Proceedings of the International Conference on Live Coding (ICLC) 2025, Barcelona, May 27th - 31st

Screamer: A High-Level Language for Live Coding Ray Marchers

Charlie Roberts
Worcester Polytechnic Institute
charlie@charlie-roberts.com

ABSTRACT

I describe *Screamer*, a high-level language for live coding performances using ray marching. My motivation for Screamer was enabling faster iteration of three-dimensional scenes, especially in comparison with my prior work, the live coding environment marching.js. I discuss the underlying rendering engine of Screamer and its language features, and compare its notation with other live coding systems to help explain my design decisions. I conclude by discussing performances given using Screamer and future research directions.

1 Introduction

Ville-Matias Heikkilä (aka viznut) describes *Core Demoscene Activity* as the "discovery of new techniques to be used in demo art", supporting new audiovisual outputs that are either "…not seen on your platform before, or something not seen anywhere before" (Heikkilä 2010). He further describes four circles of creative demo activity, based on their distance from this fundamental core. Although the heterogeneous goals of the live coding don't necessarily align with that of the demoscene (Borzyskowski 1996), I argue that the bulk of live coding activity can be found in Heikkilä's second circle, *Application-level programming*, which includes practices such as "content creation via programming" and "demo composition".

Ray marching is a rendering technique for non-traditional geometries that is commonly used in the demoscene; think of liquid, recursive, and combinatorial shapes that smoothly evolve over time. While live coding ray marchers is a common practice at demoscene events, it is less common in the live coding community despite notable practitioner-advocates such as Char Stiles and Sol Sarratea. One reason this might be the case is due to the low-levels of abstraction often used when programming ray marchers. Ray marchers are coded in *fragment shaders*, which are small programs that run on the graphics card of a computer and generate an output color value for every pixel they are applied to. Although working at the level of individual pixels in a fragment shader opens up the possibility of discovery and new algorithms (Lawson 2015), many live coding practitioners instead prefer to push against constraints in live-coding environments with higher-levels of abstraction. For visual live coding, popular environments that arguably fit this model include Hydra (Jack, n.d.), p5.js (McCarthy, Reas, and Fry 2015), and LiveCodeLab (Della Casa and John 2014).

My first live coding environment for ray marching, *marching.js* (Roberts 2019), has been used by a variety of performers since its introduction, and contains both abstractions bringing ray marching into the second circle of creative demo activity as well as low-level features that make it arguably suitable for first circle activity. However, in my performance practice I often found it too verbose to use in my preferred style of "blank slate" live coding. While using JavaScript as the language for marching.js helped ensure that knowledge transfers from other systems—such as Hydra and p5.js—JavaScript contains constraints that make it difficult to use for the type of rapid world construction I am interested in. To work around these constraints, described further in Section 3, I created a new domain-specific language for live coding ray marchers, *Screamer*, that uses marching.js as its underlying graphics engine, and an accompanying browser-based environment for live coding. The motivation for Screamer was to abandon the first-circle features of marching.js in favor of greater fluidity and expressiveness.

In this paper, I argue that Screamer is better-suited for second-circle live-coding activities than marching.js. I begin with a brief discussion of the underlying graphics engine used by Screamer, and then discuss its core language features with comparison to other systems as appropriate. I conclude by discussing performances I've given with Screamer, and areas for future improvement.

2 Underlying Graphics Engine

We previously described the core technical capabilities of the marching.js rendering engine (Roberts 2019) and additional functionality for texturing 3D geometries (Roberts 2020). At its core, marching.js is a JavaScript graphics library that

