Visor in Practice: Live Performance and Evaluation

Jack Purvis
Victoria University of Wellington, New Zealand
jack.purvis@ecs.vuw.ac.nz

Craig Anslow
Victoria University of Wellington, New Zealand
craig@ecs.vuw.ac.nz

James Noble
Victoria University of Wellington, New Zealand
kjx@ecs.vuw.ac.nz

 ICLC 2020, February 5-7, 2020, University of Limerick ,Limerick, Ireland

Abstract

Visor is a new environment for live visual performance that was developed to demonstrate code jockey practice (CJing), a new hybrid performance practice that combines live coding and VJing to harness the strengths of both practices. CJing draws on live coding for the ability to improvise content at a low level by coding in textual interfaces. VJing is drawn on for its ability to manipulate content at a high level by interacting with GUIs and hardware controllers. Combining these aspects of both practices enables flexible performances where content can be controlled at both low and high levels. We build on previous work by reflecting on the use of Visor in live performances and evaluating feedback gathered from creative coders, live coders, and VJs who experimented with the environment. We conclude by discussing Visor's effectiveness and whether CJing effectively combines live coding and VJing along with areas for future work.

Introduction

The creation of visuals to accompany music is an essential part of any audiovisual experience. Live coding and VJing (video jockey practice) are live performance practices that offer the ability to improvise and manipulate visuals that synchronize with music in real-time. Live coding makes use of live programming techniques, enabling code to be executed at runtime with immediate feedback (Tanimoto 2013). Live coding is used to explore how algorithms can generate music or visuals and is typically performed live at events called Algoraves (Collins et al. 2003). Video jockeys (VJs) are live visual artists that mix content in real-time, often complimenting music played by disc jockeys (DJs) to create audiovisual marriages that engage the senses (Faulkner and D-fuse 2006). VJs typically perform by layering multiple video clips together, applying video effects, and interacting with effect parameters using hardware devices such as MIDI controllers.

Live coding focuses on writing code to improvise or manipulate content. This focus provides fine-grained, low level control of visuals,

but does not provide high level control, impairing usability as all interactions must occur through a textual interface. VJs instead focus on interacting with comprehensive graphical user interfaces (GUIs) and hardware controllers to improvise or manipulate content. This focus provides overarching, high level control of visuals, but does not provide low level control, preventing improvisation of content from scratch or the ability to make fine-grained adjustments to existing content. The CJing practice was introduced to overcome the limitations of live coding and VJing by combining the practices together (Purvis et al. 2019). 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. CJing has been demonstrated by Visor, a new live coding environment that embodies the practice (Purvis et al. 2019). Visor has been purpose-built following a practice-based, user-centered approach to offer features for both live coding and VJing to enable live visual performances. To determine the effectiveness of Visor and whether CJing can effectively combine live coding and VJing, in this paper we reflect on Visor's use in live performances and evaluate feedback gathered from creative coders, live coders, and VJs who experimented with the environment.

Background

CJING

Code jockey practice (CJing) is a new hybrid performance practice that was first proposed by Purvis et al. (2019). CJing harnesses the strengths of live coding and VJing to enable flexible performances while simultaneously removing limitations identified in each practice. 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 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, an aesthetic of 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.

