Live Coding Ray Marchers with Marching.js

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

ABSTRACT

We describe a new library, marching.js, designed for ray marching and constructive solid geometry in the browser. The API was designed to be easy to use for live coding while enabling volumetric rendering techniques—inspired by the demoscene—that would be difficult to achieve via the rasterization of geometries or by writing fragment shaders from scratch. The library and an associated live coding environment enable programmers to easily experiment with these techniques without needing to know details about the underlying ray marching or lighting algorithms.

1 Introduction

Programming *fragment shaders* is one method for creating visuals in live coding performances used by multiple live-coding environments. Fragment shaders are designed specifically to run on the graphics programming unit (GPU) of a computer and run once for each pixel displayed; the GPUs they run on are highly parallelized so that multiple instances of the program can be running concurrently. Visual performers programming fragment shaders often create abstract patterns and employ audio analysis to synchronize properties of these patterns to music. While these patterns are primarily two-dimensional in content, in the competitive *demoscene* community live programming of fragment shaders often takes a very different form. In demoscene competitions, *ray marching*, a technique for mathematically calculating three-dimensional scenes, is used to create complex virtual worlds over the course of a performance/competition.

Creating a raymarcher live, while competing under time constraints, is a difficult programming challenge that requires knowledge both of shader programming techniques and the mathematics to create formulae describing three-dimensional geometries. Our research provides a high-level library and associated live-coding environment, *marching.js*, that attempts to abstract away many of the complexities of writing raymarchers, making them accessible to a broader audience of live coders and other programmers interested in exploring constructive solid geometry and volumetric rendering. This paper begins by discussing how raymarchers work, moves to discussing the visuals programmed in the live coding and demoscene communities, and ends with technical and aesthetic discussions of marching.js.

2 Background and Terminology

For readers who are not familiar with volume rendering techniques, a brief introduction is provided. We then describe various live-coding environments that have influenced the research presented here, and provide an overview of the demoscene community.

2.1 Raymarching and Rasterization

Many live-coding environments use *rasterization* to convert 3D geometries to a 2D bitmap suitable for display on a screen. Geometries are first created on the CPU of a computer, and then subdivided into triangles. The coordinates of the three vertices of each triangle are then sent from the CPU to the GPU of the computer, where a rasterization algorithm then calculates how these triangles are projected onto the final 2D bitmap. The most important reason for this rendering technique is its efficiency, which enables realtime rendering of 3D graphics on hardware from well over twenty years ago. However, by defining geometries in terms of triangles and their associated vertices, we lose flexibility in how we can manipulate geometries. Combining multiple geometries together, repeating a geometry over a given area, or subtracting one geometry from another all become relatively difficult operations that are difficult if not impossible to perform on the GPU alone.

Instead of using vertices and triangles, ray marchers—first explored in the context of medical imaging by Tuy and Tuy (1984)— use mathematical formulae to describe geometries. The input to these formulae can then be flexibly manipulated to create shapes that would be very difficult to achieve using rasterization. To render these formulae to a 2D plane, we cast *rays* (lines) from the position of a of a virtual camera through a *viewport* defining the visible part of virtual world (often roughly corresponding to a GUI window), and use the formulae to determine whether or not any objects in the scene are struck by each ray. If so, that object can be seen by the camera and should be rendered. A ray is generated for every pixel in the viewport; the renderer "marches" through the pixels on the screen row by row, column by column, determining the color for each pixel based on any object the ray strikes.

The most important detail from the above description of ray marching is that the formulae can be easily manipulated and combined to create complex forms that can be smooth and liquid in appearance. Another popular rendering technique, *ray tracing*, is conceptually similar to ray marching but with important differences. In raymarching, mathematical formulae are used to describe geometries; geometric combinators can merge these geometries in a variety of ways that generate a single equation capable of defining an entire three-dimensional scene. In comparison, typical raytracing engines perform a series of tests to see if each individual geometry in the scene is struck by each ray; this involves multiple equations and tests. If a ray does strike a geometry in a raytracing engine, there are often a number of new rays that are subsequently generated in order to determine reflections, refractions, shadows, and occlusions that might affect the final color of the current pixel being evaluated. These extra rays add significant computational expense, often making raytracing unfeasible for realtime use depending on the configuration of the rendering engine. It is only in the last year that we are beginning to see realtime support for raytracing complex virtual scenes added to GPUs and low-level graphics programming APIs. While raytracing is required to correctly model many physical behaviors, some such behaviors (like shadows and occlusion) can also be approximated using simple tricks in raymarching engines.

