

Composable function systems as a general-purpose rendering framework

JAMES SCHLOSS, Massachusetts Institute of Technology, USA

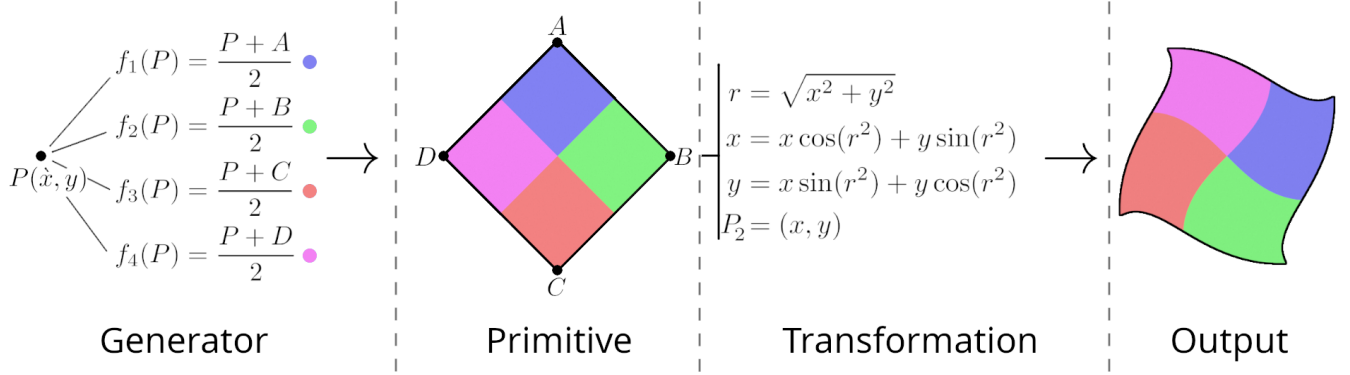


Fig. 1. A single step in a simple CFS workflow, including the generation of an object primitive with an iterated function system followed by a spatial transformation with human-readable mathematical operation. Note that the initial point (P) is distinct from the final output point (P_2).

Function systems exist as a natural language for the meshless creation and manipulation of complex objects while maintaining minimal memory on the Graphics Processing Unit (GPU) or Central Processing Unit (CPU). This paper proposes a new method for general-purpose (non-fractal) visualizations and simulations with function systems and introduces Quibble, a metaprogramming framework for composing such systems on the GPU. This is the first time users can compose (for example) Iterated Function Systems (IFSs) with simple algebraic manipulations to create complex scenes. We also discuss several core advantages of this method including runtime performance, the creation of topologically non-trivial objects, and interoperability with other graphical algorithms. Beyond general-purpose imagery and animations, this method can also be used to give artists more control over in-between frames in low-framerate animations, controllably deform point clouds, and metaprogram difficult animation workflows.

CCS Concepts: • **Computing methodologies** → **Rendering**; **Point-based models**; **Graphics processors**; *Parallel programming languages*.

Additional Key Words and Phrases: Iterated Function Systems, Rendering

1 Introduction

Function systems are powerful tools that have been used to generate mathematical art for decades [Barnsley et al. 2003; Barnsley and Demko 1985; Barnsley et al. 1989; Duff 1992; Elliott 2003; Keeter 2020]. Iterated Function Systems (IFSs), in particular, are likely the most well-studied functional approach for creating and studying self-similar (fractal-like) behaviour [Bouke 2025; Diaconis and Freedman 1999; Ghosh and Marecek 2022; Iosifescu 2009] and have been used for many exciting applications, such as spatial sampling and fractal image compression [Fisher 1994; Kominek 1995]. Solving IFSs is typically done with a random sampling method known as the chaos game, where a single point chooses a single function at random every step [Barnsley and Vince 2011]. After transformation by the chosen function, the point then colors the appropriate pixel on screen before moving on to the next step. As the simulation continues, the point

drifts towards the final shape, known as the attractor of the system, and after several thousand iterations, the shape should appear on the screen as well. An example of such a function system is shown in Figure 1 (Generator / Primitive), where the set of four functions (f_1 , f_2 , f_3 , and f_4) are used to generate a square primitive for later use. It is important to note that the square shown in the “Primitive” section can be thought of as a point cloud, where each point has been histogrammed or splatted on to some output layer and the colors indicate the function used on each individual point to land on its final position. Because each function is chosen at random, it is typically difficult for compilers to inline these functions without calling them as function pointers. The method proposed in this paper is the first to allow for additional transformations on IFSs, as shown in Figure 1 “Transformation” and “Output”. Other functional approaches to graphics [Duff 1992; Elliott 2003] exist with different advantages and drawbacks. These will be discussed in Section 2; however, all approaches are advantageous in that they allow for the creation of complex objects without a large amount of data stored on the Graphics Processing Unit (GPU) as the point cloud to be output to screen does not (necessarily) need to be stored in a global address space such as with GL_POINTS-based methods often used in photogrammetry and LiDAR [Leberl et al. 2010] as well as many other applications [Du et al. 2023; Farshian et al. 2023; Iglhaut et al. 2019]. This means that for many workflows, the only data needed to be stored in GPU global memory is the final output buffer of the image, itself, as well as a few bytes for necessary function arguments.

