# A Massive Fractal in Days, Not Years

Theodore Kim
Yale University

Tom Duff
Pixar Animation Studios

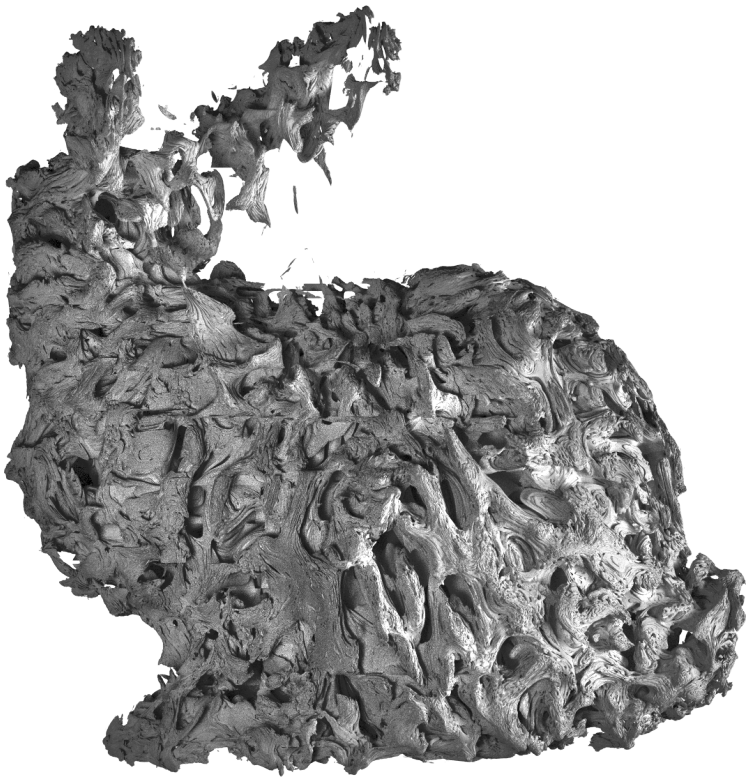

**Figure 1**. Path-traced rendering of a quaternion Julia set in the shape of the Stanford Bunny containing 10.48 billion triangles © Disney/Pixar.

## Abstract

We present a new, numerically stable algorithm that allows us to compute a previously-infeasible, fractalized Stanford Bunny composed of 10 billion triangles. Recent work [Kim 2015] showed that it is feasible to compute quaternion Julia sets that conform to any arbitrary shape. However, the scalability of the technique was limited because it used high-order rationals requiring 80 bits of precision. We address the sources of numerical difficulty and allow the same computation to be performed using 64 bits. Crucially, this enables computation on the GPU, and computing a 10-billion-triangle model now takes 17 days instead of 10 years. We show that the resulting mesh is useful as a test case for a distributed renderer.

 ISSN 2331-7418

## 1. Introduction

Quaternion Julia sets, 4D generalizations of Mandelbrot and Julia sets, were first discovered and popularized in the 1980s [Norton 1982; Hart et al. 1989]. However, it was never clear how to introduce high-level controls that generated anything but a limited set of taffy-like shapes, so research on the topic stalled.

Recently, Kim [2015] showed that an expressive set of high-level controls can be introduced by using quaternion rational functions. Using these, Julia sets can conform to any arbitrary shape while retaining their characteristic fractal appearance. Unfortunately, the rational functions involve very high-degree polynomials that approach the limits of floating-point computation. Consequently, the shapes had to be computed in 80-bit precision, which is an architecturally slow, non-optimized execution path. This is unfortunate, because much of the appeal of fractals is that details continually emerge at higher, more computationally-intensive resolutions.

In this work, we diagnose the numerical trouble and show how to perform the computations in 64-bit precision. We also obtain a negative result that suggests that a 32-bit version is infeasible under the current formulation. Computing Figure 1 would previously have taken approximately 10 years on a 12-core system, but our 64-bit algorithm allows the computations to be pushed to the GPU, and the shape was instead computed in 17 days. The result is a triangle mesh that contains non-trivial structures across four spatial orders of magnitude. Aside from the object's aesthetic appeal, we have also found it useful for testing algorithm scalability. The model does not fit into the memory of a typical Pixar render node, so we describe how we have found it useful as a non-trivial test for an experimental distributed renderer.