To summarize, in typical 3D engines geometries are subdivided into triangles and vertices that are then rendered to a 2D bitmap using a rasterization algorithm. Raymarching performs raycasting for every single pixel on a screen using a (often single) mathematical formula that is built from a variety of geometric combinators; the inputs to these formulae can be flexibly manipulated to create forms that are difficult to achieve with triangles and vertices. Raytracing is similar to raymarching, but enables more realistic rendering (with corresponding computational expense) by enabling rays to reflect, refract, and strike many objects in a virtual scene.

2.2 Live Coding Visuals

There is an interesting array of environments developed for live coding visuals. Here we concentrate on more recent environments for live coding performance, but note that there is a long history of REPLs and interactive graphics programming, including languages that go back over forty years, such as Logo (Papert 1980) and Grass (DeFanti 1976).

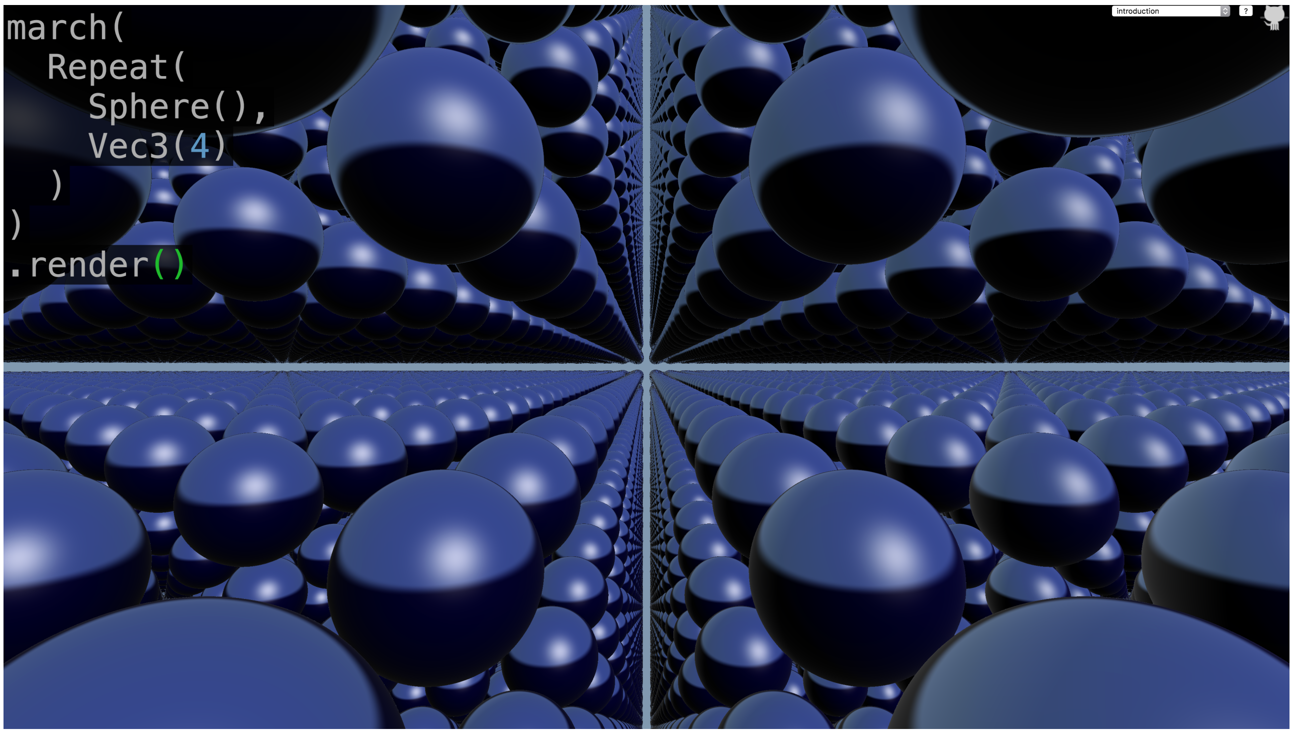
More recent live coding environments broadly fall into two camps. The first consists of environments that use the previously described rasterization algorithm described previously. Many live coding environments for visuals, such as Fluxus (Griffiths 2014), LiveCodeLab (Della Casa and John 2014), and Cyril (https://medium.com/cyril-live-coding), use rasterization for rendering. Additionally, many graphics engines for audiovisual systems, such as LuaAV (Wakefield, Smith, and Roberts 2010) and Gibber (Roberts et al. 2014), also (primarily) adopt this approach.

