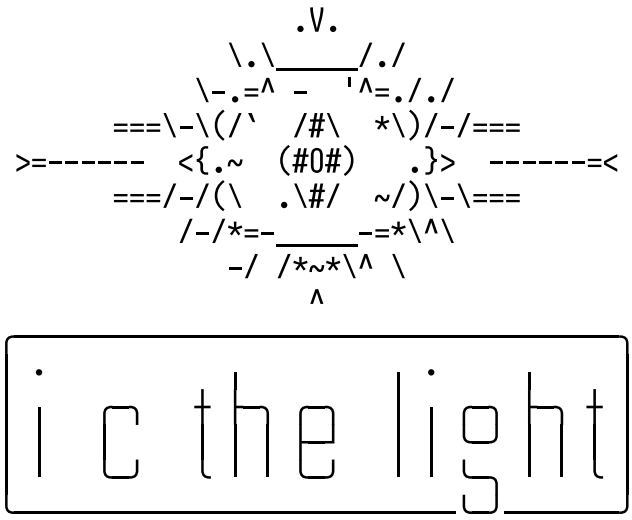

Figure 1: *I C the Light*'s start-up graphics, depicting a large single eye

I C the Light: Ray Marching My Dreams

Rebecca Turner[1]

We implement a ray-marching renderer using the distance-estimation algorithms found in papers by Hart et al.[2] and Crane[3] with an eye on details and specifics, intended for readers more unfamiliar with 3D vector graphics and related topics. Distance-estimating ray marchers operate by computing an *estimate* of the distance to the object being rendered, allowing complex or infinitely detailed mathematically-defined objects to be rendered. Github: `9999years/i-c-the-light`

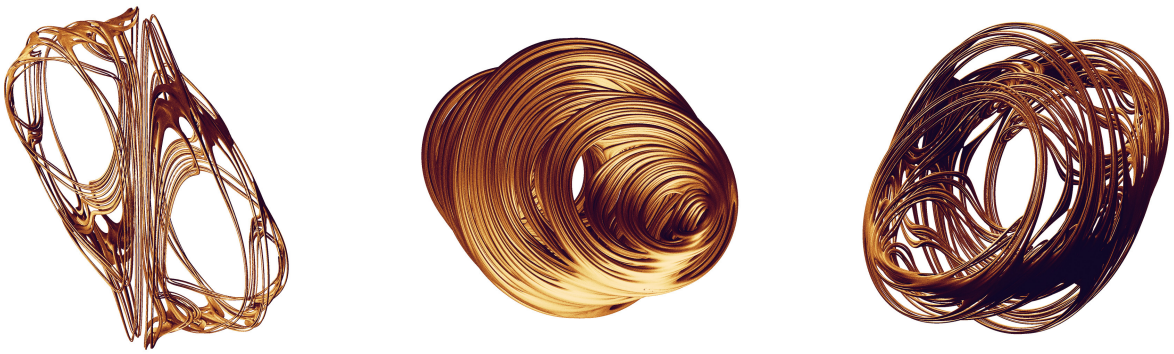

Figure 2: Renders of quaternion Julia sets produced by *I C the Light*

[1] Direct questions, comments, and concerns to `637275@gmail.com` or by other means as directed on `becca.ooo`

[2] Hart, J. C., Sandin, D. J., and Kauffman, L. H. (1989). *Ray Tracing Deterministic 3-D Fractals*. *SIGGRAPH Comput. Graph.* *23*(3), 289–296. doi:10.1145/74334.74363.

[3] Crane, K. (2005). *Ray Tracing Quaternion Julia Sets on the GPU*. University of Illinois at Urbana-Champaign.

Contents

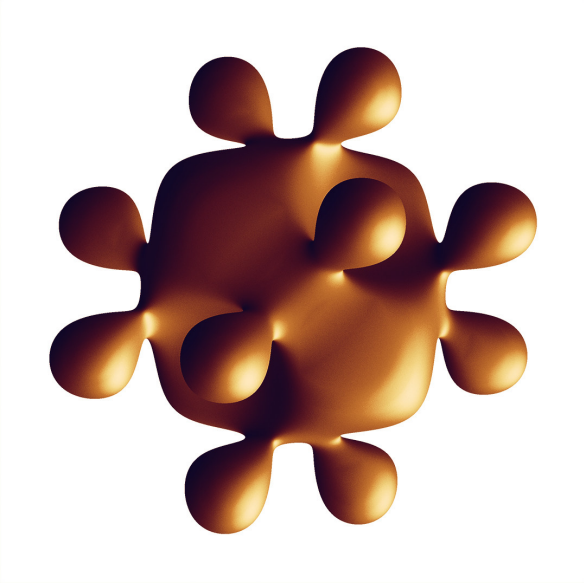

Figure 3: *I C the Light* render T1484281708 — a sinusoidally-displaced sphere.

1 How are 3D graphics generated?

How was the image in figure 3 made? How do computers translate scene information — geometry, colors, lighting, and so on — into an image?

Usually, computers create digital images (called *renders*) from scene information by simulating the universe — or at least a small part of it, in a simplified manner. Calculating the results of a fully-featured simulation of the interactions of every particle and wave bouncing throughout a virtual world might be interesting, but it would be too slow and over-involved[4] to be useful for anything practical. Instead, a simulation may consist of firing a burst of photons from a camera and measuring their interactions with the geometry.

Therein lies the rub — given an arbitrary ray (an infinite line originating from a point), how can its intersection with the *scene* (an arbitrary collection of geometry) be found?

Broadly speaking, there are two solutions. Ray *tracers* solve discrete equations to find a formula for the

[4] Could the reader imagine placing every speck of dirt by hand just to conjure some soil?

exact intersection between a primitive[5] and a scene — consequently, ray tracers are limited to rendering combinations of the shapes their authors solved equations for, usually spheres and triangles. Figure 4 shows a ray-traced image.

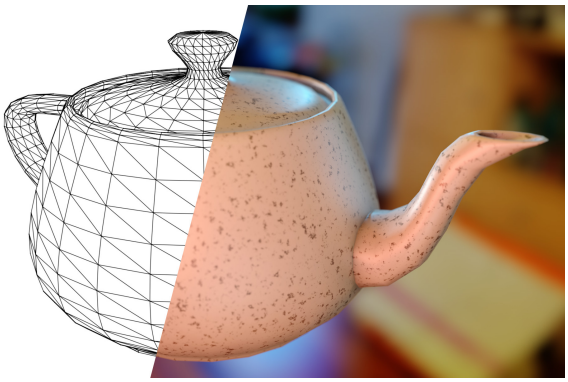

Figure 4: A ray-traced render of the Utah Teapot, a famous model created in 1975 by Martin Newell. On the left-hand side is a *wireframe* render, showing the individual triangles that are joined together to form a whole object. On the right, a shading model and several physical effects such as depth of field and ambient occlusion have been added to make the teapot's appearance more realistic.

Second and more interestingly exist *ray marchers*, a class of renderer that approximates the intersection of a scene and a ray by using a function called a *distance estimator*, or DE, that estimates a bound on the distance from a *point* to a scene. Then, by *marching* along the pre-determined ray in step sizes determined by the DE, an intersection between a ray and a scene may be approximated to arbitrary precision.[6]

Rather than calculate an exact intersection with a scene in $O(1)$ complexity,[7] like a ray tracer, ray marchers *estimate* the directionless distance to the closest point in the scene from any point in space in an unknown but bounded complexity $O(x)$, $x < M$, where M represents the maximum step count, a user-defined constant that determines the ray marcher's accuracy and execution time. The maximum step-count of a ray-marcher is analogous to the iteration count of a fractal;[8] increasing the step count decreases the error between the rendered image and the true geometry. A ray-marched render is shown in 5.

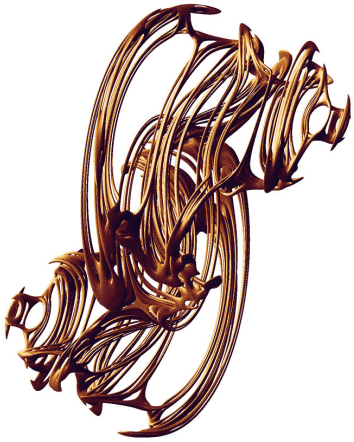

Figure 5: A ray-marched render of a quaternion Julia set (described in more detail in sections 5.3 and 6.3.3). The important difference between this image and a ray-traced render such as in figure 4 is that the geometry is not defined as a collection of triangles but with a mathematical formula ($\lim_{n\to\infty} q_n = q_{n-1}^2 + c \neq \infty$).

