Live Coding Procedural Textures of Implicit Surfaces

Charles Roberts
Department of Computer Science
Worcester Polytechnic Institute
charlie@charlie-roberts.com

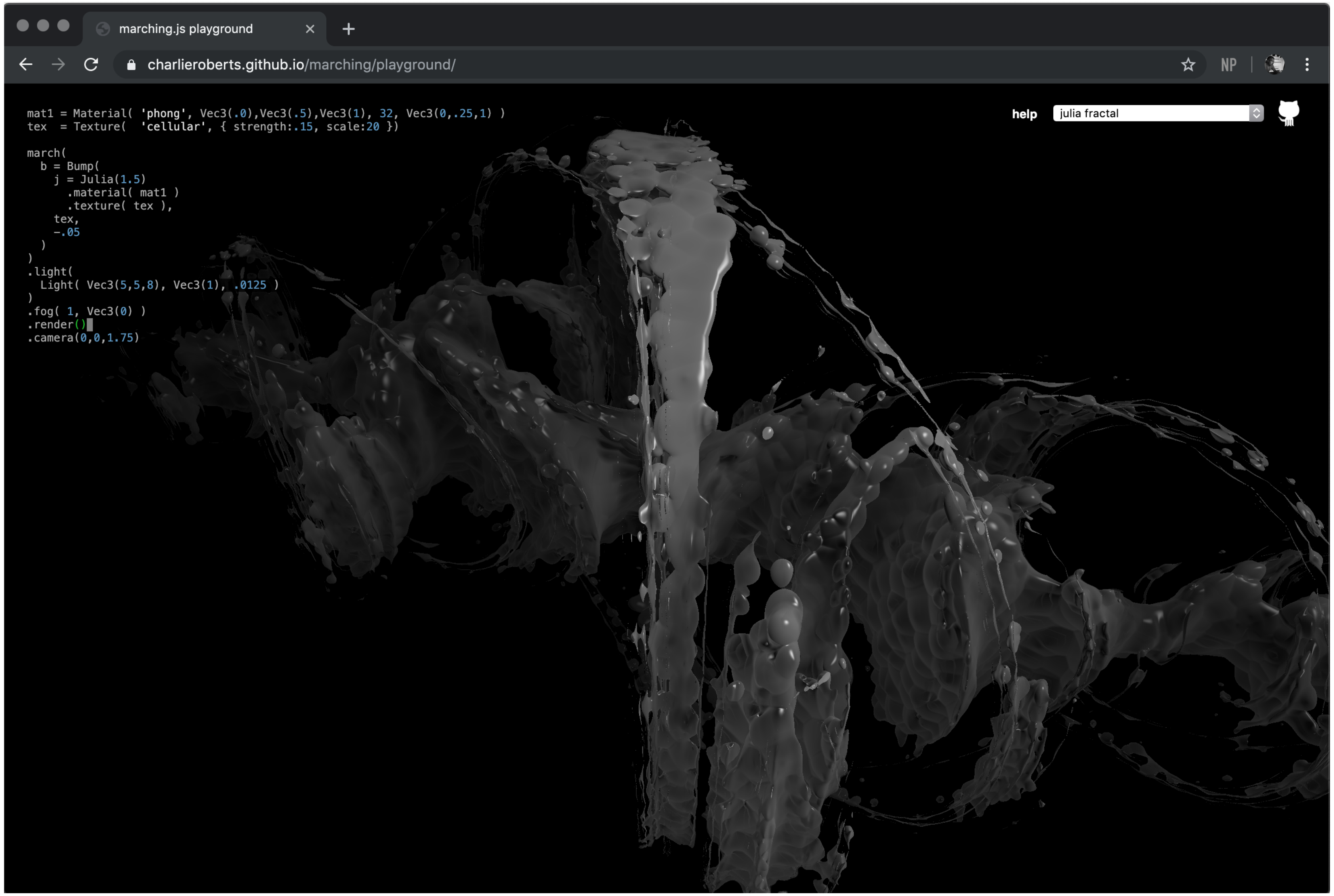
Abstract

We describe a system for live coding procedural textures on implicit surfaces, and how its implementation led to foundational changes in the end-user API for the live coding environment marching.js. The texturing additions to marching.js enable users to use predefined texture presets, to live code their own procedural textures, or to use other systems for fragment shader authoring, such as Hydra, to generate textures. It also affords using the browser's 2D canvas API to define textures, providing an entry point for performers who might be familiar with web APIs but unfamiliar with lower-level GPU languages like GLSL. We describe how demoscene culture led us to initially adopt design decisions that were inappropriate for our particular system, and the changes to both our underlying engine and end-user interface that resulted from reconsidering these decisions in the context of procedural texturing.

Introduction

We previously developed a library, marching.js, that exposes a ray marching engine for live coding performance. This system enables programmers to describe 3D scenes in JavaScript, which are then compiled into fullscreen GLSL fragment shaders. In our initial writings on the library (Roberts 2019) we described how the scenes generated by the system often felt "…'technical', 'clinical', or perhaps even 'cold'." While post-processing filters were mentioned as one possible solution for this problem, the research presented here instead investigates a variety of techniques to enable procedural texturing of the implicit surfaces (Hart 1993) rendered by marching.js.