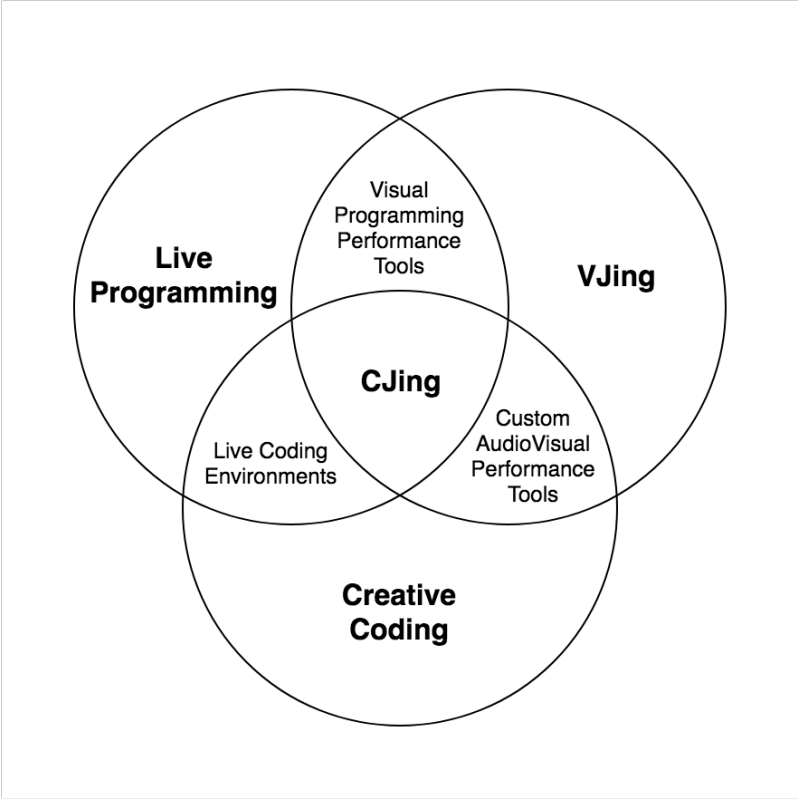

Figure 1: "Software that supports CJing lies at the intersection of creative coding, live programming, and VJing related software tools." (Purvis et al. 2019)

CJing practice is formulated from the broader subject areas of

creative coding, live programming, and VJing (Purvis et al. 2019). As shown in Figure 1, software that supports CJing is placed at the intersection of these subject areas. Purvis et al. (2019) proposes that CJing practice is based on three key ideas:

1. Code as a universal language: Creative coding should be used to produce content from scratch, while live coding enables the coded content to be manipulated on the fly at runtime. In CJing, code is the predominant content that is used to instruct the visual output, similar to how musical tracks are treated in DJing, and how video clips are treated in VJing.

2. Complete content control: CJing should allow for content to be manipulated at both low and high levels, enabling flexible control of the visual output during performances. The low level aspect is provided by live coding, allowing content to be improvised or edited using code. The high level aspect is provided by VJing, allowing content to be organised into layers and manipulated through effects, parameters, or hardware controls.

3. User interfaces as an abstraction: Code should be abstracted upon by user interfaces, providing high level functionality that would not be easy to achieve by simply writing code. Therefore, CJing software maintains a relationship between user interfaces and code, for example, providing contextual interfaces that 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.

VISOR

Visor[1] is a new hybrid environment that was developed with the primary goal being to embody CJing, exploring how live coding and VJing can be combined into a single environment to harness the strengths of both practices (Purvis et al. 2019). Visor achieves this goal by offering a number of core features to facilitate live performance: Live coding: Visor offers the ability to live code visuals with the Processing API (Reas and Fry 2006) in the Ruby language.

- State management: Visor supports a state management interface that automatically visualises and provides GUIs to update live coded variables. For example, numeric variables are presented as sliders with configurable ranges and boolean variables are presented as checkboxes, enabling parameters to be manipulated at a high level without the need to write any code.

- Layers: Visor offers the ability to organise code into multiple layers that are composited together into the final visual output using a variety of blend modes. Each layer acts as an independent Processing sketch with its own state and draw loop.

- FFT: Visor supports the fast Fourier transform (FFT) algorithm to generate a frequency spectrum from an audio input in real-time. The spectrum is visualised in the interface and made accessible in the code, enabling audio reactive visuals. Tap tempo: Visor offers the ability to set a tempo that can be referenced in the code to animate visuals to the beat of live music. The tempo can be set by repeatedly clicking a button in the interface or by using a keyboard shortcut.

- MIDI: Visor supports a framework for configuring inputs from external devices such as MIDI controllers using the MIDI protocol. The framework enables sliders, knobs, and buttons to become directly accessible in the code or mapped to state parameters. Visor supports a number of other notable features including support for multiple code tabs, support for multiple display outputs, a console, an in-app tutorial, and in-app documentation. A number of existing features were also refactored since Purvis et al. (2019), most notably, the REPL editor and

[1] `https://www.visor.live`

the draw loop editor were merged into a single editor. The implication of this is that the code that should be executed every frame should now be scoped within a method called draw, akin to how Processing traditionally operates. Figure 2 shows Visor in action by presenting the Visor interface and corresponding visual output.

Design Methodology

Visor was developed using a practice-based approach where the environment was tested in a performance context throughout development to ensure it met the needs of a performer (in this case, the first author). Unlike conventional software engineering processes, this approach considers the act of developing software as a form of craft research (Blackwell and Aaron 2015) and places emphasis on the need for 'research through design' (Gaver 2012). Using Visor in a performance context during development meant that the effectiveness of the environment could be evaluated iteratively: Existing features could be validated, features that needed improvement could be refined, and ideas for new features could be generated.

