CJing: Combining Live Coding and VJing for Live Visual Performance

by

Jack Voldemars Purvis

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the requirements for the degree of
Master of Science in Computer Graphics.

Victoria University of Wellington
2019

Abstract

Live coding focuses on improvising content by coding in textual interfaces, but this reliance on low level text editing impairs usability by not allowing for high level manipulation of content. VJing focuses on remixing existing content with graphical user interfaces and hardware controllers, but this focus on high level manipulation does not allow for fine-grained control where content can be improvised from scratch or manipulated at a low level. This thesis proposes the *code jockey* practice (*CJing*), a new hybrid practice that combines aspects of live coding and VJing practice. In CJing, a performer known as a *code jockey* (*CJ*) interacts with code, graphical user interfaces and hardware controllers to create or manipulate real-time visuals. CJing harnesses the strengths of live coding and VJing to enable flexible performances where content can be controlled at both low and high levels. Live coding provides fine-grained control where content can be improvised from scratch or manipulated at a low level while VJing provides high level manipulation where content can be organised, remixed and interacted with. To illustrate CJing, this thesis contributes *Visor*, a new environment for live visual performance that embodies the practice. Visor's design is based on key ideas of CJing and a study of live coders and VJs in practice. To evaluate CJing and Visor, this thesis reflects on the usage of Visor in live performances and feedback gathered from creative coders, live coders, and VJs who experimented with the environment.

Acknowledgments

This thesis has taken me on an incredibly inspiring and fulfilling journey that would not have been possible without the input of many people. I wish to thank everyone who has been involved and in particular the following individuals and organisations:

To my research supervisors Craig Anslow and James Noble for their support, encouragement and guidance throughout this thesis. To Craig for introducing me to the world of live coding and providing me with many networking opportunities. To James for his performance based outlook and constructive feedback.

To members of the University who provided me with support and guidance throughout this thesis. In particular, to the HCI Group and Computer Graphics Research Group. To my office mates. To the School of Engineering and Computer Science administration staff. To Diana Siwiak for her advice and hosting the productive writing sessions. To Victoria University of Wellington for the Victoria Masters by Thesis Scholarship. To the Faculty of Science for the Faculty Strategic Research Grant that funded my travel to present at the ICLC conference in Madrid. Also to Craig Anslow for the additional funding that went towards this travel.

To all of the interviewees that gave up their valuable time to contribute to this thesis. Also to all of those who have started using Visor and provided me with valuable feedback.

To Optimal Workshop for providing me with flexible employment throughout my studies. To my colleagues for their support and understanding, in particular when I needed to take leave to travel or focus on this thesis. Also for providing me with the opportunity to pick up skills in user research, development, and design.

To Art∼Hack for providing me with a space to test Visor and launch my own performance practice. To all of the members for their support and feedback towards my work. In particular, to Mikey Williams for leading the organisation of the meetups, exhibitions, and concerts. Also to Matt McKegg for collaborating with me and sharing his experiences building software for live performances.

To my new friends that I met as part of the live music and visual community in New Zealand. I am grateful to have had the opportunity to work alongside you and look forward to future events. In particular, to Daniel Aston for the insightful chats on live coding and VJing practice. To Alexia George for her inspiring work which played an active role in motivating this thesis. To Léo Podechard for collaborating with me. To Darrell Smith for sharing his VJing experiences with me and supporting my practice.

To everybody I met as part of my travels for the ICLC conference in Madrid and other events in Barcelona, London, and Berlin for their support, critique, and encouragement towards my work. The live coding community was incredibly friendly and I enjoyed getting to meet and perform alongside many of you. These people include Ryan Kirkbride, Kofi Oduro, Neil Smith, Charlie Roberts, Shawn Lawson, Sam Aaron, Dimitris Kyriakoudis, Evan Raskob, Ulysses Popple, Marianne Teixido, Emilio Ocelotl, Lina Bautista, Ivan Paz, Olivia Jack, Alexandra Cárdenas, Diego Dorado, and many others. Also to Alicia Champlin and Niklas Reppel for kindly hosting me in Barcelona.

To my friends who supported and encouraged me throughout this journey. In particular, to Robbie for the late night jam sessions and joining me on stage. To Orion for landing me that first underground gig. To Ben for sharing this academic journey with me. To Kelly for introducing me to the VJing scene and many key people in Wellington, I don't think I could have achieved much of this without your input.

To my family and close family friends. To Mum, Dad and Tara for their unconditional love, support and understanding throughout this journey. You each show such passion and dedication to life. Your influence has shaped me into who I am today and enabled me to tackle great challenges such as this.

Publications

This thesis covers research that was published in a conference paper:

> **Jack Purvis**, Craig Anslow, and James Noble. CJing Practice: Combining Live Coding and VJing. In Proceedings of the International Conference on Live Coding (ICLC), Madrid, Spain, 2019.

Figures

All photographs, diagrams and other figures in this thesis are authored by myself unless another source is explicitly credited or cited.

Contents

1 Introduction **1**
1.1 Research Questions . . . 3
1.2 Research Methodology . . . 4
1.3 Research Contributions . . . 4
1.4 Outline . . . 5

2 Background **7**
2.1 Creative Coding . . . 7
2.2 Live Programming . . . 10
2.3 Live Coding . . . 12
2.4 VJing . . . 17
2.5 Summary . . . 20

3 Live Coding and VJing: Interviews **23**
3.1 Interview Procedure . . . 24
3.2 Participants . . . 24
3.3 Analysis . . . 25
3.3.1 Features . . . 27
3.3.2 Interactions . . . 30
3.3.3 Practice . . . 32
3.4 Discussion . . . 36
3.5 Summary . . . 39

4 CJing and Visor **43**
4.1 Code Jockey Practice 45
4.1.1 Context 46
4.1.2 Key Ideas 47
4.2 Visor: Design 48
4.3 Visor: Features 50
4.3.1 Live Coding 50
4.3.2 State Management 52
4.3.3 Layers 53
4.3.4 Fast Fourier Transform 55
4.3.5 Tap Tempo 57
4.3.6 MIDI 58
4.3.7 Other 60
4.4 Visor: Composing Layers 61
4.4.1 Model Layer 62
4.4.2 Particles Layer 63
4.4.3 Mask Layer 66
4.4.4 MIDI Control 69
4.4.5 Final Composition 69
4.5 Visor: Implementation 72
4.5.1 Architecture 72
4.5.2 Client 74
4.5.3 Server 74
4.5.4 Live Coding 75
4.5.5 Handling State 76
4.5.6 Integration with Processing 76
4.6 Visor: Development Approach 78
4.7 Discussion 81
4.8 Summary 84

5 Live Performances with Visor **87**

5.1 Performance Setup 89
5.2 Performance Approach 92
5.3 Visor in Action 93
5.3.1 Live Coding 94
5.3.2 State Management 95
5.3.3 Layers 96
5.3.4 MIDI 98
5.3.5 Fast Fourier Transform 102
5.3.6 Tap Tempo 103
5.4 Summary 105

6 Visor Feedback Survey **107**
6.1 Survey Procedure 108
6.2 Participants 108
6.3 Results 109
6.3.1 Usage 110
6.3.2 Learning 111
6.3.3 Live Coding 112
6.3.4 State Management 112
6.3.5 Layers 114
6.3.6 Fast Fourier Transform 115
6.3.7 Tap Tempo 116
6.3.8 MIDI 117
6.3.9 Interface 118
6.3.10 Suggested Improvements 118
6.4 Discussion 119
6.5 Summary 124

7 Conclusions **127**
7.1 Research Contributions 130
7.2 Future Work 130

Appendices **135**

A Human Ethics Documents **137**

B Interview Information Sheet **149**

C Interview Schedule **153**

D Feedback Survey Information Sheet **155**

E Feedback Survey Questionnaire **159**

List of Figures

2.1 Abstract pattern coded in Processing [27]. 8
2.2 Tree fractal coded in VVVV [45]. 9
2.3 Light Table's real-time visualisation of program state in-line with the associated code [13]. 11
2.4 Chemical Algorave, Newcastle 2017 [74]. 13
2.5 Live coding visuals in LiveCodeLab [15]. 14
2.6 Live coding music and shaders in Gibber [9]. 15
2.7 Live coding music and visuals in Praxis LIVE [26]. 16
2.8 VJ setup featuring PCs, a MacBook, and MIDI controllers [63]. 18
2.9 Taniwha's Den 2019 music festival. 19

3.1 Thematic map for the *Features* theme. 28
3.2 Thematic map for the *Interactions* theme. 30
3.3 Thematic map for the *Practice* theme. 32

4.1 Visor interface in action. 44
4.2 Visual output rendered from Visor based on Processing. . . . 45
4.3 CJing in the context of the broader subject areas that formulate it. 46
4.4 Live coding in Visor. 51
4.5 State management in Visor. 53
4.6 Layers in Visor. 54
4.7 FFT in Visor. 56

4.8 Tap tempo in Visor. 57
4.9 MIDI in Visor. 59
4.10 Visor Learn Hub. 60
4.11 Model layer rendered independently. 62
4.12 Code for the Model layer. 63
4.13 Particles layer rendered independently. 64
4.14 Code for the Particles layer. 65
4.15 Particles layer rendered independently. 66
4.16 Code for the Mask layer. 67
4.17 Mask layer rendered independently. 68
4.18 Mask layer rendered independently. 68
4.19 MIDI controller annotated with the names of the mapped parameters and MIDI variables used by the code for the Model, Particles, and Mask layers. 69
4.20 Model and Mask layers composited together. 70
4.21 Model and Particles layers composited together. 71
4.22 Model, Particles, and Mask layers composited together. . . . 71
4.23 Model, Particles, and Mask layers composited together. . . . 72
4.24 Visor's client-server architecture. 73
4.25 Traditional approach to handling the matrix stack with the Processing API in Visor. 77
4.26 New approach to handling the matrix stack using a method that accepts a block as an argument in Visor. 77
4.27 Art∼Hack meetup where Visor was often tested in a live performance context. 79
4.28 Maker Faire Wellington where Visor was demonstrated to the public as part of the Art∼Hack stall. 80

5.1 My typical setup for live performance with Visor. 89
5.2 ICLC 2019 Algorave performance. 90

5.3 Taniwha's Den 2019 Cliff performance where Visor was used to render visuals that were projected onto a large limestone cliff face. 91
5.4 Limestone cliff face that was used as a projection surface during the Taniwha's Den 2019 festival. 91
5.5 Taniwha's Den 2019 Mainstage performance. 92
5.6 TOPLAP 15th Birthday Livestream performance. 94
5.7 Eyegum Wednesdays performance. 96
5.8 Rendered visuals from the Taniwha's Den 2019 Mainstage performance. 97
5.9 Vertigo gig performance. 102
5.10 Rendered visuals from the Burrowing Pufferfish Party performance. 104

List of Tables

2.1 Taxonomy presenting how the related software spans creative coding, live programming, live coding, and VJing practices. 21

3.1 Interview participants background. 26
3.2 Important features, interactions, and aspects of the participant's performance setup and practice. 40

4.1 Visor's core features and how they map to the three key ideas of the CJing practice. 50

5.1 Performances conducted throughout this thesis. 88

6.1 Feedback survey participants background, estimated time spent using Visor, and context in which they might use Visor. 109

Chapter 1

Introduction

The creation of visuals to accompany music is an essential part of any audiovisual experience. Live coding and VJing (video jockey practice) are both live performance practices that offer the ability to improvise and manipulate visuals that sync with music in real-time. Live coding is the application of live programming techniques to the performing arts [79]. Live coding offers the ability to improvise content such as generative music or visuals by creative coding over the course of a performance [57]. Video jockeys (VJs) are the visual counterpart to the musical disc jockeys (DJs); improvising visuals to accompany live music and create audiovisual marriages that engage the senses [61]. VJs tend to work with pre-rendered content such as looping video clips and conduct their performances by layering and mixing clips together or applying parameterised video effects.

Live coding focuses on writing code to improvise or manipulate content, providing fine-grained, low level control of the final output. VJs instead focus on interacting with comprehensive graphical user interfaces (GUIs) and hardware controllers to improvise or manipulate visuals, providing overarching, high level control of the final output. This thesis proposes the

code jockey practice (*CJing*), a new hybrid practice that combines aspects of live coding and VJing. In CJing, a performer known as a *code jockey* (*CJ*) interacts with code, GUIs, and hardware controllers to improvise or manipulate visual content in real-time. CJing harnesses the strengths of live coding and VJing to enable flexible performances where content can be controlled at both low and high levels.

This thesis claims that combining live coding and VJing can simultaneously remove limitations identified in each practice. Live coding environments focus solely on textual interfaces. All interactions with the content must be conducted at a low level by modifying the code in these textual interfaces. This limits a live coder's ability to interact with content, for example, parameters in the code can only be assigned to discrete values [67] that do not allow for easy exploration of the parameter space. In addition, text editors do not offer visualisation of the underlying program state to improve comprehension of the live code. While the live coding community [41] focuses on textual interfaces as an aesthetic choice, this choice impairs the usability of the practice. These limitations can be removed by incorporating features of VJing software into a live coding environment. VJing software offers GUIs made up of many parameter controls, tools to organise content such as layers, and provides mappings to external hardware such as MIDI controllers. These features can be used to abstract upon live code and provide high level interactions to manipulate parameters or visualise the state of the underlying program.

VJing software focuses on working with pre-rendered content such as video clips that can only be manipulated by configuring the speed and direction of playback or by applying video effects. All interactions with the content must be conducted at a high level through graphical user interfaces or hardware controllers. This limits VJs as they do not have fine-grained control over the video content itself [51]. In addition, VJs cannot improvise content from scratch during a performance and must rely on

their existing material. These limitations can be removed by incorporating live coding within VJing software. Live coding can be used as a method to provide fine-grained control of content through the creative coding of visuals. Generative content can be improvised from scratch or manipulated at a low level during a live performance by live coding.

The goal of this thesis is to explore how CJing can combine live coding and VJing to harness the strengths of both practices while simultaneously removing limitations identified in each practice. This thesis demonstrates CJing by contributing a new environment called *Visor* that embodies the practice. Visor has been purpose-built following a user-centered, practice-based approach to offer features of both live coding and VJing to enable live visual performances. This research involved interviewing live coders and VJs to better understand their practice, the development of Visor, and the evaluation of Visor through live performances and an online feedback survey.

1.1 Research Questions

The goal of this thesis can be articulated into three research questions:

RQ1: What are the needs and expectations of performers who practice live coding and VJing?

RQ2: Can features of VJing software improve the usability of a live coding performance?

RQ3: Can live coding be used as an effective method for improvising visual content or manipulating existing content during a VJ performance?

1.2 Research Methodology

The underlying methodology of this research is user-centered design [49]. To begin, seven live coders and VJs were interviewed to better understand their needs and expectations. A thematic analysis [55] of the interview results was applied to explore the prominent ideas of the performers' practice and inform the design of the new environment, *Visor*. Visor was then developed using a practice-based approach (§4.6) that involved regular testing of the environment in a live performance context. Two methods were then used to evaluate Visor's effectiveness. The first evaluation method was to reflect on my own live performances with Visor as part of the practice-based approach. The second evaluation method was to analyse the results of an online feedback survey that collected responses from six creative coders, live coders, and VJs who had experimented with Visor.

1.3 Research Contributions

This thesis makes the following research contributions:

- The **CJing practice**, a new hybrid practice that combines live coding and VJing.
- **Visor**, a new environment for live visual performance that demonstrates the CJing practice by combining features of live coding and VJing software.
- An **evaluation** of Visor, based on a reflection of my own live performances conducted throughout this thesis and the results of an online feedback survey completed by six creative coders, live coders, and VJs.

1.4 Outline

The remainder of this thesis is as follows:

Chapter 2 summarises the related work in creative coding, live programming, live coding, and VJing.

Chapter 3 presents a study of live coders and VJs in practice, based on interviews that identified important features, interactions, and aspects of their performance practice.

Chapter 4 provides an in-depth description of the proposed CJing practice and presents the Visor environment including details about the environment's design, important features, implementation, development approach, and comparison with related work.

Chapter 5 reflects on my usage of Visor in live performances including details of my performance setup, performance approach, and a discussion of observations I made from the performances.

Chapter 6 presents a study of creative coders, live coders, and VJs who evaluated Visor and provided feedback through an online survey.

Chapter 7 concludes this thesis by discussing the overarching strengths and limitations of Visor and the underlying CJing practice along with opportunities for future work.

Chapter 2

Background

This chapter provides an overview of work related to this thesis: First, the practice of creative coding and related tools is described; second, the practice of live programming and related applications is described; third, the practice of live coding and related environments is described; fourth, the practice of VJing and related tools is described; finally, the related work is summarised and placed in the context of this thesis.

2.1 Creative Coding

Creative coding is the artistic application of programming to create content [52]. Unlike traditional artistic methods that make use of physical instruments like paint brushes or musical devices, creative coding harnesses algorithms and computational techniques to generate outputs where distinct snippets of code can produce different outputs. Like other art forms, creative coding can be used to generate content such as static artworks, animations, music, posters, visualisations, and more. The practice of creative coding is different from conventional software engineering where the aim is to develop a well thought out solution. Instead, the programmer

acts as an artist might and sketches to quickly iterate on variations of a loosely defined idea [50]. Sketches can be quickly created, iterated upon, or discarded depending on if they meet the programmer's intent. Creative coding practice is continually developing. For example, the concept of code bending shows how existing creative coding languages can be adapted such that programs behave as plugins and the output of each can be dynamically combined to create a single output [52].

Processing is a creative coding language for the visual arts used by artists, designers and students with the intent to teach the fundamentals of computer programming [72]. Processing uses a sketch based model, allowing the programmer to quickly prototype ideas and iterate on them for creative exploration. Figure 2.1 demonstrates what it looks like to program in Processing. Some creative coding tools and libraries also offer similar

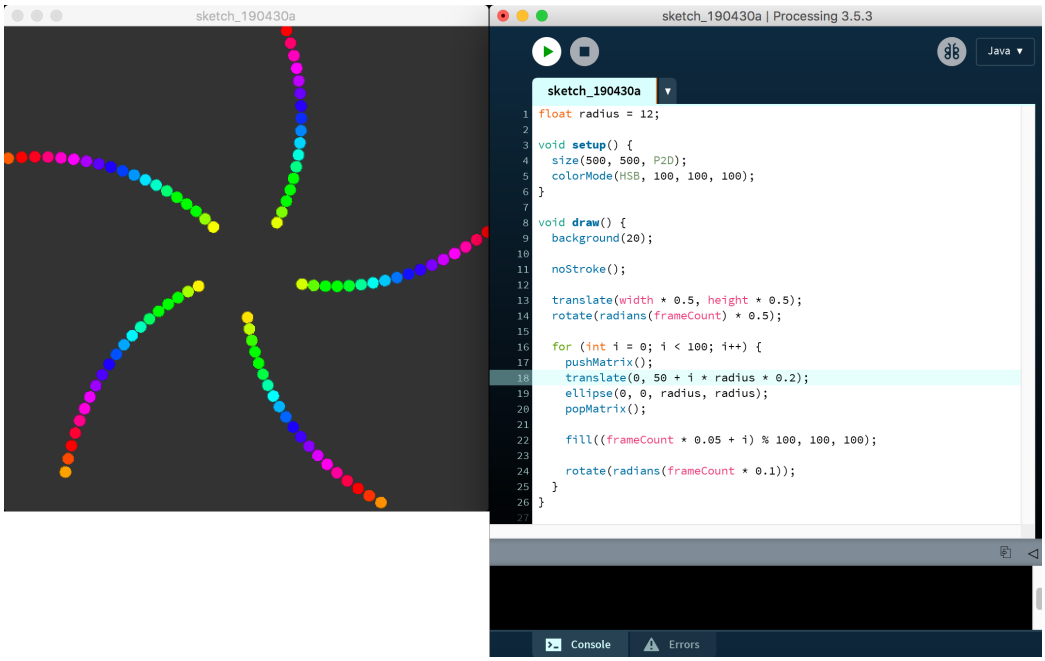

Figure 2.1: Abstract pattern coded in Processing [27]. The Processing development environment (right) displays the code used to render the visuals (left).

APIs (application programming interface) to Processing but in different languages such as p5.js [25] for the web or OpenFrameworks [23] for C++ applications. Another adaption is JRubyArt [11] which wraps Processing to enable creative coding in the Ruby language. While these languages focus on textual creative coding, others utilise visual programming, otherwise known as patching. Patch-based tools produce sound or visuals based on interconnected networks of nodes. Examples of patch-based creative coding tools include Max/MSP/Jitter, Pure Data, and VVVV [19, 28, 46]. A VVVV patch to generate a fractal tree is shown in Figure 2.2.

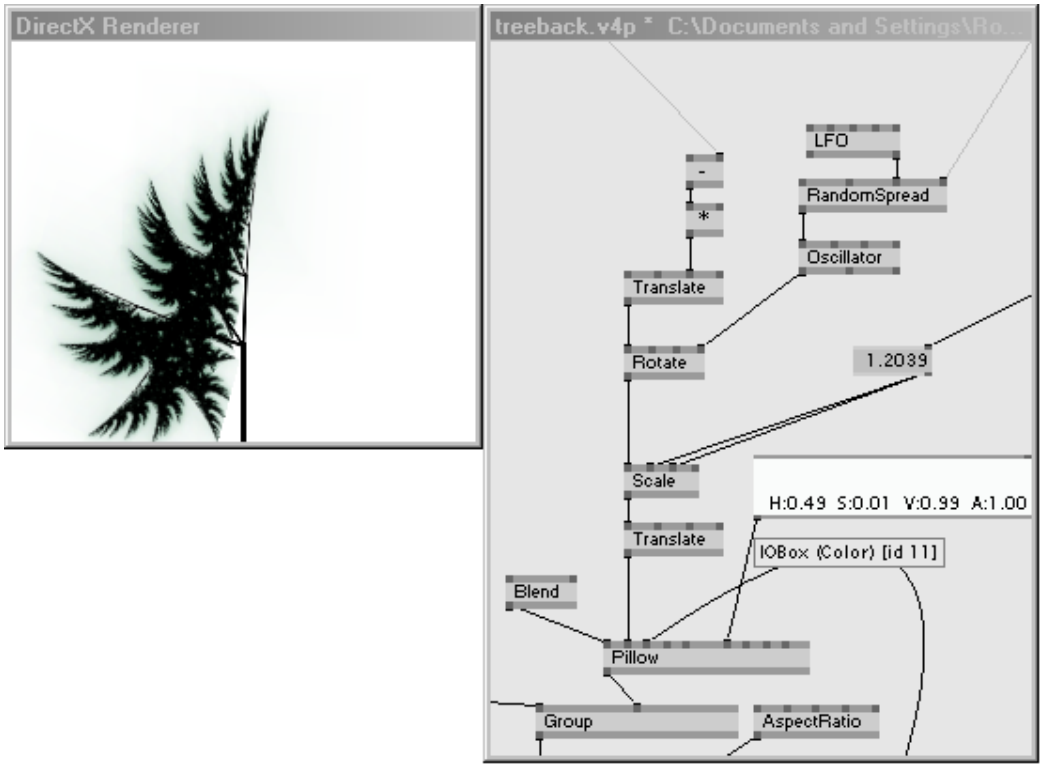

Figure 2.2: Tree fractal coded in VVVV [45]. The VVVV patch (right) instructs the rendered visuals (left).

2.2 Live Programming

Live programming [79] is a software engineering practice that manifests itself in many forms including interactive debuggers, interactive programming environments for learning, integrated development environment (IDE) inspectors and interpreter environments such as a Read-Eval-Print-Loop (REPL). Live coding is also one of the practical uses of live programming, helping to offer new insights into software engineering processes [53]. Live programming is mainly concerned with the notion of liveness [79], described as when a program exhibits a minimal latency between a programming action occurring and seeing its effect as well as allowing for dynamic manipulation of the program code at runtime. The program development cycle normally involves editing, compiling and running a program. In live programming the cycle is fundamentally different, the program instead continuously runs even while edits occur to its source code [79].

The Smalltalk language and environment [33] is an example of software that embodies live programming. In Smalltalk, the state of the code and every object is inspectable and editable at runtime within graphical user interfaces. It has been observed that developers frequently manipulate the state of a program when it is accessible through a graphical user interface (GUI) [66]. It has also been discussed how even though a user interface can give immediate control of some state, the underlying code still presumes a fixed delimitation of what can be changed at runtime [76].

A key motivation of live programming is making the act of programming more comprehensible and accessible by incorporating liveness [73]. The visualisation and manipulation of the state are discussed in the context of coding visuals using Processing [81]. By exposing the state of variables in a piece of code it can be better understood by the programmer. If the state is tangible or visualised over time the programmer can create a connection

between the output of the program and the code itself. One IDE that incorporates this kind of state visualisation is Light Table [14]. Light Table features a 'live document' where all code typed is evaluated immediately and the state of the program is monitored at runtime for effective real-time debugging. Figure 2.3 shows the Light Table IDE where the state of variables are presented next to the associated code at runtime.

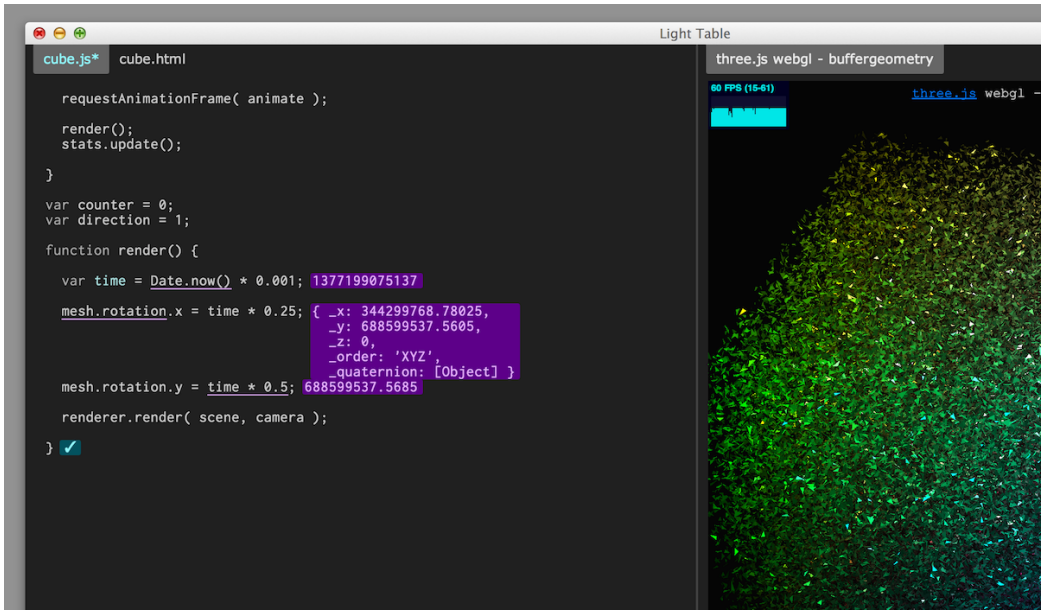

Figure 2.3: Light Table's real-time visualisation of program state (highlighted) inline with the associated code [13].

Incorporating direct manipulation of program output is an interesting extension of live programming. This behaviour has been seen in Circa [62], a purpose-built language and environment for live coding where the state of the running program is available during the editing process. Circa is a dataflow-based language where a program is represented as a directed graph of terms and each term is a function that takes in inputs to produce a single output. Circa features a hybrid textual and visual approach to editing and inspecting the code and state at runtime. Circa is demonstrated by drawing elements to a canvas and allowing direct manipulation

of each element by clicking and dragging on them in the canvas. The piece of code that is responsible for the clicked element is identified using a backwards propagation technique and visualised. Direct manipulation is also shown by Palimpsest [54], a purely visual language for image processing. Palimpsest aims to bridge paradigms seen in image editors and programming languages in a single environment. While minimising the separation between editing and runtime, Palimpsest also provides performance capabilities for live coding such as initialising colours to random values as a starting point for creative exploration.

2.3 Live Coding

Live coding [41] is the practice of live programming techniques in the performing arts [79]. The notion of liveness in live coding is motivated by the ability to support programmers to produce effects in the real world through computation [73]. These effects are often in the form of musical or visual content, similar to the outputs of creative coding, except live coding offers the ability to improvise content over the course of a performance [57]. The output evolves over time as the live coder modifies the code. Live coding lends itself to performance as shown by the *algorave* movement [56] where artists produce electronic music and visuals for the enjoyment of a live audience. It is common practice at algoraves to project the source code onto a screen as shown by Figure 2.4, this way the audience can observe how changes to the code affect the musical and visual outputs. The projection of live code enables the audience to engage with the performance in a new way other than just dancing to the music or watching the visuals. Live coders have expressed how they feel their musical style is encoded in the code they write [70], aligning with the idea that live coding illuminates the way in which programming can be an artistic practice [53].

Figure 2.4: Chemical Algorave, Newcastle 2017 [74].

There are a number of live coding environments for producing music including Chuck, TidalCycles, and Sonic Pi, each using domain-specific languages [3, 47, 69]. In particular, Sonic Pi is built off the Ruby language with the intent of being used by musicians as well as to teach programming [47]. Ruby has a relatively concise syntax but powerful meta-programming abilities. Sonic Pi utilises these traits to support features like multiple instruments (which require concurrency) with a simple block-based syntax, demonstrating how Ruby can make even a complicated programming paradigm relatively easy to grasp. Sonic Pi illustrates how live coding plays an important role in computing education [53].

In terms of live coding for visuals, there are a number of environments. LiveCodeLab [60] is a web-based environment that allows for creative coding in a domain specific language. Figure 2.5 shows LiveCodeLab in action. In LiveCodeLab each frame is rendered purely as a function

of time and the frame number with no other state, a deliberate choice to avoid having to maintain data structures when the code changes. LiveCodeLab also exhibits a transient behaviour when typing code as the program is updated on every keystroke, causing the output to exhibit sudden changes. This behaviour is intended to be part of the constructive nature of the performance. Interestingly, one live coding environment that exhibits the same transient behaviour as LiveCodeLab has been discussed with respect to performative strategies that help to avoid unintended behaviour or errors when live coding GLSL shaders [67]. A similar live coding environment to LiveCodeLab is Cyril [4]. Cyril uses a similar domain-specific language to LiveCodeLab but instead resides as a desktop application.

Figure 2.5: Live coding visuals in LiveCodeLab [15].

Another web-based live coding environment for producing sound and visuals is Gibber [75]. Gibber uses the general purpose language JavaScript for live coding and favours high level abstraction for enabling creative ex-

pression. Such abstractions allow features such as multi-modal mappings between audio and visual elements as well as methods for scheduling events or sequencing data. In Gibber, specific lines of code can be executed from the editor allowing REPL like behaviour where arbitrary code can be evaluated at any time. Gibber objects are stored in a graph that holds state between code iterations. 2D graphics are available through access to HTML canvas objects and 3D graphics are implemented by wrapping the Three.js library [39]. The ability to live code GLSL shaders to define visuals is also possible in Gibber, an approach taken by other live coding systems such as KodeLife [12]. Figure 2.6 shows the live coding of music and shaders in Gibber.

