Chris Kerr and Daniel Holden

Optimizing Code for Performance: Reading *./code --poetry*

Introduction: code poetry's compound eye

There is something unnerving about staring at an eye, especially one that meets your gaze. In our book *./code --poetry*, "compound_eye.rb" (see Holden and Kerr 2016a) spans two facing pages, and it appears to look at the reader from each page. "compound_eye.rb" is a code poem, written in the Ruby programming language.[1] We can think of code poetry as a subset of Alan Sondheim's category "'Codework' – the computer stirring into the text, and the text stirring the computer." Sondheim also describes codework as part of a movement "concerned with the intermingling of human and machine" (Sondheim 2001, 1). A close reading, and seeing, of "compound_eye.rb" will serve as an introduction to our project *./code --poetry*, and to the visual aesthetics of code poetry in general.

On the right-hand page is a functional computer program that has been altered poetically in a way that is human-readable. The code poem is arranged on the page so that the central section looks like a hexagonal ommatidium: one of the many units that make up an insect's eye. On the left-hand side of the page are five snapshots from an animation that the code poem creates when it is run on a computer. Like the code poem, the output is made of alphanumeric characters. The output is an example of ASCII art, comprised of characters defined by the American Standard Code for Information Interchange. The text describes a patient who has undergone surgery. The patient experiences insects crawling on their body. The insects may or may not be hallucinations.

So far, we have described the print version of "compound_eye.rb," but it also exists online. The website https://code-poetry.com (see Holden and Kerr 2016b) hosts some of the code poems from our book. In the online version, we see not static flipbook frames but a moving ASCII animation, a looping GIF of the code poem's output. There is no single object to look at here: the code poem exists as input and output, in print and online (Fig. 18).

W. J. T. Mitchell writes that "the very notion of vision as a *cultural* activity necessarily entails an investigation of its non-cultural dimensions, its pervasiveness as a sensory mechanism that operates in animal organisms all the way from the flea to the elephant" (Mitchell 2002, 92). "compound_eye.rb," with its flea's

1 https://www.ruby-lang.org/en/.

https://doi.org/10.1515/9783111299334-009

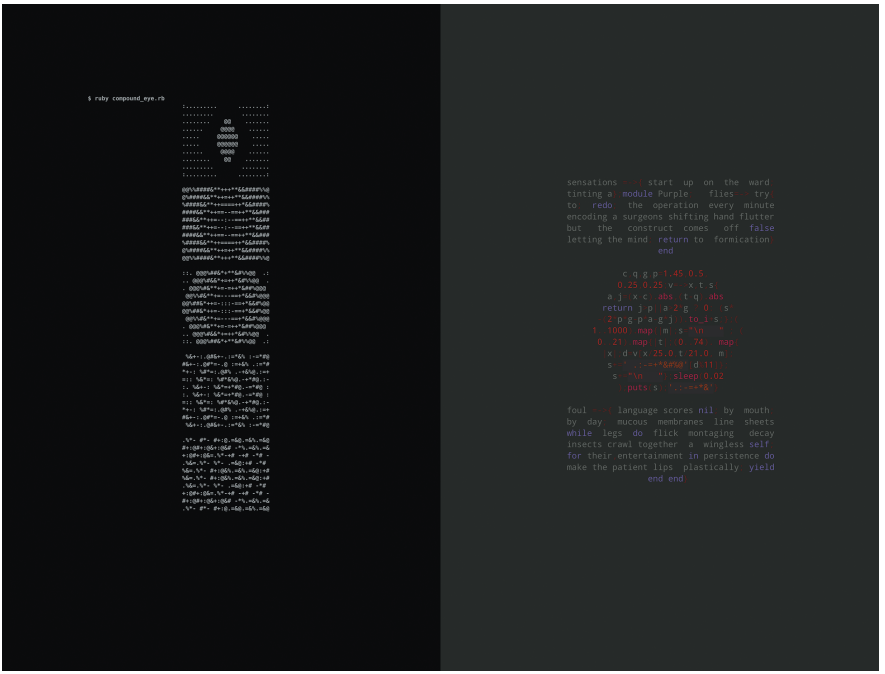

Fig. 18: Daniel Holden and Chris Kerr. “compound_eye.rb.” *./code --poetry*. 2016a.

eye, can be seen as an investigation of the non-cultural, or scientific dimensions of vision. Code poetry, grounded in computer science, might be an ideal medium for this scientific element of vision. However, code poetry should not be seen as a neutral escape from the cultural aspect of vision: it has a many-eyed, cultural element, too, because science is always culturally embedded.