Ray marching does not estimate the distance to the scene along a ray, just the distance to the scene from a point — this is why it's called a *directionless* estimate. It's impossible to know from the distance estimate (DE) alone if the ray being travelled upon is pointed in the direction of the scene or not — until finished analysing any given ray, no information about its potential intersection(s) are known.

Due to the lack of directionality in the distance estimate, a ray to march along must be predetermined, and a complete image may be assembled by marching thousands or millions of rays in a grid. Around 250,000–10 mil-

5 The simplest form of a geometric object like a sphere or a single triangle.

6 Read: With accuracy proportional to the reader's patience for sitting around waiting for a computer to generate an image.

7 The *complexity class* of an algorithm, represented in "big-O" notation, describes the growth of the time the algorithm takes to complete with relation to its input.

8 What's an iteration count? See section 5 for an explanation.

lion rays[9] are fired to render a scene — one or more[10] per pixel.

2 Why ray march?

Additionally, due to the lack of an exact intersection, ray marchers can only *approximate* the boundary of a scene and therefore have an unknown complexity class.

As such, faster and simpler ray tracers are used whenever possible, delegating ray marchers to enthusiast and research circles.

Why, then, would anyone use the comparatively inefficient ray marching model? Generally, whenever defining a DE is easier than finding an intersection equation (almost always) or when the surface of an object is undefined, such as in infinitely complex fractals.

And as for the reverse — if finding a DE is easier than finding an intersection equation and ray marchers are capable of rendering objects and shapes that ray tracers are incapable of, why does anyone use ray tracers? First, ray marchers are much slower; as previously stated, ray tracers can calculate an intersection in $O(1)$ time while ray marchers take an unknown $O(x)$ time, and in cases where speed is critical (such as video games or animated movies) the extra effort to optimize inputs so that they are ray-traceable has massive speed benefits.[11] The second notable reason that ray marchers aren't more widespread is that they simply aren't that useful for every-day applications of 3D graphics — although the work of Benoît Mandelbrot and others in developing the theory of fractals (more on that in section 5) has been important in developing a deeper understanding of the natural world and creating digital approximations of it (as in animated movies), it's easier to adapt fractals to fit the triangle-and-sphere worlds of ray tracers than it is to convert entire rendering systems to incorporate ray marchers natively.

3 How does ray marching work?

Humans discover the shape of the spaces we inhabit through optical vision, where photons bouncing off of objects enter our eyes, and our retina sense the regions where photons cluster together to assemble a dark-and-light image of the portion of the world in front of us — a process very similar to ray tracing, where positions of photon collisions are directly measured to render an image. How, then, does a bat find its way through the world, blind[12] and alone?

Bats use a technique referred to as *echolocation*,[13] a system akin to SONAR,[14]. When echolocating, bats emit high-pitched tones and measure how long their echoes take to return, detecting the distance to the nearest ob-

[9] These are *normal* ray counts and not unusually large — movies like Pixar's *Finding Dory* (2016) use "billions of individual light rays per frame, with probably ten reflections and refractions in each ray."[a]

1 ray per pixel × 500 pixels wide × 500 pixels tall = 250,000 rays

4 rays per pixel × 1920 pixels wide × 1080 pixels tall = 10,000,000 rays

[10] Such as with SSAA (super-sample anti-aliasing), a technique that renders smoother images by rendering multiple points per pixel and averaging them to avoid jagged edges.

[11] And although progress has been made to make ray marchers *faster* by Crane, Hart et al., and others, ray marchers are still nowhere near as fast as ray tracers.

[12] Although "The extent to which bats rely on vision […] is unknown,"[b] it is clear that their vision is perfectly adequate, albeit "adapted for nocturnal vision"[c] and inferior to their echolocation for detecting things such as insects in the dark.[d]

[13] Layne, J. N. (1967). *Evidence For The Use Of Vision In Diurnal Orientation Of The Bat Myotis Austroriparius*. *Animal Behaviour*, *15*(4), 409–415. doi:10.1016/0003-3472(67)90037-1.

[14] Sound Navigation And Ranging

[a] Sciretta, P. (2016). *How Unprecedented New Technology Made It Harder to Produce 'Finding Dory'*. Retrieved from slashfilm.com/the-tech-of-finding-dory

[b] Boonman, A., Bar-On, Y., Yovel, Y., and Cvikel, N. (2013). *It's Not Black or White — On The Range of Vision and Echolocation in Echolocating Bats*. *Frontiers in Physiology*, *4*. doi:10.3389/fphys.2013.00248

[c] Ibid.

[d] Ibid.

jects without using their eyes.[15] Echolocation is like ray marching and the sonorous "pings" bats emit are like a distance estimator — bats cannot directly see the geometry of the spaces they inhabit, but can deduce it by listening to the timing of echoes.

Ray marching requires a function $D(\vec{p})$ that yields the directionless distance to the scene from a point represented by the position vector $\vec{p}$.

A distance estimation function can be as simple as $D(\vec{p}) = \|\vec{p}\| - r$ for a circle of radius r about the origin. Although the scene is outlined in accompanying figures, this is solely for clarity — ray marching is used to determine the locations of these boundaries, which are not known beforehand.

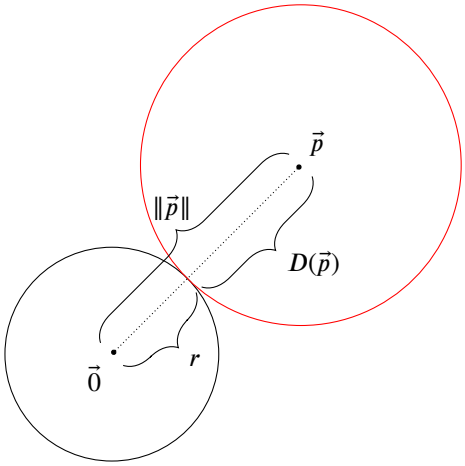

Figure 6: The geometric visualization of a simple de of a 2d sphere. Unbounding volumes in figures are shown in red and geometry is shown in black

First, a ray $\vec{r}$ to march along is chosen. We will attempt to find the intersection of this ray and the scene, defined as the set of points $\mathcal{S}$ that compose the object(s) to be rendered. Remember, in a ray marcher, the scene is defined by a series of equations, as contrasted to ray tracers, which define the scene as a list of points and their connections.

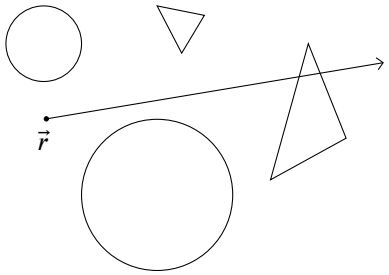

Figure 7: A ray $\vec{r}$ within a scene

Next, we place our "sampling position" $\vec{s}_o$ at the origin of $\vec{r}$, we may sample $D(\vec{s}_0)$ and draw an n-sphere[16] of radius $D(\vec{s}_0)$. Hart refers to these n-spheres as "unbounding volumes" because they are a bound on the space *not* inhabited by the scene. Because we know that the scene is present nowhere inside the bounding volume, we may move our sampling position distance $D(\vec{s}_0)$ along $\vec{r}$ to find a new sampling point that is equally guaranteed to not be inside of the scene. If $\vec{r}$ is pointed towards the scene, $D(\vec{s}_n)$ will decrease as we "march" along $\vec{r}$, until we reach an acceptably small value of $D(\vec{s}_n)$, at which point we can declare the scene to be found at point $\vec{s}_n$ and the process can be restarted with the next ray. If the ray *isn't* pointed towards the set, $D(\vec{s}_n)$ will eventually exceed a maximum allowable value $\vec{s}_{\max}$.

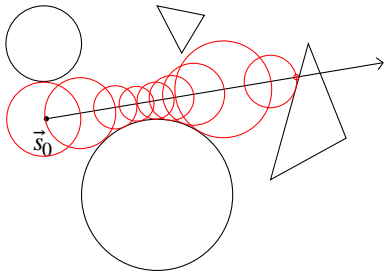

Figure 8: A ray $\vec{r}$ within a scene and the unbounding volumes used to find its intersection with the scene

By repeating this process many times with many different rays we may assemble a "point cloud" of the scene, and by choosing enough rays we may render a complete and contiguous image.

[15] This is, like any analogy, a simplification; the notable advantage bats have is the possession of two ears, angled outward. Much like humans, hearing in stereo (in two different directions with two different listening devices) allows bats to discern the *directionality* of sounds, information they use in tandem with the echo delays, and, yes, their eyes, to gain an extremely precise awareness of their surroundings.[e]

[e] Ibid.

[16] Where n corresponds to the dimensionality of the render space.

