CJing Practice: Combining Live Coding and VJing

Jack Purvis, Craig Anslow, James Noble
School of Engineering and Computer Science
Victoria University of Wellington, New Zealand
{jack.purvis, craig, kjx}@ecs.vuw.ac.nz

ABSTRACT

Live coding has typically focused on textual interfaces to interact with code, but this reliance on typing text impairs the usability of live coding as a performance practice. VJing practice is limited by a lack of fine-grained content control and a number of audiovisual tools are limited by their ability to be reused or shared in new performances. We propose the *code jockey* practice (*CJing*), a new hybrid practice where a performer known as a *code jockey* (*CJ*) improvises, manipulates and composites code to generate procedural graphics. CJing combines elements from live coding and VJing by offering user interfaces that abstract upon the code and provide high level interactions. To illustrate CJing we introduce *Visor*, a live coding environment for producing graphics in audiovisual performances. We present a typical Visor performance scenario and discuss Visor's underlying implementation details.

1 Introduction

Live coding is the application of live programming techniques to the performing arts (Tanimoto 2013). Live coding offers the ability to improvise completely new content such as procedurally generated music or graphics by creative coding over the course of a performance (Collins et al. 2003). Live coding environments tend to use simple user interfaces centered around text editors. All interactions with the program must be performed by modifying the live code in this textual interface. Parameters of a program must be modified inside the live code and can only be assigned to discrete values (Lawson 2015). Restricting interaction with the program only through textual interfaces is an aesthetic choice of the live coding community[1] that impairs the usability of live coding by not affording interactions that allow for easy exploration of the parameter space.

Video jockeys (VJs) are the visual counterpart to the musical disk jockeys (DJs); improvising graphics to accompany music and create audiovisual marriages that engage the senses (Faulkner and D-Fuse 2006). VJs tend to work with pre-rendered content such as looping video clips, and improvise their performances by layering and mixing them 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. VJing software is concerned with remixing existing content and typically does not allow for improvisation of new content like live coding. The VJ can control the speed and direction of playback or apply video effects but does not have fine-grained control over the video content itself (Bergström and Lotto 2008). Programmable behaviour is also limited as scripting often only appears where visual programming is supported such as in TouchDesigner[2] or VVVV[3] but does not offer complete control of the program behavior or straightforward live coding. The Praxis LIVE environment offers a hybrid model for live coding and visual programming that overcomes this limitation; maintaining that live code should be treated as a first-class citizen and never at a disadvantage of built-in components (Smith 2016).

One study of VJ practice from an HCI perspective observed performer's methods of expressive interaction and identified a number of characteristics that had an effect on their practice (Hook et al. 2011). The observations made in this study have influenced the design of a number of customised audiovisual performance tools such as residUUm and ABP (Olowe, Moro, and Barthet 2016; Correia 2015); developed following a user-centered approach (Correia and Tanaka 2014). While these tools succeed in delivering ready-made procedural graphics that are straightforward to interact with, they are not easily reused in other performances. In contrast, Bergström and Lotto (2008) suggests that code should be organised into reusable modules that can be shared among programmers and even non-programmers to allow for new performance compositions.

[1]https://toplap.org
[2]http://derivative.ca
[3]http://vvvv.org

There are a number of issues in existing performance practices and tools including the usability of live coding based on its reliance on textual interfaces, the lack of fine-grained content control in VJing, and the difficulty of reusing and sharing content in audiovisual tools. In this paper we propose the *code jockey* practice (*CJing*), a new hybrid practice performed by *code jockeys* (*CJs*) that addresses these issues by combining elements of live coding and VJ practice. CJing improves the usability of live coding by making use of interactions seen in VJing besides using a text editor. Code remains as the main source of content generation however, allowing for fine-grained control of existing content and improvisation of new content with the potential to be composited together and shared in reusable modules. In the next section we discuss the key ideas that enable CJing and then illustrate the CJ practice with *Visor*, a live coding environment for producing graphics in audiovisual performances. By focusing on usability, Visor aims to open up live coding to new audiences.

2 CJing Practice

We propose the new code jockey practice (CJing) to address the issues in existing performance practices and tools: the usability of live coding, the lack of fine-grained content control in VJing, and the difficulty of reusing and sharing content in audiovisual tools. The CJ practice is enabled when performance software supports three key ideas that encapsulate solutions to these issues: *User interfaces as an abstraction*, *complete content control*, and *code as a universal language*. We can place CJing in context of its broader subject areas: creative coding, live programming, and VJing as shown in Figure 1. 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. While we focus on VJing which produces graphical content, the same ideas could be applied with DJing for producing musical content.

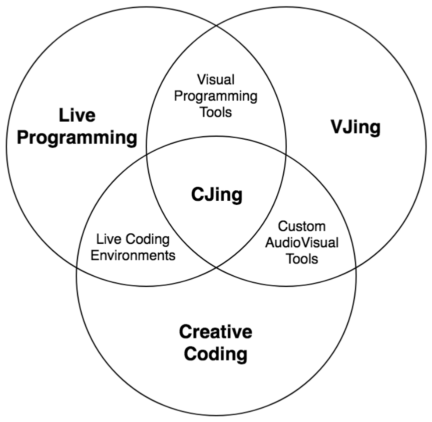


Figure 1: Software that supports CJing lies at the intersection of creative coding, live programming, and VJing related software tools.

1. *User interfaces as an abstraction*

CJing employs user interfaces that abstract upon live code, detecting changes to the live code and providing contextual interfaces to interact with the program. For example, a slider widget or hardware knob could manipulate parameters in the program, removing the need for the CJ to inspect and manipulate the live code using textual interfaces. A CJing GUI could also visualise the state of the program to improve the CJ's awareness of the inner workings of their live code. It is important that any incorporated user interfaces are only an abstraction of the live code, supplementing the program rather than replacing or implementing unique behaviour that would render live code obsolete.