“compound_eye.rb” is written in a monospaced font, the signature aesthetic of code poetry. Programmers use these fonts, where each character occupies the same horizontal space. It is used in *code {poems}* as well, an anthology of code poetry edited by Ishac Bertran, for example in “UNTITLED (LOVE)” by Nataliya Petkova (2018). This aspect of code poetry owes a debt to twentieth century typewriter art, for instance Peter Finch’s “texture poem for the moons of stars” (1972). We argue that code poetry is a continuation of these practices with digital means and is therefore digital concrete poetry. Both the input and output of “compound_eye.rb” consist of ASCII characters. For example, the characters “= - > {” appear n the first line of the code poem, where they are a functional part of the program. In the output, these symbols are used in a purely visual way, to paint an animation. Unusual syntax is another characteristic of code poetry. In “mucous mem-

branes line sheets while legs do flick," for instance, the word "do" is seemingly redundant, even archaic. This syntactical contortion is a sign that the text is being written for two audiences, one human, one computational. The word "do" is in fact a functional keyword in the Ruby programming language. Addressing different kinds of "readers" as well as combining alphanumeric symbols and unusual syntax are central features of code poetry in general such as Francesco Aprile's "laravel poems" (2020).

Finally, the code poem has coloured, bold, italic, underlined and emphasized text, which replicates the syntax highlighting of code text editors, which colour elements of the code according to their category in the language, and might, like this poem, have a dark background. The colouring of the syntax highlighting helps programmers to read the code more efficiently. In "compound_eye.rb," the syntax highlighting, too, sometimes serves a functional role as a readability aid: "do" is purple because it is a keyword in the programming language Ruby. At other times, the syntax highlighting has a more aesthetic function. The central hexagon of the code poem is bright red, to mimic the brilliant red of an ommatidium. This is an aesthetic choice: the colouring can be freely adapted for artistic effect. The red draws the reader's eye, as if to a cross-section of a visual organ, held on a microscope's slide by the surrounding text. In addition, this red hexagon might also remind the reader of the logo for the Ruby language: its clean angles resemble a faceted gem. For an example of syntax highlighting in another code poem, see "//Lockdown Dérive" by Brian James (2020, 11).

The aesthetics of syntax highlighting can be read in the light of visual culture studies. One of Mitchell's "eight counter-theses on visual culture" is as follows: "Visual culture entails a meditation on blindness, the invisible, the unseen, the unseeable, and the overlooked; also on deafness and the visible language of gesture; it also compels attention to the tactile, the auditory, the haptic, and the phenomenon of synesthesia" (Mitchell 2002, 90). "compound_eye.rb" asks us to look at a visual organ too small to be seen clearly by the eye, on an animal we might be too disgusted to look at. This disgust is compounded by the tactile impression conveyed by the word "formication" in the code poem: this impression becomes entangled with the texture of the ASCII characters in the output. When the viewer looks at this animation, are they looking at a single eye, or a swarm of ASCII character insects, out of which an eye emerges like a hallucination? Moreover, we can even detect Mitchell's synaesthetic component in syntax highlighting, which recalls colour-grapheme synaesthesia, where alphanumeric characters are strongly associated with specific colours (cf. Bhanoo 2011).

Code poetry, with its visual, tactile, synaesthetic language is a rich site for exploring visual culture in Mitchell's terms. Code poetry is a multisensory field, where visual art, literature, print and online words mix. But what about Mitch-

ell's "auditory" component? In "compound_eye.rb" the "patient lips plastically [...] yield" to the insects. The lips are the patient's, yet they are also patient in another sense, waiting, silent, passive, to be entered. Can the animation be seen as a mouth, as well as an eye? This code poem, in its muteness, seems to be crying out to be voiced, or performed. If code poetry is concerned with how language looks, it is also concerned, synaesthetically, with how the visual sounds.

Human and computer performance: a code poetry typology

Performance has three aspects in the context of code poetry. First, like other types of poetry, code poetry can be performed by a human, i.e., it can be read aloud. Second, a computer can also read (that is to say, interpret, or run) a program. Thus, reading code poetry entails both human and non-human types of reading. Performance has another meaning in a computing context. The Oxford English Dictionary defines one meaning of "performance" as "the capabilities, productivity, or success of a machine, product, or person when measured against a standard" (OED Online 2022). In an environment where computer memory is scarce, the performance of a program can refer to how efficiently it uses memory. Conversely, in an environment with more abundant memory, the speed at which the program runs might be more important. Code can be optimized for performance, whether at work, to keep the client happy, or after hours, for the love and challenge of it. Finally, some code poetry can perform, or produce an output. In other words, code poems can have outputs just as "compound_eye.rb" performs its animated output. Code poetry can be optimized for performance in all three senses.

Scholars of digital literature have highlighted performance as an important, definitive dimension of code poetry. Geoff Cox writes that "[l]ike poetry, the aesthetic value of code lies in its execution, not simply its written form" (Cox et al. 2001, 30). Elsewhere, Cox, Alex McLean and Adrian Ward see "code as performative: that which both performs and is performed" (Cox et al. 2004, 161). We can use the three senses of performance that we have identified to build a tentative typology of code poetry: human-readable code poetry, machine-readable code poetry, and code poetry that produces an output. For an example of another typology of code poetry, see Alan Sondheim's three-part "tree [. . .] taxonomy" of codework (Sondheim 2001, 4). In brief, Sondheim's taxonomy is hierarchical, differentiating between work with "surface language" and work with "submerged code." This "submerged code" may or may not become "emergent content," break-

ing through to the surface (Sondheim 2001, 4). Sondheim's typology moves from the surface leafs to the invisible roots of a tree. By contrast, our typology is a non-hierarchical matrix, which uses types of performance rather than surface and depth as its criteria.