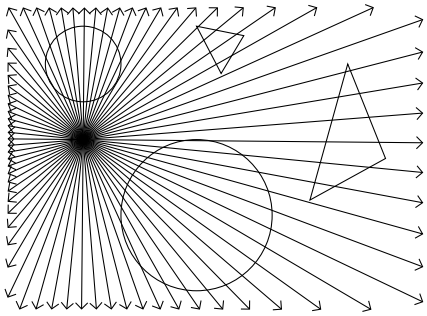

Figure 9: Rays within a scene — the combination of all the intersections between the rays and the scene comprises the data used by the shading model to turn the series of equations into a visual image. Note that only the *first* intersection between a ray and the geometry will be logged and used for the render — as the back sides of objects in a scene are occluded (not visible), they may be safely ignored.

4 Previous Work

The whole field of 3D fractals owes itself to *On Quaternions; or on a new System of Imaginaries in Algebra*[17] an extension of the complex numbers into the fourth dimension. By constructing the same fractal equations with quaternions instead of complex numbers and keeping one component fixed, 3D Julia sets may be rendered.

For example, if z_n represents the nth iteration of a 2D complex Julia set and q_n represents the nth iteration of a 4D quaternion Julia set, then

$$z_n = z_{n-1}^2 + c \tag{1}$$

$$q_n = q_{n-1}^2 + c \tag{2}$$

Quaternions are also used in computer programs to describe rotation due to their low space requirements and easy interpolation (among a few other pleasing qualities).

Aside from fractals, infinitely complex shapes (more on that in section 5), and rotational coordinates, quaternions are used in studying "the Lorentz group, the general theory of relativity group, the Clifford algebra, [...] the conformal group[, ...] crystallography, the kinematics of rigid body motion, the Thomas precession, the special theory of relativity, classical electromagnetism, the equation of motion of the general theory of relativity, and Dirac's relativistic wave equation."[18]

Fractals posed an interesting problem to the field of applied mathematics: nobody knew what they looked like. Fortunately, computers were rapidly gaining popularity, and a machine capable of rapidly performing millions of arithmetic operations was ideal for visualizing these chaotic and complex new sets. In 1978, Brooks and Matelski published the first-ever rendering of the Mandelbrot set,[19] an image so rudimentary it was printed as a 31×68 grid of asterisks, comprising just 2,108 bits of data — less than two tweets' worth![20]

Although it's possible to "brute force" a fractal by iterating over thousands of points in a 2D plane to render a 2D fractal, doing the same in three dimensions is much less practical, by an order of magnitude. How, then, may 3D fractals be visualized?

In *Generation and Display of Geometric Fractals in 2-D*[21], Alan Norton proposed a method for determining the surface of a fractal system. Norton brute-forces the problem, starting with one known point on the surface of the fractal (ex. $2+0i$ for the two-dimensional Mandelbrot set) and successively boundary-testing neighbors to find more border points. The duration of Norton's fractal renders using this method are not included in the paper, implying they were embarrassingly large. In the paper, Norton states that "[f]ractal surfaces are not differentiable"[22] — although true, the non-existence of a

[17] Hamilton, W. R. (1844). *On Quaternions; or on a new System of Imaginaries in Algebra. The London, Edinburgh and Dublin Philisophical Magazine and Journal of Science*, *25*.

[18] Girard, P. R. (1984). *The Quaternion Group and Modern Physics. European Journal of Physics*, *5*(1), 25.

[19] Brooks, R. and Matelski, J. P. (1978). *The Dynamics of 2-Generator Subgroups of PSL(2, C)*. In *Reimann surfaces and related topics: Proceedings of the 1978 stony brook conference* (pp. 65–67). Princeton University Press.

[20] In the simplest possible case, where a tweet consists of 140 codepoints in the C0 Controls and Basic Latin Unicode block (`U+0000`–`U+007F`) and no images or embedded media, a tweet contains 1,120 bits of data.

[21] Norton, A. (1982). *Generation and Display of Geometric Fractals in 2-D. ACM SIGGRAPH Computer Graphics*, *16*. doi:10.1145/965145.801263.

[22] Norton, A. (1982). *Generation and Display of Geometric Fractals in 2-D. ACM SIGGRAPH Computer Graphics*, *16*. doi:10.1145/965145.801263, p. 64.

derivative for a function doesn't imply that a derivative cannot be *estimated* at a point, the discovery that fueled John Hart's research.

However, Norton's method, which John Hart refers to as *boundary tracking*, was too computationally complex to be very useful. As such, a method faster than $O(k^n)$ for visualizing 3D fractals was badly needed, and so, much like early calculators[23] created calculus in order to better estimate values of difficult-to-calculate functions, new approaches were required.

In *The Science of Fractal Images*[24], six authors discuss the finer points of fractals and their applications. Notably for ray marching, Heinz-Otto Peitgen provides proofs for distance estimation functions of the Mandelbrot and Julia sets in section 4.2.5, which may then be extended into the fourth dimension to render 3D slices of Quaternion fractals.

In 1989, John Hart published *Ray Tracing Deterministic 3-D Fractals*[25], where he used Peitgen's distance estimators to create a ray marcher[26] making use of "an unusual construction called the unbounding volume"[27], a volume the object being rendered is guaranteed *not* to be in, estimated using the Hubbard-Doudy potential of a shape. By "marching" a sampling point along a ray, arbitrary volumes could be rendered, even unsolvable but estimable (i.e., differentiable) systems like Julia sets. This paper presents a reimplementation of Hart's algorithms.

Later, in 1994, Hart published a more detailed analysis of ray marchers (now called "sphere tracers"), in *The Visual Computer*, filled with more detailed mathematical proofs, rendering techniques and optimizations, distance functions, and analysis.

In 2005, Crane wrote a program to adapt Hart's algorithm to a GPU[28]— because GPUs are ideal for doing the same calculation to many similar entities and because each pixel in a ray-marched image may be rendered independently, ray marchers are ideal for GPU implementations. Most notably, Crane 's program is open-source[29] — Hart et. al. speak in vagueties, and although a symbolic representation is undoubtedly valuable, a function presented without a ballpark-range of valid inputs can be aggravatingly difficult to implement. Where Hart defined the normal vector $\vec{N}$ at a point $\langle x, y, z\rangle$ as

$$N_x = D(x+\epsilon, y, z) - D(x-\epsilon, y, z) \quad (3)$$

$$N_y = D(x, y+\epsilon, z) - D(x, y-\epsilon, z) \quad (4)$$

$$N_z = D(x, y, z+\epsilon) - D(x, y, z-\epsilon) \quad (5)$$

(where $D(x, y, z)$ represents the distance estimate at a point $\langle x, y, z\rangle$.), Crane defines an explicit value for ϵ: 0.0001. Crane suffered from the consequences of Hart's abstract paper as well, noting before defining several constants that they "were determined through trial and error and are not by any means optimal."[30]

5 Fractals

So far, Julia and Mandelbrot sets have been mentioned several times without any real explanation as to what

[23] The word choice here was a matter of some debate; both "calculographers" and "calculotitians" were considered, but ultimately it was realized that the word for someone who performs calculus is simply a *calculator* [1]. Unfortunately, in the past 50 or 60 years, "calculator" has, much like "computer," come to be associated with electronics rather than humans. Alas, no terminology more satisfactory can be found at the time of this writing, although the reader may rest assured that this author will be on the lookout for new terminology as it may arise.

[24] Barnsley, M. F., Devaney, R. L., Fischer, Y., Mandelbrot, B. B., McGuire, M., Peitgen, H.-O., …, and Voss, R. F. (1988). *The Science of Fractal Images*. New York, NY, USA: Springer-Verlag New York, Inc.

[25] Hart, J. C., Sandin, D. J., and Kauffman, L. H. (1989). *Ray Tracing Deterministic 3-D Fractals*. *SIGGRAPH Comput. Graph.* *23*(3), 289–296. doi:10.1145/74334.74363.

[26] The term "ray marcher" hadn't been invented yet, so Hart called his creation a "ray tracer"

[27] Hart et al., "Ray Tracing Deterministic 3-D Fractals," p. 289.

[1] Thompson, S. P. (1910). *Calculus made easy: Being a very-simplest introduction to those beautiful methods of reckoning which are generally called by the terrifying names of the differential calculus and the integral calculus*. MacMillan and Co.

[28] Graphical processing unit, a specialized piece of hardware in a computer specifically designed for graphical calculations — contrast this with the all-purpose CPU, or central processing unit.

[29] The source code is freely available online to examine and modify.

[30] Crane, *Ray Tracing Quaternion Julia Sets on the GPU*, p. 8.

they are or why they're useful or interesting. What is a fractal, and what are the Julia and Mandelbrot sets in specific?