2. *Complete content control*

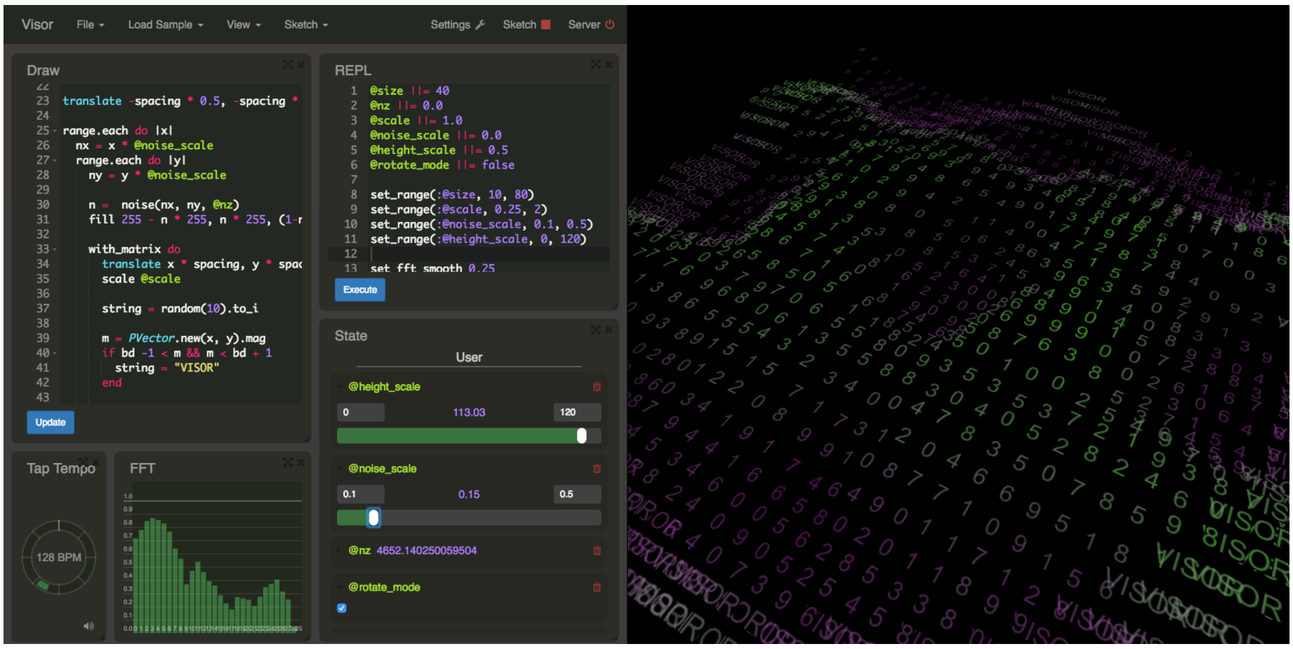
CJing should allow for complete control of procedurally generated content, enabling the performer to edit any part of the final output. CJs can improvise content from scratch, remix existing content, and composite multiple content elements together. For example, live coding provides the low level aspect of content creation and manipulation while VJing provides the high level aspect of orchestrating the final output through 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 make use of the strengths of live coding and VJing to perform with procedurally generated content.

3. *Code as a universal language*

CJing uses code to generate content from scratch, constructing the final output primarily through the use of algorithms. 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 CJ practice. If a DJ mixes musical tracks and a VJ mixes video clips then a CJ must mix code in their performances.

3 Visor

To illustrate CJing we developed Visor[4], a live coding environment for producing graphics in audiovisual performances. Live coding is performed in the Ruby language and graphical capabilities are provided by Processing (Reas and Fry 2006). The main features of Visor were developed with the key ideas of the CJ practice in mind, as shown in Table 1. The Visor interface shown in Figure 2 consists of a number of GUI components that make up the core user experience. There is the *REPL editor* (read-eval-print-loop) which is a space for performing actions that only occur once, such as defining a variable or method. These variables are displayed in the *state management interface* and can be mapped to hardware devices for interactive manipulation. The *draw loop editor* specifies the code that is run every frame. The code in the draw loop editor can access variables and methods defined by the REPL editor. Finally the *tap tempo interface* and *FFT display* are interfaces for audio-reactive techniques that can be accessed from the live code. Each of these GUI components can be moved and resized to allow the performer to customise the interface.



```
translate -spacing * 0.5, -spacing *

range.each do |x|
  nx = x * @noise_scale
  range.each do |y|
    ny = y * @noise_scale

    n =  noise(nx, ny, @nz)
    fill 255 - n * 255, n * 255, (1-r

    with_matrix do
      translate x * spacing, y * spac
      scale @scale

      string = random(10).to_i

      m = PVector.new(x, y).mag
      if bd -1 < m && m < bd + 1
        string = "VISOR"
      end
```

```
@size ||= 40
@nz ||= 0.0
@scale ||= 1.0
@noise_scale ||= 0.0
@height_scale ||= 0.5
@rotate_mode ||= false

set_range(:@size, 10, 80)
set_range(:@scale, 0.25, 2)
set_range(:@noise_scale, 0.1, 0.5)
set_range(:@height_scale, 0, 120)

set fft smooth 0.25
```

@height_scale 0 113.03 120

@noise_scale 0.1 0.15 0.5

@nz 4652.140250059504

@rotate_mode ✓

Figure 2: The Visor user interface (left) with corresponding rendered output (right).