## 2. Julia Set Computation

The fractal generated is a *Julia set*, a close cousin of the Mandelbrot set. The algorithm is simple enough that it appears in introductory GPU texts [Sanders and Kandrot 2010] and in the CUDA SDK. In 3D, the bottleneck is the quaternion function evaluation that determines whether a point $\mathbf{q}$ lies inside the *filled* Julia set:

**function** INSIDEFILLEDJULIA($\mathbf{q}$)
  **while** $|\mathbf{q}| \leq$ maxRadius and $i <$ maxIterations **do**
    $\mathbf{q} = \mathcal{F}(\mathbf{q})$
    $i = i + 1$;
  **end while**
  **if** $i ==$ maxIterations **then**
    **return** False
  **end if**
  **return** True
**end function**

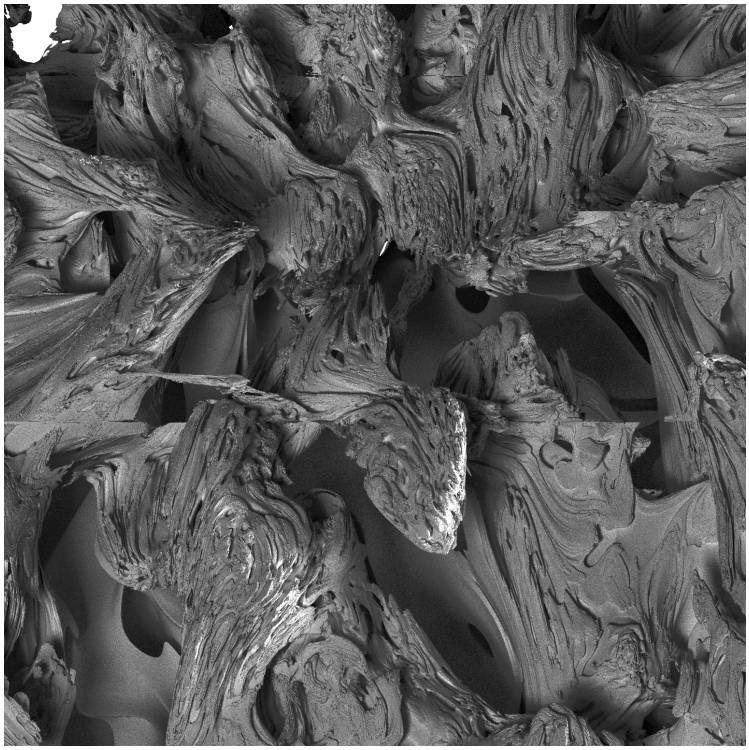

**Figure 2**. Detail from 10.48-billion-triangle fractal Stanford Bunny © Disney/Pixar.

In the code segment, $\mathbf{q}$ denotes a quaternion, and $\mathcal{F}(\mathbf{q})$ denotes an arbitrary (and expensive) function. The classic form is the quadratic function $\mathcal{F}(\mathbf{q}) = \mathbf{q}^2 + \mathbf{c}$, where $\mathbf{c}$ is some constant. However, Kim [2015] showed that Julia sets with arbitrary shapes can be obtained by using a power rational composed of $\mathcal{T}$op and $\mathcal{B}$ottom polynomials:

$$\mathcal{F}(\mathbf{q}) = \frac{\mathcal{T}(\mathbf{q})}{\mathcal{B}(\mathbf{q})} = \frac{(\mathbf{q}-\mathbf{t}_1)^{\alpha_1}(\mathbf{q}-\mathbf{t}_2)^{\alpha_2}\dots(\mathbf{q}-\mathbf{t}_T)^{\alpha_T}}{(\mathbf{q}-\mathbf{b}_1)^{\beta_1}(\mathbf{q}-\mathbf{b}_2)^{\beta_2}\dots(\mathbf{q}-\mathbf{b}_B)^{\beta_B}}. \tag{1}$$

