

Facultad de Ciencias



ETSII
Escuela Técnica Superior
de Ingenierías Informática
y de Telecomunicación



UNIVERSIDAD
DE GRANADA

DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y MATEMÁTICAS
TRABAJO DE FIN DE GRADO

Fractales

Fundamentos matemáticos y visualización con Ray-Tracing

Autor: Juan Antonio Villegas Recio

Tutor de Matemáticas: Manuel Ruiz Galán
Departamento de Matemática Aplicada

Tutor de Informática: Carlos Ureña Almagro
Departamento de Lenguajes y Sistemas
Informáticos

CURSO 2021–2022

GRANADA, A 15 DE JUNIO DE 2022

AGRADECIMIENTOS

A mis padres, Juan y Toñi, por hacer el gran esfuerzo que habéis hecho en estos cinco años para permitirme cumplir mi sueño de estudiar esta carrera lejos de casa. Todo lo que consiga en la vida y quién llegue a ser jamás habría sido posible sin vosotros. A mi hermano Marcos y a mi hermano Óscar, por aprovechar vuestra experiencia y los años de diferencia para intentar siempre llevarme por el cauce adecuado. A mi cuñada Pilar, mis sobrinos Marcos David y Elsa y a toda mi familia por siempre confiar en mí, por apoyarme y hacerme disfrutar de cada momento que pasamos juntos.

A mis amigas y amigos, por estar siempre a mi lado y acompañarme en el día a día de esta aventura. Gracias por animarme en los peores momentos, por compartir mis alegrías en los buenos y por soportarme cuando no podía aguantarme ni yo mismo. A mis compañeras y compañeros de clase, por los momentos compartidos y por el esfuerzo que hemos hecho para llegar hasta donde estamos. De todo corazón os deseo el mejor futuro.

A mis tutores, Carlos y Manuel, por su disponibilidad durante todo este tiempo, por su eficacia a la hora de hacer reuniones, correcciones, contestar correos... No puedo pedir una mejor atención en el desarrollo de mi TFG que la que vosotros me habéis dado.

Por último, y no por ello menos importante, quiero hacer una dedicatoria especial a mi ángel de la guarda y al de toda mi familia, que desde arriba estará viendo a lo que he llegado y debe estar lleno de orgullo. Tito Camilo, siempre me dejaste claro que creías en mí y en mi capacidad para lograr cualquier objetivo que me proponga. Hoy he conseguido acabar esta carrera que tan feliz te hizo saber que iba a comenzar, va por ti.

*La delgada línea
entre el arte
y las matemáticas*

RESUMEN

Es muy probable que a lo largo de su vida haya visto en internet, revistas o periódicos alguna imagen con patrones que se repiten hasta donde el ojo humano alcanza a ver, y con total seguridad ha tratado con estructuras tan naturales como un árbol, la hoja de un helecho y algunos tipos de verduras como la coliflor en las que la parte y el todo parecen tener una forma muy similar, fíjese por ejemplo en las imágenes 1.1 de esta memoria. Los fractales, desde un punto de vista matemático son figuras o imágenes que obedecen a patrones de repetición y similaridad, y son los protagonistas de este TFG.

No existe una clara definición de lo que es un fractal, pero muchas de las existentes y en especial la introducida por *Benoit Mandelbrot*, el considerado padre de la geometría fractal, tienen en común dos rasgos: la autosimilaridad y la dimensión. La autosimilaridad es una propiedad que tiene un conjunto que está compuesto por copias reducidas de sí mismo. Por ejemplo, las ramas de un árbol a su vez tienen forma de árboles más pequeños que el original. Sobre la dimensión, es más difícil abordarla de forma intuitiva, pero podemos afirmar que los fractales al tener formas muy complejas, aunque se puedan representar en dos o tres dimensiones realmente su dimensión puede incluso no ser un número entero, al contrario de lo que dictamina la intuición. Ayudados de ejemplos clásicos y como introducción a esta materia, ahondaremos en los distintos conceptos de autosimilaridad y dimensión fractal.