3.1 Performance Scenario

We now describe the typical actions conducted by a CJ during a Visor performance with corresponding figures 3-7. The first action is related to the configuration of the software and only occurs at the beginning of the performance while the remaining actions are conducted throughout the performance.

1. Configuration

Visor offers a number of settings to allow the CJ to configure their performance. These include configuring the audio input device, MIDI device and the Processing renderer display settings. An option to overlay the draw loop code onto the rendered output is also available, though the same code string can be accessed in the live code using a method.

2. Live coding

Live coding is performed by the CJ in the draw loop editor. The draw loop code typed here is executed every frame as per the `draw` method used by Processing. Syntax errors and run-time errors are shown in the editor by highlighting the line the error occurred and presenting the error message. Figure 3 shows our live coded program that renders a ring of geometric shapes.

3. State definition

The CJ can define state in Visor using Ruby's instance variables, indicated by the `@` character at the start of a variable name. State can be read and written to in both the draw loop editor and the REPL editor. Any code submitted from the REPL editor is only executed once and is used for defining or modifying instance variables, methods, and classes. Visor's REPL is represented as a multi-line text field and the code is kept after execution. Any CJ defined state in Visor

[4] https://visor-live.github.io

is automatically visualised in the state management interface. Figure 3 shows how instance variables have been used to define state and methods in our live coded program; parameterising the number of shapes drawn, the size of the shapes, and keeping track of how far the ring has rotated.

4. State manipulation by the state management interface

If a variable in the state is a number then a range is automatically estimated so the CJ can scrub through its parameter space using a slider in the state management interface. The CJ can also manually adjust the minimum and maximum values of the range in the interface or by using the `set_range` method in the live code. If a variable is not needed anymore then it can be deleted from the interface or collapsed to take up less visual space. Figure 4 shows how ranges have been assigned for variables in our live coded program and how the rendered result changes when the sliders are scrubbed.

5. State manipulation by MIDI controller

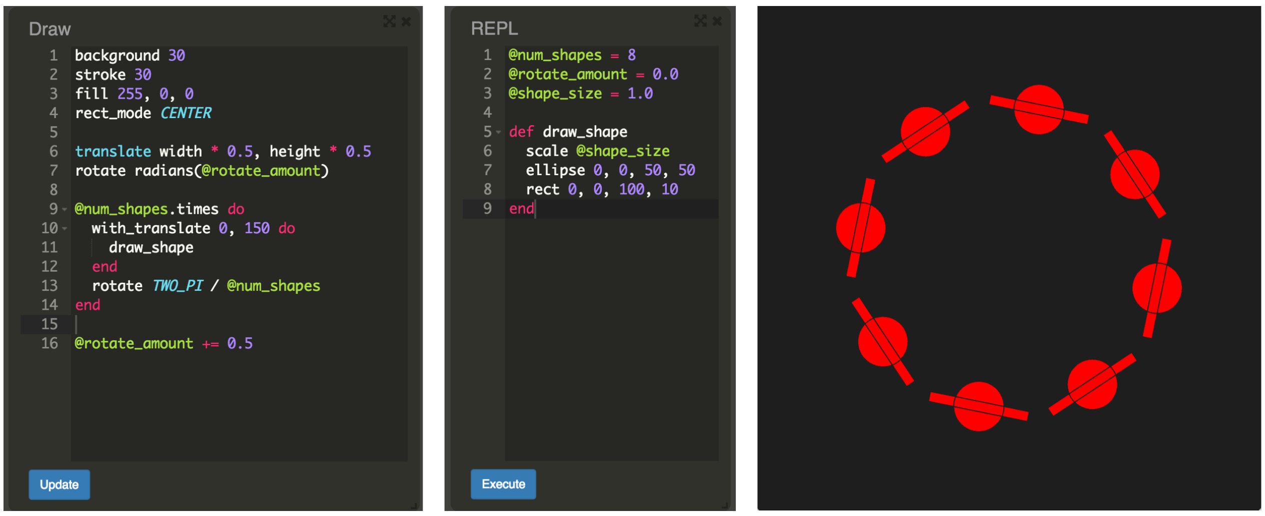
A CJ can also use a MIDI controller to interact with the live code. The state of control sliders, knobs and note button events can be accessed using methods in the live code. For example, the `interpolate` method can be used in conjunction with a slider method to control the value of a variable in the state. Figure 5 shows how the number of shapes drawn and the size of the shapes are now mapped to two different sliders in our live coded program. The rotation speed of the ring is also increased when the first note button is pressed down.

6. Tap tempo integration

The tap tempo enables beat based behaviour in Visor. The tap tempo interface allows the performer to specify a tempo in BPM (beats per minute) that matches the music they are currently listening to. The tempo is set by repeatedly hitting the tap tempo button in the tap tempo interface or by using a keyboard shortcut. The tap tempo interface displays the current BPM and visualises the temporal progress through the current beat. The `on_beat` method can then be used to trigger events in the live code each time a beat is hit. The `beat_progress` method can also be used to access a normalized value of the temporal progress. Figure 6 shows how the tap tempo has been integrated into our live coded program; each time a beat is hit the ring will change colour and pulse outwards from the center of the screen.