In somewhat of a surprise, the implementation of these features led to fundamental API changes in our system and rethinking culturally derived assumptions about how the rendering engine for our system should function. We will describe some of the background that led to these assumptions, and how the implementation of procedural texturing for implicit surfaces led to both a terser end-user programming



```
mat1 = Material( 'phong', Vec3(.0),Vec3(.5),Vec3(1), 32, Vec3(0,.25,1) )
tex  = Texture(  'cellular', { strength:.15, scale:20 })

march(
  b = Bump(
    j = Julia(1.5)
      .material( mat1 )
      .texture( tex ),
    tex,
    -.05
  )
)
.light(
  Light( Vec3(5,5,8), Vec3(1), .0125 )
)
.fog( 1, Vec3(0) )
.render()
.camera(0,0,1.75)
```

Figure 1: The quaternion Julia set, textured and bump-mapped with cellular noise, as rendered by marching.js

interface and low-level engine optimizations. We then outline various levels of interface for our texturing system, and our attempts to ensure the idealized "low threshold, high ceiling, wide walls" (Resnick et al. 2005) design space that help characterize successful creativity support tools. We conclude with technical and aesthetic directions for future research.

Background

In this section we begin by briefly describing ray marching as a rendering technique. We then discuss how the demoscene (Carlson 2009) affected many of our design decisions when initially authoring marching.js, and contextualize the use of procedural texturing in marching.js within the broader community of live coders.

Ray Marching

Ray marching is a method of rendering (primarily) three-dimensional scenes. It is perhaps best understood in contrast to a much more common 3D graphics pipeline, which incorporates tessellation and rasterization. In this process, geometries are subdivided into triangles (tessellation). The vertices for these triangles are then sent to the GPU, where the triangles are reassembled and projected from their 3D location to the 2D viewing plane (rasterization).

In contrast, ray marching is a physically-informed rendering technique that enables programmers to use mathematical formulae to define and combine geometries, without having to worry about tessellation or rasterization. In ray marching, a ray is projected from a virtual camera through each pixel in the output and into a three-dimensional scene; if this ray strikes an object in the 3D scene the pixel that the ray travels through is assigned the color of that particular object. This rendering technique makes a variety of operations that are complex to perform with tessellated triangles much simpler, such as fluidly morphing from one shape to another, or infinitely repeating a geometry throughout a space. However, tessellation and rasterization are extremely efficient, while ray marching typically requires a fairly powerful graphics card to perform realtime rendering on high resolution displays. We describe ray marching in greater detail in our prior writings about marching.js (Roberts 2019).

Cultural Assumptions in Marching.js

Many popular introductory tutorials on ray marching are presented in the context of the demoscene, a culture that emphasizes the production of audiovisual sketches (termed *demos*) that explore the boundaries of what is possible within technical constraints. These constraints can include the adoption of a particular low-resource technology platform, artificial constraints on the number of bytes a program can occupy in memory, or, in the case of many live demoscene competitions, the challenges of creating complex three-dimensional worlds in realtime in a competitive head-to-head "battle" setting. The demoscene features a variety of online venues for promoting discussion and dissemination of the idiomatic techniques used within it. These include the labyrinthine pouet.net—a popular forum for discussing techniques ranging from shader programming to analog techniques involving overhead projectors and paper cutouts—and shadertoy.com, a site for viewing, editing, and sharing demos realized in the browser using GLSL (Graphics Language Shader Language), one of the most widely used languages for authoring programs that are parallelized to run on the graphics programming unit (GPU) of a computer. The first version of marching.js was heavily influenced by the cultural focus of the demoscene on GLSL demos and the ready availability of related references and tutorials. This led to questionable design decisions that we re-examined in the context of procedural texturing.

Live Coding of Texture

Our research is particularly interested in the application of texture to three-dimensional geometries; however, we note there is also a broader discussion of texture within the live coding community as it relates to both musical pattern and computational craft (McLean 2013).

Additionally, there is an established practice of live coding fullscreen fragment shaders in the live coding community; these shaders can be thought of as textures for simple rectangles that fill the entire projection. Popular environments for GLSL live coding of this type include The Force (Lawson & Smith 2017), KodeLife (Fischer n.d.), and Veda (Amagi n.d.), while the demoscene community typically uses a standardized system named Bonzomatic (Szelei n.d.) for live competitions. Other visual live coding systems, such as La Habra (Hennigh-Palermo n.d.) and Visor (Purvis, Anslow & Noble 2019) primarily use 2D programming APIs, such as Processing (Reas & Fry 2006) in the case of Visor or a ClojureScript environment for programming Scalable Vector Graphics in the case of La Habra.

The live coding system Hydra (Jack n.d.) adopts a different approach, providing an end-user JavaScript API that wraps a code generation engine for writing GLSL shaders. Hydra is "...a modular and distributed video synthesizer" (ibid.), and similar to most analog video synthesis systems the output can typically be considered two-dimensional.