The second camp consists of environments that are designed to edit fragment shaders programs, which are written to determine the color of each individual pixel often using pixel coordinates, time, and audio analysis as inputs that create variety over space and time. Live coding fullscreen fragment shaders is a more recent development in live coding performance, but now includes environments such as The Dark Side of the Force (Lawson and Smith 2018), KodeLife ("hexler.net | KodeLife," n.d.), and VEDA (Amagi, n.d.).

Perhaps the most conceptually similar environment to marching.js is Hydra, by Olivia Jack ("Hydra," n.d.). Hydra takes high-level descriptions of algorithms approximating analog video synthesis techniques, and uses these descriptions to write and compile a fullscreen fragment shader. Marching.js works in a very similar way, but instead of modeling analog video synthesis techniques, marching.js uses ray marching and constructive solid geometry techniques for inspiration. However, the overall concept is still very similar: a high-level description of a scene is given in JavaScript, and marching.js then writes and compiles a corresponding fragment shader.

2.3 The Demoscene and Ray Marchers

The demoscene has a long cultural history in audiovisual programming that goes back over thirty years (Carlsson 2009). One original hallmark of the demoscene is working within the limited constraints of computing platforms to create realtime graphics and visuals (Burger, Paulovic, and Hasan 2002). However, the computing power available today has removed many such constraints; we argue this has lead to the development of artificial contraints in the demoscene competitions, such as creating a *demo* (realtime audiovisual scene) within defined memory constraints (for some JavaScript examples of



```
march(
  Repeat(
    Sphere(),
    Vec3(4)
  )
)
.render()
```

Figure 1: *Infinite spheres, repeating every four units on all three axes, programmed and rendered in marching.js.*

this, see http://js1k.com). Another constraint that has been added to some demoscene competitions is realtime programming, which brings the community closer to live coding. Programmers at demoscene competitions often live program fullscreen fragment shaders. These competitions cam take a ladder format, where programmers are judged against one another according to the complexity of the scenes they create within a specified time limit. Because of the visual complexity and flexibility that raymarchers provide, they have become the preferred technique to use in many of these events. However, since demoscene competitions typically require coding from a blank slate, these raymarchers are coded from scratch, which means that it might take five to ten minutes before any significant visuals are seen; arguably this level of abstraction is inappropriate for typical live coding performances. While initial development might progress slowly, the virtual worlds that are eventually created in demoscene events often grow to be fantastically complex.

Many raymarching techniques were originally discovered / evangelized by demoscene programmers and other graphics experts; it is our hope that marching.js will make such techniques easily accessible to the live coding community for experimentation.

3 Marching.js

Marching.js is a JavaScript library with an associated live coding environment that runs in the browser. Typically, live coding in marching.js is a two-stage process. In the first stage, the performer describes a 3D scene or world they would like to render; this description can be almost entirely declarative. The second stage consists of applying time-varying transformations to the scene, which might use time, audio analysis, or user interaction as input parameters.

3.1 Defining the world / scene

The scene description is written in JavaScript with a terse API. For example, in Figure 1 we show an infinite field of repeating spheres. In raymarching, repeating geometries requires a single modulo operation in fragment shaders, making it much less expensive to perform than in raster-based engines. The `Sphere` constructor creates a *primitive* in marching.js, while the `Repeat` constructor generates a *domain transformation* that is applied to the primitive. Other domain transformations include polar repetition, translation, rotation, and scaling. There are a wide variety of primitives included with marching.js; there are also more complex forms that can be rendered, as shown later in this section.