Practice-based approaches have been demonstrated by the Sonic Pi and Palimpsest environments (Aaron 2016; Blackwell 2014), both of which were developed with a consideration for craft practice and were discussed in the context of their use in live performance. Here we take a similar approach by reflecting on the use of Visor in live performances, as presented in Section 3. Visor is also evaluated following a more conventional user-centered approach (Abras et al. 2004) by analysing feedback collected from respondents of an online survey, similar to the approach taken to evaluate the Gibber live coding environment (Roberts et al. 2014). This evaluation is presented in Section 4.

Related Work

Figure 3: Mainstage of the Taniwha's Den 2019 music festival. The rendered visuals are projected across multiple screens around the DJ booth using multiple projectors. The VJ booth is situated behind where this photograph was taken.

There are a number of software applications that support features that coincide with Visor or the broader CJing practice. Sonic Pi (Aaron 2016) is a Ruby based live coding environment for creating music that focuses on simplicity to excel in computing education. Mother (Bergström and Lotto 2008) is an extension to Processing that enables VJing performances where multiple sketches can be layered together to instruct the final visual output. Auraglyph (Salazar 2017) is a live coding environment that supports visual programming of music using a touch screen interface, offering gestural manipulation akin to using a MIDI controller. Praxis LIVE (Smith 2016) is an audiovisual live coding environment and IDE that supports creative coding with Processing and offers GUIs that work hand in hand with coded content. Siren (Toka et al. 2018) is a hybrid system for the