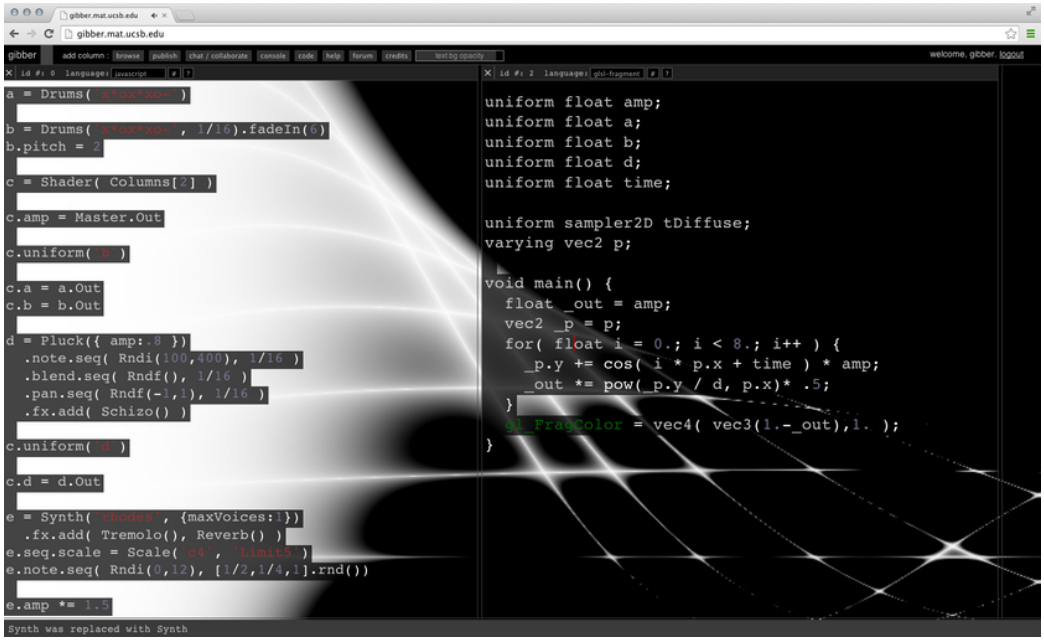

Figure 2.6: Live coding music and shaders in Gibber [9].

Praxis LIVE [78] is a live coding environment and IDE for producing music and visuals. Praxis LIVE presents a hybrid approach by supporting visual programming of a graph of interconnected components where each component also has a textual representation that can be live coded at runtime.

Figure 2.7 shows Praxis LIVE in action. In Praxis LIVE, programming is performed in Java, allowing access to a multitude of libraries including the Processing API for 2D and 3D creative coding. Praxis LIVE is based on a declarative model where a component's code represents its state. Praxis LIVE also offers GUIs to interact with and visualise the state of components. Similar to Praxis LIVE, Siren [80] is a hybrid system for the composition of algorithmic music and live coding performances that offers GUIs to interact with code. Siren enables live coding of music at a low level through the TidalCycles language and makes use of many GUI features to interact with content at a high level. Examples of these features include the pattern roll that visualises the pattern of a function in TidalCycles code or the scene interface that enables organisation of many pattern functions.

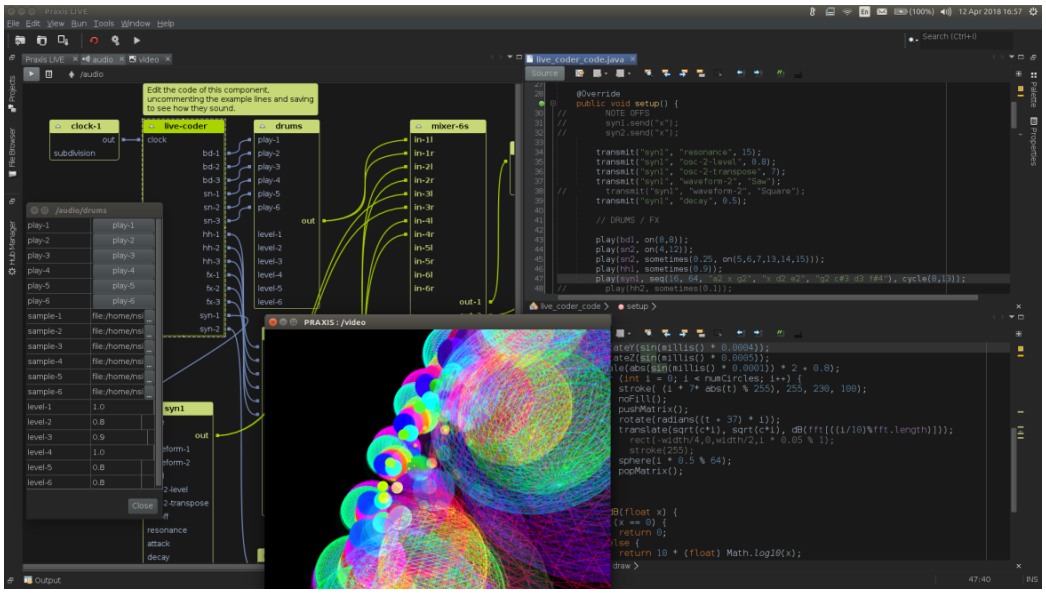

Figure 2.7: Live coding music and visuals in Praxis LIVE [26]. Visual programming is shown (left) along with live coding (right), and the rendered visual output (bottom-middle).

While the aforementioned live coding environments predominantly focus on textual interfaces, new developments in live coding practice explore how gestural and embodied interaction can be incorporated. For example, Auraglyph [77] is a programming environment that allows for gestural interactions in musical performance. Auraglyph achieves this by making use of touch interfaces to interact with visual objects that program the musical output, distancing itself from textual interfaces that are typical in live coding environments. Visual programming environments such as Max/MSP/Jitter and VVVV are also often considered for 'live patching' performances. These performances are similar to live coding except the content is produced using patches instead of lines of code.

2.4 VJing

Video jockeys (VJs) are the visual counterpart to the musical disc jockeys (DJs); improvising visuals to accompany live music and create audiovisual marriages that engage the senses [61]. VJs tend to work with pre-rendered content such as looping video clips and improvise their performances by layering and mixing clips together or applying parameterised video effects. Unlike live coding software, VJing software offers comprehensive GUIs made up of many individual parameter controls, video clip preview windows and mappings to external hardware. In combination with software, VJs utilise hardware such as MIDI controllers, touch interfaces and other miscellaneous devices such as game console controllers to produce and manipulate their visuals. Figure 2.8 shows a VJ setup with multiple computers and MIDI controllers.

VJs make use of industry standard software such as Resolume and Modul8 [21, 30] for live visual performance. A variety of content can be used as input in these software packages including video clips, camera feeds, images and even real-time visual effects. This content is manipulated using filters and special effects as well as being mixed with other content to produce a final image. The resulting image is projected for the audience

Figure 2.8: VJ setup featuring two PC's running Resolume (top-left, top-right), a MacBook running video editing and patching software for supplemental content (bottom-left), and numerous MIDI controllers (bottom-right) [63].

to see using projectors, LED walls, and digital displays. An example of this VJ pipeline is used in a performance installation where the output of an interactive Processing sketch is used as an input into Resolume and then to MadMapper [17] to be projection mapped onto a surface [65]. The Syphon and Spout frameworks [32, 36] are often used by VJs to stream visuals in real-time between applications. Patch-based software such as Touch Designer and VVVV are also commonly used to generate real-time visual content [42, 46]. Figure 2.9 shows another example where visual content is rendered by a VJ and projected onto multiple screens at a music festival.

Figure 2.9: Taniwha's Den 2019 music festival. A VJ (left) is working with Resolume to render visuals that are projected across multiple screens around the DJ booth using multiple projectors.

One study of VJing practice from an HCI perspective observed performers' methods of expressive interaction and identified a number of characteristics that had an effect on their practice [64]. The observations made in this study have influenced the design of a number of customised audiovisual performance tools such as residUUm and ABP [58, 71]; developed following a user-centered approach [59]. While these tools succeed in delivering ready-made generative visuals that are straightforward to interact with, they are not easily reused in other performances. In contrast, Bergström et al. [51] suggest that code should be organised into reusable modules that can be shared among programmers and even non-programmers to allow for new performance compositions. This concept is demonstrated by Mother [51], a set of tools that act as a middleware between Processing sketches to enable VJ performance. Mother allows for the composition of multiple sketches that can be shared as reusable code modules. Each

sketch exposes parameters that Mother can interact with over OSC (Open Sound Control). Mother is also discussed as an example of the aforementioned code bending practice [52].

2.5 Summary

This chapter has described the practices of creative coding, live programming, live coding, and VJing. There are a number of software applications and environments that span these practices, but no software has been purpose-built to offer features from each practice to support live performance of visuals. In particular, no related work explicitly combines aspects of live coding and VJing to harness the strengths of both practices, addressing RQ2 and RQ3 (§1.1). Table 2.1 displays a subset of the related software that best overlaps these practices.

This thesis builds upon the gap in the related work by contributing a new purpose-built environment called Visor. Visor combines aspects of both live coding and VJing to embody the CJing practice. Visor also incorporating aspects of creative coding and live programming practices.

The next chapter presents a study of live coders and VJs in practice based on interviews that identified important features, interactions, and aspects of their performance practice.

Table 2.1: Taxonomy presenting how the related software spans creative coding, live programming, live coding, and VJing practices. Y (yes) indicates that the software explicitly supports this practice. S (some) indicates that the software supports some aspects of this practice.

Software	**Creative coding**	**Live programming**	**Live coding**	**VJing**
Max/MSP/Jitter [19]	Y	S	S	S
VVVV [46]	Y	S	S	S
Palimpsest [54]		Y	S	S
LiveCodeLab [60]	Y		Y	S
Cyril [4]	Y		Y	S
Gibber [75]	Y	S	Y	S
KodeLife [12]	Y	S	Y	S
Praxis LIVE [78]	Y	Y	Y	S
Siren [80]	Y	Y	Y	
Resolume [30]	S			Y
Modul8 [21]	S			Y
Touch Designer [42]	S	S	S	S
Mother [51]	Y			Y
Visor	Y	Y	Y	Y

Chapter 3

Live Coding and VJing: Interviews

As part of a user-centered design process [49], seven live coders and VJs were interviewed to better understand their practice. The intent of the interviews was to inform the design of new hybrid environments that combine and improve upon live coding and VJing. The interviews solicited qualitative insights around the needs and expectations of live coders and VJs (RQ1 §1.1). Each interview participant was asked about their background, the software and hardware components that made up their typical performance setup, the types of interactions that they conducted during a typical performance, and the important qualities or limitations of their performance setup and practice. The analysis of the interview results revealed a number of themes with respect to the important features, interactions, and aspects of their performance practice. Conducting these interviews was also useful for developing my own understanding of live performance practice, having never performed before. This research was approved by the Victoria University of Wellington Human Ethics Committee (refer to Appendix A).

3.1 Interview Procedure

Interview participants were recruited based on their existing experience with some combination of live coding, creative coding, VJing, and live performance. A targeted recruitment strategy was used to select participants that were local to the Wellington region so that interviews could be performed in person. Participants were identified based on my existing networks or were referred to me by other people in these networks. Participants were provided with the information sheet found in Appendix B.

The interviews were semi-structured. The schedule in Appendix C outlines the base questions that were asked to each participant. The schedule included: 4 questions to learn about the participant's performance background; 1 question to learn about their programming skills and how they use code in performance; 5 questions to learn about their typical performance setup, the common actions conducted during their performances, the important qualities of their practice or performance setup, and the limitations of their practice or performance setup; and 1 question to learn about their future plans for developing their performance practice. The interviews were conducted openly so that it was possible to explore the topic of live performance more broadly than the questions set out in the schedule. Notes were taken during each interview to record qualitative data. Each set of notes was supported by an audio recording to justify the findings and back up any claims.

3.2 Participants

The 7 participants that were interviewed came from a range of backgrounds. 2 participants (P4, P5) were experimental musicians with musical live coding experience while the other 5 participants (P1, P2, P3, P6, P7) were visual artists with VJing experience. One of the visual artists

(P2) also had visual live coding experience. Both musicians also had some experience creating visuals using Resolume or Cyril (P4), and OpenProcessing (P5). Both musicians (P4, P5) had programming experience but of the 5 participants with VJing experience, only 4 (P1, P2, P3, P6) had programming experience and only 3 (P2, P3, P6) utilised coded content in their performances. None of the interviewed participants classified themselves entirely as live coders, meaning the participants were predominantly VJs and hybrid or experimental musical performers. Further details on each participant's background including their classification, software and hardware used, and relevant experience are shown in Table 3.1.

3.3 Analysis

The qualitative data provided from the participants' responses to the interview questions was analysed using thematic analysis [55]. This analysis was applied using a step-by-step approach [68]. Firstly, the notes were read to gain familiarity with the data. The data was organised by editing the notes and splitting them into discrete observations. Each observation was coded using an inductive approach where the codes were developed based on the content of the data. Themes and subthemes were constructed by grouping codes together and validating if the underlying data supported the theme. A number of iterations took place to refine the codes, themes, and subthemes. As a result, 3 themes were produced and represented using thematic maps. Each thematic map presents the relationship between the theme, subthemes, and codes. Themes are displayed as blue circles, subthemes as blue boxes, and codes as white boxes. The lines between themes, subthemes, and codes represent hierarchy while lines between codes represent an observed correspondence between codes.

Table 3.1: Interview participants background.

ID	Classification	Software and hardware used	Live coding experience	VJing experience	Programming experience
1	Live video artist	Laptop, Resolume, MIDI keyboard, Syphon Recorder		8 years	2 years
2	Live digital visual artist	Laptop, VVVV, TouchDesigner, joystick, game controller, MIDI controller, custom software	8 years	10+ years	10+ years
3	Creative coder and VJ	Desktop Computer, Processing, Resolume, Sublime Text, MIDI controller		1 year	1 year
4	Experimental sound artist	Laptop, Pure Data, Max/MSP, SuperCollider, Chuck, Cyril, Resolume, MIDI pad controller, custom built hand controller	5 years		9 years
5	Audio programmer and improviser	Laptop, Chuck, Max/MSP, OpenFrameworks, drumset, Game Boy synthesiser, custom MIDI controller, custom MIDI foot pedal	5 years		5 years
6	Visual artist	Desktop Computer, Unity, VDMX, MIDI controller		7 years	6 years
7	VJ, lighting, artist, project manager, all rounder	Desktop Computer, Resolume, Magic, video cameras, video capture cards, MIDI controller		3 years	

The finalised themes were: *Features*, *Interactions*, and *Practice*.

Features highlights the prominent features of the software and hardware used in the participants' performance setup including subthemes corresponding to audio responsive behaviour, communication protocols, and content arrangement.

Interactions highlights the prominent interactions that the participants conducted using the software and hardware in their performance setup.

Practice highlights interesting trends or ideas from within the general performance practice of participants including interconnected subthemes corresponding to the usability of utilising code, the use of instruments, visible interactions, and the role of other people.

The findings surrounding each theme are now discussed in detail.

3.3.1 Features

The first theme was *Features* which highlighted a number of prominent or common features of the software and hardware utilised in the participant's performance setups. The thematic map for the *Features* theme is presented in Figure 3.1.

The use of features that enabled *Audio responsive* visuals was a subtheme reported by the 5 participants with VJing experience. This was achieved using two *"real-time audio analysis"* approaches, The first approach was concerning *"audio frequency"* analysis (P1, P2, P3, P6), also known as *"FFT"* (P6) which stands for 'fast Fourier transform', an algorithm for transforming sound signals into the frequency domain. The second approach was *"beat responsive"* behaviour (P1, P2, P7). Participants reported audio responsive visuals were important because it *"makes VJing look and feel more responsive"* (P1), and allowed them to *"create graphics that are actually in time with the music"* (P3).

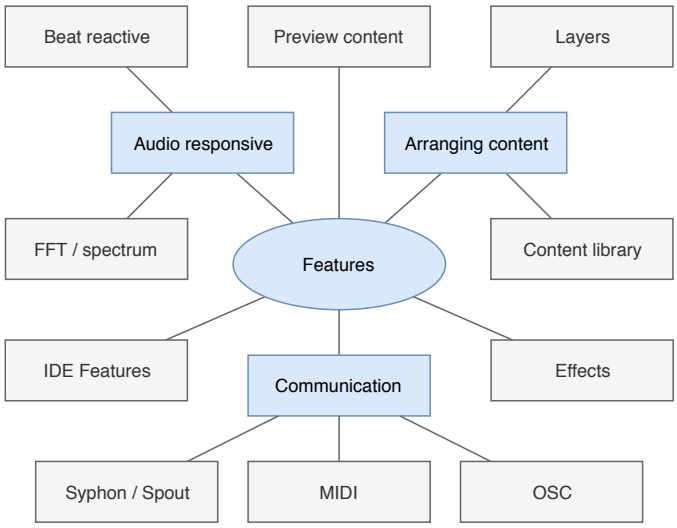

Figure 3.1: Thematic map for the *Features* theme and the relationship between the theme, subthemes, and underlying codes. The subthemes are *Audio responsive*, *Communication*, and *Arranging content*.

Another subtheme was *Communication* which highlighted the use of different protocols supported by participants' software to enable communication with other software or hardware devices. The first protocol was MIDI which was used by all participants to communicate with some form of *"MIDI controller"* device. The second protocol was Syphon / Spout and was used by the 5 participants with VJing experience and by one musician (P4) when they had performed visuals. P6 describes the protocol as *"a big feature, sharing graphics textures between programs."* P1 mentioned that they used *"Syphon Recorder to record sets"* while P3 mentioned that they would *"use Syphon to get Processing into Resolume."* The last protocol was OSC which was used by 2 participants. P4 used OSC for hardware controllers by building *"OSC ready devices."* P6 used OSC to communicate between software programs, for example, they mentioned *"VDMX would OSC out to Unity for audio analysis data, which could then manipulate models, materials, shaders."*

The last subtheme was *Arranging content* which was common amongst the 5 participants with VJing experience and highlighted two ways that content was arranged in VJ software. One way content could be arranged was through layers. P3 mentioned how their final visual output was made up of multiple clips in Resolume where *"each clip in the column/layer grid is a sketch that is running"* and P1 mentioned how they would composite video clips in different layers by *"combining layers using alpha blends."* Three of the participants with VJing experience also expressed how they made use of a content library to arrange content, for example P7 *"makes a whole new library of content for each gig."* Similarly, P3 would *"listen to the artists that I am going to be playing with, create a library or adjust existing libraries to fit their music style."* P1 also mentioned that they would *"come prepared with layers, sometimes find clips from library"* when performing.

A number of other features were also mentioned by multiple participants as important aspects of their performance software. The importance of features standard in integrated development environments (IDEs) was discussed by P2 and P4 when live coding. For example, P2 mentioned that *"error highlighting is great"* while P4 mentioned that it *"would be nice to use the same toolkit as an IDE"*, for example by allowing *"error highlighting, syntax highlighting, autocompletion"* when live coding. Another prominent feature discussed by 3 participants (P1, P2, P3) was the ability to preview content, in particular, the ability to visualise what it would be like to play a certain clip. The importance of this was emphasized by P1 who mentioned that *"being able to visualise is less risky as you only push up things that look good"* and P3 who said that without this ability it was *"hard to test out new ideas in a live show without showing the audience."* The final feature was the ability to apply effects onto content which was explicitly mentioned by 6 of the 7 participants. P5 used effects in music such as by applying a *"modulated filter to affect someone's voice in real-time"* while P3 used effects in visuals *"like messing with colour or a polar kaleidoscope."*

3.3.2 Interactions

The second theme, *Interactions*, highlights a number of prominent or common interactions conducted by participants during their performances. The thematic map for the *Interactions* theme is presented in Figure 3.2.

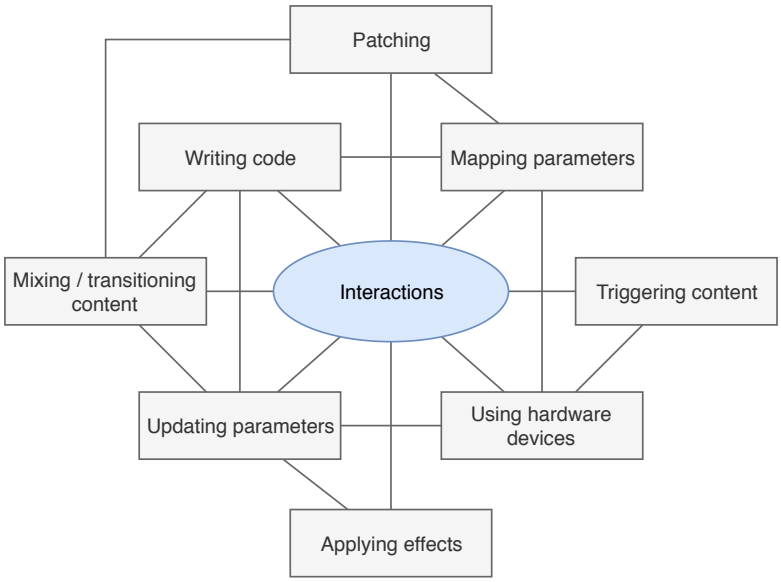

Figure 3.2: Thematic map for the *Interactions* theme and the relationship between the theme and underlying codes.

One common interaction was the performer's ability to trigger new content, as shown by the 5 participants with VJing experience, for example (P1, P2, P6) mentioned how they would be *"triggering clips"* throughout their performances. Applying or updating effects was also mentioned by the participants such as how P3 would *"tweak with effects"* and how P7's use of Resolume meant they *"can try different effects."* Another common interaction was to mix or transition content as mentioned by P2 and P6 based on how they would be *"video mixing."* P1 and P5 also mentioned how they would *"crossfade"* visuals and music respectively. Mapping parameters within the software or to hardware was also a common interaction men-

tioned by 3 of the participants. P2 mentioned how they would *"arbitrarily map controller to individual values"* while P4 emphasized the *"ability to map anything with anything (visual, text, numbers), all represented by data."* In a more concrete example, P6 mentioned how by *"using MIDI controller to map to VDMX"* they could *"drive an opacity layer or frequency band."*

Participants also described interactions at a low level, describing how they achieved something rather than what they achieved. Three low level interactions were identified, the first being writing code, as mentioned by the 3 participants with live coding experience through *"editing GLSL"* (P2), *"writing a script"* (P4), and *"changing code"* (P5). The second low level interaction was patching as mentioned by 2 participants who live coded in visual programming languages by *"shuffling nodes around"* (P2), and *"repatching Max MSP with one hand"* (P5). The third low level interaction was through using hardware devices as mentioned by (P5) when they *"used the foot pedal to use MIDI to change parameters"* or by (P6) through *"using the MIDI controller to map to VDMX."* Other interactions are assumed to have been performed through the mouse or keyboard, for example by clicking through the software graphical user interface as mentioned by P1: *"clicking on a clip shows it in the preview"* or by using keyboard shortcuts as mentioned by P6: *"simple interactions using keyboard shortcuts."*

Finally, updating parameters was shown to be one of the most prominent interactions due to how highly connected it was to other interactions. Updating parameters was conducted in a number of ways such as through writing code as mentioned by P5 where they would *"recompile code on stage with different parameters"* and by using hardware controllers as mentioned by P4 with respect to their hand controller where the *"hand does continuous control of parameters."* Updating parameters was used to achieve a variety of results including how P1 would update effects by changing the *"hue, saturation, pixelation"* or layer properties *"width, height, opacity, speed."* Changing the opacity of a layer can be assumed to have the

same effect as mixing or transitioning content. Similar to the relationship between updating parameters and using hardware controllers, controllers were also shown to be used to trigger content as mentioned by P4 with respect to their *"grid-based pads controller"* which would do *"discrete stuff, note selection, octave selection."*

3.3.3 Practice

The third theme, *Practice*, highlights a number of interesting trends or ideas of general performance practice that were identified across the participants. The thematic map for the *Practice* theme is presented in Figure 3.3.

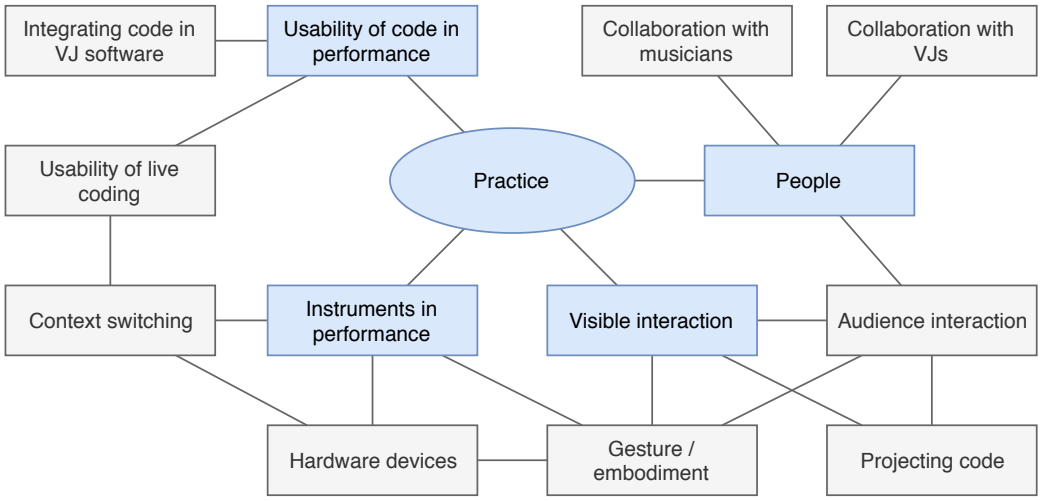

Figure 3.3: Thematic map for the *Practice* theme and the relationship between the theme, subthemes, and underlying codes. The subthemes are *Usability of code in performance*, *Instruments in performance*, *Visible interaction*, and *People*.

The *Usability of code in performance* was a subtheme that arose based on how participants with VJing experience had gone about integrating coded content into their practice as well as how participants reported on the usability of live coding. The use of coded content in performance was shown by P2 who was *"now writing GLSL (in live code)"* during the performance as

they found it easier than patching: *"sometimes extending VVVV on the spot, but it wasn't particularly built for live stuff."* They went on to comment on the usability of visual programming languages where a *"problem with most graphical programming languages is that every object requires a click or selection through a menu. Instead of typing a line of a code, click by click is slower."* P6 mentioned how they also *"used visual scripting"* and preferred it over textual coding as they *"don't have to worry about text side of things, likes to see how things are connected, pin down what's affecting what"*, but they did not do any live patching during performance. P3 also demonstrated the use of coded content in performance by combining Processing sketches with VJ software where the *"coding done in Processing using Sublime, mix that in Resolume, projected from Resolume. Use Syphon to get Processing into Resolume."* They also discussed how they prepared the sketch code upfront and that they *"don't always do a lot of work with live code at the gig, just because it was easier not to."*

The need for using coded content in performance was emphasized by 2 participants. P1 mentioned that *"working with pre-recorded content is not expressive"* and while the *"level of detail of video is great"*, they would prefer *"to work with both mediums"*, be it video content and coded content as coded content could allow for *"real-time graphics that are smooth, drawn, and with interactive parameters."* P3 also emphasised that code was an *"easy way of doing audio-reactive graphics, can quickly apply colour to different ranges of music, apply movement, transformations. Not sure what kind of software gives you that kind of fine control that you get with procedural graphics."* While some participants managed to integrate coded content in their performances, it was not without its limitations. Three participants reported limitations when integrating coded content in VJing performances. P1 mentioned how they wanted to use coded content in performance but could not due to limitations of their software: *"would write own FFGL plugins for Resolume, but no documentation, not user-friendly."* P2 also reported issues writing Python code in VJ software due to how *"TouchDesigner builds Python but*

can't use Python within TouchDesigner." P3 also reported usability issues with their Processing to Resolume setup due to the fact that they *"can't launch Processing sketches from Resolume"* and that it *"takes a long time for Processing to launch a sketch."*

The second subtheme was *Instruments in performance* due to how common it was amongst the participants to utilise some kind of external hardware controller or instrument in combination with their software to produce music or visuals. As previously mentioned under the *Features* theme, all of the participants used some kind of MIDI controller in their performance setup. P4 mentioned how when *"live coding, using a controller as a sound input in the software is like using a live instrument in real-time."* The use of instruments in performance was also linked to the usability of live coding by P5 who used a drumset in their performance while live coding. They mentioned that when using the computer that the *"ability to interact is extremely slow"* whereas *"on a drum set can interact instantly."* Through their performances they found that *"repatching Max MSP with one hand is a disaster"* and that it was *"hard to act impulsively in a music setting with code. As it is quite abstract, requires mathematical thinking. Very hard to play the drums at the same time as it contrasts so much."* While this shows the benefits of instrumental performance, it also highlights a usability issue due to having to context switch between live coding and using the controller. Similarly, P2 mentioned how they would be *"writing code with one hand and the other hand is fading between things that are already loaded up"*, but they also admitted that there was *"always going to be a bit of context switching."* Another benefit of instruments was discussed by P4 who used a custom hand controller. They mentioned that it was *"important to establish language"* with the audience and reported that controllers allowed for building a *"relationship between physical gesture in audio (or something visual)."* P5 also emphasized how the opposite held true for computers by stating that there are *"not many situations where you are more disconnected, when you are embodied in the computer."*

The discussion of gestural or embodied interaction in performance leads into the next subtheme, *Visible interaction*, which highlights how the audience perceives the performer's interactions. P2 discussed how *"audience interaction is pretty key, some performers are pretty gestural"* and went on to mention a specific DJ that was *"so quick and expressive in the way he moves. Physicality in his performance is half of what the audience see. The live aspect is pretty inspiring."* P5 mentioned that it was important to *"communicate the creative process going into the performance."* Code projection was also mentioned by 2 participants when discussing live coding as it allowed for showing part of the performer's process to the audience. P2 mentioned how as they *"do not sit on stage as can see better from behind audience"* that *"projecting code on the screen allows for appreciation of technique, rather than just appreciation of the result."* P4 also mentioned that *"projected code makes music an artefact"* because music is *"always time-based but the text is static, contrasting with the physical live embodiment."* Despite this, both participants were also critical of code projection. P2 mentioned that they *"can project code but only in right contexts"* and *"generally focuses on creating results that hold for their own merits."* P4 also stated that they *"don't subscribe to showing projection on screen, live coding is a method, does not define what can or cannot be seen on stage."*

The importance of audience interaction also overlaps the last subtheme, *People*, which highlights the importance of collaborating or interacting with others. P7 reported how their use of live cameras when VJing would *"provide crowd interaction"* and that the audience *"love to see themselves."* All of the participants emphasised the importance of collaboration with other people to some degree. The participant's collaboration involved working with others that were responsible for the music, lighting or visuals. P1 mentioned the need to *"make your content fit the environment"* and also the need for *"collaboration with lighting people, focus on lights or projections, add smoke."* P1 also reported *"worked with another VJ, two VJ environments between a mixer, crossfade or blend in middle"*, and the result was *"chaotic or*

cool." P7 reported they would use a video capture card for collaboration by *"capturing another 'Guest' VJs work"* and that their *"main pride is in working with other people."* P6 who worked closely with a musician reported that they would be *"triggering clips throughout performance based on what I was hearing in the audio world or what was rehearsed."* Similarly, P3 reported that they *"pays most attention to the DJ"* and *"puts on what they feel fits with the music."* P2 reported *"collaborating with bands, developing a look for each band."* P4 also expressed an interest in collaboration with respect to live coding, stating that they were *"interested in how a live coder would work with a live improv person."* This kind of collaboration could solve the issue of context switching by enabling two performers to work together to live code and improvise using a hardware controller at the same time.

3.4 Discussion

The analysis of the interview data unearthed some central ideas around important features, interactions, and aspects of performance practice. The following summarises and discusses the findings.

Features: A number of features in live performance software were commonly used by the interviewed participants. The majority of these features were for VJing and is likely due to the larger number of participants with VJing experience over live coding experience. Regardless, the results still highlighted important features of VJing software.

The ability to create visuals that responded to music was prominent amongst the participants with VJing experience and was either based on analysing the audio frequency spectrum or the beat of the music in real-time. Support for different communication protocols was also very common, in particular, the MIDI protocol due to how it enabled support for external hardware devices like the MIDI controllers that were used by all

of the participants. The OSC protocol was also used by two of the participants for communicating with hardware or between multiple software applications. Syphon and Spout were also commonly used as a method for sending visual content between applications or recording the visuals for later playback.

The ability to arrange content at a high level was common amongst the participants with VJing experience, for example, the ability to create layers of video content that are composited together into the final output. Having access to a content library, the ability to preview content before it's rendered to the audience and the ability to apply effects onto content were also commonly used or important to the participants with VJing experience. In terms of features related to live coding, some usability features that are standard in IDEs were raised as important including syntax highlighting, error highlighting, and code completion.