5.1 Definition of a Fractal

A fractal is a shape of infinite detail, meaning that any image of a fractal can always be resolved to greater detail, and that the error between an approximation of a fractal (such as a digital picture, a stone carving, or a neatly-aligned list of numbers) and the fractal itself will always exist.

Immediately, this poses some significant problems: human eyes cannot see infinite detail, human brains cannot comprehend infinity, and there exists no possible way to create an accurate representation of an object with infinite detail, due to the small but finite size of atoms. However, while this may initially seem like a drawback, it turns out to be our panacea; because humans are incapable of *perceiving* infinite detail, there is no need to *create* truly infinite detail when creating representations of fractals — a representation that is simply "detailed enough" will do just fine.

Approximations of fractals are usually created through the repeated application of a rule to a starting object or condition — each application of the rule is referred to as an *iteration*.

A fractal may be as simple as a spiral, like the ones seen in figure 10. Note that although the radius *approaches* 0 as more and more iterations are rendered, the curve will never reach the center point — as such, the true form of this fractal (and all fractals) can only exist in a theoretical sense.

Fractals like the Koch snowflake,[31] as seen in figure 11, are formed by repeated application of a rule to a starting condition. This sort of recursive definition can be formally specified with an *L-system*, named after Aristid Lindenmayer who created them to describe the behavior of plant growth.[32] The Koch snowflake above may be defined with an alphabet of f (move forward), $+$ (turn 60° clockwise), and $-$ (turn 60° counter-clockwise). Starting from an *axiom* of $f++f++f$, the string is rewritten where f is replaced with $f-f++f-f$, and this repetition is repeated n times to generate the nth iteration of the fractal. Although L-systems won't be mentioned again in this paper, it's important to emphasize that all fractals can be formally specified with a set of unambiguous rules and can always be refined to reveal progressively finer details.

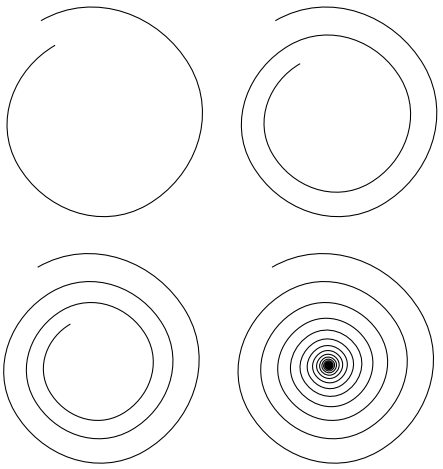

Figure 10: The first three iterations and the 20th iteration of a fractal created by curving a full turn around a center point while decreasing the radius of the curve by 25%

5.2 The Mandelbrot Set

The Mandelbrot set $\mathcal{M}$ is defined to be the set of points such that

$$\begin{gathered} c \in \mathbb{C},\ z_0 = 0, \\ \lim_{n\to\infty} z_n = z_{n-1}^2 + c \neq \infty \ \Rightarrow\ c \in \mathcal{M} \end{gathered} \tag{6}$$

Where calculating z to z_n is referred to as calculating the n-th iteration of the fractal. (Confused about notation? Check section 10.1 for a glossary.)

In practical scenarios, calculating infinite iterations of z_n is both impossible and impractical, as some points don't even converge in the limiting case, such as $c = -1$, where $z_n = {}^{((-1)^n-1)}/_2$, which alternates in a 2-state

[31] Weisttein, E. W. (2017). *Koch snowflake*. Retrieved from mathworld.wolfram.com/KochSnowflake.html.

[32] Lindenmayer, A. (1968). *Mathematical models for cellular interactions in development i. filaments with one-sided inputs*. *Journal of theoretical biology*, *18*(3), 280–299.

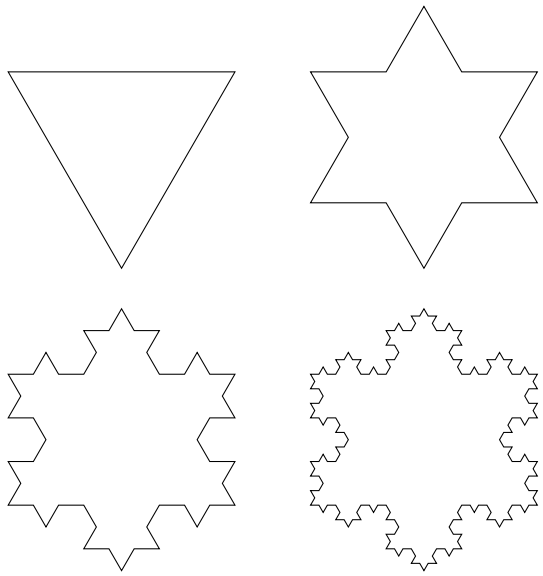

Figure 11: The first four iterations of the Koch Snowflake, a 1904 fractal proposed by Helge von Koch

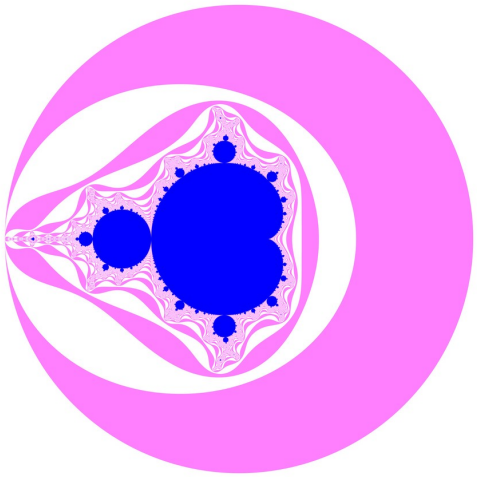

Figure 12: A rendering of the Mandelbrot set. The contiguous blue shape at the center is the Mandelbrot set, and the pink bands outside of it show the detail resolved with each successive iteration, starting with the large black circle of radius 2, the one-iteration approximation of the Mandelbrot set

orbit[33] between 0 and -1 (although it should be noted that as $\infty \neq \text{DNE}$, $-1 \in \mathcal{M}$ despite the lack of a limit).

All values of $c \in \mathcal{M}$ will cause z_n to fall into a pattern of orbits or converge to a limit.

See a visualization of the Mandelbrot set in figure 12. Perhaps notably, no points that are not included in a previous iteration of the set are included in the next iteration of the set; that is, no points are erroneously excluded through the approximation.

However, it may be proven that no values of c such that $|c| > 2$ exist, simplifying the limit from $\lim_{n\to\infty} \neq \infty$ to $z_n < 2$. As n is not specified here, it may be arbitrarily large; because points closer to the boundary of $\mathcal{M}$ take more iterations to diverge, increasing the iteration count increases the visible detail of the set. The implication here is that *infinite* iterations are required to render the full detail of the set. However, as no methods for displaying anything of infinite detail exist, a high but finite iteration count is sufficient for all practical purposes.

5.3 Julia Sets

Julia sets are very similar to the Mandelbrot set, with two notable differences:

1. Whereas the Mandelbrot set are defined in terms of an $z_n = z_{n-1}^2 + c$, Julia sets are defined in terms of $z_n = f(z_{n-1})$, where $f(z)$ is a complex rational function.[34]
2. Whereas the Mandelbrot set is defined in terms of a c that varies across the complex plane and a z_0 that stays at a constant 0, Julia sets are defined in terms of a z_0 that varies across the complex plane and a c that is kept constant across the whole render. In this way, the Mandelbrot set is a visualization of all the possible quadratic Julia sets for $z_n = z_{n-1}^2 + c$.

Julia sets are seen in figures 15 and 16, smoothly col-

[33] An *orbit* occurs when the sequence of z_ns oscillates between several constant values; the term may be used to describe the behavior of a point on a fractal that neither converges or diverges, always moving and never straying from its path.

[34] A function $f(x)$ is rational if it can be written in the form of $f(x) = P(x)/Q(x)$, where $P(x)$ and $Q(x)$ are two arbitrary polynomials, where a polynomial is a function consisting of the multiplication, addition, subtraction, and integer exponentiation of variables and real numbers.