With the recent resurgence of splatting techniques (in particular Gaussian splatting) [Bao et al. 2025] and the relatively high prices of consumer memory, it is relevant to revisit such techniques as a possible methods to generate large scenes stored as point clouds. Unfortunately, it is difficult to efficiently solve the chaos game on GPU hardware due to a large number of limitations, including warp divergence from random function selection [Howes and Thomas 2007], the usage of function pointers [Zhang et al. 2021], and atomic

operations for histogramming the final point location to the output buffer [Shams et al. 2007]. Though there exist several techniques to circumvent these issues [Schied et al. 2011], this work proposes another direction. Rather than building a single function system to be solved with the chaos game, we instead generalize the creation of different object primitives and allow for users to create a Composable Function System (CFS) framework to specify precisely how all the points in the final cloud are intended to move at later stages in the pipeline. A single step of an example workflow is shown in Figure 1, where an IFS is used for the generation of the starting square before another (notably non-affine) function is used to generate the final object. The initial step of generating the object primitive and shading it can be generalized to any approach. It is possible, for example, for a set of objects to be created by photogrammetry and shaded based on lighting in the scene. The user can then specify how they desire for the points to move on screen without the need to explicitly rig an object. It is also possible to first create an object primitive with an IFS (for example) and then shade it with another approach, such as signed distance fields [Friskens and Perry 2006; Jones et al. 2006], ray tracing [Glassner 1989; Slusallek et al. 2005], or other software rendering techniques [Laine and Karras 2011]. In this way, CFSs are highly inter-operable with other, existing methods in the literature.

This paper is organized in the following way. In Section 2, we discuss several related methods to CFSs. In Section 3 we provide a simple example of function composability as well as the software interface (Section 3.1) and a motivating example (Section 3.2). Finally, we conclude and discuss future work in Section 4.

2 Related work

IFSs and related approaches have been an active field of research interest for decades, with many methods showcasing their versatility and usability in various aspects. As such, CFSs build upon and extend several other existing techniques in the literature. For example, in [Kunze et al. 2008], the authors propose extending IFSs to a new technique they call the Iterated Multifunction System, which allows for greater flexibility in the creation of fractal-like objects; however, their approach is limited in terms of allowed mappings. The fractal flame algorithm [Draves and Reckase 2008] is also similar in that it allows for users to inject functions into a pre-existing IFS structure. Several other approaches limit themselves to affine maps [Husain et al. 2022; Hutchinson 1981; Patiño Ortiz et al. 2023], where no such limitations exist for this work. There are also methods for rendering that do not necessarily use IFSs, and instead use function systems more generally [Elliott 2003] or with recursive methods [Duff 1992; Keeter 2020]. In contrast to all of these methods, this work focuses on building a metaprogramming framework to allow users to flexibly mix and match their preferred methods for composing functional units and create general-purpose (non-fractal) artwork, though we focus on IFSs for the generation of object primitives.

For general-purpose rendering, CFSs must also contrast themselves to other methods such as meshes, ray tracing, and raymarching with signed distance fields. In contrast to traditional mesh-based approaches, this work does not necessitate the storing of any vertex points or textures in memory and is capable of generating

n -dimensional scenes on-the-fly to be splat on to screen. Objects and object locations do not necessarily need to be stored in memory either and can instead be directly embedded into the function systems, themselves. In this way, it is similar to approaches using work-graphs and mesh shaders for realtime rendering [Kuth et al. 2024, 2025]. If users wish to use mesh-based methods, it is possible to do so by generating triangles and creating the mesh through software rendering approaches [Laine and Karras 2011]; however, the software implementation of CFSs written for this work primarily relies on computational kernels (acting as shaders) and lacks certain hardware features like rasterization and raytracing. It is also possible for users to implement raytracing and raymarching methods as shaders to objects created with CFSs to be manipulated by (for example) non-affine transformations. This is particularly interesting in that it extends the capabilities of raymarching with signed distance fields by enabling users to create complex smear effects without needing to march through non-Euclidean spaces [Friskens et al. 2000; Seyb et al. 2019].

