6^q{!20} .6q{!20} .6^q{20^} .6q .6^q{!23}
      .6q .6^q{!23^} .6q .6^q{!23} .6q{!23} .6^q{!23^} .6q{!23}
      .6^q{!23^}
35 .6q{!22c} .6^q .6q{!22^} .6^q{!22} .6q{!22} .6^q{22^} .6q .6^q{!22}
      .6q .6^q{!22^} .3q .3^q{!22} .4q{!22} .4^q{!22^} .5q{!22}
      .5^q{!22^}
36
37 .6q .6^q .3q .3^q .4q .4^q .5q .5^q
38 .6q{.18} .6^q{.18^} .3q{.15} .3^q{.15^} .4q{.16} .4^q{16^} .5q{.17}
      .5^q{.17^}
39 .6q{.18} .6^q{.18^} .3q{.15} .3^q{.15^} .4q{.16} .4^q{16^} .5q{.17}
      .5^q{.17^}
40 .6^c{.18^} 0c .6q{.22}
```

H.2 Equivalent Stave Notation

Virtual Music Instrument




25
ff
29
33
37
41
f mf mp p
pp
p mp mf
45
f


49
53
57
61

I Étude 8

I.1 Virtual Music Shorthand Notation

```
[B=132;]
>.3vq7{!>.15} .3vq3 .3vq .3vq .3vq .3vq .3vq .3vq >.3vq{!>.14} .3vq .3vq >.3vq{!>.14} .3vq .3vq >.2vq{!>.10} .2vq
.3vq .3vq .3vq .3vq .3vq .3vq .3vq .3vq .3vq{!.11} .3vq .3vq >.3vq{!>.14} .3vq .3vq >10vvc<7{!16}

>.3vq7{!>.15} .3vq3 .3vq .3vq .3vq .3vq .3vq .3vq >.3vq{!>.14} .3vq .3vq >.3vq{!>.14} .3vq .3vq >.2vq{!>.10} .2vq
.3vq .3vq .3vq .3vq .3vq .3vq .3vq .3vq .3vq .3vq .3vq{!.11} .3vq{!.10} .3vq{!.8} .3vq .10vq{!4c} .10vq

.3vq{!7b>1} .3vq .3vq .3vq .3vq .3vq .3vq .3vq .8vvq .8vvq .8vvq .8vvq .8vvq .8vvq .8vvq .8vvq
.10vvq .10vvq .10vvq .10vvq .10vvq .10vvq .10vvq .10vvq .11vvq .11vvq .11vvq .11vvq .10vvq .10vvq .10vvq .10vvq("2")

.3vq{!15b-b} .3vq .3vq .3vq .3vq .3vq .3vq .3vq .8vvq .8vvq .8vvq .8vvq .8vvq .8vvq .8vvq .8vvq
.10vvq{!22b-b<6} .10vvq .10vvq .10vvq .10vvq .10vvq .10vvq .10vvq .11vvq .11vvq .11vvq .11vvq .10vvq4 .10vvq .10vvq5 .10vvq

>.3vvq6{>.15^q} 0q 0q >.3vvq 0c >.3vvq 0q 0q >.3vvq 0c .10vvq 0q >10vvc
>.3vvq 0q 0q >.3vvq 0c >.3vvq 0q 0q >.3vvq 0c .10vvvq .17vvvq .22vvvq .17vvvq
>.3vvq{>.15^q} 0q 0q >.3vvq 0c >.3vvq 0q 0q >.3vvq 0c .10vvq 0q >10vvc
>.10vvvq 0q 0q >.10vvvq 0c >.10vvvq 0q 0q >.10vvq 11vvq 10vvq 8vvc 6vvc>5

.8vvq .8vvq .8vvq{!3c} .8vvq .8vvq{!8c} .8vvq .8vvq{!11c} .8vvq .10vvq{!14m.} .10vvq .10vvq .10vvq .10vvq .10vvq .10vvq{!14q} .10vvq{!15q}
.11vvq{!17c.} .11vvq .11vvq .11vvq{!15q-c} .11vvq .11vvq .11vvq{!14c} .11vvq .13vvq{!17c.} .13vvq .13vvq .13vvq{!15q-m} .13vvq .13vvq .13vvq .13vvq
.8vvq .8vvq .8vvq{!3c} .8vvq .8vvq{!8c} .8vvq .8vvq{!11c} .8vvq .10vvq{!14m.} .10vvq .10vvq .10vvq .10vvq .10vvq .10vvq{!14q} .10vvq{!15q}
.11vvq{!17c.} .11vvq .11vvq .11vvq{!15q-c} .11vvq .11vvq
```

```
   .11vvq{!14c} .11vvq .10vvq3{!14^b3<7} .10vvq .10vvq4 .10vvq
   .10vvvq5 .10vvvq .10vvvq6 .10vvvq

>.3vq7{>.15^} .3vq3("1") .3vq .3vq .3vq .3vq .3vq .3vq >.3vq{!>.14}
   .3vq .3vq >.3vq{!>.14} .3vq .3vq >.2vq{!>.10} .2vq
.3vq .3vq .3vq .3vq .3vq .3vq .3vq .3vq .3vq{!.11} .3vq .3vq
   >.3vq{!>.14} .3vq .3vq >10vvc<7{!16}

>.3vq7{!>.15} .3vq3 .3vq .3vq .3vq .3vq .3vq .3vq >.3vq{!>.14} .3vq
   .3vq >.3vq{!>.14} .3vq .3vq >.2vq{!>.10} .2vq
.3vq .3vq .3vq .3vq .3vq .3vq .3vq .3vq .3vq .3vq .3vq{!.11}
   .3vq{!.10} .3vq{!.8} .3vq .10vq{!4c} .10vq

.3vq{!7b>1} .3vq .3vq .3vq .3vq .3vq .3vq .3vq .8vvq .8vvq .8vvq
   .8vvq .8vvq .8vvq .8vvq .8vvq
.10vvq .10vvq .10vvq .10vvq .10vvq .10vvq .10vvq .10vvq .11vvq
   .11vvq .11vvq .11vvq .10vvq .10vvq .10vvq .10vvq("2")

.3vq{!15b-b} .3vq .3vq .3vq .3vq .3vq .3vq .3vq .8vvq .8vvq .8vvq
   .8vvq .8vvq .8vvq .8vvq .8vvq
.10vvq{!22b-b<6} .10vvq .10vvq .10vvq .10vvq .10vvq .10vvq .10vvq
   .11vvq .11vvq .11vvq .11vvq .10vvq4 .10vvq .10vvq5 .10vvq
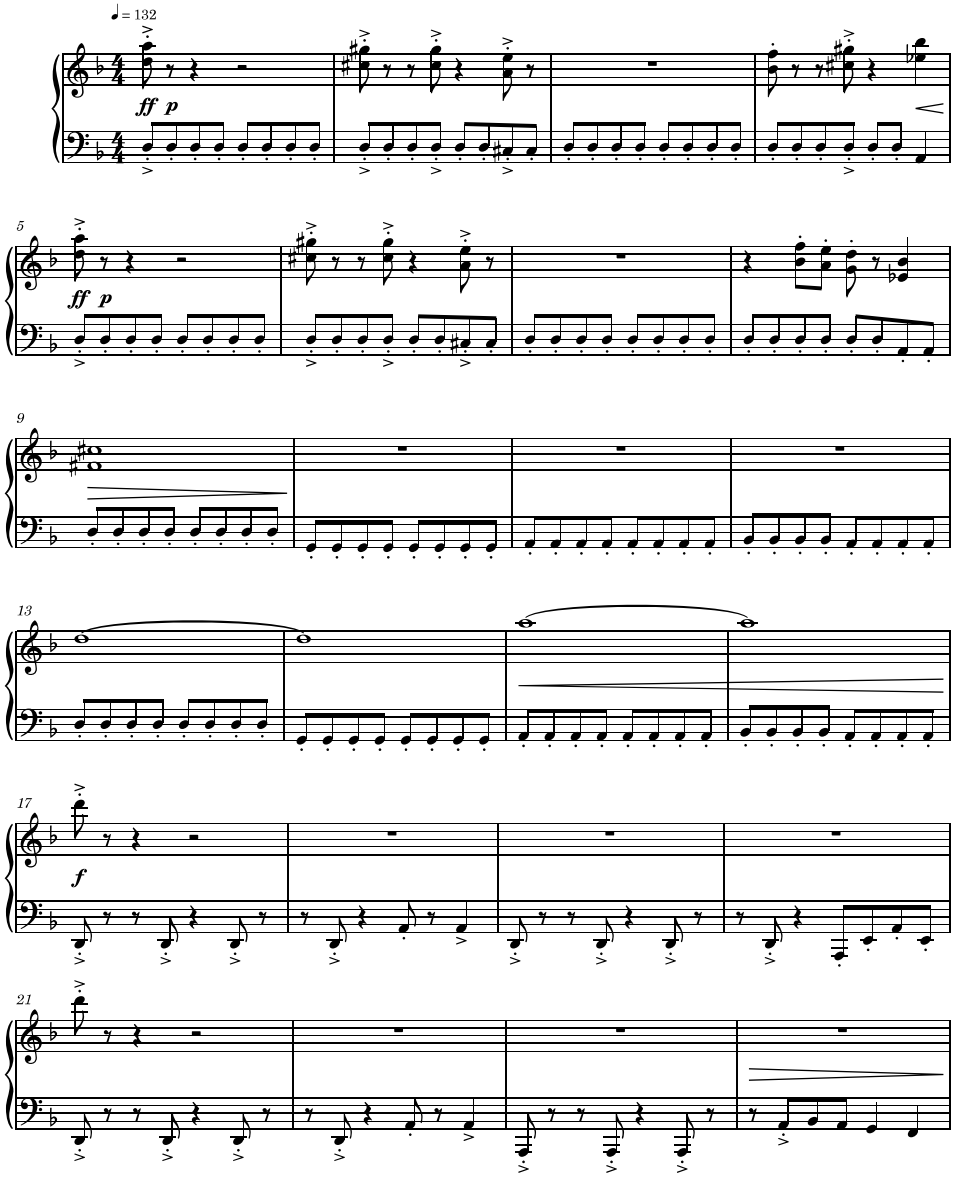
>.3vvq7{>.15^q} 0q 0c 0m >.10q{>.22q} 0q 0q >.10q{>.22q} 0c
   >.3vvq{>.15q}
```

I.2 Equivalent Stave Notation
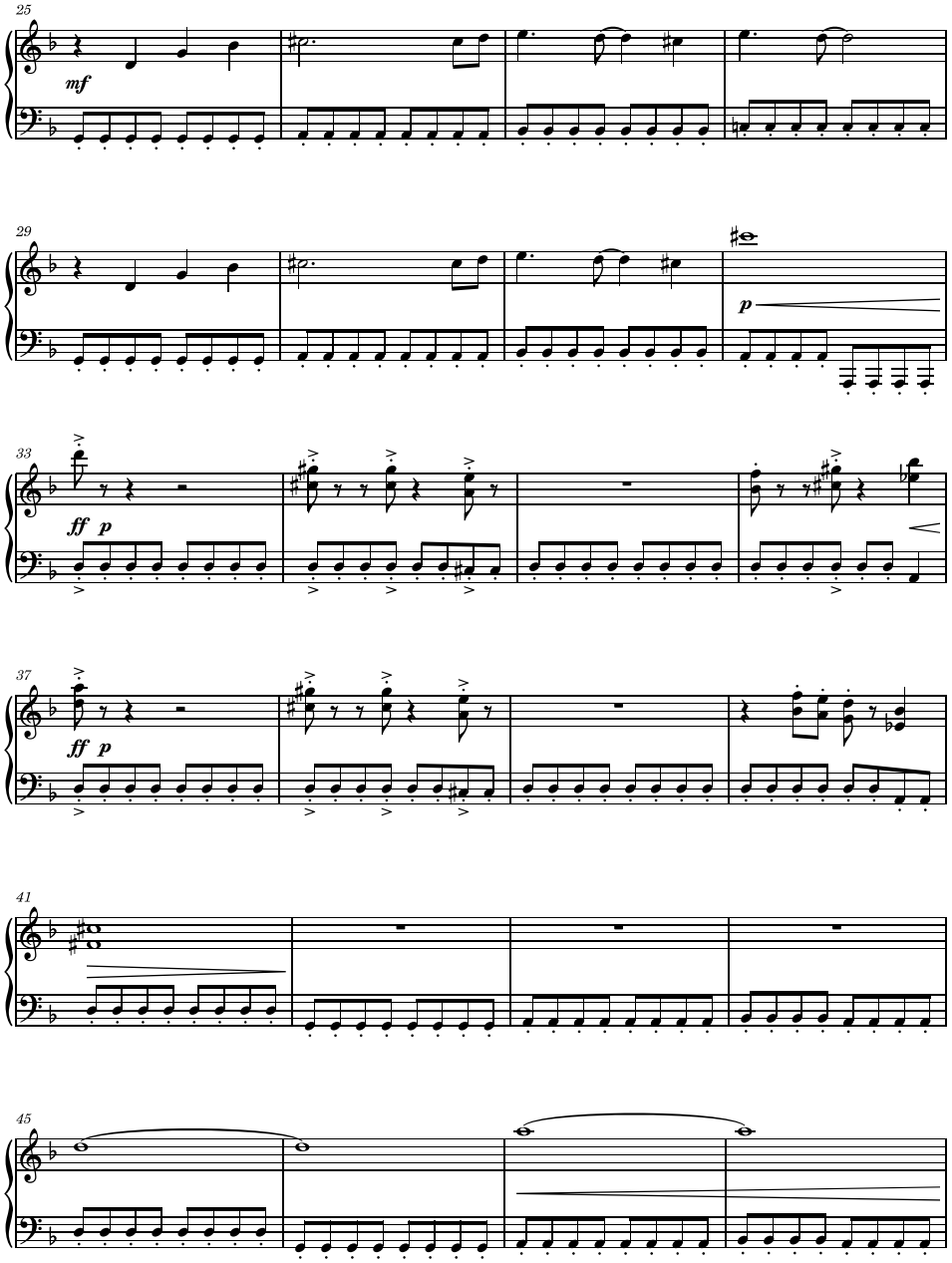
Virtual Music Instrument

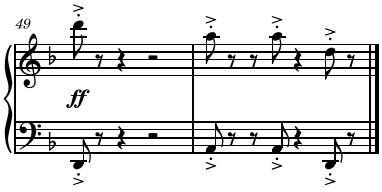
Étude 8

Nathan Cummins



2 Virtual Music Instrument




49
ff

J Peril (But Not Quite Doom)

J.1 Virtual Music Shorthand Notation

Stave 1