7. FFT integration

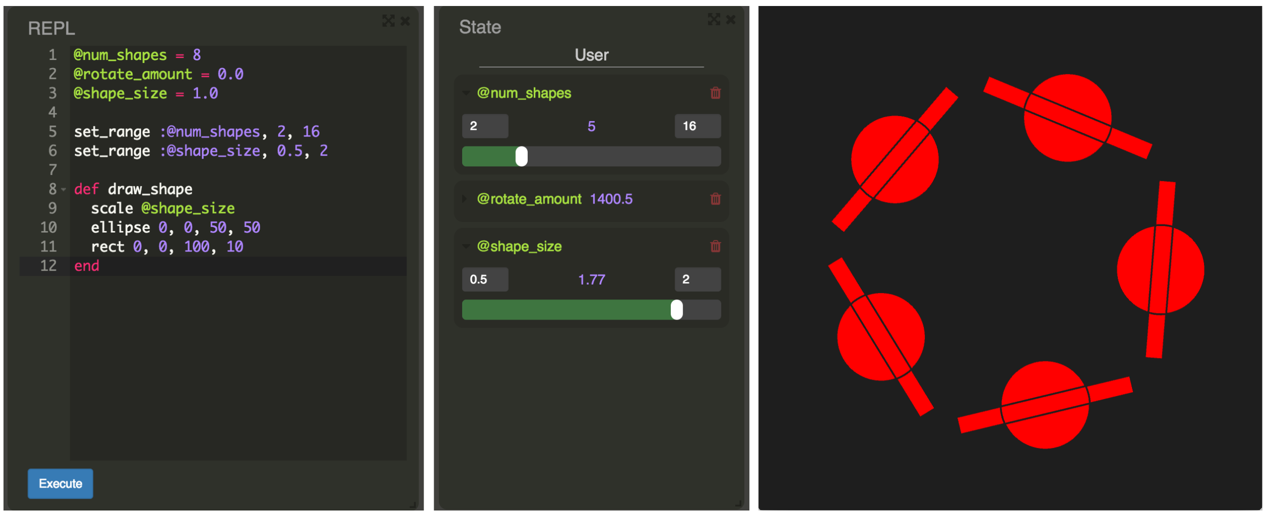
Visor is grounded on real-time audio analysis techniques to produce audio-reactive content, removing any reliance on the musician to provide specifically formatted data on protocols like OSC that might not always be available. The FFT display visualises the current state of any live music in a frequency spectrum. The display interface allows the performer to make informed decisions about how the graphics should react to the music. The display graphs audio amplitude against frequency where the amplitude is graphed between zero and one. The FFT data can be accessed in the live code using methods such as `volume` which returns the overall amplitude of the audio or `fft_range(n1, n2)` which returns the sum of the frequency bands in the range `n1` to `n2`. Figure 7 shows how the FFT data has been integrated into the `draw_shape` method in our live coded program; the ellipse diameter is now mapped to low to mid frequency sounds and the rectangle width is now mapped to high frequency sounds.



```
 1  background 30
 2  stroke 30
 3  fill 255, 0, 0
 4  rect_mode CENTER
 5
 6  translate width * 0.5, height * 0.5
 7  rotate radians(@rotate_amount)
 8
 9  @num_shapes.times do
10    with_translate 0, 150 do
11      draw_shape
12    end
13    rotate TWO_PI / @num_shapes
14  end
15
16  @rotate_amount += 0.5
```

```
1  @num_shapes = 8
2  @rotate_amount = 0.0
3  @shape_size = 1.0
4
5  def draw_shape
6    scale @shape_size
7    ellipse 0, 0, 50, 50
8    rect 0, 0, 100, 10
9  end
```

Figure 3: Live coding in Visor. Draw loop editor (left), REPL editor (center) and the rendered output (right). Note how state variables have been defined using Ruby's instance variables and are accessible from both the draw loop and REPL editors (e.g: `@num_shapes` on line 9 in the draw loop editor and line 1 in the REPL editor).



```
 1  @num_shapes = 8
 2  @rotate_amount = 0.0
 3  @shape_size = 1.0
 4
 5  set_range :@num_shapes, 2, 16
 6  set_range :@shape_size, 0.5, 2
 7
 8  def draw_shape
 9    scale @shape_size
10    ellipse 0, 0, 50, 50
11    rect 0, 0, 100, 10
12  end
```