Finally, as this work introduces Quibble, a GPU metaprogramming framework, it is also important to differentiate it from other, existing graphical and computational programming interfaces. At the current time, there are few interfaces that allow for flexible and efficient calling of function pointers on the GPU by default [Zhang et al. 2021]. This means that implementing generic, user-defined function calls on the GPU can be troublesome for many applications. Quibble exists as a metaprogramming framework to solve this problem by taking advantage of features from both graphical and computational workflows. Unlike graphical interfaces, like Vulkan, OpenGL, DirectX, and others, Quibble focuses primarily on the creation of high-quality computational kernels instead of the traditional graphical pipeline with (for example) vertex and fragment shaders. Unlike computational interfaces, like CUDA [Sanders and Kandrot 2010], KernelAbstractions.jl [Churavy 2024], SyCL [Reyes and Lomüller 2016], and kokkos [Edwards et al. 2014], Quibble instead focuses on compiling a large, custom kernel from user-provided functions at runtime. Though several modern languages, such as Julia, Rust, and C++ provide flexible metaprogramming capabilities, it is difficult to use such tools directly for this application, as we instead need the staged compilation approach commonly found in graphical languages like Vulkan [Sellers and Kessenich 2016] and OpenGL [Neider et al. 1993]. This feature is also present in OpenCL [Munshi 2009], which Quibble transpiles to. For this reason, Quibble is more similar to kernel fusion frameworks [Alomairy et al. 2024; Yadav et al. 2025].

3 Brief introduction to composable function systems for rendering

Simply put, a function system is a set of functions with defined inputs and outputs. In the context of rendering, the functions in the system must somehow color pixel coordinates on an output image. The simplest strategy to accomplish this with traditional software (such as OpenGL or Vulkan) might be to create a fragment shader on some quad spanning the size of the desired image; however, much more complex functions may also exist. Signed distance fields, for example, are used with raymarching to create complex scenes. In

the case of IFSs, each function (f_n) maps some position ($P_n(x, y)$) on to some new position ($P_{n+1}(x, y)$). Most IFSs are solved via the chaos game, which starts with a single point randomly chosen in real space. Every iteration of the chaos game, a new function is chosen at random and applied to that point. After, the point is then splat or histogrammed on to screen. One core advantage of creating objects by IFSs is that there is no need to keep any of the constituent points in global memory on the GPU.

CFS frameworks, on the other hand, do not have any rigorously defined strategy for function solving and are instead intended to allow users to mix and match different rendering methods much more efficiently. An example of this is shown in Figure 1, where the initial object is generated by an IFS, shaded by a method similar to a fragment shader, and then later manipulated by an arbitrary spatial function. By providing users the flexibility to choose their preferred rendering and shading method as well as the ability to choose the stage of the rendering pipeline when each function is called, users have unprecedented control over the entire rendering pipeline and can create complex objects without the need to model anything by hand. This allows for users to create more complex procedural visualizations or effects on non-procedurally created objects and scenes.

As an example, consider the creation of a unit circle centered about the origin with a non-affine IFS:

$$\begin{aligned}
 r &= \sqrt{x^2 + y^2} \\
 \theta &= \text{atan}(y/x) \\
 \theta_2 &= \pi(r^2 + f_{id}) \\
 r_2 &= \sqrt{\frac{\theta}{2\pi}} \\
 x_2 &= r^2 \cos(\theta_2) \\
 y_2 &= r^2 \sin(\theta_2)
 \end{aligned} \tag{1}$$

Here, x and y are the positions of the point, r is the distance from the origin, and θ is its corresponding angle. The values x_2 , y_2 , r_2 , and θ_2 represent the values of the next point in the iteration and f_{id} represents either 0 or 1 depending on which function is chosen. This function system can be interpreted as a map from the original circle on to either the top or bottom half of itself. This is shown in Figure 2(A).

Now imagine that the user would like to transform the circle by stretching it and squeezing in the center with the following transformation:

$$\begin{aligned}
 x_2 &= 2x \\
 y_2 &= y - \cos(x)
 \end{aligned} \tag{2}$$

In this case, the user would likely desire the image shown in Figure 2(B). The simplest way to generate this would be to first generate all the points in the circle, then save those to some array in global memory, and then transform the points separately; however, this might lead to the unnecessary storage of potentially millions of points. Such a method comes at a high memory cost and should