Our research on using textures within marching.js builds off of many of the ideas found in these other systems, enabling live coders to define textures that are created using the built-in HTML ¡canvas¿ element, and to use Hydra as a texture for the implicit surfaces marching.js provides.

Rethinking the Live Coding Interface for Marching.js

As we implemented procedural texturing for the first time in marching.js, we faced a difficult problem.

The code in Listing 1, creates a box that is rotated on its x-axis and scaled before rendering. When we first tried to map textures to such geometries, the effect became that of a textured blanket layered over the top of the geometry: the box would rotate underneath and scale appropriately, but the "blanket" would just hang in place while barely moving, instead of being wrapped tightly around it so that as the geometry rotated, the texture did as well. More succinctly: our geometry rotated but our texture did not.

```
march(
  Rotate(
    Scale( Box(), .5 )
    Vec3( 1,0,0 ),
    Math.PI / 3
  )
).render()
```

Listing 1: A scaled and rotated box in marching.js

This problem is more complex when we look at aggregate objects that contain transformations applied to individual members of the aggregate as well as the aggregate itself. For example, consider code in Listing 2.

```
march(
  Rotate(
    Union(
      Rotate( Box(), Vec3(0,1,0), Math.PI/3 ),
      Sphere( 1.25 )
    ),
    Vec3(1), Math.PI/5
  )
).render()
```

Listing 2: A rotated box and a sphere combined via a Union combinator which is then also rotated

In Listing 2 we apply a rotation to our box, and then apply a rotation to the union of the rotated box and the sphere. If we textured the resulting aggregate geometry, we would need to take both of these rotations into account at different parts of the texturing process.

The code generation engine in marching.js is effectively divided into two stages. In the first, we determine whether or not rays traveling through a pixel on our screen hit an object in the scene; if so, we need to color that pixel based on the material / texture of the object, and on the lighting of the scene. The second lighting stage calculates this color, however, in marching.js version 1 the lighting stage cannot access the transforms of the geometry that is being lit. We had to significantly refactor the code generation engine in order provide access to these transformations, which in turn led to questioning some of our underlying assumptions about how our engine should function. These changes enable users to freely assign transformations at any level of hierarchy, as shown in Figure 2.

Transform Everything

The first change we made was to assign matrix-encoded transforms for rotation, translation, and scale to every operation in marching.js. Explicitly wrapping individual functions in such transforms was no longer necessary, and a transformation matrix is automatically applied to all geometries.

Listing 3: Comparing old and new syntaxes for rotating a box in marching.js captionpos

```
// old syntax
march( Rotate( Box(), Vec3(1,0,0), Math.PI/2 ) )
  .render()

// new syntax
march( Box().rotate( 45, 1,0,0 ) ).render()
```

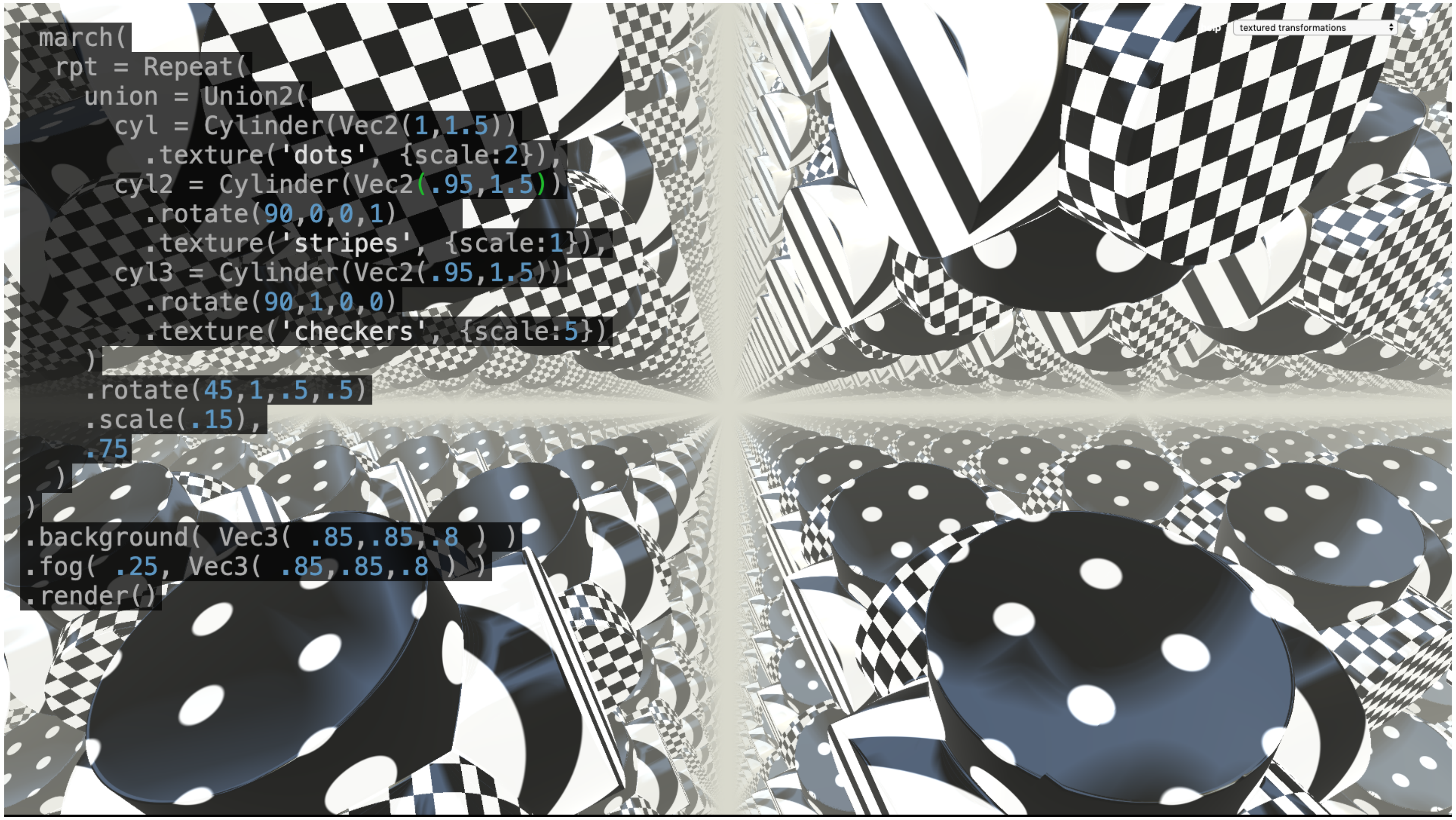
One effect of our changes is that the API for applying transformations immediately became much terser; in our opinion its clarity is also improved. Listing 4 shows a more complex example for comparison:

Listing 4: Comparing syntaxes for applying a variety of transformations to a box in marching.js captionpos

```
//old syntax to rotate,translate,and apply material
march(
  Rotate(
    Box( null, Vec3(1,0,0), Material('glue') ),
    Vec3( 1,0,0 ),
    .5
  )
).render()

//new syntax to rotate,translate,and apply material
march(
  Box()
    .rotate( Math.PI / 3, 1,0,0 )
    .translate( 1,0,0 )
    .material( 'glue' )
).render()
```

The new syntax is more explicit about what is occurring, making it easier to read, while only being two characters greater in length in Listing 4. It also helps to avoid deep nesting which can difficult to parse and awkward to augment with additional code. Given that scaling, movement, and rotation are typically important parts of performing with 3D geometries, we feel that improving the application of such transformations is significant.

A tradeoff to these benefits is consistency. In first release of marching.js, nested functions were used to create geometries, transform them, and apply domain operations that could radically alter a scene; in the newest version, transforms, texturing, and application of material are instead achieved by method calls. We experimented with also applying domain operations using method calls, however, we found the resulting syntax to be ambiguous and difficult to apply consistently. In its new form, the programming interface is currently non-homogenous in how various operations are applied, but we still



```
 march(
  rpt = Repeat(
    union = Union2(
      cyl = Cylinder(Vec2(1,1.5))
        .texture('dots', {scale:2}),
      cyl2 = Cylinder(Vec2(.95,1.5))
        .rotate(90,0,0,1)
        .texture('stripes', {scale:1}),
      cyl3 = Cylinder(Vec2(.95,1.5))
        .rotate(90,1,0,0)
        .texture('checkers', {scale:5})
    )
    .rotate(45,1,.5,.5)
    .scale(.15),
    .75
  )
)
.background( Vec3( .85,.85,.8 ) )
.fog( .25, Vec3( .85,.85,.8 ) )
.render()
```

Figure 2: Three cylinders with different rotations, scaled and repeated, with coherent procedural textures.

believe it is clearer than our prior solution.

Improving Efficiency and Code Generation

As discussed previously, the implementation for marching.js was created using online references and code examples. The majority of these references were authored by demoscene participants, who commonly perform all graphics processing on the GPU. This is an aesthetic choice that places all graphics code in a (typically) single file using a single language (GLSL), making it easier for viewers and programmers to understand. However, some operations, such as the transformations described in this section, are in fact more efficient to perform on the CPU.

The reason for this is the parallel nature of GPUs, which makes it difficult to share information across various invocations of the main fragment shader function. Since this main function is invoked once per pixel being rendered, GPU based transformations are thus calculated thousands of times per frame, and then must be repeated on every additional frame. For some transformations, like translation, this is not a significant cost, however, for others such as rotation it is an expense best avoided.

Now that every operation in marching.js has a transformation matrix associated with it, we can calculate this matrix a single time on the CPU, transfer the matrix to the GPU, and then use the same data for rendering every pixel in the operation. The transform doesn't need to be recalculated unless the it is changed in some fashion (for example, increasing rotation), meaning in some cases we only need to calculate the transformation a single time. This is clearly a win over having to recalculate it for every pixel on every frame, regardless of whether any changes to the transformation have occurred. Such optimizations are perhaps obvious in hindsight, but were only achieved by reconsidering the context of the demoscene tutorials, references, and libraries that influenced marching.js.