In the following table (Tab. 1), we propose that code poetry can be classified into eight types, according to whether it can be performed by humans, by computers and whether it, in turn, performs, or produces an output.

Tab. 1: Code poetry typology by Daniel Holden and Chris Kerr.

Type	Human performable?	Computer performable?	Performs (output)?
1	N	N	N
2	N	Y	N
3	N	Y	Y
4	N	N	Y
5	Y	N	N
6	Y	Y	N
7	Y	Y	Y
8	Y	N	Y

In the following, we will provide examples of some of these types of code poetry. Due to the page limit of the article, however, there is not enough space to discuss all the types listed in the table.

Type 5 (YNN) code poetry can be performed by humans, but not computers, and therefore does not produce any output on-screen. An example is Mez Breeze's pseudo-code language mezangelle, where "syntactical notation is taken from wildcards and regular expressions in programming languages and Unix command line interpreters forming an archetypical world" (Cramer 2005, 11). In this extract from Breeze's poem "_archi[ng]pelagos of d.sire_ (2004-02-08 17:27)" from *Human Readable Messages* square brackets are used to reveal words hidden in other words: "[eyes.of.smolder+graceless.ener[vation]gy]" (Breeze 2011, 76). When a reader reads this text aloud, they must choose whether to say "energy" or "enervation" or perhaps both in quick succession. The path that the reader chooses opens and closes off potential ways of performing the work. As McKenzie Wark writes: "Mez introduces the hypertext principle of multiplicity into the word itself. Rather than produce alternative trajectories through the text on the hypertext principle of 'choice', here they co-exist within the same textual space" (Wark 2001, 5). While it is challenging to perform, this work is nevertheless human-performable, as the title of Breeze's book *Human Readable Messages* suggests.

Type 6 (YYN) code poetry can be read aloud by humans, read by a computer, but the code itself does not perform any action. In other words, the computer recognizes the poem as valid, according to the rules of the programming language, but the program does not produce an output. An example is "Black Perl" (2003), an anonymous poem written in the Perl programming language. This poem is made of English words and a few other symbols, so it is easy for a person to read. The first line of the poem is "BEFOREHAND: close door, each window & exit; wait until time." When the Perl interpreter executes the code poem, it exits upon reaching the function "exit" in the first line.[2] The code poem terminates before it can produce an output. By contrast, a reader performing this poem aloud can read all twenty-four lines.

Type 7 (YYY) code poetry can be performed by humans and computers, and the code poems also perform an action. The code poems in our *./code --poetry* project fall into this type. This includes "compound_eye.rb," and another poem we will now turn our attention to, "chernobyl.rkt" (see Holden and Kerr 2016a). The code poem (the input) is written in the style of a report by a Communist Party official recording the fictional words of a resident of Chernobyl, or perhaps nearby Pripyat, after the disaster of 1986. The visual art that the code poem creates (the output), is a mutated version of the same report: in the animation the text scrolls faster and faster until it is replaced with non-semantic alphanumeric characters. This effect adds a glitch aesthetic to the artwork. "chernobyl.rkt" is written in the racket language,[3] and this is an aesthetic choice: each code poem in *./code --poetry* is written in a different programming language, and expresses the unique character of that language. Racket contains a lot of nested parentheses, which are used to define a hierarchical tree structure in the code.[4] "chernobyl.rkt" celebrates this feature of the language. The parentheses are markers of the meandering sub-clauses slurred by the drunken speaker of the poem. At the same time, the parentheses function to define the hierarchy of the text in the program, which determines how the text is spliced together in the output (Fig. 19).

This code poem is amenable to vocal performance for two reasons. First, the more unpronounceable symbols and keywords are hidden at the top of the poem. This encourages the human performer to focus on the body of the code poem, which contains readable English speech. Second, the parentheses help the performer, by acting like stage directions for a series of nested asides in a dramatic monologue. They suggest pauses and words to emphasize and also map the psy-

2 https://perldoc.perl.org/functions/exit.

3 https://racket-lang.org/.

4 https://docs.racket-lang.org/pict/Tree_Layout.html.