```
[B=128;O=3;]
.3q6("b","s") .3q .15q .3q .3q .14q .3q .3q .11q .3q .3q .10q .3q
   .3q .9q .10q
.3q("b","s") .3q .15q .3q .3q .14q .3q .3q .11q .3q .3q .9q("s")
   .10q("s") .11q("s") .9q("s") .10q("s")
.3q("b","s") .3q .15q .3q .3q .14q .3q .3q .11q .3q .3q .10q .3q .3q
   .9q .10q
.3q("b","s") .3q .15q .3q .3q .14q .3q .3q .14q .7q .13q .6q .12q
   .5q .11q("s") .4q("s")

.3q("b","c") .3q .15q("s","h") .3q .3q("b","h") .14q("b")
   .3q("s","h") .3q .11q("b","h") .3q .3q("s","h") .10q .3q("b","h")
   .3q("b") .9q("s","h") .10q("b")
.3q("b","h") .3q .15q("s","h") .3q .3q("b","h") .14q("b")
   .3q("s","h") .3q .11q("b","h") .3q .3q("s","h") .9q("s")
   .10q("b","s","h") .11q("b","s") .9q("s","h") .10q("b","s")
.3q("b","c") .3q .15q("s","h") .3q .3q("b","h") .14q("b")
   .3q("s","h") .3q .11q("b","h") .3q .3q("s","h") .10q .3q("b","h")
   .3q("b") .9q("s","h") .10q("b")
.3q("b","h") .3q .15q("s","h") .3q .3q("b","h") .14q("b")
   .3q("s","h") .3q .11q("b","h") .3q .3q("s","h") .9q("s")
   .10q("b","s","h") .11q("b","s") .14q("s","h") .17q("b","s")

15b-m>3 .22vq .20vq .18vq .17vq
15vb-c.<7 21vvq 22vvq 23vvq 21vvq 22vvq

0b 0b
0b 0b

15m.5 17c 18m 15c 18c
22q 23q 21q 22b. 15q
15q 13c 11b. 21vq
22vb. 22q 20q 18q 17q

15m. 17c 18m 18c 22c
```

```
25q 23q 22q 18b. 15q
17q 15c 13b. 13q
14b>1 26b<6

27q .25c 23b. 21q
22q 23q 21q 22b.-q
22q .20c 18b. 15q
17b-b

27q .25c 23b. 18q
26q .24c 22b.-q
18q .17c 15b. 15q
14b>1 26b<5

15m. 17c 18m 15c 18c
22q 23q 21q 22b. 15q
15q 13c 11b. 21vq
22vb. 22q 20q 18q 17q

15m. 17c 18m 18c 22c
25q 23q 22q 18b. 15q
17q 15c 13b. 13q
14b>1 26b<6

.3q6("b","s","c") .3q .15q .3q .3q .14q .3q .3q .11q .3q .3q .10q
   .3q .3q .9q("b","s","c") .10q("b","s")
.11vq("b","s","c") .11vq .23vq .11vq .11vq .22vq .11vq .11vq .20vq
   .11vq .11vq .17vq("b","s") .18vq("b","s","c") .20vq("b","s")
   .17vq("b","s","c") .18vq("b","s")
.8vq("b","s","c") .8vq .20vq("b") .8vq .8vq("b") .18vq .8vq("b")
   .8vq .17vq("b") .8vq .8vq("b") .15vq .8vq("b") .8vq .14vq("b")
   .15vq
.10vq("b","s","c") .10vq .22vq("b") .10vq .10vq("b") .20vq
   .10vq("b") .10vq .18vq("b") .10vq .10vq("b") .3q("s")
   .5q("b","s") .10q("s") .14q("b","s") .17q("s")

.3q("b","c") .3q .15q("s","h") .3q .3q("b","h") .14q("b")
   .3q("s","h") .3q .11q("b","h") .3q .3q("s","h") .10q .3q("b","h")
   .3q("b") .9q("s","h") .10q("b")
.11vq("b","h") .11vq .23vq("s","h") .11vq .11vq("b","h") .22vq("b")
   .11vq("s","h") .11vq .20vq("b","h") .11vq .11vq("s","h")
   .17vq("s") .18vq("b","s","h") .20vq("b","s") .17vq("s","h")
   .18vq("b","s")
.8vq("b","h") .8vq .20vq("s","h") .8vq .8vq("b","h") .18vq("b")
   .8vq("s","h") .8vq .17vq("b","h") .8vq .8vq("s","h") .8vq
   .8vq("b","s") .8vq("b") .14vq("s","h") .15vq("b")
.10vq("b","h") .10vq .22vq("s","h") .10vq .10vq("b","h") .20vq("b")
   .10vq("s","h") .10vq .18vq("b","h") .10vq .10vq("s","h") .3q("s")
   .4q("b","s","h") .7q("b","s") .12q("s","h") .16q("b","s")

```

```
.5q7("b","s","c","key") .5q .17q .5q .5q .16q .5q .5q .13q .5q .5q
   .12q .5q .5q .11q("b","s","c") .12q("b","s")
.13vq("b","s","c") .13vq .25vq .13vq .13vq .24vq .13vq .13vq .22vq
   .13vq .13vq .19vq("b","s") .20vq("b","s","c") .22vq("b","s")
   .19vq("b","s","c") .20vq("b","s")
.10vq6("b","s","c") .10vq .22vq("b") .10vq .10vq("b") .20vq
   .10vq("b") .10vq .19vq("b") .10vq .10vq("b") .17vq .10vq("b")
   .10vq .16vq("b") .17vq
.12vq("b","s","c") .12vq .24vq("b") .12vq .12vq("b") .22vq
   .12vq("b") .12vq .20vq("b") .12vq .12vq("b") .11q("s")
   .12q("b","s") .13q("s") .16q("b","s") .19q("s")

17m. 19c 20m 17c 20c
24q 25q 23q 24b. 17q
17q 15c 13b. 23vq
24vb. 24q 22q 20q 19q

17m. 19c 20m 20c 24c
27q 25q 24q 20b. 17q
19q 17c 15b. 15q
16b>1 28b<6

29q .27c 25b. 23q
24q 25q 23q 24b.-q
24q .22c 20b. 17q
19b-b

29q .27c 25b. 20q
28q .26c 24b.-q
20q .19c 17b. 17q
16b>1 28b<6

.5q("b","s","key") .5q .17q .5q .5q .16q .5q .5q .13q .5q .5q .12q
   .5q .5q .11q .12q
.5q("b","s") .5q .17q .5q .5q .16q .5q .5q .13q .5q .5q .11q("s")
   .12q("s") .13q("s") .11q("s") .12q("s")
.5q("b","s") .5q .17q .5q .5q .16q .5q .5q .13q .5q .5q .12q .5q .5q
   .11q .12q
.5q("b","s") .5q .17q .5q .5q .16q .5q .5q .16q .9q .15q .8q .14q
   .7q .13q("s") .6q("s")