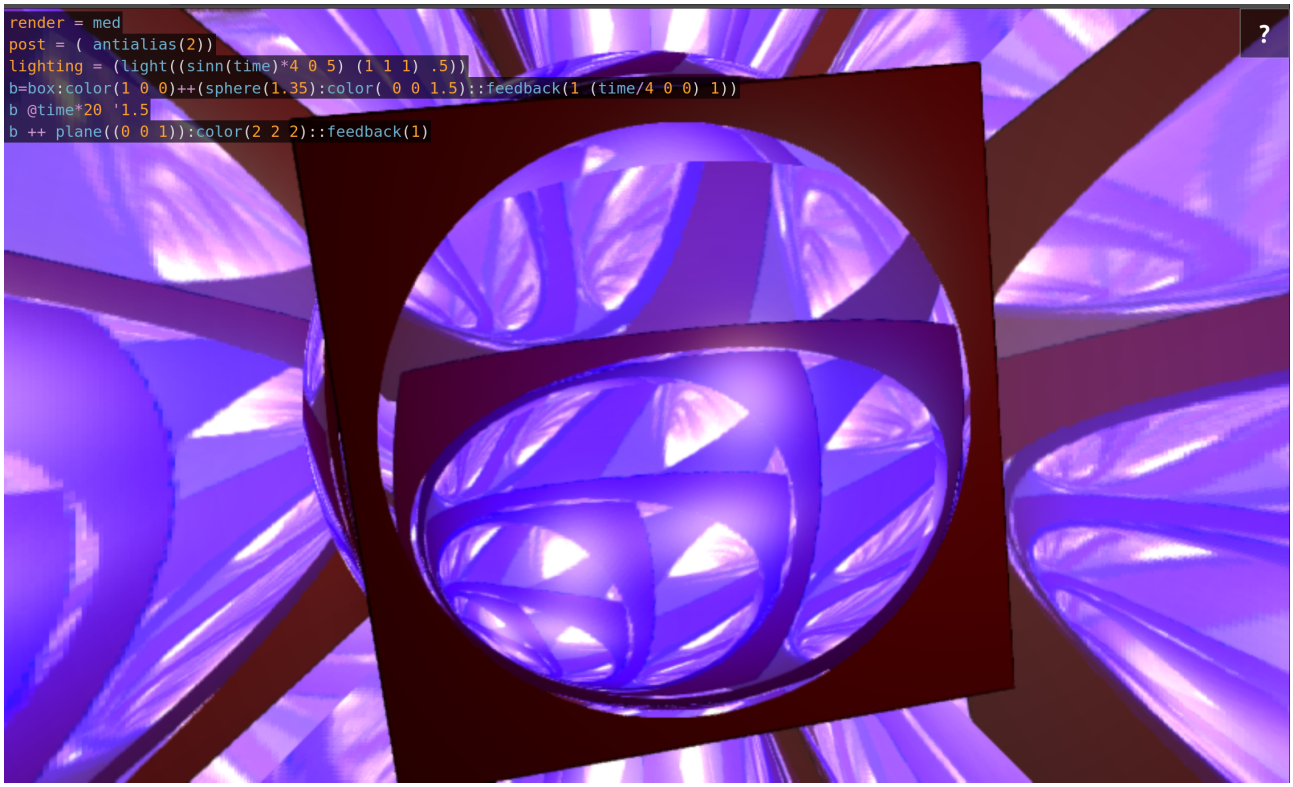

Figure 1: *Feedback applied as a texture to the surface of a sphere and a back plane in Screamer.*

runs in the browser, with an accompanying browser-based REPL for live coding. Users create a high-level graph of 3D geometries, transformed by a variety of available operations; this graph is subsequently compiled to a fragment shader that fills the background of the editor window. In addition to canonical 3D forms such as spheres, rings, and boxes, the geometries in marching.js include recursive / fractal shapes that rely on ray marching in order to dynamically render them in realtime. Procedural textures can easily be applied to geometries, and their properties can also be live coded; this includes textures created using other systems such as Hydra and p5.js. Marching.js also provides common ray marching operations to manipulate *space*, and geometry positioning / orientation in space, such as repetition and mirroring.

More recent features that have not yet been reported include:

- A custom post-processing library including effects such as depth of field, volumetric lighting, antialiasing, motion blur, and more.
- An alternative, voxel-based rendering algorithm for quantizing visual worlds in the style of games like Minecraft.
- Support for using the rendered output of the scene as a texture for individual objects (feedback), as shown in Figure 1.

