Exploring 3D Fractal Rendering

Noah Márquez, Oscar de Caralt, Alejandro Guzmán, and Adrià Alfonso
(Dated: June 7, 2023)

CONTENTS

I. ABSTRACT

This project provides a comprehensive exploration of the visualization of 3D fractals and procedural terrains, taking into account various techniques and theoretical frameworks.

In the first section we explore the generation and visualization of procedural terrains using Gradient Fractal Noise and Chunks. Here, we use Perlin noise and octaves to generate height maps for realistic terrains, highlighting the importance of manipulating parameters for realism. We further delve into chunk generation and discuss the relevance of Level of Detail (LOD) algorithms in resource management for successful open world gaming.

Secondly, the article delves into Iterated Function Systems (IFS) and their visualization in 3D using raymarching. The process of raymarching and its application for IFS fractals is examined in detail, highlighting its strengths and challenges. Special emphasis is given to handling high computational load, managing abrupt color changes, and achieving re-

alistic visual effects through optimal parameter selection.

Quaternion Julia Sets, a unique type of 3D fractal, are generated through quaternion algebra and represented using a technique called Distance Estimation. Real-time rendering challenges such as dimension reduction, inverse iteration algorithm, and lightning model are discussed in detail.

Lastly, the project explores L-Systems and their recursive rule-based nature that results in complex 3D structures. The utilization of Turtle Graphics to visualize L-systems is explained, with discussions on branching and stochastic L-systems, and other applications.

The applications of this study span across numerous fields including computer graphics, video game development, animation, scientific visualization, virtual reality, and even areas like architecture and natural sciences for the representation of natural phenomena and structures.

Overall, this project gives an in-depth understanding of the process, challenges, and solutions associated with the visualization of complex 3D structures, providing a stepping stone for further exploration and development in the field.

II. INTRODUCTION

As computer graphics and visualizations continue to evolve, the importance of accurately rendering 3D structures has become increasingly significant. This paper explores the visualization of 3D fractals using various techniques and theoretical frameworks, with a focus on Iterated Function Systems, Gradient Fractal Noise, Quaternion Julia Sets, and L-systems.

By examining the complexities of these systems and their visualization techniques, we aim to provide insights into the generation and portrayal of intricate structures, opening new doors for advancements in various sectors such as gaming, animation, and scientific visualization.

The underlying concepts of each method, their strengths, and challenges, and how to overcome these hurdles are discussed in depth. This work also delves into the implications and applications of these techniques in real-world scenarios, revealing their potential impact and significance.

The ensuing discourse is a comprehensive journey that uncovers the secrets behind creating realistic and fascinating virtual worlds, illuminating the path towards a future where digital and physical realities are indistinguishable. We trust this exploration will serve as a valuable resource for academics, practitioners, and enthusiasts alike in the field of 3D graphics and beyond.

III. TECHNIQUE 1: GENERATING AND VISUALIZING PROCEDURAL TERRAINS USING GRADIENT FRACTAL NOISE AND CHUNKS.

A. Theoretical Background.

1. Chunks.

In computer graphics, chunks refer to discrete, self-contained portions of data used to efficiently organize and render large-scale environments or virtual worlds. These chunks divide the virtual space into manageable units. By loading and unloading chunks dynamically based on the player's position, computational resources can be focused on the visible portions of the scene, improving performance and enabling seamless exploration. Chunks are commonly employed in games and simulations to handle the rendering of complex and expansive environments in a resource-efficient manner.

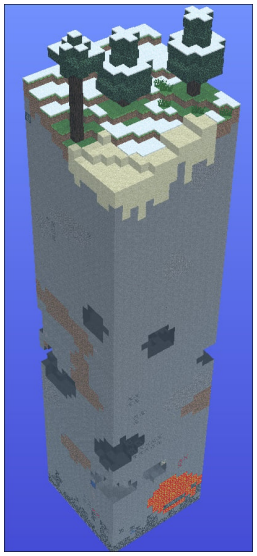


FIG. 1: Minecraft chunk.

2. Gradient Fractal Noise.

Fractal noise refers to a type of procedural texture that exhibits self-similar patterns at different scales. It is created by layering multiple octaves of noise together. Octaves, in this context, are successive layers of noise. Each octave contributes to the overall complexity of the final result. By adjusting the amplitude and frequency of each octave, a variety of natural and organic patterns can be generated. One of the most extended types of gradient noise is Perlin.

Perlin noise, developed by Ken Perlin, is a popular type of gradient-based coherent noise. It creates smooth, continuous patterns that simulate natural textures and ran-

domness. Perlin noise is generated by interpolating values from a grid of random gradient vectors. The smooth interpolation creates coherent and visually pleasing patterns that are widely used in computer graphics, terrain generation, and procedural content creation.

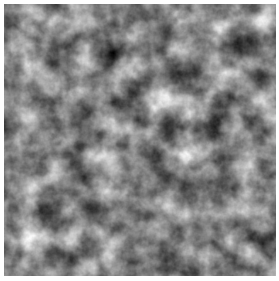


FIG. 2: Perlin noise.

B. Generation of procedural terrain using Perlin noise and octaves.

Procedural terrain generation is a technique used in computer graphics and game development to create realistic and diverse landscapes automatically. One popular method for generating procedural terrain is by utilizing Perlin noise with octaves. In this first technique overview, we will explore the concept of height maps, the role of octaves in adding detail and complexity, and how we convert Perlin noise into terrain.

1. Height maps.

Height maps serve as a fundamental representation of terrain, encoding elevation data in a two-dimensional grid. Each point on the height map corresponds to a specific position on the terrain and represents the height or elevation value at that point. When generating terrain using a noise height map (extracted from a Perlin noise function) without incorporating octaves, the resulting landscape tends to exhibit a lack of complexity and variation. This occurs because the noise values are sampled at regular intervals, leading to a grid-like pattern that lacks the intricate details found in real-world terrains. The absence of octaves limits the range of frequencies present in the height map, resulting in a monotonous and less realistic representation of the terrain.

2. Introducing octaves for enhanced realism.

To overcome the limitations of using a single noise height map, we can introduce octaves, which involve layering multiple noise maps of varying frequencies on top of each other. Each octave adds detail to the overall height map, capturing different levels of variation present in natural landscapes.

To further illustrate this concept, let's consider the example of a sine function, sin(x). This function is monotonous and infinite, lacking any variations. If we were to use trigonometric functions alone to generate our procedural world, it would result in a terrain with no diversity or randomness.

However, by incorporating octaves into the process, the function used to create our noise height map becomes more detailed and random. Just as adding octaves to a sine function introduces additional frequencies and amplitudes, our height map benefits from the increased complexity. This enhancement leads to a more realistic representation of the terrain, with a multitude of variations and intricate details.

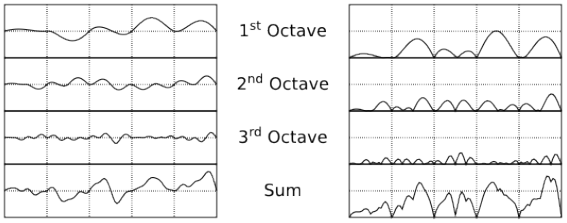


FIG. 3: Octave Layering.

Further information about the usage of octaves in Minecraft. [1]

Algorithm 1: Generate non-fractal Perlin noise.

Input : $width, height, scale$
Output: $noise\_map$
Function ***perlin***$(width, height, scale)$**:**
  $noise\_map = create\_array(height, width)$
  for $y = 0$ **to** $height - 1$ **do**
    for $x = 0$ **to** $width - 1$ **do**
      $noise\_value =$
       $noise.snoise2(x/scale, y/scale)$
      $noise\_map[y][x] = noise\_value$
  return $noise\_map$

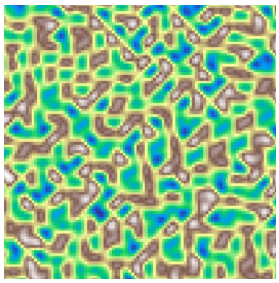


FIG. 4: Non-Fractal Perlin noise.

Algorithm 2: Generate fractal Perlin noise.

Input : $width, height, scale, octaves$
Output: $noise\_map$
Function ***perlin***$(width, height, scale, octaves)$**:**
  $noise\_map = create\_array(height, width)$
  for y *from* 0 *to* $height - 1$ **do**
    for x *from* 0 *to* $width - 1$ **do**
      $amplitude = 1$
      $frequency = 1$
      $noise\_value = 0$
      for _ *from* 0 *to* $octaves - 1$ **do**
        $noise\_value+ =$ noise.snoise2$(x \cdot frequency/scale, y \cdot frequency/scale) \cdot amplitude$
        $amplitude* = 0.5$
        $frequency* = 2$
      $noise\_map[y][x] = noise\_value$
  return $noise\_map$

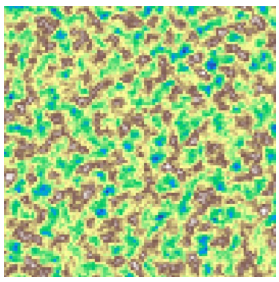


FIG. 5: Fractal Perlin noise.

3. Manipulating parameters.

When working with Perlin noise, manipulating certain parameters can significantly impact the generated patterns. By increasing the "scale" parameter, you can create larger and smoother patterns, which is particularly useful for generating expansive terrains or landscapes. Additionally, adjusting the "width" and "height" parameters allows you to control the size of the generated noise map. Adding more octaves increases the level of intricacy in the pattern, while reducing the number of octaves simplifies it.

Through experimentation with these parameters, you have the flexibility to customize the Perlin noise generation to suit your specific needs, whether it's generating vast terrains or fine-grained textures.

C. Chunk generation for optimization.

In the rapidly advancing era of technology, the demand for efficient utilization of machine resources has become paramount. As computing power continues to scale, so does the need for intelligent resource management strategies that maximize performance while minimizing resource consumption. One such strategy that has gained significant traction in recent years is the utilization of chunks.

Chunks, as I mentioned before, refer to the division of data or tasks into smaller, manageable units. They are used in order to break down complex problems into more digestible portions for our computers.