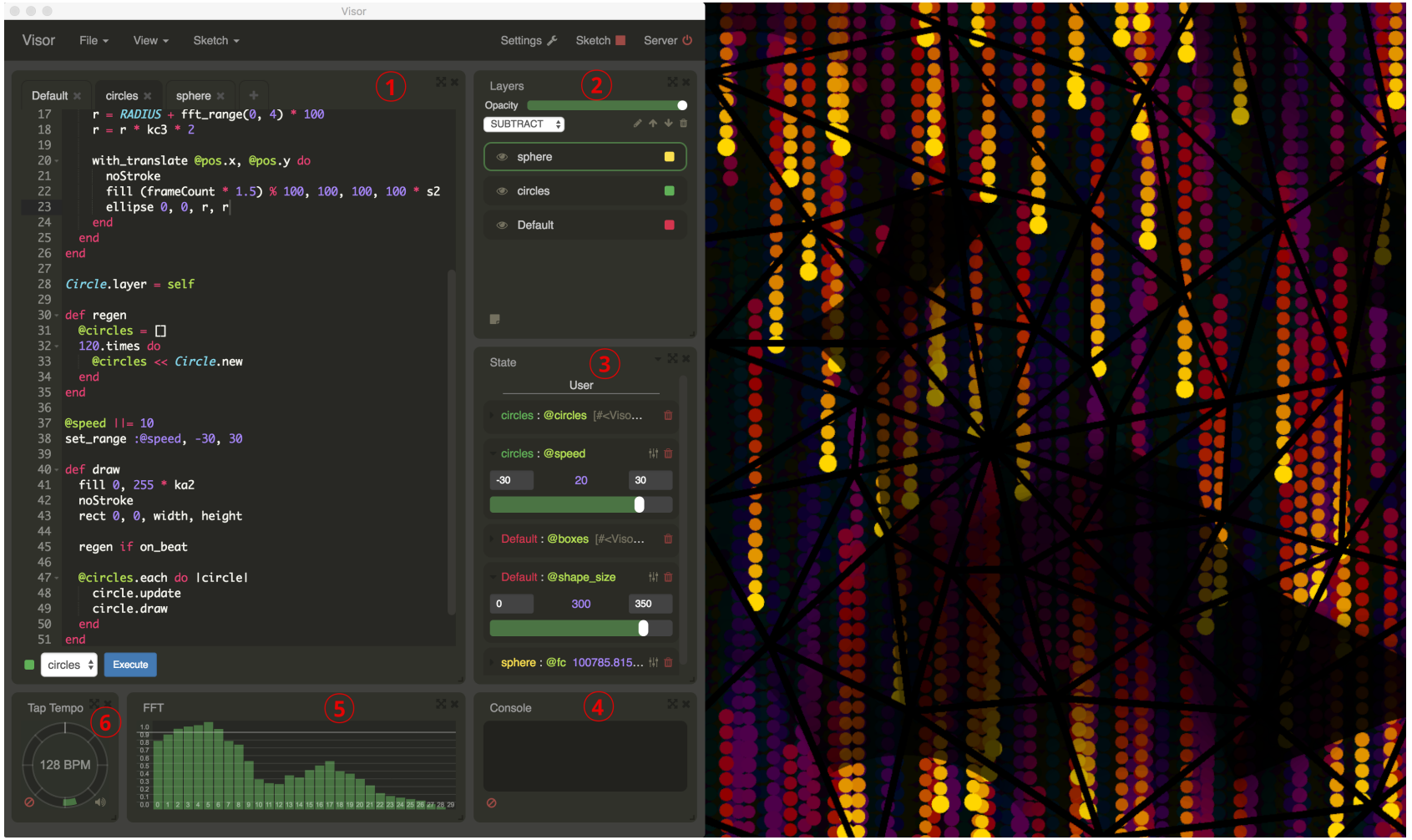

Figure 2: Visor in action. The interface (left) is made up of multiple GUI elements including a live code editor (1), layer interface (2), state management interface (3), console (4), FFT display (5), and tap tempo interface (6). The visual output corresponding to this state of the interface is also shown (right).

composition of algorithmic music and live coding performances that offers GUIs to interact with code. Resolume[2] is a widely used tool for VJing that supports video mixing, a variety of effects, and allows parameters to be animated to music in real-time using an FFT or tap tempo.

Live Performances

Visor was used by the first author in 20 live performances to reflect on the effectiveness of the environment as part of a practice-based approach. The intent 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. The performances were conducted alongside a variety of collaborators at a variety of different events including gigs, algoraves, livestreams, exhibitions, research group meetings, private parties, and music festivals. By using Visor in real performances, we explored and demonstrated what it meant to perform with an environment that embodies CJing. We now describe the typical performance setup, approach to using Visor in live performances, and two notable issues concerning CJing that were observed.

Performance Setup

The typical performance setup consisted of a number of hardware and software components. 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. Where available, a line-in from the sound desk was used as input to Visor's FFT, otherwise, the laptop microphone was used. Projectors were usually 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 onto complex surfaces. An example of this is shown in two performances that were conducted at the Taniwha's Den 2019 music festival. In the first performance, the visuals were projected onto multiple screens using multiple projectors, as shown in Figure 3. In the second performance, the visuals were projected onto a large limestone cliff face, as shown in Figure 4. This offered a novel live coding and VJing experience.

Figure 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.

Performance Approach

The conducted performances 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

[2] https://resolume.com/

[3] https://tinyurl.com/visor-toplap15/

an empty screen. This approach was used in 10 of the performances including the TOPLAP 15th Birthday[3] performance shown in Figure 5. 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 times throughout the performance. Live coding also established mappings between parameters and MIDI variables, 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 10 of the performances including the Taniwha's Den[4] performance shown in Figure 3. 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 focused on interaction with the MIDI controller as the content and MIDI mappings had been defined in advance. Live coding also occurred during these performances to improvise content or to manipulate the existing content, for example, to toggle predefined states and attach or detach parameters from the FFT and tap tempo.

In addition to these two approaches, during two performances, a collaborator was invited to perform alongside the author for a small section of each performance. This collaborator focused solely on interacting with the MIDI controller while the author focused on live coding and interacting with the GUI. Overall, 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.

[4] https://tinyurl.com/visor-taniwhasden-2019/

Reflection

Using Visor in live performances helped to identify general usability issues and aspects of the core features that could be improved. More notably, two issues were identified concerning the broader CJing practice. The first issue relates to the idea of user interfaces as an abstraction and highlights the need for careful consideration when designing the relationship between code and GUIs in CJing environments. This issue was observed when interaction with layers using both the GUI and the code would result in conflicting behaviour. For example, when the code to set the blend mode was specified (i.e. set_blend_mode), the blend mode set through the GUI could be unintentionally overridden by the executing code. To avoid this behaviour, the live performances almost exclusively used the GUI to set blend modes. This issue highlights the importance of the relationship between code and user interfaces in CJing environments: careful consideration should be taken when designing a flexible CJing environment where both live coding and VJing can be used to interact with specific features.