In addition to defining the objects in our scene and various transformations that can be applied to them, in the first stage we also define various other rendering characteristics, such as lighting and camera position. Figure 2 displays a scene similar to Figure 1 with custom lighting, fog, and camera positioning specified:

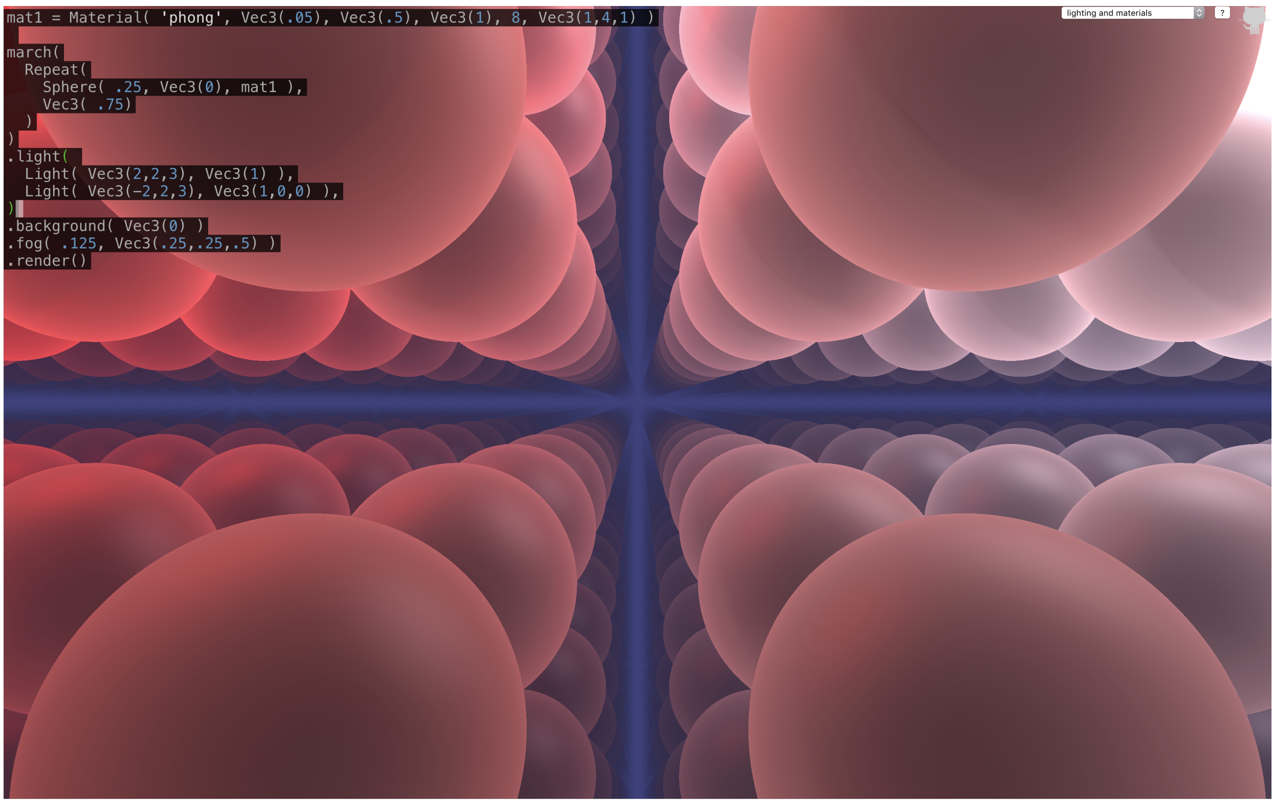

```
mat1 = Material( 'phong', Vec3(.05), Vec3(.5), Vec3(1), 8, Vec3(1,4,1) )

march(
  Repeat(
    Sphere( .25, Vec3(0), mat1 ),
    Vec3( .75)
  )
)
.light(
  Light( Vec3(2,2,3), Vec3(1) ),
  Light( Vec3(-2,2,3), Vec3(1,0,0) ),
)
.background( Vec3(0) )
.fog( .125, Vec3(.25,.25,.5) )
.render()
```

Figure 2: *Infinite spheres, with fog, custom lights, and a custom material.*

The call to the `.render()` method of the raymarcher is particularly important. When no arguments are passed to `.render()`, the renderer creates a high-resolution static render of the scene; the raymarching engine is set to maximum quality. Raymarching is computationally expensive, and extending the raymarching field to distances far from the camera can quickly bring realtime rendering to a standstill. By passing a relatively low *quality* argument (depending on the graphics card being used), and then passing a value of `true` to the *dynamic* rendering flag, the performer can use settings more appropriate for live coding performance.

In general, during the course of performances changing the world / scene will occur less frequently than making changes to shader uniforms, described below.