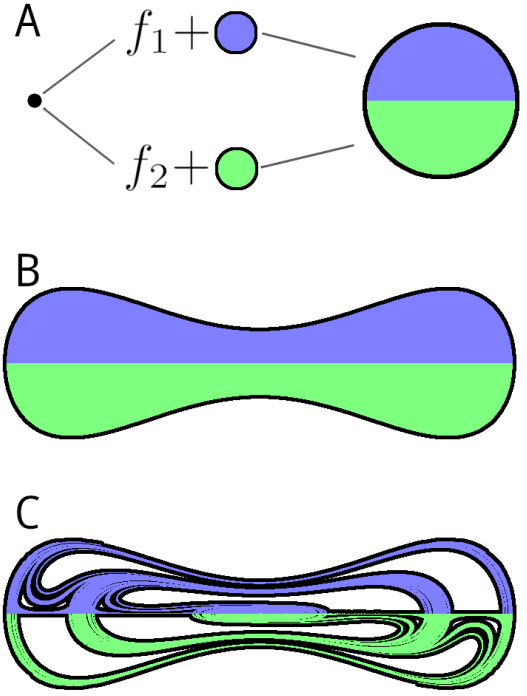


Fig. 2. An example of how different methods of function composition can lead to different outcomes. A is the generation of a circle with the equations provided in the text. In B, we use a copy of the point used for generating the primitive before proceeding with additional transformations. In C, we reuse the same point instead.

be avoided, but allows for similar staging to traditional mesh-based methods.

Instead, we should incorporate the transformation function into iteration step. This can lead to two different results depending on how it is composed with the function set 1. If the user re-uses the point used to generate the circle and outputs that to screen, it will result in Figure 2(C); however, if the user creates a copy of the point and instead transforms that, then they can still reuse their original point for generating the next point in the circle (or other arbitrary primitive) while also allowing for arbitrary transformations before splatting to screen and create the desired image in Figure 2(B). (C) is more closely related to methods like fractal flames [Draves and Reckase 2008], while (B) is more likely what the user would expect and is similar to [Elliott 2003]. A more complex example will be discussed in Section 3.2 in the generation of Figure 4; however, it should be clear that additional flexibility in the rendering pipeline allows for more complex imagery to be created.

3.1 API overview and motivating examples

Quibble is intentionally simple and introduces a new set of kernel semantics called *scribbles*, which are interoperable with OpenCL kernel syntax. This means that scribbles are written in simple C with

```

1  __verse mult_by(__global float *verse_a |
2  float factor = 8;){
3  verse_a[_idx] *= factor;
4  }
5
6  __stanza stanza_check(__global float *stanza_a,
7  __global float *stanza_b,
8  __global float *stanza_c |
9  float r = 1.0;){
10 if (_idx < array_size){
11 stanza_c[_idx] = stanza_a[_idx] + stanza_b[_idx];
12 __split_stanza();
13 stanza_c[_idx] += 1;
14 }
15 }
16
17 __poem poem_check(__global float *a,
18 __global float *b,
19 __global float *c,
20 int stanza_num,
21 int array_size){
22 if (stanza_num == 0){
23 @SCALL stanza_check(a, b, c | r = 5;){
24 @VCALL mult_by(c | factor = 2;);
25 }
26 if (_idx < array_size){
27 @VCALL mult_by(c | factor = 0.5;);
28 }
29 }
30 }

```

```

1  __kernel void poem_check(__global float * a,
2  __global float * b,
3  __global float * c,
4  int stanza_num,
5  int array_size){
6  int _idx = get_global_id(0);
7
8  if (stanza_num == 0){
9  {
10 __global float * stanza_a = a;
11 __global float * stanza_b = b;
12 __global float * stanza_c = c;
13 float r = 5;
14
15 if (_idx < array_size){
16 stanza_c[_idx] =
17 stanza_a[_idx] + stanza_b[_idx];
18
19 {
20 __global float *verse_a = c;
21 float factor = 2;
22
23 verse_a[_idx] *= factor;
24 }
25
26 stanza_c[_idx] += 1;
27 }
28 }
29
30 if (_idx < array_size){
31 {
32 __global float *verse_a = c;
33 float factor = 0.5;
34
35 verse_a[_idx] *= factor;
36 }
37 }
38 }
39 }

```

Fig. 3. This shows the compilation of a quibble scribble (left) into an OpenCL kernel (right) and showcases many features of the quibble kernel language. Here, the code is colored based on where it is defined within the quibble scribble. Red is from a poem, blue is from a stanza, and green is from a verse.

a few user-directed guidelines to more easily compose function systems together. This section is not intended to provide an exhaustive API for quibble, but is instead meant to showcase interesting and useful features that will later be expanded upon. Namely, Quibble introduces:

verses these are function-like objects that allow for users to express operations without compiling them immediately. verses can be called from within other scribble constructs with the @VCALL macro.

stanzas these are similar to verses, but allow for users to use an additional __split_stanza() function to split the block and inject additional operations in to the stanza. These additional operations may be verses or any other syntactically correct C code. stanzas can be called within poems with the @SCALL macro.

poems these are directives that compose verses and stanzas together into OpenCL kernels.