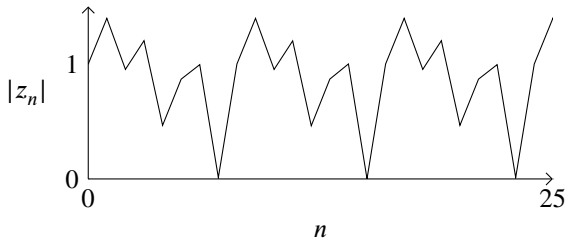

Figure 13: The regular 8-state orbit of z_ns in the Mandelbrot set for $c = -0.023077195 + 0.999033603i$

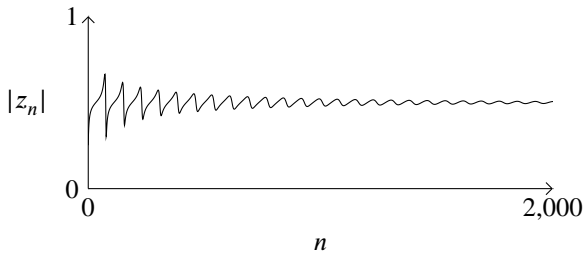

Figure 14: The series of z_ns for $c = 0.25161410494239694 - 0.0001816770978022922i$ converges to $|z_\infty| \approx 0.5\ldots$

ored by escape time — the number of iterations a point takes to diverge past the bounding radius (2 for the Mandelbrot set, usually $\approx$ 30 for Julia sets). Note that the fractal seen in figure 17 is *not* a Julia set. Also note that Julia sets are radially symmetrical proportional to the degree of $f(z)$.

6 *I C the Light*

I C the Light is a bespoke, open-source ray marcher written in C99 by the author. We will examine the inner workings of *I C the Light* from the perspectives of dependencies, program flow, and its render model.

6.1 Dependencies

I C the Light's C header files have a dependency hierarchy; figure 18 shows which headers include which; `main.c` is compiled and includes `main.h`, which then includes every other file in the project.

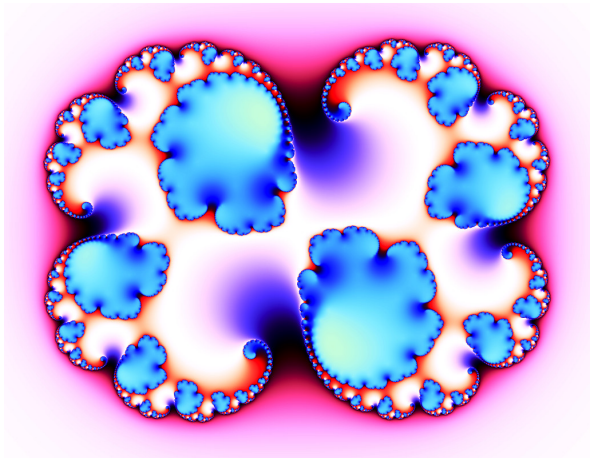

Figure 15: Julia set for $f(z) = z^2 + c,\ c = 0.285 + 0.01i$

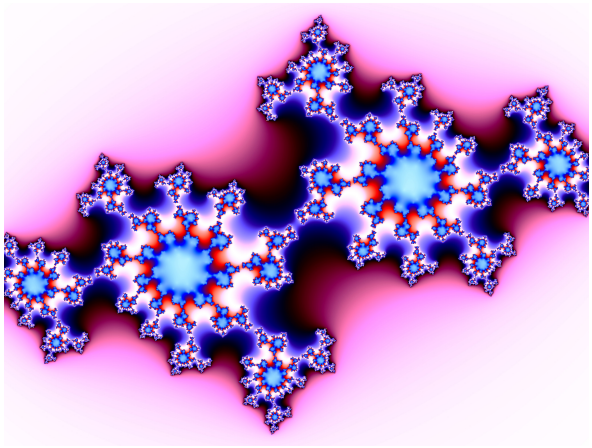

Figure 16: Julia set for $f(z) = z^2 + c,\ c = -0.70176 + -0.3842i$

6.2 Zones and Modules

I C the Light is roughly divisible into three main *zones*, each composed of *modules* that support the program itself; infrastructure, arithmetic, and raster. Each module is composed of one `.c` / `.h` pair, and can depend on other modules.

6.2.1 Infrastructure Zone

The infrastructure zone supports the development and base operations of *I C the Light*, and includes global variables, diagnostic logging, bitmasking flag operations, and common functions useful to many different

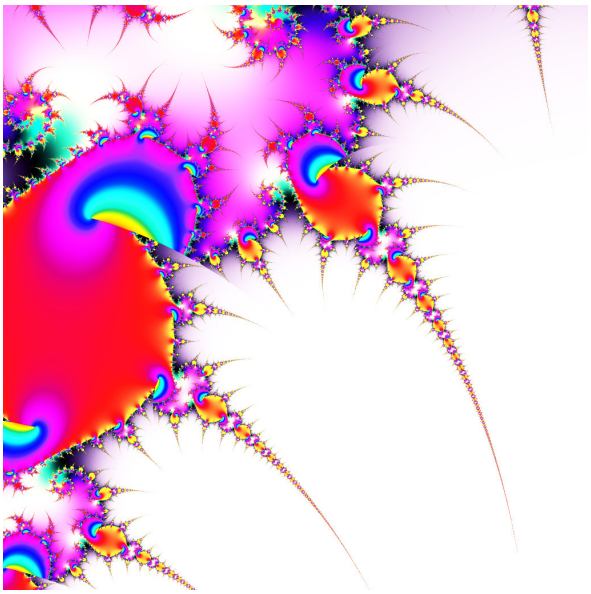

Figure 17: Not a Julia set: $f(z) = \sqrt{\sinh z^2} + c$, $c = 0.064 + 0.122i$, rendered about $(1.67, 0.59)$. $f(z)$ is not a rational function, so this rendering is, despite its beauty, not a Julia set.

modules.

`common.c` contains functions and definitions common to all the modules in *I C the Light*. Examples include definitions for various π-related constants,[35] random integer functions, an approximation for $\sin\theta$, and several functions for dealing with floats: linear interpolation, minimum / maximum functions, and a function (`scale()`) for mapping $n \in [i_{\min}, i_{\max}] \mapsto n' \in [o_{\min}, o_{\max}]$.[36][37] The actual calculation of n' from n is:

$$n' = \frac{n - i_{\min}}{\Delta i}\Delta o + o_{\min} \tag{7}$$

[35] π^2, 2π, π, $\pi/2$, and $\pi/4$

[36] Floating-point arithmetic only approximates the real numbers, so classifying `scale()` as injective, surjective, or bijective would be somewhat pointless. However, if $\Delta i < \Delta o$ (that is, $[i_{\min}, i_{\max}]$ is a smaller range than $[o_{\min}, o_{\max}]$), `scale()` functions as injective. If $\Delta i > \Delta o$, `scale()` functions as surjective, and if $\Delta i = \Delta o$, `scale()` is a bijective mapping (probably — floating-point arithmetic is notoriously unreliable and these statements should be taken with a grain of salt).

[37] Note that the notation n' simply indicates that n' is a value *related* to n, as opposed to a derivative of n.

- `main.h`
  - ⟶ `icthelight.h`
    - ⇒ Infrastructure zone
      - `globals.h`
      - `logging.h`
      - `common.h`
      - `flags.h`
    - ⇒ Math zone
      - `vector.h`
      - `quaternion.h`
      - `complex.h`
      - `color.h`
    - ⇒ Raster zone
      - `ppm.h`
      - `plot.h`

Figure 18: *I C the Light*'s header dependency tree

`common.c` also contains functions for searching `*argv[]` for given strings to locate option / value pairs, and functions for finding the minima / maxima / average of float arrays.

`logging.c` contains a single function, `initializelogfile()`, to initialize `FILE *logfile` for writing.

A global flag variable is used to pass options to customize *I C the Light*'s behavior. To support this, a set of constants `B0` through `B15` are defined for masking (which are then aliased to a higher-level option definition like `CONVERT_IMMEDIATELY` or `USER_QUATERNION`). Then, two macro functions are defined:

```
#define FLAG(F)            (flags & F)
#define FLAGSET(F) flags = (flags | F)
```

`FLAG(F)` for checking if a flag is set via a bitwise `AND`, and `FLAGSET(F)` for setting a flag by combining `flags` and the user-supplied flag `F` via a bitwise `OR`.

Although *I C the Light* never required the removal of a flag, such a function would be easy to implement:

```
#define FLAGUNSET(F) flags = (flags ^ F)
```

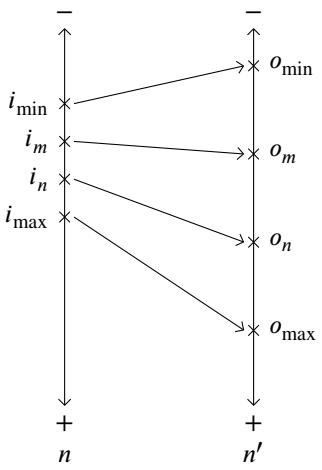

Figure 19: Diagram of linear map $[i_{\min}, i_{\max}] \mapsto [o_{\min}, o_{\max}]$

6.2.2 Arithmetic Zone

The arithmetic zone consists of all the tools to perform math with vectors, colors, complex numbers, and quaternions.