In Equation (1), $\mathbf{t}_1 \dots \mathbf{t}_T$ and $\mathbf{b}_1 \dots \mathbf{b}_B$ denote quaternion-valued root locations, and $\alpha_1 \dots \alpha_T$ and $\beta_1 \dots \beta_B$ are scalars. Values that produce a tooth, a frog, and the Stanford Bunny were found in Kim [2015], and we re-use them here (Table 1).

Critically, $\mathcal{F}(\mathbf{q})$ had to be evaluated in *80-bit extended precision*, because `Infs` and `NaNs` otherwise appeared that polluted the final results. In order to accommo-

| **Example** | **Total Top Roots,** $T$ | **Total Bottom Roots,** $B$ | **Sum of Top Powers,** $\alpha_i$ | **Sum of Bottom Powers,** $\beta_i$ |
|---|---|---|---|---|
| Bunny | 178 | 153 | 251.998 | 89.3455 |
| Frog | 194 | 170 | 59.9387 | 33.2539 |
| Tooth | 65 | 43 | 372.339 | 127.73 |

**Table 1**. Polynomial degrees of the rationals used to generate several Julia sets.

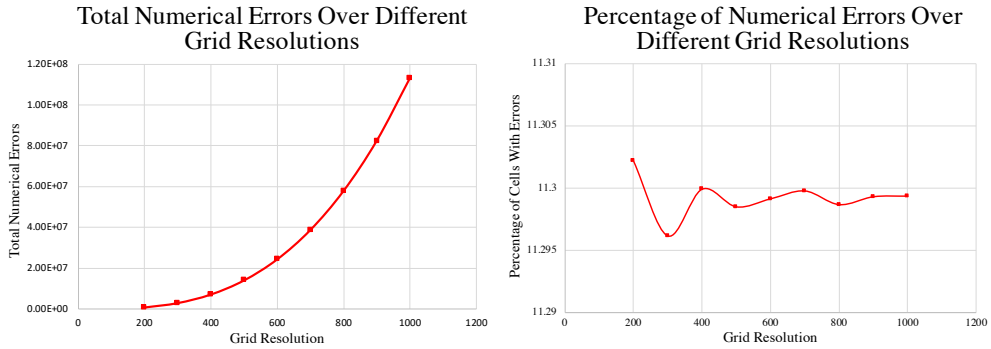


**Figure 3**. Left: Total `Infs` and `NaNs` produced as the grid resolution increases, using the original, error-producing algorithm and 64 bits. Right: Percentage of cells containing an `Inf` or `NaN`, which remains stable across grid resolutions at approximately $11.3\%$.

date this slower execution path, Kim [2015] ran across multiple cores using OpenMP. A CUDA implementation was not possible because GPUs only support up to 64 bits of precision. Over the regular grids that we used, numerical issues consistently appeared in a non-trivial percentage of the computations ($11.3\%$ of the cells, see Figure 3) for the Stanford Bunny model. While it would be possible to catch these errors by prolifically inserting `isinf` and `isnan` checks, it was not clear what to do once an error is detected.

The iterate could be oscillating wildly between the super-attracting fixed points at $\|\mathbf{q}\| = 0$ and $\|\mathbf{q}\| = \infty$, with $\mathcal{T}$ pulling it towards zero and $\mathcal{B}$ pulling it towards $\infty$. Thus, trapping the numerical issues still leaves Julia-set membership undetermined, and so we analyze this issue further in Section 3.

We follow the overall approach of Kim [2015], which evaluates $\mathcal{F}(\mathbf{q})$ over a high-resolution regular grid, and then performs Marching Cubes to extract the final triangles. Traditional Marching Cubes assumes that the level-set values vary linearly between grid cells, uses linear interpolation to find the root location between cells, and converts the root to a vertex. In our case, the values are dramatically non-linear between grid cells, so linear interpolation tends to snap vertices to cell centers or edges, which creates an incorrect and unacceptably blocky result.