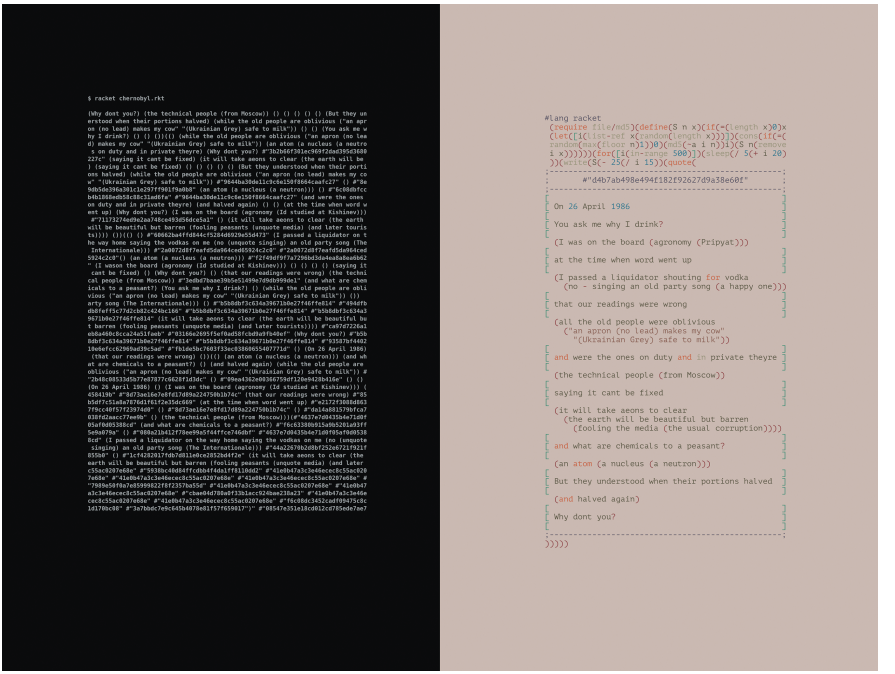

Fig. 19: Daniel Holden and Chris Kerr. “chernobyl.rkt.” *./code --poetry*. 2016a.

chology of the speaker’s voice. One could say that the character of the programming language becomes the character of its speaker.

The next two examples in this typology incorporate images and play with the visual aspects of performing code poetry. Hannes Bajohr presents Allison Parrish’s *Ahe Thd Yearidy Ti Isa* (2019), a visual-textual artwork made with multi-modal AI and points out that Parrish takes a neural network that is optimized for generating images, and feeds the neural network images of text. In Bajohr’s words, Parrish’s artwork “treats text *as* image, reverses the appropriate neural net architectures, and plays with the asemic effects” (Bajohr 2022, 228). Bajohr describes the images that the neural network produces as follows: “bitmaps of words are human-readable, but not machine-readable; they do not register as text” (Bajohr 2022, 228). These images are not programs: they do not do anything. Therefore, these images are examples of Type 5 (YNN) which moreover illustrate how different this work is to Breeze’s of the same type. In the final step of the process, Parrish feeds the images into character recognition software (cf. Bajohr 2022, 229), making the work machine-readable after all. This demonstrates that the definitional borders around the proposed categories are blurred.

Finally, an atypical work in our *./code --poetry* project uses images. "bark.png" (see Holden and Kerr 2016a) might look like the output of a process, but it is nevertheless in fact the input. The image is a code poem written in the esoteric programming language Piet, named after Piet Mondrian, the twentieth century Dutch artist.[5] "bark.png," a framed image of a bark-like texture, consists of a series of instructions that tell the Piet compiler to output a haiku, which is in turn about bark. "bark.png" is an example of Type 3 (NYY): its input cannot be read by humans, but it is computer readable, and it produces a poetic output. Later in this article, we will see how artists and poets have invented strategies for making visual artwork performable by the human voice. Nevertheless, "bark.png" has no human-readable information on its surface (Fig. 20).

Fig. 20: Daniel Holden and Chris Kerr. "bark.png." *./code --poetry*. 2016a.

The computer programmer Daniel Shiffman demonstrates that, in a digital context, alphanumeric characters and images are not as distinct as we might think: "[a] digital image is nothing more than data – numbers indicating variations of

5 https://www.dangermouse.net/esoteric/piet.html.

red, green, and blue at a particular location on a grid of pixels. Most of the time, we view these pixels as miniature rectangles sandwiched together on a computer screen." (Shiffman 2015, 301) Shiffman's words suggest another way of reading "bark.png" and its haiku output: this artwork is a transformation of image into text, yes, but it is also, simultaneously, alphanumeric data translated into alphanumeric data. These last two examples show that any framework of code poetry performance must account for the visual qualities of text, and the numerical and textual properties of images. In the next section, we will describe how code poems like "compound_eye.rb" and "chernobyl.rkt" can be made, or optimized for human and computer performance.

Poetic practice

In our book, *./code --poetry*, the creation of each code poem is performed via the merging of two texts: a conventional poem, and a minimal, functional program that produces the output seen on the left-hand page. This merging process is bidirectional. In an effort to retain the functionality of the program, we must constrain, restrict, and adjust the original poem to fit into a form dictated by the programming language, and the syntax and structure it allows. Meanwhile, the original poem serves as the main inspiration for the choice of adjustments we make to the program: the poem guides the process of transforming the program from a form that is purely functional to a form that is both poetic and functional. Indeed, in the final code poem, even the functional aspects are poetic as the functional elements are an intrinsic part of the code poem. The process of combination is afforded by the technology of the programming language and us as poets. James J. Gibson defines the concept of affordances as follows:

> The *affordances* of the environment are what it *offers* the animal, what it *provides* or *furnishes*, either for good or ill. The verb *to afford* is found in the dictionary, but the noun *affordance* is not. I have made it up. I mean by it something that refers to both the environment and the animal in a way that no existing term does. It implies the complementarity of the animal and the environment. (Gibson 2015 [1979], 119)

This concept provides another framework for viewing the process, in addition to that of Oulipian constraint: the poets and the technology are in an ecological, complementary relationship. This integration process is a unique and challenging form of constrained poetry that requires programming expertise. We must capture the essence of the poem while limiting ourselves to transformations that do not change the functionality of the program. While many transformations are possible, there are various tactics that can be used to integrate a poem into a pro-

gram without changing its behaviour, many of which are common across multiple programming languages. For example, arbitrary text can be inserted into programs as "comments" – additional text used to help readers of the code understand the intention of the programmer. The computer ignores these comments when it runs the program. In code poetry, "comments" can be used to directly insert segments of the original poem. While this technique is simple, it fails to account for the unique characteristics of the program, or the programming language it is written in.