.5q("b","c") .5q .17q("s","h") .5q .5q("b","h") .16q("b")
   .5q("s","h") .5q .13q("b","h") .5q .5q("s","h") .12q .5q("b","h")
   .5q("b") .11q("s","h") .12q("b")
.5q("b","h") .5q .17q("s","h") .5q .5q("b","h") .16q("b")
   .5q("s","h") .5q .13q("b","h") .5q .5q("s","h") .11q("s")
   .12q("b","s","h") .13q("b","s") .11q("s","h") .12q("b","s")
.5q("b","c") .5q .17q("s","h") .5q .5q("b","h") .16q("b")
   .5q("s","h") .5q .13q("b","h") .5q .5q("s","h") .12q .5q("b","h")
   .5q("b") .11q("s","h") .12q("b")
```

```
.5q("b","h") .5q .17q("s","h") .5q .5q("b","h") .16q("b")
   .5q("s","h") .5q .13q("b","h") .5q .5q("s","h") .11q("s")
   .12q("b","s","h") .13q("b","s") .16q("s","h") .19q("b","s")

.17q 0q 0q .17q 0q .17q 0q .17q 0q >29q7 .29q 0q >.17vq
```

Stave 2

```
.15vvq6 0q 0c 0m 0b
.15vvq 0q 0c 0m 0b
.15vvq 0q 0c 0m 0b
.15vvq 0q 0c 0m 0b

.15vvq .15vvq 0q .15vvq .15vvq 0q .15vvq .15vvq 0q .15vvq .15vvq 0q
   .15vvq .15vvq 0c
.15vvq .15vvq 0q .15vvq .15vvq 0q .15vvq .15vvq 0q .15vvq .15vvq 0q
   0m
.15vvq .15vvq 0q .15vvq .15vvq 0q .15vvq .15vvq 0q .15vvq .15vvq 0q
   .15vvq .15vvq 0c
.15vvq .15vvq 0q .15vvq .15vvq 0q .15vvq .15vvq 0q .15vvq .15vvq 0q
   0m

.3vc("b","c") .3vc("b") .3vc5("b") .3vc("b") .3vc4("b") .3vc("b")
   .3vc3("b") .1vc("b")
.11vc("b") .11vc("b") .11vc4("b") .11vc5("b") .11vc6("b") .11vq("b")
   9vvq 10vvq("b") 11vvq("b") 9vvq("s") 10vvq

.3vq6("b","s","c") .3vq .15vq .3vq .3vq .14vq .3vq .3vq .11vq .3vq
   .3vq .10vq .3vq .3vq .9vq("s") .10vq("s")
.3vq5("b","s","c") .3vq .15vq .3vq .3vq .14vq .3vq .3vq .11vq .3vq
   .3vq .10vq .3vq .3vq("b") .9vq("b","s","h") .10vq

.3vq("b","h") .3vq("h") .3vq("s","h") .3vq("h") .3vq("b","h")
   .3vq("b","h") .3vq("s","h") .3vq("h") .3vq("b","h") .3vq("h")
   .3vq("s","h") .3vq("h") .3vq("b","h") .3vq("b","h")
   .9vvq("s","h") .10vvq("b","h")
.11vvq("b","h") .11vvq("h") .11vvq("s","h") .11vvq("h")
   .11vvq("b","h") .11vvq("b","h") .11vvq("s","h") .11vvq("h")
   .11vvq("b","h") .11vvq("h") .11vvq("s","h") .11vvq("h")
   .11vvq("b","h") .11vvq("b","h") .5vvq("s","h") .6vvq("b","h")
.8vvq("b","h") .8vvq("h") .8vvq("s","h") .8vvq("h") .8vvq("b","h")
   .8vvq("b","h") .8vvq("s","h") .8vvq("h") .8vvq("b","h")
   .8vvq("h") .8vvq("s","h") .8vvq("h") .8vvq("b","h")
   .8vvq("b","h") .8vvq("s","h") .9vvq("b","h")
.10vvq("b","h") .10vvq("h") .10vvq("s","h") .10vvq("h")
   .10vvq("b","h") .10vvq("b","h") .10vvq("s","h") .10vvq("h")
```

```
   .10vvq("b","h") .10vvq("h") .10vvq("s","h") .10vvq("h")
   .10vvq("b","h") .10vvq("b","h") .2vq("s","h") .5vq("b","h")
21
22 .3vq("b","h") .3vq("h") .3vq("s","h") .3vq("h") .3vq("b","h")
   .3vq("b","h") .3vq("s","h") .3vq("h") .3vq("b","h") .3vq("h")
   .3vq("s","h") .3vq("h") .3vq("b","h") .3vq("b","h")
   .9vvq("s","h") .10vvq("b","h")
23 .11vvq("b","h") .11vvq("h") .11vvq("s","h") .11vvq("h")
   .11vvq("b","h") .11vvq("b","h") .11vvq("s","h") .11vvq("h")
   .11vvq("b","h") .11vvq("h") .11vvq("s","h") .11vvq("h")
   .11vvq("b","h") .11vvq("b","h") .11vvq("s","h") .12vvq("b","h")
24 .1vq("b","h") .1vq("h") .1vq("s","h") .1vq("h") .1vq("b","h")
   .1vq("b","h") .1vq("s","h") .1vq("h") .1vq("b","h") .1vq("h")
   .1vq("s","h") .1vq("h") .1vq("b","h") .1vq("b","h")
   .8vvq("s","h") .9vvq("b","h")
25 .10vvq("b","h") .10vvq("h") .10vvq("s","h") .10vvq("h")
   .10vvq("b","h") .10vvq("b","h") .10vvq("s","h") .10vvq("h")
   .10vvq("b","h") .10vvq("h") .10vvq("s","h") .10vvq("h")
   .10vvq("b","h") .10vvq("b","h") .2vq("s","h") .5vq("b","h")
26
27 .11vvq6("c","b") .11vvq .11vvq("s","h") .11vvq("h") .11vvq("b","h")
   .11vvq("b","h") .11vvq("s","h") .11vvq("h") .11vvq("b","h")
   .11vvq("h") .11vvq("s","h") .11vvq("h") .11vvq("b","h")
   .11vvq("b","h") .11vvq("s","h") .11vvq("b","h")
28 .10vvq("b","h") .10vvq("h") .10vvq("s","h") .10vvq("h")
   .10vvq("b","h") .10vvq("b","h") .10vvq("s","h") .10vvq("h")
   .10vvq("b","h") .10vvq("h") .10vvq("s","h") .10vvq("h")
   .10vvq("b","h") .10vvq("b","h") .10vvq("s","h") .10vvq("b","h")
29 .3vq("b","h") .3vq("h") .3vq("s","h") .3vq("h") .3vq("b","h")
   .3vq("b","h") .3vq("s","h") .3vq("h") .3vq("b","h") .3vq("h")
   .3vq("s","h") .3vq("h") .3vq("b","h") .3vq("b","h") .3vq("s","h")
   .9vvq("b","h")
30 .10vvq("b","h") .10vvq("h") .10vvq("s","h") .10vvq("h")
   .10vvq("b","h") .10vvq("b","h") .10vvq("s","h") .10vvq("h")
   .10vvq("b","h") .10vvq("h") .10vvq("s","h") .10vvq("h")
   .10vvq("b","h") .10vvq("b","h") .10vvq("s","h") .10vvq("b","h")
31
32 .11vvq("b","h") .11vvq("h") .11vvq("s","h") .11vvq("h")
   .11vvq("b","h") .11vvq("b","h") .11vvq("s","h") .11vvq("h")
   .11vvq("b","h") .11vvq("h") .11vvq("s","h") .11vvq("h")
   .11vvq("b","h") .11vvq("b","h") .11vvq("s","h") .11vvq("b","h")
33 .10vvq("b","h") .10vvq("h") .10vvq("s","h") .10vvq("h")
   .10vvq("b","h") .10vvq("b","h") .10vvq("s","h") .10vvq("h")
   .10vvq("b","h") .10vvq("h") .10vvq("s","h") .10vvq("h")
   .10vvq("b","h") .10vvq("b","h") .10vvq("s","h") .10vvq("b","h")
34 .10vvq("b","h") .10vvq("h") .10vvq("s","h") .10vvq("h")
   .10vvq("b","h") .10vvq("b","h") .10vvq("s","h") .10vvq("h")
   .10vvq("b","h") .10vvq("h") .10vvq("s","h") .10vvq("h")
   .10vvq("b","h") .10vvq("b","h") .10vvq("s","h") .10vvq("b","h")
35 .10vvq("b","h") .10vvq("h") .10vvq("s","h") .10vvq("h")
```