Instead, Kim [2015] uses a non-linear version of Marching Cubes, which executes a midpoint search along each edge between cell centers, until a root is located (Figure 4). This search only occurs along edges that exhibit a sign flip of $\log\left(\|\mathcal{F}(\mathbf{q})\|_2\right)$, so a root is guaranteed to exist along the interval. The midpoint search repeatedly evaluates $\mathcal{F}(\mathbf{q})$, so the computation must be both fast and robust. Alternatives, such as the approach of Hart [1989], cannot be applied, because there is no known generalization to the quadratic "distance estimator" that is needed by that algorithm.

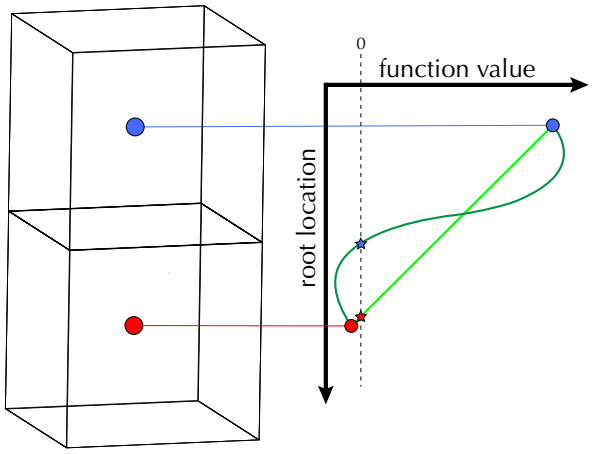


**Figure 4.** Marching Cubes locates roots by assuming that the function varies linearly between cells (light green curve). If the function is highly non-linear, this assumption can produce spurious roots near cell centers (red star). We perform a midpoint search over the actual non-linear function (dark green curve) and accurately locate the root (blue star).

## 3. Solving the Numerical Issues

### 3.1. Isolating The Problem

We have found that numerical trouble occurs in $\mathcal{F}(\mathbf{q})$ when the bottom polynomial, $\mathcal{B}(\mathbf{q})$, grows quickly and exceeds the available precision. This specifically occurs when computing (the $*$ denotes a quaternion conjugate)

$$\mathcal{B}(\mathbf{q})^{-1} = \frac{\mathcal{B}(\mathbf{q})^*}{\|\mathcal{B}(\mathbf{q})\|^2}$$

as a precursor to

$$\mathcal{F}(\mathbf{q}) = \frac{\mathcal{T}(\mathbf{q})}{\mathcal{B}(\mathbf{q})}.$$

When $\|\mathcal{B}(\mathbf{q})\|^2$ is computed, each component of $\mathbf{q}$ is squared, and the terms are then summed. If $\|\mathbf{q}\|$ was already very large, this computation doubles the polynomial degree of $\mathcal{B}(\mathbf{q})$, which triggers an `Inf`-type overflow that pollutes the rest of the computation.

The problem consistently appears with $\mathcal{B}(\mathbf{q})$ instead of $\mathcal{T}(\mathbf{q})$ because the algorithm does not require the magnitude of $\mathcal{T}(\mathbf{q})$ to be computed. The component-wise exponents of $\mathcal{T}(\mathbf{q})$ are not raised to an additional second power, and they are never summed into a single scalar .