Una herramienta muy utilizada también en el mundo de los fractales es la iteración de procesos. De hecho, muchos de los fractales clásicos como el conjunto de Cantor (imagen 1.2) o el triángulo de Sierpinski (imagen 1.3) se originan gracias a procesos iterativos sobre figuras como son un intervalo cerrado de \mathbb{R} o un triángulo equilátero. Este es también el caso de los Sistemas de Funciones Iteradas, cuya aplicación reiterada nos ofrece bellos resultados. Además, si nos restringimos a las funciones complejas, la propia aplicación reiterada de una función compleja puede darnos figuras tan hermosas como enrevesadas. En este ámbito destacan los conocidos ‘conjuntos de Julia’ o el ‘conjunto de Mandelbrot’ (imágenes 3.1 y 3.2), que es para muchos el objeto más complejo de la matemática. Lo sorprendente es que estos conjuntos son el resultado de iterar una función compleja tan simple como $P_c(z) = z^2 + c$ para un $c \in \mathbb{C}$ fijo.

Se explicarán las bases y la teoría matemática asociada a todos estos ámbitos de la geometría fractal, todo ello acompañado de imágenes y ejemplos aclarativos.

Por otro lado, la geometría fractal es una disciplina muy visual. Las figuras, las imágenes y los gráficos son esenciales para entender la teoría y además son en parte el objetivo de este

ámbito: visualizar imágenes fractales a partir de algoritmos del análisis numérico. Por esto, se ha aprovechado la existencia de herramientas que aprovechan la Unidad de Procesamiento Gráfico (GPU) de los ordenadores, utilizadas en el ámbito de la Informática Gráfica para, en conjunción con los conocimientos que nos aportan las matemáticas, visualizar de manera interactiva algunos de los fractales 2D que se presentan.

Si damos un salto a las tres dimensiones, la situación es mucho más compleja, ya que es fácil identificar una imagen con un fragmento del plano, pero representar el espacio 3D en una imagen 2D es más difícil. Para ello hemos aplicado una herramienta tan conocida como costosa: El Ray-Tracing. Gracias a ella podremos visualizar tanto escenas sencillas con un aceptable grado de realismo como escenas con fractales tridimensionales originados por la generalización a 3D de los conjuntos de Julia y de Mandelbrot (imágenes 9.13 y 9.15).

Toda esta tarea de visualizado se ha desarrollado de forma interactiva, creando como producto software una web fácilmente accesible para cualquier usuario que disponga de un ordenador y un navegador.

Palabras clave

Fractales, autosimilaridad, dimensión fractal, iteración, conjuntos de Julia, conjunto de Mandelbrot, Sistemas de Funciones Iteradas, Ray-Tracing, Sphere Tracing, Mandelbulb.

ABSTRACT

Probably you had ever seen on the Internet, magazines or even journals any picture with some patterns that repeat over and over again. And pretty surely you had seen or touched such natural structures as trees, ferns or some kind of vegetables, for example a cauliflower, where the whole object seems to be composed of small copies of itself, look at images 1.1. Fractals, from a mathematical point of view, are objects, shapes or pictures that have repeating and similar patterns. The fractals are, in fact, the focus of this text.

There are lots of types of fractals and ways to generate them. However, many of them are based on the concept of ‘iteration’. In this document we explain carefully the theory while we show some *Mathematica* codes we had used and the results we got. In case you did not know, *Mathematica* is a widely used mathematical software all over the world.

We also created an interactive website where you can visualize 2D and 3D fractals. All the methods and tools we had used is explained and documented along this text.

Now, we will explain in detail the chapters of this document and their contents.

Chapter 1: The concept of ‘fractal’

In this first chapter we give a first insight about what is a fractal. There is not a unique definition of fractal, but in most cases, and specially the one introduced by *Benoit Mandelbrot*, the fractal geometry’s father, this term is related to the ideas of self similarity and dimension.

Firstly, we compare fractals with known objects like trees, romanescu or rays. With this real life examples, we introduce the concept of ‘self similarity’ and present some of the most popular and classic fractals: the Cantor set, the Sierpinski triangle, the Sierpinski Carpet, the Menger Sponge, the Koch curve and the Koch snowflake (respectively, images 1.2, 1.3, 1.4, 1.5, 1.6 and 1.7). We explain how each one is generated and some of their properties, including of course pictures.