```
   .10vvq("b","h") .10vvq("b","h") .10vvq("s","h") .10vvq("h")
   .10vvq("b","h") .10vvq("h") .10vvq("s","h") .10vvq("h")
   .10vvq("b","h") .10vvq("b","h") .10vvq("s","h") .10vvq("b","h")
36
37 .3vq5("b","h") .3vq("h") .3vq("s","h") .3vq("h") .3vq("b","h")
   .3vq("b","h") .3vq("s","h") .3vq("h") .3vq("b","h") .3vq("h")
   .3vq("s","h") .3vq("h") .3vq("b","h") .3vq("b","h")
   .9vvq("s","h") .10vvq("b","h")
38 .11vvq("b","h") .11vvq("h") .11vvq("s","h") .11vvq("h")
   .11vvq("b","h") .11vvq("b","h") .11vvq("s","h") .11vvq("h")
   .11vvq("b","h") .11vvq("h") .11vvq("s","h") .11vvq("h")
   .11vvq("b","h") .11vvq("b","h") .5vvq("s","h") .6vvq("b","h")
39 .8vvq("b","h") .8vvq("h") .8vvq("s","h") .8vvq("h") .8vvq("b","h")
   .8vvq("b","h") .8vvq("s","h") .8vvq("h") .8vvq("b","h")
   .8vvq("h") .8vvq("s","h") .8vvq("h") .8vvq("b","h")
   .8vvq("b","h") .8vvq("s","h") .9vvq("b","h")
40 .10vvq("b","h") .10vvq("h") .10vvq("s","h") .10vvq("h")
   .10vvq("b","h") .10vvq("b","h") .10vvq("s","h") .10vvq("h")
   .10vvq("b","h") .10vvq("h") .10vvq("s","h") .10vvq("h")
   .10vvq("b","h") .10vvq("b","h") .2vq("s","h") .5vq("b","h")
41
42 .3vq("b","h") .3vq("h") .3vq("s","h") .3vq("h") .3vq("b","h")
   .3vq("b","h") .3vq("s","h") .3vq("h") .3vq("b","h") .3vq("h")
   .3vq("s","h") .3vq("h") .3vq("b","h") .3vq("b","h")
   .9vvq("s","h") .10vvq("b","h")
43 .11vvq("b","h") .11vvq("h") .11vvq("s","h") .11vvq("h")
   .11vvq("b","h") .11vvq("b","h") .11vvq("s","h") .11vvq("h")
   .11vvq("b","h") .11vvq("h") .11vvq("s","h") .11vvq("h")
   .11vvq("b","h") .11vvq("b","h") .11vvq("s","h") .12vvq("b","h")
44 .1vq("b","h") .1vq("h") .1vq("s","h") .1vq("h") .1vq("b","h")
   .1vq("b","h") .1vq("s","h") .1vq("h") .1vq("b","h") .1vq("h")
   .1vq("s","h") .1vq("h") .1vq("b","h") .1vq("b","h")
   .8vvq("s","h") .9vvq("b","h")
45 .10vvq("b","h") .10vvq("h") .10vvq("s","h") .10vvq("h")
   .10vvq("b","h") .10vvq("b","h") .10vvq("s","h") .10vvq("h")
   .10vvq("b","h") .10vvq("h") .10vvq("s","h") .10vvq("h")
   .10vvq("b","h") .10vvq("b","h") .2vq("s","h") .5vq("b","h")
46
47 .15vvq 0q 0c 0m 0b
48 .23vvvq 0q 0c 0m 0b
49 .20vvvq 0q 0c 0m 0b
50 .22vvvq 0q 0c 0m 0b
51
52 .15vvq .15vvq 0q .15vvq .15vvq 0q .15vvq .15vvq 0q .15vvq .15vvq 0q
   .15vvq .15vvq 0c
53 .23vvvq .23vvvq 0q .23vvvq .23vvvq 0q .23vvvq .23vvvq 0q .23vvvq
   .23vvvq 0q 0m
54 .20vvvq .20vvvq 0q .20vvvq .20vvvq 0q .20vvvq .20vvvq 0q .20vvvq
   .20vvvq 0q .20vvvq .20vvvq 0c
55 .22vvvq .22vvvq 0q .22vvvq .22vvvq 0q .22vvvq .22vvvq 0q .22vvvq
```

```
   .22vvvq .22vvvq .24vvvq .24vvvq .24vvvq .24vvvq
56
57 .17vvq7 .17vvq 0q .17vvq .17vvq 0q .17vvq .17vvq 0q .17vvq .17vvq 0q
   .17vvq .17vvq 0c
58 .25vvvq .25vvvq 0q .25vvvq .25vvvq 0q .25vvvq .25vvvq 0q .25vvvq
   .25vvvq 0q 0m
59 .22vvvq6 .22vvvq 0q .22vvvq .22vvvq 0q .22vvvq .22vvvq 0q .22vvvq
   .22vvvq 0q .22vvvq .22vvvq 0c
60 .24vvvq .24vvvq 0q .24vvvq .24vvvq 0q .24vvvq .24vvvq 0q .24vvvq
   .24vvvq 0q 0m
61
62 .5vq("b","h") .5vq("h") .5vq("s","h") .5vq("h") .5vq("b","h")
   .5vq("b","h") .5vq("s","h") .5vq("h") .5vq("b","h") .5vq("h")
   .5vq("s","h") .5vq("h") .5vq("b","h") .5vq("b","h")
   .11vvq("s","h") .12vvq("b","h")
63 .13vvq("b","h") .13vvq("h") .13vvq("s","h") .13vvq("h")
   .13vvq("b","h") .13vvq("b","h") .13vvq("s","h") .13vvq("h")
   .13vvq("b","h") .13vvq("h") .13vvq("s","h") .13vvq("h")
   .13vvq("b","h") .13vvq("b","h") .7vvq("s","h") .8vvq("b","h")
64 .10vvq("b","h") .10vvq("h") .10vvq("s","h") .10vvq("h")
   .10vvq("b","h") .10vvq("b","h") .10vvq("s","h") .10vvq("h")
   .10vvq("b","h") .10vvq("h") .10vvq("s","h") .10vvq("h")
   .10vvq("b","h") .10vvq("b","h") .10vvq("s","h") .11vvq("b","h")
65 .12vvq("b","h") .12vvq("h") .12vvq("s","h") .12vvq("h")
   .12vvq("b","h") .12vvq("b","h") .12vvq("s","h") .12vvq("h")
   .12vvq("b","h") .12vvq("h") .12vvq("s","h") .12vvq("h")
   .12vvq("b","h") .12vvq("b","h") .4vq("s","h") .7vq("b","h")
66
67 .5vq("b","h") .5vq("h") .5vq("s","h") .5vq("h") .5vq("b","h")
   .5vq("b","h") .5vq("s","h") .5vq("h") .5vq("b","h") .5vq("h")
   .5vq("s","h") .5vq("h") .5vq("b","h") .5vq("b","h")
   .11vvq("s","h") .12vvq("b","h")
68 .13vvq("b","h") .13vvq("h") .13vvq("s","h") .13vvq("h")
   .13vvq("b","h") .13vvq("b","h") .13vvq("s","h") .13vvq("h")
   .13vvq("b","h") .13vvq("h") .13vvq("s","h") .13vvq("h")
   .13vvq("b","h") .13vvq("b","h") .13vvq("s","h") .14vvq("b","h")
69 .3vq("b","h") .3vq("h") .3vq("s","h") .3vq("h") .3vq("b","h")
   .3vq("b","h") .3vq("s","h") .3vq("h") .3vq("b","h") .3vq("h")
   .3vq("s","h") .3vq("h") .3vq("b","h") .3vq("b","h")
   .10vvq("s","h") .11vvq("b","h")
70 .12vvq("b","h") .12vvq("h") .12vvq("s","h") .12vvq("h")
   .12vvq("b","h") .12vvq("b","h") .12vvq("s","h") .12vvq("h")
   .12vvq("b","h") .12vvq("h") .12vvq("s","h") .12vvq("h")
   .12vvq("b","h") .12vvq("b","h") .4vq("s","h") .7vq("b","h")
71
72 .13vvq("c","b") .13vvq .13vvq("s","h") .13vvq("h") .13vvq("b","h")
   .13vvq("b","h") .13vvq("s","h") .13vvq("h") .13vvq("b","h")
   .13vvq("h") .13vvq("s","h") .13vvq("h") .13vvq("b","h")
   .13vvq("b","h") .13vvq("s","h") .13vvq("b","h")
73 .12vvq("b","h") .12vvq("h") .12vvq("s","h") .12vvq("h")
```

```
   .12vvq("b","h") .12vvq("b","h") .12vvq("s","h") .12vvq("h")
   .12vvq("b","h") .12vvq("h") .12vvq("s","h") .12vvq("h")
   .12vvq("b","h") .12vvq("b","h") .12vvq("s","h") .12vvq("b","h")
.5vq("b","h") .5vq("h") .5vq("s","h") .5vq("h") .5vq("b","h")
   .5vq("b","h") .5vq("s","h") .5vq("h") .5vq("b","h") .5vq("h")
   .5vq("s","h") .5vq("h") .5vq("b","h") .5vq("b","h") .5vq("s","h")
   .11vvq("b","h")
.12vvq("b","h") .12vvq("h") .12vvq("s","h") .12vvq("h")
   .12vvq("b","h") .12vvq("b","h") .12vvq("s","h") .12vvq("h")
   .12vvq("b","h") .12vvq("h") .12vvq("s","h") .12vvq("h")
   .12vvq("b","h") .12vvq("b","h") .12vvq("s","h") .12vvq("b","h")