### 3.2. The Solution

We have found that the following condition resolves the numerical issues. If $\|\mathcal{B}(\mathbf{q})\| > 10^{\tau}$, we halt computation at the moment that the iterate appears to be diverging to-

wards the super-attracting fixed point, $\|\mathbf{q}\| = \infty$, and conclude that this is the best estimate of set membership that can be determined within the available precision. The key is to define the constant

$$\tau = 308 - 1.05 \left( \log_{10}(\|\mathcal{B}(\mathbf{q})\|) + \sum_{i=1}^{T} \beta_i \right). \tag{2}$$

The $\tau$ constant estimates whether $\mathbf{q}$ has sufficient exponent bits remaining to support another full evaluation of $\mathcal{F}(\mathbf{q})$. First, it extracts the exponent of $\mathcal{B}(\mathbf{q})$ using $\log_{10}(\|\mathcal{B}(\mathbf{q})\|)$. Next, it estimates how the exponent will amplify in the next iteration by summing the exponents of $\mathcal{B}(\mathbf{q})$, using $\sum_{i=1}^{T} \beta_i$. The result is then subtracted from 308, the maximum possible exponent of a 64-bit, IEEE 754 floating-point number. To avoid skirting too close to the limit of exponent precision, a slight conservatism is introduced by scaling the sum by 1.05. More conservative scalings (e.g., 2.0) worked equally well in our experiments, but looser scalings (e.g., 1.01) consistently failed. The CUDA code that implements this solution is shown in Listing 1.

### 3.3. Validation and Discussion

We validated this formula across all available large-scale Julia set models: the Bunny, Tooth, and Frog examples from Kim [2015]. In all cases, the `Infs` and `NaNs` were successfully eliminated from double-precision computations, and the resulting output was identical to the 80-bit results up to working precision.

An additional experiment was performed to validate Equation (2). The Tooth model has the highest-degree bottom polynomial (Table 1). Substituting the summed $\beta_i$ from the lower-degree Bunny and Frog models re-introduced numerical issues. Thus, accounting for the summed $\beta_i$ of each model is clearly a necessary condition for robust computation.

This overall approach misleadingly suggests that an expression for 32-bit floats could be constructed. Following the same approach, the 308 in Equation (2) could be replaced with 38, the maximum 32-bit exponent. However, the summed polynomial degrees of $\mathcal{B}(\mathbf{q})$ for both the Bunny and Tooth models already exceed 38, so even the first iteration does not fit into the available precision. This explains why our previous attempts at performing computations with 32 bits consistently failed.

## 4. Additional Optimizations

We attempted a variety of orthogonal micro-optimizations, but found them to be either ineffective, or only mildly effective. Raising a quaternion to a scalar power, using a fused `sincos` function, resulted in an 8% speedup. Explicit loop unrolling yielded no speedups, because the CUDA compiler already performs them automatically. Demoting any portion of the computation to single precision altered the final results unacceptably, which is consistent with our findings with regard to Equation (2).

```
__device__ inline double nonlinearValue(const double3& center)
{
  double4 iterate = make_quaternion(center.x, center.y, center.z,
                                    quaternionSlice);
  double magnitude = magnitude4(iterate);
  int totalIterations = 0;

  double bottomPowerSum = bottomPowers[0];
  for (int x = 1; x < totalBottomRoots; x++)
    bottomPowerSum += bottomPowers[x];

  const double4 bail = make_double4(DBL_MAX, DBL_MAX,
                                    DBL_MAX, DBL_MAX);
  bool bailed = false;
  while (magnitude < escape && totalIterations < maxIterations)
  {
    const double4 topEval = evaluateTop(iterate);
    const double4 bottomEval = evaluateBottom(iterate);
    const double bottomMagnitudeSq = magnitudeSq4(bottomEval);
    const double bottomMagnitude = sqrt(bottomMagnitudeSq);

    // compute guard value from Eqn.  2
    const double rhs = bottomPowerSum +
                       log(bottomMagnitude) / log(10.0);
    const double topLimit = 308.0 - 1.05 * rhs;

    // if the division is tiny, bail
    if (bottomMagnitude < pow(10.0, topLimit))
    {
      const double4 bottomInv =
        conjugateScaled(bottomEval, 1.0 / bottomMagnitudeSq);
      multiply(topEval, bottomInv, iterate);
    }
    else
    {
      iterate = bail;
      bailed = true;
    }

    const double scaling = (!bailed) ? expScaling : 1.0;
    scale(iterate, scaling);
    magnitude = (!bailed) ? magnitude4(iterate) : DBL_MAX;
    totalIterations++;
  }

  return log(magnitude);
}
```

**Listing 1**. Iterating and evaluating Equation (1) in `NONLINEAR_SLICE_CUDA.cu` such that precision issues are detected and avoided.