All these classic examples help us to understand what self similarity is, so it is time to learn about dimension. There are several types, but the most common fractals dimensions are the ‘box-counting dimension’ and the ‘Hausdorff dimension’. We talk about both and about the ‘topological dimension’, and after it we expose some relations between the different definitions given.

Chapter 2: Iteration

The iteration is the basis of many algorithms in many fields of science. Particularly, given a complex function $f : \mathbb{C} \rightarrow \mathbb{C}$ and $z_0 \in \mathbb{C}$, you can calculate $f(z_0)$, and $f(f(z_0))$, $f(f(f(z_0)))$, and so on. That is the basic idea in order to generate the first fractals images in this chapter.

Remember we talked about Mathematica, and now is when we start to use it. It includes support for many fields of mathematics, in particular 2D graphics, complex numbers and iteration of functions, and these will be the most useful tools for us.

Using the Newton method to find roots of complex functions and evaluating the convergence speed of the method on each $z \in \mathbb{C}$, we can render using Mathematica the first pictures with fractal appearance, look at images 2.3 and 2.4.

Chapter 3: Julia and Mandelbrot Sets

This would not be a fractals project if we do not talk about Julia and Mandelbrot sets. We present the complex function $P_c(z) = z^2 + c$ for a fixed $c \in \mathbb{C}$ and its importance in this chapter. From this function we explain the algorithms used to visualize the Julia set of a complex number c (\mathcal{J}_c) and the Mandelbrot set (\mathcal{M}). Look at images 3.1 and 3.2.

We also show some self-similar regions they present, and finally we explain and visualize some generalizations of these sets. Everything is accompanied by lots of clarifying images.

Chapter 4: Iterated Function Systems

A function can be iterated over sets, not only over one complex number or point. In this chapter, we move to the \mathbb{R}^2 euclidean space and present the affine maps. Thanks to sets of contractive affine maps we can iterate a subset of the plane and get beautiful results. In a more general context, we define what a Iterated Function System (IFS) is and why its repeating application to any subset always converges to a set we call ‘the attractor of the IFS’.

Thanks to IFS we can also calculate so easily the box counting dimension of self-similar sets. We explain the relation between these two concepts and give some examples of this calculation.

In addition, although you can get images from SFIs, the inverse problem is also interesting. *I.e.*, given an image, is it possible to find an IFS whose attractor is the given image? We will see how the famous ‘collage theorem’ answers this question.

Chapter 5: Introduction to the render tools

At this point, we could render many fractals images using Mathematica. The problem is that Mathematica is not a render tool, it computes each pixel color sequentially, so the execution is very slow. We will give a little introduction to what we call ‘rasterization APIs’, tools used to render 2D and 3D images. The main advantage of rasterization APIs is they use the computer’s Graphics Processing Unit (GPU) besides the central processor, so they are much faster. Hereby, we need to specify some fragments of code written in a special language, these are called ‘shaders’.

Specifically, we will use WebGL, an API which was designed to be used in JavaScript programs, so we can develop a web application and execute it using a web browser. WebGL

needs the shader to be written using the ‘GL Shading Language’ (GLSL). In this chapter we expose the main components of a WebGL based application and code step by step a very simple example.

Chapter 6: Software planning and analysis

Once we know the mathematical basis of fractals and we have introduced render tools, it is time to describe the software product that will be developed in the following chapters.

First, we specify the temporal planning by identifying the main development phases and assigning them some particular weeks. After it, we approximate the project’s budget, which is divided in hardware, software and human resources.

Next, we do a requirements analysis in order to know what and how we want this software to do. Once the requirements are specified, we do a use case analysis, showing some clarifier diagrams. Finally, we present the future web wireframes.

Chapter 7: Rendering 2D fractals

After initiate us in WebGL and present the project specification, it is time to render 2D fractals. In this chapter, we frequently reference the methods and theorems we defined in the third chapter, because the goal is develop an interactive web page where it is possible to render Julia and Mandelbrot sets modifying some parameters.