The second issue relates to the idea of complete control and was the need to switch contexts between live coding and using the MIDI controller. This issue was also observed in the user feedback presented in Section 4. As both contexts required almost full attention, it seemed impossible to live code and perform with the MIDI controller at the same time. This was mostly observed in the early stages of the "from scratch" performances. In these performances, parameters of existing content could not be tuned using the MIDI controller while the focus was placed on live coding new content. The opposite holds true for later in the performance when the focus was placed on the MIDI controller and live coding was mostly used to make minor adjustments to

the existing 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 when working across multiple modalities. This highlights the importance of automated features such as the FFT and tap tempo which continuously produce dynamic visual effects without requiring the attention of the performer.

An approach to mitigating the friction caused by context switching in CJing was observed when a collaborator performed alongside the first author during two performances. Utilising two performers meant that one performer could focus on live coding while the other focused on the MIDI controller, enabling content to be improvised from scratch while the parameters of existing content were performed at the same time.

Evaluation

An online feedback survey was constructed to evaluate the effectiveness of Visor as part of the user-centered design process (Abras et al. 2004). The survey solicited feedback from people with creative coding, live coding, and VJing experience who had used Visor. The Visor users who participated in the survey were asked to complete a questionnaire that asked about their background, their usage of Visor, their outlook on Visor's core features, how difficult they found Visor to learn, the context in which they might use Visor, and what they liked or disliked about Visor in general. These questions were formatted as either Likert scales, multiple choice, or free form text fields.

ID	Ruby experience	Processing experience	Live coding experience	VJing experience	Visor Time	Visor Context
P1	Little	Pro	Little	Little	1-5 hours	Performance (Live coding)
P2	Little	Little	None	None	1-5 hours	Performance (VJing)
P3	Little	Pro	None	Little	1-5 hours	Creative coding
P4	Little	Pro	Pro	Pro	5-10 hours	Teaching
P5	Fair	Fair	Little	Little	10+ hours	Performance (VJing)
P6	None	Pro	None	None	1-5 hours	Creative coding
P7	None	Fair	None	None	10+ hours	Performance (VJing)
P8	Pro	Little	None	None	1-5 hours	Creative coding
P9	Pro	Little	Little	None	10+ hours	Creat. cod. & Perf.
P10	None	Pro	Little	None	10+ hours	Performance (Live coding)
P11	None	Fair	None	None	1-5 hours	Teaching

Table 1.1: Feedback survey participants background, estimated time spent using Visor, and the context in which they might use Visor.