Interactions: A number of interactions conducted during live performances were also shown to be important by the interviewed participants. Updating parameters was the most prominent interaction due to the variety of results it could achieve. For example, updating the opacity parameter of a layer allows for mixing content while parameters of individual effects can also be manipulated to change the final output. The ability to easily tweak parameters enables performers to make changes in the visuals that reflect changes in the music. Some participants explicitly mentioned how they used live coding or hardware controllers to change parameters. This also emphasises the importance of the ability to easily create mappings between parameters and hardware controllers, another common interaction conducted by the participants.

The ability to easily switch between or trigger new content was also a common interaction conducted by the participants. Unlike updating parameters, triggering content was described as an action that would make discrete changes to the final output, whereas updating parameters would

allow for smooth changes to the final output. Patching was also an interaction commonly discussed by the participants who used visual programming languages. It was apparent amongst these participants that patching during live performance had usability issues and they instead favoured to have the patches set up before a performance. One participant also favoured live coding in a textual language over visual patching during performances. Live coding and the act of writing code itself was also a common interaction. Finally, the ability to apply effects that manipulate the final output was a common interaction conducted by the participants.

Integration of code with VJ software: The need for more effective integration of code with VJ software was apparent based on the practice of some of the participants. While a number of the participants with VJing experience used coded content in their performances, they also identified limitations based on how they went about integrating code. Some of these limitations were due to the VJing software itself not providing an effective approach to integrating coded content or providing an approach to easily live code. There were also limitations where the visual content had to be communicated between applications to achieve the desired results. For example, one participant sent the output of Processing to other VJ software to enable performances with coded content, an approach that came with limitations and usability issues. These limitations highlight the need for VJing software that provides an easy to use approach to incorporating coded content in live performance.

Hardware controllers: The use of hardware controllers was a very prominent aspect of the performance practice of the interviewed participants. Each of the 7 participants used some form of physical controller to enable gestural control during their performance. While the use of hardware controllers creates an opportunity to introduce gestural and embodied interaction into live coding performance, it also comes at the cost of introducing usability issues such as the need to switch contexts between the computer

and the controller, as reported by the participants. This emphasises the need to make interactions between code and controllers as easy as possible and therefore reduce the friction caused by context switching. Hardware controllers were also deemed important by the participants due to how they enabled visible interaction of their practice to the audience. This new form of transparency offers another opportunity to improve upon live coding practice where the performer is typically hidden away behind a laptop screen.

Visible interaction: An aspect of live coding practice discussed by the participants with live coding experience was the ability to project the code onto a screen. While the participants admitted that code projection was a useful technique for showcasing a performer's process, overall they reported that it was not important due to how they treated live coding as a method. The participants placed more importance on the final output of their work and stated that code projection forces the performance into a specific genre. It is important to note that none of the participants with live coding experience had performed at algoraves where code projection is standard practice. The participants with VJing experience were also clearly observed to not provide this kind of visible interaction and instead focused on the quality of their visual output. This raises a key difference between live coding and VJing. Live coders appear to be concerned with making their process fully visible, even if it means revealing errors to the audience, but VJs are more polished in the sense that they make use of features such as the ability to preview content as a form of quality control and care less for visible interaction.

3.5 Summary

This study of live coders and VJs in practice has identified a number of important features, interactions, and aspects of performance practice. Ta-

ble 3.2 summarises these outcomes as specific aspects that will help to inform the design of future tools such as Visor. This study also raised a number of important discussion points for consideration when evaluating CJing and Visor including the usability of integrating coded content in VJ software, the issue of context switching, and the differing stance on visible interaction between live coding and VJing.

Table 3.2: Important features, interactions, and aspects of the participant's performance setup and practice. Each aspect is shown alongside the participants that explicitly stated it.

Theme/Subtheme	**Aspect**	**Participants stated**
Audio responsive	FFT	P1, P2, P3, P6
	Beat based	P1, P2, P7
Communication	MIDI	P1, P2, P3, P4, P5, P6, P7
	OSC	P4, P6
	Syphon / Spout	P1, P2, P3, P4, P6, P7
Arranging Content	Layers	P1, P2, P3, P6, P7
	Content library	P1, P3, P7
Features	Syntax highlighting	P4
	Error highlighting	P2, P4
	Code completion	P4
	Preview content	P1, P2, P3
	Effects	P1, P2, P3, P5, P6, P7
Interactions	Updating parameters	P1, P2, P3, P4, P5, P6, P7
	Mapping parameters	P2, P4, P6
	Triggering content	P1, P2, P3, P6, P7
	Patching	P2, P5, P6
	Writing code	P2, P4, P5
Practice	Integrating coded content	P1, P2, P3, P4, P6
	Using hardware controllers	P1, P2, P3, P4, P5, P6, P7

The next chapter introduces the new hybrid CJing practice and the new Visor environment that embodies CJing. The aspects identified as a result of this study are important characteristics of CJing and a number of them were incorporated into the design of Visor.

Chapter 4

CJing and Visor

A new hybrid environment has been developed as part of this thesis to embody CJing, exploring how live coding and VJing can be combined to harness the strengths of both practices while simultaneously removing limitations identified in each practice (RQ2, RQ3 §1.1). This new environment is called Visor [44]. Visor combines aspects of live coding and VJing software to enable live visual performance. In Visor, live coding is conducted in the Ruby language and visuals are rendered using the Processing API. A number of graphical user interface (GUI) elements are also provided to support live performance. Figures 4.1 and 4.2 show Visor in action by presenting the Visor interface and corresponding visual output respectively. Visor has captured the interests of many live coders, creative coders, and VJs, and has been downloaded more than 500 times since January 2019. This chapter begins by providing an in-depth description of the CJing practice, then describes Visor's design, features, implementation, and development approach, and concludes by comparing CJing and Visor with the related work.

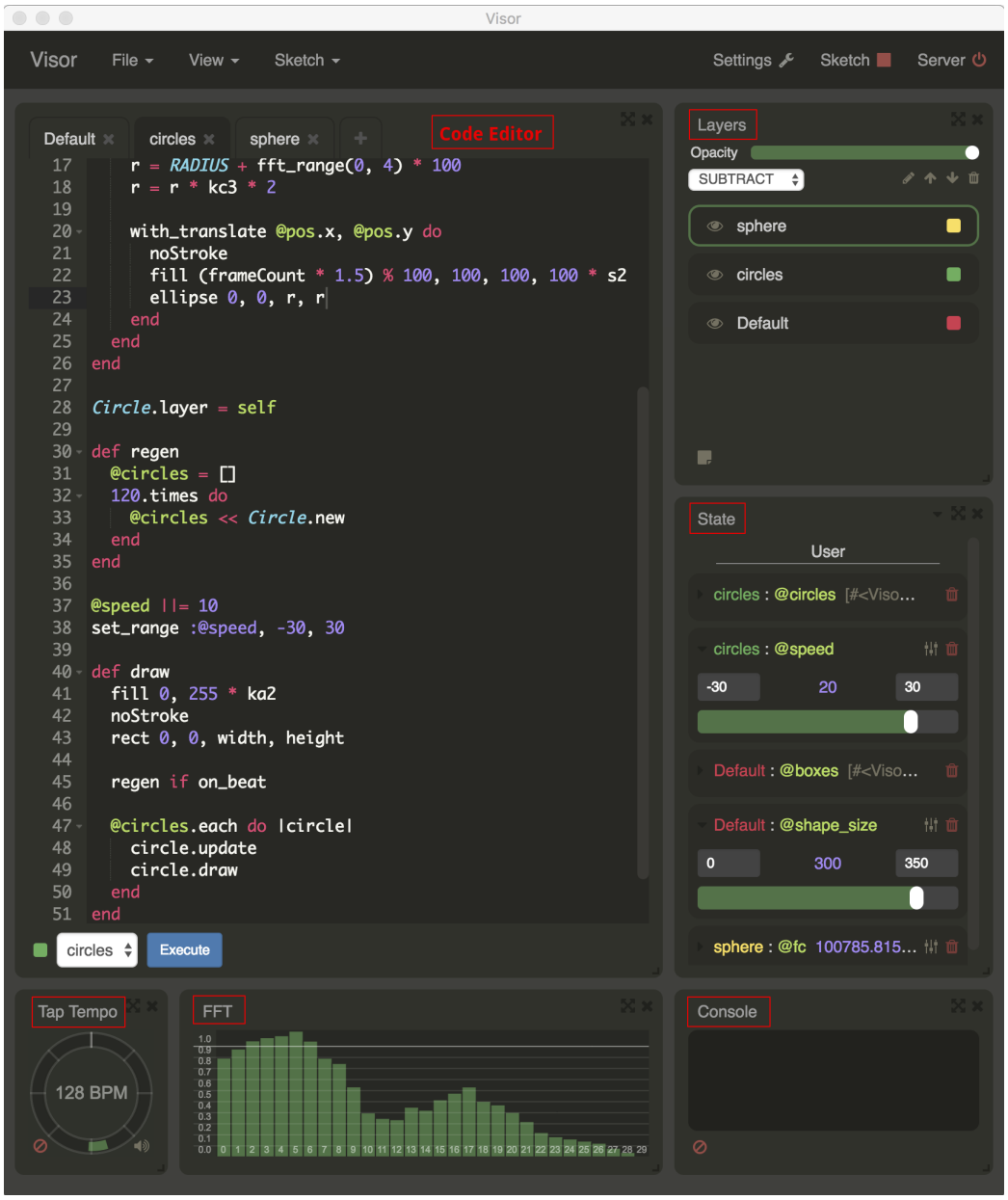

Figure 4.1: Visor interface in action. The interface is made up of multiple GUI elements that offer different functions including a live code editor (left), layer interface (top-right), state management interface (right-middle), console (bottom-right), audio frequency spectrum / FFT display (bottom-middle), and tap tempo interface (bottom-left). The visual output corresponding to this state of the interface is shown in Figure 4.2.

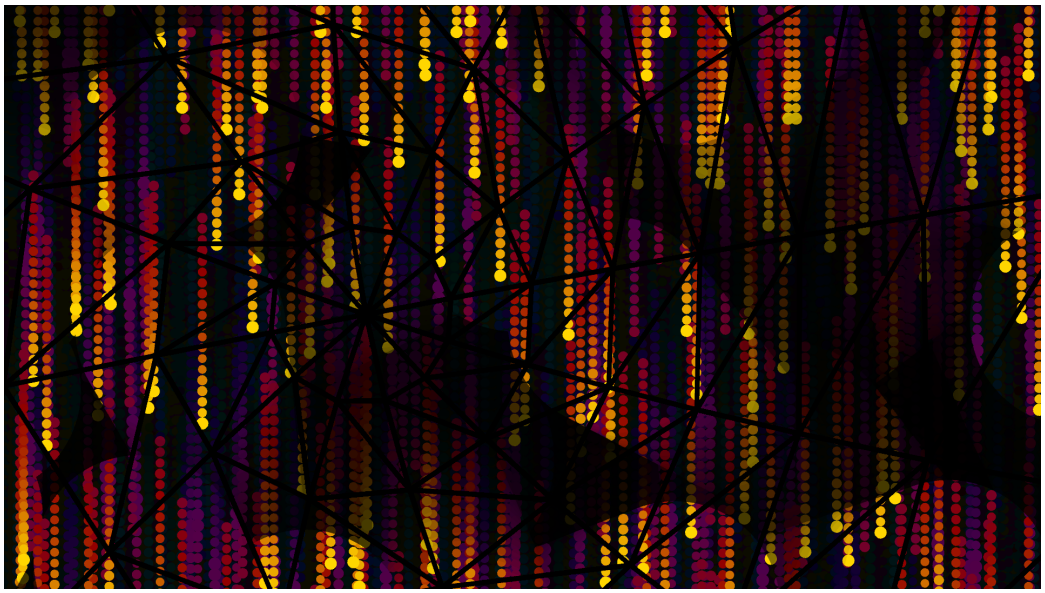

Figure 4.2: Visual output rendered from Visor based on Processing. These visuals correspond to the state of the interface shown in Figure 4.1.

4.1 Code Jockey Practice

The new hybrid code jockey practice (CJing) combines aspects of live coding and VJing. In CJing, a performer known as a code jockey (CJ) interacts with code, GUIs, and hardware controllers to improvise or manipulate visual content in real-time. CJing harnesses the strengths of live coding and VJing to enable flexible performances while simultaneously removing limitations identified in each practice. CJing is designed to complement live coding and VJing by providing a new approach that enables performers to utilise aspects of both practices in the same performance. For example, a performance aesthetic of the CJing practice is the utilisation of live coding as a method to improvise 'visual instruments' on the fly. Once defined, visual instruments can be performed using GUIs and hardware controllers to generate live visuals. CJing pushes live coding and VJing

practices in new directions by blurring the lines between them. The live coding practice is directed towards the incorporation of GUIs other than text editors and the use of gestural inputs such as MIDI controllers. The VJing practice is directed towards focusing on generative content that can be edited at runtime using live coding. While this thesis focuses on VJing to produce visual content, the same ideas could be applied with DJing for producing musical content. CJing is now placed in the context of the related subject areas and key ideas that it is based on.

4.1.1 Context

Figure 4.3 places CJing in the context of the broader subject areas that formulate it: creative coding, live programming, and VJing. CJing combines the expressive nature of creative coding to generate content with live code and the utilisation of visualisation and interface techniques based on live programming; all placed in the context of dynamic audiovisual performance as illuminated by VJing.

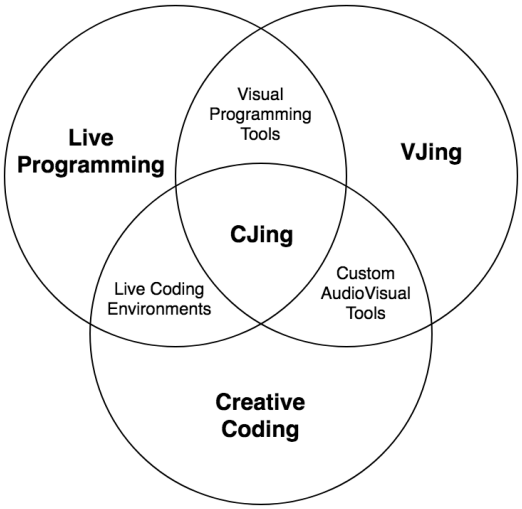

Figure 4.3: CJing in the context of the broader subject areas that formulate it.

4.1.2 Key Ideas

CJing practice is based on three key ideas, each the synthesis of aspects from creative coding, live programming, live coding, and VJing practices. The first idea is *code as a universal language* that combines aspects of creative coding and live coding, the second is *complete content control* that combines aspects of live coding and VJing, and the final idea is *user interfaces as an abstraction* that combines aspects of live programming and VJing. The aspects of the related practices were identified from the related work and the study of live coders and VJs in practice (Chapter 3).

1. *Code as a universal language*

CJing utilises creative coding to generate content from scratch, constructing the final output primarily through the use of algorithms. Live coding enables coded content to be manipulated on the fly at runtime. Code is flexible enough that it can allow for endless permutations of possible content, but also precise enough that it can allow an artist to produce a specific aesthetic. Using code as a common language can enable an ecosystem to be formulated and be treated akin to how looping video clips are treated in the VJ community. CJs can also use supporting assets such as images, videos, or 3D models if they are specified programmatically in code. Software that relies on code to instruct the final output is fundamental to the CJing practice. If a DJ mixes musical tracks and a VJ mixes video clips, then a CJ must mix code in their performances.

2. *Complete content control*

CJing should allow for content to be manipulated at both low and high levels, enabling flexible control of the final output during performances. CJs can improvise content from scratch, remix existing content, and composite multiple content elements together. Live coding provides the low level aspect of content creation and manipulation, while VJing provides the high level aspect of orchestrating the final output through the organisation

of layers and control of effects and parameters. CJing combines the two, with all aspects of the final output accessible through code or manipulable through user interfaces. Providing this kind of flexibility enables a CJ to harness the strengths of live coding and VJing to perform with generative content.

3. *User interfaces as an abstraction*

CJing employs user interfaces that abstract upon live code, providing high level functionality that would not be easy to achieve by simply writing code. CJing systems should maintain a relationship between user interfaces and code. Contextual interfaces should detect when changes occur to the code and update themselves accordingly. At the same time, code should be able to access the state of interactive user interfaces. For example, as with VJing software, slider widgets or hardware controller knobs could be used to manipulate parameters in the program, removing the need for the CJ to inspect the code or manipulate parameters through live coding. The same GUIs could also visualise the state of the program, improving the CJ's comprehension of the inner workings of their live coded program.

4.2 Visor: Design

Visor's primary design goal was to embody the CJing practice, demonstrating how aspects of live coding and VJing software can be combined into a single environment. Visor achieves this goal by offering a number of features to facilitate live performance. These features are based on the gaps identified in the related work (Chapter 2), the results of interviewing live coders and VJs (Chapter 3), and the three key ideas of the CJing practice (§4.1.2).

Visor's design consists of a number of core features that facilitate live performance. These features include the following: the ability to live code visuals with the Processing API in the Ruby language; a state management interface that allows for automatic visualisation of and interactions to update live coded parameters; the ability to organise code into layers that can be composited together into the final output using a variety of blend modes; an interface to visualise the FFT spectrum of an audio input which can be inspected and referenced in the code to create audio responsive visuals; a tap tempo interface for setting a tempo that can be referenced in the code to animate visuals to the beat; and a framework for configuring MIDI input devices such as hardware controllers where the individual controls can be directly referenced in the code or mapped to state parameters. These core features are designed to embody the key ideas of the CJing practice, as presented in Table 4.1.

A number of other notable features were factored into Visor's design including an in-app tutorial, in-app documentation, support for multiple display windows, support for Syphon [36], a reconfigurable interface, a console, code syntax highlighting, code error highlighting, and an option to project the code on top of the rendered visuals. Visor was also designed with a focus on usability, aiming to make it as easy as possible to integrate coded content into live performances. Visor achieves this by offering a straightforward to use API that abstracts as much of the complexity of its core features as possible, reducing the time spent by the performer writing boilerplate code.

Due to time constraints, a number of features and interactions that were identified as part of the interviews in Chapter 3 were not included in the current version of Visor. The excluded items were: OSC support, content libraries, code completion, previewable content, effects, and patching. These features and interactions are left as potential ideas for future work.

Table 4.1: Visor's core features and how they map to the three key ideas of the CJing practice. Y (yes) indicates if a feature is designed to embody a given idea of the CJing practice.

Feature	**User interfaces as an abstraction**	**Complete content control**	**Code as a universal language**
Live coding		Y	Y
State management	Y	Y	
Layers	Y	Y	
FFT	Y		
Tap tempo	Y		
MIDI		Y	

4.3 Visor: Features

Visor offers a number of features to facilitate the live performance of visuals. These features are designed to meet the goal presented in Section 4.2. The following describes in detail how each of Visor's core features can be used in live performance. A demonstration of Visor's features in live performance can be seen in the TOPLAP 15th Birthday Livestream video[1].

4.3.1 Live Coding

Live coding in Visor is the primary method for creating visual content. To generate visuals, the performer can access methods from the Processing API such as `background`, `fill`, `translate`, `rect`, and many others. The performer can also use methods from an additional Visor API to access features like the FFT, tap tempo, and many others. A key difference between Visor and Processing is that programming in Visor is conducted with the Ruby programming language, rather than Java. The use of Ruby

[1] `https://tinyurl.com/visor-toplap15`

means that the performer does not need to specify types, and syntax features like curly brackets and parameters are completely optional. The result is that Processing code written in Visor is much less verbose than Processing code written in Java. The performer also has access to all of the language features of the Ruby language such as enumerators and blocks. Another difference between Visor and Processing is that there is no need for a `setup` method due to Visor's REPL like behaviour where arbitrary code can be evaluated at runtime. Therefore, code that would typically go in the `setup` method, such as code to define variables, should instead be placed outside of the `draw` method.

Figure 4.4 presents a basic Visor program used to generate visuals. The `draw` method defines the draw loop, the code that is run every frame as per the usual Processing behaviour. The performer can run this code from the code editor at any time to update the draw loop without restarting

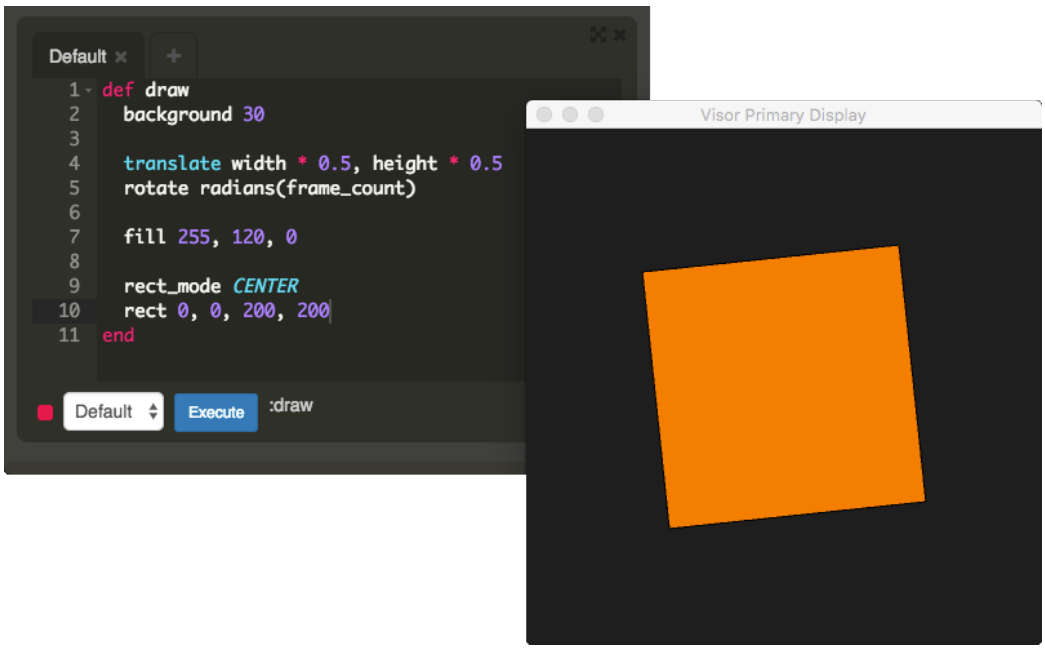

Figure 4.4: Live coding in Visor. The live code editor (left) presents a basic program to generate visuals using Processing (right).

the program. The code editor itself is made up of one or more code tabs, an execute (run) button, the result returned from the last time the code was run, and a dropdown to select which layer the code tab is currently targeting. In Visor, each layer specifies an independent draw loop and holds its own state. When a code tab is run, the code is evaluated within the targeted layer. More details on layers are provided in Section 4.3.3. The code editor presents syntax errors and runtime errors by highlighting the line the error occurred on and presenting an error message.

4.3.2 State Management

State management enables visualisation and interaction with live coded state in the Visor GUI, similar to interactions seen with parameters in VJing software. The performer can define state variables in their code using Ruby instance variables, indicated by the `@` character at the start of a variable name. Visor will automatically detect the presence of state variables and show them in the interface, saving the performer from writing boilerplate code to generate a GUI. Figure 4.5 shows how state variables have been defined in a Visor program to parameterise three different variables. The Ruby conditional assignment operator (`||=`) is used here to assign to variables only if they have not already been defined, preventing previous values from being overwritten when the code editor is run again. Once a state variable is defined, it will be shown in the state GUI with a contextual interface based on the variable's type. For example, numbers are presented as sliders, booleans are presented as checkboxes, and strings are presented as text fields. These GUI widgets provide immediate interaction with parameters without requiring the performer to modify the code. Number variables also support a range that the performer can manually adjust using the `set_range` method or by directly modifying the minimum and maximum values in the interface. State variables can be deleted if they are no longer required.

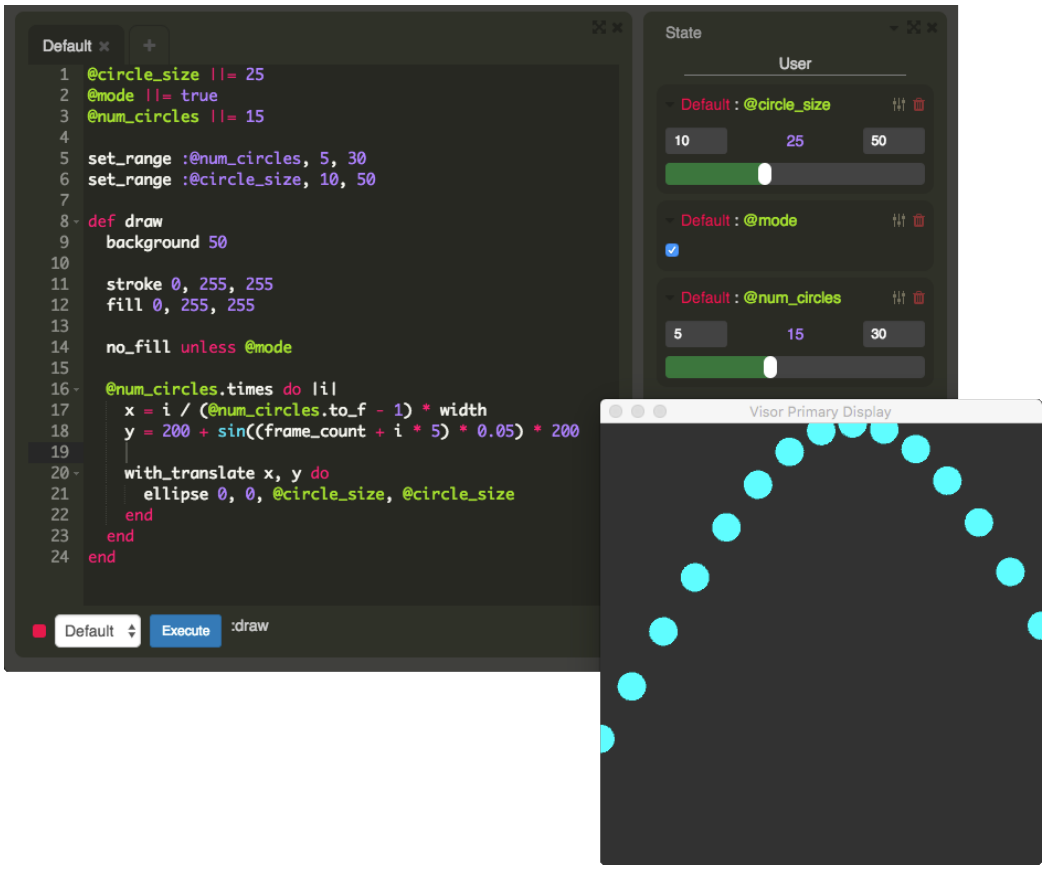

Figure 4.5: State management in Visor. The live code editor (left) defines state variables `@circle_size`, `@mode`, and `@num_circles` which are then displayed in the state interface (top-right), along with the rendered visuals (bottom-right).

4.3.3 Layers

Like in VJing or image editing software, Visor supports layering of content to compose complex final outputs. In Visor, each layer specifies an independent draw loop and holds its own state, allowing for clean separation of coded content. Multiple code tabs can be used in the code editor where each code tab is responsible for a different layer. Visor's layer interface provides the ability to create new layers, delete layers, toggle layer visibility,

or change a layer's blend mode or opacity. Layers can also be rearranged to modify the order that they are drawn in. Figure 4.6 shows how layers can be used to compose a final image. Section 4.4 demonstrates layers in-depth by presenting how three distinct layers can be composed together.

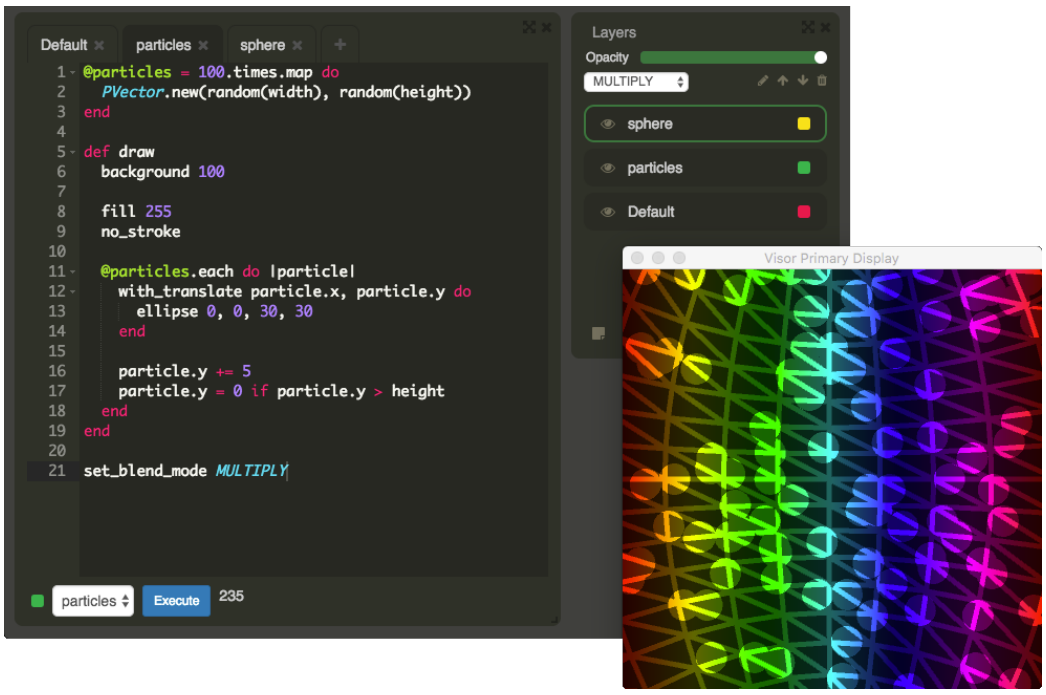

Figure 4.6: Layers in Visor. The live code editor (left) maintains three code tabs called *Default*, *particles*, and *sphere*, each of which corresponds with a layer in the layer interface (top-right). *Default* generates the rainbow colour while *particles* and *sphere* produce the shapes. The result of composing the three layers with different blend modes is shown in the rendered visuals (bottom-right).

Layers in Visor provide a useful abstraction by enabling creative possibilities without requiring the performer to write boilerplate code to implement similar features. Configuring blend modes is the most effective way to create interesting effects, for example, the `MULTIPLY` blend mode can

be used on a layer to apply a masking effect onto the layers below. Other blending modes are also available such as `ADD`, `SUBTRACT`, and `SCREEN` as per Processing. The performer can also use the Visor API to interact with layers, for example, the `set_opacity` and `set_blend_mode` methods can be used to update layer properties. The performer also has the option to put all of their layer code into one code tab using the `layer` method. This method accepts a layer name and a block of code, evaluating the given code within the layer of the given name. Similar to the state management feature, providing these kinds of API methods gives the performer the choice to interact with layers through the code as an alternative to using the GUI.

4.3.4 Fast Fourier Transform

Visor makes use of real-time audio analysis techniques to enable audio responsive visuals. The fast Fourier transform (FFT) interface visualises the frequency spectrum of a configured audio input device over time, allowing the performer to make informed decisions about how the visuals should react to the music. The visualisation consists of multiple bands where each band represents a group of frequencies. The lowest frequencies appear on the left and increment to the higher frequencies on the right. The frequencies are graphed against the band amplitude on a scale from 0 to 1. The FFT data can then be accessed in the code using methods such as `fft(n)` which returns the amplitude of the frequency band at index `n`, or `fft_range(n1, n2)` which returns the average of the frequency bands between the indexes `n1` and `n2`. Figure 4.7 shows how the FFT has been integrated into a Visor program to animate visuals to music in real-time.

A number of other API methods are also available to configure the FFT in Visor. The `set_fft_scale` method can be used to scale the amplitude of every frequency band to account for variations between audio devices or volume levels. The `set_fft_smooth` method can be used to adjust the damping applied to each frequency band over time, making the amplitudes animate more or less smoothly. The audio input device can be configured through the Visor settings menu. Providing audio input device configuration and in-built methods for accessing FFT values saves the performer from writing boilerplate code to conduct their own audio analysis, as is typical in Processing.