@include directives to easily compose different scribbles together by combining them in .qbl files, similar to those used in languages like Python and Julia.

Figure 3 shows an example of many of these scribble constructs and how they transpile into an OpenCL kernel. Both verses and stanzas allow for key-word arguments and may use variables defined from within the poem context from which they are called. Aside from these modifications, verses are similar in functionality to inlined functions. All scribble features can be compiled at the user's request in C by either calling on external .qbl files or in a single-source style by using the QBINLINE macro. By delaying compilation, users can inject complex transformations (with either functions or verses) into stanzas to compose function systems together and create complex and dynamic scenes. This functionality is essential to creating a CFS system.

Quibble provides several data structures to work with point clouds, colors, and images, as well as their corresponding data structures in the C api. There are also pre-made scribbles for standard operations users might request, such as file input and output, random

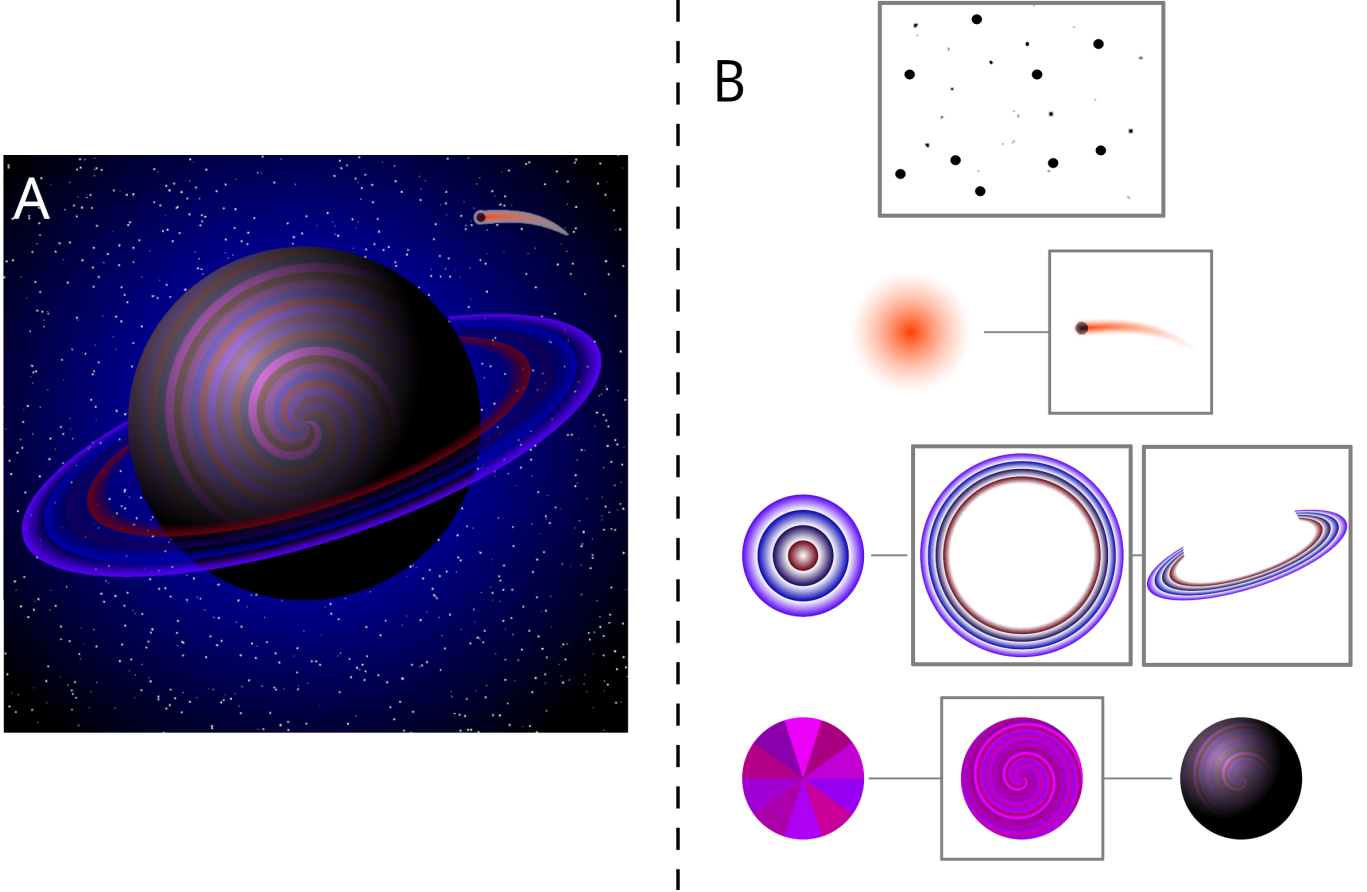


Fig. 4. A simple render designed with Quibble that showcases the capabilities of the current software and interesting features of the algorithm. A is a render of a planet with rings, stars, and a comet. B shows all the transformations necessary from the generated object primitive to create the final image. All objects highlighted in gray boxes are of spatial transformations.

number generation, and the creation of several object primitives like triangles, circles, and squares with IFSs. From C, Quibble provides a more flexible user-interface than OpenCL with helper functions to (for example) more easily set arguments, transfer data to and from the device, and run programs. Full examples of the C code and corresponding quibble scribbles for generating shaders, fractals, and smears are all available in the documentation [Schloss 2026]. Now is a good time to introduce a motivating example as to why these language constructs are helpful.