The challenge here is: How can we ensure that chunks are generated and fitted correctly? How can we create the illusion of a cohesive world?

1. Resource management challenge for illusory world-building.

In this topic, we will explore the concept of terrain optimization using chunks. The key idea behind this approach is to dynamically load and render only the chunks that are within the observer's immediate vicinity. By doing so, we can avoid rendering distant or occluded parts of the terrain, saving both processing power and memory. This technique is especially valuable in open-world games or virtual environments where the terrain can span vast distances.

But how can the chunks fit correctly? To ensure that the generated terrain is well-fitted and makes sense, we can use offsetX and offsetY values when generating the chunks. These values represent the position of each chunk relative to a reference point, such as the observer's position or the center of the terrain. By using these offsets, we can align the chunks seamlessly and create a smooth transition between neighboring chunks. This allows for a more natural and visually appealing terrain generation. The specific values of offsetX and offsetY will depend on the chosen chunk size and the algorithm used for generating the terrain.

2. Level of detail (LOD) algorithm.

In the context of open world rendering with chunks, let's consider a terrain as an example. Each chunk is represented by a mesh, consisting of a collection of vertices and triangles that define its shape.

To determine the level of detail for each chunk, the LOD algorithm employs a distance-based metric. The distance between the observer (camera) and the chunk is measured, and based on this distance, the algorithm determines the LOD level for the chunk. This LOD level determines the number of vertices used to render the chunk's mesh.

When a chunk is far away from the observer, the LOD algorithm reduces the level of detail by decreasing the number of vertices in the chunk's mesh. This reduction is typically achieved through various techniques, such as mesh simplification or using pre-generated lower-detail versions of the chunk's mesh called "LOD meshes."

As the observer moves closer to a chunk, the LOD algorithm incrementally increases the level of detail by adding more vertices to the chunk's mesh. This process can involve techniques such as mesh refinement or using higher-detail LOD meshes. By dynamically adjusting the level of detail based on the viewer's proximity, the algorithm ensures that objects appear more detailed as the viewer approaches them, enhancing the visual fidelity of the scene.

The LOD algorithm continuously monitors the distance between the observer and the chunks in the scene, making real-time adjustments to maintain an optimal balance between visual quality and performance. This approach allows rendering engines to handle large, open worlds efficiently, as only the necessary level of detail is computed and rendered for each chunk based on its distance to the observer.

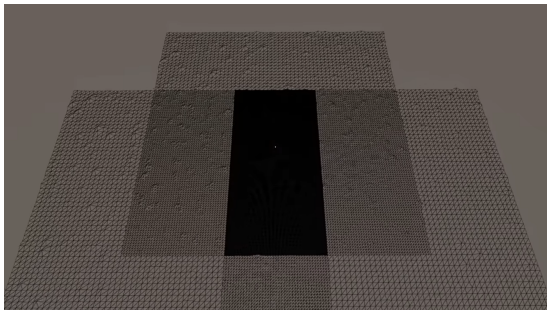


FIG. 6: LOD algorithm viusalization. [2]

D. Three-dimensional Perlin noise for PCG.

2D Perlin noise is useful for generating two-dimensional features on the surface of the procedural generated terrain, such as hills, mountains, and valleys. It is effective in simulating smooth and continuous variations in elevation and other terrain attributes.

However, when aiming to create complex structures in three dimensions, such as underground caves, cliffs, or rock formations, 3D Perlin noise becomes necessary. 3D Perlin noise adds an additional dimension of variation to the terrain, allowing for more realistic simulation of three-dimensional details and features.

To understand better this non-intuitive concept, let's try to explain the density values of the 3D grid in Voxel worlds like Minecraft.

1. Density value for voxel-based worlds.

In voxel-based procedural worlds, 3D Perlin noise is used to generate terrain and determine the presence of blocks within the world. Instead of using the noise value as a height indicator, it is treated as density. If the noise value is greater than or equal to 0, a block is placed, otherwise, it represents empty space (air).

The noise function is evaluated for every potential block position, and if the noise value is positive, a block is generated. This approach creates varied landscapes where lower areas are mostly solid and higher areas are mostly empty.

To further refine the generated terrain, additional algorithms may be employed to determine the type of block to be placed and to carve out caves. However, these processes are separate from the direct usage of the noise function.

2. Marching cubes.

In worlds like Minecraft, the usage of 3D Perlin noise as described earlier is effective because the terrain is represented by voxels. However, in polygon-based worlds, simply sampling the noise function would not be sufficient. Instead, additional algorithms are required to convert the noise samples into a surface representation and create polygons that approximate this surface.

One commonly used algorithm for this purpose is marching cubes. It examines the density values from the noise function at each voxel location and determines the configuration of polygons needed to approximate the surface. By connecting the vertices of these polygons, a

smooth and continuous surface can be generated within the polygon-based world.

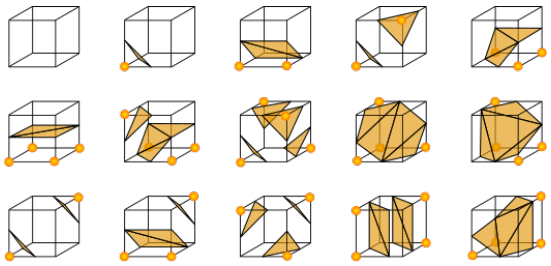


FIG. 7: MC algorithm viusalization. [3]

IV. TECHNIQUE 2: VISUALIZING 3D IFS FRACTALS USING RAY MARCHING.

A. Theoretical Background.

1. *Iterated Function Systems.*

The Iterated Function Systems (IFS) are a mathematical method primarily used for generating intricate and often beautiful fractals. IFS use a collection of transformation functions, each applied repeatedly to the output of the previous one in a process known as iteration. These transformations can include scaling, rotation, translation, and other affine transformations. The result is a set of points that form a fractal.

While the mathematical underpinnings of IFS might seem complex, the results they produce have far-reaching implications, not only within mathematics, but also in diverse fields like digital art, nature simulations, and data compression. IFS offer a structured and mathematically rigorous approach to creating fractals, an area of study that continues to captivate scientists, mathematicians, and artists alike due to its intriguing blend of chaos and order.

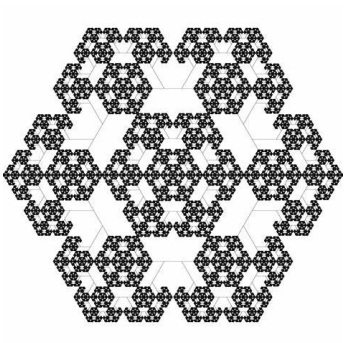


FIG. 8: Example of a 2D fractal using IFS. [4]

2. *Mathematical Foundations of IFS.*

Iterated Function Systems are often used to generate fractal patterns. These patterns are widely recognized for their complex, self-similar shapes, which can resemble natural phenomena like trees or coastlines.

Imagine an IFS as a group of transformations – a bit like functions in a programming language – that are applied repeatedly to a starting point or set of points. These transformations might involve scaling (like zooming in or out), rotating, or moving the points.

In more formal terms, an IFS $\mathcal{F} = \{f_i\}_{i=1}^{n}$ is a set of transformations $f_i : X \to X$ where X is the set of points we're working with. Each f_i is a 'contraction mapping', which basically means it brings points closer together. This is defined by a 'contractivity factor' s_i, a value between 0 and 1 that determines how much closer together points get.

In other words, for every pair of points x, y in X, applying f will make the distance between the transformed points no greater than s times the original distance:

$$d(f(x), f(y)) \leq s \cdot d(x, y)$$

The reason we're interested in these contractions is that when we apply all of them over and over again to our starting set, we end up with a fractal – a shape that's self-similar at different scales. This is a direct consequence of the Collage Theorem, which states that if the transformations in our IFS $\mathcal{F}$ collectively approximate a certain shape, then the repeated application of $\mathcal{F}$ will also approximate that shape.

In practice, to generate these fractal shapes, we can use an algorithm called the 'Chaos Game'. Starting from an arbitrary point x, we randomly pick one of our transformations f_i and update x to $f_i(x)$. If we repeat this process enough times, the points x will trace out a fractal pattern.

This is just scratching the surface of IFS, and there are many more concepts to explore, including probabilities, non-contracting IFS, and more. But hopefully, it provides a solid foundation for understanding how these fascinating fractals are generated.

B. Visualizing 3D IFS fractals using raymarching.

Raymarching, an advanced rendering technique, provides an excellent tool for creating visually impressive and accurate representations of 3D IFS fractals [5]. Leveraging the principles laid out in the Collage Theorem, raymarching brings fractals to life by accurately rendering their intricate designs in three-dimensional space.

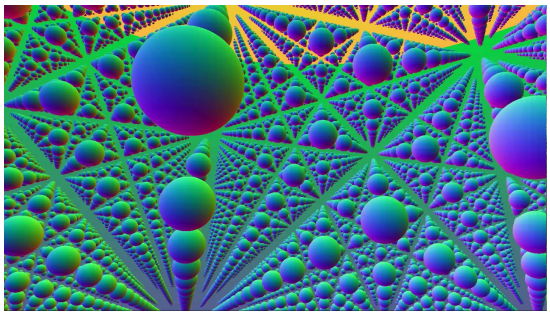


FIG. 9: Example of a 3D fractal using raymarching. [6]

1. *The Process of raymarching.*

Raymarching estimates the intersection between a ray and an object by incrementally advancing the ray in steps. Refer to [7], the step size is determined by a distance estimator function, which provides an approximation of the distance from the current point on the ray to the nearest surface in the 3D scene. Thus, the accuracy of the rendering and the efficiency of the raymarching algorithm is heavily dependent on the quality of the distance estimator function.

For every pixel in the image, a ray is cast from the camera position through the pixel into the scene. It is this ray for which we perform raymarching, calculating the color of the pixel based on how and where this ray interacts with the surfaces within the scene.

Here is a simple representation of the raymarching algorithm in pseudocode for every ray:

Algorithm 3: Raymarching algorithm.