3.2 Time-varying changes

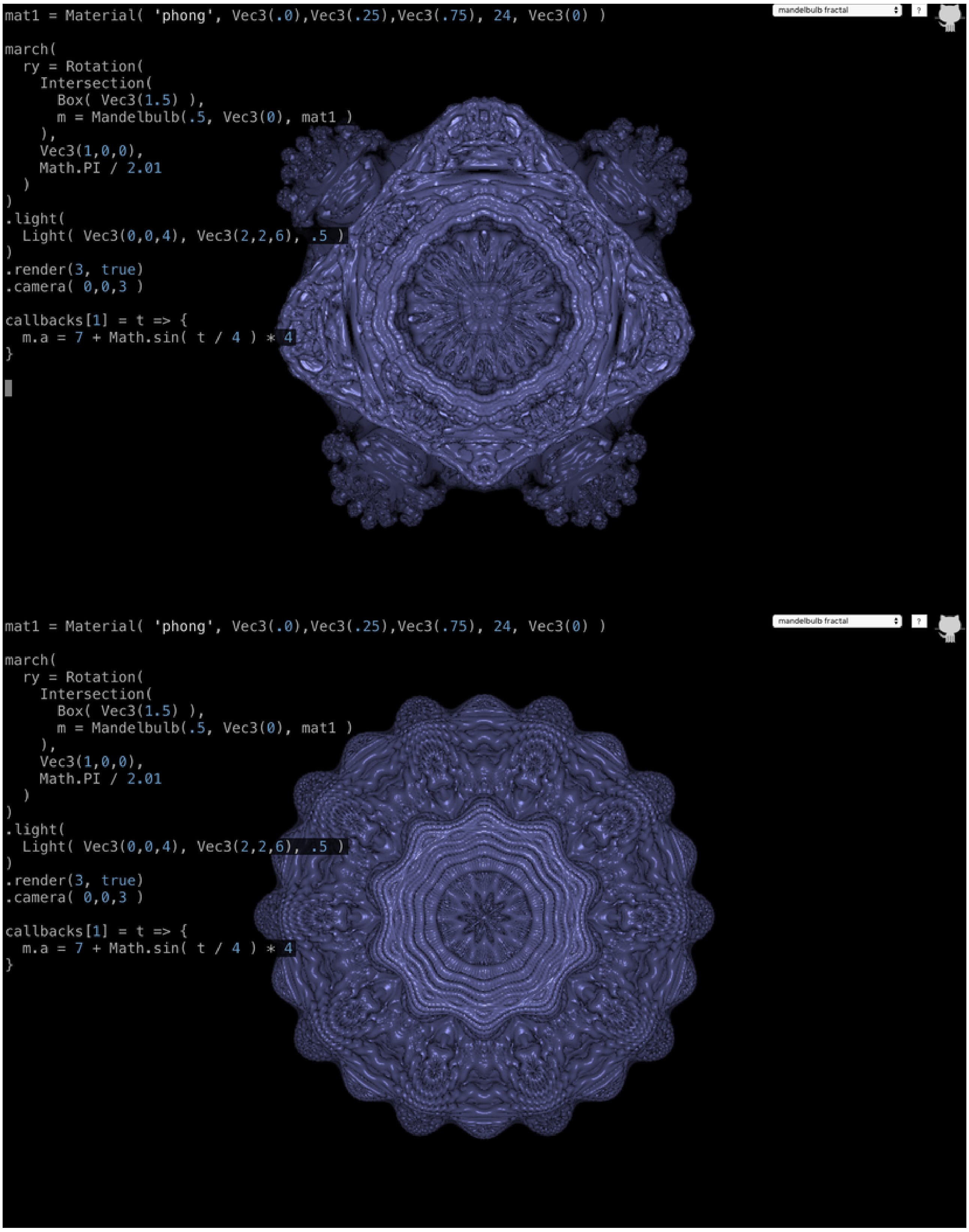
Performers can specify an array of callbacks that are each called, in succession, on each frame of video. A time argument is passed to each callback to use as a parameter for manipulating the raymarching scene. Every primitive, distance operation, and domain transformation provides *shader uniforms* (access to memory locations in the GPU from the CPU) that can be easily manipulated in these callbacks. In Figure 3, the `a` coefficient in the Mandelbulb equation (a three-dimensional version of the Mandelbrot fractal) is changed over time.

These callbacks can also be used to look at current output values of an FFT analysis applied to an incoming audio feed (separated into low, mid, and high bins), an RMS averaging of an incoming audio feed, or mouse position. Additional callbacks can be registered to listen for incoming MIDI and OSC messages. OSC support requires running a small node.js server in conjunction with the marching.js playground while MIDI requires a browser supporting the WebMIDI API.