3.2 Multiple objects from a single primitive

Non-photorealistic rendering with mathematical expressions is a popular area of commercial and hobby interest, but software frameworks for such activities are often limited in their design and usability. Vector-based libraries (such as manim [Sanderson 2026; Zhang 2025]) are often difficult to parallelize and are non performant for real-time applications. Shader-based libraries (such as ShaderToy [Quilez and Jeremias 2017]) require a deeper understanding of the graphical pipeline, which is not necessary for visualizations

made through mathematical expressions. In addition, the mathematics necessary for shader-based approaches is often unintuitive, as in the case of signed distance fields. CFSs are in a unique position in that the transformations for each object primitive can be both easily parallelized and simply understood based on the user’s preference. With CFSs, it is possible to create a point cloud representing single object (such as a square, triangle, or circle) on-the-fly and re-use the points from this object to create many other objects in a scene without allocations in global memory beyond the final output image.

One example of this is shown in Figure 4. The primitive was constructed with an IFS chaos game for the circle, shown in Figure 2(A). After moving a point to the next position in the chaos game, but before writing to screen, we repositioned that point draw the stars, rings, planet, and comet (all shown in Figure 4(B)).

Each of the objects in the final image showcase an interesting transformation that can be challenging for traditional methods. The stars show duplication, where all star locations can be generated on-the-fly with a simple PRNG scheme and do not require explicit allocation on individual threads. The comet shows a somewhat

non-trivial smear. The planet shows how users can use spatial manipulations to create complex textures as well, effectively mixing the capabilities of both fragment and vertex shaders. Finally, the rings show the creation topologically non-trivial object, which can be difficult to do with mesh or geometry shaders.

The approximate runtimes for generating Figure 4 on consumer hardware (AMD 6700 XT) are approximately 0.0215s (average of 1024 runs) or 45 frames per second, with the wall time for the generation of a single image at approximately 0.745s (average of 1024 runs). Here, we note that this is using a naive chaos game implementation and the run time has the potential to be sped up significantly with more advanced methods. Currently, the compilation time comprises the largest portion of the total wall time for the generation of a single image. Though compilation is slow, it is still possible to consider on-the-fly compilation for certain situations in live environments.

Overall, this example shows how it is possible to create complex scenes from relatively simple object primitives while allocating minimal memory on each thread. In this case, there were less than ten floating-point values stored per thread and no additional data allocated on global memory except for what was required of the final output image (1920 × 1080 RGBA8888 values or roughly 16 MiB).

4 Conclusion

In this paper, we have introduced a new technique for the creation of general-purpose animations with composable function systems (CFSs) alongside Quibble, a GPU-accelerated metaprogramming framework for this purpose. We have provided some motivating examples (Figure 2 and figure 4) that highlight the utility of this work. This technique has the ability to produce interesting, non-affine, and even topologically non-trivial transformations on any object defined in a scene and is therefore particularly useful for smear frame and in-between generation for rendering methods where such techniques are otherwise difficult. In the future, it will be interesting to combine this work with other point-based rendering methods used in (for example) photogrammetry to store n -dimensional point clouds as function systems for later manipulation in software.

Even so, there several core limitations of this work. For example, even though the generation of stand-alone images (without accounting for file output) is fast enough to consider this method as a promising candidate for real-time graphical work, Quibble currently transpiles to OpenCL which is inconvenient for game development or other real-time, interactive experiences. In addition, rendering environments with density estimation, such as with fractal flame generation and caustics, still require atomic operations, which can also limit performance. Similarly, this method often requires users to over-populate the point cloud representing the object primitive to account for transformations of that object with additional functions. Because the function systems are all well-defined, it should be possible to automatically generate the appropriate initial distribution of points in an object primitive to account for all possible transformations with tools similar to autodifferentiation [Moses and Churavy 2020]. All of these are limitations on the overall performance of the method and are topics of further investigation.