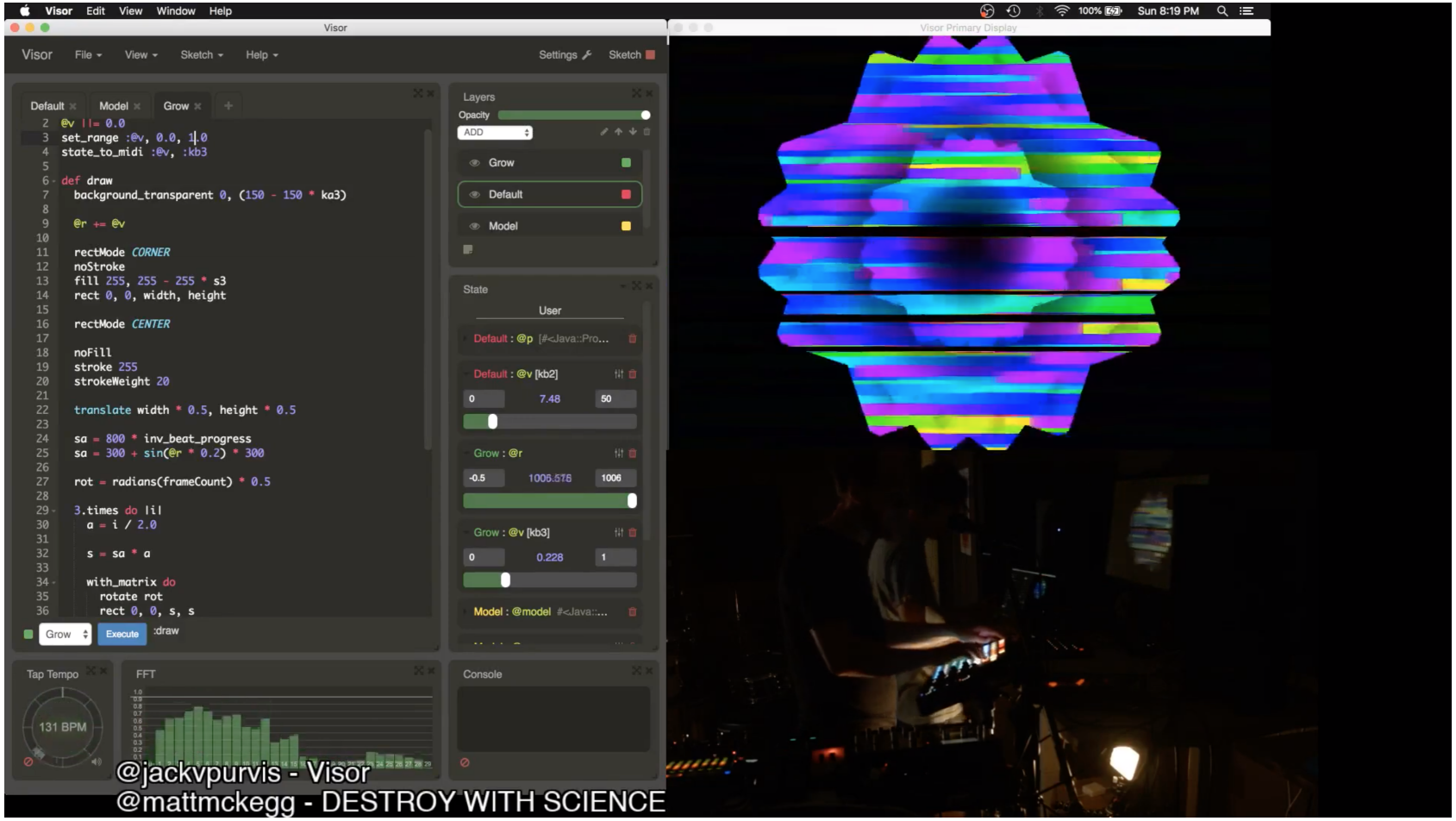

Figure 5: Screenshot from the livestream of the TOPLAP 15th Birthday performance. The Visor GUI (left) is displayed alongside the rendered visuals (top-right) and a camera recording of the physical performance by the first author and a musical collaborator, DESTROY WITH SCIENCE (bottom-right).

Participants

In total, 11 participants completed the feedback survey since it was launched in January of 2019. Participants were recruited through recruitment messages placed on the Visor website 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. The environment was also advocated through the author's existing networks of live coders, creative coders, and VJs. Visor itself has been downloaded more than 900 times since January 2019.

Participants were asked to provide background information with respect to their experience with general programming, 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 1.1. All of the participants reported having more than three years of programming experience except for P7, who had 1 to 2 years of experience.

Results

The results are grouped based on Visor's usage, learning difficulty, core features, and ease of use. The results about Visor's usage and learning difficulty were reported based on a combination of multiple choice questions and comments received in free form questions. The remainder of the results were reported based on direct quotes from free form questions. These questions asked participants how effective they found each of Visor's core features and why, as well as what features of Visor they enjoyed most or least. Usage: The participants were asked to report how much time they had spent using Visor and the context in which they might use Visor. The responses to these questions are also presented in Table 1.1. All of the participants reported using Visor for at least one hour while five had used it for five hours or more. P1 and P10 reported that they would use Visor for live coding new material in performance; P2, P5, and P7 reported that they would use Visor to VJ with pre-prepared material; P3, P6, and P8 reported that they would use Visor for creative coding; P4 and P11 reported that they would use Visor in teaching; and P9 reported that they would use Visor for creative coding and in both performance contexts. This variety of responses that were received for this question indicates the potential versatility for Visor to be used by different audiences including creative coders, live coders, and VJs. Two of the participants also made additional comments on their current or intended usage of Visor:

> "I see a lot of potential is this program, I'm trying to learn everything as soon as possible, and have people interested already [sic] in applying it in real clubs. I haven't had this much fun with a program in a while." (P7)

> "I've used visor for two creative coding projects. In one of them, I used visor + a genetic algorithm gem that I've published to "evolve" visualizations ... The project got a great response and I don't think I would have been able to pull it off so smoothly without Visor." (P9)

Learning difficulty: The participants were asked to report how difficult they found Visor to learn. Seven of the participants disagreed or strongly disagreed that it was difficult (P1, P2, P4, P5, P7, P8, P11); three of the participants were neutral (P3, P9, P10); and one of the participants agreed that it was difficult (P5). Some of the participants commented on the effectiveness of the documentation. Three stated that it was useful for helping them get started (P4, P6, P9). It was suggested by another participant that the learning difficulty depended on the user's creative coding experience:

> "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)

Live coding: Most of the participants reported enjoying the live coding experience in Visor. Reasons for this included its similarity to Processing (P1, P2), utilisation of Ruby (P4, P5, P9), and the fast iteration time that was provided (P1, P6). Some of the participants also reported issues with the live coding experience in Visor. One participant struggled to improvise quickly, but put it down to their lack of experience with the IDE (P4), another stated that it was tedious to have to execute different code tabs individually (P5), and one other mentioned that they would be more productive if they could use their own editor (P8).

State management: The state management interface was reported to be effective for a number of reasons. These reasons included the ability to set ranges on values (P2), slider interactions (P5, P11), visual confirmation for debugging (P3, P5, P6), and being able to change variable values without inspecting the code (P2, P7, P9). One participant also discussed specifically how the feature 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." (P6)

Two of the participants also brought up a usability issue with the state management interface, reporting that it became cluttered as their programs got larger and there were no options for organisation (P1, P2). One participant with extensive live coding experience did not use the feature due to how it required them to take their hands off the keyboard and reach for the mouse (P4), something they were not accustomed to doing when live coding. This insight emphasises the issue of context switching that was also identified in the live performances described in Section 3. The participant went on to discuss how they would have used the state management interface, MIDI controller, and tap tempo more if they were conducting pre-composed performances.

Layers: Visor's ability to organise code into layers proved to be one of the most popular features amongst all of the participants. Reasons for the feature's popularity included how they enabled switching between scenes (P2, P7), combining sketches (P1, P2), organising content (P4), provided a way to develop or test a piece of code in isolation (P4, P9), provided more visual variety from less code (P8), and were fun to experiment with (P3, P6).

Supporting comments from two of the participants were:

> "Very useful. Especially if you're used to Photoshop, the metaphor for composing layers like that makes a lot of intuitive sense." (P1)

> "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." (P6)

Usability issues with layers were also identified by some of the participants. For example, P4 raised a concern about the lack of a keyboard shortcut to easily switch between code tabs. FFT: A number of participants claimed that they found the FFT effective in Visor. Reasons for this included its ease of use (P1, P3, P5, P6), visualisation of the frequency spectrum (P2, P8), enabling audio reactive visuals (P7, P9), or that it was something they were accustomed to (P4). Two participants requested a need for more control over the FFT including the smoothing (P2), volume level (P5), and adding support for multiple channels (P5). One participant couldn't configure an audio input (P10).

Tap tempo: The tap tempo was reported by participants to be effective for a number of reasons including its visual design (P2), ease of use (P3), ability to sync visuals with a tempo (P3, P5, P6, P9), and it's availability in situations where neither MIDI or an audio input are available (P8). One participant's experience with the feature was:

> "Tap tempo and the 'beat' features was [sic] 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." (P6)

One participant was unsure about the tap tempo and suggested adding options to manually tune the BPM and offset without the need to tap (P1). One participant also reported that they did not use the feature because they didn't initially recognise its use in the visual arts (P4).

MIDI: Visor's support for MIDI was a feature that could not be evaluated effectively as part of the feedback survey. Only one participant managed to use a controller successfully (P4). Another participant tried to use a MIDI controller but could not use it effectively due to an issue with Visor or the controller (P2). The remaining 9 participants did not use the MIDI feature. This is likely due to a lack of access to a controller, highlighting a disadvantage of this type of remote study. A better approach to testing Visor's MIDI support would be to conduct an in-person user study where a controller is provided for the participants to use.

Ease of use: One prominent theme in the results was the ease of use of some of Visor's features. For example, the audio input for the FFT was reported to be easily configured through the GUI, and the built-in methods to access the data were straightforward to use. In general, this aspect can be summed up from the following comments:

> "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." (P6)

> "I liked that [sic] many options for getting dynamic input (FFT, tap tempo, setting up buttons and sliders) and that it was straightforward to access them within the sketch. I found these features to be better to explore/control the sketch's style than typing up variables." (P6)

The last comment also touches on the participant's enjoyment of the high level functionality that Visor's features provided. This reiterates one of the motivations for CJing in that the high level control provided by features of VJing software can improve the usability of live coding.