`quaternion.c` contains the definition of a quaternion datatype as a four-tuple of floats (`r` for the real component, and `a`, `b`, and `c` for the imaginary i, j, and k components[38]) and associated arithmetic functions.

The most complex (and unintuitive) of these are multiplication and squaring. Mathematically, the product of two quaternions x and y may be defined as follows, with x_r representing the real component of x, x_i representing the i-component of x, and so on.

$$\begin{aligned} xy \;=\; & x_r y_r - x_i y_i - x_j y_j - x_k y_k \\ & + (x_r y_i + x_i y_r + x_j y_k - x_k y_j)i \\ & + (x_r y_j - x_i y_k + x_j y_r + x_k y_i)j \\ & + (x_r y_k + x_i y_j - x_j y_i + x_k y_r)k \end{aligned} \tag{8}$$

`vector.c` contains functions for operating on two- and three-dimensional vectors. Functions for operating on 2D vectors are postfixed with a "2," and functions for operating on 3D vectors are postfixed with a "3." Functions that operate on a vector with a scalar are postfixed with an "s." For example, `mult2()` multiplies two 2D vectors together, and `mult2s()` multiplies a 2D vector with a scalar.

For the most part, `vector.c` is filled with unremarkable arithmetic. However, there are a few notable functions for the manipulation of vectors.

`fromdirection3()` generates a 3-vector $\vec{v}$ from a magnitude m, an angle α in the xy plane, and an angle β in the yz plane.

$$\begin{aligned} v_x &= m \cos\beta \sin\alpha \\ v_y &= m \cos\beta \cos\alpha \\ v_z &= m \sin\beta \end{aligned} \tag{9}$$

`color.c` contains the definition of a `struct rgbcolor` and functions for manipulating and converting colors. Functions for averaging colors, linearly interpolating colors, shifting hues, adding colors, and converting colors to and from unsigned 24-bit[39] integers (in `0xrrggbb` format). Future versions of *I C the Light* may contain an alpha channel as well.

`complex.c` contains the definition of a complex number type as a two-tuple, functions for complex arithmetic, and functions for computing the Mandelbrot set. Although the point-testing and distance functions for the Mandelbrot set may be better categorized in `distance.c`, the 2D Mandelbrot set is not rendered anywhere within the main branches of *I C the Light* — however, a branch from *I C the Light*'s 2D ages is still (somewhat) maintained, which utilizes the Mandelbrot functions.

6.2.3 Raster Zone

The raster zone contains raster operations: raster surface types, flood-filling surfaces, image output, pixel plotting, and so on.

`ppm.c` contains `writeppm()` for outputting images (as unsigned integer arrays) to portable pixel-map images, the simplest possible format. Following a simple width / height / type header, a literal listing of the colors delimited by spaces (`rrggbb rrggbb rrggbb...`) composes up a `.ppm` image. Although `ppm`s are massive in comparison

[38] In retrospect, this seems fairly nonsensical, given that the imaginary components already *had* one-letter names.

[39] In practice, 32-bit, but the highest 8-bits aren't utilized.

to a more reasonable format like `.png`,[40] integrating the whole `png` specification would be difficult[41] and out of the scope of the project. Instead, shelling out to ImageMagick for conversion is enabled via an option or through `make convert`.

`plot.c` contains two significant functions: `getpixel()` and `plot()`, for getting and setting pixels in a surface.

Ultimately, the code isn't that complex,[42] boiling down to

```
((unsigned int *)screen->pixels)
        [x + y * screen->w] = color;
```

but it's a useful and critical abstraction in the backbone of *I C the Light*'s graphical output.

6.3 Program Flow

6.3.1 Function Call Hierarchy

- `main()`
  - ⟶ `searchargs()`
  - ⟶ `handleevents()`
  - ⟶ `saveframe()`
  - ⟶ `render()`
    - ⇒ (assorted vector functions)
    - ⇒ (assorted color functions)
    - ⇒ (assorted array, summing functions)
    - ⇒ `de()`
      - `distancejulia()`
    - ⇒ `blinnphong()`
      - `getnormal()`

6.3.2 Render Flow

First, constants and variables are initialized.

Memory is allocated for `float values[]` and `int coords[]`, which both have the same cardinality as the output surface. `values` stores raw outputs from the render, pre-coloring; it can be a combination of illumination (from the Blinn-Phong function), step-count, distance, or some other model. Because values are stored with no pre-processing, the minima and maxima of `values` can be calculated and used later to determine image exposure. `coords` stores the locations of the set coordinates in `values`, which is to say `values[i]` corresponds to `screen[coords[i]]`. This strange layout is used to prevent null values skewing exposure calculations.

The viewport is calculated from four values: a focal length f = `float focallength`, which determines the perspective of the viewport (how "zoomed in" the image is, and how close to parallel parallel lines appear as in an output image), an offset `vec3 viewport_ofs` to the center of the viewport that determines the camera's position in space, and $\vec{v}_w$ = `viewport_width` and $\vec{v}_h$ = `viewport_height` unit vectors, which determine the viewport's aspect ratio and rotation. Because $\vec{v}_w$ and $\vec{v}_h$ are both arbitrary 3-vectors, any rotation of the viewport can (and must) be described through the rotation of these vectors. An interesting (and largely unexplored by the author) consequence of this is that the camera could be, if desired, a parallelogram or of an entirely different aspect ratio as the screen.

Rays are shot from the *camera*, a point constructed by moving perpendicularly backwards from the plane constructed from the two viewport vectors by distance f *through* points on the rectangle constructed from the two viewport vectors.

Perhaps more simply,

$$\vec{c} = f(\vec{v}_w \perp \vec{v}_h) + \vec{v}_{\text{offset}} \tag{10}$$

Where $\vec{c}$ represents the camera, f the focal length, and $\vec{v}_w \perp \vec{v}_h$ a unit vector perpendicular to $\vec{v}_w$ and $\vec{v}_h$.

If the user has not specified a quaternion constant to render, a constant is generated by setting the four quaternion components to random values from -1 to 1.

With initial setup completed, rendering may begin. The following process is repeated for every pixel in the screen where i or y represents the y-coordinate of the pixel currently being rendered and j or x represents its x-coordinate. w and h will represent the width and height

[40] Portable network graphics, supposedly pronounced "ping" (Adler, M., Boutell, T., Brunschen, C. et al. [1996]. *PNG (Portable Network Graphics) Specification*. W3C)

[41] Ibid.

[42] If you're not familiar with C or pointers, this may be safely ignored.

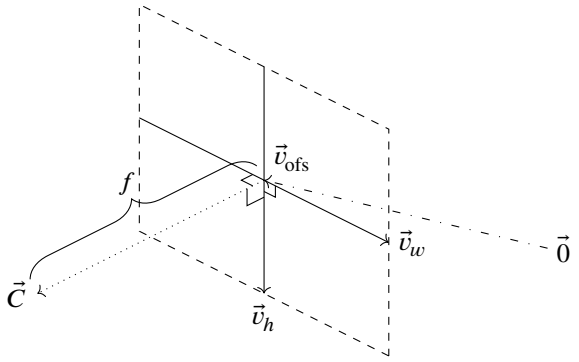

Figure 20: Diagram of the camera's construction

of the screen, respectively.

First, the `distance` and t = `totaldistance` counters are reset to 0. Next, `scale()` maps $x \in [0, w] \mapsto x' \in [-1/2, 1/2]$ and $y \in [0, h] \mapsto y' \in [-1/2, 1/2]$. This is so that `viewport_ofs` represents the center, rather than the top-left point on the viewport, making positioning the camera simpler and more intuitive. The ray's origin, the first sample position, is then calculated as

$$\vec{o} = x'\vec{v}_w + y'\vec{v}_h + \vec{v}_{\text{offset}} \tag{11}$$

Essentially, a coordinate (x, y) is mapped from the screen onto its equivalent position `ray_orig` $= \vec{o}$ in space on the viewport for arbitrary viewport locations, widths, and heights, as seen in figure 21.