.13vvq("b","h") .13vvq("h") .13vvq("s","h") .13vvq("h")
   .13vvq("b","h") .13vvq("b","h") .13vvq("s","h") .13vvq("h")
   .13vvq("b","h") .13vvq("h") .13vvq("s","h") .13vvq("h")
   .13vvq("b","h") .13vvq("b","h") .13vvq("s","h") .13vvq("b","h")
.12vvq("b","h") .12vvq("h") .12vvq("s","h") .12vvq("h")
   .12vvq("b","h") .12vvq("b","h") .12vvq("s","h") .12vvq("h")
   .12vvq("b","h") .12vvq("h") .12vvq("s","h") .12vvq("h")
   .12vvq("b","h") .12vvq("b","h") .12vvq("s","h") .12vvq("b","h")
.12vvq("b","h") .12vvq("h") .12vvq("s","h") .12vvq("h")
   .12vvq("b","h") .12vvq("b","h") .12vvq("s","h") .12vvq("h")
   .12vvq("b","h") .12vvq("h") .12vvq("s","h") .12vvq("h")
   .12vvq("b","h") .12vvq("b","h") .12vvq("s","h") .12vvq("b","h")
.12vvq("b","h") .12vvq("h") .12vvq("s","h") .12vvq("h")
   .12vvq("b","h") .12vvq("b","h") .12vvq("s","h") .12vvq("h")