Changing Code Generation

In the generated shaders, each operation references a matrix that represents the operation's cumulative transformation. This includes transformations applied directly to the operation, transformations applied to any domain operations that wrap the operation, and transformations that might be applied to any higher-level geometry that the operation is a part of. As these various transformations are applied (usually via matrix multiplication, with the code generation engine ensuring correct application order by explicitly writing it into the generated shader), the code generation engine stores each step of the transformation as needed so that it can be referenced during texturing.

Textures

marching.js enables users to approach texturing in a variety of different ways, providing texturing options for beginning programmers as well as more advanced programmers who are fluent in GLSL. In order of increasing complexity, these techniques include:

1. Predefined 2D GLSL textures that can be wrapped around objects
2. Predefined 3D GLSL textures
3. Using a standard image file (.png, .gif, .jpg etc.)
4. Using the 2D <canvas> API provided by the browser
5. Using Hydra and other systems that output to <canvas> elements
6. Writing custom GLSL textures

Predefined Textures

The predefined textures included with marching.js (shown in Fig.2) are accessible via presets that can referenced by name, as shown in Listing 5.

```
march(
  Box().texture( 'dots' )
).render()
```

Listing 5: Using a texture preset in marching.js

Texture objects can also be defined and used in multiple geometries. Additionally when a call to .texture() is used on a geometric combinator (Union, Intersection, Difference etc.) the texture is applied to all surfaces belonging to the combinator; this also applies to domain operations like Repetition. Listing 6 provides code examples of both methods.

```
// define a texture used by multiple objects
tex = Texture( 'truchet' )
march(
  Box().texture( tex ),
  Sphere( 1.35 ).texture( tex )
).render()

// or use a combinator to apply texture
march(
  Difference(
    Box(),
    Sphere( 1.25 )
  ).texture( 'truchet' )
).render()
```

Listing 6: Applying one texture across multiple geometries via reusing a texture and applying a texture to a combinator

Using the HTML <canvas> Element as a Texture

Many beginning web and graphics programmers experiment with the HTML 2D <canvas> API. By offering <canvas> as one of the options for texturing in marching.js, we enable these programmers to easily experiment with texturing without having to learn GLSL. These textures can be animated and updated in the onframe method that marching.js uses for animation.