In order to write readable and reusable code, and to easy the debug process, we use the Object Oriented paradigm and structure the code into JavaScript classes. After it, we use GLSL to translate the Mathematica code we wrote in chapter 3 with the aim of render the Julia and Mandelbrot set using WebGL. Last, but not least, we introduce a edge smoothing technic called ‘Supersampling Antialiasing’, which we’ll use in order to care about the realism and images’ quality.

Chapter 8: Introduction to Ray-Tracing

It is easy to identify the central point of a pixel in the screen with a point in the complex plane \mathbb{C} . This makes the 2D process so easy. However, render a 3D scene with fractals is not that simple. There are different ways of representing a 3D scene in a 2D screen. We will use the ‘Ray-Tracing’ technique, which consists of computing, for each pixel in the image, which object, if any, is projected onto that pixel, and then computing the pixel color accordingly.

In this chapter, we give a detailed definition of this method and its components. The goal now is develop a simple, but complete, ray-tracer. It means, render a scene with simple objects, like spheres and a plane. Now we are going to introduce the roadmap we followed.

First, we describe the creation of the rays, after all there is not ray-tracing without rays. After that, we assign a color to each pixel asuming the scene is empty. It means, we define which colors will we assign to the pixels whose rays miss. Next, we add some spheres and a plane to the scene, programming the intersection sphere-ray and plane-ray, which fortunately are so easy. Once we have filled the scene with objects, it’s time to move the camera. We modify the ray creation code so as to let us change freely the point of view, allowing us to see different perspectives of

the same scene. Later, we introduce in detail and implement a lighting model in order to give volume, shine, shadows and realism to the scene, assigning a material to each object, creating light sources and evaluating this lighting model at each intersection point. Specifically, we will use the ‘Phong lighting model’, which is not very complex, but enough for our purpose. We also generalize SSAA with the aim of apply this technic on 3D scenes, achieving much more realism and quality.

Chapter 9: Rendering 3D fractals

This is the last chapter, the icing on the cake. Starting from the code we wrote in the previous chapter, we have to modify it and include some new code in order to visualize 3D fractals. The first fact: calculate the intersection is not that simple as it is when you render Spheres and Planes. There is not an easy way to solve an equation like before. Nevertheless, for each object, we can try to find a signed distance function (SDF). This is a scalar field which yields, for each point of the space, a lower bound of the distance to the object’s surface. So, you can identify a surface with the set of points where the SDF is equal to 0.

We can render scenes with objects defined by their SDFs, by using the so called ‘sphere-tracing’ variant of ray-tracing. It consist of marching iteratively along the ray until a intersection is found. Starting from the observer’s position, at every step we calculate the minimum distance to any object of the scene. Unless this distance is so small (in that case we consider we reached an intersection), we advance that minimun distance along the ray, repeating the calculation until the ray misses or an intersection is found.

With these two new ingredients, we can modify the ray-tracer in order to calculate the intersections using sphere-tracing instead of resolving equations. In the following sections, we describe adequate SDFs for the purpose of render Julia and Mandelbrot 3D generalizations (images 9.13 and 9.14), which are based on quaternions. We also render the Mandelbulb set, a non rigorous but beautiful generalization of the Mandelbrot set (image 9.15).

Keywords

Fractals, self similarity, fractal dimension, iteration, Julia Sets, Mandelbrot Sets, Iterated Function Systems, Ray-Tracing, Sphere Tracing, Mandelbulb.