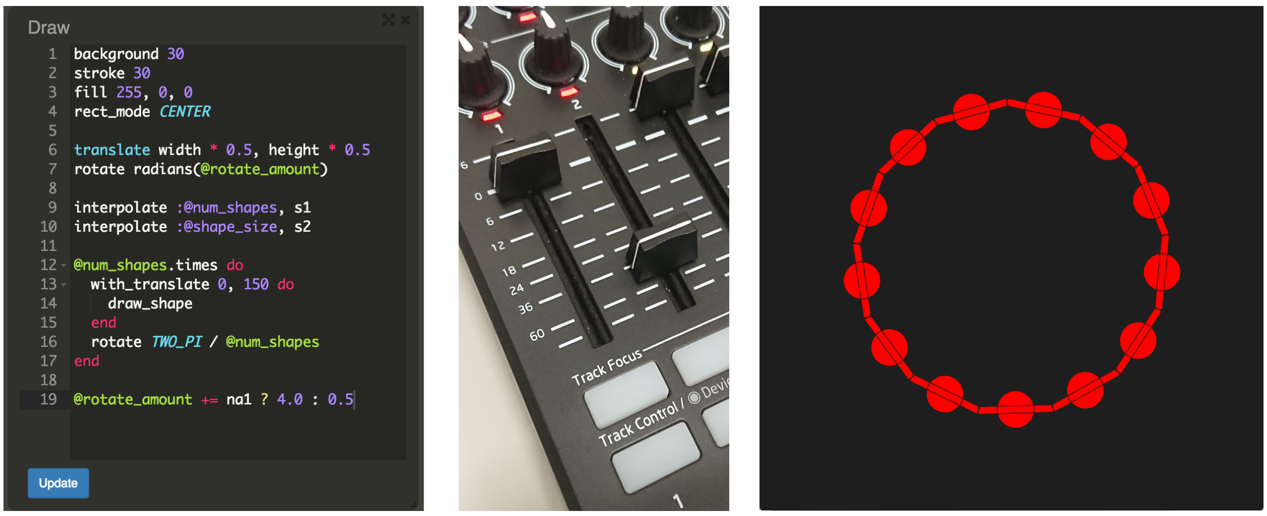
User

@num_shapes
2 5 16

@rotate_amount 1400.5

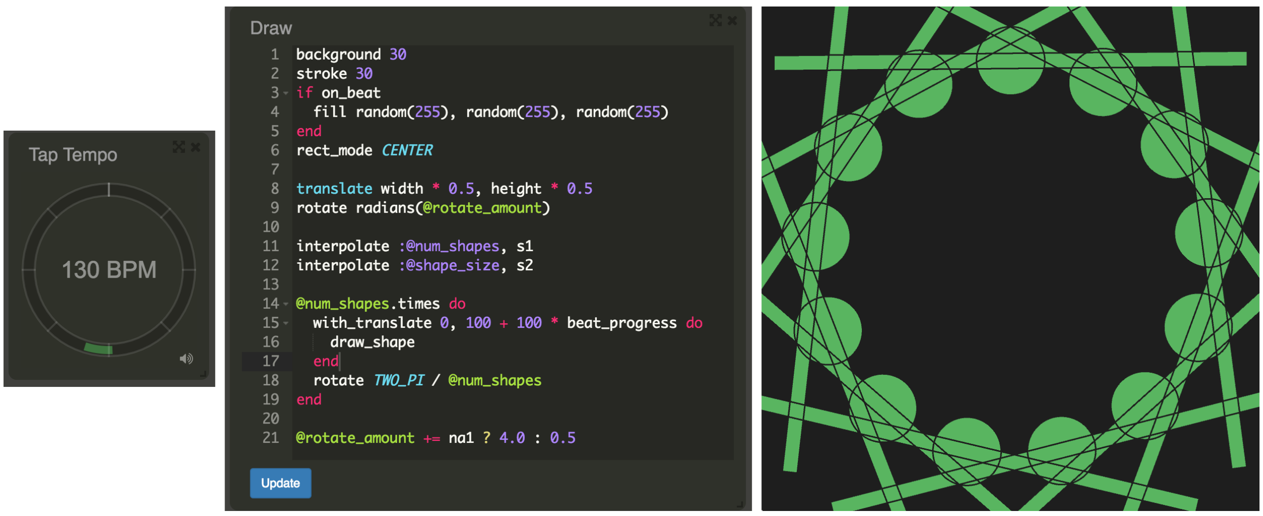
@shape_size
0.5 1.77 2

Figure 4: State manipulation by the state management interface in Visor. REPL editor (left), state management interface (center) and the rendered output (right). Note how ranges have been assigned to the state variables using the set_range method (lines 5 & 6) and how scrubbing the sliders has affected the rendered output.



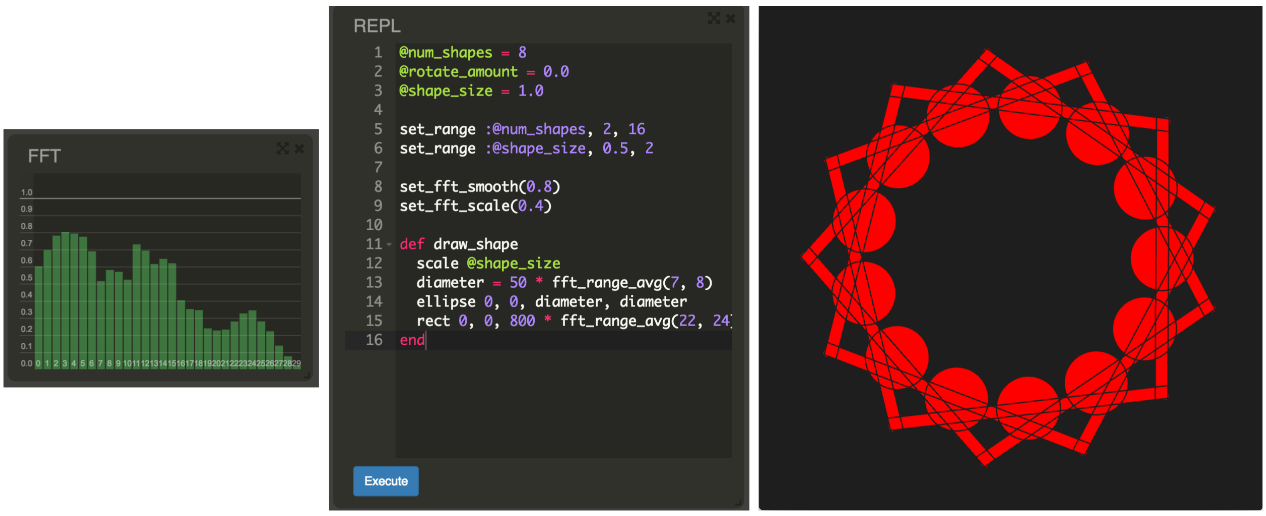
```
 1  background 30
 2  stroke 30
 3  fill 255, 0, 0
 4  rect_mode CENTER
 5
 6  translate width * 0.5, height * 0.5
 7  rotate radians(@rotate_amount)
 8
 9  interpolate :@num_shapes, s1
10  interpolate :@shape_size, s2
11
12  @num_shapes.times do
13    with_translate 0, 150 do
14      draw_shape
15    end
16    rotate TWO_PI / @num_shapes
17  end
18
19  @rotate_amount += na1 ? 4.0 : 0.5
```