Figure 4.7: FFT in Visor. The FFT interface (top-right) displays the audio frequency spectrum and the code editor (left) refers to the FFT data. The `fft` method (line 17) and the `fft_range` method (line 21) access the FFT data to inform the visuals, as shown in the rendered output (bottom-right).

4.3.5 Tap Tempo

The tap tempo enables beat based behaviour in Visor, similar to the approach seen in VJing software. The tap tempo allows the performer to set a tempo by repeatedly clicking the tap tempo button in the interface or by using a keyboard shortcut. The interface also visualises the current BPM (beats per minute), temporal progress through the current beat, and offers buttons to mute or clear the current tempo. The tap tempo can be utilised in the code by using the `on_beat` method to trigger events each time a beat occurs. The `beat_progress` method can also be used to access a normalized value of the temporal progress. Figure 4.8 shows how the tap tempo has been integrated into a Visor program to animate visuals to the beat.

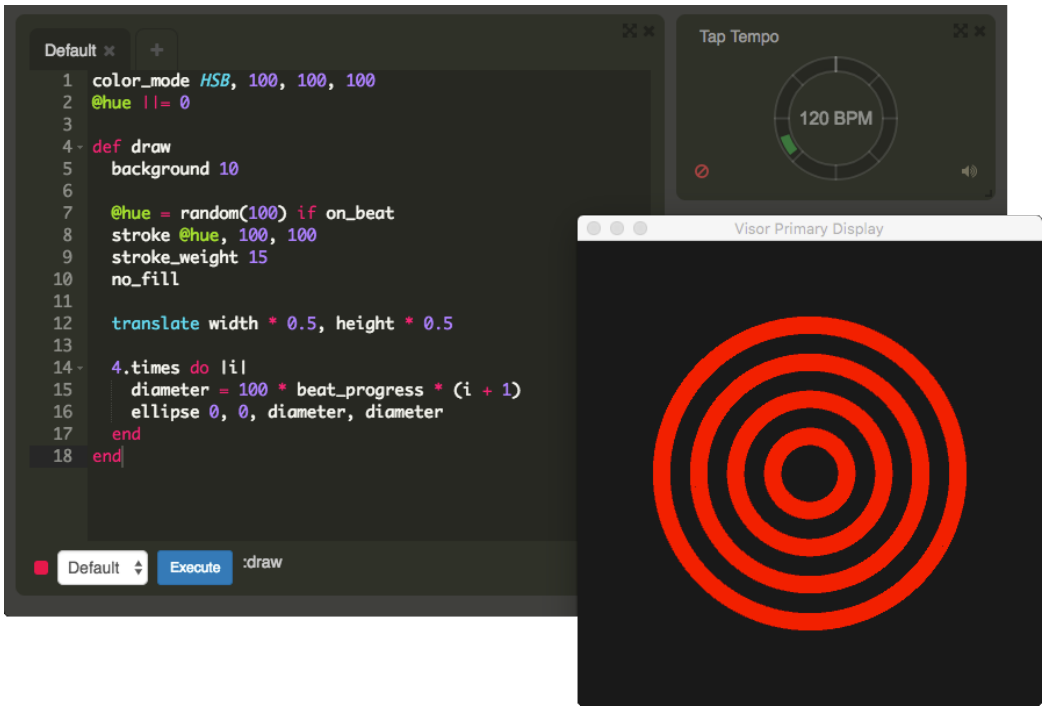

Figure 4.8: Tap tempo in Visor. The tap tempo interface is shown (top-right) and the code editor (left) refers to the tap tempo data. The `on_beat` method (line 7) and the `beat_progress` method (line 15) access the tap tempo data to inform the visuals, as shown in the rendered output (bottom-right).

A number of other tap tempo API methods are available in Visor such as `beat_count` which returns the total number of beats elapsed, `inv_beat_progress` which returns the inverted `beat_progress` value, and `request_tap` which can be used to programmatically trigger a tap, a useful feature for setting the tempo in response to MIDI events. The `beat_progress` method can also take an optional argument to specify a period, for example, to make visuals animate every four beats or every half beat instead of every one beat.

4.3.6 MIDI

Visor supports the use of MIDI inputs to interact with Visor code, allowing hardware devices such as MIDI controllers to be used for gestural interaction with coded content. Visor provides a framework for configuring MIDI devices and accessing individual controls in the code. The framework enables individual knobs, sliders, notes, or buttons on a MIDI controller to be mapped to unique variables that can be accessed directly in the code to query the state of the controls. Slider and knob MIDI variables simply return a normalized value of the state of the control. Button or note MIDI variables return an object with methods such as `down` for detecting if the button is pressed, `velocity`, and `on` or `off` which can be used to trigger events when the button is pressed or released. Figure 4.9 shows how MIDI variables can be integrated into a Visor program to multiply numbers by a scalar, trigger conditional code, or map directly to state variables. Visor's framework for configuring MIDI controllers allows parameters to be remapped on the fly through live coding and saves the performer from writing boilerplate code to handle MIDI inputs. Figure 4.19 in Section 4.4 presents a MIDI controller annotated with parameters and MIDI variables to illustrate a mapped controller.

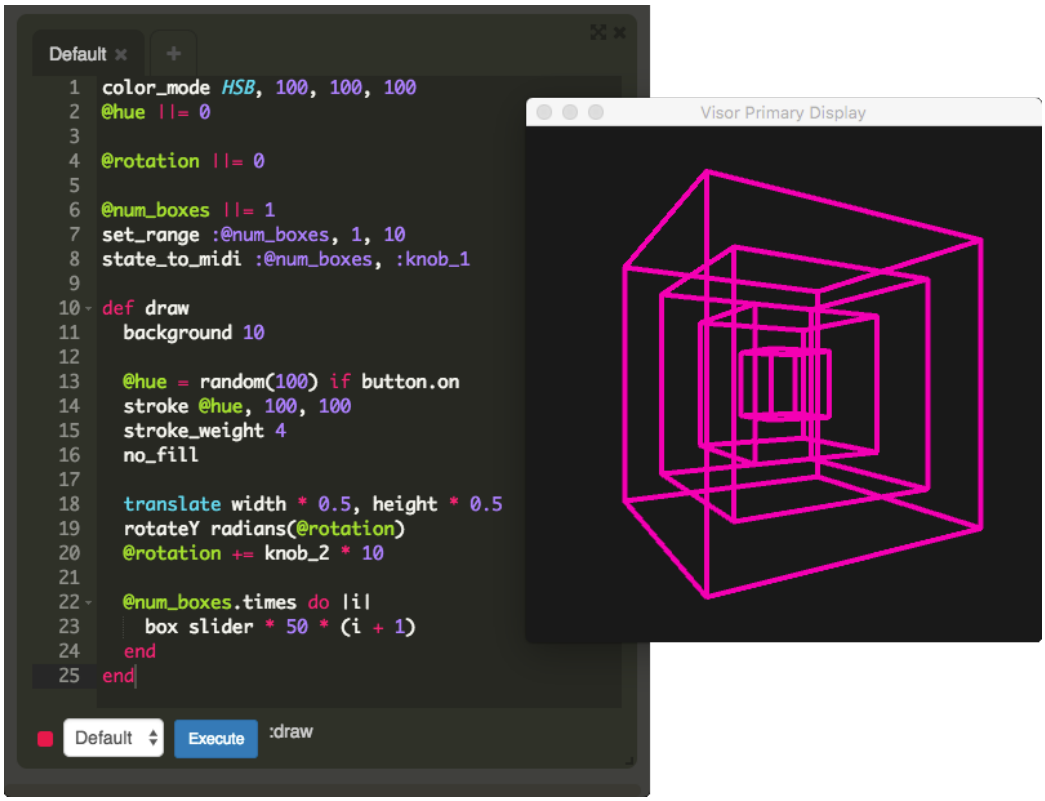

Figure 4.9: MIDI in Visor. The code editor (left) makes use of four MIDI variables to interact with the rendered visuals (right). `knob_1` (line 8) is directly mapped to the state variable controlling the number of boxes drawn, `knob_2` (line 20) is used to control the velocity of the boxes rotation, `slider` (line 23) is used to control the size of the boxes, and `button` (line 13) changes the colour of the boxes every time it is pressed.

Visor also supports direct mappings between state variables and MIDI variables. The `state_to_midi` method can be used to map a state variable to a MIDI variable. Number variables will be interpolated within their specified range while boolean variables will be toggled on or off. Mappings can also be created by clicking on the mapping icon on a state variable in the state interface, presenting an interface to select which MIDI variable should be mapped to the given state variable.

4.3.7 Other

Visor also includes a number of other notable features. One important feature is the *Learn hub*, a space for presenting documentation including a tutorial, a list of keyboard shortcuts, and an API reference. Figure 4.10 shows what the Learn Hub looks like in the application.

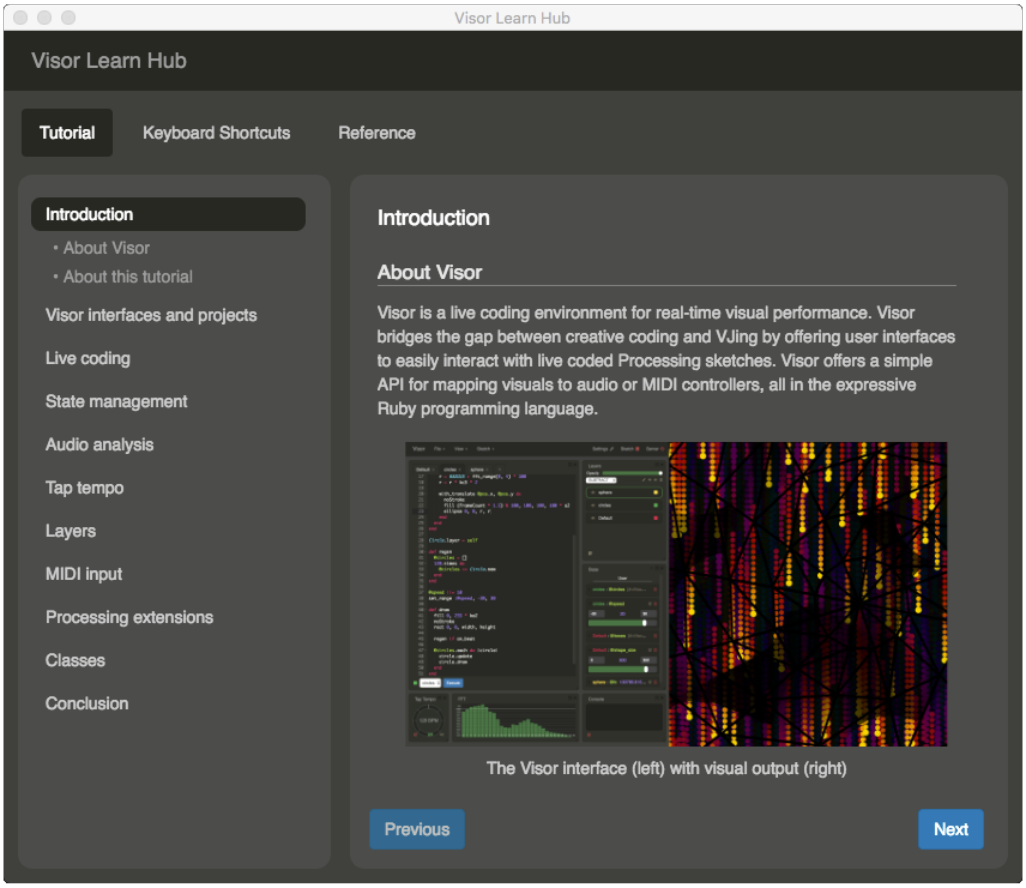

Figure 4.10: Visor Learn Hub. The Learn Hub contains documentation including a tutorial, list of keyboard shortcuts, and an API reference. Each type of documentation can be accessed from the tabs at the top of the screen while individual chapters can be navigated using the list on the left or the buttons at the bottom of the screen.

Another important feature is the customisable interface. GUI elements in Visor can be added, removed, rearranged, and resized by the performer to create a customised layout. Visor also features a console for debugging the output of `puts` statements placed in the Ruby code. Another feature is the ability to take screenshots of the visual output using a keyboard shortcut or a button in the interface. Screenshots are written to a path specified in the settings menu.

A number of additional features can also be enabled from the settings menu, along with the configuration of audio and MIDI input devices. These include Syphon, which can be used for sending the rendered output of Visor to other software for recording videos or mixing with other VJ software. Code projection can also be enabled to draw the code within the currently focused code tab on top of the rendered visuals. This same code string can be accessed in the code using the `code_string` API method. Display settings are also incorporated to enable configuration of the Processing sketch display properties including the window width, height, pixel density, and fullscreen toggle. Multiple displays are also supported, enabling creation of preview windows or duplicating the visuals onto multiple displays.

4.4 Visor: Composing Layers

This section presents a concrete example of Visor code and rendered outputs for three distinct layers. The three layers presented are called *Model*, *Particles*, and *Mask*. The visual output of each layer is presented alongside the layer's code and description. Each layer makes use of state variables, some of which are mapped to MIDI controls. The MIDI mappings are presented by annotating an image of the MIDI controller itself. The visual output when all three layers are composited together using different blend modes is then presented. Multiple images are presented to illustrate how the rendered output varies due to animations based on time, the tap tempo, audio, and MIDI inputs.

4.4.1 Model Layer

The Model layer renders a 3D model in the center of the screen (Figure 4.11) from the code in Figure 4.12. The model itself is stored in the `@model` state variable, loaded from an external file (line 4). The model rotates over time (line 15). The scale of the model oscillates over time (line 16). The colour of the model (lines 20, 22) is determined based on the HSB (hue, saturation, brightness) colour model (line 8). The hue of the colour is stored in a state variable (line 1) that is randomised on every tap tempo beat (line 19). The thickness of the model's wireframe is determined by the amplitude of high frequency sounds (line 21). The scale of the model can be controlled using the `knob_1` MIDI knob (line 17). The transparency of the model can be controlled using the `slider_1` MIDI slider (lines 20, 22).

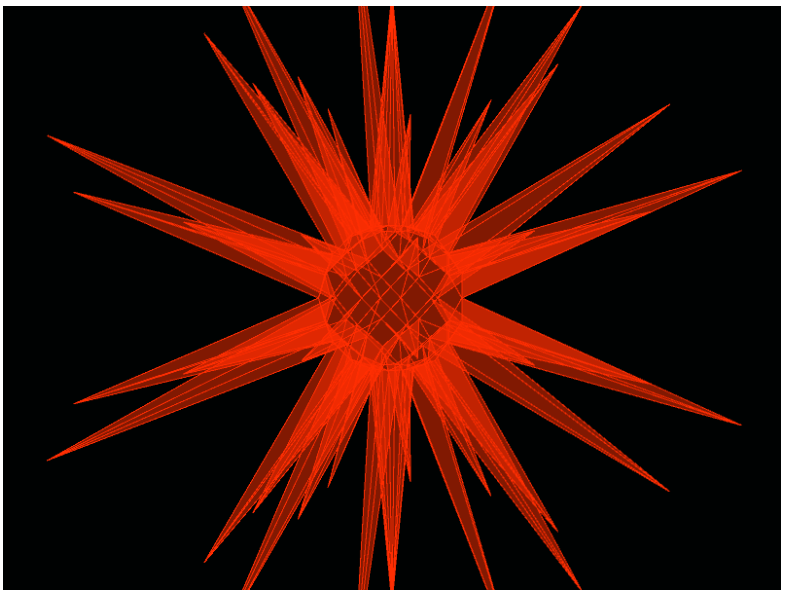

Figure 4.11: Model layer rendered independently.

```
1  @hue ||= 0.0
2
3  if @model == nil
4    @model = load_shape 'path/to/model.obj'
5    @model.disable_style
6  end
7
8  color_mode HSB, 100, 100, 100, 100
9  hint DISABLE_DEPTH_TEST
10
11 def draw
12   clear
13
14   translate width * 0.5, height * 0.5
15   rotate_y radians(frame_count)
16   scale 200 + sin(frame_count * 0.05) * 50
17   scale knob_1
18
19   @hue = random(100) if on_beat
20   stroke @hue, 100, 100, 100 * slider_1
21   stroke_weight 1 + fft_range(20, 26) * 1
22   fill @hue, 100, 100, 30 * slider_1
23
24   shape @model
25 end
```

Figure 4.12: Code for the Model layer.

4.4.2 Particles Layer

The Particles layer renders a 2D particle system where each particle is a random digit (Figure 4.13) from the code in Figure 4.14. The coordinates of 250 particles are stored in the `@positions` state variable (line 1). Each particle moves downwards over time (line 31). If a particle hits the bottom of the screen it will be moved to the top of the screen (line 32). The velocity

of each particle is determined by a state variable (line 8), initialised with a range (line 9), and mapped to the `knob_3` MIDI knob (line 10). The colour of each particle is based on a function of time (line 23). The transparency of each particle is mapped to the `slider_2` MIDI slider (line 23). The size of each particle is determined by the amplitude of low frequency sounds (line 27). A parallax scrolling effect is created by making larger particles move faster while smaller particles move slower (lines 21, 27, 31), giving the illusion of depth. Particles leave a trail that fades over time, achieved by mapping the transparency of the background colour to the `knob_2` MIDI knob (line 18). This works because transparent background colours only partially clear the last rendered frame, unlike opaque background colours which completely clear the last rendered frame. Figure 4.15 shows another variation of the rendered layer.

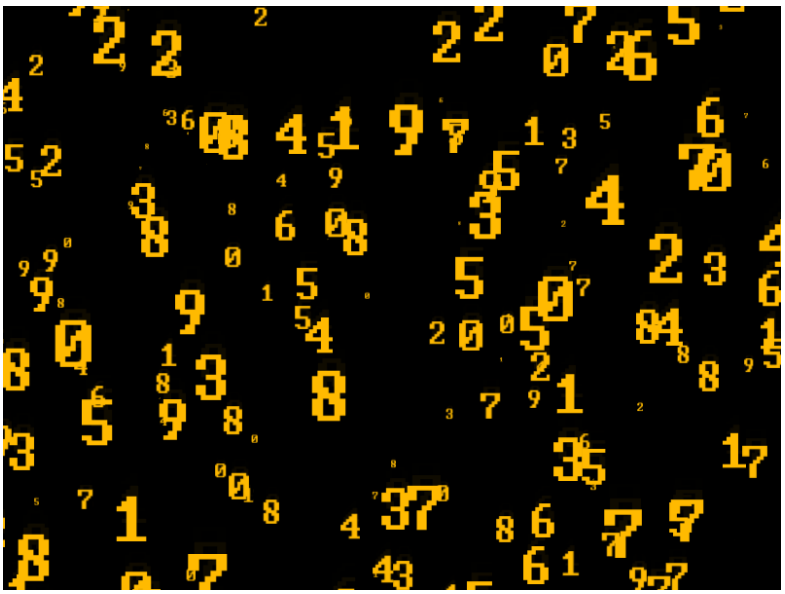

Figure 4.13: Particles layer rendered independently.

```
@positions ||= 250.times.map do
  PVector.new(
    random(-100, width + 100),
    random(-100, height + 100)
  )
end

@velocity ||= 15
set_range :@velocity, 0, 50
state_to_midi :@velocity, :knob_3

color_mode HSB, 100, 100, 100, 100
text_font createFont('Code', 64, nil, nil)
text_size 64
text_align CENTER, CENTER

def draw
  background_transparent 0, 255 - 255 * knob_2

  @positions.each.with_index do |position, i|
    norm = i / @positions.size.to_f

    fill frame_count * 0.3 % 100, 100, 100, 100 * slider_2
    no_stroke

    with_translate position.x, position.y do
      scale norm * fft_range(0, 6) * 1.5
      text (i % 10).to_s, 0, 0
    end

    position.y += @velocity * norm
    position.y -= height + 200 if position.y > height + 100
  end
end
```

Figure 4.14: Code for the Particles layer.

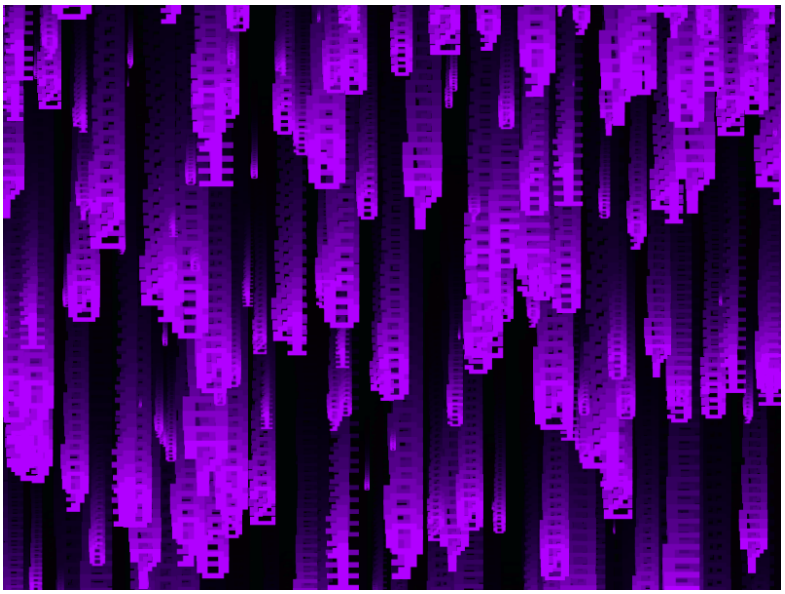

Figure 4.15: Particles layer rendered independently.

4.4.3 Mask Layer

The Mask layer, represented by the code in Figure 4.16, renders a set of rings that ripple outward from the center of the screen (Figures 4.17 and 4.18) with the intention to be used for masking. The shape of all the rings can be either a circle or a diamond (line 23, 25). Which shape is drawn is determined by the `@shape` state variable (line 1) that is mapped to the `button_1` MIDI button (line 2) such that pressing the button toggles which shape should be drawn (line 22). The width of the rings can be controlled using the `knob_4` MIDI knob (line 13). The ripple effect is created by offsetting the size of the rings by a function of the tap tempo (line 20). The rings are coloured white (line 12) to indicate which part of the screen should be masked. The background colour is based on a grayscale value mapped to the `slider_3` MIDI slider (line 7) so that the mask can be faded in or out.

```
@shape ||= false
state_to_midi :@shape, :button_1

rect_mode CENTER

def draw
  background_transparent 255 - 255 * slider_3

  translate width * 0.5, height * 0.5
  rotate QUARTER_PI

  stroke 255
  stroke_weight 100 - 100 * knob_4
  no_fill

  num_rings = 8
  ring_spacing = 1.5 * width / num_rings.to_f

  num_rings.times do |i|
    ring_size = i * ring_spacing + ring_spacing * beat_progress

    if @shape == true
      ellipse 0, 0, ring_size, ring_size
    else
      rect 0, 0, ring_size, ring_size
    end
  end
end
```

Figure 4.16: Code for the Mask layer.

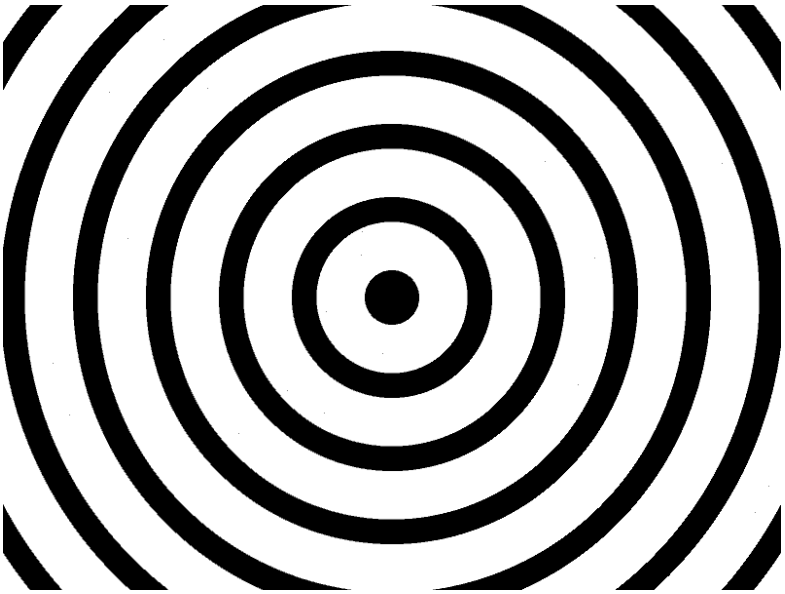

Figure 4.17: Mask layer rendered independently.

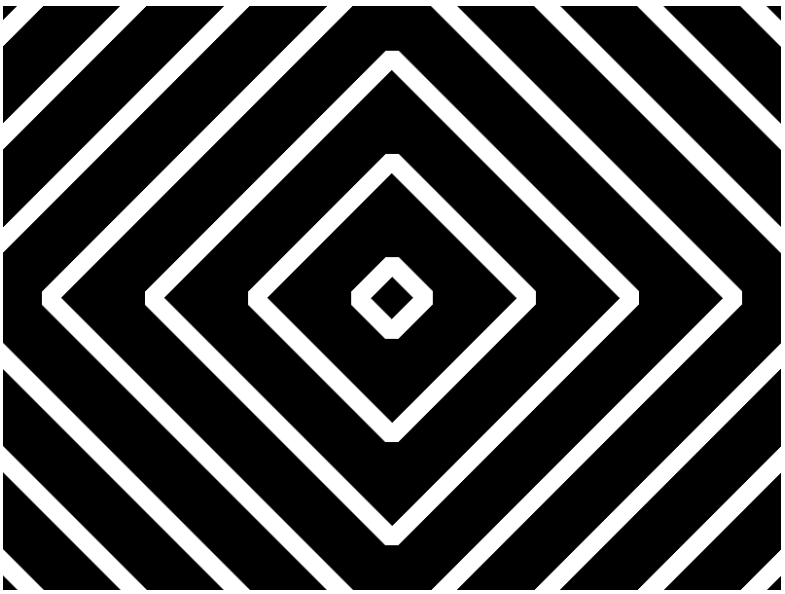

Figure 4.18: Mask layer rendered independently.

4.4.4 MIDI Control

Visor's support for MIDI inputs is used to map parameters of the three layers to individual knobs, sliders, and buttons of a MIDI controller. Figure 4.19 presents the MIDI controller used for this example and annotates which parameters and MIDI variables are mapped to which controls.

Figure 4.19: MIDI controller annotated with the names of the mapped parameters and MIDI variables used by the code for the Model, Particles, and Mask layers.

4.4.5 Final Composition

The three layers can be composited together to create a result that is more visually interesting than rendering the layers independently. Figure 4.20 shows the rendered result of compositing the Model and Mask layers to-

gether, demonstrating the `MULTIPLY` blend mode. The Model layer is rendered first followed by the Mask layer using the `MULTIPLY` blend mode. The result is that the Model layer is only displayed in the white parts of the Mask layer.

Figure 4.21 shows the rendered result of compositing the Model and Particles layers together, demonstrating the `EXCLUSION` blend mode. The Particles layer is rendered first followed by the Model layer using the `EXCLUSION` blend mode. The result is that both the Particles and Model layer are displayed, but the colour of the Particles layer is inverted in the parts that overlap the Model layer.

Figures 4.22 and 4.23 show the rendered result when all three layers are composited together. The Particles layer is rendered first, followed by the Model layer using the `EXCLUSION` blend mode, and finally the Mask layer is rendered using the `MULTIPLY` blend mode. The result is that both the Particles and Model layer are displayed, but the colour of the Particles layer is inverted in the parts that overlap the Model layer, and both layers are only displayed in the white parts of the Mask layer.

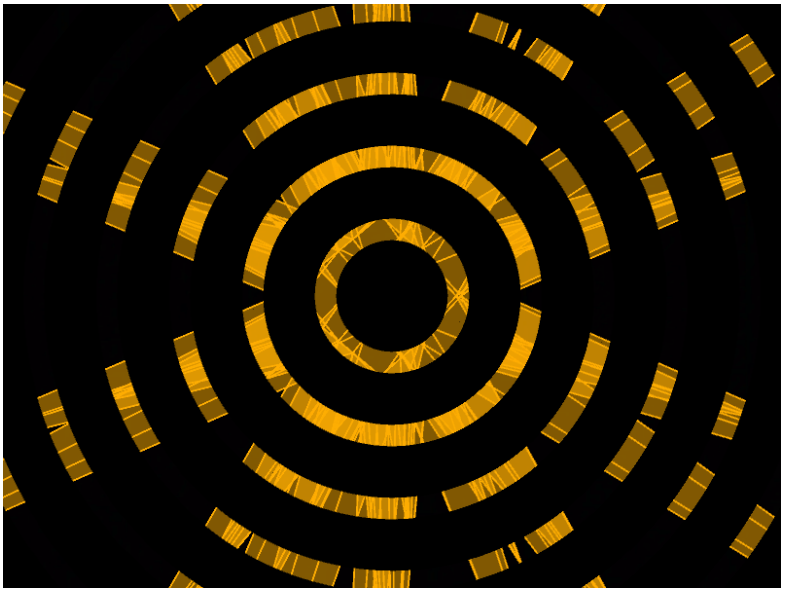

Figure 4.20: Model and Mask layers composited together.

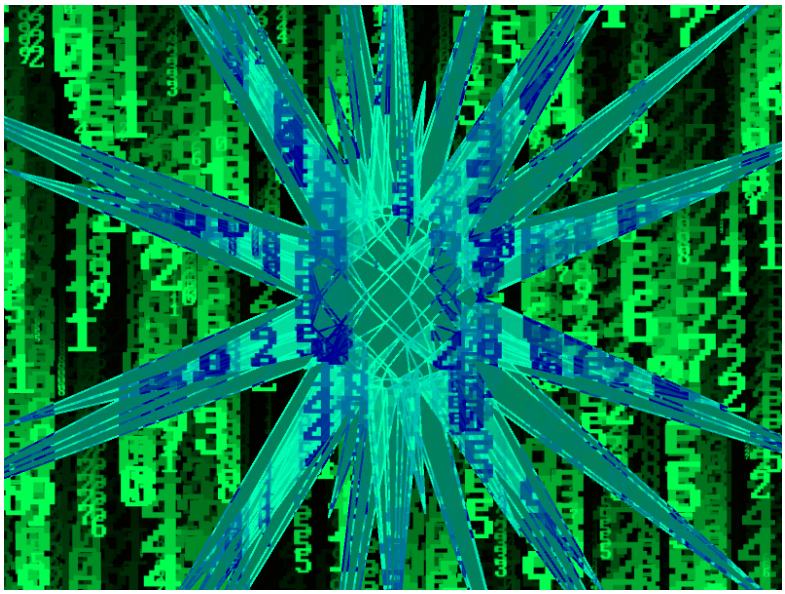

Figure 4.21: Model and Particles layers composited together.

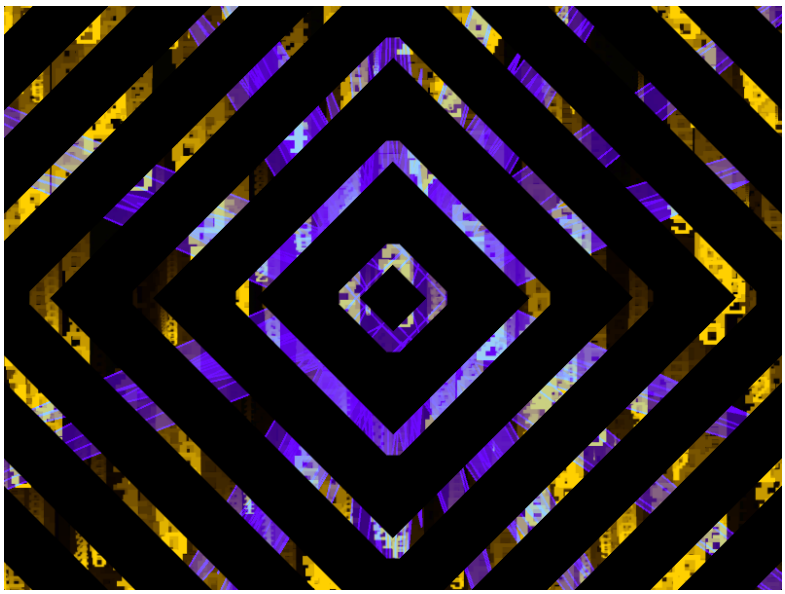

Figure 4.22: Model, Particles, and Mask layers composited together.