Input : *origin* - the starting point of the ray,
direction - the direction of the ray,
max_steps - the maximum number of steps the ray can take,
max_distance - the maximum distance the ray can travel
bg_color - the color of the background
Output: *color* - the color at the hit point or the background color

```
Function rayMarch(origin, direction, max_steps, max_distance):
    for step from 0 to max_steps − 1 do
        distance = estimateDistance(origin)
        if distance < threshold then
            return getSurfaceColor(origin) - return the color of the hitted surface
        origin+ = direction × distance - move the ray forward by the step size
    return bg_color - the ray did not hit a surface, return the background color
```

As we can see in the provided algorithm, we iterate over all steps specified in the raymarching algorithm, and for each one, we estimate the distance to the surface of the fractal. If the distance is less than a certain threshold, the step at which the ray has hit the surface is returned. If the ray has not hit the surface, we update the origin of the ray, moving it according to the size of the step taken. In the event the ray does not hit the surface of the object, the steps that have been taken are returned.

The execution of this algorithm can be visually observed in the figure, where it can be seen that at each step, a distance is established, and it stops at the step where the distance is minimal to the threshold.

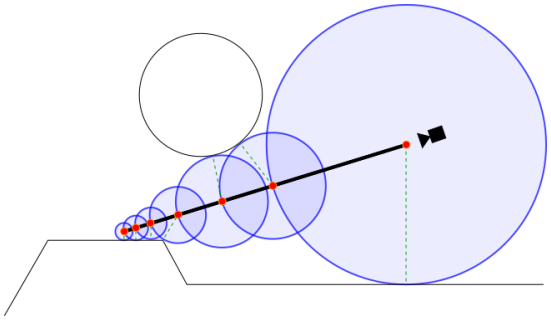


FIG. 10: Example of the execution of the raymarching algorithm. [8]

Also is need to remember that the raymarching function is called for each pixel, following the next algorithm:

Algorithm 4: Render objects using raymarching.

Input : *rayOrig* - the position of the camera,
rayDir - the direction of the ray,
maxSteps - the maximum number of steps the ray can take,
maxDist - the maximum distance the ray can travel
Output: *color* - the color of the pixel

```
Function renderObject(rayOrig, rayDir, maxSteps, maxDist):
    for each px in image do
        rayOrig = camPos - set the origin of the ray to the camera position
        rayDir = getRayDir(camPos, px) - get the direction of the ray through the pixel
        color = rayMarch(rayOrig, rayDir, maxSteps, maxDist) - apply the rayMarch algorithm to this ray
        px.setColor(color) - set the color of the pixel
```

As we can see in Algorithm 4, for each pixel in the image, a ray origin is computed and the ray direction is obtained. Then, the function defined in Algorithm 3 is called, which calculates the color at that pixel according

to the applied raymarching.

As a brief conclusion, the raymarching process is computationally straightforward, although there are important considerations for correctly rendering certain objects, such as accurately estimating the distance from the ray to the object's surface or determining the moment when the distance is minimal to define a collision. These aspects become more crucial when rendering computationally complex objects, as we will see next with IFS fractals.

2. Rendering IFS fractals using raymarching.

In order to render a fractal generated by an Iterated Function System (IFS) using raymarching, we need to compute a custom Signed Distance Function (SDF) for that fractal. The Signed Distance Field is a representation of a shape used for raymarching, where for every point in space, we calculate the shortest distance to the surface of that shape.

In other words, if raymarching is being performed through an IFS fractal, the SDF of the IFS can be used as the distance estimation function in the raymarching algorithm (see Algorithm 3). Recall that the distance estimation function is used to determine how far the ray can "march" along its path before hitting a surface.

Computing an SDF from an IFS can be challenging, mainly because fractals created with IFS often have an infinitely complex structure. In practice, various approximate and heuristic approaches are employed.

One of the most common methods for computing an SDF for an IFS is the "Closest Point Iteration" method, which involves repeatedly iterating through the transformations of the IFS until we converge to a fixed point.

Here is the basic process:

1. We start with an arbitrary point in space, call it **p** (the point of the ray from which we want to measure the distance to the fractal).

2. We apply all the transformations of the IFS to p and find the transformation that moves **p** closest to itself (i.e., we find the closest fixed point to **p**).

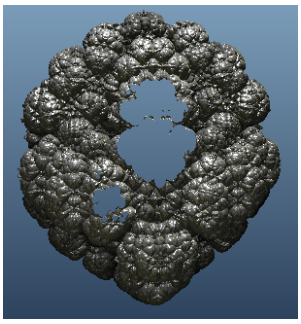
3. We repeat this process until **p** no longer changes significantly. This is our closest fixed point to **p**.

4. The distance between **p** and the closest fixed point is a good candidate for the signed distance function.

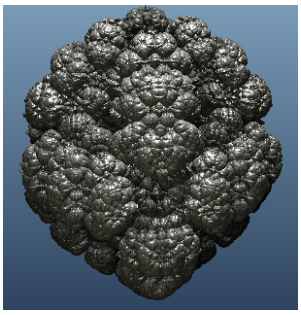
This method is not perfect and does not produce a true SDF in all cases (the resulting function may not always be a distance function in the strict mathematical sense), but it is usually good enough for most raymarching applications. Plus, various optimizations and heuristics can be employed to improve the speed and quality of the approximation.

Once we have a function that can approximately estimate the distance from a ray to the surface of a fractal generated by an IFS, we can render that fractal. In the context of raymarching, a point is considered to be 'inside' the fractal if the estimated distance to the fractal is less than a certain threshold. This threshold, like the distance estimator function, plays a critical role in determining the quality and accuracy of the rendered fractal.

A low threshold could cause us to terminate our march prematurely, leading to an under-sampling of the fractal and a potential loss in detail. Conversely, a high threshold might cause us to march too far, potentially overshooting the fractal surface and missing key features. Balancing this threshold is therefore crucial for producing a rendering that accurately reflects the intricate details of the fractal, while also maintaining computational efficiency. In the figure 11 can see the difference between choose a good threshold or not.



(a) IFS fractal rendered with an overly high threshold



(b) IFS fractal rendered with an appropriate threshold

FIG. 11: Comparison between rendering the same IFS fractal with different thresholds [9]

Up to this point, we have covered the basic process of rendering IFS fractals using the raymarching algorithm. However, there are various considerations to take into account, such as calculating the SDF and choosing an optimal threshold, in order to optimize the results obtained. As we will see later on, rendering this type of fractal using raymarching is more efficient compared to other rendering algorithms like raytracing and z-buffer.

3. Raymarching, Raytracing or Z-buffer for IFS Fractals?

Raytracing, raymarching, and z-buffering are all techniques used for rendering three-dimensional scenes,

but they each have different strengths and weaknesses when applied to IFS fractals.

Raytracing is a technique that works by tracing the path of light through pixels in an image plane and can produce highly realistic images by accurately modeling the physical behavior of light. However, it typically requires precise geometric data and can be computationally expensive due to necessary intersection tests. Rendering IFS fractals, which are characterized by intricate, self-similar patterns, can thus be particularly challenging with raytracing due to its need for exact geometric detail.

Raymarching, also known as sphere-tracing, incrementally advances along a ray until a surface is hit, with the step size determined by a distance estimator function. This function approximates the distance from the current point on the ray to the nearest surface in the 3D scene. Raymarching can often be the more favorable method for rendering IFS fractals due to a few reasons:

1. **Approximation Tolerance**: Raymarching allows for a level of approximation tolerance, beneficial for dealing with the infinite complexity of IFS fractals. The distance estimator function can provide a good approximation of the fractal surface without the need for exact geometric calculations.

2. **Efficiency**: Raymarching's incremental stepping can be more computationally efficient when dealing with complex fractal structures, as opposed to raytracing's need for computing intersections with potentially infinite detail.

3. **Flexibility**: Raymarching can be easily adapted to render different kinds of fractal structures by changing the distance estimator function.

Z-buffer, a technique used in rasterization-based rendering pipelines, stores depth information for each pixel, ensuring that objects closer to the viewer are drawn in front of objects further away. However, it assumes distinct, polygonal surfaces to rasterize, which is not the case with IFS fractals. The issues with applying z-buffering to IFS fractals include:

1. **Depth Complexity**: IFS fractals often exhibit depth complexity, where a single view ray might intersect the fractal at multiple points. Z-buffering usually only keeps track of the closest intersection, adequate for polygonal scenes but insufficient for complex fractals.

2. **Absence of Polygons**: IFS fractals are not comprised of distinct polygonal surfaces, upon which z-buffering relies. Consequently, it's less suited to fractal rendering.

3. **Efficiency**: Similar to raytracing, z-buffering can be less efficient when dealing with complex fractal structures. The entire scene needs to be rasterized, even if only a portion of it is visible, which can be computationally demanding.

In conclusion, while each of these methods has their uses and contexts where they shine, raymarching often provides a more accurate, efficient, and flexible method for rendering IFS fractals compared to raytracing and z-buffering.

C. Challenges and solutions in rendering fractals and other objects with raymarching.

Rendering fractals and other objects using raymarching can present several challenges. These include computational inefficiencies, difficulties in achieving realistic visual effects, choosing optimal parameters, and managing abrupt color changes. Let's explore these challenges further and propose potential solutions.

1. Handling High Computational Load.

Rendering fractals, especially in three dimensions, can require a significant amount of computational resources. Refer to [7], this is due to the large number of rays to march, the steps required for each ray, and the complex calculations needed for distance estimation.

One possible solution is leveraging parallel processing capabilities, such as using the compute shader functionality provided by modern Graphics Processing Units (GPUs). This allows us to process multiple rays concurrently, greatly reducing the overall computation time.

2. Achieving Realistic Visual Effects.

Creating realistic visual effects like shadows, reflections, and proper lighting involves complex calculations, which add another layer of computational demand to the rendering process. However, these effects are essential for producing high-quality visualizations.

A potential solution is to implement an efficient lighting model such as the Blinn-Phong illumination model. This model considers three components - ambient light, diffuse reflection, and specular reflection - to simulate realistic lighting. The addition of shadows and reflections can also be achieved with careful modification of the raymarching process, as discussed earlier.