   .12vvq("b","h") .12vvq("h") .12vvq("s","h") .12vvq("h")
   .12vvq("b","h") .12vvq("b","h") .12vvq("s","h") .12vvq("b","h")
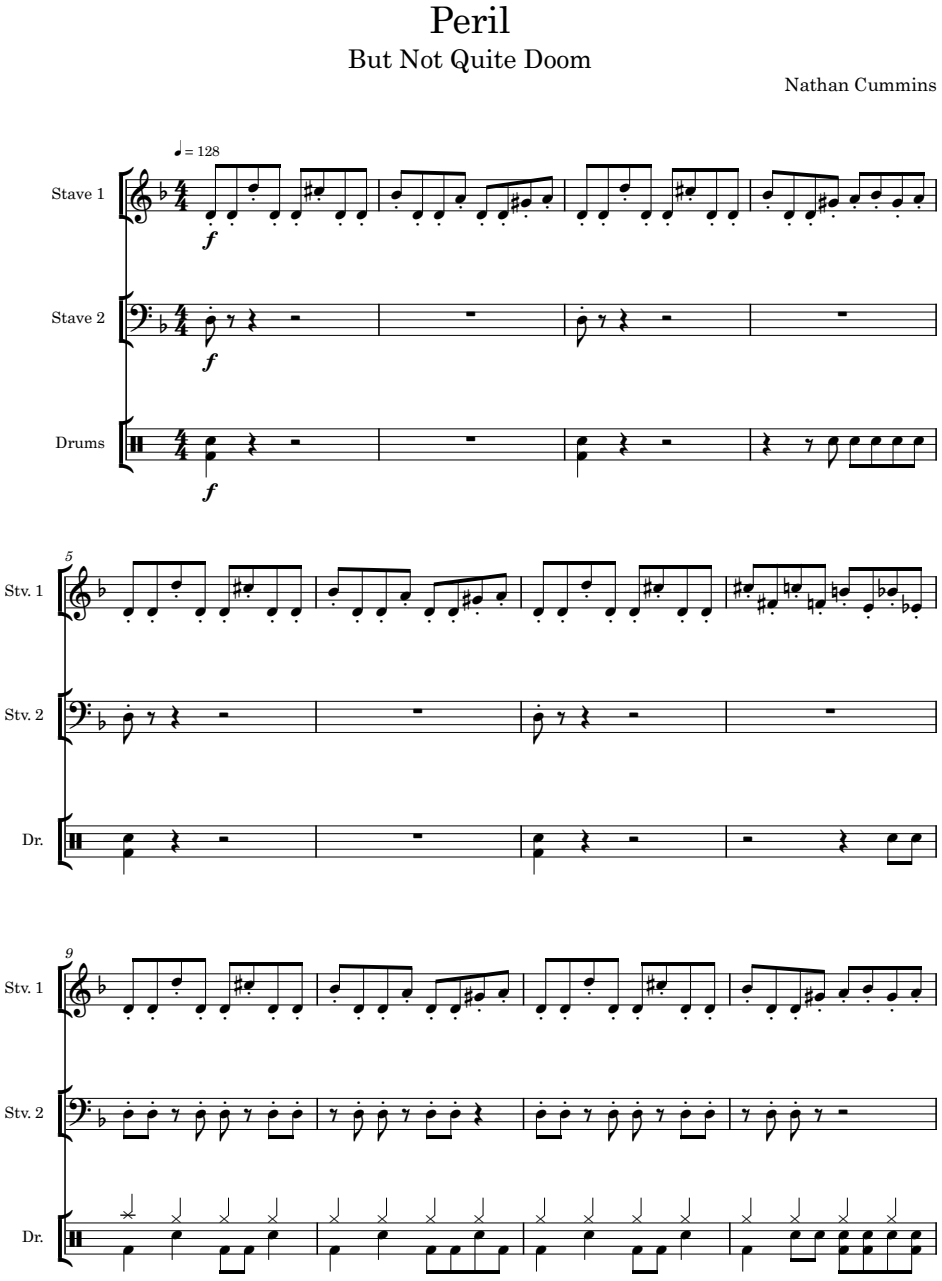
.17vvq 0q 0c 0m 0b
.17vvq 0q 0c 0m 0b
.17vvq 0q 0c 0m 0b
.17vvq 0q 0c 0m 0b
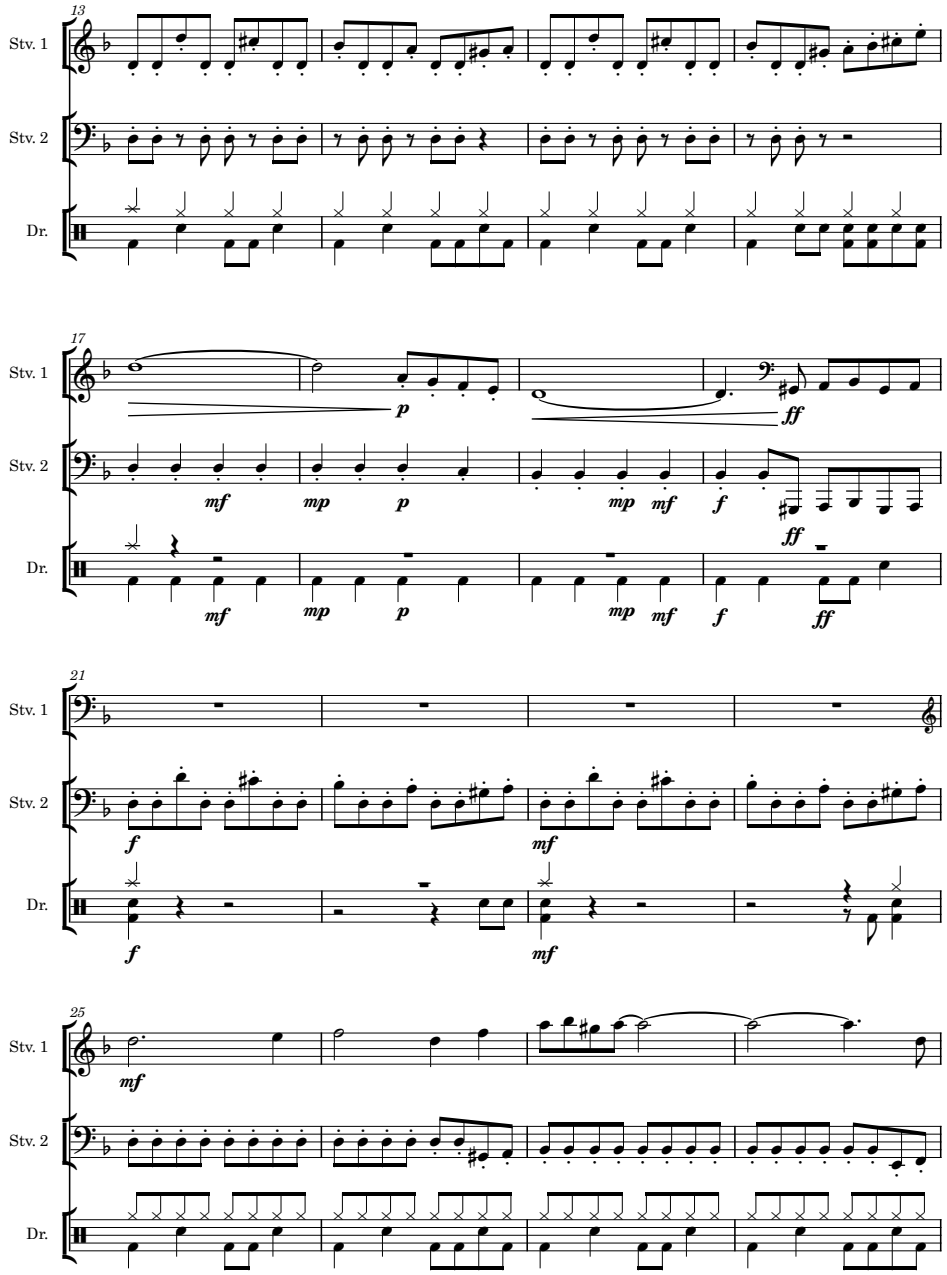
.17vvq .17vvq 0q .17vvq .17vvq 0q .17vvq .17vvq 0q .17vvq .17vvq 0q
   .17vvq .17vvq 0c
.17vvq .17vvq 0q .17vvq .17vvq 0q .17vvq .17vvq 0q .17vvq .17vvq 0q
   0m
.17vvq .17vvq 0q .17vvq .17vvq 0q .17vvq .17vvq 0q .17vvq .17vvq 0q
   .17vvq .17vvq 0c
.17vvq .17vvq 0q .17vvq .17vvq 0q .17vvq .17vvq 0q .17vvq .17vvq 0q
   0m
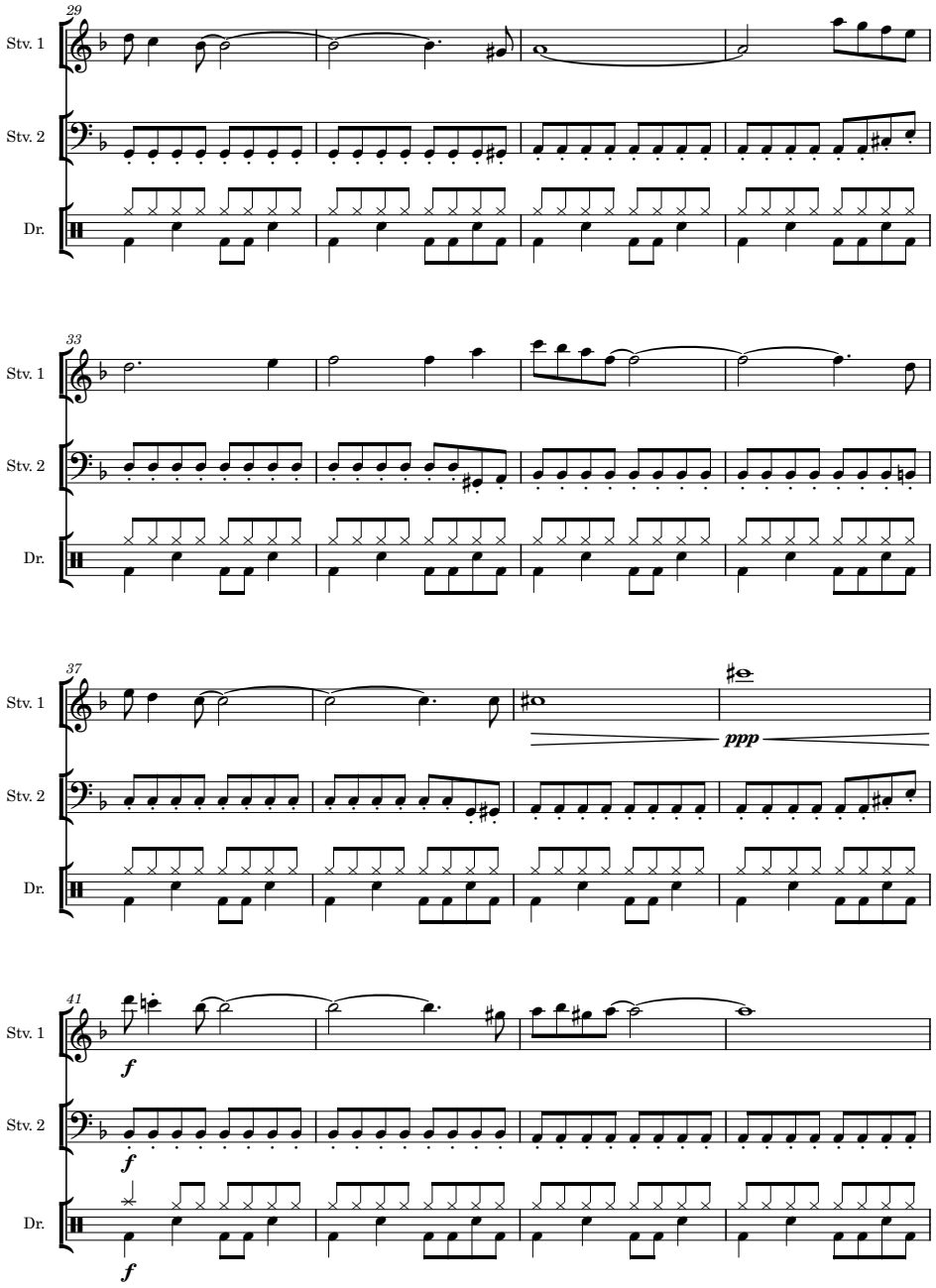
.5vq 0q 0q .5vq 0q .5vq 0q .5vq 0q >17vvvq7 .17vvvq 0q >.5vvvq
```

J.2 Equivalent Stave Notation
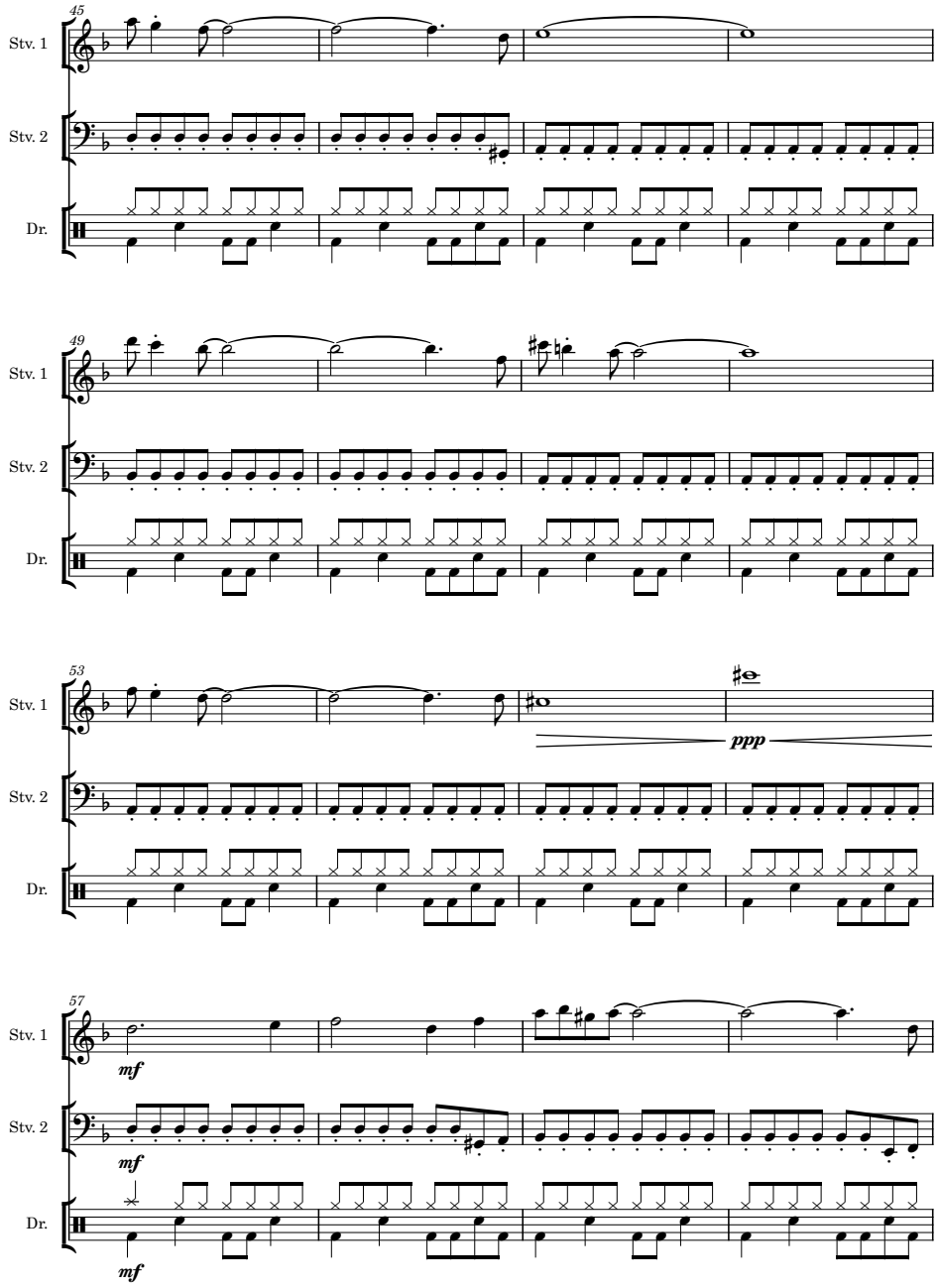
Virtual Music Instrument