Figure 5: State manipulation by MIDI controller in Visor. Draw loop editor (left), MIDI controller (center) and the rendered output (right). Note how sliders have been assigned to the state variables using the interpolate method in combination with the s1 and s2 (lines 9 & 10) methods and how moving the sliders has affected the rendered output.



```
 1  background 30
 2  stroke 30
 3  if on_beat
 4    fill random(255), random(255), random(255)
 5  end
 6  rect_mode CENTER
 7
 8  translate width * 0.5, height * 0.5
 9  rotate radians(@rotate_amount)
10
11  interpolate :@num_shapes, s1
12  interpolate :@shape_size, s2
13
14  @num_shapes.times do
15    with_translate 0, 100 + 100 * beat_progress do
16      draw_shape
17    end
18    rotate TWO_PI / @num_shapes
19  end
20
21  @rotate_amount += na1 ? 4.0 : 0.5
```

Figure 6: Tap tempo usage in Visor. Tap tempo interface (left), draw loop editor (center) and the rendered output (right). Note how the on_beat and beat_progress methods (lines 3 & 15) are used and how the rendered output is affected.



```
 1  @num_shapes = 8
 2  @rotate_amount = 0.0
 3  @shape_size = 1.0
 4
 5  set_range :@num_shapes, 2, 16
 6  set_range :@shape_size, 0.5, 2
 7
 8  set_fft_smooth(0.8)
 9  set_fft_scale(0.4)
10
11  def draw_shape
12    scale @shape_size
13    diameter = 50 * fft_range_avg(7, 8)
14    ellipse 0, 0, diameter, diameter
15    rect 0, 0, 800 * fft_range_avg(22, 24
16  end
```

Figure 7: FFT usage in Visor. FFT display (left), REPL editor (center) and the rendered output (right). Note how the `fft_range` method (lines 13 & 15) is used to reference the data shown in the FFT display and how the rendered output is affected.

3.2 Design and Architecture

Visor is designed using a client-server architecture as illustrated in Figure 8. While both client and server applications can be run on the same machine, the architecture opens up the possibility to run the client and server on different machines. This enables remote live coding where the performer live codes on one machine and the output is rendered on another. The client application is written as a desktop application using the Electron[5] 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 draw loop editor and state management interface. The server application is written in JRuby[6], an implementation of the Ruby programming language on the Java virtual machine. The server application is responsible for hot swapping live code, maintaining program state, managing MIDI devices, calling into the Processing API, and keeping a handle to the Processing rendering window. The client and server applications communicate using HTTP and WebSockets.

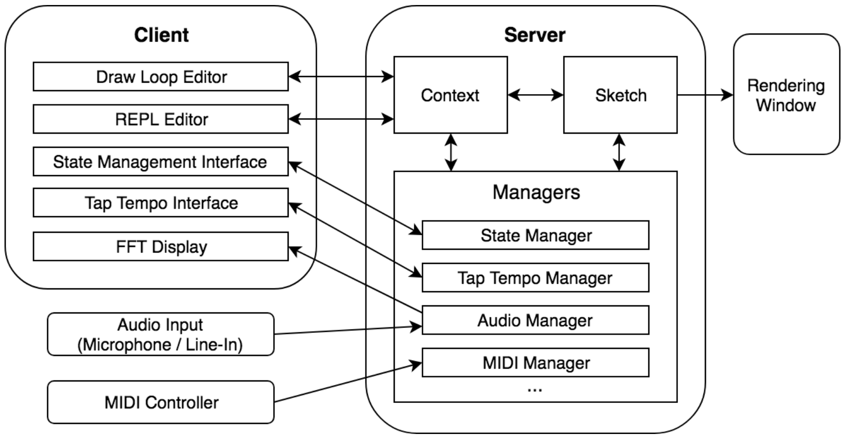


Figure 8: Visor's client-server architecture. The client application consists of GUI components that interface with the server. The server application consists of classes that interface with each-other, the client, the Processing API and external inputs.

The server application consists of a number of classes illustrated in Figure 8. The `sketch` class 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` classes, each manager is concerned with a specific part of Visor's internal behaviour such as the real-time FFT, tap tempo algorithm or state management. Each manager provides callback methods to be called from the Processing sketch lifecycle. For example, code to be called before the `draw` method is run can be specified.

The `context` is a class that encapsulates the live coded behaviour and state defined by the performer. The context defines one method named draw that executes the live code. The sketch class calls this `draw` method on the context every frame.

[5]http://electronjs.org
[6]http://jruby.org

Delegation is used to allow the context to access methods in the sketch and manager classes. Each manager can choose to expose a number of methods to the context. For example, this allows the live code to access the `on_beat` method from the tap tempo manager. Each manager can also expose its own internal state to the context 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.

Visor makes use of Ruby's metaprogramming features to enable live coding by redefining program behaviour and evaluating arbitrary code strings. There are two sources of live code in Visor that are evaluated at different times. The first source is the draw loop code specified in the draw loop editor of the client. This code is placed in the `draw` method of the context class and is executed every frame as per the Processing sketch lifecycle. Whenever the performer submits the contents of the draw loop editor, the server application dynamically redefines the context's `draw` method to run the new code, effectively hot swapping it with the previous code. The second source of live code is from the REPL editor of the client. The REPL code is intended to be executed only once and so all of it is simply evaluated within the context class before the next draw loop occurs.

State is handled in Visor by utilising Ruby's instance variables. Live code is executed within the context class such that any defined instance variables are placed on the context instance itself. This means any state defined through the REPL can be read from and written to in the draw loop code and vice-versa. 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 context or any exposed state on managers. 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 context or a specific 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. All other data types such as strings, arrays, hashes or arbitrary objects are visualised using their string representations.