Another similar technique that is common across many programming languages is to manipulate the names of "variables": named values that are used to refer to data stored in memory. The name given to variables can be changed to match words in the poem, or additional, unused variables can be declared and used in a way that does not change the program's functionality. Often, if the functional part of the program is too long or difficult to merge with the non-functional text, it can be minimized and hidden, for instance, by using single letters and symbols for variables, removing spaces and tabs, and compacting the program as much as possible. Then, the poem can be inserted into a non-functional section of the program, creating a code poem in two parts: a small functional program connected as an appendage to the larger, non-functional body. An example of this strategy can be seen in "chernobyl.rkt."

However, the goal of this merging process is to find more complex and satisfying techniques that involve some symbiosis between poem and program, preserving the signature elements of the poem, the program, and the programming language. Throughout this process, the layout of the code poem on the page can be adjusted. Because most programming languages ignore any whitespace when they functionally interpret the program, there is great freedom here. The text can be laid out on the page to create visual impressions and ASCII art, as "compound-_eye.rb" demonstrates. The whitespace, discounted by the programming language, nevertheless plays an active aesthetic role for the reader, like the role it plays in concrete poetry. Finally, the syntax highlighting can be adjusted, and colours, bold, italic, underlined formatting and highlighting can be applied to specific sections and keywords in the program in a way that foregrounds both the character of the programming language, and the poem. We can see this whole process as a form of optimization, where instead of execution speed or memory usage, a program is optimized for artistic value. In brief, this is a dual optimization with both human and computer performers in mind.

Visual scores for performance

We will now return to the human performance of code poetry. As we have noted above, "chernobyl.rkt" is relatively easy for a person to voice. But what if a reader wanted to perform the symbols and keywords from the top of "chernobyl.rkt," as well as the English words in the main body of the poem? "by_conspiracy_or_design.js" (see Holden and Kerr 2016a), another poem in *./code --poetry*, presents even greater challenges for vocal performance, because the English words are mixed with large numbers of alphanumeric symbols (Fig. 21).

Fig. 21: Daniel Holden and Chris Kerr. "by_conspiracy_or_design.js." *./code --poetry*. 2016a.

This code poem is written in JavaScript.[6] It features the voice of a ranting American conspiracy theorist who believes they have discovered the golden ratio in the vapour trails left by airplanes in the sky (or as the speaker has it "CHEMTRAILS"). The golden ratio (which is related to the golden section) is an irrational number

6 https://www.ecma-international.org/publications-and-standards/standards/ecma-262/.

said to have aesthetic properties.[7] The speaker appropriates feminist and Native American perspectives in a patronizing and shallow way, and mixes in pseudoscientific quantum physics concepts, all in the service of their illogical spell against supposed government conspiracy. The first part of the infinite golden ratio appears in the second line of the code poem. This number defines the shape of the visual output: a spiral that conforms to this ratio. It is made of ASCII characters, including plus, minus, equals, asterisk, ampersand signs and more. Karen Cham provides a visual culture context for the golden section:

> Any discussion of aesthetics and interactivity must first transgress the divide in modern western Art History between art and technology. Despite the fact that technical principles have always underpinned fine art production (rules of perspective, proportion and the golden section for example) photography, film, television and video are still marginalised in art-historical dialogues. (Cham 2009, 15)

To extend Cham's words from discussion to artforms, code poetry bridges the divide between literature and art on the one hand and science and technology on the other. "by_conspiracy_or_design.js" is well placed to explore the frictions at this divide: in this code poem, the golden ratio is a functional and mathematical rule as well as a spur for a kind of paranoid aesthetic thinking, where the art historian's desire to systematize and fit artwork to a scheme might have something in common with a conspiracy theorist's conviction that a pattern is everywhere they look. Nevertheless, the undeniable presence of the golden ratio in "by_conspiracy_or_design.js," might encourage the reader to consider parallels between this code poem and other visual art.

We will now turn to some examples of visual art being used as scores for sound poetry performance, to inform methods of vocally performing "by_conspiarcy_or_design.js." In this context, a score is the equivalent to printed notation of a musical or dance performance. The first example is the visual scores for sound poetry performance created by poets associated with the British Poetry Revival movement in the 1960s and 1970s. In one videoed collaborative performance with P. C. Fencott, Cobbing and Fencott use visual artworks as scores for elaborate, live performance, making animalistic sounds (see Johnson 1982). Bob Cobbing, a central figure in this movement, and Fencott talk about the relationship between the marks in the artwork and the dynamics of their performance, e.g., by equating the thickness of the mark with the volume of their voices. McCaffery and Nichol quote Sten Hanson, who asserts that Cobbing's sound poem scores have a significant visual component, cautioning that the sound element should not be privileged over the visual, or vice versa:

7 https://www.britannica.com/science/golden-ratio.