ÍNDICE GENERAL

Lista de Abreviaturas	15
Lista de Imágenes	19
Introducción	21
Objetivos	27
1. El concepto de <i>fractal</i>	29
1.1. Ejemplos clásicos	30
1.1.1. El conjunto de Cantor	30
1.1.2. El triángulo de Sierpinski	31
1.1.3. La alfombra de Sierpinski y la esponja de Menger	32
1.1.4. La curva de Koch	33
1.1.5. El copo de nieve de Koch	34
1.2. Conceptos de dimensión fractal	35
1.2.1. Dimensión por cajas	36
1.2.2. Medida y dimensión de Hausdorff	38
1.2.3. Dimensión topológica	39
1.2.4. Relación entre los distintos tipos de dimensión fractal	40
2. Iteración	43
2.1. Iteración de funciones	43
2.1.1. Convergencia a un punto fijo	44
2.1.2. Velocidad de convergencia	45
2.2. El método de Newton y cuencas de atracción	46
2.2.1. Autosimilaridad	49
3. Conjuntos de Julia y Mandelbrot	51
3.1. Iteración convergente y no convergente	52
3.2. Conjuntos de Julia	53
3.2.1. Representación gráfica de los conjuntos de Julia	53

3.3.	Distinción entre conjuntos de Julia conexos y polvaredas	55
3.4.	El conjunto de Mandelbrot	56
3.4.1.	Representación gráfica del conjunto de Mandelbrot	57
3.5.	Autosimilaridad de los conjuntos de Julia y Mandelbrot	59
3.5.1.	Autosimilaridad en conjuntos de Julia	59
3.5.2.	Autosimilaridad en el conjunto de Mandelbrot	61
3.6.	Conjuntos de Julia y Mandelbrot generalizados	61
3.6.1.	Familia $\{z^m + c\}_{c \in \mathbb{C}}$	62
3.6.2.	Conjuntos de Julia con funciones no polinómicas	64
4.	Sistemas de Funciones Iteradas	67
4.1.	Transformaciones afines en el plano euclídeo y SFI	67
4.2.	Convergencia de SFI	70
4.2.1.	El Espacio de Fractales y la Métrica de Hausdorff	71
4.2.2.	Aplicaciones contractivas en el espacio de fractales	72
4.2.3.	El espacio de fractales y los SFI	73
4.3.	SFI y conjuntos autosimilares	75
4.4.	El problema inverso	77
5.	Introducción a las herramientas de visualización	81
5.1.	Síntesis de imágenes en GPUs. WebGL	81
5.2.	Componentes de WebGL	83
5.2.1.	Contexto de WebGL	83
5.2.2.	El programa <i>Shader</i>	83
5.2.3.	Los <i>Buffer</i>	85
5.3.	Primera imagen generada	86
6.	Planificación y Análisis del Software	89
6.1.	Planificación	89
6.1.1.	Infraestructura básica	89
6.1.2.	Visualización de fractales 2D	90
6.1.3.	Construcción de un programa ray-tracer	90
6.1.4.	Visualización de fractales en 3D	90
6.1.5.	Realismo y optimización en la escena 3D	91
6.1.6.	Añadir estilo y UX a la web	91
6.2.	Presupuesto del producto software	92
6.2.1.	Recursos hardware	92
6.2.2.	Recursos Software	92
6.2.3.	Recursos humanos	93
6.2.4.	Presupuesto final	93
6.3.	Ánálisis de requisitos	93
6.3.1.	Requisitos funcionales	93
6.3.2.	Requisitos no funcionales	94
6.3.3.	Requisitos de datos	95
6.4.	Ánálisis de casos de uso	95