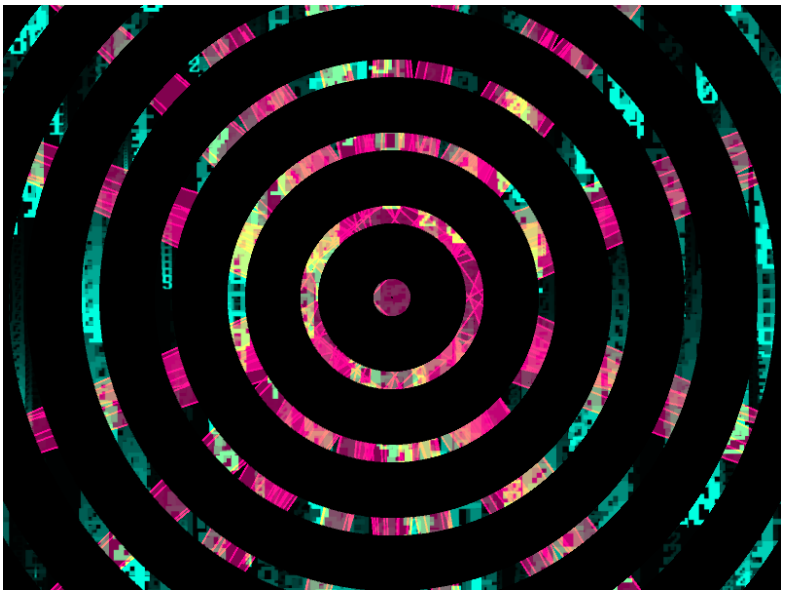

Figure 4.23: Model, Particles, and Mask layers composited together.

4.5 Visor: Implementation

This section describes in detail how Visor is implemented including its system architecture, notable third party software used, approach to handling live coding, approach to handling state management, and approach to integrating with Processing.

4.5.1 Architecture

Visor is designed using a client-server architecture, as illustrated in Figure 4.24. The client application is written as a desktop application using the Electron [6] framework, enabling the use of web technologies such as HTML, CSS, and JavaScript. The client application is responsible for handling all of the GUI aspects including the code editor, state management, and other interfaces. The server application is written in JRuby

[10], an implementation of the Ruby programming language on the Java virtual machine. The server application has a number of responsibilities including hot swapping code, maintaining program state, managing input devices, and calling into the Processing API to render the visuals onto the screen. The client and server applications communicate using HTTP and WebSockets. While both client and server applications are typically run on the same machine, this architecture opens up a future possibility where the client and server are run on different machines, enabling remote live coding where the performer live codes on one machine while the output is rendered on another.

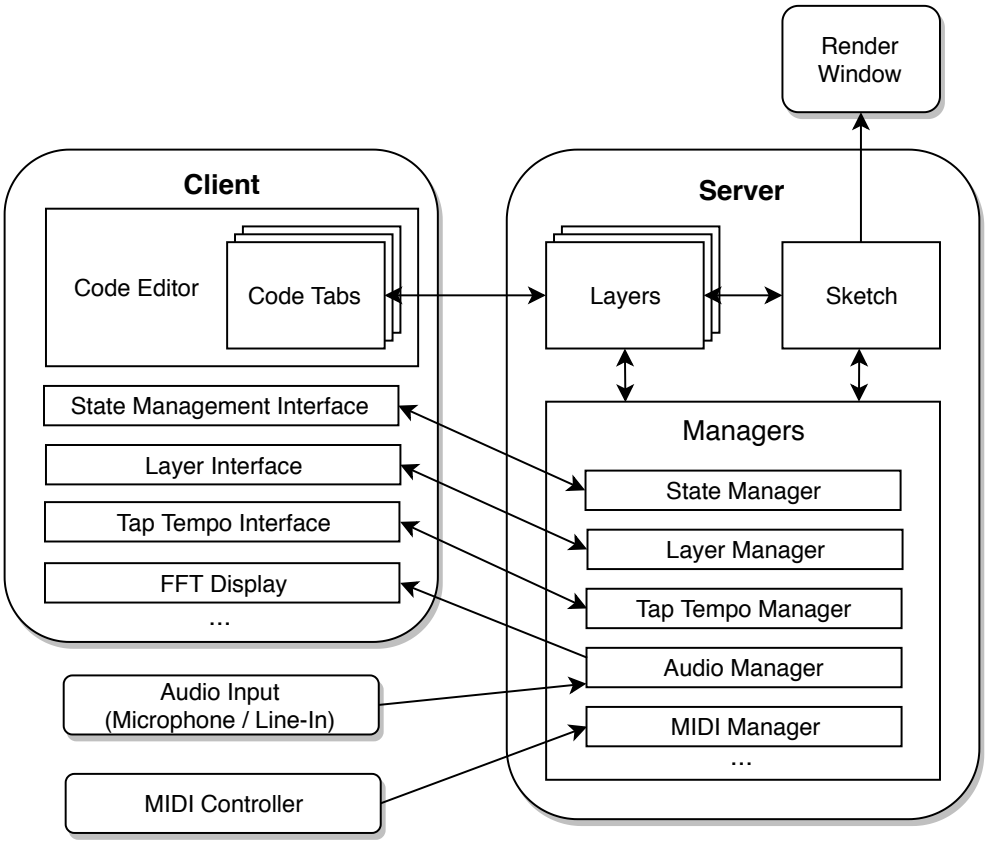

Figure 4.24: Visor's client-server architecture. The client application consists of multiple GUI components that communicate with the server. The server application consists of multiple classes that communicate with each-other, the client, the Processing API, and external inputs.

4.5.2 Client

The client application is constructed using many open source JavaScript packages including Electron. One key package that is used is the React [29] framework, enabling the Visor GUI to be built out of nested, modular, reusable components. Each component is concerned with a separate piece of the GUI, for example, individual code tabs, menus, and modals. A number of components are also maintained within a grid layout component to enable the reconfigurable interface. These components include the live code editor, state management interface, layer interface, FFT visualisation, tap tempo interface, and console. The visual style of Visor was inspired by the sleek design of modern code editors and makes use of the colour palette from the popular Monokai [22] theme.

4.5.3 Server

The server application consists of a number of objects, as illustrated in Figure 4.24. The `sketch` object supplies the `draw` method that is called every frame and allows for calls to be made into the Processing API. Next, there are the `manager` objects, each `manager` is responsible for a specific part of Visor's internal behaviour such as the FFT for audio analysis, the tap tempo algorithm, state management capabilities, layers, and more. Each `manager` provides callback methods to be called from the Processing sketch lifecycle. For example, each `manager` can specify code to be called before the `draw` method is run, after the `draw` method is run, when a WebSocket message is received, and so on.

The server also maintains multiple `layer` objects where each object represents a single Visor layer. Each `layer` object encapsulates its own `draw` method and state as defined by the performer when live coding. The layer `manager` calls the `draw` methods on each `layer` every frame in the

correct order. Delegation is used to allow each `layer` to access methods in the `sketch` and `manager` objects, thereby allowing the performer to access the Processing and Visor APIs. Each `manager` explicitly chooses which methods should be exposed to `layer` objects. For example, the tap tempo `manager` exposes the `on_beat` method. Each `manager` can also expose its own internal state to `layer` objects and the client GUI. This is useful in the case of an internal variable that should be made accessible to the performer, for example, the audio analysis algorithm exposes the parameter that controls the amount of smoothing that is applied to the FFT data.

4.5.4 Live Coding

Visor makes use of Ruby's metaprogramming features to enable live coding by evaluating arbitrary code strings to define and redefine state or program behaviour. Whenever the performer submits the contents of a code tab from the live code editor, the server application dynamically executes that code in context of the `layer` object associated with the code tab. If a `draw` method is specified in this code, then the existing `draw` method will be overwritten, essentially performing a hot swap and causing the rendered visuals to update on the next frame. As code is simply evaluated in the context of objects in Visor, code can be deleted in the code editor while the state of the program remains unchanged, therefore the code does not present a declarative view of the current state of the program. Instead, Visor's state management feature helps to keep track of what state has been defined by inspecting the `layer` objects in real-time.

4.5.5 Handling State

State in Visor is handled using Ruby's instance variables. Live code is executed within the context of `layer` objects such that any defined instance variables are placed on the `layer` object itself. This state is persisted throughout the course of the program, even between modifications to the code. To enable manipulation of the state in the client, the state is abstracted upon by the state `manager`. The state `manager` uses reflection to read from and write to any defined state on the `layer` objects or any exposed state on other `manager` objects. The state `manager` observes changes to the state and sends them to the client to be visualised in the state management interface. In turn, any changes to the state made in the state management interface are sent back to the state `manager` to be applied to the respective `layer` or `manager`. The state management interface supports manipulation of integer and floating point data types using sliders where the range of each variable is adjusted dynamically based on the variable's observed values over time or by manual adjustment from the performer. Manipulation of boolean data types is also supported using checkboxes. String data types can also be manipulated using text fields. All other data types such as arrays, hashes, or arbitrary objects are currently visualised using their string representations.

4.5.6 Integration with Processing

Due to the seamless integration between Java and Ruby in JRuby, Java methods can be transparently called from Ruby code. Therefore, there is no need for a substantial amount of explicit binding between Visor and Processing. Visor's ability to access the Processing API enables all kinds of graphics rendering including 2D and 3D primitives, text, images, shaders, and 3D models. Visor utilises a number of Processing libraries to implement internal behaviour. The MidiBus [38] is used for handling

MIDI input devices, the Syphon Processing implementation [34] is used for enabling Syphon support, and Minim [20] is used for handling audio input devices and performing the FFT algorithm.

Visor also extends Processing by making use of Ruby language features. Ruby supports the use of blocks, also known as lambda expressions. A Ruby method can accept a block as an argument and apply useful operations before and after executing the block. For example, the `with_matrix` method uses a block to replace the need for separate calls to the `push_matrix` and `pop_matrix` methods. This is helpful as forgetting to call `pop_matrix` can cause unintended program behaviour. Figure 4.25 demonstrates the traditional approach with the Processing API while Figure 4.26 demonstrates the new approach with the Visor API.

```
push_matrix
translate 100, 100
rect 0, 0, 200, 200
pop_matrix
```

Figure 4.25: Traditional approach to handling the matrix stack with the Processing API in Visor.

```
with_matrix do
  translate 100, 100
  rect 0, 0, 200, 200
end
```

Figure 4.26: New approach to handling the matrix stack using a method that accepts a block as an argument in Visor.

4.6 Visor: Development Approach

Visor was developed using a practice-based approach, similar to usability testing in a user-centered design process [49]. This meant the environment was tested in a performance context throughout development to ensure it met the needs of a performer. In this case, I was the performer. The motivation for this approach is similar to Aaron et al. [48] who developed a practice regime to maintain a research focus on live coding as a performance practice. It was important that this research also focused on developing and evaluating Visor in the context of live performance due to Visor's intention to embody CJing. A practice-based approach can be used to validate the effectiveness of features, identify features that need improvement, and inform the development of new features, as demonstrated by the approach used by the Sonic Pi and Palimpsest environments [47, 54]. Both of these environments were also evaluated by reflecting on their use in live performance. This thesis takes a similar approach to evaluating Visor, as presented in Chapter 5.

The practice-based development approach was achieved through my regular attendance of a meetup called Art∼Hack [1], a weekly event that provides a space for like-minded individuals to hack together on projects related to art and technology. As some of the regular members of the collective were electronic musicians, they often performed live electronic music, providing an opportunity to create live visuals at the same time. In addition, the space that Art∼Hack occupied always had projectors available. This infrastructure provided by Art∼Hack and my own performance equipment meant I had all that was needed to perform with Visor on a regular basis. Figure 4.27 shows Visor in action during an Art∼Hack meetup. In addition to these regular performances, I performed with Visor at a number of organised events. The performances conducted at these events are reflected on in Chapter 5 as part of Visor's evaluation.

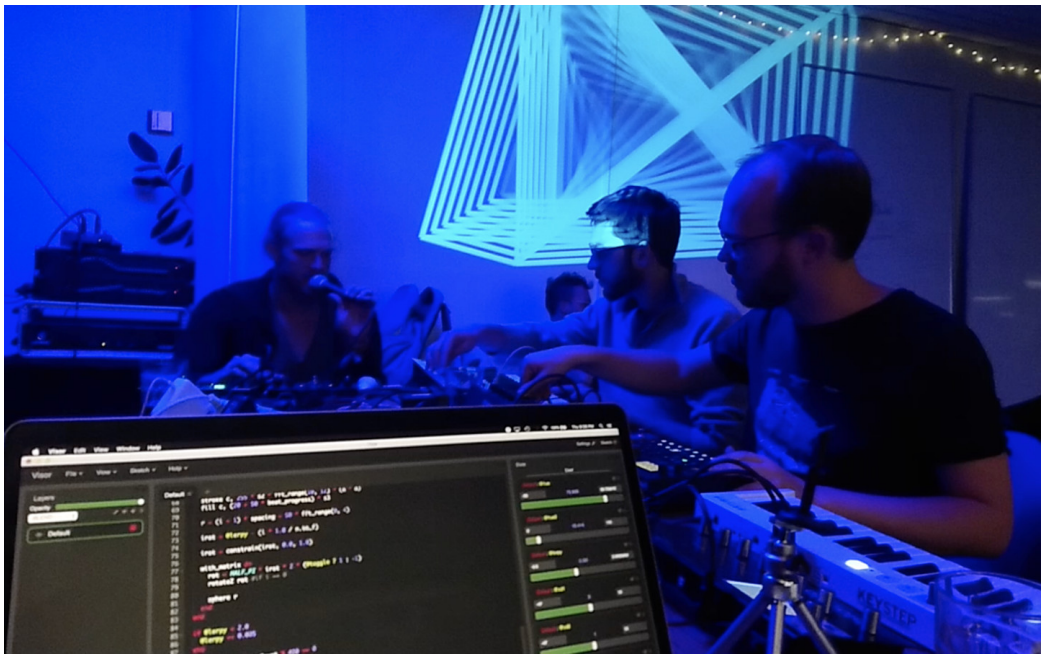

Figure 4.27: Art∼Hack meetup where Visor was often tested in a live performance context. Live music is being produced by members of the collective using a variety of electronic musical instruments. Visor (bottom-left) is used to render visuals that are then projected (top) to accompany the music.

Performing with Visor on a regular basis provided an opportunity to validate the effectiveness of existing features, identify features that needed improvement, and inform the development of new features. For example, the need for allowing Visor to support multiple MIDI input devices had not been considered until working with Livestock Pixel [16]. Allowing multiple MIDI devices was crucial for our collaboration during the performance at the Vertigo gig, as described in Chapter 5. Another example is shown by an improvement to the usability of the Visor API. The API method for querying the sum of a range of FFT bands was `fft_range` and the method for querying the average of a range of FFT bands was `fft_range_avg`. Based on performances with Visor, it was identified that the method for querying the sum was not used while the method

for querying the average often was. Based on this observation, the API was consolidated to provide only one method for querying a range of FFT bands. The finalised method was called `fft_range` and it queried the average of a range of FFT bands. Using Visor in live performance provided a means to identify usability issues, bugs, and validate the need for features such that the effectiveness of the environment could be improved.

Being a member of Art∼Hack also enabled opportunities to demonstrate Visor to the public. Figure 4.28 shows Maker Faire Wellington [18] where Visor was demonstrated to the public as part of the Art∼Hack stall. Visor managed to capture the attention of many people, in particular children, who really enjoyed playing with the MIDI controller and watching the visuals evolve in unison.

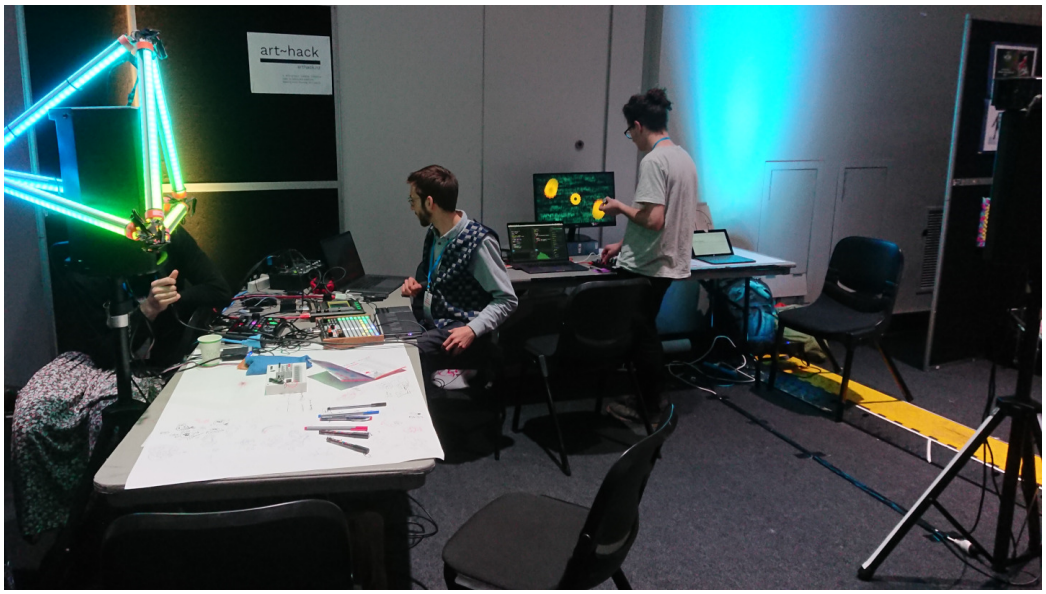

Figure 4.28: Maker Faire Wellington where Visor was demonstrated to the public as part of the Art∼Hack stall.

4.7 Discussion

Visor shares traits with a number of existing software applications for live coding, VJing, and audiovisual performance. The following discussion compares CJing and Visor with these software applications.

CJing and Visor can be compared with the characteristics of expressive interaction observed in VJing practice [58, 64]. Visor presents all of the observed characteristics except for one with respect to 'visible interaction.' While Visor supports the ability to project the source code on top of the visuals or utilise the code string in the live code, it does not offer a complete representation of the performer's process due to the introduction of user interfaces hidden behind the laptop screen. The whole interface could be projected to the audience, but that may detract from the quality of the performance. This issue highlights the differing stance on visible interaction between live coding and VJing that was discussed as part of the interviews in Chapter 3.

Visor can be compared with a number of live coding environments. LiveCodeLab [60] allows for on the fly live coding where each keystroke automatically updates the rendered output. The rendered output of LiveCodeLab is a function of time and does not allow for any state to be defined between frames, thus the performer does not need to be concerned about maintaining the state. Visor takes an opposite approach and allows the performer to define persistent state, enabling complex effects such as particle systems that require lists of objects to be maintained over multiple frames.

Auraglyph [77] makes use of touch interfaces to interact with visual objects that program the musical output, distancing itself from textual interfaces when live coding. Visor makes use of similar interface abstractions through the state management interface and mappings to MIDI controllers, offering both indirect and gestural manipulation respectively.

Visor can be compared to live coding environments for producing visuals using shaders such as Kodelife [12]. Shaders can provide highly complex and engaging visuals with high performance due to making use of the capabilities of the GPU. Visor instead trades performance for usability by making use of CPU based technologies like Processing to produce visuals. Processing is aimed at novice programmers and does not require the shift in mindset towards parallel computing that shaders and GPU programming require.

Visor can also be compared against existing approaches that enable performance with Processing. JRubyArt [11] enables live coding with Processing in the Ruby language. It achieves this through a command line application that has the ability to watch a source code file for changes over time while also providing REPL like interactions. Visor instead opts for GUIs with code editing and interactive features built-in and aimed specifically for use in live coding or VJing performances.

Mother [51] provides a set of tools to enable VJing performances with multiple layered Processing sketches, illustrating code bending [52]. Visor is similar to Mother in that it allows for VJing with Processing, but it improves upon Mother by allowing new content to be live coded, instead of just using pre-written sketches. While Mother requires boilerplate code to be written to enable sketches to communicate, Visor instead abstracts away this complexity through internal behaviour and a GUI to interact with layers. What Visor does not provide is a system for sharing layers as reusable modules, which is possible with sketches in Mother, but these ideas have directly influenced the proposed CJing practice and are planned as future work for Visor. In terms of code bending, Mother uses OSC to allow Processing sketches to communicate and be composited together during live performances. CJing is also concerned with how existing content can be modified and composited together, but not from the perspective of networking multiple programs. Instead, CJing focuses on providing a single purpose-built environment where content can be live coded in a common language, as shown by Visor.

Praxis LIVE [78] can be used for live coding performances with Processing and offers a number of features that can be used for VJing. These features include support for OSC and MIDI. Praxis LIVE exhibits features of CJing by incorporating user interfaces that abstract upon live code, all while treating code as the main source of content generation. Visor differs from Praxis LIVE in that live coding is performed in Ruby and that its user interfaces focus on improving the usability of live performance. For example, by providing audio responsive tools by default and by automatically exposing state in the client rather than having to manually type code or navigate visual nodes to see it. Visor differs from Praxis LIVE in that the code does not offer a declarative representation of the live program. While Praxis LIVE offers this clear view, it loses flexibility in that the state of components cannot be maintained between iterations of the code unless the state is explicitly annotated or referenced from other components. Visor takes a different approach by focusing on REPL like behaviour where arbitrary code can be evaluated against objects instead of being used as a complete representation of objects. This approach ensures that the state is maintained by default between iterations to the code. While this means Visor gains flexible control over the state, performers cannot rely on just the code to understand the underlying program and must instead rely on features such as the state management interface to inspect the state of the program.

Visor can also be compared with VJing software such as Resolume [30]. Resolume offers support for audio responsive features such as a real-time FFT or tap tempo that can both be used to drive video effect parameters. Visor incorporates both of these audio responsive features by visualising the FFT and tempo in the client GUI, and allowing the code to access them through exposed methods to generate audio responsive content. While Resolume focuses on traditional video mixing, Visor instead allows for improvisation of content by live coding.

4.8 Summary

This chapter introduced Visor, a new environment for live visual performance. Visor was designed to embody the CJing practice, demonstrating how aspects of live coding and VJing can be combined to harness the strengths of both practices while simultaneously removing limitations identified in each practice (RQ2, RQ3 §1.1). Visor illustrates the CJing practice by implementing features that embody the three key ideas of CJing:

- *Code as a universal language* is demonstrated by Visor's focus on live coding to enable creative coding of visual content using the Processing API. In Visor, all content is represented using code, even assets such as images and 3D models are loaded using Processing API methods. Code can be organised into separate layers where each layer is responsible for a separate visual element. Layers can then be mixed or blended together, enabling the performer to mix the outputs of different pieces of code.
- *Complete content control* is demonstrated by Visor's ability to live code in the Ruby language. Live coding provides low level control where content can be improvised from scratch or manipulated at a fine-grained level. More specifically, writing a line of code can generate new visual content while editing a line of code can manipulate the existing content. GUIs and hardware controllers then provide interactions to orchestrate high level aspects of performance. The state manager provides manipulation of individual parameters while the layers provide high level functionality that controls how different visual elements should be composited together. Visor's combination of low and high level interactions enable flexible control of the final output.

- *User interfaces as an abstraction* is demonstrated by Visor's support for GUIs that abstract upon live code to support different aspects of performance. Visor maintains relationships between GUIs and code in a number of ways. The state manager detects changes to the program state by visualising the state of variables in real-time and enables interaction with them using widgets such as sliders or checkboxes. The FFT GUI visualises the state of an audio input in real-time and can be accessed in the code using the Visor API. Similarly, the tap tempo illustrates an interactive interface that can be accessed in the code to provide high level functionality.

The next chapter discusses and reflects on Visor's use in live performances to evaluate the environment's effectiveness in a live context.

Chapter 5

Live Performances with Visor

This chapter reflects on my usage of Visor in 14 live performances to evaluate the effectiveness of the environment as part of a practice-based approach (§4.6). The performances are listed in Table 5.1. The intent of most of the performances was to provide visuals to accompany music performed by DJs, live coders, and other musicians, demonstrating Visor's effectiveness in a live context. By using Visor in real performances I hoped to explore and demonstrate what it meant to perform with an environment that embodies CJing. I had never live coded or performed as a VJ before undertaking this thesis and as a result, my own performance skills have developed alongside the development of Visor.

This chapter describes my typical performance setup, my approach to using Visor in live performances, and my reflections on using Visor in live performances. I have reflected on a number of valuable insights, usability issues, and areas for improvements that were observed with respect to each of Visor's core features. The utility of Visor as an environment for live coding, VJing, and CJing (RQ2, RQ3 §1.1) is also discussed along with issues raised about the design of CJing environments and the broader CJing practice.

Table 5.1: Performances conducted throughout this thesis. 'From scratch' performances were fully improvised while 'Prepared' performances made use of pre-written code. Please note that this list excludes many casual or impromptu performances that occurred such as those at the Art∼Hack meetup.

Date	Performance	Location	Type	Approach	Video link
23/06/2018	Art∼Hack Winter Expo	Newtown Community & Cultural Centre, Wellington, New Zealand	Exhibition	From scratch	
13/10/2018	FREAKS 001	Wellington, New Zealand	Gig	From scratch	[8]
14/11/2018	Computer Graphics Meeting	Victoria University, Wellington, New Zealand	Research group meeting	From scratch	
17/11/2018	Burrowing Pufferfish Party	Wellington, New Zealand	Private party	Prepared	[2]
01/12/2018	Vertigo	Valhalla, Wellington, New Zealand	Gig	Prepared	
16/01/2019	ICLC 2019	Nave De Terneras, Madrid, Spain	Algorave	From scratch	
25/01/2019	VIU: Algorave	Hangar.org, Barcelona, Spain	Algorave	From scratch	
27/01/2019	livecodenyc Hosts In Exile	New River Studios, London, United Kingdom	Algorave	From scratch	
28/01/2019	Sonic Pi meets Visor	Streamed from Cambridge, United Kingdom	Livestream	From scratch	[31]
01/02/2019	Sound Night 09: Algorave	panke.gallery, Berlin, Germany	Algorave	From scratch	
09/02/2019	Taniwhas's Den 2019 Mainstage	Hinakura, Martinborough, New Zealand	Festival	Prepared	[37]
10/02/2019	Taniwhas's Den 2019 Cliff	Hinakura, Martinborough, New Zealand	Festival	From scratch	
17/02/2019	TOPLAP 15th Birthday	Streamed from Wellington, New Zealand	Livestream	From scratch	[40]
06/03/2019	Eyegum Wednesdays	San Fran, Wellington, New Zealand	Gig	Prepared	[7]

5.1 Performance Setup

My typical performance setup consisted of a number of hardware and software components. The hardware components are shown in Figure 5.1. A MacBook Pro laptop was used to run the Visor software. A Novation Launch Control XL was used as a MIDI controller, offering 8 sliders, 24 knobs, and 16 buttons. A USB sound card was used to get a line-in from the sound desk. If no line-in was available then the laptop microphone was used instead. The line-in or the microphone was then configured as the audio input device for Visor's FFT. Visor's support for multiple displays was often used to create preview windows depending on the physical environment. An additional preview window was used if I could not clearly see the projected visuals. For example, in the ICLC 2019 Algorave I was positioned facing the audience while the visuals were projected behind me, as shown in Figure 5.2. In addition to Visor, the Syphon Recorder [35] software was occasionally used to record the rendered visuals.

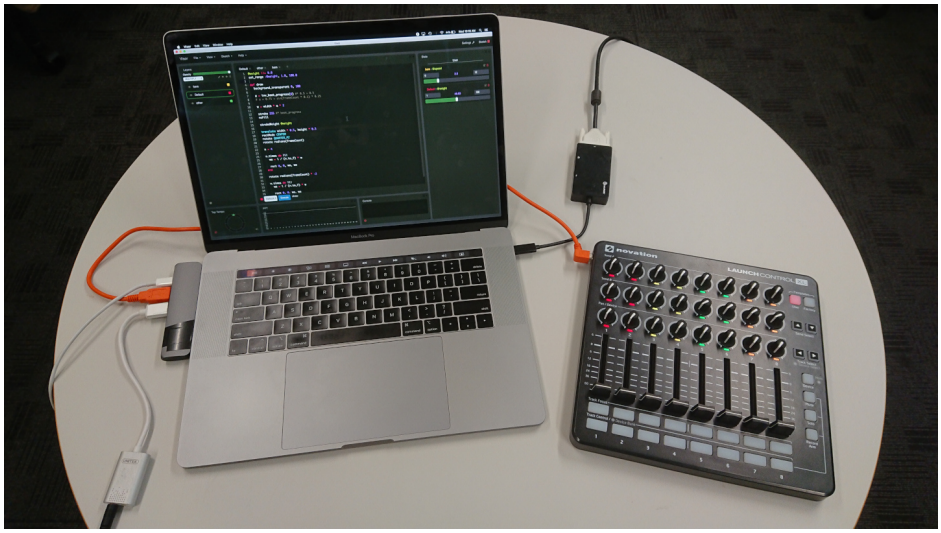

Figure 5.1: My typical setup for live performance with Visor.

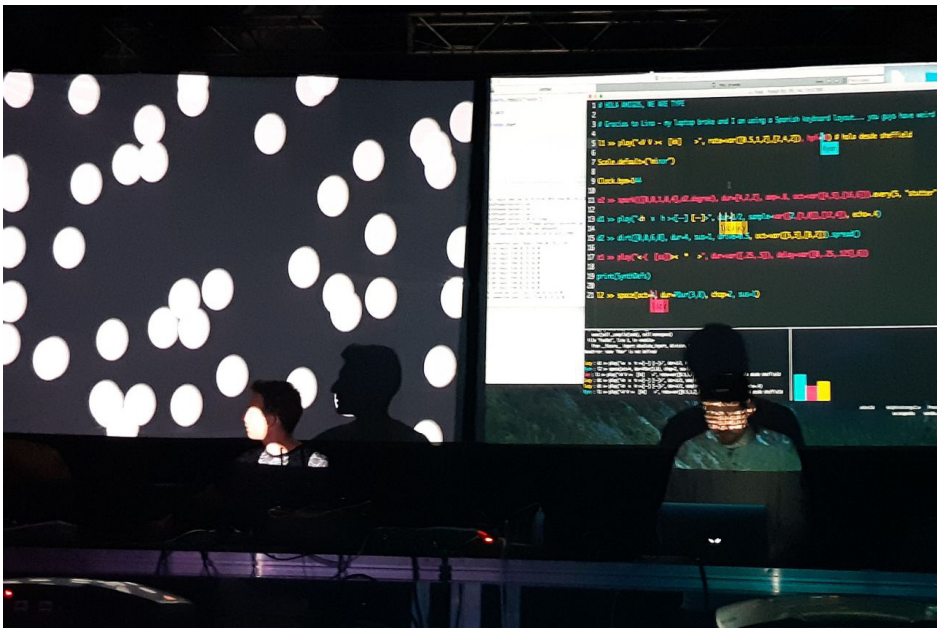

Figure 5.2: ICLC 2019 Algorave performance. Shows live coding of visuals by myself (left) and live coding of music by TYPE [43] (right). Photo credit: Steven David Delvalle.

Projectors were always provided by the venue and were either connected directly to the laptop or routed through another computer running the Resolume VJ software. Resolume was used to projection map the rendered output of Visor's onto complex surfaces. This approach was used for both performances at the Taniwha's Den 2019 festival. The Taniwha's Den 2019 Cliff performance involved projecting the visuals rendered from Visor onto a large limestone cliff face. This offered a novel live coding and VJing experience. A photograph of Visor in action on the cliff face is shown in Figure 5.3 while Figure 5.4 shows what the cliff face looks like during the day. The Taniwha's Den 2019 Mainstage performance involved projecting the visuals rendered form Visor onto multiple screens using multiple projectors, as shown by Figure 5.5.