3 Design Considerations for Screamer

My primary motivation for designing the Screamer language was to enable fast iteration, specifically faster than what is possible in marching.js. I used three language features to realize this, and describe each in more detail later in this section:

1. Use terse operators to combine and manipulate geometries (and in at least one case, define control flow), instead of calls to functions with longer, human-readable names and extra syntax requirements. “Stack” related operators together, making it simple to change between similar operations.
2. Support the use of reactive expressions affording the declarative use of time, audio signal analysis, and cursor position to drive properties, instead of relying on callbacks or other more complex logic.
3. Simple abstractions for grouping geometries and enabling reuse.

I wrote the grammar for Screamer using the Parsing Expression Grammar (PEG) formalism (Ford 2004), using the peggy.js library.

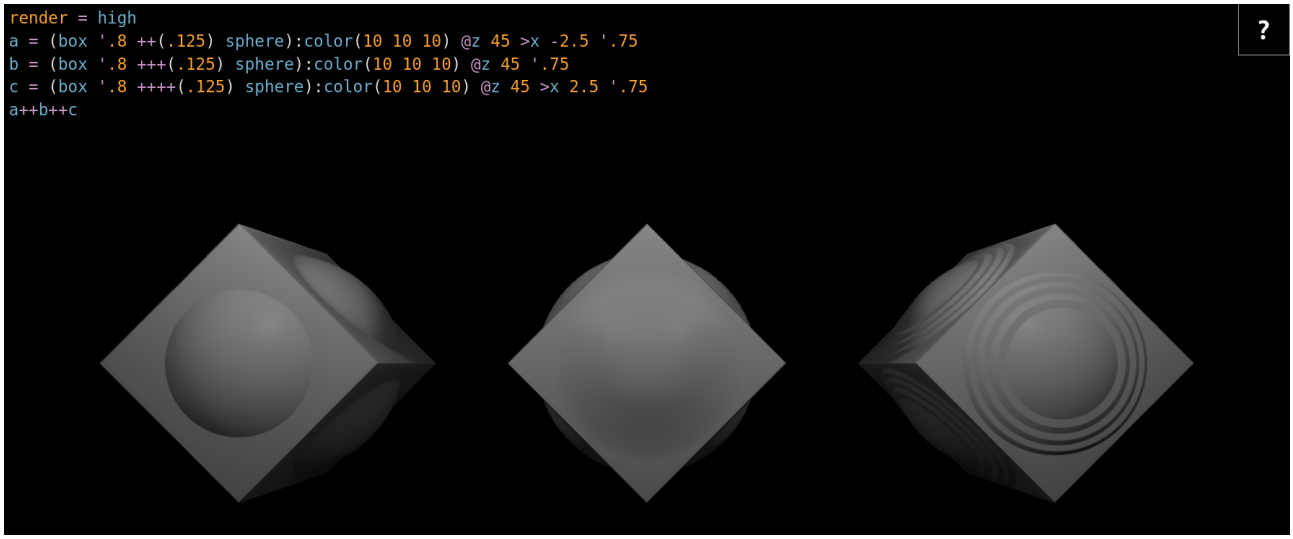

Figure 2: *Three different addition combinators. From left to right: ++(union) +++(smooth union) and ++++(stairs union).*

3.1 Terse & Stackable Operators

Marching.js features relatively readable names for common operations. For example, `SmoothUnion` combines two geometries for a smooth transition; `Repeat` repeats an object over space. In Screamer, I abandoned many such names in favor of operators that are 1–4 characters in length; this would have been impossible in a language like JavaScript which does not permit operator overloading. While I considered mapping all functions to such operators, I chose to leave geometry names and the names of textures and colors as human readable, while common operations like translation, rotation, repetition, mirroring etc. were applied with operators. By keeping common geometry and texture names in code, Screamer is designed so that some code is understandable to audiences unfamiliar with it, while also decreasing the number of operators performers need to internalize. I also grouped operators together whenever possible via duplication instead of providing unique operators for every instance. For example, the symbol for the *difference* operator, to subtract one geometry from another, is `--`.[1] The symbol for *smooth difference*, which subtracts a geometry from another while smoothing the transition, is `---`. And the symbol for *stairs difference*, which turns the transition area into a series of steps, is `----`. This both thematically groups the operators and makes it simple to rapidly switch between the different variations; these variations can be seen in Figure 2.