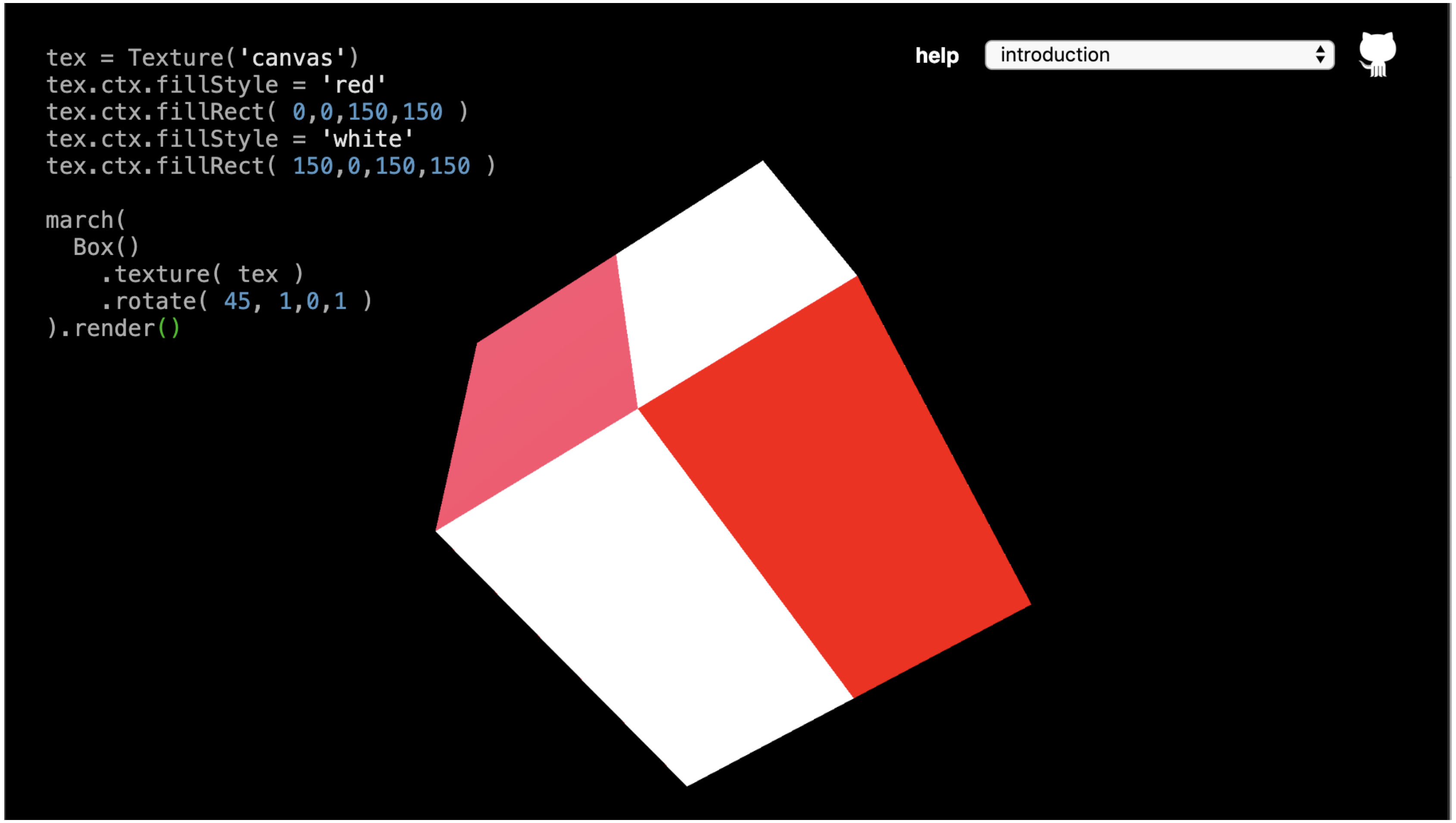
Integrating with Hydra

Hydra is a popular live coding system that operates on a similar principle to marching.js: users provide a high-level description in JavaScript of a representation which is then compiled to a GLSL shader for display. In Hydra, the operations are typically derived from analog video synthesis techniques, while in marching.js the operations relate to volumetric rendering and constructive solid geometry.

Performers use Hydra to create 2D patterns that change over time, making it a perfect candidate to use as texture generator for marching.js. Fig 4 shows Hydra being used to texture a Mandelbox fractal. The Hydra graph can be edited and redefined at any time to update the applied shader texture. We imagine future collaborative performances where one user could program textures in Hydra while another programmed 3D scenes that used the generated textures.

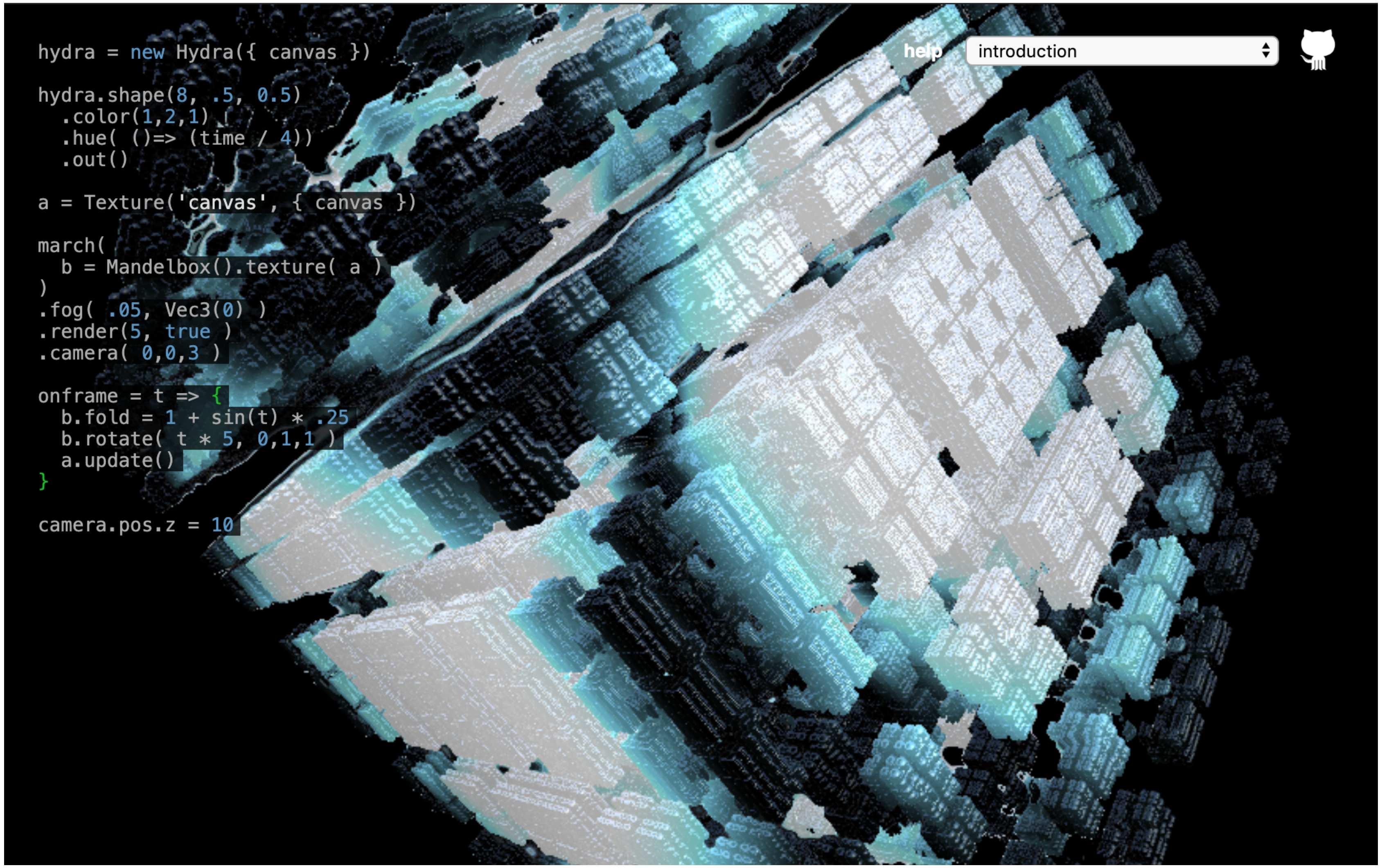
GLSL Textures

Fragment shaders for texturing can also be authored directly inside of marching.js, via the same API that is used internally to define the various texture presets included in marching.js. This API enables end-users to define points for interacting with their texture as well as the raw GLSL code that is needed to calculate an output color value.