Figure 5.3: Taniwha's Den 2019 Cliff performance where Visor was used to render visuals that were projected onto a large limestone cliff face. Note the reflection of the visuals in the water.

Figure 5.4: Limestone cliff face that was used as a projection surface during the Taniwha's Den 2019 festival. Note the size of the people standing at the base of the cliff.

Figure 5.5: Taniwha's Den 2019 Mainstage performance. The rendered visuals are projected across multiple screens around the DJ booth using multiple projectors. The VJ booth is situated out of shot, behind where this photograph was taken.

5.2 Performance Approach

The performances I conducted were generally approached in one of two ways. The first approach was to live code **from scratch** and was most similar to a traditional live coding performance due to starting with an empty screen. This approach was used in 10 of my performances as outlined in Table 5.1. This approach involved live coding the visual content throughout the course of an entire performance. This included the live coding of visual elements such as shapes, animations, and colours. Individual layers of content were created progressively and introduced, manipulated or removed at different points in time throughout the performance. Live coding of mappings between parameters and MIDI variables also occurred, followed by the performance of these parameters on the MIDI controller. This approach showcased the performance aesthetic of

the CJing practice where live coding can be used as a method to improvise visual instruments that are then performed using GUIs and hardware controllers.

The second approach was to perform with **prepared** code. This approach involved coding the visual content in preparation for the performance and was most similar to a traditional VJ performance due to primarily making use of existing content. This approach was used in 4 of my performances as outlined in Table 5.1. This approach involved organising visual elements into layers where parameters of each layer were assigned to groupings of controls on the MIDI controller. These performances mostly involved interacting with the MIDI controller as the content and MIDI mappings had already been defined in advance. Live coding also occurred during these performances to improvise content or to manipulate the existing content.

In addition to these two approaches, during the FREAKS 001 and Taniwha's Den 2019 Mainstage performances I invited a collaborator to perform alongside me for a small section of each performance. This collaborator focused solely on interacting with the MIDI controller while I focused on live coding and interacting with the GUI.

5.3 Visor in Action

Visor's core features are now reflected upon based on my experience using them in live performance. This reflection describes the positive aspects of each feature, a number of usability issues that were identified, a number of improvements that could be made to address these issues, and a discussion of issues that arose with respect to the overall design of CJing environments and the broader CJing practice.

5.3.1 Live Coding

Live coding was conducted during all of my performances to either improvise content from scratch or manipulate existing content. Live coding made it possible to iteratively develop visual elements where changes to the code were immediately reflected in the visuals. This was demonstrated in the TOPLAP 15th Birthday Livestream performance where visual elements were built up over time in three separate layers. Each code execution committed a new version of the code to a layer and often resulted in updating the rendered visuals. The performance began with the creation of a particle system layer, followed by an animated mask layer, and then finally a layer to render a 3D model. The remainder of the performance involved the manipulation of these three layers.

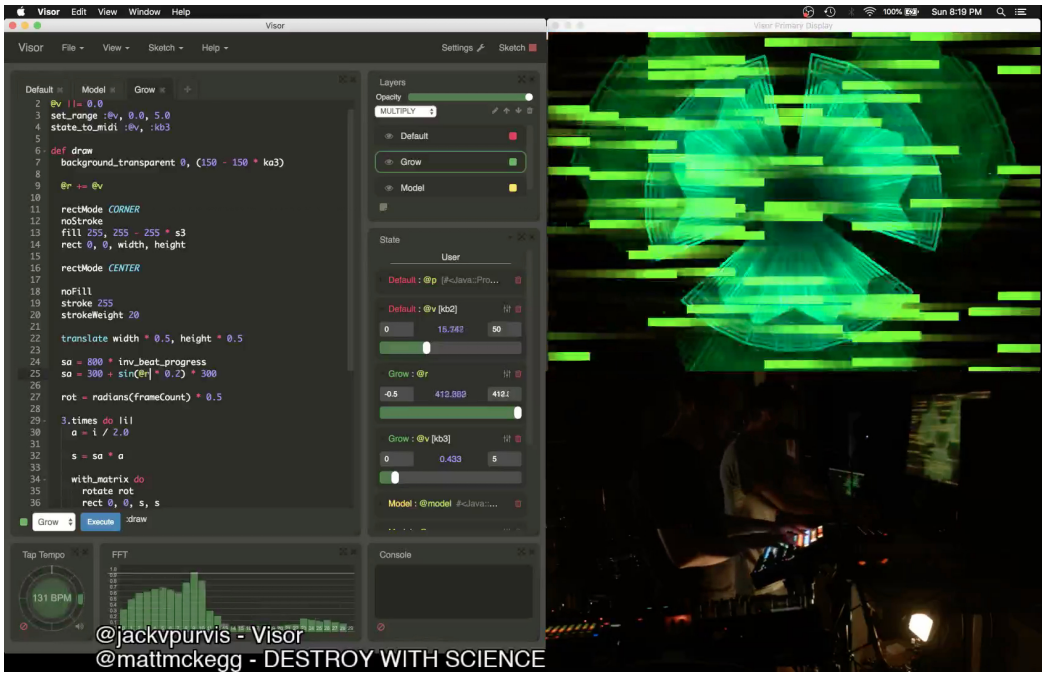

Figure 5.6: TOPLAP 15th Birthday Livestream performance. The Visor GUI (left) is displayed alongside the rendered visuals (top-right) and a camera recording of the physical performance by myself and DESTROY WITH SCIENCE (bottom-right).

5.3.2 State Management

The state management interface was primarily utilised for the visualisation that it offered during performances. Inspecting the interface was effective for debugging, for example, to check if a state variable had been initialised, if the value of a state variable was updating over time, or if a MIDI variable had been correctly mapped to a state variable. One usability issue was observed where many state variables would overpopulate the interface. State variable widgets were usually collapsed to mitigate this issue by taking up less space. This issue could be solved by providing options to filter which parts of the state are displayed or organise how the state is arranged. For example, the state could be filtered by layer or organised by type.

The MIDI controller was primarily used to interact with numeric and Boolean state variables, rather than making use of the GUI sliders and checkboxes in the state management interface. The MIDI controller was preferred as it offered an effective embodied approach to manipulating parameters and allowed for updating multiple parameters at the same time. Interacting with the GUI interface using the trackpad was less effective as it only allowed for manipulation of one parameter at any one time. It is possible the state management interaction features would have been used more in my performances if a MIDI controller was not available. The interface was used to interact with string based state variables to achieve a live typing effect where text was typed onto the rendered visuals in real-time. This effect was used in the VIU Algorave and the Eyegum Wednesdays performances to communicate with the audience. Figure 5.7 shows how the musician's name was displayed to the audience at one point during the Eyegum Wednesdays performance.

Figure 5.7: Eyegum Wednesdays performance. Shows live performance of music by DESTROY WITH SCIENCE (left) and live performance of visuals by myself (right). Recording credit: Matt Mckegg.

5.3.3 Layers

The use of layers was very effective at enabling new creative possibilities during performances. The feature was implemented between the Vertigo and the ICLC 2019 Algorave performances, meaning all earlier performances such as the Freaks 001 and Burrowing Pufferfish Party could not make use of layers, while later performances such as the Eyegum Wednesdays and Taniwha's Den 2019 Mainstage performance could make use of the feature. In the later performances, it was used to create new compositions by combining layers using different blend modes. One technique was to create a masking effect by applying the `MULTIPLY` blend mode to a layer placed above other layers. Figure 5.8 demonstrates this effect where a masking layer that rendered circular geometry was placed above two other layers, one that rendered a particle system, and another that rendered a set of 3D double helix models. The result is that the circular geometry conceals parts of the layers underneath.

Figure 5.8: Rendered visuals from the Taniwha's Den 2019 Mainstage performance.

Code tabs were also found to be an effective approach to organising layers where the code for each layer was stored in distinct code tabs. For example, in the TOPLAP 15th Birthday Livestream performance, three code tabs were used to represent the three layers. The *Default* tab held the particle system code, the *Grow* tab held the animated mask code, and the *Model* tab held the 3D model code. This clean separation of code made it easy to identify which parts of the code corresponded to which visual elements.

One issue was observed where conflicting behaviour would be caused when interacting with layers using both the GUI and the Visor API. For example, when `set_blend_mode` was used, the GUI blend mode could be unintentionally overridden after executing the code. The result was that the GUI was almost exclusively used to manipulate blend modes except for in pre-prepared content where setup code was sometimes used to initialise blend modes. This setup code was structured in a way that the API code only ran the first time it was executed, for example by checking if

a state variable had been defined. This was useful for initialising the blend mode through one code execution without overriding any later changes made through the GUI. Regardless, this issue highlights the importance of the relationship between code and user interfaces in CJing environments. Careful consideration should be taken when designing a CJing environment such that it offers the flexibility to be used by both live coding and VJing approaches without forcing the performer to adopt one approach.

5.3.4 MIDI

Support for MIDI inputs in Visor was utilised in all of the performances through the use of a MIDI controller (see Figure 5.1). This use of external hardware allowed for dynamic visuals where parameters could be tuned in reaction to what was happening in the music. This was demonstrated by my physical interactions with the MIDI controller in the Eyegum Wednesdays performance video recording.

MIDI controller interactions offered the ability to easily explore the parameter space of a live coded program, often generating unique and unexpected visual results. An example of this was demonstrated in the Eyegum Wednesdays performance where a rotating 3D model was positioned using a MIDI knob to create a scan line effect. The experience interacting with the MIDI controller felt most akin to playing an instrument, unlike live coding where immediate feedback only occurs when code is executed. Interactions with the MIDI controller also offered an instinctive feel to changing parameters in comparison to the effects of automated features like the FFT, which was more sporadic, and the tap tempo, which was more systematic. Interacting with the MIDI controller offered a VJ-like experience where different layers could be selectively mixed in or out. Parameters of each layer could be tuned individually or at the same time as parameters of other layers, resulting in changes to multiple visual elements at the same time.

One usability issue that arose during performances was the need to switch contexts between live coding and using the MIDI controller, an issue that was also observed in the results of the interviews in Chapter 3. As both contexts required almost full attention, it seemed impossible to effectively live code and perform with the MIDI controller at the same time. This was observed in the early stages of the TOPLAP 15th Birthday Livestream performance. In this performance, parameters of existing content could not be tuned using the MIDI controller while I was focused on live coding content from scratch. The opposite holds true for later in the performance where I was focused on the MIDI controller, only using live coding to make minor adjustments to the content. This issue emphasises that CJs must not only develop their skills in live coding and using the controller, but must also learn to strike an effective balance between working between the two modalities. This highlights the importance of automated features such as the FFT and tap tempo which produce dynamic effects without requiring the attention of the performer.

One feature that could be implemented to reduce the friction of context switching would be the ability to record parameter interactions. Recorded interactions could be replayed using a hotkey while live coding or automatically in time with the tap tempo. Another way to completely remove the issue is to utilise two performers where one performer live codes while the other uses the controller. This collaborative approach was experimented with during the FREAKS 001 and Taniwha's Den 2019 Mainstage performances when a collaborator was invited to perform alongside me for a small section of each performance. The result was that content could be improvised while the parameters of existing content were being performed at the same time.

Setting up mappings between parameters and MIDI variables was an important interaction for utilising the MIDI controller in live performances. Mappings were either live coded, such as in the TOPLAP 15th Birthday Livestream performance, or were defined upfront, such as in the Tani-

wha's Den 2019 Mainstage performance. Mappings were typically organised where parameters of individual layers were mapped to separate columns on the MIDI controller. The MIDI controller consisted of 8 columns where each column was made up of a slider, three knobs, and two buttons. The slider was always mapped to the layer opacity, the knobs were typically mapped to numeric parameters on the layer, and the buttons were sometimes mapped to trigger events or toggle boolean parameters on the layer. An example of this organisation is presented in Figure 4.19 of Chapter 4 where parameters of three layers are mapped to separate columns on the MIDI controller.

One issue that arose during the Taniwha's Den 2019 Mainstage performance was the inability to flexibly remap the MIDI controller to enable interaction with content improvised during the performance. This was due to the existing layers utilising almost all of the controls on the MIDI controller. This constrained the performance where live coding tended to manipulate existing layers, rather than create new content. While no attempt was made to remap any controls during the performance, there would be two approaches to doing so, each with limitations. Firstly, an existing layer could simply be overwritten with new content that utilised the same mappings, but then the old content would not be available. The code for the old content could be kept for later reuse, but even then, both sets of content could not be used at the same time, limiting creative possibilities. Secondly, a new layer could be created that maps to the same MIDI controls as an existing layer, but now the parameters of one layer cannot be updated without also updating the parameters of the other layer.

While the issue of remapping MIDI controls could be solved with hardware, for example, by using more MIDI controllers or MIDI controllers that offer more controls, Visor could instead be improved to provide a flexible solution. For example, a method could be implemented to easily re-assign layer parameters to groups of MIDI controls, such as a group per

column on the MIDI controller. With the concept of arbitrary groups of MIDI controls in place, a GUI could be used to easily update mappings between groups and layers. This way, layers could be easily repositioned on the MIDI controller during a performance.

An inconsistency in Visor's behaviour was also observed with respect to how MIDI mappings could be created using either the Visor API in the code or through the GUI. The inconsistency was that using the API methods enabled mappings to be recreated the next time Visor was run due to being persisted in the code. In contrast, mappings set up using the GUI were not persistent when Visor was shut down and needed to be reconfigured. While this inconsistency could be solved by implementing a solution that saves the state of the GUI alongside the code, the issue once again highlights the importance of the relationship between code and user interfaces in CJing environments.

Visor's support for MIDI was also used to receive MIDI messages from inputs other than a MIDI controller in two performances. In these performances I collaborated closely with musicians who provided me with MIDI data generated from their own performance setup. In one performance, Vertigo, I collaborated with Livestock Pixel [16] who performed by interacting with cubes on a table surface, as shown in Figure 5.9. To put it simply, each time a cube was placed on the table or moved, a new instrument would start or stop playing and a MIDI message was sent over a local area network to Visor, triggering a change in the visuals. In the other performance, Eyegum Wednesdays, I collaborated with DESTROY WITH SCIENCE [5] who performed with synths and other MIDI devices. Each time a kick sound was played, a MIDI message was sent over a MIDI cable to Visor, triggering beat based behaviour with the intention to replace the tap tempo.

Figure 5.9: Vertigo gig performance. Livestock Pixel (featured) is performing music using a tabletop instrument. The visuals rendered by Visor are projected onto the surrounding surfaces. Photo credit: Marie-Sophie Fabre.

5.3.5 Fast Fourier Transform

The use of the FFT was effective at creating relationships between the visuals and the music in performances. An example of how the FFT was used in the live performances was during the FREAKS 001 performance where particles grew in size based on certain frequencies. Similarly, the FFT was used to enlarge the size of rendered text during the Eyegum Wednesdays performances. A screenshot of the latter is shown in Figure 5.7. The FFT was effective because it automatically reacted to changes in the music, unlike the MIDI controller which required manual input to make changes to the visuals. The FFT visualisation in the GUI was also used to effectively identify distinct sounds in the music based on how the

amplitude of specific bands changed over time. The indexes of these bands were then used directly in the code through the `fft_range` method to access the sound data and influence the visuals.

Despite the effectiveness of the FFT visualisation, it did take a few seconds to inspect the visualisation to identify the band indexes to use as arguments for the FFT code. This is unlike the tap tempo and MIDI features which could be utilised by simply adding a `beat_progress` or MIDI variable straight into the code. The current approach to utilising the FFT in the code could be improved by implementing new functionality to the FFT visualisation. For example, clicking and dragging across a range of FFT bands and releasing the mouse could automatically generate the code required to access that FFT data. This code could then be pasted into the code editor where the cursor is placed, reducing the effort required by the performer to incorporate the feature. Alternatively, simpler API methods to access the FFT could be provided such as a *low*, *mid*, and *high* methods that each return predetermined frequency ranges.

5.3.6 Tap Tempo

The tap tempo feature was used in all of the performances and proved useful for creating animated visuals that mapped directly to the tempo of the music. These animations effectively communicated the beat of the music. Examples of how the tap tempo was used in the performances included a mask that pulsed to the beat during the TOPLAP 15th Birthday Livestream performance, particles that regenerated on the beat during the Taniwha's Den 2019 Mainstage performance, and models that scaled to the beat during the Burrowing Pufferfish Party performance. A screenshot of the latter is shown in Figure 5.10.

Figure 5.10: Rendered visuals from the Burrowing Pufferfish Party performance.

Despite the feature's convenience, some difficulty was observed when timing the taps such that the tempo looked correct. Sometimes the tempo would drift out of sync over time or would be offset from the correct value. Some music was easy to follow and predict the tempo, such as house or techno, while other types of music were harder to follow, such as jungle or drum and bass. In some occasions it took a few attempts to set a correct tempo, though the skill got easier with practice. This difficulty could be solved by improving the usability of the tap tempo in a number of ways. For example, a keyboard shortcut to clear the current tempo or a button to add a small offset to the tempo could potentially improve the performer's ability to use the feature.

Alternatively, more effort could be put towards utilising data from the musician. An example of this was conducted in an experiment during the Eyegum performance where MIDI events based on a kick sound were used to replace the tap tempo. The result was that the kick was not a good substitute for indicating when a beat occurred, as it was often un-

known when the next kick would occur. Configuring the kick was also less convenient than the tap tempo as it required the use of a MIDI cable and extra configuration in Visor. In contrast, the tap tempo does not rely on external data from the musician and therefore does not need any configuration. A better approach to utilising external data would be to synchronize the tempo with a MIDI clock from the musician if one is available. This approach would still require extra configuration upfront but it would remove the need for the performer to manually set the tempo, preventing inaccuracies and being more predictable than the kick sound experiment.

5.4 Summary

This chapter has described my use of Visor in live performances. Performing on 14 separate occasions in a variety of settings with a variety of collaborators has demonstrated Visor's effectiveness in this context, at least in combination with my own performance skills.

Two approaches to performance were described: live coding content from scratch, and performing with existing content. These approaches demonstrate respectively how Visor can be used for live coding and VJing style performances. The crossover of these two approaches also highlights Visor's demonstration of the CJing practice where aspects of both live coding and VJing can be used together in the same performance. Live coding can be used as a method to improvise visual instruments that are then performed using GUIs and hardware controllers, addressing both RQ2 and RQ3 §1.1. During the live coding style performances, I was able to make use of VJing features such as the layers, tap tempo, and MIDI controllers to interact with live coded content at a high level, addressing RQ2. Also, during the VJing style performances, I was able to improvise or manipulate existing content at a low level using live coding, addressing RQ3.

This chapter has also reflected on the usage of Visor's core features in live performance. Overall, each feature was effective to some extent as demonstrated by their use in multiple performances. A number of usability issues were also identified with respect to these features along with potential improvements to solve the issues. Two issues were also raised around the wider CJing practice. One of these issues highlighted the need for careful design of the relationship between code and user interfaces in CJing environments, an important aspect of a system that aims to offer interactions from both live coding and VJing. The issue of context switching was also raised in the CJing practice where the focus of the performer must be split between live coding and using the MIDI controller. While collaboration between two performers was demonstrated as a potential solution to this issue, it remains a notable aspect of CJing that challenges the virtuosity of the performer and invites further consideration to the design of interactions between both modalities.

The next chapter presents a study of creative coders, live coders, and VJs who experimented with Visor and provided feedback through an online feedback survey.

Chapter 6

Visor Feedback Survey

This chapter describes the results of an online feedback survey that was used to evaluate the effectiveness of Visor as part of the user-centered design process [49]. The survey solicited feedback from people with creative coding, live coding, and VJing experience who had used Visor. This feedback was used to evaluate the effectiveness of Visor.

Each Visor user who participated in the survey was asked to complete the questionnaire presented in Appendix E. The questionnaire asked each participant about their background, their outlook on some of Visor's specific features, how difficult they found Visor to learn, the context in which they might use Visor, and what they liked or disliked about Visor.

The discussion of the participants' responses was used to evaluate the effectiveness of Visor, and in turn, CJing, to help address both RQ2 and RQ3 §1.1. Results of the survey will be used to inform the future development of Visor. This research was approved by the Victoria University of Wellington Human Ethics Committee (refer to Appendix A).

6.1 Survey Procedure

Survey participants were recruited through recruitment messages placed on the Visor website [44] and within the software itself. Visor was advocated through various online forums, chat channels, and social media groups relating to live coding, creative coding, Processing, and VJing. Visor was advocated through existing networks of live coders, creative coders, and VJs. Participants who started the survey were provided with the information sheet presented in Appendix D before continuing on to the survey questionnaire. Participants were invited to participate if they had used Visor in any capacity but were also asked how much time they spent using Visor in the questionnaire. The survey questionnaire contained 15 questions consisting of both multiple choice and free form answer questions, providing a mix of quantitative and qualitative data in the results. The survey was expected to take between 5 and 10 minutes for participants to complete.

6.2 Participants

In total, six participants completed the feedback survey. While a larger number of participants would have been preferred, there were a number of factors as to why it was difficult to recruit for this study. The target audience for this study was particularly specialised in that it required people who had used Visor for some period of time while also having some experience with live coding, VJing, Processing or Ruby. It was difficult to find participants who met most of these criteria on a large scale, in particular within the time restrictions of a master's thesis. With more time, more participants could be expected to complete the survey.

Participants were asked to provide background information with respect to their experience with Ruby, Processing, live coding, and VJing. To answer these questions, participants could choose from the following options: no experience, a little experience, a fair amount of experience, or professional experience. The results are shown in Table 6.1. All six participants reported having more than three years of programming experience.

Table 6.1: Feedback survey participants background, estimated time spent using Visor, and context in which they might use Visor. For the experience questions, participants could choose from having either: no experience (None), a little experience (Little), a fair amount of experience (Fair), or professional experience (Professional).

ID	**Ruby experience**	**Processing experience**	**Live coding experience**	**VJing experience**	**Visor time**	**Visor context**
1	Little	Professional	Little	Little	1-5 hours	Performance (live coding)
2	Little	A little	None	None	1-5 hours	Performance (VJing)
3	Little	Professional	None	Little	1-5 hours	Creative coding
4	Little	Professional	Professional	Professional	5-10 hours	Teaching
5	Fair	Fair	Little	Little	10+ hours	Performance (VJing)
6	None	Professional	None	None	1-5 hours	Creative coding

6.3 Results

The following section reports the results of the feedback survey. The results were grouped based on usage, learning, each of Visor's core features, and the general user interface.

The results on the participants' usage of Visor were reported based on two multiple choice questions. The remainder of the results were reported based on direct quotes from the free form questions that solicited qualitative feedback from the participants. The direct quotes were sourced from multiple questions and grouped for the results. In terms of the free form questions that were created to solicit feedback about Visor's features, participants were asked to report what aspects of each feature they found to be effective or not.

6.3.1 Usage

The participants were asked to report how much time they had spent using Visor. Participants could choose from one of the following options: less than 1 hour, 1 to 5 hours, 5 to 10 hours, and more than 10 hours. While the time participants spent using Visor could not be controlled, it was estimated that participants would need to spend at least one hour experimenting with Visor before they could provide any feedback. The results reported that all of the participants used Visor for at least one hour before completing the survey. P1, P2, P3, and P6 used Visor for between 1 and 5 hours, P4 used Visor for between 5 and 10 hours, and P5 used Visor for more than 10 hours. The responses to this question are also presented in Table 6.1.

The participants were also asked to report the context in which they might use Visor. Participants could choose from one of the following options: performance (live coding new material), performance (VJing with precoded material) creative coding, and other (please specify). The results reported that P1 would use Visor in performances for live coding new material, P2 and P5 would use Visor in performances for VJing with precoded material, P3 and P6 would use Visor for creative coding, and P4 would use Visor for teaching. The responses to this question are also presented in Table 6.1.

6.3.2 Learning

The participants were asked to report how difficult they found Visor to learn. Participants could choose from one of the following options: strongly agree, agree, neutral, disagree, and strongly disagree. The results reported that P2 strongly disagreed it was difficult, P1 and P4 disagreed, P3 was neutral, and P5 agreed it was difficult. The participant who responded neutral (P3) commented in the extra feedback section why they chose this response, reporting that Visor's learning difficulty was dependent on the user's coding experience:

> *"Regarding the 'Visor is difficult to learn' question. I responded with 'neutral'. To explain: for someone with creative coding experience, I picked it up easily. For an absolute newbie coder, I'd put it on a par with something like Processing."* (P3)

The participant who agreed that Visor was difficult to learn (P5) had spent more than 10 hours using Visor. The participant commented in the extra feedback section the reason why they agreed, reporting that they found it difficult due to a lack of help and went on to suggest how it could be improved:

> *"I put difficult to learn as not a great deal of help, although improving. I found the most useful way to learn was to work through published videos, building the same in my copy of visor as the video progressed ... Also a bit more help in syntax changes when using Processing commands in Ruby context. Basically adding more examples would help..."* (P5)

P3, P4, and P6 reported explicitly on the effectiveness of Visor's documentation. P4 stated that they enjoyed the documentation in that *"reading through the Learn Hub and the examples jump-started where I wanted to be fairly quickly"*, P6 mentioned that *"the learning hub was super useful to get started"*, and finally P3 commented that *"the documentation is fine for somebody with*

coding experience. However, a beginner may struggle." P4 also discussed how their existing experience with Processing helped them to get started with Visor: *"learning curve was pretty good, I knew Processing already, had a sense of what I could draw and how to draw it."*

6.3.3 Live Coding

All of the participants reported on the effectiveness of live coding (§4.3.1) in Visor. P1 and P2 liked Visor's use of or similarity to Processing. P4 and P5 both enjoyed using Ruby, stating that *"I can see how it's really efficient for live-coding"* (P4) and *"Ruby like code. Fits nicely with Sonic Pi which I use a lot."* (P5). P6 discussed the benefits of being able to live code with Processing: *"Having sketch constantly updating vs having to restart sketch all the time (like what typically happens in Processing) helped a lot with keeping in the flow of exploring ideas."* P1 also mentioned how they enjoyed the *"fast iteration time"* that Visor provided through live coding. P2 simply stated how they enjoyed that Visor *"enables live coding."* P4 and P5 also raised some difficulties they experienced when live coding in Visor. P4 mentioned how they *"struggled on the improvisation side making changes quickly to get results I wanted - this could also be my lack of experience with the IDE."* P5 raised a usability issue with respect to code tabs due to how it was a *"bit tedious having to execute different code files individually."*

6.3.4 State Management

Five of the participants (P1, P2, P3, P5, P6) claimed that they found the state management features (§4.3.2) in Visor effective for particular purposes. P1 reported that *"it's neat, and I can definitely see myself using it in future."* P2, P3, P4 and P6 described a variety of tasks for which they found the feature effective. P2 mentioned the ability to set ranges on

values: *"it's an amazing feature (very convenient the fact that you can set limits in code). Great way to expose controls."* P3, P4 and P6 discussed how it was effective for interaction and visual confirmation or debugging of what the underlying code was doing: *"found state management particularly useful for experimenting with how Visor operated - like, what did what"* (P3), *"especially for debugging, and visual confirmation of values e.g: MIDI input (although I also used puts statements to see these in the console). Also for setting values with sliders."* (P4), *"useful for making high level adjustments within the sketch (eg. a speed multiplier that affects the whole sketch, that I can use the change the mood based on the music)."* (P6), and *"a nice way to keep track of variables you want to monitor (vs. just using "puts" to the console)"* (P6). P6 also discussed specifically how the state management provided high level control of a coded sketch: *"It was also useful for compartmentalizing the sketch into different key pieces I can control once they were setup."*

P1 and P2 reported a usability issue around how the state management interface could become cluttered. P1 stated that *"although it does get quite cluttered as the program gets bigger. I find myself changing numbers in the code because I have an easier time finding them."* Both P1 and P2 suggested that this issue could be solved by adding the ability to organise or hide parts of the state. They mentioned that: *"I'd probably use the state management more if there were more options for organisation"* (P1) and *"would be nice to have a way to ignore some of the variables, to avoid polluting UI."* (P2).

P4 reported that they did not use the state management: *"Oddly enough, I didn't use them. I tended to bind FFT values in where I probably may have wanted to have some other type of control."* They then went on to mention how they saw the value in the feature and reasoned why they didn't make use of it: *"I can see now where having them would have been useful instead of changing/testing values by typing. I guess I didn't want to reach for the mouse?"* This raises a potential issue where the feature requires the performer to take their hands off the keyboard, which for a dedicated live coder like P4, may not be something that they are accustomed to doing.

6.3.5 Layers

All 6 of the participants claimed that they found the layers feature (§4.3.3) in Visor effective. P1 mentioned that they enjoyed the layering and that it was *"very useful. Especially if you're used to Photoshop, the metaphor for composing layers like that makes a lot of intuitive sense."* P2 discussed how it allowed for the composition of visual content: *"it felt essential, as it allows to combine different sketches, and switch between scenes. Very useful the fact, that you can manipulate layers programmatically."* P3 talked about the novelty of the feature, in particular for someone who had come from a Processing background: *"I would imagine layers are useful in a VJ setting - but I just enjoyed messing around with them because it's a pretty novel feature compared to what I'm used to (more standard Processing stuff)."* P5 mentioned how one of the demonstration videos inspired them to use the feature: *"especially after seeing their power in video 'Visor and Destroy with Science' Once I got the hang of it, they were useful to use."* P6 mentioned that the *"blending modes especially were fun to experiment with since it was easy to make many variations with just a few sketches"* and *"it was also great to have a completely new sketch to branch into once the starting sketch got too complicated."* P4 reported how the layers and code tabs enabled effective organisation of content: *"could more easily separate visual elements more cleanly and know that changes here were not affecting changes elsewhere by accident."*

P4 and P5 also reported on a number of usability issues with the layers feature. P4 highlighted issues with both the Visor interface and API: *"because I had to flip between tabs and I was having trouble remembering how to assign classes to layers. NoStroke and noFill gave me trouble, until I kept them local to each layer's draw function."* They then suggested that Visor needed a *"key command for switching tabs"* which would alleviate the friction caused by needing to switch code tabs. The second issue could be addressed by refactoring the API or improving documentation. P5 also reported on the lack of ability to *"reset blend modes"* and that it: *"took me some time [sic] to*

spot where to delete a layer." These issues could be solved by implementing an action to reset a layer's blend mode and improving the design of the user interface respectively.

6.3.6 Fast Fourier Transform

All 6 participants claimed that they found the FFT feature (§4.3.4) in Visor effective for a variety of reasons. It was also apparent that it was a feature to which they were already accustomed. P1, P3, P5 and P6 all reported its ease of use: *"very helpful - really nice for VJing to have it ready to go, no fuss no muss"* (P1), *"it was definitely easier than my previous experiences analysing audio (in ActionScript and Processing)"* (P3), *"built in functions make it easy to use"* (P5), and *"was really easy to set up and throw into something that needed just a little bit of variation or randomness"* (P6). P2 also discussed the ability to visualise the frequency spectrum: *"it is useful to see the frequency profile of the music."* P4 discussed how it differed from what they were used to due to the resolution of the spectrum: *"I had to get used to having more options for frequency spectrum. I spent more time looking at the FFT to see where interesting sound was showing up."* P3 and P6 also reported on how Visor supported them to utilise audio reactive behaviour despite not having worked with audio visualisation much before: *"I haven't messed around much with visualising audio much, but I was able to put something together very easily."* (P3), and *"for someone with little experience with working with audio, the visualization for tempo and FFT were helpful to understand how to use them."* (P6).