> The written versions of Cobbing's sound poems are not to be regarded merely as score for performance. They are poems in their own right and have important visual qualities which alone justify their existence as printed poetry. They can be appreciated without knowledge of their sound interpretations even though that knowledge would add a dimension to them. (McCaffery 1978, 102)

The second example of visual art being used as score for performance is the Fluxus art movement, also from the 1960s and 1970s. One important element of Fluxus was the production of scores for performance "events." Jackson Mac Low is an American poet associated with Fluxus who created poetic scores for performance. He described his "Asymmetries" series as "poems of which the words, punctuation, typography and spacing on the page are determined by chance operations" (Mac Low 1963). In his "BASIC METHOD" for reading Asymmetries, Mac Low defined what blank spaces, typography and punctuation indicated to the performer: "Blank spaces before, after and between words or parts of words, between lines of words, and before whole poems are rendered as silences equal in duration to the time it wd [sic!] take to read aloud the words printed anywhere above or below them" (Mac Low 1963). In this performance score, blank spaces become meaningful in the performance of the poem. This feature is analogous to some programming languages, for instance Python,[8] in which whitespace impacts how or whether programs run.

Both these examples offer possible methods for vocally performing visual art, with its marks and white spaces. Cobbing's methods in particular could be used to perform the seemingly unperformable "bark.png." Indeed, Cobbing and the poet-artist Paula Claire used patterned objects like bark as scores for performance:

> One of Cobbing's lasting contributions to text-sound activity is his revolutionising of what can constitute a 'text'. Cobbing (along with Paula Claire) has frequently abandoned the graphic imprint and received 'song signals' from natural objects: a cross-section of a cabbage, a stone, a piece of rope, the textured surface of bricks, cloth etc. (McCaffery and Nichol 1978, 14)

However, the vocal performance of visual art is perhaps necessarily idiosyncratic and unrepeatable across multiple live performances and performers, because its methods are so individual and impressionistic. That said, this art-historical context can encourage the viewer to see the works in *./code --poetry* in a new light. Almost all the code poems in this project take text as input and produce text-based visual artworks as output (with the exception of "bark.png," which reverses this operation to produce text from an image). Therefore, these code poems can be seen as ekphrastic machines, which enact the translation between visual and

8 https://docs.python.org/3/reference/lexical_analysis.html.

textual modes in a way that is repeatable, in contrast to Cobbing and Claire's virtuosic performances.

Performing alphanumeric symbols

The context for the performance of visual art that we have outlined above will be helpful to explore methods of vocally performing alphanumeric symbols, like those that appear in "by_conspiracy_or_design.js." Our exploration of how to perform this code poem is informed by approaches to performing visual art scores as well as alphanumeric symbols. We will describe three ways of voicing the (in common usage) peculiar signs in code poetry. One strategy is to pass over the alphanumeric symbols silently as the British-Canadian artist and writer, J. R. Carpenter, did when she was invited to perform her poem "Etheric Ocean":

> I devised a print script for two voices, to be read along with a projection of the digital work being navigated from left to right. The print script soon became annotated with stage directions scribbled in pencil. As I would be reading long lists of arguments copied and pasted from JavaScript arrays, I found it useful to ['retain the JavaScript syntax in the print script', 'to guide me through otherwise grammatically impossible sentences']. (Carpenter 2021)

Carpenter uses the square brackets and single quotation marks that define JavaScript arrays to inform her performance, but she does not voice these symbols. In a second method for performing symbols, their names are read out in full. Gary Barwin's *Servants of Dust: Shakespeare Sonnets 1–20* (2021) uses a voice synthesizer to read out only the punctuation marks from Shakespeare's sonnets, like "comma" and "colon," over music. The sonnet form is surfaced indirectly, through the proxy formal container of punctuation. A third method for reading alphanumeric symbols aloud is to assign sounds to them, rather than pronouncing their names. The British performance artist Nathan Walker created an artwork using JavaScript called "Sounding.js." Walker describes this piece as follows:

> My work, "Sounding.js", is both a live sound-poem and an online interactive website that enables the user to "activate" a recording of the performance by moving their cursor over the digital score. Composed using a complete and fully functional JavaScript, the code is presented as a phonic utterance; a vocal exploration of non-phonic programming language and the spatial and temporal possibilities of embodying the digital text in performance. (Walker 2013, 63)

Another example of this third method of performing alphanumeric symbols is more functional than artistic. Tavis Rudd, a software developer, created a speech-to-text interface in the Python programming language that allowed him to code

by voice, after experiencing a repetitive strain injury from typing (cf. Rudd 2013). Naming each symbol would take too long, so he defined a short vocal sound for each symbol. As well as assigning sounds to individual symbols, Rudd's system also accounts for the macroscopic structures of code. For example, for square brackets, he uses rhyming sounds to open ("lack") and close ("rack") the brackets.