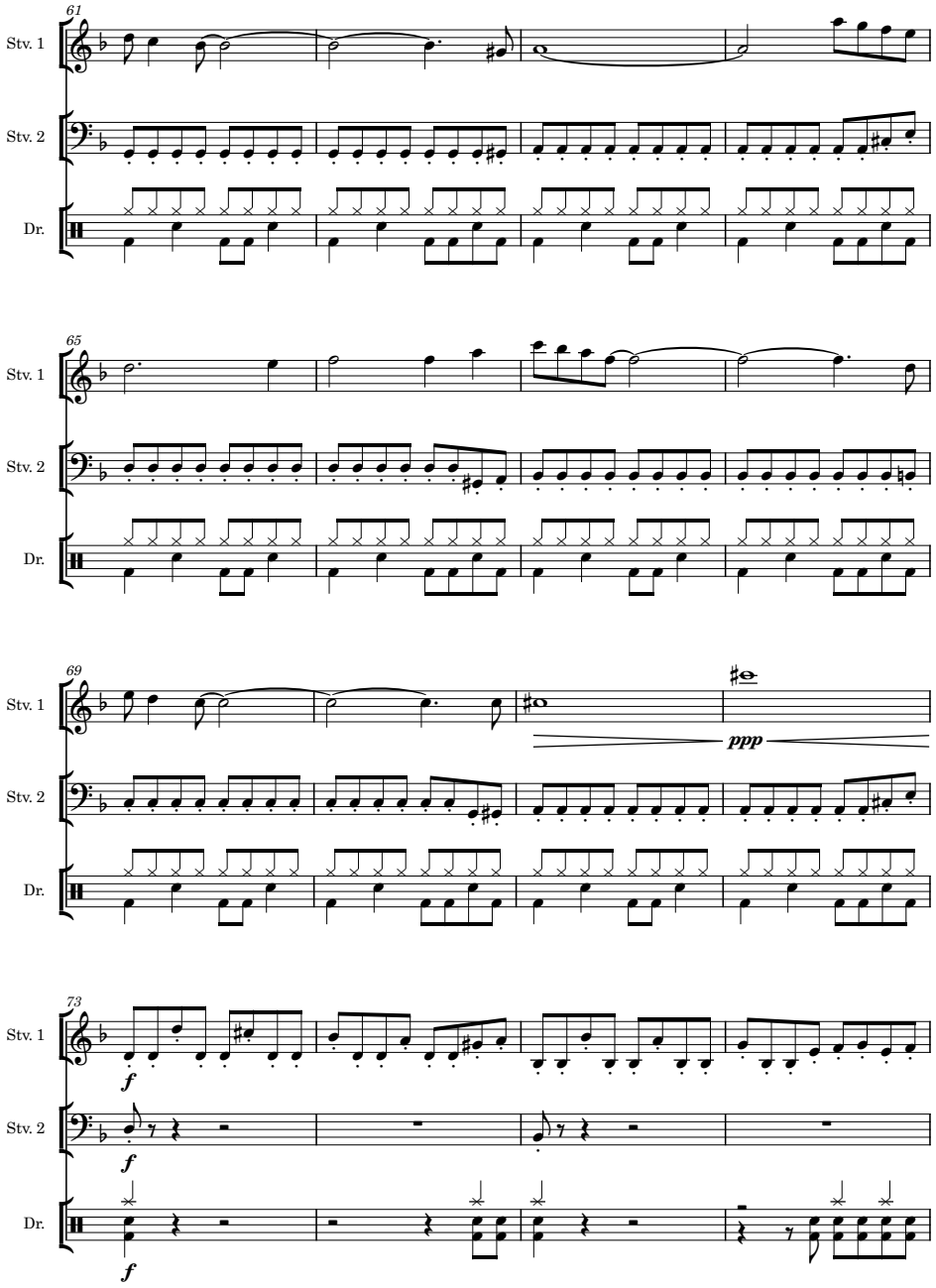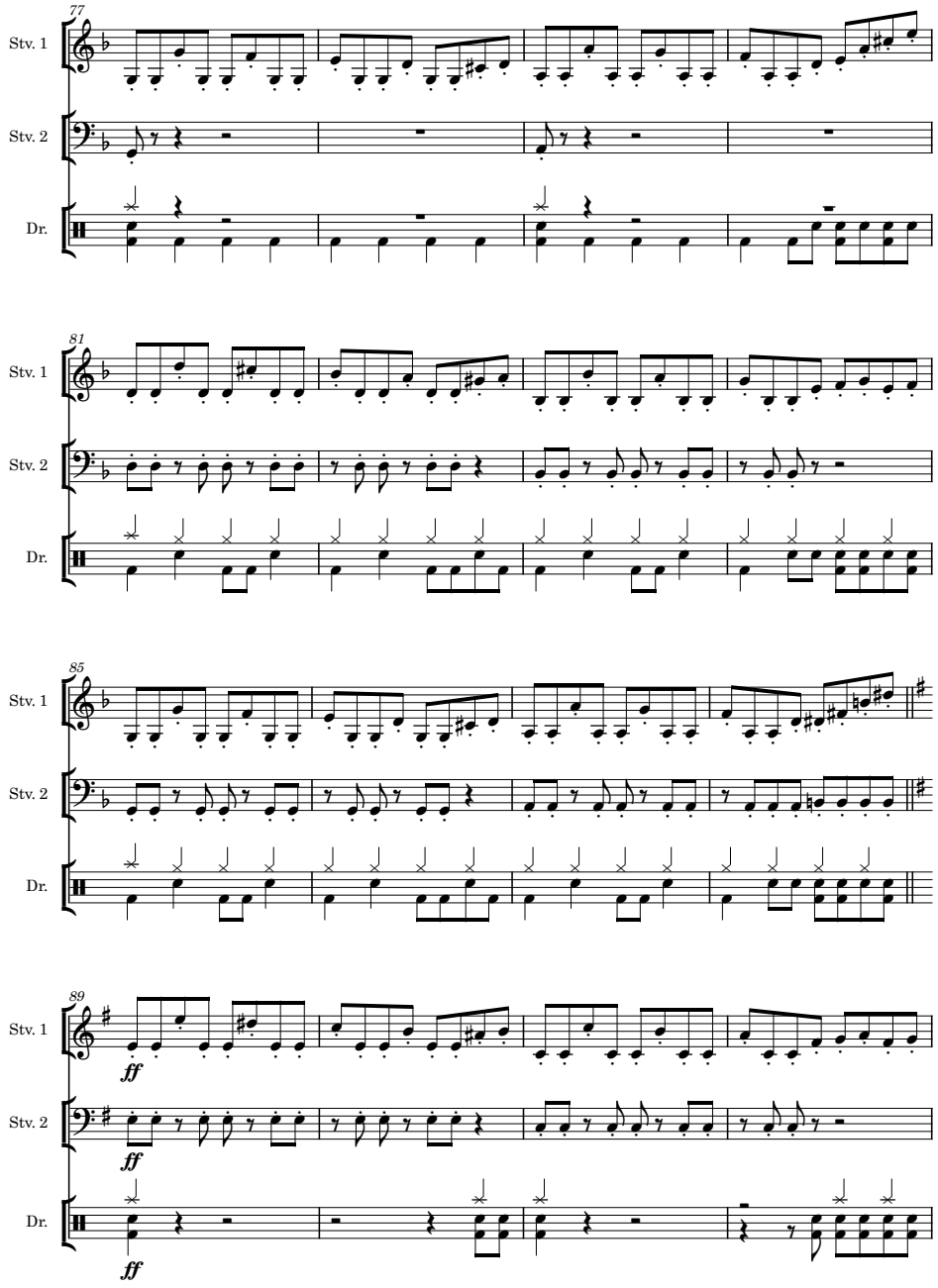



2
Virtual Music Instrument
13
Stv. 1
Stv. 2
Dr.
17
p
ff
mf
mp
p
mp
mf
f
ff
21
f
mf
25
mf
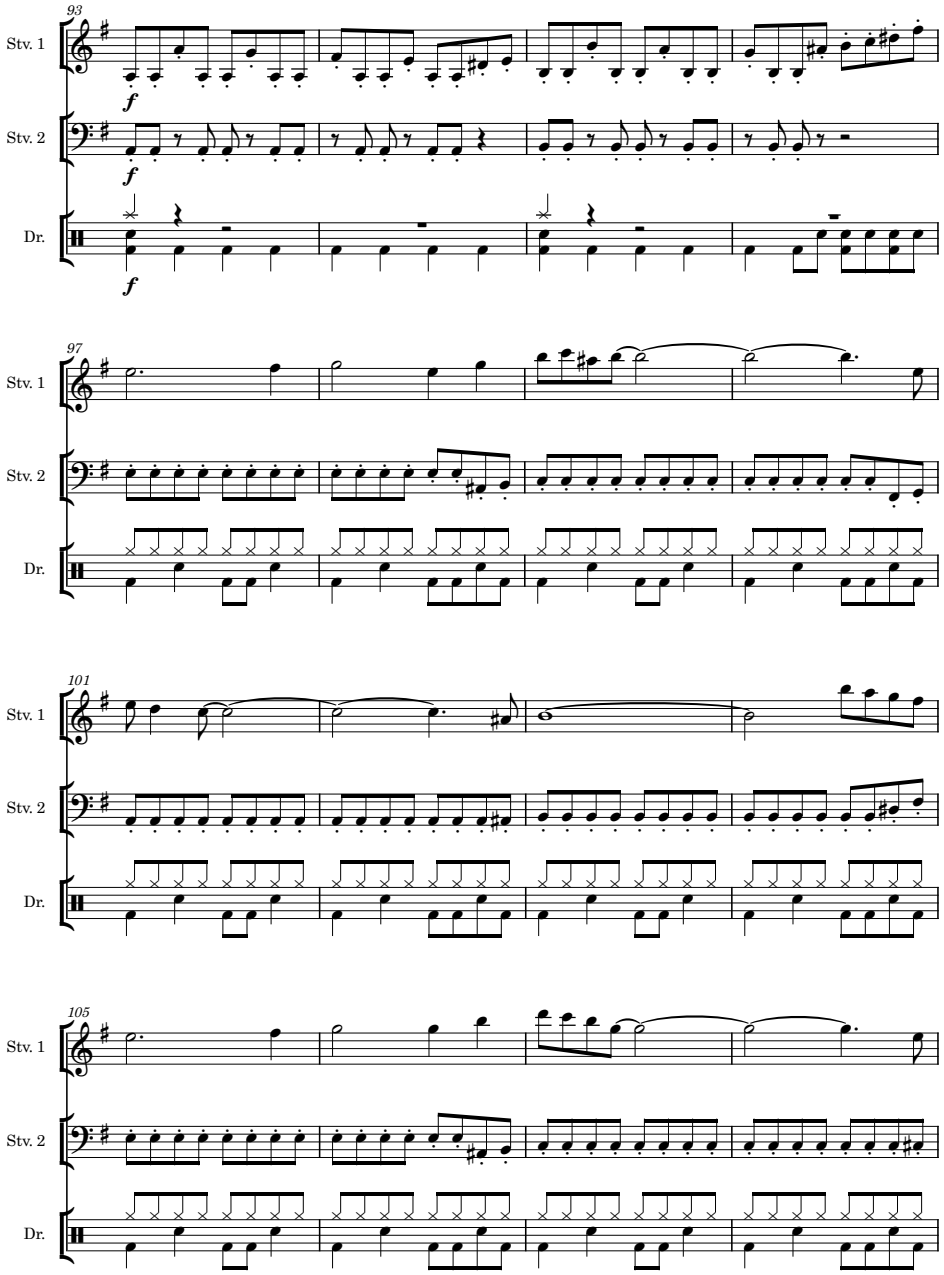

29
Stv. 1
Stv. 2
Dr.
33
Stv. 1
Stv. 2
Dr.
37
Stv. 1
ppp
Stv. 2
Dr.
41
Stv. 1
f
Stv. 2
f
Dr.
f

4 Virtual Music Instrument
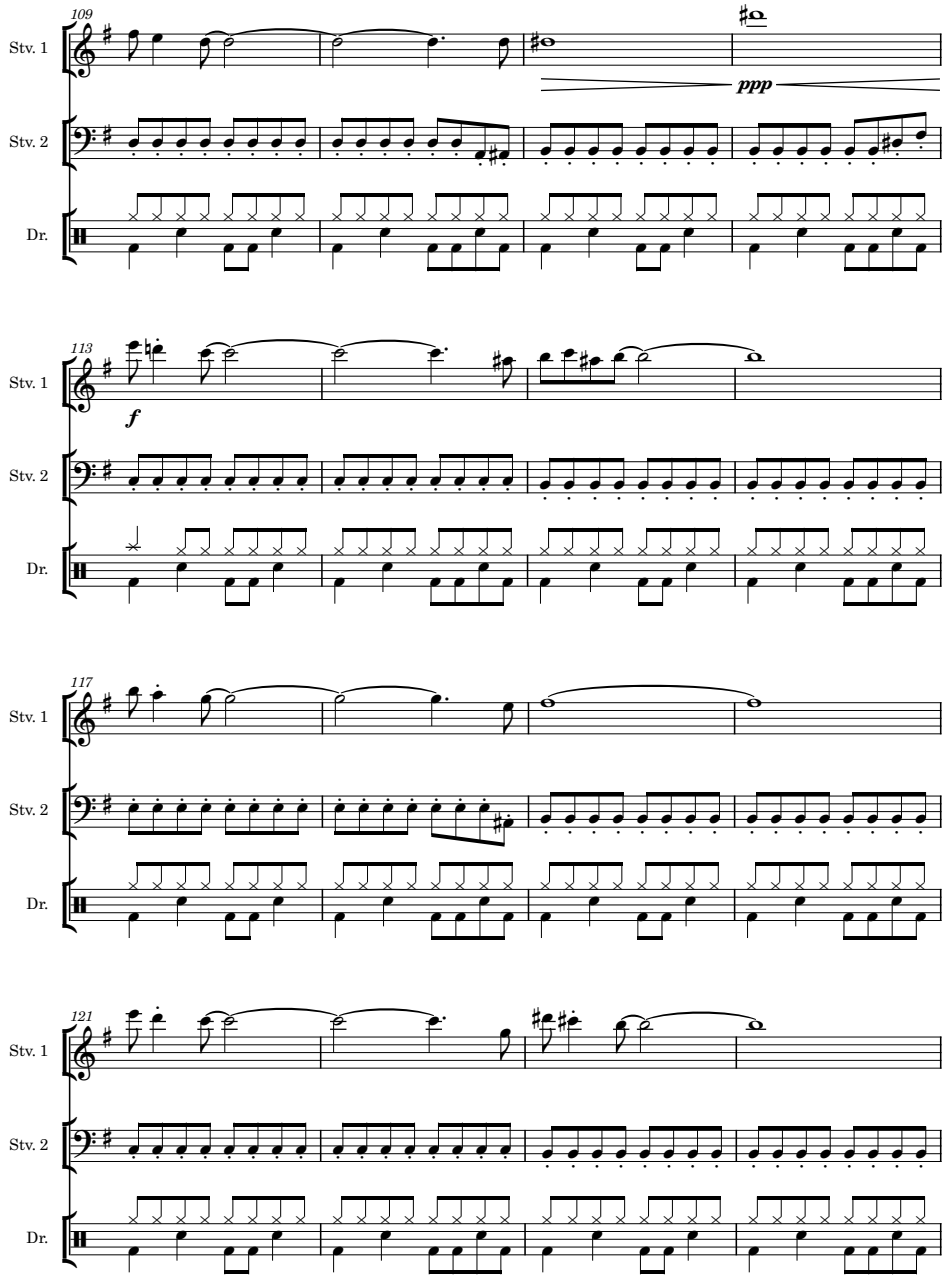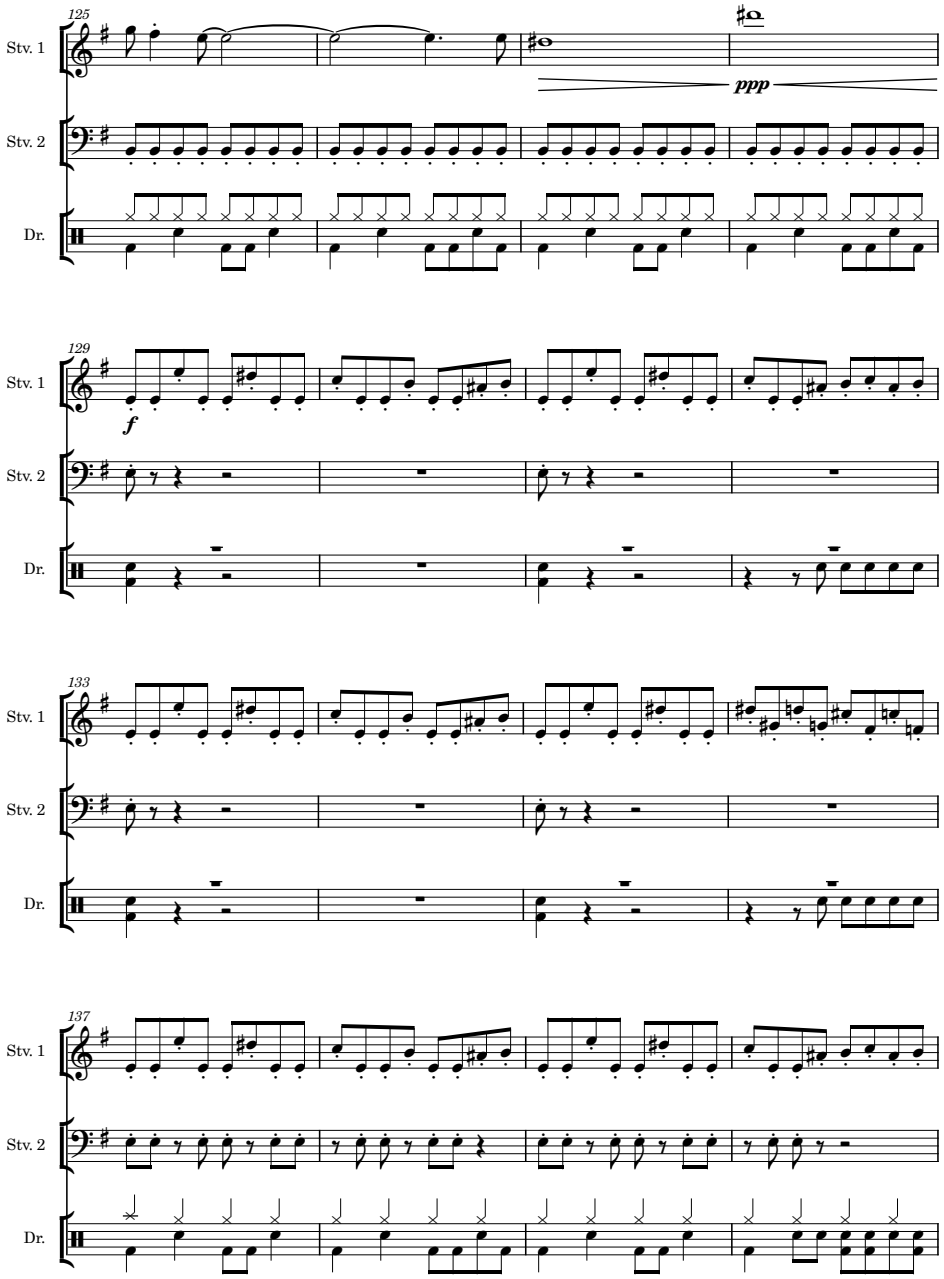



61
Stv. 1
Stv. 2
Dr.
65
Stv. 1
Stv. 2
Dr.
69
Stv. 1
ppp
Stv. 2
Dr.
73
Stv. 1
f
Stv. 2
f
Dr.
f


77
Stv. 1
Stv. 2
Dr.
81
Stv. 1
Stv. 2
Dr.
85
Stv. 1
Stv. 2
Dr.
89
Stv. 1
Stv. 2
Dr.
ff
ff
ff
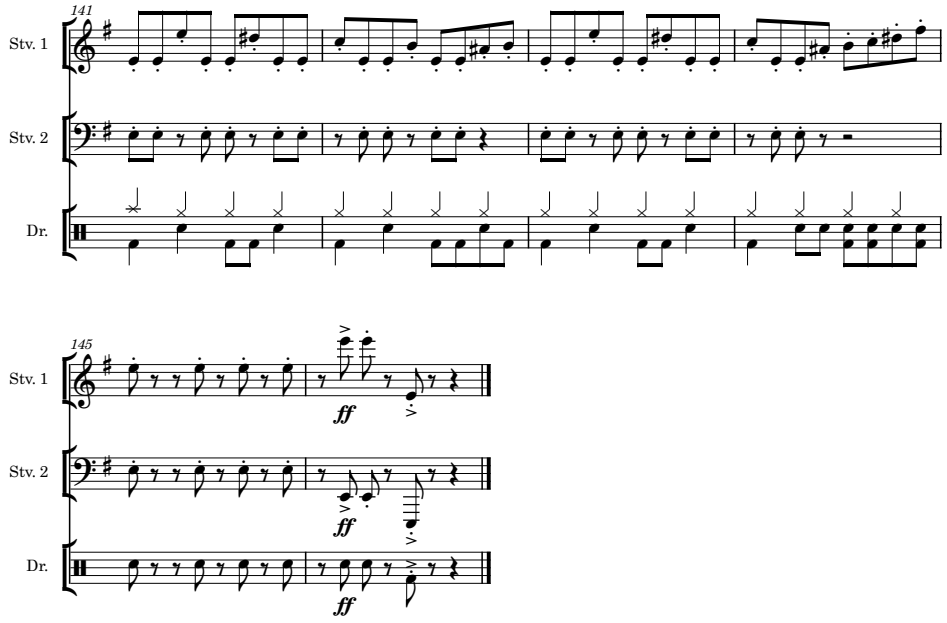

93
Stv. 1
Stv. 2
Dr.
f
97
101
105

8 Virtual Music Instrument




125
Stv. 1
Stv. 2
Dr.
ppp
129
f
133
137

10 Virtual Music Instrument



K Yurndu akanatha

K.1 Virtual Music Shorthand Notation