The code was restructured to support a streaming architecture, because one of our basic assumptions is that the output will not fit in-core. The central Marching Cubes algorithm was modified so that only two $z$-slices of the underlying regular grid were ever held in memory simultaneously. The triangles for each slice were written to disk immediately after each slice computation completed. Paradoxically, the successful demotion of the algorithm to 64-bit floating point also meant that the vertex indices had to be promoted to 64-bit `int`s, because the number of vertices being generated now exceeded the addressing capacity of 32-bit `int`s.

## 5. Results and Discussion

### 5.1. Geometry Generation

With the described modifications in place, it became feasible to compute a fractal in 64-bit precision using CUDA on a NVIDIA Quadro M6000. The GPU yielded a greater than $200\times$ speedup, far exceeding the $10\times$ memory CPU / GPU bandwidth differential. The algorithm is clearly compute-bound. Over 99% of the time was spent in core arithmetic kernels, such as quaternion exponentiation. We ran the algorithm on a $11,500^3$ grid, which is two orders of magnitude larger than previously possible. The final data contained 10,483,747,635 triangles, 31,451,242,905 vertices, and consumed 676 GB on disk as an uncompressed OBJ.

### 5.2. Rendering the Geometry

We have found that this algorithm presents an interesting case study in rendering massive geometry. Now that we have the mesh, how do we even look at it? More generally, we now have a method for generating an arbitrarily challenging triangle mesh that exceeds the capacity of whatever rendering resources are available.

Such an algorithm is useful, because ever since *A Bug's Life* in 1998, there is usually at least one scene in each Pixar movie that exceeds the capacity of even the most memory-rich node in the render farm. The scene usually appears well after R&D has concluded, so ad hoc solutions must then be employed, e.g., manually trimming the scene until it barely fits in-core. Scaling algorithms can be tested on static models, such as the Stanford Bunny or Michelangelo's David, but while these were considered large for their time, they are not challenging by modern standards. Our algorithm allows a data set to be generated that exhibits non-trivial statistics compared to toy examples, e.g., 10 billion random triangles or a uniformly loop-subdivided Stanford Bunny, but it does not require any additional modeling effort from a user.

The current 10 billion triangle fractal is already a daunting scene. We generated images (Figures 1 and 2) using an experimental path tracer that distributes the scene data among several tracing servers and passes rays over a local TCP/IP network. Each

server is responsible for all geometry within a given convex region. The regions are non-overlapping and all occupy roughly equal amounts of storage. The servers all have copies of an index that describes each region and the address of its owner. Since the regions are convex and non-overlapping, a ray that enters a given region must exit it before encountering any other region and never re-enter any region. Upon determining that a ray does not hit any of its geometry, a server has all the information it needs to forward the ray to the next candidate server.

Path tracing requires that we accumulate the transmittances at each ray hit along the path. Following the Toro system [Pharr et al. 1997], we forward this accumulation with the rays, which precludes certain non-physical effects. For example, the color of a surface cannot change depending on the intensity of the light hitting it, since the server that processes the hit never gets called back after the path reaches the light. This is generally not a problem for scenes shaded using physically plausible surfaces.

Additionally, there are two servers that simulate an imaging system. One generates camera rays and forwards them to the appropriate tracing servers. The other is responsible for accumulating the output image, and it is the target to which results are sent by rays that hit a light source or otherwise exit the scene.

We used a network of 64 tracing servers to make the image of the 10 billion triangle bunny. The total memory occupancy among the servers was 2 TB—an average of 31.3 GB per server. The render took roughly 36 hours to complete. The time is almost completely determined by network latency as the servers typically ran at about 10% to 20% CPU utilization. (We intend to speed this up by bundling rays to reduce network overhead.) We do not view slowness as a fatal problem, because our conventional single-host renderers cannot render this image at all. They cannot fit the geometry in main memory, and the alternative of caching geometry from secondary storage is unrealistically expensive.