Which of these three methods is appropriate for "by_conspiracy_or_design.js"? Given that the speaker of this code poem is a logically incoherent ranting conspiracy theorist, the third method (assigning sounds to symbols) seems the most appropriate, because these sounds make his speech more unintelligible. Different styles of performance suit different code poems. When the symbols are assigned to sounds in a similar way to Rudd's system, a script for the third, fourth and fifth lines of "by_conspiracy_or_design.js" might look like this:

> caps INFINITE naps eek spiral dot log par DIVINE tar slash par spiral dot PI slash 2 tar sem
> if par 0 tar brace function caps CHEMTRAILS naps par spirit tar brace return brack
> spiral dot cos par spirit tar star spiral dot exp par caps INFINITE naps star spirit tar com

Note that the capitalized words in this script were capitalized in the original code poem, inspired by the typography of conspiratorial websites. That said, capitalization does encourage the performer to shout these words. The speed of a vocal performance can be optimized by analogy to the way in which a program is optimized to perform quickly on a computer. Like a computer, a human performer can run slowly and quickly. Still, in vocal performance, there are also variables without direct computer analogues, like volume and pitch. A vocal performance could also move away from words like "naps" to other forms of human vocal articulation, like clicks.

Finally, a vocal performance of "by_conspiracy_or_design.js" could respond to the visual qualities of the code poem. The code poem is centre-aligned on the page, and the lines are of varying length. Inspired by Cobbing and adjusting his voice to the width of the marks on the page, a performer could say the shorter lines more quietly and the longer lines more loudly. A performer could also use syntax highlighting as part of their script, employing different colours as performance cues. Then, the syntax highlighting would be arranged with both vocal performance as well as visual balance in mind.

Conclusion: "optimized for performance"

In this article, we have demonstrated some ways in which code poetry can be optimized for human as well as computer performance. The techniques presented

are informed by the history of the performance of visual art and alphanumeric symbols. Furthermore, code poetry can be categorized in terms of whether it can be performed by humans, computers, and whether it in turn performs, or outputs something. We assert that the struggle of human and computer voice over a single artistic and functional space presents new opportunities for dramatic ironies. The speaker of the poem can do things that the program does not register, and the program can do things that the speaker does not know about. Like plays, code poems are made of characters, in the sense of letters, numbers, and other symbols. Code poems present an opportunity for the visual personality of each of these characters to be performed on multiple levels. As J. R. Carpenter writes, a "single missing or misplaced comma, bracket, or quotation mark will cause a program not to run" (Carpenter 2021). The attention to detail required by a programmer can train the reader and viewer of code poetry to pay a similar level of attention to individual glyphs. To focus in this way on linguistic signs is to see them as visual artefacts, and not only elements of text. Such a focus is encouraged in *./code --poetry* where ASCII characters are used as the elementary units of visual artwork. Consequently, the categories of text and image blur.

The dramatic character of code poetry can be seen clearly in the esoteric programming language Shakespeare, where programs appear to be play scripts (cf. Lang 2017). In "shakespeare.spl," a currently unpublished poem in our *./code --poetry* project that is written in this language, the following words form part of a program that outputs a list of Shakespeare's sonnets: "Thou art the sum of a handsome beautiful fair flower and a plum. Thou art the sum of the square of thyself and a golden rose." Similarly, J. R. Carpenter sees JavaScript as inherently performative and dramatic: "It is no coincidence that the word 'script' appears in the name 'JavaScript' – JavaScript is a procedural language. Like a script for live performance, JavaScript must be written and read in a particular order in order to be performed by the web browser" (Carpenter 2021). *./code --poetry* also explores characters in a different sense, namely the character or personalities of programming languages themselves. There is a ghostly voice behind each code poem, which is the voice of the language it is written in, whether that is the digressive and meandering quality of Racket, with its endless asides or the chaotic energy of JavaScript, freely filling the space on the page or screen like a conspiracy theorist. In this regard, code poetry is moreover an ideal medium for dramatic irony. In "by_conspiracy_or_design.js," the syntax "/native-American/" is highlighted pink. The computer knows something about the word "native" that the deluded speaker of the poem does not: it is a reserved word in JavaScript, i.e., it cannot be used to define a variable, function or label name. The casual racism of the speaker is complicated by the functioning of the technology of the code poem. Finally, this dramatic irony has a visual component. The pink syntax highlighting acts like a stage direction here, drawing atten-

tion to the double status of the word "native," and perhaps alluding to the identity of the code poem's speaker, who a reader might read as white. In code poetry, the visual, too, is optimized for performance.

References

Anonymous. "Black Perl updated for Perl 5." 2003, https://www.perlmonks.org/?node_id=237465 (January 15, 2023).

Aprile, Francesco. "laravel poems." *Code Poems. 2010–2019*. Minneapolis, MN: Post-Asemic Press, 2020. 17–20.

Bajohr, Hannes. "Algorithmic Empathy: Toward a Critique of Aesthetic AI." *Configurations* 30 (2022): 203–231.

Barwin, Gary. *Servants of Dust. Shakespeare Sonnets 1–20*. 2021, https://www.youtube.com/watch?v=PvQiuigBm7s (January 15, 2023).