6.5.	Bocetos de la web	96
7.	Visualización de fractales en 2D	99
7.1.	Estructurando el código	100
7.2.	Identificando la pantalla con el plano	101
7.3.	La función $P_{c,m}(z) = z^m + c$	103
7.4.	Asignación de colores	104
7.5.	Visualizando conjuntos de Julia	106
7.6.	Visualizando conjuntos de Mandelbrot	107
7.7.	Alternando conjuntos de Julia y Mandelbrot	109
7.8.	Supersampling Antialiasing	109
8.	Introducción al <i>Ray-Tracing</i>	115
8.1.	Definición de Ray-Tracing	115
8.2.	Elementos y estructura de un Ray-Tracer	117
8.3.	Creación de rayos primarios	119
8.4.	El background	123
8.5.	Visualizando una escena sencilla	124
8.5.1.	Visualizando una esfera	124
8.5.2.	Visualizando varias esferas	127
8.5.3.	Visualizando un plano con textura de ajedrez	129
8.6.	Configurando la cámara	132
8.7.	Modelo de iluminación de Phong	137
8.7.1.	Componente ambiental	138
8.7.2.	Componente difusa	139
8.7.3.	Componente especular	140
8.7.4.	Sombras arrojadas	140
8.7.5.	Evaluación del modelo de iluminación	142
8.7.6.	Implementación del modelo de Phong	142
8.8.	SSAA en Ray-Tracing	148
9.	Visualización de fractales en 3D	151
9.1.	El algoritmo Sphere-Tracing	151
9.1.1.	Signed Distance Functions (SDFs)	153
9.1.2.	Implementación de Sphere-Tracing en pseudocódigo	155
9.1.3.	Implementación de Shere-Tracing en GLSL	155
9.1.4.	Comentarios sobre Sphere-Tracing	161
9.2.	Visualización tridimensional de conjuntos de Julia	163
9.2.1.	Aproximando la normal	166
	Método 1: Gradiente de la SDF	166
	Método 2: Técnica del tetraedro	167
9.2.2.	Implementación en GLSL	167
9.3.	Visualización tridimensional del conjunto de Mandelbrot	171
9.4.	El conjunto de Mandelbulb	173
9.5.	Comparación con los fractales 2D	176

9.5.1.	Comparación entre conjuntos de Julia 2D y 3D	177
9.5.2.	Comparación entre el conjunto de Mandelbrot 2D y 3D	178
9.5.3.	Comparación entre el conjunto de Mandelbub y \mathcal{M}_8	179
9.6.	Posibles optimizaciones	180
9.6.1.	Esferas englobantes	180
9.6.2.	Optimización de sombras con esferas englobantes	181
Conclusiones		183
Bibliografía		184
Apéndices		191
A. Documentación del código JavaScript		191
A.1.	Componentes de WebGL comunes a 2D y 3D	191
A.1.1.	Fichero <code>shader.js</code>	191
Enumerado <code>ShaderType</code>	191	
Clase <code>Shader</code>	192	
Clase <code>ShaderProgram</code>	192	
A.1.2.	Fichero <code>buffer.js</code>	193
Clase <code>Buffer</code>	193	
A.1.3.	Fichero <code>scene.js</code>	193
Clase <code>Scene</code>	194	
A.2.	Visualización de fractales 2D	195
A.2.1.	Fichero <code>scene2D.js</code>	195
Clase <code>Scene2D</code>	195	
A.2.2.	Fichero <code>fractals-2D.js</code>	199
A.3.	Ray-Tracing y fractales 3D	199
A.3.1.	Fichero <code>scene3D.js</code>	200
Clase <code>Scene3D</code>	200	
A.3.2.	Fichero <code>fractals-3D.js</code>	204
A.4.	Diagrama de clases UML	205
B. Instalación del producto software		207

LISTA DE ABREVIATURAS

- ALU: Arithmetic-Logic Unit (Unidad Aritmético-Lógica)
- API: Application Programming Interface (Interfaz de Programación de Aplicaciones)
- CPU: Central Processing Unit (Unidad Central de Procesamiento)
- DC: Device Coordinates (Coordenadas de Dispositivo)
- GPU: Graphics Processing Unit (Unidad de Procesamiento Gráfico)
- MIL: Modelo de Iluminación Local
- SDF: Signed Distance Function
- SFI: Sistema de Funciones Iteradas
- RT: Ray-Tracing
- WC: World Coordinates (Coordenadas de Mundo)