For many of the operators we tried to use characters that could visually represent the underlying operation or provide a metaphor. The # character visually consists of a 3x3 grid; we use this as the repetition operator in Screamer. The `@` character at least partially depicts a spiral, so we chose it for rotation. Translation is performed using the > operator. However, it is not clear that these correspondences are immediately clear to other users, and there are additional concerns regarding the availability of some characters on non-English keyboards. After soliciting feedback from users we changed the scale operator from ^ to '; the ^ was deemed relatively difficult to access on non-English keyboards. We chose the apostrophe for scale as it is simple to create on all keyboards and also doesn't require using the shift modifier. An example of these operators in use on a fractal is provided below:

```
// scale a fractal 2x, move it .75 units, mirror it, and rotate it 45 degrees
mandelbulb '2 > .75 | @45
```

Operations that transform geometries or the space in which geometries are rendered can be selectively applied to individual dimensions by listing them after the operator; if none are listed it is assumed that the operation is applied uniformly across all three axes. The one exception to this is scaling, as non-uniform scaling of distance functions (the function often used to represent geometries in ray marchers) is difficult to achieve without glitches.

```
//rotate a box on the z-axis 45 degrees, and repeat it across the xy axes every three units
box @z 45 #xy 3
```

[1] The single - operator is for arithmetic subtraction.

3.2 Reactive Expressions

Math expressions in Screamer are tokenized into individual functions and then composed together. Time-varying values (e.g. `time`, `mousex`, `low`, `high` etc.) are treated as literals, enabling time-varying functions to be easily composed and assigned to visual properties. This mimics the idea of *behaviors* found in the functional reactive paradigm proposed for animation by Elliott and Hudak (1997), however, their concept of discrete *events* is not found within Screamer.

To enable reactive expressions in Screamer, we changed marching.js to support assigning functions to properties; this automatically adds functions to a global callback executed once per frame of video, similar to functionality found in Hydra. The code example below shows one example of how the design of Screamer also led to improvements in its underlying engine for live coding.

```
// in Screamer
sphere( .5 + sin(time) * .5 )

// previous versions of marching.js
march( s = Sphere() ).render()
onframe = time => s.radius = .5 + sin(time) * .5

// in the current version of marching.js
march( Sphere( time => .5 + sin(time) * .5 ) ).render()
```

In the example above, we see how Screamer is tersest for this type of expression; in my opinion it is also conceptually the most elegant notation, as there is no need to explicitly think about creating functions that are assigned. The second marching.js example shows how live coders can now avoid assigning the sphere to a variable, and potentially also avoid needing to define a global callback function. It is possibly more difficult to read such functions inline in geometry constructors, however, at the very least this gives users the option to choose which notation they prefer.

3.3 Abstraction

Green and Petre (1996) define three levels of abstraction for programming environments: *abstraction hating*, *abstraction tolerant*, and *abstraction hungry*. Screamer is probably best classified as *abstraction hating*, as it contains little ability to define abstractions. The exceptions to this are the ability to capture geometric nodes as variables, and the ability to group nodes using parentheses for shared operations. For example:

```
// create variable eyes storing a mirrored sphere
eyes = sphere(.1) >x .25 |
// create variable mouth
mouth = cylinder((.1 .5)) @z 90
// group eyes and mouth, then scale and color togehter
(eyes >y .5 ++ mouth) '3 :color( 1 0 0 )

// alternative in one line
((sphere(.1) >x .25 |) >y .5 ++ cylinder((.1 .5)) @z 90) :color(1 0 0) '3
```