3.3 Constructive Solid Geometry and Live Coding

Raymarching provides very different opportunities for manipulating shapes than vertex-based meshes. In a raymarching engine, all primitives are described by simple mathematical formulae (called *distance estimators*), which can then be combined and manipulated in a variety of ways. The techniques for combining primitives broadly fall under the category of *constructive solid geometry*, and are also widely applied in computer-assisted-design (CAD) and 3D modeling applications.

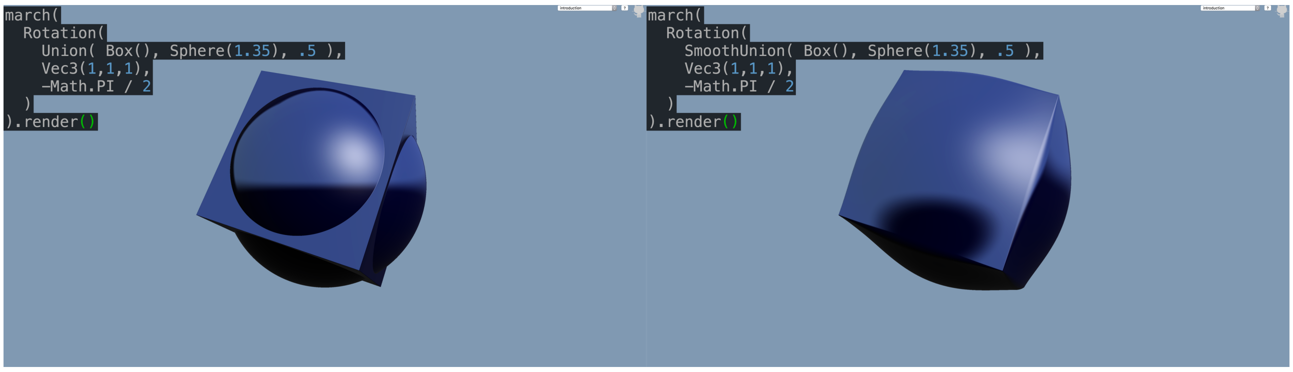
Distance operations in raymarchers are operators that can manipulate distance estimators (primitives). We believe these operations provide unique opportunities for live coding performance. The simplest operation, *Union*, simply combines



```
mat1 = Material( 'phong', Vec3(.0),Vec3(.25),Vec3(.75), 24, Vec3(0) )

march(
  ry = Rotation(
    Intersection(
      Box( Vec3(1.5) ),
      m = Mandelbulb(.5, Vec3(0), mat1 )
    ),
    Vec3(1,0,0),
    Math.PI / 2.01
  )
)
.light(
  Light( Vec3(0,0,4), Vec3(2,2,6), .5 )
)
.render(3, true)
.camera( 0,0,3 )

callbacks[1] = t => {
  m.a = 7 + Math.sin( t / 4 ) * 4
}
```

```
mat1 = Material( 'phong', Vec3(.0),Vec3(.25),Vec3(.75), 24, Vec3(0) )

march(
  ry = Rotation(
    Intersection(
      Box( Vec3(1.5) ),
      m = Mandelbulb(.5, Vec3(0), mat1 )
    ),
    Vec3(1,0,0),
    Math.PI / 2.01
  )
)
.light(
  Light( Vec3(0,0,4), Vec3(2,2,6), .5 )
)
.render(3, true)
.camera( 0,0,3 )

callbacks[1] = t => {
  m.a = 7 + Math.sin( t / 4 ) * 4
}
```