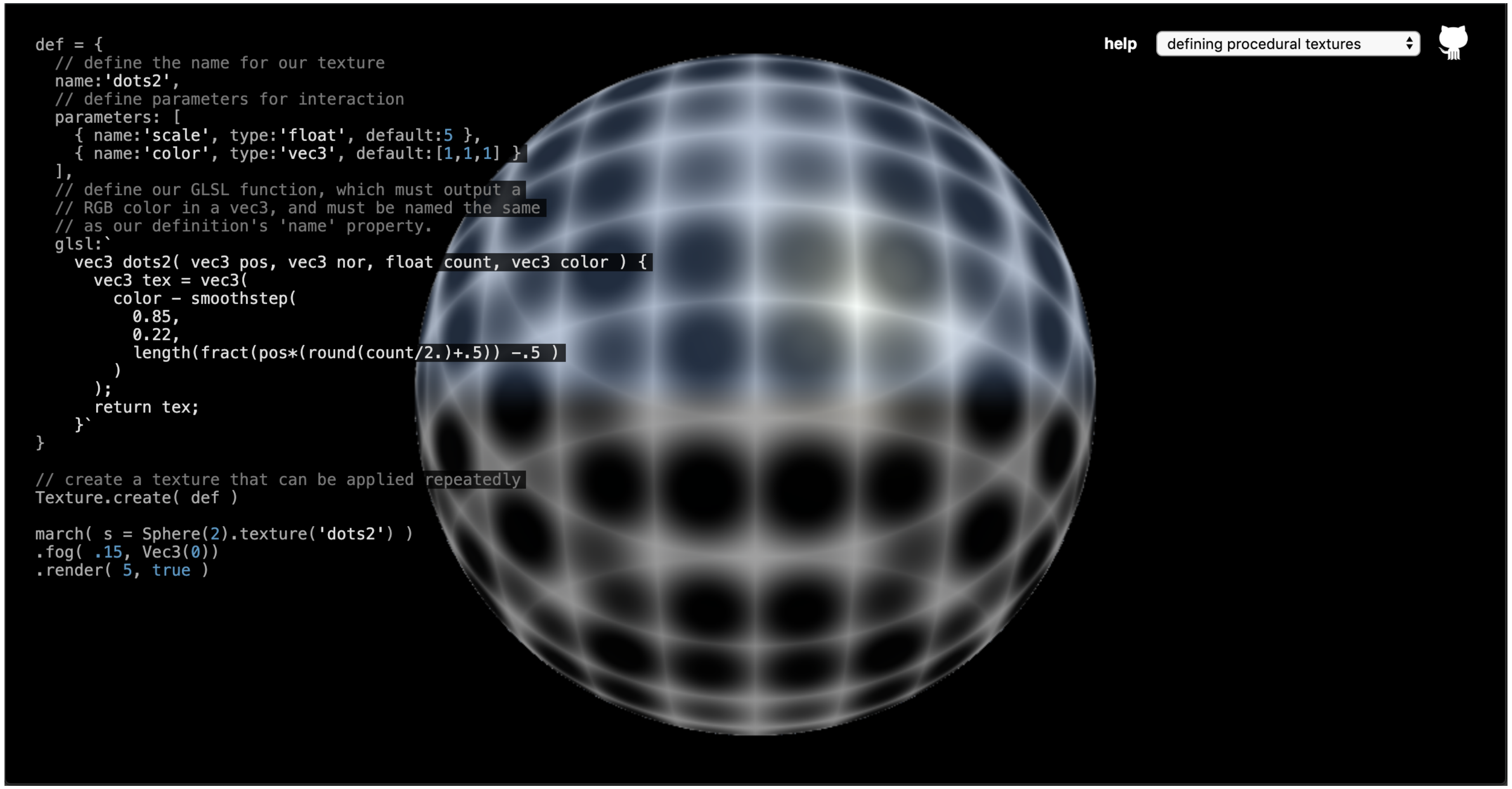
```
tex = Texture('canvas')
tex.ctx.fillStyle = 'red'
tex.ctx.fillRect( 0,0,150,150 )
tex.ctx.fillStyle = 'white'
tex.ctx.fillRect( 150,0,150,150 )

march(
  Box()
    .texture( tex )
    .rotate( 45, 1,0,1 )
).render()
```