Conclusions

Visor is a new environment for live visual performance that was developed to demonstrate CJing, a new hybrid performance practice that combines aspects of live coding and VJing, drawing on the strengths of both practices while simultaneously removing limitations identified in each practice. To determine the effectiveness of Visor and whether CJing can effectively combine live coding and VJing, we have reflected on Visor's use in live performances, as well as conducted an evaluation of feedback gathered from creative coders, live coders, and VJs who experimented with the environment.

The use of Visor in performances has demonstrated Visor's ability to effectively produce visuals in a live context, at least in combination with the first author's own performance skills. Two approaches to performance were described: live coding content from scratch, and performing with prepared content. These approaches demonstrate how Visor can be used effectively for conducting aspects of live coding, VJing, and both together in the same performance, demonstrating CJing. The feedback gathered from Visor users suggests 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. A number of usability issues and suggested improvements were also identified. Overall, the feedback was highly supportive of Visor and participants generally enjoyed using the environment. The evaluation of Visor has demonstrated Visor's effective support for aspects of both live coding and VJing, improving the usability of live coding through high level user interfaces and providing fine-grained control of content while VJing. This showcases Visor's demonstration of CJing, and in turn, how CJing can be used to effectively combine live coding and VJing.

Two prominent issues with CJing were also identified that need to

be considered in future work. The first was the need for careful design of the relationship between code and user interfaces when designing CJing environments. 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.

A number of opportunities for future work have also been identified. These include further development of the Visor software to improve usability and to offer more features to enhance the environment's performance capabilities. Ideally, we would then conduct a more comprehensive evaluation of the environment through a controlled user study. We also hope to explore how collaboration with live coders, DJs, VJs, and CJs can play a role in CJing practice. In addition, we hope to explore how CJing can be applied in other live coding environments and in particular, within the context of music.

Acknowledgements

To Victoria University of Wellington for the Victoria Masters by Thesis Scholarship. To the Faculty of Science for the Faculty Strategic Research Grant. To the participants who completed the feedback survey. To those who collaborated with or provided the first author with the opportunity to perform on various occasions.

References

Aaron, S. Sonic Pi performance in education, technology and art. (2016). In: International Journal of Performance Arts and Digital Media 12, 2, 171–178.

Abras, C., Maloney-krichmar, D., and Preece, J. User-centered design. (2004). In: Bainbridge, W. Encyclopedia of Human-Computer Interaction. Thousand Oaks: Sage Publications 37, 4, 445–456.

Bergström, I., and Lotto, B. (2008). 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. pp. 11–12.

Blackwell, A. F. (2014). Palimpsest. In: J. Vis. Lang. Comput. 25, 5, 545– 571.

Blackwell, A. F, and Aaron, S. (2015). Craft practices of Live Coding Language Design. In: Proceedings of the First International Conference on Live Coding.

Collins, N., Mclean, A., Rohrhuber, J., and Ward, A. (2003). Live coding in laptop performance. In: Organised sound 8, 3, 321–330.

Faulkner, M., and D-fuse. (2006). VJ: Audio-Visual Art and VJ Culture: Includes DVD. Laurence King Publishing.

Gaver, W. (2012). What should we expect from research through design? In: Proceedings of the SIGCHI conference on human factors in computing systems, ACM. pp. 937-946.

Purvis, J., Anslow, C., and Noble, J. (2019). CJing Practice: Combining Live Coding and VJing. In: Proceedings of the International Conference on Live Coding (ICLC).

Reas, C., and Fry, B. (2006). Processing: programming for the media arts. In: AI & SOCIETY 20, 4, 526–538.

Roberts, C., Wright, M., Kuchera-morin, J., and Höllerer, T. (2014). Gibber: Abstractions for creative multimedia programming. In: Proceedings of the International Conference on Multimedia, ACM. pp. 67–76.

Salazar, S. (2017). Searching for Gesture and Embodiment in Live Coding. In: Proceedings of the International Conference on Live Coding (ICLC).

Smith, N. C. (2016). Praxis LIVE - hybrid visual IDE for (live) creative coding. In: Proceedings of the International Conference on Live Coding (ICLC).

Tanimoto, S. L. (2013). A Perspective on the Evolution of Live Programming. In: Proceedings of the International Workshop on Live Programming. pp. 31–34.

Toka, M., Ince, C., and Baytas, M. A. (2018). Siren: Interface for pattern languages. In: Proceedings of the International Conference on New Interfaces for Musical Expression (NIME). pp. 381–386.