LISTA DE IMÁGENES

1.1.	Objetos de la naturaleza con estructura fractal	29
1.2.	Iteraciones del proceso de generación del conjunto de Cantor	31
1.3.	Generación del triángulo de Sierpinski	32
1.4.	Generación de la alfombra de Sierpinski	33
1.5.	Esponja de Menger	33
1.6.	Iteraciones del proceso de generación de la curva de Koch	34
1.7.	Generación del copo de nieve de Koch	34
1.8.	Curvas de Koch que componen el copo de Koch	34
1.9.	Possible movimiento de un punto en posibles objetos de \mathbb{R}^n	35
1.10.	Segmento, cuadrado y cubo recubiertos por objetos de lado $\frac{1}{2}$	36
1.11.	Una forma de calcular la dimensión por cajas de la curva de Koch	37
1.12.	Figuras representativas de los ejemplos	40
2.1.	Representación de dos órbitas en \mathbb{C}	44
2.2.	Cuencas de atracción de $f(z) = z^2 - 1$	48
2.3.	Cuencas de atracción de distintas funciones coloreadas.	48
2.4.	Evaluación de la velocidad de convergencia en cada punto	49
2.5.	Cuencas de atracción de $f(z) = z^3 - 1$	49
2.6.	Diferentes regiones ampliadas de la figura 2.5	50
3.1.	Primeras imágenes de conjuntos de Julia	51
3.2.	Primeras imágenes del conjunto de Mandelbrot	52
3.3.	Conjuntos de Julia graficados con Mathematica	55
3.4.	Resultados de la orden ‘JuliaSetPlot’	56
3.5.	Representación de \mathcal{M} y detalle en $[-0.65, -0.4] \times [0.47, 0.72]$	59
3.6.	Diferentes regiones ampliadas de la figura 3.3 (b)	60
3.7.	Detalles autosimilares de algunos conjuntos de Julia	60
3.8.	Detalles autosimilares de \mathcal{M}	61
3.9.	Ampliaciones del bulbo principal de \mathcal{M}	62
3.10.	Conjuntos de Julia con $P_{-0.55+0.48i,m}$ para distintos valores de m	63
3.11.	Conjuntos de Mandelbrot \mathcal{M}_m para $m = 3, 4, 8, 10$	64

4.1. Ejemplos de aplicaciones de transformaciones lineales	69
4.2. Representación gráfica de T y $w(T)$	70
4.3. Otra posible semilla para iterar el SFI	70
4.4. Resultado de iterar 8 veces w con distintas figuras iniciales	70
4.5. Contraejemplo a la distancia entre conjuntos	71
4.6. Atractores de los SFI definidos en las tablas 4.1 y 4.2.	75
4.7. Imagen cuyo SFI debemos determinar	79
5.1. <i>Clip space de WebGL</i>	84
5.2. Cuadrado de colores graficado con WebGL	87
5.3. Componentes de WebGL e interacciones entre ellos	88
6.1. Diagrama de Gantt de la planificación del desarrollo del producto software	92
6.2. Diagrama de Casos de Uso	96
6.3. Diagrama de Actividad	96
6.4. Wireframe de la portada de la página	97
6.5. Wireframe de la pantalla en la que se visualizan fractales	97
7.1. Síntesis de una imagen vía una función	99
7.2. Paleta de colores elegida para la visualización de fractales 2D	105
7.3. Gradiente generado por la paleta de colores 7.2	105
7.4. Generación de algunos conjuntos de Julia con WebGL	107
7.5. Representación de algunos conjuntos de Mandelbrot con WebGL	109
7.6. Detalle de $\mathcal{J}_{-0.53+0.53i}$ antes de aplicar antialiasing	110
7.7. Representación de los rayos que se lanzan a una pantalla	110
7.8. Construcción de los puntos por píxel en SSAA	111
7.9. Detalle de $\mathcal{J}_{-0.53+0.53i}$ tras aplicar SSAA	113
8.1. Esquema básico del funcionamiento del Ray-Tracing	116
8.2. Elementos que participan en RT	121
8.3. Gradiente utilizado para el fondo de la escena	123
8.4. Primera escena vacía visualizada	124
8.5. Escena con una esfera	127
8.6. Escena con varias esferas	129
8.7. Textura de ajedrez a partir de las partes enteras	131
8.8. Escena compuesta por esferas y un plano	132
8.9. Field Of View	133
8.10. Vectores que forman una base del plano de proyección	134
8.11. Representación gráfica de los campos de ‘Camera’	135
8.12. Escena 8.8 desde otros puntos de vista	137
8.13. La calima: un ejemplo del efecto de la componente ambiental	139
8.14. Esquema de los vectores utilizados en la componente difusa	139
8.15. Esquema de los vectores utilizados en la componente especular	140
8.16. Punto sobre el que no incide la luz	141
8.17. Ejemplo de objeto no convexo y punto a la sombra	141