Bhanoo, Sindya N. "Getting a Handle on Why 4 Equals Green." *The New York Times*. November 21, 2011, https://www.nytimes.com/2011/11/22/science/mapping-grapheme-color-synesthesia-in-the-brain.html (January 15, 2023).

Breeze, Mez. *Human Readable Messages*: *[Mezangelle 2003–2011]*. Vienna: Traumawein, 2011.

Carlson, Stephan C. "golden ratio." 2023, https://www.britannica.com/science/golden-ratio (January 15, 2023).

Carpenter, J. R. "Straight Quotes, Square Brackets: Page-Based Poetics Inflected with the Syntax and Grammar of Code Languages." *Hybrid* 7 (2021): n. pag. https://journals.openedition.org/hybrid/684 (January 15, 2023).

Cham, Karen. "Aesthetics and Interactive Art." *Digital Visual Culture. Theory and Practice*. Ed. Anna Bentkowska-Kafel, Trish Cashen, and Hazel Gardiner. Bristol and Chicago, IL: Intellect Books, 2009. 15–21.

Cox, Geoff, Alex McLean, and Adrian Ward. "Coding Praxis: Reconsidering the Aesthetics of Code." *read_me: Software Art and Cultures*. Ed. Olga Goriunova and Alexei Shulgin. Aarhus: Aarhus Univ. Press, 2004. 161–174.

Cox, Geoff, Alex McLean, and Adrian Ward. "The Aesthetics of Generative Code." *Hard Code*. Ed. Eugene Thacker. Boulder, CO: ALT-X, 2001. 22–34.

Cramer, Florian. *Words Made Flesh: Code, Culture, Imagination*. Rotterdam: Piet Zwart Institute, 2005.

"ECMA-262." Ecma International. June 2022, https://www.ecma-international.org/publications-and-standards/standards/ecma-262/ (October 31, 2022).

"exit." Perldoc Browser. https://perldoc.perl.org/functions/exit (January 15, 2023).

Finch, Peter. "Texture Poem for the Moons of Stars." *Typewriter Poems*. Cardiff: Second Aeon Publications, 1972. 34.

Gibson, James J. *The Ecological Approach to Visual Perception*. New York: Psychology Press, 2015 [1979].

Holden, Daniel, and Chris Kerr. *./code --poetry*. Authors' edition: CreateSpace self-publishing service, 2016a. n. pag.

Holden, Daniel, and Chris Kerr. *./code --poetry*. 2016b, https://code-poetry.com/ (January 15, 2023).

James, Brian. "Lockdown Dérive." *code::art* 1 (2020): 11.

Johnson, Josephine (Dir.). *Performing Concrete Poetry*. 1982, https://www.ubu.com/film/cobbing_fencott.html (January 15, 2023).

Lang, Mirco. “Shakespeare Programming Language.” 2017, https://www.dev-insider.de/shakespeare-programming-language-a-617430/ (October 31, 2022).
“Lexical analysis.” Python. 2023, https://docs.python.org/3/reference/lexical_analysis.html (January 15, 2023).
Mac Low, Jackson. “Methods for Reading Asymmetries.” *An Anthology of Chance Operations*. Ed. La Monte Young and Jackson Mac Low. New York: Authors’ edition, 1963. n. pag.
McCaffery, Steve, and bpNichol. “Biographies.” *Sound Poetry. A Catalogue*. Ed. Steve McCaffery and bpNichol. Toronto: Underwhich Editions, 1978. 91–111.
McCaffery, Steve. “Sound Poetry. A Survey.” *Sound Poetry. A Catalogue*. Ed. Steve McCaffery and bpNichol. Toronto: Underwhich Editions, 1978. 6–18.
Mitchell, W. J. T. “Showing Seeing: A Critique of Visual Culture.” *The Visual Culture Reader*. Ed. Nicholas Mirzoeff. London and New York: Routledge, 2002. 86–101.
Morgan-Mar, David. “DM’s Esoteric Programming Languages – Piet.” 2018, https://www.dangermouse.net/esoteric/piet.html (January 15, 2023).
“performance, n.” OED Online. September 2022, www.oed.com/view/Entry/140783 (January 15, 2023).
Petkova, Nataliya. “UNTITLED (LOVE).” *code {poems}*. Ed. Ishac Bertran. Barcelona: Impremta Badia, 2018. 21.
Racket. https://racket-lang.org/ (January 15, 2023).
Ruby Programming Language. https://www.ruby-lang.org/en/ (January 15, 2023).
Rudd, Tavis. “Using Python to Code by Voice.” 2013, https://www.youtube.com/watch?v=8SkdfdXWYaI (January 15, 2023).
Shiffman, Daniel. *Learning Processing. A Beginner’s Guide to Programming Images, Animation, and Interaction*. Amsterdam: Elsevier, 2015.
Sondheim, Alan. “Introduction: Codework.” *American Book Review* 22.6 (2001): 1, 4.
Walker, Nathan. “Transitional Materialities and the Performance of JavaScript.” *Performance Research* 18.5 (2013): 63–68.
Wark, McKenzie. “Essay: Codework.” *American Book Review* 22.6 (2001): 1, 5.