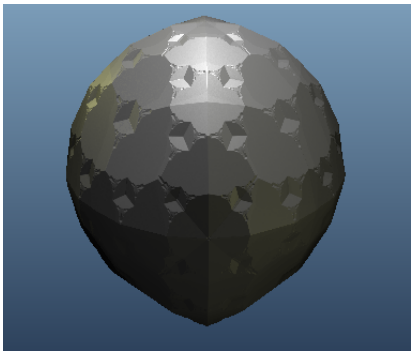


FIG. 12: Example of a 3D IFS fractal rendered using raymarching and Blinn-Phong shading. [9]

3. *Choosing Optimal Parameters.*

The parameters of the raymarching algorithm, such as the step size and the threshold for determining when a ray intersects the fractal, have a significant impact on the quality and accuracy of the final image.

Finding optimal parameters often requires a balance between rendering time and visual quality. Parameters can be adjusted dynamically based on various factors like the complexity of the fractal, the desired level of detail, and the computational resources available.

4. *Managing Abrupt Color Changes.*

Smooth transitions in coloring can make the resulting fractal image more visually appealing. Abrupt color changes can be managed by using smooth coloring techniques. This involves modifying the color assignment process to smooth out transitions between different color ranges, providing a gradient effect.

A basic implementation of smooth coloring might involve interpolation between colors based on the number of steps taken by the raymarching algorithm before intersecting the fractal:

```
smoothColor = mix(color1, color2,
smoothstep(minSteps, maxSteps, currentStep))
```

Addressing these challenges can greatly improve the quality and efficiency of IFS fractal rendering with raymarching, leading to more impressive and insightful visualizations.

V. TECHNIQUE 3: QUATERNION JULIA SETS

A. Julia Sets

Julia Sets were introduced by the French mathematician Gaston Julia around 1918 [10]. They are renowned types of fractals created via iterative calculation. A "filled-in" Julia Set consists of all points, represented as z, in a repeatedly applied complex function $f(z)$ that do not become infinite as the function's iterations reach infinity. The actual Julia set is the edge of this filled-in set. While this definition can include a multitude of functions, the analysis is typically confined to a subset of complex functions.

For our consideration, we use a well-known quadratic expression $f(z) = z^2 + c$, where z is a variable in the form $a + bi$ (a and b are real numbers, $i = \sqrt{-1}$ symbolizes the imaginary unit), and it can take all values in the complex plane. The element c is also a complex number, but it remains constant for each individual Julia set, thus it is called a parameter. Hence, there are countless Julia sets, each one defined for a particular value of c.

Every pixel in the image complex plane symbolizes the initial point z_0 for the iterative series $f(z_n) = z_{n+1} = z_n^2 + c$. For any given initial value of z, or z_0, the iterated values of $f(z)$ can either keep growing indefinitely or remain bounded as n increases towards infinity. Points z_0 that don't stay limited during successive iterations of $f(z)$ are considered part of the escape set E_c. The rest of the points that remain limited as n goes to infinity are called prisoners and are considered part of the prisoner set P_c defined for a certain c. Every point must belong to either of these sets. The Julia set J_c, defined for a specific c, is the shared boundary between the escape set and the prisoner set.

The question of whether the boundary points (the Julia set) are part of the prisoner set appears to be a matter of debate. However, it must be true since the complex plane is exclusively divided into prisoner and escaping points. Consequently, the boundary points have to be in P_c as they cannot be in E_c (since they don't escape with repeated iterations). Julia sets, which are boundary points, do not always surround any interior prisoner points. Since P_c is what's left of the complex plane after E_c is removed, the boundary and prisoner points must align. An example of such a set is J_c for $c = i$ (refer to Fig.13).

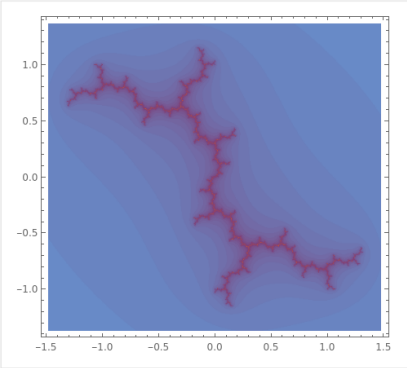


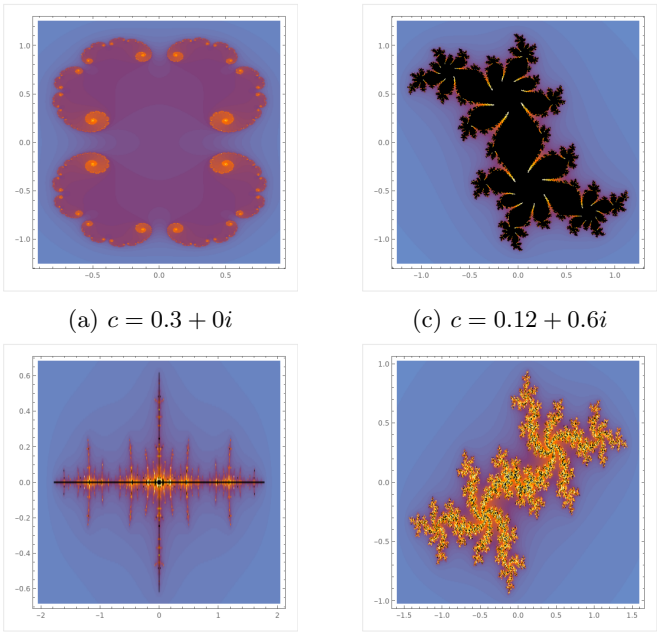
FIG. 13: Julia set for $c = 0 + i$. *(Image generated using ChatGPT + Mathematica plugin.)*

The method to sketch a Julia Set involves calculating the iterative series $z_{n+1} = z_n^2 + c$ for each pixel. If the series tends towards infinity, the pixel is painted one color; if not, it is painted a different color. The divergence or convergence of the series is not always apparent, and a high number of iterations might be necessary for discernment. A series is presumed to trend towards infinity if its value exceeds a particular threshold. If the series doesn't diverge after a specific number of terms, it's classified as part of the prisoner set. Both of these determinations can be adjusted for creating detailed images that demand more computation time. Moreover, the speed at which a point diverges towards infinity can be used for coloration, which adds an interesting effect.

For computer execution, a threshold radius $r_c = \max(|c|, 2)$ serves as a helpful test criterion. If an orbit z_k ever surpasses this threshold radius r_c, it's certain to escape towards infinity, indicating that the starting point belongs to the escape set [11].

Julia Sets can either be connected (known as Fatou sets) or disconnected (referred to as Cantor sets or Fatou dust). These disconnected sets, often labeled as 'dust', are composed of distinct points irrespective of the resolution at which they're observed. The famous Mandelbrot set M, which was identified by Benoit B. Mandelbrot around 1979, is defined as the collection of all c parameter values for which the corresponding Julia sets are connected. Julia sets are connected if c values are chosen from within the Mandelbrot set and are disconnected when c values are selected from outside the Mandelbrot set. Additionally, the Mandelbrot set M can be defined as the set of c values for which the orbits (successive iterations) of $z = 0 + 0i$ stay bounded [12]. This point $0 + 0i$ is referred to as the critical point for Julia sets. The simple test of boundedness of iterations of 0 reveals whether a Julia set is connected, a result independently discovered by Julia and Fatou.

Julia sets have another feature concerning the different domains of c. If c is entirely real (of the form $c = a + i0$), the Julia set is mirrored about the real axis, meaning the part of the set below the real axis is a reflection of the part above. Other c values with a non-zero imaginary component result in Julia sets having 180° rotational symmetry. The constant c determines the fractal's shape, and significant variations in shapes can be seen by adjusting c.



(a) $c = 0.3 + 0i$ (c) $c = 0.12 + 0.6i$

(b) $c = -1.4 + 0i$ (d) $c = -0.55 - 0.55i$

FIG. 14: (a) & (b) Reflection symmetrical Julia sets and (c) & (d) Rotation symmetrical Julia sets. *(Images generated using ChatGPT + Mathematica plugin.)*

B. Quaternion Julia fractals

Typical complex numbers of the form $a + bi$ facilitate a two-dimensional representation of a fractal. However, this concept can be expanded to higher dimensions. Alan Norton discovered in 1982 that Julia sets are not confined to the complex plane but also extend to the 4-dimensional quaternions. Quaternionic Julia sets are nontrivial and possess many characteristics not found in complex Julia sets [13]. Quaternion Julia fractals are generated using the same method as the standard Julia set, but they utilize 4-dimensional quaternions instead of 2-dimensional complex numbers.