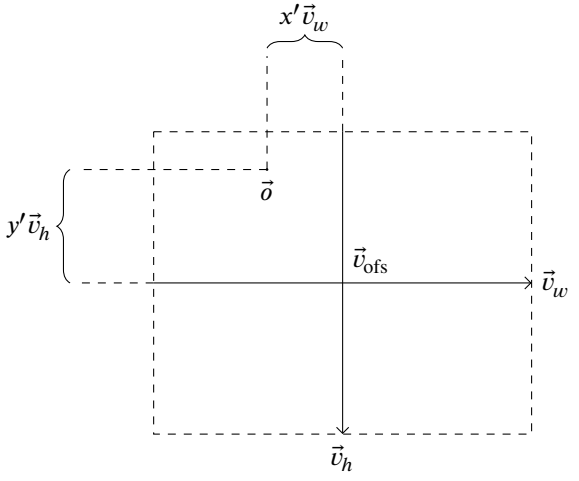

Figure 21: A head-on view of the viewport, showing the location of the ray's origin $\vec{o}$ in terms of x' and y'

In addition, a directional unit vector $\hat{r}$ is calculated as

$$\hat{r} = \frac{\vec{c} - \vec{o}}{\|\vec{c} - \vec{o}\|} \tag{12}$$

Then, with the ray setup complete, the actual distance estimation process may be started with a sampling position $\vec{s}_n$ starting at $\vec{s}_0 = \vec{o}$. In more general terms, $\vec{s}_n$ may be defined as

$$\vec{s}_n = \vec{s}_{n-1} + \hat{r}D(\vec{s}_{n-1}) \tag{13}$$

or in practice

$$\vec{s}_n = \vec{o} + \hat{r}t \tag{14}$$

where t represents the total distance from $\vec{o}$ marched in n steps ($t = \sum_{i=0}^{i=n} D(\vec{s}_i)$).

A smattering of render-wide constants will now become relevant. First, σ, an upper bound on the maximum distance travelled before $\vec{s}_n$ is considered to have escaped from the scene (a bounding volume[43] on the scene).

Also relevant is ϵ, a bound on the maximum value of $D(\vec{s}_n)$ such that $\vec{s}_n \in \mathcal{S}$ — in other words, $\vec{s}_n$ is considered to be a point in the set of points making up the scene of object(s) to be rendered. In reality, $\vec{s}_n$ will not be in the scene, but for an arbitrarily small ϵ the approximation of the boundary points of $\mathcal{S}$ will become arbitrarily close to the actual set of the boundary points of $\mathcal{S}$.

Finally, we define a constant scaling factor k as the portion of the unbounding volume defined by an n-sphere of radius $D(\vec{s}_n)$ about $\vec{s}_n$ to be outside of $\mathcal{S}$. Although a true distance-estimating function would yield a perfect distance between a point and the scene, our practical functions will only approximate the ideal function (but become arbitrarily accurate as the sample position approaches the boundary of $\mathcal{S}$). As such, it is salient to only consider some portion of radius $kD(\vec{s}_n)$ to be a valid unbounding volume about $\vec{s}_n$. Reasonable values of k are between 0.5 and 0.8.

If $D(\vec{s}_n) \leq \epsilon$, the point is considered part of $\mathcal{S}$, a shading model is applied (*I C the Light* uses Blinn-Phong), and we move on to the next ray. If $D(\vec{s}_n) > \sigma$, the ray is abandoned for the opposite reason (it is considered that $\vec{s}_n$ will never approach $\mathcal{S}$).

[43] Not to be confused with an *un*bounding volume.

6.3.3 Quaternion Julia Set Distance Estimate

Within the estimate, we let $\vec{s}_n = \vec{p}$, for simplicity.

First, we initialize two quaternion variables, q and q', the running derivative of q:

$$q_0 = \vec{p}_x + \vec{p}_y i + \vec{p}_z j + 0k \qquad (15)$$
$$q'_0 = 1$$

We may therefore define recursive definitions for their n-th iteration, using the definition of the complex Julia set and the chain rule of derivatives

$$q_n = q_{n-1}^2 + c \qquad (16)$$
$$q'_n = 2q_{n-1}q'_{n-1}$$

Hopefully, these definitions look somewhat familiar — save for q (for *quaternion*) being a quaternion rather than a complex number[44] denoted by z.

Then, we check if

$$\lim_{n\to\infty} \|q_n\| \geq 4 \qquad (17)$$

If, after a reasonable N iterations, q hasn't diverged, we estimate the distance $D(\vec{p})$ as

$$d = \frac{\|q\| \log \|q\|}{2\|q'\|} \qquad (18)$$

The derivation of this equation is found in *The Science of Fractal Images*[45], p. 192.

Next, the point $s_f = \vec{o} + \hat{r}d$ is passed to the shader.

6.4 Shading

I C the Light uses the Blinn-Phong shading model, an equation that determines how brightly lit a point is.

$$I = i_a + \underbrace{\hat{L} \cdot \hat{N}}_{\text{specular}} + \underbrace{(\hat{N} \cdot \hat{H})^{\alpha}}_{\text{diffuse}} \qquad (19)$$

Where I is the light intensity at a point, i_a is the intensity of the ambient light, $\hat{H}$ is a normalized vector halfway between the direction of the light $\hat{L}$ and the direction of the camera, $\hat{N}$ is the normal vector at the point, and α is a shininess constant, determining how glossy the object appears — a higher α creates a glossier surface.

Essentially, the Blinn-Phong model states that how brightly lit a surface is corresponds to how directly it's facing the light (as the dot product $\hat{L} \cdot \hat{N}$ maximises when the angle θ between the two vectors is 0).

For multiple lights, this equation is repeated and the illumination intensities are summed together, and for a more granularly controlled light constant modifiers can be added to determine the intensity of each of the light's components (specular and diffuse).

6.5 Conclusion of *I C the Light*'s functionality

I C the Light is a fully-developed ray marcher capable of rendering quaternion Julia set fractals. However, it has several notable limitations: it can't export transparent images or render multiple lights, colored lights, shadows, reflections, or shading. Although these limitations would make *I C the Light* useless in many contexts (e.g. for animated movies or video games), remember that *I C the Light*'s primary purpose is to visualize quaternion Julia sets, a purpose it completes admirably — reflections, colored lighting, and shadows aren't nec-

[44] Apparently, because complex numbers are written in the form $x + yi$, z was chosen simply for being the next letter in the alphabet.[f] An alternate explanation for the convention of naming complex numbers z was posited by Tim Hanson, who observed that "[u]nlike statistics and probability, complex analysis is boring, especially contour integrals. When I used to do my complex analysis homework in graduate school I would often fall asleep with my head on the desk and snore: '…zzzzzzzzzzzzzzzzzzzzzzzzz.' "[g]

[f] Meha, M. (2016). *Why are complex numbers denoted by z?* Retrieved from quora.com/Why-are-complex-numbers-denoted-by-z/answer/Mecofe-Meha

[g] Hanson, T. (2016). *Why are complex numbers denoted by z?* Retrieved from quora.com/Why-are-complex-numbers-denoted-by-z/answer/Tim-Hanson-4

[45] Barnsley, M. F., Devaney, R. L., Fischer, Y., Mandelbrot, B. B., McGuire, M., Peitgen, H.-O., …, and Voss, R. F. (1988). *The Science of Fractal Images*. New York, NY, USA: Springer-Verlag New York, Inc.

essary to visualize these sets, just an added decoration.

7 Discussion

Perhaps more of *I C the Light*'s value is in what it taught me rather than its functionality. Even within the ray marchers, *I C the Light* is not an outlier in any way — it's not quite as fast as WebGL-enabled marchers like those on Iñigo Quílez's Shadertoy[46] and it lacks the versatility, power, and user-friendly aspects of programs like Krzysztof Marczak's Mandelbulber.[47]

So what, then, is *I C the Light*'s value? Before I set out to create *I C the Light*, I knew very little about fractals.[48] On the way, I learned an enormous amount about fractals and mathematics — everything from RGB matrix transforms to shift the hue of a color to the formal definition of a Julia set to the history of quaternions and the history of fractals to things as simple as the multiplication of complex numbers. By creating software to do exactly what I required, I ended up with a product that is both perfect for me and imparted a much deeper understanding of the subject matter than research alone would have provided. If you've enjoyed this paper and the renders I've included, I implore you: Strike out at the world, go yonder and create your own!

8 Further Reading

This paper's bibliography contains many excellent papers and books which I highly suggest reading or at least skimming, covering a wide range of mathematical and computational concepts. Many are ground-breaking papers in their fields, and all are high-quality research. However, reading dense academic papers does rarely a passion make, and fields notorious for their inaccessibility are even more worthy of a potential reader's wariness. As such, this author suggests some of the following materials for the discerning reader looking to expand their knowledge of ray marching or even make their own ray marcher![49]

Syntopia's Mikael Hvidtfeldt Christensen presents *Distance Estimated* 3D *Fractals*, an eight-part series of blog posts detailing the theory of distance estimation — this is the post that made me realize creating a ray marcher is something I could do. `goo.gl/ahxqTV`

Iñigo Quílez provides a list of dozens of working, proven distance estimator functions, reference implementation included, in his article *Modeling with Distance Functions*. If the reader finds themself making a ray marcher, they should make use of this invaluable resource. `goo.gl/evfU5b`

Or maybe the reader is less interested in geometric primitives and more in the swirly whipped-cream structures of quaternion Julia sets — and who could blame them? In that case, they should read Paul Bourke's introduction to quaternion arithmetic and quaternion Julia sets. `paulbourke.net/fractals/quatjulia`

I also recommend reading Keenan Crane's paper *Ray Tracing Quaternion Julia Sets on the* GPU, a delightful open source and mostly-well-commented program that can elucidate some of the critical details that Hart et. al. leave as an exercise to the reader. `goo.gl/r3eNin`

9 Acknowledgments

Wow! Thanks for reading the whole paper!