## 6. Conclusions and Future Work

The numerically stabilized algorithm we have presented suggests a variety of interesting directions.

- Equation (2) suggests that in order to preserve numerical stability, Julia sets with smaller summed polynomial degrees should be preferred. This condition could be added to the optimization, thus making it possible to discover Julia sets with a larger number of roots, which in turn conform to more complex shapes.

- Does the fractal bunny contains infinite bunny copies? Recent results in 2D [Lindsey and Younsi 2019] suggest that copies may exist, but exploring the fractal has previously been too difficult for the computational reasons that we have now resolved.

- We computed the number of triangles generated across a range of grid resolutions and estimated the fractal dimension of the bunny to be 2.1868. Do denser, more space-filling settings exist that provide better "bang-per-voxel"?
- Is there a different formulation under which a model of comparable quality can be computed in a numerically stable way using 32-bit precision?

Finally, can we generate the same (or bigger) dataset in a shorter amount of time? We have provided the CUDA source code so that others may make an attempt.

## References

HART, J. C., SANDIN, D. J., AND KAUFFMAN, L. H. 1989. Ray tracing deterministic 3-D fractals. In *Proceedings of SIGGRAPH*. ACM, New York, NY, USA, 289–296. URL: https://dl.acm.org/doi/10.1145/74334.74363. 27, 29

KIM, T. 2015. Quaternion Julia set shape optimization. *Comput. Graph. Forum 34*, 5 (Aug.), 167–176. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12705. 26, 27, 28, 29, 31, 35

LINDSEY, K., AND YOUNSI, M. 2019. Fekete polynomials and shapes of Julia sets. *Trans. Amer. Math. Soc 371*, 8489–8511. URL: https://www.ams.org/journals/tran/2019-371-12/S0002-9947-2019-07440-8/home.html. 34

NORTON, A. 1982. Generation and display of geometric fractals in 3-D. In *Proceedings of SIGGRAPH*. ACM, New York, NY, USA, 61–67. URL: https://doi.org/10.1145/800064.801263. 27

PHARR, M., KOLB, C., GERSHBEIN, R., AND HANRAHAN, P. 1997. Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of SIGGRAPH*. ACM, New York, NY, USA, 101–108. URL: https://doi.org/10.1145/258734.258791. 34

SANDERS, J., AND KANDROT, E. 2010. *CUDA by example: An introduction to general-purpose GPU programming*. Addison-Wesley Professional, Upper Saddle River, NJ, USA. URL: http://www.mat.unimi.it/users/sansotte/cuda/CUDA_by_Example.pdf. 27

## Index of Supplemental Materials

The complete C++ and CUDA source code needed to generate Figures 1 and 2 is provided at jcgt.org/published/0009/02/02/xxx.zip. This code has been successfully built and run on macOS 10.10 (Yosemite) and Red Hat Enterprise Server 7.6 (Maipo). For comparison, the numerically unstable version from Kim [2015] is also available online at http://www.tkim.graphics/JULIA/source.html.

## Author Contact Information

Theodore Kim
Yale University
Department of Computer Science
P.O. Box 208285
New Haven, CT 06520-8285
theodore.kim@yale.edu

Tom Duff
Pixar Animation Studios
1200 Park Avenue
Emeryville, CA 94608
td@pixar.com

Kim and Duff, A Massive Fractal in Days, Not Years, *Journal of Computer Graphics Techniques (JCGT)*, vol. 9, no. 2, 26–36, 2020
http://jcgt.org/published/0009/02/02/

| | | | |
|---|---|---|---|
| Received: | 2019-09-13 | | |
| Recommended: | 2020-02-18 | Corresponding Editor: | Bernd Bickel |
| Published: | 2020-06-16 | Editor-in-Chief: | Marc Olano |