1. Quaternions

a. Definition of quaternions

Quaternions were first introduced by Sir William Rowan Hamilton in 1843 [14]. A quaternion, denoted as q, is ex-

pressed as $q = (q_r, \vec{q}) = q_r + q_i i + q_j j + q_k k$, where q_r represents the scalar part and $\vec{q} = q_i i + q_j j + q_k k$ composes the vector part. The four-tuple of distinct real values (q_r, q_i, q_j, q_k) can be viewed as a vector in the 4D quaternion space and is associated with one real axis and three orthogonal imaginary axes: i, j, and k. These quaternions comply with the following properties:

$$i^2 = j^2 = k^2 = ijk = -1,$$
$$ij = -ji = k, jk = -kj = i, ki = -ik = j$$

b. Quaternion operations

1. Addition:

   $p+q = (p_r+p_i i+p_j j+p_k k)+(q_r+q_i i+q_j j+q_k k) = (p_r + q_r) + (p_i + q_i)i + (p_j + q_j)j + (p_k + q_k)k$

2. Scalar multiplication

   $\lambda q = (\lambda q_r, \lambda \vec{q}) = \lambda q_r + \lambda q_i i + \lambda q_j j + \lambda q_k k$

3. Multiplication

   $pq = (p_r + p_i i + p_j j + p_k k) \cdot (q_r + q_i i + q_j j + q_k k) = (p_r q_r - p_i q_i - p_j q_j - p_k q_k) + (p_r q_i + p_i q_r + p_j q_k - p_k q_j)i + (p_r q_j + p_j q_r - p_i q_k + p_k q_i)j + (p_r q_k + p_k q_r + p_i q_j - p_j q_i)k$

   It can also be written as:

   $pq = (p_r, \vec{p}) \cdot (q_r, \vec{q}) = (p_r q_r - \vec{p} \cdot \vec{q}, p_r \vec{q} + q_r \vec{p} + \vec{p} \times \vec{q})$

4. Conjugate

   $q^* = (q_r, -\vec{q}) = q_r - q_i i - q_j j - q_k k$

5. Modulus

   $|q| = \sqrt{qq^*} = \sqrt{q^* q} = \sqrt{q_r^2 + q_i^2 + q_j^2 + q_k^2}$

6. Inverse

   $q^{-1} = \frac{q^*}{|q|^2}$

Note that quaternion multiplication is non-commutative ($pq \neq qp$). A quaternion with zero scalar part ($q_r = 0$) is called a pure quaternion, and a quaternion with unit module ($|q| = 1$) is called a unit quaternion.

c. Quaternion trigonometric and polar representations

Any quaternion may be represented in trigonometric form.

$$\begin{aligned} q &= q_r + q_i i + q_j j + q_k k = q_r + \vec{q} \\ &= |q|(\frac{q_r}{|q|} + \frac{\vec{q}}{|\vec{q}|} \cdot \frac{|\vec{q}|}{|q|}) \\ &= |q|(cos\theta + \mu sin\theta) \end{aligned}$$

where

$$cos\theta = \frac{q_r}{|q|} = \frac{q_r}{\sqrt{q_r^2 + q_i^2 + q_j^2 + q_k^2}}$$
$$sin\theta = \sqrt{1 - cos^2\theta} = \frac{\sqrt{q_i^2 + q_j^2 + q_k^2}}{\sqrt{q_r^2 + q_i^2 + q_j^2 + q_k^2}} = \frac{|\vec{q}|}{|q|}$$
$$\mu = \frac{\vec{q}}{|\vec{q}|} = \frac{q_i}{|\vec{q}|} i + \frac{q_j}{|\vec{q}|} j + \frac{q_k}{|\vec{q}|} k$$

is a unit pure quaternion, and $0 \leq \theta \leq \pi$.

Similar to the complex case $e^{i\,\theta} = cos\theta + isin\theta$, with the help of the Taylor series, one can prove a quaternion can be written in polar representation:

$$q = |q|(cos\theta + \mu sin\theta) = |q| e^{\mu\,\theta}$$

Proof:

Since $\mu^2 = -\mu \cdot \mu + \mu \times \mu = -1$, with help of the Taylor series:

$$\begin{aligned} e^{\mu\theta} &= 1 + \frac{\mu\theta}{1!} + \frac{(\mu\theta)^2}{2!} + \frac{(\mu\theta)^3}{3!} + \frac{(\mu\theta)^4}{4!} + \frac{(\mu\theta)^5}{5!} + \frac{(\mu\theta)^6}{6!} + \cdots \\ &= 1 - \frac{\theta^2}{2!} + \frac{\theta^4}{4!} - \frac{\theta^6}{6!} + \ldots + \mu\frac{\theta}{1!} - \mu\frac{\theta^3}{3!} + \mu\frac{\theta^5}{5!} - \ldots \\ &= \cos\theta + \mu \sin\theta \end{aligned}$$

C. Quaternion Julia Sets

The creation of quaternion Julia sets follows the same procedure as ordinary Julia sets, but utilizes quaternions in place of complex numbers. In order to generate a quaternion fractal, a function $q_{n+1} = f(q_n)$ is iterated. If the series remains bounded as n tends towards infinity, then the quaternion q_0 is considered a part of the Julia set. However, if the series diverges, then q_0 is deemed outside the Julia set. The most fascinating functions tend to be nonlinear. In the following, we use the simplest nonlinear function $q_{n+1} = q_n^2 + c$, where c is a constant specifying the particular quaternion. q_0 is a point in quaternion space. Another frequently used function is a cube, namely $q_{n+1} = q_n^3 + c$.

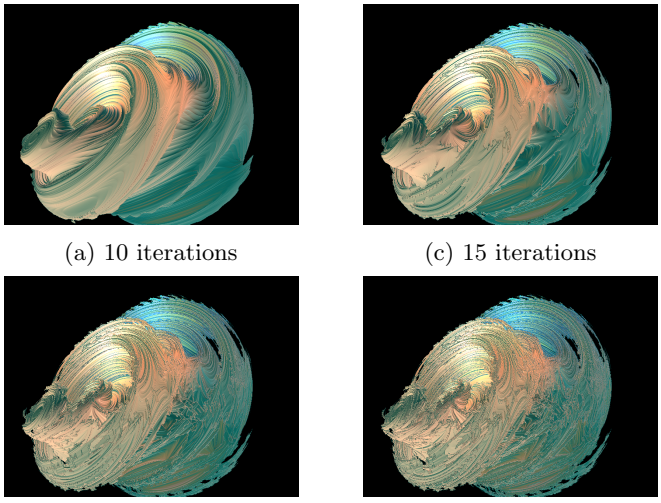
Unlike ordinary Julia sets that exist within a 2-dimensional space, quaternion Julia sets are housed within a 4-dimensional space. Their visualization in 2D is challenging, therefore, we usually observe their intersections with a 3-dimensional space (a hyperplane of the 4D space).

D. Real-time Rendering of Quaternion Julia Sets

a. Practical Considerations

Logically, it is impossible to iterate each point towards infinity.

Instead, a maximum number of iterations is predetermined and it is observed whether a point diverges before reaching this limit. If the series hasn't diverged by this point, the point is considered to be within the set. Practically, a reasonably small number of iterations produces satisfactory results. A higher number of iterations yields a more precise fractal surface, but this heightened accuracy can sometimes lead to less visually appealing results due to small-scale turbulence (refer to Fig.15). Moreover, an advantageous property of Julia sets is that they are constrained within a sphere of radius 2 centered at the origin. Consequently, if an iterated point deviates more than 2 units from the origin, it can be confidently discarded.



(a) 10 iterations (c) 15 iterations

(b) 20 iterations (d) 25 iterations

FIG. 15: Quaternion Julia sets for $c = -0.45 + 0.447i + 0.181j + 0.306k$ with different number of iterations

b. Dimension Reduction

Quaternion Julia sets, while being a natural extension of complex Julia sets into a 4-dimensional quaternion space, present a challenge for rendering due to the extra two dimensions. As we are unable to directly visualize a 4-dimensional object, a method of mapping these four dimensions into three or fewer is needed. There are a couple of methods we can use: one is to only render a single 3D slice of the quaternion space, while the other is to project the quaternion space into three dimensions, which gives us the 3D shadow of the Julia set. The method applied here involves intersecting the 4D object with a plane; this essentially makes one of the quaternion components (or dimension) a constant value.

In order to get a real sense of the quaternion fractal's true nature, it's necessary to generate a series of slices along an axis perpendicular to the plane of the slice and 'stack' the resulting 3D solid objects along the 4th axis. However, this is a complex task due to the limitations of our 3D visual system.

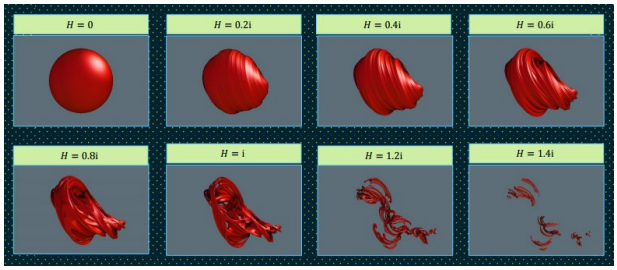


FIG. 16: Slices of quaternion Julia set [15]

c. Inverse iteration algorithm

The most immediate strategy for creating points in the set is to split the contracted 4D space into discrete segments and scrutinize each one. Though easy to execute, this approach is critically hindered by the time it consumes to achieve any meaningful level of accuracy, such as matching the size of the grid cells.

An alternative and superior rendering approach could be the inverse iteration algorithm. This algorithm is highly efficient in rapidly displaying the overarching structure of the Julia set [16]. Utilizing modern hardware, the convergence of z can be evaluated for every pixel in the complex plane, thereby enabling the production of detailed portrayals of intricate Julia sets at responsive rates. Despite its merits, this methodology is presently inefficient for dimensions exceeding two. A workaround could be to only compute and display the boundary points of the Julia set using inverse iteration. The terms in the sequence $z_{n+1} = \sqrt{z_n - c}$ gravitate towards the boundary of the complex Julia set for c. Each complex number's square root yields two values, one being the other's negative. By randomly selecting either the positive or negative root and plotting n z at every step, an approximate representation of the Julia set's boundary can be obtained. This method is almost identical in quaternion space, except for the handling of quaternion's square root, which in certain cases offers infinite values. In practice, we choose the two $sqrt(q)$ values that align on the i axis, thereby utilizing the inverse iteration $q_{n+1} = \sqrt{q_n - c}$ to calculate the estimated boundary points of the quaternion Julia set.

d. Ray tracing algorithm and distance estimation

The ray tracing technique can generate high-definition images of quaternion Julia sets at various levels of intricacy [17]. Additionally, the distance estimation method for the Julia set can also be taken into account [18]. The distance estimator provides the least distance from any given point outside the fractal to the fractal, and it is defined as follows:

$$\text{distance} \geq \alpha \cdot (|q_n| / |q_n^{'}|),$$
$$|q_n^{'}| = 2 \cdot |q_{n-1}| \cdot |q_{n-1}^{'}|.$$

The parameter α shares a similarity to the grid size, with smaller values (< 0.01) being essential for accurate outcomes. Larger values might lead to the formation of comparatively coarse fractals. Once a lower-bound for the distance is established, we progress along our ray until we either collide with the fractal or exceed the number of steps without encountering the fractal. A minor efficiency boost can be achieved by first colliding rays with a sphere of radius 2 centered at the origin. If the rays intersect this sphere, they are then tested against the fractal surface. This technique accelerates the tracing of rays that wouldn't intersect the fractal in any case.

e. Lighting Model

Given that a 3D model is employed to depict quaternion Julia sets, a suitable lighting model becomes a necessity. The Phong model was chosen due to its excellent performance and fast computation times [19]. Despite the need for surface normals in lighting computations, fractals possess infinitely disrupted surfaces. Consequently, we can only aim for a rough estimation of the surface's general direction. This is achieved by sampling the surface at nearby points.