10 Variable and Notation Reference

10.1 Notation

$\vec{v}$ — A vector of arbitrary dimensions. The arrow, not the letter, indicates that $\vec{v}$ is a vector.

$\|\vec{u}\|$ — The magnitude or absolute value of a vector $\vec{u}$. If $\vec{u}$ is n-dimensional,

$$\|\vec{u}\| = \sqrt{\sum_{i=0}^{i=n} \vec{u}_i^2} \tag{20}$$

[46] `shadertoy.com`

[47] `mandelbulber.com`

[48] I mean, I *thought* I knew a lot about fractals. They have been a passion of mine for a long time, but I've only ventured into their creation rather recently. It turns out it's a lot easier than I thought! (Well, for simple 2D fractals at least — I can't pretend coding *I C the Light* wasn't a challenge.) Dunning-Kruger?

[49] Step one is to give it a cool name, preferably with a pun!

$\mathcal{A}, \mathcal{B}, \mathcal{C}...$	Capital script letters indicate that a variable represents a set of points.
$\vec{a} \perp \vec{b}$	A vector perpendicular to both $\vec{a}$ and $\vec{b}$. Also written as $\vec{a} \times \vec{b}$
$\hat{v}$	A unit or direction vector of magnitude 1.
$\mathbb{A}, \mathbb{B}, \mathbb{C}...$	A special set, such as the integers ($\mathbb{Z}$). Usually a set of numbers (or, more generally, numerical objects, a definition that can be extended to include n-tuples). Contrast with $\mathcal{A}, \mathcal{B}, \mathcal{C}...$, which always refer to sets of *points*.
$\mathbb{R}$	The set of real numbers. Includes all the rational numbers (of form $p/q,\ p, q \in \mathbb{Z}$) as well as roots and transcendentals.
$\mathbb{C}$	The set of complex numbers. Includes the set of real numbers $\mathbb{R}$ and the set of numbers $ni,\ \forall n \in \mathbb{R}, i = \sqrt{-1}$.
$x \in y$	x is *in* y or x is a member of y.

10.2 Variables

$\vec{c}$	Camera position vector, the location rays are "shot" from.
f	Camera focal length. Describes how "zoomed in" the render is.
$D(\vec{p})$	Distance estimator. Estimates the distance to $\mathcal{S}$ from point $\vec{p}$.
ϵ	A threshold for considering a point $\vec{s}_n$ to be an element of $\mathcal{S}$. $D(\vec{s}_n) \leq \epsilon \implies \vec{s}_n \in \mathcal{S}$. Note that *considering* $\vec{s}_n$ to be an element of $\mathcal{S}$ does not imply that $\vec{s}_n$ is *actually* a member of $\mathcal{S}$ — just that it approximates the boundary of $\mathcal{S}$ to a precision of ϵ.
k	The portion of the unbounding volume defined by an n-sphere of radius $D(\vec{s}_n)$ about $\vec{s}_n$ considered to be outside of $\mathcal{S}$. Distance estimators often over-estimate the distance bound, so it is salient to only trust some portion of radius $kD(\vec{s}_n)$ to be considered truly outside of $\mathcal{S}$
$\mathcal{M}$	The set of points in the Mandelbrot set.
$\vec{o}$	The ray origin, where the rays are shot from in direction $\hat{r}$. $\vec{o} = \vec{s}_0$
$\hat{r}$	Ray direction. Describes the direction an individual ray is shot in. Unit vector ($\|\hat{r}\| = 1$).
$\mathcal{S}$	The set of points in the scene to be rendered.
σ	An upper bound on the distance of the scene from the origin. $\forall s \in \mathcal{S},\ \sigma \not> s$
$\vec{s}_n$	The n-th sample position. Represents $\vec{p}$ in $D(\vec{p})$.
t	The sum of all the distance estimates.
$\vec{v}_{\text{offset}}$	Position vector describing the location of the viewport's center.
$\vec{v}_w, \vec{v}_h$	Viewport width and height vectors, describing the horizontal or vertical center lines of the viewport.
w, h	Screen width and height, in pixels.
x, y	Horizontal and vertical location of the pixel being rendered.
x', y'	Number describing the horizontal or vertical offset of the ray origin from the viewport center $\vec{v}_{\text{offset}}$. Both have values between $-1/2$ and $1/2$.

11 Glossary

DE — Distance estimate. An estimate on the distance from a point to the closest point in a scene.

Float — Floating-point number. A basic data-type that approximates a real number. Available in a variety of bits for various levels of accuracy.

Module — A small piece of the program that performs a single, designated task, such as vector arithmetic or image output.

Ray-based renderer — A program that renders images by locating intersections of rays.

Ray tracer — A ray-based renderer that finds the intersection between a ray and a scene by solving a discrete equation in $O(1)$ time.

Ray marcher — A ray-based renderer that finds the intersection between a ray and a scene by stepping smaller distances along the ray.

Scene — The collection of objects that compose a physical environment. Generally refers to the union of all objects in a render.

Zone — A collection of program modules that perform a general set of tasks.

Works Cited

Adler, M., Boutell, T., Brunschen, C. et al. (1996). *PNG (Portable Network Graphics) Specification*. W3C.

Barnsley, M. F., Devaney, R. L., Fischer, Y., Mandelbrot, B. B., McGuire, M., Peitgen, H.-O., … Voss, R. F. (1988). *The Science of Fractal Images*. New York, NY, USA: Springer-Verlag New York, Inc.

Boonman, A., Bar-On, Y., Yovel, Y., & Cvikel, N. (2013). *It's Not Black or White — On The Range of Vision and Echolocation in Echolocating Bats*. *Frontiers in Physiology*, *4*. doi:10.3389/fphys.2013.00248

Brooks, R. & Matelski, J. P. (1978). *The Dynamics of 2-Generator Subgroups of PSL(2, C)*. In *Reimann surfaces and related topics: Proceedings of the 1978 stony brook conference* (pp. 65–67). Princeton University Press.

Crane, K. (2005). *Ray Tracing Quaternion Julia Sets on the GPU*. University of Illinois at Urbana-Champaign.

Girard, P. R. (1984). *The Quaternion Group and Modern Physics*. *European Journal of Physics*, *5*(1), 25.

Hamilton, W. R. (1844). *On Quaternions; or on a new System of Imaginaries in Algebra*. *The London, Edinburgh and Dublin Philisophical Magazine and Journal of Science*, *25*.

Hanson, T. (2016). *Why are complex numbers denoted by z?* Retrieved from quora.com/Why-are-complex-numbers-denoted-by-z/answer/Tim-Hanson-4

Hart, J. C. (1994). *Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces*. *The Visual Computer*, *12*, 527–545.

Hart, J. C., Sandin, D. J., & Kauffman, L. H. (1989). *Ray Tracing Deterministic 3-D Fractals*. *SIGGRAPH Comput. Graph. 23*(3), 289–296. doi:10.1145/74334.74363

Layne, J. N. (1967). *Evidence For The Use Of Vision In Diurnal Orientation Of The Bat Myotis Austroriparius*. *Animal Behaviour*, *15*(4), 409–415. doi:10.1016/0003-3472(67)90037-1

Lindenmayer, A. (1968). *Mathematical models for cellular interactions in development i. filaments with one-sided inputs*. *Journal of theoretical biology*, *18*(3), 280–299.

Meha, M. (2016). *Why are complex numbers denoted by z?* Retrieved from quora.com/Why-are-complex-numbers-denoted-by-z/answer/Mecofe-Meha

Norton, A. (1982). *Generation and Display of Geometric Fractals in 2-D*. *ACM SIGGRAPH Computer Graphics*, *16*. doi:10.1145/965145.801263

Sciretta, P. (2016). *How Unprecedented New Technology Made It Harder to Produce 'Finding Dory'*. Retrieved from slashfilm.com/the-tech-of-finding-dory

Thompson, S. P. (1910). *Calculus made easy: Being a very-simplest introduction to those beautiful methods of reckoning which are generally called by the terrifying names of the differential calculus and the integral calculus*. MacMillan and Co.

Weisstein, E. W. (2017). *Koch snowflake*. Retrieved from mathworld.wolfram.com/KochSnowflake.html