Most visual live coding systems are instead abstraction tolerant, with multiple mechanisms for defining reusable functions that can provide any level of desired abstraction. In Hydra it is trivial to define presets for oscillators that can easily be reused; the example below shows three different possibilities, each enabling a different end-user syntax.

```
// custom function returning an parameterized oscillator
fastosc = _ => osc(20,1,5)
fastosc().out() // render to screen

// member function
osc.fast = _ => osc(20,1,5)
osc.fast().out() // render to screen
```

```
// create custom property
Object.defineProperty( osc, 'fast', {
  get() { return osc( 20,1,5 ) },
  set() {}
})
osc.fast.out() // render to screen
```

The variety of approaches above, each with different syntax, points to the flexibility that using a general-purpose language like JavaScript can provide. Users can effectively create their own domain-specific languages on top of Hydra, enabling a degree of customization that Screamer does not enable. While some of the above syntax would be difficult to add to Screamer, we do plan to improve the language by adding the ability to define custom functions that return geometries or operations.

3.4 Loops

Screamer contains one abstraction for loops. It is quite common in ray marching to explore "folding", which works roughly as follows:

1. Begin with a basic geometry, like a box or a ring
2. Apply a series of transformations to the geometry, such as scaling, translation, and rotation
3. Mirror or repeat the geometry to create copies of it
4. Continue to apply steps 2 and 3, gradually increasing the complexity of the output

Writing out each loop iteration individually can be verbose. As an alternative, Screamer's sole operation for control flow is the *loop* operator, `[]`. This operator was created after discussion with potential users of Screamer in Discord, and makes it much faster to iterate scenes employing folding. For example, in Figure 3 we show an Octahedron that is translated, rotated, and mirrored eight times with the following single line of code:

```
// syntax: [initialGeometry numberOfIterations transformations]
[octahedron(.125) 8 >(.25,.1,.05) @@(45,cos(i+time/3),0,1) | ]
```

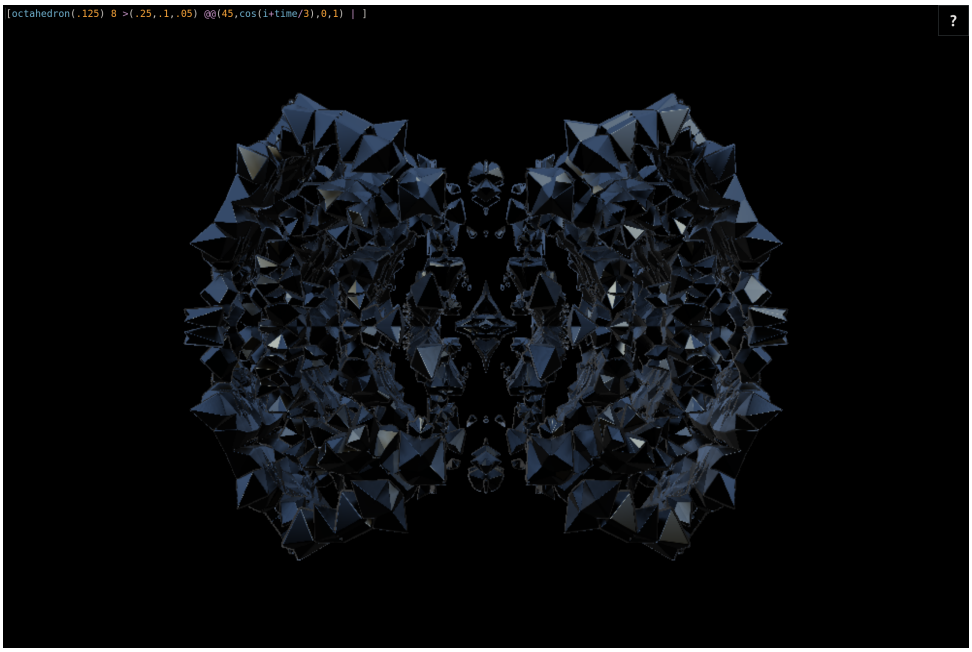

Figure 3: An octahedron that has been mirrored eight times, with a variety of transformations applied on each iteration