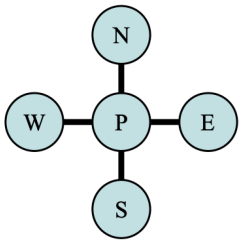


FIG. 17: Four neighboring points of a given point

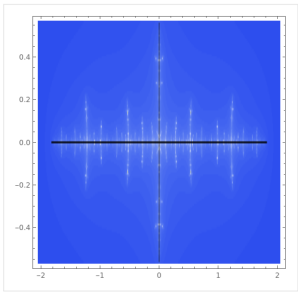
Upon identifying a point P within the Julia set, the four points neighboring it (as shown in Fig.17) are examined to confirm whether they lie inside or outside the set. These points are essential in the creation of a surrounding 'surface' and the computation of the associated normal vector $\vec{n}$. The code used to calculate the normal vector and the images shown is provided below [20] [21]. If each point is defined by a triplet (x, y, z) and N, S, W, E are the neighboring points, then the following holds:

$$\begin{cases} \vec{n} \cdot x = -(W.z - E.z) \cdot 2dy \\ \vec{n} \cdot y = -(S.z - N.z) \cdot 2dx \\ \vec{n} \cdot z = 2dx \cdot 2dy \end{cases}$$

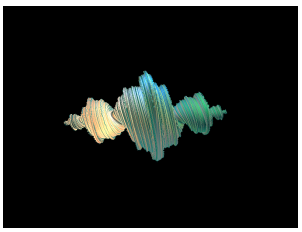
The coordinates of the normal vector $\vec{n}$ will subsequently be used to determine the intensity of the light ray I in accordance with the illumination model. The values of $\vec{n}$ are normalized. For further technical details and code implementation of the illumination model, refer to [19].

E. Examples and analysis

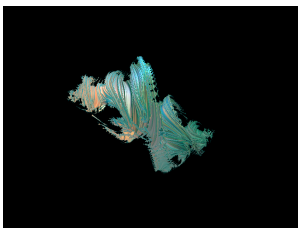
The following are all examples of quaternion Julia sets and related complex Julia set for different parameter c.



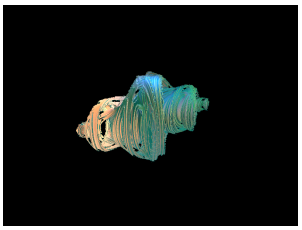
(a) $c = -1.5 + 0i$



(b) $c = (-0.8, 0.156, 0.0, 0.0)$



(c) $c = (-0.138, 0.864, 0.0, 0.0)$



(d) $c = (-0.55, -0.15, 0.45, -0.25)$

FIG. 18: Quaternion Julia sets and related complex Julia set

From the fractal examples, we can see that:

1. The conventional complex Julia set and quaternion Julia set exhibit key traits like intricate hierarchy, irregular boundaries, symmetrical structures, and the self-similarity of infinite recursive details, representing both chaos and order.

2. Due to the inclusion of the j and k components, quaternion Julia sets carry comprehensive spatial depth information, pronounced three-dimensional effects, and the ability to display fortifications, intertwining, irregular fringes, and a plethora of transformations. In comparison to complex Julia

sets, Quaternion Julia sets possess an added symmetry along the third dimension direction, exhibiting both rotational and reflection symmetry [22].

3. The orthogonal projection of the quaternion Julia set onto the complex plane and the corresponding complex Julia set exhibit similar outlines for the same parameter c. Variances in specific areas exist solely due to the different attractors of the two types of Julia sets. This suggests a close connection between complex Julia sets and quaternion Julia sets, with the former being a special case and subset of the latter, and the latter being a natural extension of the former into a higher-dimensional space.

The advantatges of Quaternion Julia sets are the following:

1. **Rich Visual Content**: The 3D nature of Quaternion Julia Sets allows for visually impressive and complex 3D fractal images. They offer an unprecedented level of detail and beauty compared to their 2D counterparts.

2. **Real-time Rendering**: Due to the development of techniques such as Distance Estimation, it has become possible to render Quaternion Julia Sets in real-time. This is an advancement from earlier methods, which were more computationally expensive.

3. **Versatile**: The usage of quaternion algebra enables a variety of transformations and operations, which offers a wealth of opportunities for mathematical and computer graphics exploration.

However, we also have some disadvantatges:

1. **Complexity**: Quaternions are mathematically complex and can be hard to comprehend, especially for those without a strong background in algebra or complex numbers.

2. **Resource Intensive**: Rendering Quaternion Julia Sets, especially at high levels of detail, can be computationally intensive and time-consuming.

And we can see some applications of this technique in various fields:

1. **Computer Graphics**: They can be used to generate intricate 3D models and textures that would be difficult or impossible to design manually.

2. **Mathematical Visualization**: They provide a powerful tool for visualizing and exploring the properties of complex numbers and quaternions.

3. **Research**: Quaternion Julia Sets can be useful in various areas of research, including fractal theory, chaos theory, and dynamical systems.

VI. TECHNIQUE 4: L-SYSTEMS

1. What are L-systems?

L-systems, short for Lindenmayer systems, are mathematical formalisms that describe the growth and development of complex 3D structures using simple rules. In L-systems, fractals can be generated by iteratively applying production rules to a string of symbols, where each iteration creates a more detailed and intricate version of the initial shape.

To create fractals using L-systems, the production rules are typically defined in a way that allows for repeated self-replication or recursion. This means that the rules contain instructions to replace a symbol with a sequence of symbols that resemble a smaller version of the original shape.

Let's consider a simple L-system that has:

Variables: A, B

Axiom (the axiom is what you start with): A

Production Rules: exchange A for AB and exchange B for A

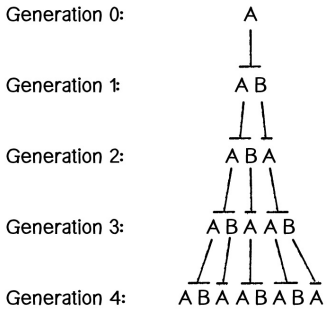


FIG. 19: Result over 4 iterations.

Refer to [23] for a more visual explanation and refer to [24] for more details.

Normally the rules include some "+" and "-" which are rotation instructions. The "+" instruction rotates the drawing direction by a certain angle in a counterclockwise direction, while the "-" instruction rotates it in a clockwise direction.

Algorithm 5: algorithm of previous L-system.

```
alphabet = 'A', 'B'
axiom = 'A'
rule1 = ['A', 'AB']
rule2 = ['B', 'A']
sentence = axiom
Function nextGeneration(sentence):
    workingSentence = ' '
    for c in sentence do
        if c == rule1[0] then
            workingSentence += rule1[1]
        else if c == rule2[0] then
            workingSentence += rule2[1]
        print(workingSentence)
        return workingSentence
```

Refer to [25] for a video explanation with the step by step development of this sequence

How can we translate this sentence into something related with 3D graphics?

Representing L-system sequences with 3D graphics involves extending the concept of drawing instructions to a three-dimensional space. Instead of interpreting the symbols as instructions for 2D movements and rotations, we can use them to define transformations in 3D space, such as translations, rotations, and scaling.

Here are some steps to represent L-system sequences with 3D graphics:

Step 1: Define a coordinate system: Set up a 3D coordinate system where you can position and orient your shapes. You'll need coordinates to represent the position and orientation of your drawing pen.

Step 2: Extend the production rules: Modify the production rules of your L-system to include 3D transformations. Instead of using just rotations, you can introduce translations along the X, Y, and Z axes.

Step 3: Interpret symbols as 3D transformations: Assign specific 3D transformations to each symbol in your L-system. For instance, you can define "F" as a line segment in 3D space and use the translation operation to move the pen forward along its current orientation. The "+" and "-" symbols can still represent rotations, but now in 3D space.

Step 4: Stack-based approach: Implement a stack data structure to handle the push and pop operations. When encountering the "[" symbol, push the current position and orientation of the pen onto the stack. When encountering the "]" symbol, pop the top position and orientation from the stack to restore the pen's state.

Step 5: Iterate and generate the 3D shape: Apply the L-system's production rules iteratively as before, generating a final sequence of symbols. Interpret the symbols using the 3D transformations you defined and draw the corresponding geometric elements, such as line segments or geometric primitives.

Step 6: Visualize the 3D shape: Render the resulting 3D shape using a suitable graphics library or framework. You can represent the line segments as connected vertices or construct complex meshes depending on the nature of the generated shape. Apply lighting, shading, and other visual effects to enhance the 3D visualization.

Visual example: That's how it looks to assign a big cube to A and a small cube to B:

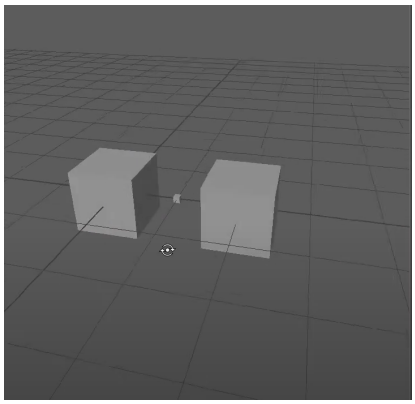


FIG. 20: ABA sentence with cubes

Refer to [25] for the video that contains the implementation of the sequence.

Refer to [26] for more details and code implementations.