Figure 3: Using a HTML <canvas> element to texture a surface.



```
hydra = new Hydra({ canvas })

hydra.shape(8, .5, 0.5)
  .color(1,2,1)
  .hue( ()=> (time / 4))
  .out()

a = Texture('canvas', { canvas })

march(
  b = Mandelbox().texture( a )
)
.fog( .05, Vec3(0) )
.render(5, true )
.camera( 0,0,3 )

onframe = t => {
  b.fold = 1 + sin(t) * .25
  b.rotate( t * 5, 0,1,1 )
  a.update()
}

camera.pos.z = 10
```

Figure 4: Hydra in use to texture a Mandelbox fractal.



```
def = {
  // define the name for our texture
  name:'dots2',
  // define parameters for interaction
  parameters: [
    { name:'scale', type:'float', default:5 },
    { name:'color', type:'vec3', default:[1,1,1] }
  ],
  // define our GLSL function, which must output a
  // RGB color in a vec3, and must be named the same
  // as our definition's 'name' property.
  glsl:`
    vec3 dots2( vec3 pos, vec3 nor, float count, vec3 color ) {
      vec3 tex = vec3(
        color - smoothstep(
          0.85,
          0.22,
          length(fract(pos*(round(count/2.)+.5)) -.5 )
        )
      );
      return tex;
    }`
}

// create a texture that can be applied repeatedly
Texture.create( def )

march( s = Sphere(2).texture('dots2') )
.fog( .15, Vec3(0))
.render( 5, true )
```

Figure 5: Defining and using a texture written in GLSL.

Manipulating Textures

After applying a texture the properties of the generated texture are added as members of the texture function itself, exposing them for realtime control. In Listing 7, the time property of a 4D simplex noise texture is changed in each frame of generated video.

```
march(
  plane = Plane().texture('noise')
).render()

onframe = function( time ) {
  plane.texture.time = time
}
```

Listing 7: Changing texture properties over time

While most texture properties vary according to the preset used, every texture has a scale property that is used as a scalar to modify the texturing coordinates internally to the shader. Most also have a 'strength' property that determines the effect of the texture in determining the final color of each pixel.

Conclusions and Future Work

We extended a system/library, marching.js, to include a variety of methods for texturing implicit surfaces. This required rethinking fundamental aspects of how its code generation and ray marching engines functioned, but resulted in a terser, more readable end-user API that should lead to more fluid live coding performances. The texturing methods we implemented help to ensure that programmers of varying experience will be able to experiment with texturing, while providing integration with the live coding system Hydra ensures that its users can transfer prior knowledge while experimenting or performing with volumetric rendering techniques. The updates to this system are open source and available online at https://charlieroberts.github.io/marching/playground/.

There are improvements to be made in the sampling algorithms for 2D textures and anti-aliasing more generally in marching.js. Additionally, integration with p5.js, a JavaScript port of Processing (McCarthy, Reas & Fry 2015), could open texturing in marching.js to the many artists and students who actively use that platform. Conversely, we are also considering porting the library to run in p5.js, so that the Processing community will have a relatively easy platform to explore volumetric rendering.

References

Amgai, T. (n.d.) VEDA-VJ app for Atom. [Online] Available at: `https://veda.gl/`. Accessed on Wed, September 25, 2019.

Carlsson, A. (2009), The forgotten pioneers of creative hacking and social networking–introducing the demoscene, Re: Live: Media Art Histories 2009 Conference Proceedings. pp. 16–20.

Hennigh-Palermo, S. (n.d.) La Habra: The Shape of Things to DOM. [Online] Available at: `https://github.com/sarahgp/la-habra` . Accessed on Mon, December 16, 2019.

Hart, J. C. (1993), Ray tracing implicit surfaces. In: Siggraph 93 Course Notes: Design, Visualization and Animation of Implicit Surfaces pp. 1–16.

Fischer, R. (n.d.) KodeLife. [Online] Available at: `https://hexler.net/products/kodelife` . Accessed on Wed, September 25, 2019

Jack, O. (n.d.) hydra. [Online] Available at: `https://hydra-editor.glitch.me`. Accessed on Mon, December 16, 2019.

Lawson, S. & Smith, R. R. (2017), The Dark Side. In: Centro Mexicano para la Musica y las Arts Sonoras (Mexico): Proceedings of the Third International Conference on Live Coding.

McCarthy, L., Reas, C. & Fry, B. (2015) Getting Started with P5. js: Making Interactive Graphics in JavaScript and Processing. Maker Media, Inc.

McLean, A. (2013) The textural x. In: Proceedings of xCoAx2013: Computation Communication Aesthetics and X. pp.81–88.

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

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

Resnick, M., Myers, B., K., N., Shneiderman, B., Pausch, R., Selker, T. & Eisenberg, M. (2005) Design Principles for Tools to Support Creative Thinking. Technical report.

Roberts, C. (2019) Live Coding Ray Marchers with Marching.js. In: Proceedings of the 2019 International Conference on Live Coding.

Szelei, G. (n.d.) Bonzomatic: Tool for the Live Coding Compo debuted at Revision 2014. [online] Available at: `https://github.com/Gargaj/Bonzomatic`. Accessed on Monday, December 16, 2019.