```
[B=120;] 13^s2 20^s 17^s 0q 13^s 20^s 17^s("1") 0s
0b 0b
13^s 20^s 17^s 0c 17^c .17^q .17^q .16^q .16^q .15^q .15^q 13^s 20^s
   17^s 0c 17^c 13^s 20^s 17^s("2")
0b 0b
23s3 24s4 .25q5 .21q .23q4 .19q .21q3 .17q 18s3 17s 16s2 15s
   14s1("3")
0b 0b
13^s2 20^s 17^s 0c 17^c 17^q 18^q 17^q 18^q 17^s 18^s 17^s 18^s 13^s
   20^s 17^s("4")

0b 0b 0b 0b

>21q4>2 .20q >21q4>2 .20q 0m .13q >21q4>2 .20q("5")
0b 0b
13^s2 20^s 17^s 0c 17^c 17^q 18^q 17^q 18^q 17^s 18^s 17^s 18^s 13^s
   20^s 17^s
0c 5c 5q 6q 5q 6q 5s 6s 5s 6s 1s 8s 5s("6")
0b 0b
>21q4>2 .20q >21q4>2 .20q >21q4>2 .20q >9q4>2 >.8q 0m >21q4>2
   .20q("7")
!>9vvq4>2 !>.8vvq 0q !.13vvvq !>9vvq4>2 !>.8vvq
0b 0b
13^c4 20^c 17^m-b>1 0b 0b 13c2 20c 17m-b>1("8")
0b
.1vvq .8vvq .1vvq .8vvq .1vvq .8vvq .1vvq .8vvq 0c .1vvq .8vvq 0q
   .1vvq .8vvq 0c !.1vvvq !.8vvvq
0b
.9vvq .4vq .9vvq .4vq .9vvq .4vq .9vvq .4vq  0c .9vvq .4vq 0q .9vvq
   .4vq 0c !.4vvq !.9vvvq

0b 0b 0b 0b

0c{1vvb-b3} 0c{8vvm.-b3} 17vvm-b3("9")
0c{9vvb-b} 0c{4vm.-b} 15vm 13vm 15vm
0c{1vvb-b3} 0c{8vvm.-b3} 17vvm-b3("10")
0c{9vvb-b<6} 0c{4vm.-b<6} 15vm<6{13v} 16vm{13v} 18vm<6{15v<6}

0c{1vvb-b6} 0c{8vvm.-b6} 17vvm-b6("11"){25v}
0c{9vvb-b} 0c{4vm.-b} 15vm{25m-b} 16vm 18vm
```

```
0c{1vvb-b6} 0c{8vvm.-b6} 17vvm-b6{20v}
0c{9vvb-b} 0c{4vm.-b} 15vm{20m-b} 15vc 13vc 18vm

0c{1vvb-b} 0c{8vvm.-b} 13c{17vvm-b} 20c 17m. 15c
15m{9vvb-b} 13m-b{4vm-b}
0c{1vvb-b} 0c{8vvm.-b} 13c{17vvm-b} 20c 17m. 15c
.15m{.9vv} 15m{13} 16m<7{13} 18m<8{15}

0c{1vvb-b} 0c{8vvm.-b} 13c{17vvm-b} 20c 17m. 15c
15m{9vvb-b} 13c-b{4vm-b} 20vs 18vs 21vc
0c{1vvb-b} 0c{8vvm.-b} 13c{17vvm-b} 20c 17m. 15c
.15m{.9vv} 15m{13} 16m{13} 18m<9("12"){15}

0c{4vvb-b} 0c{11vvm.-b} 16c{20vvm-b} 23c 20m. 18c
18m{12vvb-b} 16c-b{7vm-b} 23vs 21vs 24vc
0c{1vvb-b} 0c{8vvm.-b} 13c{17vvm-b} 20c 17m. 15c
.15m>7{.9vv} 15m>5{13} 16m>3{13} 18m>1{15}

0b 0b

13^s2 20^s 17^s("13")
0b
0c{1vvb-b3} 0c{8vvm.-b3} >21c>2 .20c >21c4>2 .20c >21c4>2 .20c("14")
   0c
0b 0b
13^s2 20^s 17^s("15") 0m 17^c 17^q 18^q 17^q 18^q 17^s 18^s 17^s
   18^s 13^s 20^s 17^s("16")
0b 5c 5q 6q 5q 6q 5s 6s 5s 6s 1s 8s 5s("17")
0b
0c{9vvb-b<6} 0c{4vm.-b<6} 15vm<6{13v} 16vm{13v} 18vm<8{15v<8}
25vq("18"){17vq}
```