2. Turtle Graphics in L-systems

Turtle graphics is a concept originated from the Logo programming language. It involves controlling a graphical cursor, known as a turtle, on a 2D plane. The turtle is a metaphorical cursor that moves on a 2D plane. The turtle can perform actions like moving forward or backward, turning left or right, lifting or lowering its pen, and changing its appearance. Turtle graphics operate on a Cartesian coordinate system, starting from a specific position. Instructions are given to the turtle, such as "move forward by 100 units" or "turn left by 90 degrees". As the turtle executes instructions, it moves and leaves a trail behind, enabling the creation of drawings and geometric shapes. It is useful because Turtle graphics provide immediate visual feedback, encouraging exploration and creativity.

Refer to [27] to see turtle movements with details

Let's just imagine:

Variables: F

Constants: +, -

Axiom: F

Production Rules: exchange F for F+F-F-F+F

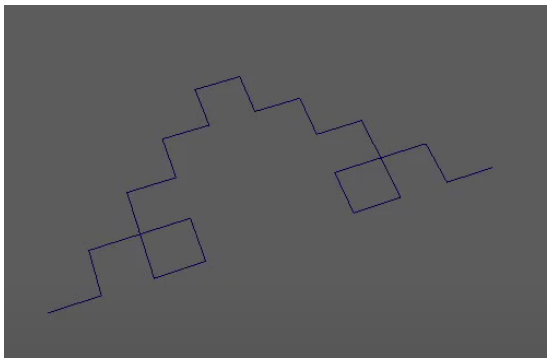


FIG. 21: Result over 2 iterations.

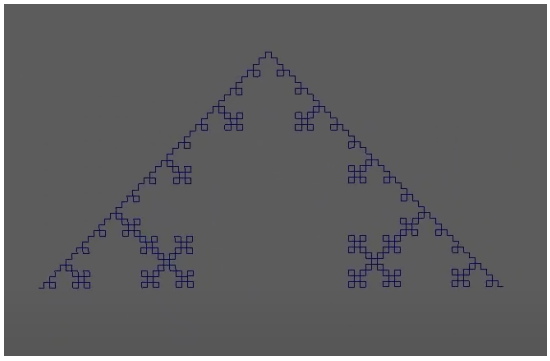


FIG. 22: Result over 4 iterations.

Refer to [28] for a video explanation with the step by step development of this sequence.

Advantages:

1. **Intuitive visualization:** Turtle graphics provide a straightforward and intuitive way to visualize L-systems, allowing for easy interpretation and understanding of the generated patterns.
2. **Dynamic and interactive:** Turtle graphics can be implemented in real-time, enabling dynamic and interactive exploration of L-systems and their visual output.

Disadvantages:

1. **Limited to 2D:** Turtle graphics are primarily focused on 2D visualizations and may require additional techniques to extend to 3D representations.
2. **Lack of fine-grained control:** Turtle graphics may not provide fine-grained control over certain graphical aspects, such as rendering effects or material properties.

3. Branching L-systems

It's a deterministic system that involves storing node locations at certain points in time and then calling those later on. By combining the turtle graphics approach with branching L-systems, we can generate visually appealing and realistic structures resembling trees or plants. The iterative application of production rules, combined with the turtle's movements and the use of stack-based branching, allows for the creation of intricate branching patterns similar to those found in nature.

Refer to [29] for a video introduction and explanation.

We need 2 new constants:

- '[' to save location
- ']' to recall to that location

This is an example of one of the simplest trees that we could recreate:

Variables: 0, 1

constants: [,]

Axiom: 0

Production Rules: exchange 1 for 11 and exchange 0 for 1[0]0

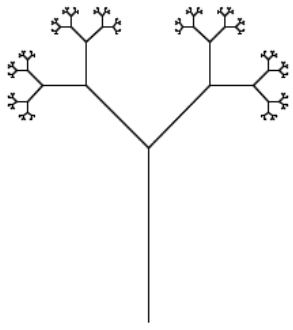


FIG. 23: Result over 7 iterations.

Refer to [24] for the development of this sequence.

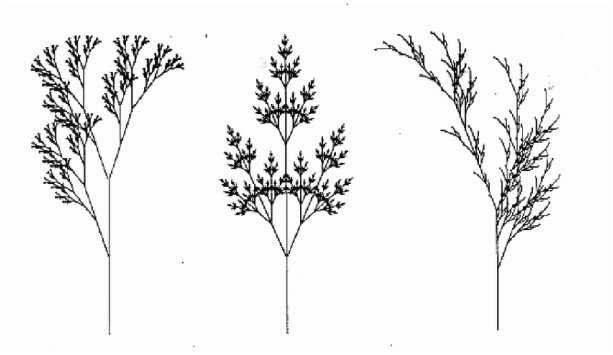


FIG. 24: Examples of plants that could be generated.

Refer to [30] for more examples.

After we have the structure defined, to convert it into a 3D graphic model we can follow the following steps:

Step 1: Convert to 3D coordinates: Map the 2D representation of the tree-like structure to 3D coordinates. You can assign a base position in 3D space and calculate the positions of each branch segment relative to its parent segment. Consider the length and angle information to determine the coordinates of each segment's endpoints.

Step 2: Define branching rules: If the tree has branching patterns, define rules to determine how branches split and grow. These rules can include parameters such as the angle of branch divergence and the length ratio between parent and child branches. By applying these rules, you can generate a branching structure in 3D space.

Step 3: Implement transformations: Use transformation operations to move, rotate, and scale the branches in 3D space. You can apply translation transformations to move each segment from its parent's position, rotation transformations to orient the segments correctly, and scaling transformations to adjust the thickness of the branches.

Step 4: Render the model: Once you have the 3D representation of the tree-like structure, you can render it using a suitable graphics library or framework. Represent the branches as geometric primitives, such as cylinders or line segments, and position them in 3D space based on the calculated coordinates. Apply appropriate materials, textures, and lighting to enhance the visual appearance of the model.

Step 5: Fine-tuning and optimization: Depending on your requirements, you may need to refine the model and optimize its performance. This could involve techniques such as LOD (Level of Detail) rendering, where you simplify the geometry of distant branches to improve rendering performance, or applying realistic materials and textures to enhance the visual realism.

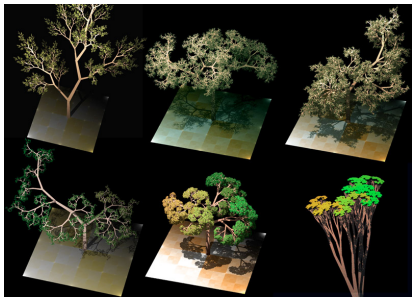


FIG. 25: Examples of trees that could be generated.

Refer to [24] for more details.

Advantages:

1. **Realistic tree-like structures:** Branching L-systems excel at generating tree-like structures, mimicking the branching patterns observed in nature.
2. **Hierarchical representation:** The hierarchical structure of branching L-systems allows for easy manipulation and modification of individual branches or segments.

Disadvantages:

1. **Increased complexity:** The introduction of branching in L-systems adds complexity to the rule set and interpretation, requiring careful management of branching angles, lengths, and other parameters.
2. **Limited to specific structures:** Branching L-systems are specialized for tree-like structures and may not be as suitable for other types of patterns or objects.

4. Stochastic L-systems

Stochastic L-systems play a vital role in 3D graphics by introducing randomness and variation into the generation process.

Stochastic L-systems allow for the introduction of random factors into the growth process of structures. In nature, many organic forms exhibit irregularities, asymmetries, and variations in their growth patterns. By incorporating randomness into the L-system rules, stochastic L-systems can generate models that closely resemble natural objects like trees, plants, and natural landscapes.

In Stochastic L-systems, each iteration of the L-system can produce different outcomes. This variation can be

seen in branching angles, branch lengths, and the overall shape of the generated objects. The ability to generate diverse structures adds visual interest and realism to 3D graphics.

Refer to [31] for a video introduction and explanation.

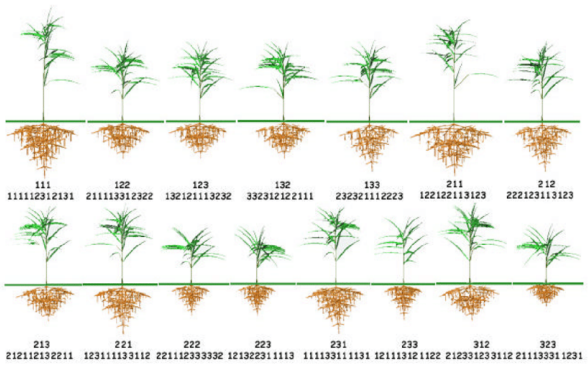


FIG. 26: Examples of plants that could be generated with a random factor of 0.33.

Refer to [32] for more details about stochastic systems on plants, and some implementations of plants structures.

Advantages:

1. **Natural and diverse patterns:** Stochastic L-systems introduce randomness and variation, resulting in more organic and diverse patterns that closely resemble natural forms.

2. **Realism and unpredictability:** The stochastic nature of these systems adds a sense of realism and unpredictability, making them suitable for simulating natural phenomena or creating unique designs.

Refer to [26] for a code implementation.

Disadvantages:

1. **Loss of determinism:** Stochastic L-systems are non-deterministic, making it challenging to precisely reproduce specific patterns or structures.

2. **Increased computational complexity:** The introduction of randomness in stochastic L-systems can lead to increased computational complexity, especially when dealing with large-scale iterations or complex rule sets.

5. Other L-systems applications

L-systems are not only limited to fractals and environment design, they can also have other uses on:

Computer Animation and Video Games: L-systems have been employed in computer animation to generate realistic and natural-looking movements. By representing characters or objects as L-systems, their motion can be defined based on the L-system rules. L-systems can simulate the motion of organic creatures or mimic natural physical phenomena, providing a tool for creating lifelike animations following a linear production pipeline, involving concept art, story boarding, 3D modeling, rigging, animation, and rendering. The incorporation of randomness, variation, and environmental interaction enables more realistic and dynamic NPC movement, enhancing the immersive experience in games and simulations.