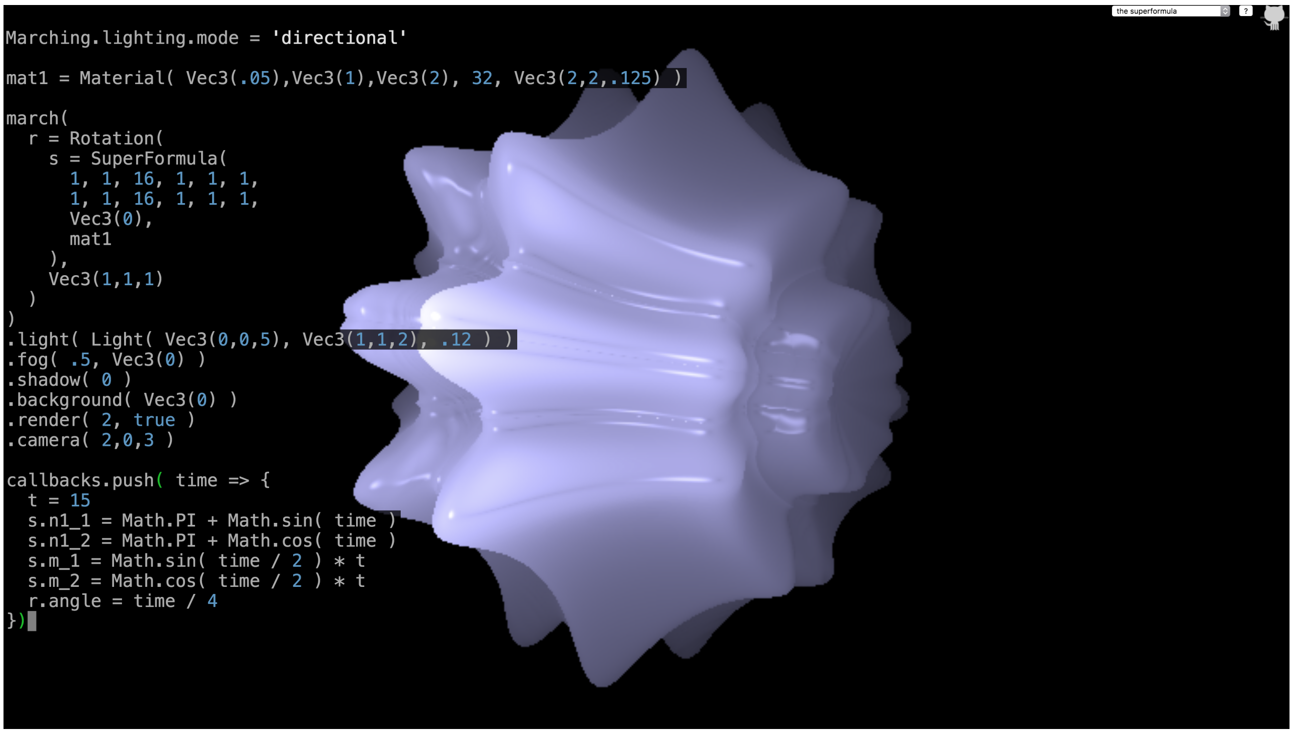
Figure 3: *Two screenshots of a Mandelbulb evolving over time.*



```
march(
  Rotation(
    Union( Box(), Sphere(1.35), .5 ),
    Vec3(1,1,1),
    -Math.PI / 2
  )
).render()
```

```
march(
  Rotation(
    SmoothUnion( Box(), Sphere(1.35), .5 ),
    Vec3(1,1,1),
    -Math.PI / 2
  )
).render()
```

Figure 4: *A union and smooth union of two primitives compared.*



```
Marching.lighting.mode = 'directional'

mat1 = Material( Vec3(.05),Vec3(1),Vec3(2), 32, Vec3(2,2,.125) )

march(
  r = Rotation(
    s = SuperFormula(
      1, 1, 16, 1, 1, 1,
      1, 1, 16, 1, 1, 1,
      Vec3(0),
      mat1
    ),
    Vec3(1,1,1)
  )
)
.light( Light( Vec3(0,0,5), Vec3(1,1,2), .12 ) )
.fog( .5, Vec3(0) )
.shadow( 0 )
.background( Vec3(0) )
.render( 2, true )
.camera( 2,0,3 )

callbacks.push( time => {
  t = 15
  s.n1_1 = Math.PI + Math.sin( time )
  s.n1_2 = Math.PI + Math.cos( time )
  s.m_1 = Math.sin( time / 2 ) * t
  s.m_2 = Math.cos( time / 2 ) * t
  r.angle = time / 4
})
```

Figure 5: *The 3D Super Formula, as rendered by marching.js*

the primitives created by two distance estimators together, with an abrupt transition. However, the *SmoothUnion* operator creates a smooth transition between the surfaces based on a provided coefficient; this coefficient can then be changed over time, or controlled by an audio signal, creating undulating combinations of forms that would be difficult to achieve using a raster-based graphics engine. Examples of a `Union` and `SmoothUnion` combining a `Sphere` primitive and a `Box` primitive are shown side-by-side in Figure 4.