4 Discussion

Shortly after I released Screamer I did a four-minute live coding screencast with it, set to music by the artist digital selves. I used many of the features described in this paper, including combing geometries, manipulating post-processing effects, and applying reactive expressions. I also focused a significant portion of the performance on folding spheres using the `[]` loop operator. It was difficult to predict how the transformations I applied in the loop would would translate to the final output; this is a "problem" that can most likely be improved through more practice with the system. Four frames taken from this screencast are shown in Figure 4.

My biggest concern with Screamer at the time of the performance had less to do with the language than the underlying rendering engine. When a Screamer program is run, that program has to be parsed and the compiled into a GLSL shader. References to memory locations on the GPU then have to be obtained in order to manipulate the running shader. Unfortunately this process takes too much time, resulting in frozen output for several frames of video and an additional potential "blanking" of the output that causes a brief flash. This made me execute code changes much more infrequently than I would have preferred if the recompilation and output process was instantaneous. I have since optimized the compilation process to improve this, however, more work remains to be done.

My first live performance with Screamer was at an Algorave held in October of 2024, alongside musical live coder Ian Hattwick. While I would personally classify the performance as successful, there were a couple of problems worth mentioning. The first was, again, the general underlying performance of the rendering engine. I had borrowed a colleague's laptop (2021 Macbook Air) to perform with, as it had more powerful integrated graphics than my Linux laptop with Iris Xe integrated graphics. However, while the Macbook was more powerful in my short tests with it, the lack of an active cooling system (fan) meant that performance wound up degrading quickly and significantly. I ended up having to abandon many of the ideas I was planning on exploring in the performance in favor of simpler material. The other problem I encountered was poor error reporting in Screamer's interface. I frequently executed code and was unable to perceive any changes occurring, but no errors were reported. At the end of the performance I opened up the browser's development console to see what happened, and was greeted with a sea of error messages that were invisible to me during the performance; on a positive note the audience seemed to enjoy seeing the bugs revealed and they were greeted with a lot of laughter.

Across both performances, I found that core language features of Screamer—such as the use of terse operators and reactive expressions—yielded a much faster iterative process, and one that left me less concerned about syntax and more focused on the visual composition I was creating. Because of this, I believe that Screamer is better than marching.js for the second-circle activities (primarily live coding performance) as I intended. At this early stage of development and use I haven't missed the first-circle features that marching.js provides (for example, creating new geometry definitions, post-processing effects, or procedural textures). But it is certainly possible that with more time to fully explore the current features of Screamer, I'll find myself wanting to create new ones. Many of these will require dropping back into marching.js to implement, and then adding language features to Screamer to expose them for use. I am hopeful that this will be a rewarding development process in the years to come, leading to continued development of both projects. In addition, while initial informal feedback from users has been positive, Screamer is still a very young project, and I look forward to obtaining additional feedback to help guide its development.

Screamer is free and open-source software. The code repository is available at https://github.com/charlieroberts/screamer and the online playground resides at https://charlieroberts.github.io/screamer/playground.