Procedural Content Generation: L-systems are extensively used in procedural content generation, where content such as landscapes, buildings, and textures is generated algorithmically. L-systems provide a framework for generating complex and detailed structures procedurally, allowing for the efficient creation of large-scale and diverse content in video games, virtual environments, and simulations.

Refer to [33] for other applications of L-systems on building designs and animations (small introduction).

6. L-systems conclusion

In conclusion, L-systems offer a powerful framework for generating intricate patterns and structures, transcending the boundaries of traditional graphics. We explained the fundamental concepts of L-systems

Overall, L-systems provide a flexible and expressive approach to generating intricate patterns and structures. With their ability to capture the recursive nature of growth and the incorporation of branching, stochasticity, and other variations, L-systems have become a valuable tool in various fields, expanding the boundaries of artistic expression, scientific exploration, and computational design.

VII. JOINT ANALYSIS OF THE STRATEGIES AND METHODS STUDIED

In our project, each member delved into unique aspects of mathematical and computational visualization, with Alejandro focusing on Iterated Function Systems (IFS), Adria on procedural terrain generation using gradient fractal noise and chunks, Noah on Quaternion Julia Sets, and Oscar on L-Systems. Each topic contributed to our collective understanding of fractal generation, three-dimensional rendering, computational efficiency, and mathematical foundations. We will now discuss our collective findings in the context of the computational

strategies and methodologies applied.

Alejandro's work on IFS illuminated the intricate mathematical foundations of these systems and the application of the Collage Theorem. He presented a novel way of visualizing 3D IFS fractals using Raymarching, a process involving the iterative projection of a ray into a fractal until it intersects with the fractal's surface. His work highlighted the computational challenges inherent in the raymarching of IFS fractals, and the strategies applied to manage high computational load, achieve realistic visual effects, select optimal parameters, and manage abrupt color changes.

Adria's study centered around the use of gradient fractal noise and chunks for the generation and visualization of procedural terrains. Through the manipulation of height maps and the introduction of octaves for enhanced realism, he illustrated the value of these concepts in the generation of complex 3D structures. A notable element of Adria's work was the use of chunk generation for optimization, demonstrating how Level of Detail (LOD) algorithms and resource management can contribute to efficient, illusory world-building.

Noah's exploration of Quaternion Julia Sets deepened our understanding of quaternion algebra and the generation of 3D fractals. His study revealed an effective approach to visualizing Quaternion Julia Sets using distance estimation, which iteratively estimates the distance between a point in 3D space and the fractal to generate color for that point. His work also highlighted practical considerations for real-time rendering, including dimension reduction, the application of inverse iteration and ray tracing algorithms, and the selection of an appropriate lightning model.

Lastly, Oscar's research on L-Systems showcased the value of recursive rule sets in generating complex 3D structures. He utilized a technique called Turtle Graphics, interpreting the rules of the L-System as a series of turtle commands to generate the 3D structure. His work also delved into different types of L-systems, such as branching and stochastic L-systems, and discussed other applications of these systems.

Collectively, our research into these varied topics has expanded our understanding of the intricacies and potential of mathematical visualization and fractal generation. From handling computational load efficiently and realistically visualizing complex 3D structures, to understanding the mathematical foundations of these systems, our studies illustrate the richness and depth of these areas of research.

VIII. WORK PERFORMED BY EACH MEMBER OF THE TEAM

A. Adrià: Generating and Visualizing Procedural Terrains using Gradient Fractal Noise and Chunks

Adrià's work revolved around the use of fractal noise to generate and visualize complex 3D structures. He focused on a specific type of noise, known as fractal noise, characterized by self-similarity. For the visualization of this fractal noise, Adrià employed LOD techniques to dynamically generate terrain chunks.

B. Alejandro: Iterated Function Systems (IFS)

Alejandro focused on the exploration and visualization of IFS Fractals within a 3D space. His approach involved applying transformations to a point within this 3D space and iterating this process numerous times. To represent IFS fractals within 3D space, he utilized a technique known as ray marching, in which a ray is cast towards the fractal, and iteratively progressed until it reaches the fractal's surface.

C. Noah: Quaternion Julia Sets

Noah focused on Quaternion Julia Sets, a type of 3D fractal that can be generated using quaternion algebra. To represent Quaternion Julia Sets, he used a technique called Distance Estimation, which involves iteratively estimating the distance between a point in the 3D space and the fractal. This distance was then used to generate a color for the corresponding point.

D. Oscar: L-Systems

Oscar's project involved L-Systems fractals, which are generated using a set of recursive rules that can create complex 3D structures. For visualization of the L-Systems fractals, he used a technique known as Turtle Graphics. This method involves interpreting the rules of the L-System as a series of turtle commands to generate the 3D structure.

[1] Henrik Kniberg. Octave layering and perlin noise in minecraft., 2022. URL https://www.youtube.com/watch?v=CSa5O6knuwI&ab_channel=HenrikKniberg.

[2] Sebastian Lague. Lod techniques for chunk generation, 2015. URL https://www.youtube.com/watch?v=417kJGPKwDg&ab_channel=SebastianLague.

[3] Wikipedia. Marching cubes algorithm., 2023. URL https://en.wikipedia.org/wiki/Marching_cubes.

[4] Gressman. Ifs fractal archive, 2020. URL https://www2.math.upenn.edu/~gressman/ifs/.

[5] Wikipedia. Raymarching, 2023. URL https://en.wikipedia.org/wiki/Ray_marching.

[6] Dolovnyak Kolom. Raymarching sphere repetition, 2020. URL https://www.youtube.com/watch?v=vv_KDQdRFuM.

[7] Kevin Horowitz. Fractal rendering by ray marching with distance estimation, 2012. URL https://inst.eecs.berkeley.edu/~cs184/sp12/assignments/Archive/HW6/Fractal%20Renderer.htm.

[8] Adria Biagioli. Raymarching distance fields: Concepts and implementation in unity, 2016. URL https://adrianb.io/2016/10/01/raymarching.html.

[9] Darkeclipz. Ifs fractals rendering with raymarching, 2019. URL https://www.shadertoy.com/view/3djGDt.

[10] Wikipedia. Julia set, 2023. URL https://en.wikipedia.org/wiki/Julia_set#.

[11] H. Jurgens H.-O. Peitgen and D. Saupe. *Chaos and Fractals: New Frontiers of Science*. 1993. URL https://shorturl.at/mruzU.

[12] Benoit B. Mandelbrot. W. H. Freeman and co. *The fractal geometry of nature*. 1982. URL https://shorturl.at/ctJUW.

[13] Alan Norton. Generation and display of geometric fractals in 3-d. 16(3):61–67, 1982.

[14] William R. Hamilton. On quaternions, 1847. URL https://www.emis.de/classics/Hamilton/Quatern2.pdf.

[15] Christi Ho. 2015-2016 quaternion julia sets, 2015-2016. URL https://shorturl.at/nrIJY.

[16] D.J. Sandin J.C. Hart, L.H. Kauffman. Interactive visualization of quaternion julia sets. 1990. URL https://shorturl.at/ixBD8.

[17] L.H. Kauffman J.C. Hart, D.J. Sandin. Ray tracing deterministic 3-d fractals. 1989. URL https://dl.acm.org/doi/10.1145/74334.74363.

[18] D.J. Sandin Y.M. Dang, L.H. Kauffman. *Hypercomplex Iterations: Distance Estimation and Higher Dimensional Fractal*. 2002. URL https://shorturl.at/jkuOZ.

[19] S.K. Feiner J.F. Hughes J.D. Foley, A. van Dam. *Computer Graphics: Principles and Practice*. 1995. URL https://shorturl.at/BFRW6.

[20] A. Rosa. Methods and applications to display quaternion julia sets. In *Differential Equations and Control Processes*, 2005. URL https://shorturl.at/fikF5.

[21] Quaternion julia fractal images. https://github.com/zhehangd/QJulia. Last accessed: 2023-06-07.

[22] Yu. A. Kurochkin V. T. Stosui A. A. Bogush, A. Z. Gazizov. On symmetry properties of quaternionic analogs of julia sets. 2000. URL https://arxiv.org/abs/nlin/0105060.

[23] Geoffrey Datema. L-system fundamentals — 3d graphics overview, 2021. URL https://www.youtube.com/watch?v=egxBK_EGauM.

[24] Canderson7. L-system, 2005. URL https://en.wikipedia.org/wiki/L-system.

[25] Geoffrey Datema. Basic l-system implementation — 3d graphics overview, 2021. URL https://www.youtube.com/watch?v=uWT45Kz4bmw.

[26] FdelMazo. cl-aristid, 2020. URL https://github.com/FdelMazo/cl-aristid.

[27] Houdini 19.5. L-system geometry node.

[28] Geoffrey Datema. Turtle graphics in l-systems — 3d graphics overview, 2021. URL https://www.youtube.com/watch?v=LSSkaygNlYI.

[29] Geoffrey Datema. Branching l-systems — 3d graphics overview, 2021. URL https://www.youtube.com/watch?v=kQz3E__WZ-Q.

[30] Paul Bourke. Lindenmayer systems. 1991. URL http://paulbourke.net/fractals/lsys/.

[31] Geoffrey Datema. Stochastic l-systems — 3d graphics overview, 2021. URL https://www.youtube.com/watch?v=4NOg2nFFnwc.

[32] Hans Georg Bock Suchada Siripant Somporn Chuai-Aree, Willi Jäger. Smooth animation for plant growth using time embedded component and growth function. page 6, 2002. URL https://shorturl.at/wCELQ.

[33] Geoffrey Datema. Other l-system applications — 3d graphics overview, 2021. URL https://www.youtube.com/watch?v=BkE8t-mg3Tk.

[34] Slawomir Nikiel. Integration of iterated function systems and vector graphics for aesthetics, 2006. URL https://www.sciencedirect.com/science/article/abs/pii/S0097849306000033.

[35] Wikipedia. Collage theorem, 2023. URL https://en.wikipedia.org/wiki/Collage_theorem.

[36] Tomasz Martyn. Realistic rendering 3d ifs fractals in real-time with graphics accelerators, 2010. URL https://www.sciencedirect.com/science/article/abs/pii/S0097849309001150.