Other distance operations provide a variety of transitions between shapes. When the effect of these transitions is sufficiently exaggerated, new and unexpected forms can emerge. Such complex forms can be programmed ahead of performances and then manipulated algorithmically; alternatively live coders could gradually create complex forms from scratch and begin manipulating them when they've achieved sufficient complexity, in the same fashion as a musical performance beginning with sine oscillators that are gradually combined to create more complex timbres.

4 Preliminary thoughts, potentials, conclusions

We would describe much of the output of marching.js as "technical", "clinical", or perhaps even "cold". The addition of post-processing filters to add noise/dirt to the renderings would help in this regard. We also look forward to experimenting with connecting the output of marching.js to Hydra, so that it can be manipulated using emulated analog video synthesis techniques, and believe more generally that marching.js can provide interesting source material for further digital manipulation. Although the output is typically cold and sterile, marching.js can also be used to create interesting glitch effects by assigning coefficients that force geometries to render in unrealistic ways.

Eventually, marching.js will replace the existing graphics engine for the live coding environment Gibber, which currently includes a variety of post-processing filters that could be applied the raymarcher's output. This will also expose the raymarching engine to Gibber's sequencing and multimodal mapping capabilities. We believe the addition of marching.js will provide for more interesting / unique visuals than what Gibber currently is capable of producing.

Another area we're excited to explore with marching.js is 3D printing. The constructive solid geometry techniques the library is based around are well-suited for such applications, and we would only need to add a mesh approximation algorithm (such as marching cubes) to generate a mesh from 3D distance function used by the raymarcher that would (potentially) be suitable for printing. Having an interactive coding environment supporting both live coding performance and 3D printing could create interesting educational possibilities around the idea of geometric forms and constructive solid geometry.

Finally, we are also interested in adding more complex 3D forms for exploration for live coders. Currently marching.js contains a variety of 3D fractal implementations (the Mandelbulb is shown in Figure 3) in addition to a 3D implementation of the Super Formula, as shown in Figure 5. Adding more complex forms will provide more opportunities for exploration and play, as well as greater variety in the types of performances that can be realized with the library.

4.1 Acknowledgments

Special thanks to Chi Shen for his initial development work on this project.

References

Amagi, Takayosi. n.d. "VEDA - VJ app for Atom." https://veda.gl/.

Burger, Boris, Ondrej Paulovic, and Milos Hasan. 2002. "Realtime Visualization Methods in the Demoscene." In *Proceedings of the Central European Seminar on Computer Graphics*, 205–18.

Carlsson, Anders. 2009. "The Forgotten Pioneers of Creative Hacking and Social Networking–Introducing the Demoscene." *Re: Live* 16.

DeFanti, Thomas A. 1976. "The Digital Component of the Circle Graphics Habitat." In *Proceedings of the June 7-10, 1976, National Computer Conference and Exposition*, 195–203. ACM.

Della Casa, Davide, and Guy John. 2014. "LiveCodeLab 2.0 and its language LiveCodeLang." In *Proceedings of the Second Acm Sigplan Workshop on Functional Art, Music, Modeling & Design*, 1–8. ACM.

Griffiths, Dave. 2014. "Fluxus." In *Collaboration and learning through live coding (Dagstuhl Seminar 13382)*, 3:149–50. 9.

"hexler.net | KodeLife." n.d. https://hexler.net/software/kodelife/.

"Hydra." n.d. https://hydra-editor-v1.glitch.me.

Lawson, Shawn, and Ryan Ross Smith. 2018. "The Dark Side." In *Proceedings of the International Conference on Live Coding*.

Papert, Seymour. 1980. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc.

Roberts, Charles, Matthew Wright, JoAnn Kuchera-Morin, and Tobias Höllerer. 2014. "Gibber: Abstractions for Creative Multimedia Programming." In *Proceedings of the Acm International Conference on Multimedia*, 67–76. ACM.

Tuy, Heang K, and Lee Tan Tuy. 1984. "Direct 2-d Display of 3-d Objects." *IEEE Computer Graphics and Applications* 4 (10). IEEE: 29–34.

Wakefield, Graham, Wesley Smith, and Charles Roberts. 2010. "LuaAV: Extensibility and Heterogeneity for Audiovisual Computing." In *Proceedings of the Linux Audio Conference*.