Due to the seamless integration between Java and Ruby in JRuby, Java methods can be transparently called from Ruby code. This means no bindings need to be specified between Visor and Processing. Visor can utilise most of the Processing API, enabling all kinds of graphics rendering including 2D and 3D primitives, text, images, shaders and 3D models. Visor also extends Processing by making use of Ruby language features. Ruby supports the use of blocks which are lambda expressions. A Ruby method can accept a block as an argument and perform useful operations before and after executing the block. For example, the `with_matrix` method makes use of blocks to replace the need to call `push_matrix` and `pop_matrix`. This is helpful as forgetting to call `pop_matrix` can cause unintended program behaviour. Listing 1 demonstrates the traditional approach with the Processing API while Listing 2 demonstrates the new approach.

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

Listing 1: Traditional approach to handling the matrix stack with the Processing API in Visor.

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

Listing 2: New approach to handling the matrix stack using a method that accepts a block as an argument in Visor.

Table 1: Visor's main features and how they map to the three key ideas of the CJ practice.

Feature	*User interfaces as an abstraction*	*Complete content control*	*Code as a universal language*
Live coding in the Ruby language.		Y	Y
Draw loop editor for updating the Processing draw loop at run-time.		Y	
REPL editor for interfacing with live coded state and defining methods or classes.		Y	
Automatic visualisation of live coded state.	Y		
Indirect manipulation of live coded state using a GUI.	Y	Y	
Tangible manipulation of live coded state using a MIDI controller.	Y	Y	
Customisable, reconfigurable interface.	Y		
Interfaces for engaging with real-time audio-reactive techniques: Tap tempo and FFT.	Y		

4 Discussion

We now analyse Visor's exhibition of the key ideas of the CJ practice and how the implementation of these ideas relates to the issues identified in existing performance practices. Visor's state management interface allows for visualisation and indirect manipulation of state, aligning with the idea of *user interfaces as an abstraction* and showing how the usability of live coding can be improved by reducing reliance on typing interfaces. The state management interface shows Visor's ability to orchestrate high level aspects of performance as manipulation of parameters using the GUI or hardware controllers can directly affect the final output without the need to edit any code. Visor also enables low level aspects of performance by supporting live coding of Processing in the Ruby language, enabling improvisation of new graphical content from scratch or editing of existing content. This support for live coding provides fine-grained control of the final output that is not present in VJing software. This combination of low and high level interactions show how Visor supports *complete content control*. The aspect of *code as a universal language* is supported as all content in Visor is produced using code.

CJing improves the live coding practice by allowing it to be embedded as a process in larger software, exposing the practice to performers such as VJs who typically only utilise GUIs. This is shown in Visor by the combination of live coding and user interfaces to produce graphics in audiovisual performances. Based on the first author's experiments with Visor in live performance, we have identified three potential use cases of Visor that cater for live coding and VJing performers with varying levels of programming ability. The first use case is the ability to live code a program from scratch to produce graphics, catering for live coding performers. The second is the ability to perform with an existing Visor program that exposes parameters to be manipulated using the GUI or with hardware devices. This second use case caters for VJs and importantly, people with little or even no programming experience. This level of accessibility allows programs that procedurally generate content to be utilised by performers without programming experience while also exposing them to live coding. This leads into the third use case which is the use of an existing Visor program as the starting point for a performance. This program can be remixed or built upon using live coding throughout the performance, catering for both live coders and VJs with programming experience. Outside live performance, we believe Visor will also cater for general creative coding audiences as it extends upon Processing with an improved user experience. This is shown by Visor's ability to automatically provide interfaces for interacting with program parameters and the speeding up of the development lifecycle due to the immediacy of live coding, both important qualities for creative coding.

5 Related Work

Visor shares traits with a number of existing live coding, VJing, and audiovisual performance tools. These tools have influenced the proposed CJ practice or include features that enable CJing.

LiveCodeLab (Della Casa and John 2014) is a web based live coding environment that makes use of its own domain specific language called LiveCodeLang. LiveCodeLab 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 (Salazar 2017) is a programming environment that allows for gestural interactions in musical performance. Auraglyph 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 user interface abstractions but instead of direct control through gestures it makes use of the state management interface and mappings to MIDI controllers that offer both indirect and tangible manipulation respectively.

Resolume[7] is a popular VJ tool. Resolume supports audio-reactive effects such as audio-analysis based on a real-time FFT where individual bands can be used to drive video effect parameters or a tap tempo functionality where the performer can tap a keyboard shortcut to the beat of the music and drive video effects by the computed BPM. Visor incorporates both of these audio-reactive features by visualising the FFT and BPM in the client GUI and allows the live code to access them through exposed methods to generate audio-reactive content. While Resolume focuses on traditional video mixing, Visor instead allows for improvisation of new content by live coding.

KodeLife[8] is a tool that enables live coding of GLSL shaders for live performance. Shaders can provide highly complex and engaging graphics 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 graphics. As Processing

[7]http://resolume.com
[8]https://hexler.net/software/kodelife

is aimed at learner programmers it is easier for performers to pick up and does not require the shift in mindset towards parallel computing that shaders and GPU programming require.

Mother (Bergström and Lotto 2008) is 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. Visor is similar to Mother in that it allows for VJing with Processing but it extends upon Mother by allowing new content to be live coded, instead of just using pre-written sketches. Visor does not support layering of different sketches through a user interface or for code to be shared as reusable modules but these ideas have directly influenced the proposed CJ practice and are planned as future work for Visor. Mother is also mentioned as an example of a new creative coding practice called code bending (Bergstrom and Lotto 2015) which aims to refurbish existing software by altering it to fit new purposes. Code bending is enabled by integrating communication protocols into existing software. For example, Mother uses OSC to allow Processing sketches to communicate and be composited together during live performance. While CJing is also concerned with how existing content can be modified and composited together, it does not approach it from the perspective of networking multiple programs but instead focuses on providing a single environment where content can be live coded in a common language.