Acknowledgments

This work will be submitted to the *NEDO Challenge, Quantum Computing “Solve Social Issues !”* for the C-9 category *Provision of New Rendering Environment* and has not received direct financial funding. The work has also been developed alongside the LeiosLabs community of passionate programmers on twitch, youtube, github, etc. I would like to thank Dr. Valentin Churavy for continued support and discussion regarding Julia, LLVM, Autodifferentiation and more.

References

- Rabab Alomairy, Felipe Tome, Julian Samaroo, and Alan Edelman. 2024. Dynamic task scheduling with data dependency awareness using julia. In *2024 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.
- Yanqi Bao, Tianyu Ding, Jing Huo, Yaoli Liu, Yuxin Li, Wenbin Li, Yang Gao, and Jiebo Luo. 2025. 3d gaussian splatting: Survey, technologies, challenges, and opportunities. *IEEE Transactions on Circuits and Systems for Video Technology* (2025).
- Michael Barnsley, John E Hutchinson, and Örjan Veenflo. 2003. V-variable fractals and superfractals. *arXiv preprint math/0312314* (2003).
- Michael F Barnsley and Stephen Demko. 1985. Iterated function systems and the global construction of fractals. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences* 399, 1817 (1985), 243–275.
- Michael F Barnsley, John H Elton, and Douglas P Hardin. 1989. Recurrent iterated function systems. *Constructive approximation* 5, 1 (1989), 3–31.
- Michael F Barnsley and Andrew Vince. 2011. The chaos game on a general iterated function system. *Ergodic theory and dynamical systems* 31, 4 (2011), 1073–1079.
- Mohamed Aly Bouke. 2025. Fractal attractors in random nonlinear iterated function systems: Existence, stability, and dimensional properties. *arXiv preprint arXiv:2505.18849* (2025).
- Valentin Churavy. 2024. KernelAbstractions. jl. *Zenodo* (2024).
- Persi Diaconis and David Freedman. 1999. Iterated random functions. *SIAM review* 41, 1 (1999), 45–76.
- Scott Draves and Erik Reckase. 2008. The fractal flame algorithm. *Citeseerx. Recuperado de http://citeseerx.ist.psu.edu/viewdoc/summary* (2008).
- Yan-Liang Du, Ting-Hua Yi, Xiao-Jun Li, Xiao-Li Rong, Long-Jun Dong, Da-Wei Wang, Yang Gao, and Zhen Leng. 2023. Advances in intellectualization of transportation infrastructures. *Engineering* 24 (2023), 239–252.
- Tom Duff. 1992. Interval arithmetic recursive subdivision for implicit functions and constructive solid geometry. *ACM SIGGRAPH computer graphics* 26, 2 (1992), 131–138.
- H Carter Edwards, Christian R Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of parallel and distributed computing* 74, 12 (2014), 3202–3216.
- Conal Elliott. 2003. Functional images. *The Fun of Programming, “Cornerstones of Computing” series*. Palgrave (2003).
- Anis Farshian, Markus Götz, Gabriele Cavallaro, Charlotte Debus, Matthias Nießner, Jón Atli Benediktsson, and Achim Streit. 2023. Deep-Learning-Based 3-D Surface Reconstruction—A Survey. *Proc. IEEE* 111, 11 (2023), 1464–1501. doi:10.1109/JPROC.2023.3321433
- Yuval Fisher. 1994. Fractal image compression. *Fractals* 2, 03 (1994), 347–361.
- Sarah F Frisken and Ronald N Perry. 2006. Designing with distance fields. *ACM SIGGRAPH 2006 Courses* (2006), 60–66.
- Sarah F Frisken, Ronald N Perry, Alyn P Rockwood, and Thouis R Jones. 2000. Adaptively sampled distance fields: A general representation of shape for computer graphics. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. 249–254.
- Ramen Ghosh and Jakub Marecek. 2022. Iterated function systems: A comprehensive survey. *arXiv preprint arXiv:2211.14661* (2022).
- Andrew S Glassner. 1989. *An introduction to ray tracing*. Morgan Kaufmann.
- Lee Howes and David Thomas. 2007. Efficient random number generation and application using CUDA. *GPU gems* 3 (2007), 805–830.
- Akhlaq Husain, Manikyala Navaneeth Nanda, Movva Sitaram Chowdary, and Mohammad Sajid. 2022. Fractals: An eclectic survey, part II. *Fractal and Fractional* 6, 7 (2022), 379.
- John E Hutchinson. 1981. Fractals and self similarity. *Indiana University Mathematics Journal* 30, 5 (1981), 713–747.
- Jakob Iglhaut, Carlos Cabo, Stefano Puliti, Livia Piermattei, James O’Connor, and Jacqueline Rosette. 2019. Structure from motion photogrammetry in forestry: A review. *Current Forestry Reports* 5, 3 (2019), 155–168.
- Marius Iosifescu. 2009. *Iterated function systems: A critical survey*. Univ.
- Mark W Jones, J Andreas Baerentzen, and Milos Sramek. 2006. 3D distance fields: A survey of techniques and applications. *IEEE Transactions on visualization and Computer Graphics* 12, 4 (2006), 581–599.