P2 and P5 requested additional features for the FFT. P5 requested the ability to access *"separate left right channels"* and for an *"input preset volume setting"* to solve an issue they experienced when switching to different audio input devices to *"compensate for different levels of input."* Similarly, P2 requested to have more control over the FFT: *"I'd like to have control*

of smoothing." Visor already supports control over the level (scale) and smoothness of the FFT, but it only does so through the API. While the documentation could be improved to articulate this, a more effective approach could be to implement sliders in the GUI that directly manipulate the scale and smoothness. P2 also suggested a new way to use the FFT to influence the visuals: *"ideally would be great to set trigger points on FFT display with mouse, which could be accessible from code (global boolean variables, or event handlers), which could be used to drive changes in visualisation."*

6.3.7 Tap Tempo

Four of the participants (P2, P3, P5, P6) claimed that they found the tap tempo feature (§4.3.5) in Visor effective. P2 reported on the interface design and the use in VJing: *"tap tempo is essential for VJing. I like the design on tap tempo widget - it [sic] visible, but not distracting."* P3, P5, and P6 discussed how the feature enabled them to easily set a tempo or sync with the music: *"this made it very easy and intuitive for me to set the tempo"* (P3), *"useful to sync to input music"* (P5), and *"Tap tempo and the 'beat' features was useful to quickly get something visually interesting that was synced to the music. As someone with limited musical experience, being able to tap to set the tempo was much more intuitive than typing a number."* P1 was unsure about the tap tempo and suggested some improvements that could give them more control over the feature: *"Still on the fence about this one. I like it in theory. I think I might like it better with a few more options for manually tuning the BPM and offset instead of having to tap."*

P4 reported that they did not use the tap tempo due to not being accustomed to using it in context of visuals and the fact that the type of music they visualised would require the tempo to be updated regularly: *"I didn't use this, but I think I would have used it. I'm more a visual artist than musician, so it's not the first thing I think of. Most of the music I was testing with*

changed beat frequently and the code was taking most of my attention." P2 and P5 also raised some usability issues with the feature that could be easily addressed. P2 stated: *"I'd like tap tempo to react on 'mouse pressed' instead on 'mouse click'. This could be very personal, but for me clicking always gives a bit of delay."* P5 also brought up an issue where Visor does not persist the tempo: *"when restarting sketches I sometimes forgot to tap in a tempo and wondered what had happened to the display."* P5 also requested a feature in that it *"would be nice to have MIDI input to supply tap."* While Visor already allows tapping to be supplied through code using the `request_tap` method, the visibility of this API method is not well documented and could be improved.

6.3.8 MIDI

Only one participant, P5, successfully made use of the support for MIDI (§4.3.6) in Visor. They reported that they *"used Oxygen 8 keyboard and also configured TouchOSC for midi input."* They also stated that they found the *"last midi event on midi settings window was very useful for confirming connectivity"* along with a usability issue where *"it is a bit of a pain to have to restart Visor if the midi devices change (on a Mac)"* and went on to suggest a solution based on the way Sonic Pi handles the same situation. P2 tried to use a MIDI controller but could not use it effectively due to an issue with Visor or the controller: *"I connected my nanoKONTROL2 and Visor detected it. But after restart it stopped reacting on events."* P2 and P5 also highlighted usability issues with the MIDI support in that they *"had to write JSON file. However it's just one off"* (P2) and the need to *"reconnect and choose MIDI JSON files each time you restart"* (P5).

The remaining 4 participants (P1, P3, P4, P6) did not use the MIDI feature. This is understandable as it would not be reasonable to expect all of the participants to own a suitable MIDI device. P4 went on to justify why they did not use the feature: *"I think for the same reason as the state management*

variables. I find it faster to zip through code than reach for a controller/mouse. Sometimes the fine tuning I need may not map to a MIDI 255 range." This highlights the same issue that P4 raised with respect to the state management: the feature requires the performer to step away from the code, something a dedicated live coder may not be used to doing. P4 went on to discuss that they might use the MIDI feature if they took a more VJ style approach to performance: *"I think if I was someone who mixed layers or build things and then blended them in (like VidVox or something) then I think I would use the state MIDI controller more."*

6.3.9 Interface

Some of the participants reported on Visor's general interface and reconfigurable layout. P2 mentioned how they enjoyed that the interface was *"simple."* In contrast, P1 and P4 commented on issues with the layout, stating: *"the interface with the rearranging panels is too fiddly; would rather have something a little simpler"* (P1) and *"on load the layout seemed jumbled up, and I had to resize window to get all the elements visible in the window"* (P4).

6.3.10 Suggested Improvements

In addition to any aforementioned suggested improvements to Visor's core features, 4 participants (P1, P2, P4, P5) made suggestions for additional features that they would like to see in Visor. P1 requested that they *"would love to have an option to superimpose the code on top of the video output; it's a big part of the live coded visual aesthetic"*. While Visor does support a simplistic code projection, this comment highlights the need to more clearly communicate how the feature can be enabled from the application settings. P2 mentioned a need for an improved user experience when editing code: *"would be great to have code assist and ability to move split screen to see two*

or more parts of my project", P4 asked if there were *"feedback buffers"*, and P5 stated: *"I would love to have the ability to use OSC messages (which are very easy from Sonic Pi and TouchOSC) to control Visor"* and also for Visor to allow for *"remembering the latest state in a project"*.

6.4 Discussion

The results of the feedback survey have revealed some key insights that can be used to evaluate the effectiveness of CJing and Visor. These insights were with respect to Visor's learnability, features, usability, and CJing.

Learning: Overall, most of the participants reported that Visor was not more difficult to learn than Processing. The documentation largely proved effective as all of the participants had no problem getting started with Visor, though all of them had at least three years of programming experience and some experience with Processing or Ruby. It was suggested by one participant (P3) that the documentation would not cater to beginner programmers. Improving the documentation to account for novices would lower Visor's barrier to entry, enabling a more diverse user base and creating an opportunity for the software to be more easily applied in education.

Core features: The results have also evaluated each of Visor's core features. Overall, Visor's core features were mostly proven to be effective for their intended purposes. Participants reported enjoying the live coding experience in Visor, in particular, due to how it used Ruby and enabled fast iteration of Processing code. The state management interface proved effective for allowing interaction with program state without the need to type code, as well as for debugging through the visualisation of the state. The layers were particularly popular due to how they effectively enabled new creative explorations through combination and organisation

of coded content. The FFT visualisation also proved effective for allowing for identification of interesting sounds to map to visual elements. The tap tempo proved effective for syncing visuals to the beat of the music.

One feature that could not be evaluated to the same extent as the other features was Visor's support for MIDI. P5 managed to use their controller successfully, P2 did not get theirs working, and the remaining 4 participants did not have access to or chose not to use a controller. This highlights a disadvantage of this type of study. A better approach to testing MIDI support would be to conduct an in-person user study where a controller is provided for participants to use.

Usability issues: A number of usability issues were identified with respect to Visor's features. In some cases, potential solutions for these issues were also suggested by the participants. One of the prominent suggestions was to implement filtering in the state management interface to prevent it from getting too cluttered, the same solution that what was discussed in Chapter 5 when reflecting on Visor's use in live performance.

Another prominent issue was the need to write a JSON file to configure MIDI controllers. While writing this JSON file is a one-off, there are potential solutions that could remove the need for a configuration file. Instead, MIDI mappings could be configured in real-time based on interaction with the controller. For example, when the performer creates a mapping, the GUI could display a prompt to select which control should be mapped. The performer could then select the control by moving the desired knob or slider on the controller, or in the case of a button, simply pressing the desired button. This interaction would remove the need to create an explicit mapping file and make it easier for performers to make use of MIDI controllers in Visor.

A number of usability issues were also reported with respect to the interface's reconfigurable layout in that it was poorly responsive and fiddly to interact with. Potential solutions for these issues could be to make the in-

terface more responsive when resizing or perhaps even remove the ability to fine-tune the layout entirely. Instead, pre-determined views could be provided that offer discrete layouts for particular use cases. For example, one layout could be aimed at live coders while another is aimed at VJs.

Context switching: The participant with the most live coding experience out of all of the participants (P4) raised a usability issue regarding the need to switch contexts between using the keyboard for live coding, using a mouse to interact with the state manager, or using a MIDI controller. Switching between from the keyboard to other modalities was not something they were familiar with doing when live coding and as a result, they chose not to use these features. They went on to discuss how they would have used the MIDI, tap tempo, and state management features more if they conducted VJ style performances: *"I think If I was loading up a composed set, then this would be nice to have things ready."* This highlights that the effectiveness of Visor's features comes down to the performer's approach. A dedicated live coder may focus on the code editor for low level content improvisation, while those taking a VJ style approach may focus on the user interfaces that provide high level interactions. This emphasises the need for CJing environments to be designed flexibly with the interactions of both live coding and VJing in mind, ensuring that either approach can harness the full capabilities of the environment. For example, in Visor, performers can create MIDI mappings programmatically or through the GUI, enabling both dedicated live coders and VJs to create MIDI mappings.

Suggested improvements: A number of suggestions were also reported by the participants to implement new functionality or extend Visor's existing features. One suggestion was to allow the user to set thresholds in the FFT GUI and trigger code when the audio amplitude meets a threshold. This feature would further demonstrate how the relationship between the GUI and code can be used to enable more complex creative expression, as per the idea of *user interfaces as an abstraction* in CJing. It was also

suggested to incorporate more control over the internal FFT scale and smoothness parameters. While this could be solved by improving the documentation around the existing FFT API, a better solution would be to provide built-in GUI widgets such as sliders to manipulate the parameters without requiring the performer to write any code.

Another suggestion was to implement the ability to remember the complete state of Visor when the application is re-opened after being closed. Persisting the state of the code, user interfaces, and Visor's internal state would allow for performances to be continued from where they were left off and would solve the reported issue where MIDI controllers need to be reconfigured every time Visor is re-opened. This feature could be implemented using mechanisms for serializing and storing state, moving Visor closer to the design of a language like Smalltalk [33] where the code and system state are heavily intertwined and persistent.

Two features were also suggested that had also appeared in the results of the interviews in Chapter 3, but were not implemented in Visor. The first feature was to support the OSC protocol, allowing for communication between Visor and other applications such as Sonic Pi. The second feature was to support code assist, improving the programming experience in Visor by suggesting API methods as the user types. These suggestions validate the importance of these features and encourage their prioritisation in future Visor development.

Ease of use: One prominent theme in the results was the ease of use of some of Visor's features that was made apparent by a number of the participants (P1, P3, P5, P6). For example, the audio input for the FFT could be configured through the GUI and the built-in methods to access the data were straightforward to use. The ability to readily access features provided an improved user experience over Processing which would usually require the user to write boilerplate code to achieve the same effect. In general, the ease of use of Visor's features can be summed up based on a comment made by P6: *"It was very quick to launch Visor and just start*

making something interesting and dynamic, and not have to worry about setting up different libraries." This comment validates why a simple API that does not necessitate external libraries or extraneous boilerplate code is important for improved user experience. The overall ease of use of Visor's features demonstrates how the environment enables easy integration of coded content in performance software, an important aspect of practice that was identified from the interviews in Chapter 3.

Use cases: The participants indicated that they would use Visor for a variety of different use cases including live coding, VJing, creative coding, and teaching. This highlights how Visor can be used for both live coding and VJing style performances. The participants' feedback around Visor's core features also emphasises its effectiveness in these contexts. Firstly, the participants' supportive feedback on live coding in Visor shows that it is effective at providing low level control of coded content. Secondly, the participants' supportive feedback on features such as the state management, layers, and tap tempo, shows that it is effective at providing high level control of coded content for VJing. Further supporting this, P2 also explicitly mentioned how the combination of features equips Visor for VJing: *"has all essential features needed for VJing (tap, fft, layers)."* P6 also made explicit mention of how they enjoyed high level control of the coded content: *"I liked that many options for getting dynamic input (FFT, tap tempo, setting up buttons and sliders) and that it was straight forward [sic] to access them within the sketch. I found these features to be better to explore/control the sketch's style than typing up variables."* This comment reiterates the motivation for this thesis in that the high level control provided by features of VJing software can improve the usability of live coding.

Visor's support for features that effectively bridge aspects of live coding and VJing were also observed. For example, the layers feature was popular due to how it effectively combined coded content with GUIs to provide high level functionality, demonstrating itself to be an exemplary feature for CJing environments. Overall, Visor's effective support for both live

coding and VJing demonstrates Visor's illustration of CJing, addressing both RQ2 and RQ3 §1.1. Additionally, the results also suggest that Visor is suitable for general creative coding based on the environment's general ease of use and how it improves upon Processing by supporting live coding and programming with Ruby.

Despite Visor's general support for CJing, there is still room for improvement in terms of Visor's support for live coding. The need for code projection was brought up by P1, highlighting how a feature standard in live coding practice is still a desired aspect in a CJing environment. This is in contrast to the results of the interviews in Chapter 3 that did not place importance on the feature. This emphasises the need for future work to improve the state of the code projection in Visor.

To conclude, 3 participants (P2, P5, P6) also expressed some overall positive comments on what they thought about Visor: *"This is an amazing application!"* (P2), *"I think it is a fantastic system, already very useful, and with tremendous potential for further development."* (P5), and *"There is nothing I can think of that I didn't enjoy. The overall experience was quite good."* (P6). These comments motivate the further development of Visor.

6.5 Summary

This chapter has evaluated the effectiveness of Visor from the perspective of creative coders, live coders, and VJs who provided feedback through an online survey. Each of Visor's core features were reported to be effective for their intended purpose except for the support for MIDI, which was only used successfully by one participant. A number of usability issues were also identified with respect to Visor's features along with potential solutions to solve the issues or improve the features in general. Visor also proved effective for supporting aspects of both live coding and VJing, demonstrating CJing, and addressing RQ2 and RQ3 §1.1.

With respect to the broader CJing practice, the issue of context switching was discussed where the performer must split their attention between live coding and other modalities, similar to the issue that was identified by the results of the interviews (Chapter 3) and the reflection of Visor's use in live performances (Chapter 5). This time the issue was expanded to include the need to switch between typing and using the mouse to interact with the GUI, rather than just the need to switch between a laptop and a MIDI controller. The need to design CJing environments such that they cater to live coding, VJing, and hybrid approaches to performance was also emphasised. Overall, the survey results were highly supportive of Visor and encourage further development of the software and exploration of CJing.

The next chapter concludes this thesis by discussing the overarching strengths and limitations of Visor and the underlying CJing practice along with future work.

Chapter 7

Conclusions

The goal of this thesis was to explore how combining live coding and VJing could harness the strengths of both practices while simultaneously removing limitations identified in each practice, as introduced in Chapter 1. The limitation of live coding was due to its focus on low level improvisation of content by coding in textual interfaces, impairing usability by not allowing for high level manipulation of content. The limitation of VJing was due to its focus on high level manipulation of pre-rendered content by interacting with GUIs and hardware controllers, limiting VJs by not allowing for content to be improvised from scratch or manipulated at a low level.

This thesis proposed the *code jockey* practice (*CJing*) to combine aspects of live coding and VJing, enabling flexible performances where content could be controlled at both low and high levels. In CJing, a performer known as a *code jockey* (*CJ*) interacts with code, GUIs, and hardware controllers to improvise or manipulate visual content in real-time.

This thesis has introduced Visor, a new environment for live visual performance that embodies CJing, as presented in Chapter 4. Visor's core features include the ability to live code visual content, a state management interface, the ability to organise code into layers, support for real-time

audio analysis using an FFT, a tap tempo, and a framework for configuring MIDI input devices. Visor's features were designed to embody three key ideas of CJing practice: *code as a universal language*, *complete content control*, and *user interfaces as an abstraction*.

Visor's design was informed based on the results of a study that interviewed seven practicing live coders and VJs, as presented in Chapter 3. This study revealed a number of key themes with respect to important features, interactions, and aspects of performance practice. Some of the prominent subthemes and codes included: audio responsiveness, communication protocols, updating parameters, use of hardware devices, content arrangement, the usability of code in performance, visible interaction, and people. These themes help to present a clearer understanding of the needs and expectations of live coders and VJs, addressing the first research question: *"What are the needs and expectations of performers who practice live coding and VJing?"* (RQ1 §1.1).

Visor was evaluated using two methods to answer the remaining research questions: *"Can features of VJing software improve the usability of a live coding performance?"* and *"Can live coding be used as an effective method for improvising visual content or manipulating existing content during a VJ performance?"* (RQ2, RQ3 §1.1). The first evaluation was a reflection on my usage of Visor's in 14 live performances, as presented in Chapter 5. These performances demonstrated Visor's ability to effectively produce visuals that accompanied music performed by DJs, live coders, and other musicians in a live context. These performances were approached as a live coder would by improvising content from scratch and also as a VJ would by interacting with or remixing existing content. The crossover of these approaches showcases the CJing practice in action where aspects of live coding and VJing practice can be used together in the same performance. Live coding is used as a method for improvising visual instruments that are then performed using GUIs and hardware controllers, addressing RQ2 and RQ3.

The second evaluation was the analysis of feedback from six live coders, VJs, and creative coders who used Visor, as presented in Chapter 6. The feedback suggested that each of Visor's core features were effective for their intended purpose except for the support for MIDI, which could not be evaluated to the extent of the other features. One feature that was particularly popular amongst the participants was the support for layers. The layers were effective due to how they enabled new creative explorations through combination and organisation of coded content. The effectiveness of layers to provide high level functionality by bridging coded content and GUIs show them to be an exemplary feature for CJing environments. Overall, the results from the feedback survey were highly supportive of Visor and demonstrated that the environment was effective at conducting aspects of both live coding and VJing, demonstrating CJing, and addressing RQ2 and RQ3.

The evaluations of Visor identified two prominent issues with respect to the broader CJing practice. The first was to emphasise the need for careful consideration of the relationship between code and user interfaces when designing CJing environments. APIs and GUIs in CJing environments should be designed to avoid conflicting behaviour when used in conjunction. The second was the issue of context switching that highlighted the need for performers to split their focus between live coding, interacting with GUIs, and using hardware controllers. The friction of context switching could be reduced with improvements to the software or by focusing on collaboration where multiple performers work across the different modalities at the same time. Nevertheless, context switching remains a notable aspect of CJing practice that challenges the virtuosity of the performer.

7.1 Research Contributions

This thesis makes the following research contributions:

- The **CJing practice**, a new hybrid practice that combines live coding and VJing.
- **Visor**, a new environment for live visual performance that demonstrates the CJing practice by combining features of live coding and VJing software.
- An **evaluation** of Visor, based on a reflection of my own live performances conducted throughout this thesis and the results of an online feedback survey completed by six creative coders, live coders, and VJs.

7.2 Future Work

This thesis presents a number of opportunities for future work. The evaluations of Visor have raised usability issues to be fixed amongst suggestions to improve existing features or implement new features. In addition, some features and interactions that were identified as part of the interviews in Chapter 3 were not implemented into Visor and left for future work. Four of the more interesting ideas are now presented.

- **Effects:** Support for effects was identified as an important feature in VJing based on the interview results. Some common video effects include the ability to mirror visuals, apply a kaleidoscope effect, perform edge detection, and adjust hue or saturation. Like the layers feature, implementing easy to use effects in Visor would open up many creative possibilities for performers. One approach to implementing effects would be to utilise a shader language such as GLSL.

While Visor could support a number of effects by default, it would also make sense to support live coding of custom GLSL shaders too, as many other live coding environments provide. Effects could then be applied to any layer using the layer GUI or Visor API methods. Effects could even be parameterised to enable dynamic interactions using the state manager, MIDI controllers or other inputs. Support for configuring effects through the GUI would also further emphasise the idea of *user interfaces as an abstraction* in CJing, while the ability to apply existing effects and live code custom effects would further emphasise the idea of *complete content control*.

- **Collaboration in CJing performance:** Collaboration was used briefly in two of the live performances conducted with Visor where one performer live coded and the other interacted with the MIDI controller. The result of these collaborations demonstrated how content could be improvised using live coding while parameters of existing content were simultaneously being manipulated on the MIDI controller. This removes the friction that occurs in solo CJing performances where the performer needs to continuously switch contexts.

  This insight motivates further exploration of what it means to collaborate in CJing performance. If CJing offers a new performance aesthetic where live coding is used to improvise visual instruments that are then performed with hardware controllers, how does collaborative CJing implicate this aesthetic? Perhaps some novel experiences can arise during performance. For example, interesting effects may occur where the live coders updates something in a way that surprises the controller user, causing them to react in unexpected ways. In addition, perhaps the controller user explores the parameter space of the program or triggers content in a way that the live coder did not expect. The exploration of collaboration in CJing performances could investigate how different communication protocols

affect the performer's collaboration. For example, how should MIDI controller mappings be communicated to the controller user as they are live coded on the fly? Perhaps the breakdown in communication presents idiosyncrasies that offer novel performance experiences, similar to the unpredictability of generative algorithms or user error that can occur in live coding practice.

Another way to introduce collaboration in CJing performances would be to incorporate multiple live coders. Live coders could work in parallel to build up different sections of code that are then combined into the final output. In the case of Visor, each performer could work on separate layers that are then composited together. This approach to collaboration would require a method for multiple computers running Visor to communicate. Fortunately, Visor's client-server architecture could be easily adapted to support this type of communication. Overall, collaboration in CJing performances offers multiple avenues for exploration. The use of collaboration would also be interesting to observe if CJing was applied in the context of an environment for live musical performance instead of visuals.

- **OSC support:** Support for the OSC protocol was identified as an important feature based on the interview results and was a suggested feature in the results of the Visor feedback survey. Support for OSC in Visor would enable the environment to communicate with other applications that also support OSC. For example, Sonic Pi could be used to send OSC messages when events occur in the music and Visor could listen for these messages. Visor could then be live coded to trigger or manipulate visuals when messages are received, similar to how MIDI messages are currently being interpreted. The result of using OSC to integrate Visor with Sonic Pi would allow live visuals to react to specific events in the music.

- **Content library:** Support for a content library was identified as an important feature in VJing based on the interview results. Support for a content library would enable performers to easily store new content or browse for existing content to utilise during a performance. The library could store different types of content including layers, general code snippets, and assets such as images or 3D models. Content could even be previewed in the GUI, another important feature identified in the interview results.

  Some reworking of the current API may need to take place to ensure that content is as modular as possible and therefore easy to incorporate from the library during live performance. For example, layers could be decoupled from specific MIDI controls to make remapping to new controls as easy as possible, an issue that was discussed as part of the reflection on live performances. If content is modularised then it would also be easy to share in an online ecosystem where performers can download, fork, and remix content, similar to how video loops are treated by VJs, and how code is treated by software developers. OpenProcessing [24] demonstrates this kind of ecosystem by enabling Processing code to be uploaded and shared amongst creative coders online. A content library GUI that provides high level organisation of content would also further emphasise the idea of *user interfaces as an abstraction* in CJing, while modularised layers or code snippets that can be shared would also further emphasise the idea of *code as a universal language*.

Appendices

Appendix A

Human Ethics Documents

ResearchMaster

Human Ethics Application

Application ID :	0000025996
Application Title :	Live Coding for Visual Performance
Date of Submission :	N/A
Primary Investigator :	Jack Purvis; Principal Investigator
Other Personnel :	Dr Stuart Marshall; Head of School (or delegate)
	Dr Craig Anslow; Supervisor
	Prof James Noble; Supervisor

Research Form

Application Type

1.

IMPORTANT: Please select type of research below and click on 'Save' to access the rest of the form.

*

Research

Application Details

Category

B

3. Application ID

0000025996

5. Title of project
*(Click the ? icon for more info)**

Live Coding for Visual Performance

6. School or research centre*

Engineering and Computer Science

7. Personnel*

1	Given Name	
	Surname	Purvis
	Full Name	Jack Purvis
	AOU	Engineering and Computer Science
	Position	Principal Investigator
	Primary?	Yes

8. Are any of the researchers from outside Victoria?*

- ○ Yes
- ◉ No

9. Is the principal investigator a student?*

- ◉ Yes
- ○ No

Student Research

9a. What is your course code (e.g. ANTH 690)?*

CGRA 591

9b. Supervisor*

1	Given Name	Craig
	Surname	Anslow
	Full Name	Dr Craig Anslow
	AOU	Engineering and Computer Science
	Position	Supervisor
2	Given Name	James
	Surname	Noble
	Full Name	Prof James Noble
	AOU	Engineering and Computer Science
	Position	Supervisor

9c. What is your email address? (this is needed in case the committee needs to contact you about this application)*

jack.v.purvis@gmail.com

Project Details

10. The following question is meant to help applicants consider their research application and any protocols that should be uploaded and to help committee members review the application. Please check the box if if your research:

- ☐ Is an anonymous questionnaire
- ☑ Uses tertiary students as participants
- ☐ Is a health or disability research project
- ☐ Includes Māori participants, or otherwise has an impact on Māori
- ☐ Includes participants from another significant cultural group, or has an impact on that group
- ☐ Uses highly sensitive information (see Policy for definition)
- ☐ Collects or uses human tissue, including blood, saliva and genetic material
- ☐ Uses noninvasive physiological procedures (e.g., EEG, heart rate monitor)
- ☐ Uses equipment (e.g., TMS) that may temporarily alter mental function
- ☐ Administers substances (e.g., food, alcohol, placebo pill) to be ingested by participants

11. Does this application relate to any previous applications submitted to an ethics committee?*

- ○ Yes
- ◉ No

12. Describe the aims and objectives of the project
Provide a brief summary in plain language of the purpose, research questions/hypothesis, and objectives of your project. *

The aim of the research project is to develop a new software environment for live coders. The new environment will be used for creative coding of visuals in live audio-visual performances. The new environment will provide an alternative to existing environments and also answer two research questions:

RQ1: How can we improve state transition capabilities in live coding?
RQ2: How can integrating VJ (Video Jockey) practices improve the capabilities of live coders?

We hypothesize that the our new environment will improve the capabilities of live coders. The new environment will demonstrate the implications of the research questions and will be evaluated to conclude whether it improves the capabilities of live coders.

The aim of the proposed interviews is to get a better perspective on the current practice of live coders and VJs. The collected data will help to inform the design of the new software environment.

13. Describe the benefits and scholarly value of the project
*Briefly place the project in perspective, explaining its significance and worthwhile outcomes. Include how this project will build on relevant literature, including references if appropriate.**

The project will provide benefits to the live coding community by providing an alternative software environment for use in the practice. The environment could then be used for entertainment in the form of live audio-visual performances as seen by the Algorave movement [1], or for education in the form of teaching creative coding as seen by the live coding software SonicPi [2]. The developed software environment will be new intellectual property with potential commercial value.

The project will provide scholarly value by exploring two research questions which have not been seen to be explicitly discussed in the live coding research field. The results and implications of the project can then be contributed to the live coding research field.

[1] COLLINS, N., AND MCLEAN, A. Algorave: Live performance of algorithmic electronic dance music. In Proceedings of the International Conference on New Interfaces for Musical Expression (2014), pp. 355–358.

[2] AARON, S. Sonic pi performance in education, technology and art. International Journal of Performance Arts and Digital Media 12, 2 (2016), 171–178.

14. Explain any ethical issues your research raises for participants, yourself as the researcher, or wider communities and institutions, and how you will address these. This is an opportunity to present what you think the key risks are in your project and show how you have taken them into account.*

The proposed research is not seen to pose any ethical risks to any party.

Key Dates

If approved, this application will cover this research project from the date of approval

15. Proposed start date for data collection*

14/05/2018

16. Proposed end date for data collection*

31/10/2018

17. Proposed end date for research project*

09/04/2019

Proposed source of funding and other ethical considerations

18. Indicate any sources of funding, including self-funding (tick all that apply)

Internally: by a University grant, such as the University Research Fund
Externally: funding from an external organisation for this project, or a scholarship awarded by an external organisation
Self-funded: paying for research costs such as travel, postage etc. from your own funds

- [ ] Internally funded
- [ ] Externally funded
- [x] Self-funded

19. Is any professional code of ethics to be followed?*

- (•) Yes
- ( ) No

19a. Name the professional code(s) of ethics *

Standard Human Computer Interaction methods will be adopted. See Jakob Neilsen - Usability Engineering 1994.

20. Do you require ethical approval from any other organisation, such as another tertiary institution in New Zealand or overseas, or a District Health Board?*

- ( ) Yes
- (•) No

Data Collection and Recruitment

21. Please select all forms of data collection you will use in your project*

- [x] Interviews
- [ ] Focus groups
- [ ] Questionnaires
- [ ] Observation
- [ ] Other

22. Provide an explanation of the sampling rationale for your study.
E.g. representative sampling of a particular population, purposive sampling, convenience sampling. Include here your eligibility criteria for potential participants -- will there be particular criteria for participants to be included in your study, or criteria that will exclude them? *

Purposive sampling will be used to select interview participants that best align with the eligibility criteria:
- Have worked with live coding environments.
- Have worked with visual jockey (VJ) software.
- Have worked with creative coding environments.
- Have computer programming experience.

The criteria will be flexible to allow for interviewing a broader range of people, for example: a person who does not have computer programming experience but has experience with VJ software.

23. How many participants will be involved in your research?
If there will be several different groups of participants, please specify how many groups and how many participants in each group. *

There will be up to 10 participants involved. It is expected around 2-3 of the participants will be tertiary students.

24. What are the characteristics of the people you will be recruiting?*

Participants will ideally be people who:
- Have worked with creative technologies.
- Have experience creating audio or visuals using software or code.
- Have experience in live performance.
- Have have some form of computer programming experience.

Participants will be a mix of professionals and tertiary students.

25. Outline in detail the method(s) of recruitment you will use for participants in your study. *Include here how potential participants will be identified, who will contact them and how. Please include copies of all advertisements, online posts or recruitment emails in the 'Documents' section.* *

All participants have been identified through existing professional relationships or by direct recommendations. Participants will be identified in the collected data by their name, email and stage name. Participants will only be contacted by the primary investigator. Participants will be recruited and contacted using email or existing social media or mobile phone connections where appropriate. Despite existing relationships with some of the participants, they will not be coerced into participation in the study. All participants will be provided with the information sheet and consent form before meeting for the interview and allowed to decide for themselves if they would like to participate.

26. Explain the details of the method of data collection. For example, describe the location of your research procedures, if appropriate (e.g. where your interviews will take place). If necessary, upload a research protocol in the 'Documents' section.*

The intended method of data collection is through interviews. The primary investigator will be the sole interviewer. The proposed location for each interview will be a public space such as a Cafe or at Victoria University, agreed upon by the participant and the interviewer. If it is more convenient for the participant and the interviewer, an online Skype call may be used. No interviews will be conducted in private residences.

27. Will your research project take place overseas?*

- ○ Yes
- ◉ No

28. Does the research involve any other situation which may put the researcher at risk of harm (e.g. gathering data in private homes)?*

- ○ Yes
- ◉ No

Participants and Informed Consent

29. Does your research target members of a vulnerable population?
*This includes, but is not limited to, children under the age of 16, people with significant mental illness, people with serious intellectual disability, prisoners, employees and students of a researcher, and people whose health, employment, citizenship or housing status is compromised. Vulnerability is a broad category and encompasses people who may lack the ability to consent freely or may be particularly susceptible to harm.**

- ○ Yes
- ◉ No

30. Have you undertaken any consultation with the groups from which you will be recruiting, regarding your method of recruitment, data collection, or your project more widely?*

- ○ Yes
- ◉ No

31. Will your participants receive any gifts/koha in return for participating?*

- ○ Yes
- ◉ No

32. Will your participants receive any compensation for participation (for instance, meals, transport, or reimbursement of expenses)?*

- ◉ Yes
- ○ No

32a. Give details of the compensation participants will receive.*

Reimbursement for travel, parking, childcare or other expenses required to attend the interview. Reimbursement will be paid in the form of supermarket vouchers. The value of each voucher will be $10. Each participant will receive a voucher.

33. How will informed consent be obtained? (tick all that apply to the research you are describing in this application)*

- ☐ Informed consent will be implied through voluntary participation (anonymous research only)
- ☑ Informed consent will be obtained through a signed consent form
- ☐ Informed consent will be obtained by some other method

Treaty of Waitangi

How does your research conform to the University's Treaty of Waitangi Statute? (you can access the statute from Victoria's Treaty of Waitangi page)*

This research conforms to the University's Treaty of Waitangi Statute by acknowledging and abiding by the principles set out in the statute and in particular, the Principle of Equality and the Principle of Reasonable Co-operation. This research is not intended to be a study that specifically involves Māori participants, all potential participants that meet the eligibility criteria (Q22, Q24) will be treated equally during the recruitment and interview process, even if they are of Māori descent.