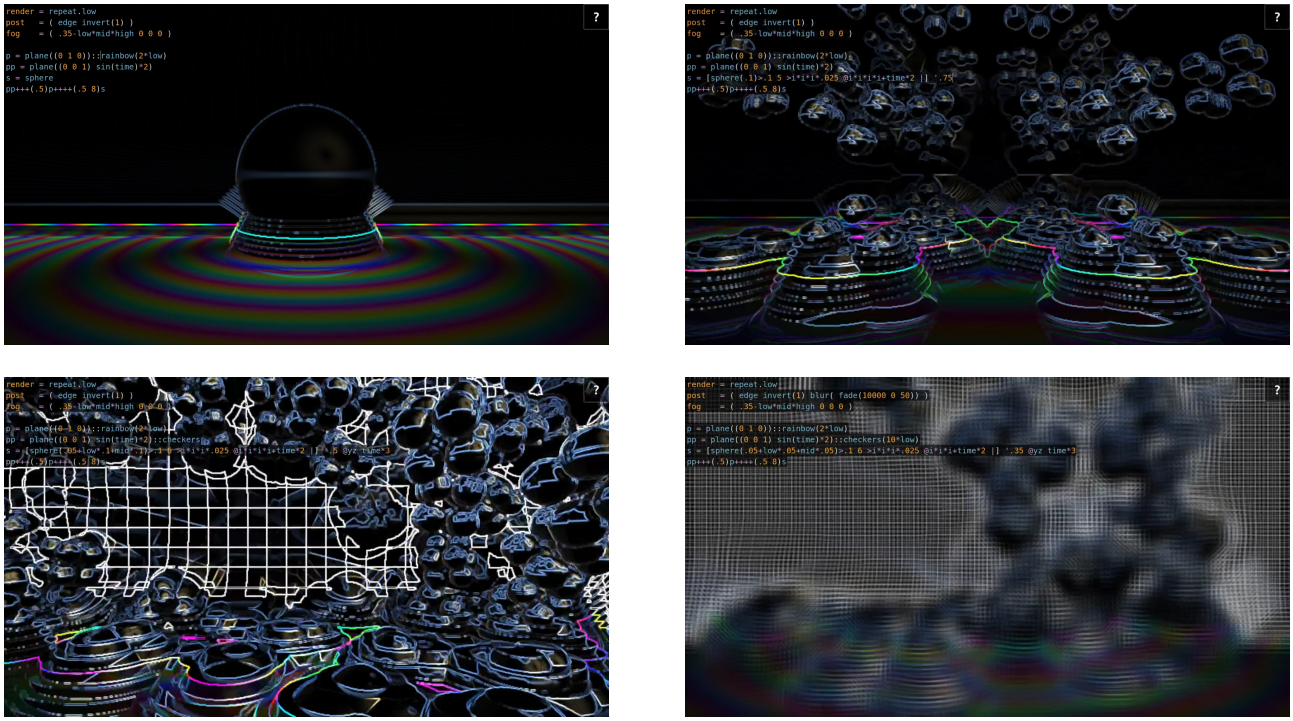

Figure 4: Four images from a four-minute performance using Screamer

References

Borzyskowski, George. 1996. “THE HACKER DEMO SCENE AND IT’s CULTURAL ARTEFACTS.” In *Cybermind Conference 1996*, 1–23.

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.

Elliott, Conal, and Paul Hudak. 1997. “Functional Reactive Animation.” In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, 263–73.

Ford, Bryan. 2004. “Parsing Expression Grammars: A Recognition-Based Syntactic Foundation.” In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 111–22.

Green, Thomas R. G., and Marian Petre. 1996. “Usability Analysis of Visual Programming Environments: A ‘Cognitive Dimensions’ Framework.” *Journal of Visual Languages & Computing* 7 (2): 131–74.

Heikkilä, Ville-Matias. 2010. “The Future of Demo Art: The Demoscene in the 2010s.” http://www.pelulamu.net/countercomplex/the/_future/_of/_demo/_art.

Jack, Olivia. n.d. “< hydra > Live Coding Video Synth.” https://hydra.ojack.xyz.

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

McCarthy, Lauren, Casey Reas, and Ben Fry. 2015. *Getting Started with P5. Js: Making Interactive Graphics in JavaScript and Processing*. Maker Media, Inc.

Roberts, Charles. 2019. “Live Coding Ray Marchers with Marching.js.” In *Proceedings of the International Conference on Live Coding (ICLC)*.

———. 2020. “Live Coding Procedural Textures of Implicit Surfaces.” In *Proceedings of the International Conference on Live Coding*, 132–44.