- Matthew J Keeter. 2020. Massively parallel rendering of complex closed-form implicit surfaces. *ACM Transactions on Graphics (TOG)* 39, 4 (2020), 141–1.
- John Kominek. 1995. Algorithm for fast fractal image compression. In *Digital Video Compression: Algorithms and Technologies 1995*, Vol. 2419. SPIE, 296–305.
- H Kunze, Davide La Torre, Edward R Vrsay, et al. 2008. From iterated function systems to iterated multifunction systems. *Commun Appl Nonlinear Anal* 15, 4 (2008), 1–15.
- Bastian Kuth, Max Oberberger, Carsten Faber, Dominik Baumeister, Matthäus Chajdas, and Quirin Meyer. 2024. Real-time procedural generation with GPU work graphs. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 7, 3 (2024), 1–16.
- Bastian Kuth, Max Oberberger, Carsten Faber, Pirmin Pfeifer, Seyedmasih Tabaei, Dominik Baumeister, and Quirin Meyer. 2025. Real-Time GPU Tree Generation. *Proceedings of High-Performance Graphics (HPG)* (2025).
- Samuli Laine and Tero Karras. 2011. High-performance software rasterization on GPUs. In *proceedings of the acm siggraph symposium on high performance graphics*. 79–88.
- Franz Leberl, Arnold Irschara, Thomas Pock, Philipp Meixner, Michael Gruber, Set Scholz, and Alexander Wiechert. 2010. Point clouds. *Photogrammetric Engineering & Remote Sensing* 76, 10 (2010), 1123–1134.
- William Moses and Valentin Churavy. 2020. Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients. *Advances in neural information processing systems* 33 (2020), 12472–12485.
- Aaftab Munshi. 2009. The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE, 1–314.
- Jackie Neider, Tom Davis, and Mason Woo. 1993. *OpenGL programming guide*. Vol. 478. Addison-Wesley Reading, MA.
- Julián Patiño Ortiz, Miguel Patiño Ortiz, Miguel-Ángel Martínez-Cruz, and Alexander S Balankin. 2023. A brief survey of paradigmatic fractals from a topological perspective. *Fractal and Fractional* 7, 8 (2023), 597.
- Inigo Quilez and Pol Jeremias. 2017. Shadertoy. Retrieved March 27 (2017), 2017.
- Ruyman Reyes and Victor Lomüller. 2016. SYCL: Single-source C++ accelerator programming. In *Parallel Computing: On the Road to Exascale*. IOS Press, 673–682.
- Jason Sanders and Edward Kandrot. 2010. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional.
- Grant Sanderson. 2026. Manim – Mathematical Animation Framework. <https://docs.manim.community/en/stable/>.
- Christoph Schied, Johannes Hanika, Holger Dammertz, and Hendrik P A Lensch. 2011. High-performance iterated function systems. In *GPU Computing Gems Emerald Edition*. Elsevier, 263–273.
- James Schloss. 2026. Quibble Docs. <http://www.leioslabs.com/quibble/>.
- Graham Sellers and John Kessenich. 2016. *Vulkan programming guide: The official guide to learning vulkan*. Addison-Wesley Professional.
- Dario Seyb, Alec Jacobson, Derek Nowrouzezahrai, and Wojciech Jarosz. 2019. Non-linear sphere tracing for rendering deformed signed distance fields. *ACM Transactions on Graphics* 38, 6 (2019).
- Ramtin Shams, RA Kennedy, et al. 2007. Efficient histogram algorithms for NVIDIA CUDA compatible devices. In *Proc. Int. Conf. on Signal Processing and Communications Systems (ICSPCS)*. 418–422.
- Philipp Slusallek, Peter Shirley, William Mark, Gordon Stoll, and Ingo Wald. 2005. Introduction to real-time ray tracing. In *ACM SIGGRAPH 2005 Courses*. 1–es.
- Rohan Yadav, Shiv Sundram, Wonchan Lee, Michael Garland, Michael Bauer, Alex Aiken, and Fredrik Kjolstad. 2025. Composing Distributed Computations Through Task and Kernel Fusion. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 182–197.
- Christina Zhang. 2025. Manim for STEM Education: Visualizing Complex Problems Through Animation. *arXiv preprint arXiv:2510.01187* (2025).
- Mengchi Zhang, Ahmad Alawneh, and Timothy G Rogers. 2021. Judging a type by its pointer: optimizing GPU virtual functions. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 241–254.