Minimisation of Harm

34. Is it possible that participants may experience any physical discomfort as a result of the research?*

- ○ Yes
- ◉ No

35. Is it possible that participants may experience any emotional or psychological discomfort as a result of the research? (E.g. asking participants to recall upsetting events, viewing disturbing imagery.)*

- ○ Yes
- ◉ No

36. Will your participants experience any deception as a result of the research?*

○ Yes
◉ No

37. Is any third party likely to experience any special hazard/risk including breach of privacy or release of commercially sensitive information? This may occur in the instance participants are asked to discuss identifiable third parties in the research.*

○ Yes
◉ No

38. Do you have any professional, personal, or financial relationship with prospective research participants? *

◉ Yes
○ No

38a. Give details and indicate how you will manage this.*

The interviews will adhere to the supplied script, focusing on the research topic. As the existing relationships I have with the prospective participants are not directly related to the research topic, the relationship should not interfere with the results.

Existing relationships will not interfere with each participants ability to freely consent or decline to participate in the study. All participants will be provided with the information sheet and consent form before meeting for the interview and allowed to decide for themselves if they would like to participate.

39. What opportunity will participants have to review the information they provide? (tick all that apply)*

☐ Will be given a full transcript of their interview and given an opportunity to provide comments
☐ Will be given a full transcript of their interview and NOT given an opportunity to provide comments
☑ Will be given a summary of their interview
☐ Other opportunity
☐ Will not have an opportunity to review the information they provide

Confidentiality and Anonymity

40. Will participation in the research be anonymous?
*'Anonymous' means that the identity of the research participant is not known to anyone involved in the research, including researchers themselves. It is not possible for the researchers to identify whether the person took part in the research, or to subsequently identify people who took part (e.g., by recognising them in different settings by their appearance, or being able to identify them retrospectively by their appearance, or because of the distinctiveness of the information they were asked to provide).**

○ Yes
◉ No

41. Will participation in the research be confidential?
*Confidential means that those involved in the research are able to identify the participants but will not reveal their identity to anyone outside the research team. Researchers will also take reasonable precautions to ensure that participants' identities cannot be linked to their responses in the future.**

○ Yes
◉ No

42. Will participation in the research be neither confidential nor anonymous, and participants will be identifiable in any outputs or publications relating to the research? *

◉ Yes
○ No

42a. Please tick all that apply to your research.*

☑ Names will be confidential, but other identifying characteristics may be published with consent
☐ Participants will be referred to by association with an organisation rather than by name
☐ Participants will be named in a list of interviewees
☐ Participants will be named and their contribution attributed to them

42a. Please explain how this will occur and ensure this is clear to participants on your information sheet.*

The participant's stage name will be the only piece of identifying information that can be published. Participants do not need to provide their stage name in the interview and will be given the option to consent to having it published or not in the consent form. If a participant chooses not to provide a stage name, they will be identified in published material by an anonymous code, e.g: P2.

Access, storage, use, and disposal of data

43. Which of the following best describes the form in which data generated in your study will be stored during the study?
*See help text for guidance on these terms. Further info available on human ethics website**

- (●) Identifiable
- ( ) Potentially identifiable
- ( ) Partially de-identified
- ( ) De-identified
- ( ) Anonymous
- ( ) Other

44. Which of the following best describes the form in which data generated in your study will be stored after the study is completed?
*See help text for guidance on these terms. Further info available on human ethics website**

- ( ) Identifiable
- (●) Potentially identifiable
- ( ) Partially de-identified
- ( ) De-identified
- ( ) Anonymous
- ( ) Other

45a. Proposed date for destruction of identifiable research data (i.e. the date when data will be de-identified and personal information on participants destroyed)

*

30/04/2019

45b. Proposed date for destruction of de-identified research data, including anonymous data

*

30/04/2019

46. Will any research data will be kept for longer than 5 years after the conclusion of the research?*

- ( ) Yes
- (●) No

47. Who will have access to identifiable, de-identified or anonymous data, both during and at the conclusion of the research?*

- ( ) Access restricted to the researcher only (whoever is named as PI)
- ( ) Access restricted to researcher and their supervisor
- (●) Access restricted to researcher and immediate research team, e.g. co-investigators, assistants
- ( ) Other

48. Are there any plans to re-use either identifiable, de-identified or anonymous data?*

- ( ) Yes
- (●) No

49. What procedures will be in place for the storage of, access to and disposal of data, both during and at the conclusion of the research? (Check all that apply)
Information regarding appropriate data storage is available on the human ethics website. Note that storing research data on USB drives is strongly discouraged for security reasons. *

- [x] All hard copy material will be stored securely e.g. in a locked filing cabinet
- [x] All electronic material will be held securely, e.g. only on University servers, password protected
- [x] All hard copy material will be appropriately destroyed (e.g. shredded) on the dates given above
- [x] All electronic data will be deleted on the dates given (ITS should be consulted on proper method)

Dissemination

50. How will you provide feedback to participants?*

Via email

51. How will results be reported and published? Indicate which of the following are appropriate. The proposed form of publications should be indicated to participants on the information sheet and/or consent form*

- [x] Publication in academic or professional journals
- [x] Dissemination at academic or professional conferences
- [x] Availability of the research paper or thesis in the University Library and Institutional Repository
- [ ] Other

52. Is it likely that this research will generate commercialisable intellectual property?
*(Click the ? icon for more info)**

- (●) Yes
- ( ) No

Documents

53. Please upload any documents relating to this application. Sample documents are available on the Human Ethics web page.

Please ensure that your files are small enough to upload easily, and in formats which reviewers can easily download and review. To replace a document, click the tick in the column to the right of the document title. A green arrow will appear - click this arrow to upload a new document. To add a new document click on 'Add New Document', at top right of the documents window. Then enter the document name in the box that appears and click the green tick. A green arrow will appear to the right of the file name which allows you to upload the new file.

Please also collate all your documents into one PDF or Word file, and upload as a new document. This should be labelled as 'Combined Documents'.*

Description	Reference	Soft copy	Hard copy
Participant information sheet(s)	Jack Purvis - Interview Information Sheet.pdf	✓	
Participant consent form(s)	Jack Purvis - Interview Consent Form.pdf	✓	
Combined Documents	Jack Purvis - Combined Documents.pdf	✓	
Interview questions or guide	Jack Purvis - Interview Schedule.pdf	✓	
Amended consent (19 Nov 2018)	Jack Purvis - Evaluation Interviews Consent Form.pdf	✓	
Amended interview info (19 Nov 2018)	Jack Purvis - Evaluation Interviews Information Sheet.pdf	✓	
Amended interview schedule (19 Nov 2018)	Jack Purvis - Evaluation Interviews Schedule.pdf	✓	
Amended questionnaire (19 Nov 2018)	Jack Purvis - Feedback Survey Questionnaire.pdf	✓	
Amended recruitment (19 Nov 2018)	Jack Purvis - Feedback Survey Recruitment Messaging.pdf	✓	
Amended survey info (19 Nov 2018)	Jack Purvis - Feedback Survey Information Sheet.pdf	✓	
Amendment explanation (19 Nov 2018)	Jack Purvis - further explanation.pdf	✓	

Amendment or extension request (available only for approved applications)

43. Are you applying for an extension, an amendment, or both?*

- ○ Extension
- ◉ Amendment
- ○ Both an extension and an amendment

43a. What changes would you like to make for this application?*

Method 1: User test the software

12. The aims of this project have progressed - we now also want to evaluate the software that has been built with user tests
18. Sources of funding - I have been offered funding to travel to Madrid for the International Conference on Live Coding to present a paper. I will use this opportunity to interview experts in the field
21. Forms of data collection - Observation checked
22. Explanation of sampling rationale - Purposive sampling will be used to select participants that have experience with some of the use cases of the software. These use cases include live coding, creative coding, VJing, and audiovisual art. Participants will be professionals or hobbyists
23. How many participants - There will be up to 3 expert participants interviewed from overseas and up to 3 more participants sourced locally
25. Method of recruitment - Some participants have been identified through existing professional relationships or by direct recommendations. These participants will be recruited and contacted using email or existing social media or mobile phone connections where appropriate. Other participants have been identified as authors of published work in live coding or as active users in online live coding forums. These participants will be recruited using email or direct messaging on the online forums. Despite existing relationships with some of the participants, they will not be coerced into participation in the study. All participants will be provided with the information sheet and consent form before meeting for the interview and allowed to decide for themselves if they would like to participate. Participants will only be contacted by the primary investigator.
26. Method of data collection - As well as interviewing the participant, they will be presented with my software and given a guided walkthrough of how it works so they can play with it and provide feedback. The proposed location for each interview will be a public space such as a Cafe or at Victoria University, agreed upon by the participant and the interviewer. Interviews performed internationally will also be held at a public space or at the location of the conference (a research lab). If it is more convenient for the participant and the interviewer, an online Skype call may be used. No interviews will be conducted in private residences. Audio will be recorded during the interview as well as a screen recording showing how the participant interacted with the software during the user test
27. Research will take place overseas is now checked

Method 2: Online feedback survey

10. Is an anonymous questionnaire is now checked
12. The aims of this project have progressed - we now also want to evaluate the software that has been built using an online feedback survey
17. Proposed end date of data collection - extended till 02/04/2019 to allow for collection of survey data
21. Forms of data collection - Questionnaires checked
23. How many participants - The exact number of participants to fill in the survey is unknown at this point but we estimate anywhere from 10 to 25
25. Methods of recruitment - Links to the online survey will be displayed on the software website and inside the software itself
26. Method of data collection - Online anonymous survey with questionnaire including likert scales and open-ended answer based questions
32a. Participant compensation details - Online survey participants will not receive any compensation
33. Informed consent - Online survey participant consent will be implied through voluntary participation is now checked
40. Will participants be anonymous - Yes, online survey participants will be anonymous
43/44. Form of data storage - Some of the data stored is now anonymous
46. Research data kept for longer than 5 years - Yes, the online survey data may be kept for much longer
48. Plans to re-use anonymous data - Yes, the online survey data may be re-used in future research and publications

43b. Please enter the date you are submitting this request*

19/11/2018

Please check that you have answered all mandatory questions and have saved the application before submitting your form. Any new or amended documents (e.g. Participant Information Sheet) to be added to your application should be emailed to ethicsadmin@vuw.ac.nz before submission. To submit your form, click on the Action tab and then click on Submit for review

44. Do you have a second amendment/extension request to make?

- (•) Yes
- ( ) No

44a. What additional changes do you wish to make? If this amendment includes a request to extend the end data of your approval, please specify your new end date.

I wish to extend the proposed end date for data collection through my online feedback survey AND the proposed end date of this research project to 08/05/2019.

The reason for this is to account for the fact that my thesis deadline has been extended by one month (from 09/04/2019 until 08/05/2019) and I wish to continue collecting data through the online survey until my project is complete.

The previous end date for data collection through the online survey was 02/04/2019 and the end date for the research project was the 09/04/2019.

44b. Please enter the date you are submitting this request

09/04/2019

45. Do you have a third amendment/extension request to make?

- ( ) Yes
- ( ) No

This question is not answered.

46. Do you have a fourth amendment/extension request to make?

- ( ) Yes
- ( ) No

This question is not answered.

Incident Reporting

Research teams must immediately advise the Human Ethics Committee if an adverse incident occurs in the course of their research project.

Adverse incidents are instances of potential or actual physical harm to participants or researchers; emotional harm or distress to participants or researchers; and any other unforeseen events that raise ethical issues.

A full incident report must be completed and emailed to ethicsadmin@vuw.ac.nz. You can download this form here (link to be added). After you have emailed the form, please complete the questions below, then click on the **Action** tab and click **Report Incident**

Do you have an incident to report?

- [ ] Yes

This question is not answered.

Do you have a third incident to report?

- [ ] Yes

This question is not answered.

Please go to the **Action** tab and click on **Report Incident** to complete the process.

TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

Phone 0-4-463 6028
Email judith.loveridge@vuw.ac.nz

MEMORANDUM

TO	Jack Purvis
FROM	Dr Judith Loveridge, Convenor, Human Ethics Committee
DATE	10 April 2019
PAGES	1
SUBJECT	Ethics Approval Number: 25996 Title: Live Coding for Visual Performance

Thank you for your application for ethical approval, which has now been considered by the Human Ethics Committee.

Your application has been approved from the above date and this approval is valid for three years. If your data collection is not completed by this date you should apply to the Human Ethics Committee for an extension to this approval.

Best wishes with the research.

Kind regards,

[redacted]

Judith Loveridge

Convenor, Victoria University of Wellington Human Ethics Committee

Appendix B

Interview Information Sheet

Live Coding for Visual Performance

INFORMATION SHEET FOR PARTICIPANTS

You are invited to take part in this research. Please read this information before deciding whether or not to take part. If you decide to participate, thank you. If you decide not to participate, thank you for considering this request.

Who am I?

My name is Jack Purvis and I am a Masters student in Computer Graphics at Victoria University of Wellington. This research project is work towards my thesis.

What is the aim of the project?

The aim of the research project is to develop a new software environment for live coders. The new environment will be used for creative coding of visuals in live audio-visual performances. The new environment will provide an alternative to existing environments and answer two research questions:

- How can we improve state transition capabilities in live coding?
- How can integrating VJ (Video Jockey) practices improve the capabilities of live coders?

The aim of this interview is to get a better perspective on the current practice of live coders and VJs. The collected data will help to inform the design of the new software environment.

This research has been approved by the Victoria University of Wellington Human Ethics Committee, with approval ID 0000025996. The end date of the research is the 9th of April 2019.

How can you help?

You have been invited to participate because you are familiar with or have worked with some combination of live coding, creative coding or VJ software. It is possible you also experience with live performance. If you agree to take part I will interview you either at Victoria University, a public Cafe or at a location of your choice. I will ask you questions about your creative and performance practice. The interview will take between 30 and 60 minutes and I will audio record the interview with your permission and write it up later. You can choose to not answer any question or stop the interview at any time, without giving a reason. You can withdraw from the study by contacting me at any time within two weeks after the interview. If you withdraw, the information you provided will be destroyed or returned to you.

What will happen to the information you give?

This research is confidential. This means that the researchers named below will be aware of your identity but the research data will be combined and your identity will not be revealed in any reports, presentations, or public documentation. However, you should be aware that persons familiar with your practice may be able to identify you based on the distinctness of the information you provide.

Only myself and my supervisors will access the information recorded from the interview. Any recorded information will be kept securely on an ECS computer or locked filing cabinet and will be destroyed one month after the research is finished.

What will the project produce?

The information from my research will be used in my Masters dissertation may be used in academic publications or presented to conferences. The developed software environment will be new intellectual property with potential commercial value.

If you accept this invitation, what are your rights as a research participant?

You do not have to accept this invitation if you don't want to. If you do decide to participate, you have the right to:

- choose not to answer any question;
- ask for the recorder to be turned off at any time during the interview;
- withdraw from the study from up to two weeks after the interview;
- ask any questions about the study at any time;
- receive a copy of your interview recording;
- read over and comment on a written summary of your interview*;*
- be able to read any reports of this research by emailing the researcher to request a copy.

If you have any questions or problems, who can you contact?

If you have any questions, either now or in the future, please feel free to contact either:

Student:

Name: Jack Purvis

University email address: purvisjack@myvuw.ac.nz

Primary Supervisor:

Name: Dr. Craig Anslow

Role: Lecturer of Software Engineering

School: Engineering and Computer Science

Phone: 04 463 6449

craig.anslow@ecs.vuw.ac.nz

Human Ethics Committee information

If you have any concerns about the ethical conduct of the research you may contact the Victoria University HEC Convenor: Dr Judith Loveridge. Email hec@vuw.ac.nz or telephone +64-4-463 6028.

Appendix C

Interview Schedule

Schedule

1. If you have one and would like to share it, what is your stage name?

2. How do you classify yourself in your creative profession?
   E.g: Live coder, creative coder, VJ.

3. What mediums do you normally work with?
   E.g: Electronic music, digital visualisations, physical art.

4. What are the sort of events you have performed at? Can you describe these events?
   E.g: Concert, gig, art installation. People, genre, environment.

5. What is your computer programming experience? If so, how do you use code in your live performance?

6. What does your typical performance setup look like? What kind of software and hardware do you use? How does each component of the setup relate and why are they important?

7. Can you describe the steps involved when using your setup to perform? What are some of the common tasks and actions you need perform during the event?
   E.g: Write new music code and fade it in, select a new video clip and mix it with the current one.

8. What aspects of your or setup performance are important to you for creating a good performance?
   E.g: Allowing improvisation, audience interaction, experimentation, collaboration with other performers.

9. What are some of the important features or interactions made possible by the software or hardware that are really important for your performance?
   E.g: Code error highlighting, ability to map parameters to MIDI controller.

10. What direction are you intending of taking your performance work in future? Are you planning on experimenting with new styles or using new software or hardware?

11. Is there any kind of limitation with your current setup? Is there some things you wish you could bring to your performance but cannot?

Appendix D

Feedback Survey Information Sheet

Live Coding for Visual Performance

INFORMATION FOR PARTICIPANTS

You are invited to take part in this research. Please read this information before deciding whether or not to take part. If you decide to participate, thank you. If you decide not to participate, thank you for considering this request.

Who am I?

My name is Jack Purvis and I am a Masters student in Computer Graphics at Victoria University of Wellington. This research project is work towards my thesis.

What is the aim of the survey?

The purpose of this survey is to evaluate Visor, a new software environment for live coding of graphics in audiovisual performances. Visor has been developed as part of a Masters research project by myself, Jack Purvis, at Victoria University of Wellington. Visor is intended to be used by live coders, VJs, creative coders, and audiovisual performers. This survey aims to evaluate the effectiveness of Visor and to inform the future development of the software. This research has been approved by the Victoria University of Wellington Human Ethics Committee, with approval ID 0000025996.

How can you help?

You have been invited to participate because you have used Visor in some capacity. If you agree to take part you will complete a survey. The survey will ask you questions about Visor. The survey will take you 10 to 15 minutes to complete.

What will happen to the information you give?

This research is anonymous. This means that nobody, including the researchers will be aware of your identity. By answering it, you are giving consent for us to use your responses in this research. Your answers will remain completely anonymous and unidentifiable. Once you submit the survey, it will be impossible to retract your answer. Please do not include any personal identifiable information in your responses.

What will the project produce?

The information from my research will be used in my Masters dissertation and may be used in academic publications or presented to conferences. The information from my research will also be used to inform future development of Visor.

If you have any questions or problems, who can you contact?
If you have any questions, either now or in the future, please feel free to contact either Jack Purvis at jack.purvis@ecs.vuw.ac.nz, or my supervisor Dr. Craig Anslow at craig.anslow@ecs.vuw.ac.nz.

Human Ethics Committee information

If you have any concerns about the ethical conduct of the research you may contact the Victoria University HEC Convenor: Dr Judith Loveridge. Email hec@vuw.ac.nz or telephone +64-4-463 6028.

Appendix E

Feedback Survey Questionnaire

Visor Feedback Survey Questionnaire

How much time have you spent using Visor? (select one)

- Less than 1 hour
- 1 to 5 hours
- 5 to 10 hours
- More than 10 hours

How many years of programming experience do you have? (select one)

- Less than 1 year
- 1 to 2 years
- 2 to 3 years
- More than 3 years

How much experience do you have with the Ruby programming language? (select one)

- No experience
- A little experience
- A fair amount of experience
- Professional experience

How much experience do you have with the Processing library / integrated development environment? (select one)

- No experience
- A little experience
- A fair amount of experience
- Professional experience

How much live coding experience do you have? (select one)

- No experience
- A little experience
- A fair amount of experience
- Professional experience

How much VJing experience do you have? (select one)

- No experience
- A little experience
- A fair amount of experience
- Professional experience

Did you find the state management interface useful? Why did you find it useful / not useful? (text field)

Did you find the layer manager interface useful? Why did you find it useful / not useful? (text field)

Did you configure an audio input? (yes / no)

    If so, were the FFT display and tap tempo interfaces useful? Why did you find them useful / not useful? (text field)
    If not, why did you not configure an audio input? (text field)

Did you configure a MIDI controller? (yes / no)

    If so, were the interactions with the MIDI useful? Why did you find them useful / not useful? (text field)
    If not, why did you not configure a MIDI controller? (text field)

Visor is difficult to learn (select one)

- Strongly agree
- Agree
- Neutral
- Disagree
- Strongly disagree

In which context would you use Visor? (select all that apply)

- Performance - live coding new material
- Performance - VJing with pre-coded material
- Creative coding
- Other (please specify)

The features(s) I enjoyed most about Visor were? (text field)

The features(s) I enjoyed least about Visor were? (text field)

Do you have anything else you would like to share? (text field)

Bibliography

[1] Art∼Hack website. http://arthack.nz/. Accessed: 01/05/2019.

[2] Burrowing Pufferfish Party performance recording. https://tinyurl.com/visor-pufferfish-2018/. Accessed: 11/05/2019.

[3] Chuck website. http://chuck.cs.princeton.edu/. Accessed: 01/05/2019.

[4] Cyril website. http://cyrilcode.com/. Accessed: 01/05/2019.

[5] DESTROY WITH SCIENCE website. http://destroywithscience.com/. Accessed: 01/05/2019.

[6] Electron website. https://electronjs.org/. Accessed: 01/05/2019.

[7] Eyegum Wednesdays performance recording. https://tinyurl.com/visor-eyegum-2019/. Accessed: 11/05/2019.

[8] FREAKS 001 performance recording. https://tinyurl.com/visor-freaks-2018/. Accessed: 11/05/2019.

[9] Gibber screenshot. https://charlie-roberts.com/work.htm/. Accessed: 01/05/2019.

[10] JRuby website. http://jruby.org/. Accessed: 01/05/2019.

[11] JRubyArt website. http://ruby-processing.github.io/JRubyArt/. Accessed: 01/05/2019.

[12] KodeLife website. https://hexler.net/software/kodelife/. Accessed: 01/05/2019.

[13] Light Table screenshot. http://lighttable.com/2013/08/22/light-table-050/. Accessed: 01/05/2019.

[14] Light Table website. http://lighttable.com/. Accessed: 01/05/2019.

[15] LiveCodeLab screenshot. http://livecodelab.net/. Accessed: 01/05/2019.

[16] Livestock Pixel website. http://livestockpixel.com/. Accessed: 01/05/2019.

[17] MadMapper website. https://madmapper.com/. Accessed: 01/05/2019.

[18] Maker Faire Wellington website. https://wellington.makerfaire.com/. Accessed: 01/05/2019.

[19] Max/MSP/Jitter website. https://cycling74.com/. Accessed: 01/05/2019.

[20] Minim website. http://code.compartmental.net/tools/minim/. Accessed: 01/05/2019.

[21] modul8 website. http://www.garagecube.com/modul8/. Accessed: 01/05/2019.

[22] Monokai author website. https://www.monokai.nl/. Accessed: 06/05/2019.

[23] OpenFrameworks website. http://openframeworks.cc/. Accessed: 01/05/2019.

[24] OpenProcessing website. https://www.openprocessing.org/. Accessed: 07/05/2019.

[25] p5.js website. https://p5js.org/. Accessed: 01/05/2019.

[26] Praxis LIVE screenshot. https://praxisintermedia.wordpress.com/. Accessed: 01/05/2019.

[27] Processing website. https://processing.org/. Accessed: 01/05/2019.

[28] Pure Data website. https://puredata.info/. Accessed: 01/05/2019.

[29] React website. https://reactjs.org/. Accessed: 01/05/2019.

[30] Resolume website. https://resolume.com/. Accessed: 01/05/2019.

[31] Sonic Pi meets Visor performance recording. https://tinyurl.com/visor-meets-sonicpi/. Accessed: 11/05/2019.

[32] Spout website. http://spout.zeal.co/. Accessed: 01/05/2019.

[33] Squeak - Smalltalk Environment website. http://squeak.org/. Accessed: 01/05/2019.

[34] Syphon Implementation for Processing website. https://github.com/Syphon/Processing/. Accessed: 01/05/2019.

[35] Syphon Recorder website. http://syphon.v002.info/recorder/. Accessed: 12/05/2019.

[36] Syphon website. http://syphon.v002.info/. Accessed: 01/05/2019.

[37] Taniwha's Den 2019 Mainstage performance recording. https://tinyurl.com/visor-taniwhasden-2019/. Accessed: 11/05/2019.

[38] The MidiBus website. http://www.smallbutdigital.com/projects/themidibus/. Accessed: 01/05/2019.

[39] three.js website. https://threejs.org/. Accessed: 01/05/2019.

[40] TOPLAP 15th Birthday Livestream performance recording. https://tinyurl.com/visor-toplap15/. Accessed: 11/05/2019.

[41] Toplap website. https://toplap.org/about/. Accessed: 01/05/2019.

[42] Touch Designer website. http://www.derivative.ca/. Accessed: 01/05/2019.

[43] TYPE website. https://typeensemble.wordpress.com/. Accessed: 01/05/2019.

[44] Visor website. http://www.visor.live/. Accessed: 01/05/2019.

[45] VVVV screenshot. https://vvvv.org/screenshots/. Accessed: 01/05/2019.

[46] VVVV website. https://vvvv.org/. Accessed: 01/05/2019.

[47] AARON, S. Sonic Pi performance in education, technology and art. *International Journal of Performance Arts and Digital Media 12*, 2 (2016), 171–178.

[48] AARON, S., BLACKWELL, A. F., HOADLEY, R., AND REGAN, T. A principled approach to developing new languages for live coding. In *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)* (2011), pp. 381–386.

[49] ABRAS, C., MALONEY-KRICHMAR, D., AND PREECE, J. User-centered design. *Bainbridge, W. Encyclopedia of Human-Computer Interaction. Thousand Oaks: Sage Publications 37*, 4 (2004), 445–456.

[50] BERGSTRÖM, I., AND BLACKWELL, A. F. The practices of programming. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2016), pp. 190–198.

[51] BERGSTRÖM, I., AND LOTTO, B. Mother: Making the performance of real-time computer graphics accessible to non-programmers. In *re) Actor3: The Third International Conference on Digital Live Art Proceedings* (2008), pp. 11–12.

[52] BERGSTRÖM, I., AND LOTTO, R. B. Code Bending: A New Creative Coding Practice. *Leonardo 48*, 1 (2015), 25–31.

[53] BLACKWELL, A., MCLEAN, A., NOBLE, J., AND ROHRHUBER, J. Collaboration and learning through live coding (Dagstuhl Seminar 13382). *Dagstuhl Reports 3* (2013), 130–168.

[54] BLACKWELL, A. F. Palimpsest. *J. Vis. Lang. Comput. 25*, 5 (2014), 545–571.

[55] BRAUN, V., AND CLARKE, V. Using thematic analysis in psychology. *Qualitative research in psychology 3*, 2 (2006), 77–101.

[56] COLLINS, N., AND MCLEAN, A. Algorave: Live performance of algorithmic electronic dance music. In *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)* (2014), pp. 355–358.

[57] COLLINS, N., MCLEAN, A., ROHRHUBER, J., AND WARD, A. Live coding in laptop performance. *Organised sound 8*, 3 (2003), 321–330.

[58] CORREIA, N. N. Prototyping Audiovisual Performance Tools : A Hackathon Approach. In *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)* (2015), pp. 12–14.

[59] CORREIA, N. N., AND TANAKA, A. User-Centered Design of a Tool for Interactive Computer-Generated Audiovisuals. In *International Conference on Live Interfaces (ICLI)* (2014).

[60] DELLA CASA, D., AND JOHN, G. LiveCodeLab 2.0 and Its Language LiveCodeLang. In *Proceedings of the ACM SIGPLAN International Workshop on Functional Art, Music, Modeling & Design (FARM)* (2014), ACM, pp. 1–8.

[61] FAULKNER, M., AND D-FUSE. *VJ: Audio-Visual Art and VJ Culture: Includes DVD*. Laurence King Publishing, 2006.

[62] FISCHER, A. Introducing Circa: A dataflow-based language for live coding. In *International Workshop on Live Programming (LIVE)* (2013), pp. 5–8.

[63] GHOSTDAD. My command center. `https://www.flickr.com/photos/ghostdad/7577003990/`, 2012. License: CC BY-SA 2.0. Accessed: 01/05/2019.

[64] HOOK, J., GREEN, D., MCCARTHY, J., TAYLOR, S., WRIGHT, P., AND OLIVIER, P. A VJ Centered Exploration of Expressive Interaction. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)* (2011), ACM, pp. 1265–1274.

[65] JUNG, H., LEE, J., CHOI, H.-J., AND KIM, H. Real-Time DJING+ VJING with Interactive Elements. *Contemporary Engineering Sciences* (2014), 1321–1327.

[66] KUBELKA, J., ROBBES, R., AND BERGEL, A. The road to live programming: insights from the practice. In *Proceedings of the International Conference on Software Engineering (ICSE)* (2018), IEEE, pp. 1090–1101.

[67] LAWSON, S. Performative Code: Strategies for Live Coding Graphics. In *Proceedings of the International Conference on Live Coding (ICLC)* (2015), pp. 35–40.

[68] MAGUIRE, M., AND DELAHUNT, B. Doing a thematic analysis: A practical, step-by-step guide for learning and teaching scholars.

AISHE-J: The All Ireland Journal of Teaching and Learning in Higher Education 9, 3 (2017).

[69] MCLEAN, A. Making programming languages to dance to: live coding with tidal. In *Proceedings of the ACM SIGPLAN international workshop on Functional art, music, modeling & design (FARM)* (2014), ACM, pp. 63–70.

[70] MCLEAN, A., AND WIGGINS, G. A. Live Coding Towards Computational Creativity. In *Proceedings of the International Conference on Computational Creativity* (2010), pp. 175–179.

[71] OLOWE, I., MORO, G., AND BARTHET, M. residUUm: user mapping and performance strategies for multilayered live audiovisual generation. In *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)* (2016), pp. 271–276.

[72] REAS, C., AND FRY, B. Processing: programming for the media arts. *AI & SOCIETY 20*, 4 (2006), 526–538.

[73] REIN, P., RAMSON, S., LINCKE, J., HIRSCHFELD, R., AND PAPE, T. Exploratory and live, programming and coding: A literature study comparing perspectives on liveness. *The Art, Science, and Engineering of Programming 3* (2018).

[74] ROBERTS, A. Chemical Algorave. `https://www.flickr.com/photos/hellocatfood/33851477483/`, 2017. License: CC BY-SA 2.0. Accessed: 01/05/2019.

[75] ROBERTS, C., WRIGHT, M., KUCHERA-MORIN, J., AND HÖLLERER, T. Gibber: Abstractions for creative multimedia programming. In *Proceedings of the International Conference on Multimedia* (2014), ACM, pp. 67–76.

[76] ROHRHUBER, J., DE CAMPO, A., AND WIESER, R. Algorithms today notes on language design for just in time programming. In *Proceedings*

of the International Computer Music Conference (2005), p. 291.

[77] SALAZAR, S. Searching for Gesture and Embodiment in Live Coding. In *Proceedings of the International Conference on Live Coding (ICLC)* (2017).

[78] SMITH, N. C. Praxis LIVE - hybrid visual IDE for (live) creative coding. In *Proceedings of the International Conference on Live Coding (ICLC)* (2016).

[79] TANIMOTO, S. L. A Perspective on the Evolution of Live Programming. In *Proceedings of the International Workshop on Live Programming* (2013), pp. 31–34.

[80] TOKA, M., INCE, C., AND BAYTAS, M. A. Siren: Interface for pattern languages. In *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)* (Blacksburg, Virginia, USA, 2018), T. M. Luke Dahl, Douglas Bowman, Ed., Virginia Tech, pp. 53–58.

[81] VICTOR, B. Learnable programming: Designing a programming system for understanding programs. `http://worrydream.com/LearnableProgramming/`, 2012. Accessed: 01/05/2019.