Praxis LIVE (Smith 2016) is a hybrid IDE that allows for live coding of Processing in the Java language and offers visual programming of node graphs where the source code of each node can be individually edited. 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 to 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-reactive 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.

JRubyArt[9] is a wrapper for Processing that enables creative coding in the Ruby language like Visor. JRubyArt allows for live coding where a source code file is watched for changes over time and a terminal provides REPL interaction. Visor instead opts for a GUI with code editing and interactive features built-in specifically for live coding in audiovisual performances. Sonic Pi (Aaron 2016) is another live coding environment that supports live coding in Ruby. Developed with a client-server architecture, Sonic Pi is used to live code music. While Visor shares a similar software design it instead focuses on producing graphics and so to a certain extent Visor is a visual counterpart to Sonic Pi.

The role of HCI in live performance has been explored by Hook et al. (2011) who observed a number of characteristics of expressive interaction in VJ practice. These characteristics are summarised by Correia (2015) as: *reconfigurable interfaces* (the ability to reorganise the controls to fit a particular performance); *haptically direct* (using hardware controllers and having a physical connection to the system); *parallel interaction* (simultaneous control of multiple parameters); *constraining interaction* (the importance of constraints and focus); *immediacy* (immediate response from the software); *manipulable media* (desire for powerful and varied manipulation of media); and *visible interaction* (to make the performer's interaction visible to the audience). Visor presents all of these characteristics except for that of *visible interaction*. While Visor supports the ability to project the source code on top of the graphics 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 need to make a distinction between live coding and CJing practice: while live coding is utilised by CJing, some aspects of live coding such as code projection may not make sense in the new practice. For example, VJs utilise interfaces that prevent errors in their performance as a matter of usability while live coders are more likely to embrace errors as shown by the projected code.

6 Conclusion

We have introduced CJing, a new hybrid practice that addresses issues in performance practices and tools, including the usability of live coding based on its reliance on textual interfaces, the lack of fine-grained content control in VJing, and the difficulty of reusing and sharing content in audiovisual tools. CJing combines elements of live coding and VJ practice based on the ideas of *user interfaces as an abstraction*, *complete content control*, and *code as a universal language*. To illustrate CJing we have introduced a new live coding environment called Visor that offers a new way to improvise graphics in audiovisual performances with a new method for live coding of Processing using the Ruby language. While we have discussed the CJ practice from the perspective of live coding graphics in Visor, we believe that the same principles could be applied to music. User interfaces from DJing software could be used instead of VJing software to influence the design of a live coding environment for producing music.

The first author has performed with Visor at multiple meetups and three live performances with positive audience feedback and interest from other performers. The first author is continuing to develop Visor to further illustrate CJing. Planned future work for Visor includes support for a system to share code, emphasising the idea of *code as a universal*

[9] http://monkstone.github.io/about

language by enabling the reusability and sharing of Visor programs. Further ideas from VJing software will also be implemented, such as the ability to organise content into layers that can be reordered, mixed with other layers, or have video effects applied to them. Support for layers in Visor would provide high level interactions based on user interfaces that abstract upon live code, further emphasising the ideas of *user interfaces as an abstraction* and *complete content control*. Finally, we intend to perform user testing on Visor to evaluate its usability and better understand how user interfaces might improve live coding.

References

Aaron, Sam. 2016. "Sonic Pi – performance in education, technology and art." *International Journal of Performance Arts and Digital Media* 12 (2): 171–78.

Bergstrom, Ilias, and R. Beau Lotto. 2015. "Code Bending: A New Creative Coding Practice." *Leonardo* 48 (1): 25–31.

Bergström, Ilias, and Beau Lotto. 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*, 11–12.

Collins, Nick, Alex McLean, Julian Rohrhuber, and Adrian Ward. 2003. "Live Coding in Laptop Performance." *Organised Sound* 8 (3): 321–30.

Correia, Nuno N. 2015. "Prototyping Audiovisual Performance Tools : A Hackathon Approach." In *International Conference on New Interfaces for Musical Expression*, 12–14.

Correia, Nuno N, and Atau Tanaka. 2014. "User-Centered Design of a Tool for Interactive Computer-Generated Audiovisuals." In *International Conference on Live Interfaces*.

Della Casa, Davide, and Guy John. 2014. "LiveCodeLab 2.0 and Its Language LiveCodeLang." In *Proceedings of the 2nd Acm Sigplan International Workshop on Functional Art, Music, Modeling & Design*, 1–8.

Faulkner, M, and D-Fuse. 2006. *VJ: Audio-Visual Art and VJ Culture: Includes DVD.*

Hook, Jonathan, David Green, John McCarthy, Stuart Taylor, Peter Wright, and Patrick Olivier. 2011. "A VJ Centered Exploration of Expressive Interaction." In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 19:1265–74. ACM.

Lawson, Shawn. 2015. "Performative Code: Strategies for Live Coding Graphics." In *Proceedings of the First International Conference on Live Coding*, 35–40.

Olowe, Ireti, Giulio Moro, and Mathieu Barthet. 2016. "residUUm : user mapping and performance strategies for multilayered live audiovisual generation." In *International Conference on New Interfaces for Musical Expression*, 271–76.

Reas, Casey, and Ben Fry. 2006. "Processing: programming for the media arts." *AI & SOCIETY* 20 (4): 526–38.

Salazar, Spencer. 2017. "Searching for Gesture and Embodiment in Live Coding." In *Proceedings of the International Conference on Live Coding.*

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

Tanimoto, Steven L. 2013. "A Perspective on the Evolution of Live Programming." In *Proceedings of the 1st International Workshop on Live Programming*, 31–34.