8.18. Ejemplos de puntos sobre los que incide y no una fuente de luz	142
8.19. Componentes aisladas del modelo de Phong	143
8.20. Acción de todas las componentes	143
8.21. Escena aplicando el modelo de Phong completo	147
8.22. Suelo antes de aplicar antialiasing	148
8.23. Suelo tras aplicar antialiasing	150
8.24. Imagen final del capítulo 8	150
 9.1. Situación a la que aplicar Sphere-Tracing	152
9.2. Dos primeras iteraciones de Sphere-Tracing	153
9.3. Posibles estados finales del algoritmo Sphere-Tracing	153
9.4. Ejemplos de puntos con distintos valores de la SDF de una esfera	154
9.5. Cálculo de la distancia de un punto a un plano	155
9.6. Escena tras implementar Sphere Tracing	159
9.7. Escena utilizando vectores no unitarios en los rayos	160
9.8. Resultado final de la modificación	161
9.9. Sombras con el parámetro $k = 2, 32$	161
9.10. Escena generada utilizando $\varepsilon = 0.1$	163
9.11. Escena generada utilizando MAX_STEPS=100	163
9.12. Representación de una montaña con curvas de nivel	165
9.13. Conjuntos de Julia 3D para distintos $c \in \mathbb{H}$	171
9.14. Detalles del conjunto de Mandelbrot generalizado	173
9.15. Conjunto de Mandelbub	176
9.16. Conjunto \mathcal{J}_{-1}	177
9.17. \mathcal{J}_{-1} visto desde abajo	177
9.18. Generalizaciones de $\mathcal{J}_{-0.71-0.31i} \subseteq \mathbb{C}$	178
9.19. Conjunto $\mathcal{J}_{-0.71-0.31i}$	178
9.20. Conjunto de Mandelbrot	179
9.21. Conjunto de Mandelbub y de Mandelbrot de orden 8	179
 A.1. Diagrama de clases JavaScript utilizadas	205

INTRODUCCIÓN

Desde la antiguedad, el ser humano siempre ha seguido procesos iterativos. Aunque normalmente la iteración en el sentido moderno se asocia a algoritmos, bucles o recursión, en realidad es un concepto ancestral. Las recetas de cocina, la recolección del cultivo, construcción de edificios, todas ellas son procesos que se basan en repetir operaciones sucesivamente una y otra vez. La iteración, y el tan ambiguo concepto del infinito matemático que ha atormentado las mentes de los pensadores desde los tiempos de Zenón hasta hace 120 años gracias a los trabajos del célebre matemático *George Cantor* son los precursores de toda la teoría fractal de la que disponemos a día de hoy. La primera aproximación a lo que hoy llamamos fractales llegó con el descubrimiento de funciones que a pesar de ser continuas no admiten derivada, esto en los trabajos de *Bernhard Riemann* y *Karl Weierstrass* en la década de 1870, dando a luz a lo que en aquellos tiempos se conocían como “monstruos matemáticos”.

Unos 50 años después, allá por 1917, *Gaston Julia* y *Pierre Fatou*, con el objetivo del *Grand Prix des Sciences mathématiques* que anunció la Academia Francesa de Ciencias publicaron independientemente trabajos sobre iteración de funciones racionales complejas. En 1918 Julia publicó un tratado extenso sobre el tema, al cual se le unió el año siguiente una trilogía de trabajos por parte de Fatou. Por su parte, el conocido matemático alemán *Felix Hausdorff* publica en este mismo año un artículo sobre la dimensión posiblemente no entera de conjuntos [16].

Fueron estos avances los que sirvieron de inspiración a *Benoit Mandelbrot*, quien a partir de 1975 con sus trabajos, y en especial su libro *The fractal geometry of nature* [21] creó lo que hoy conocemos como ‘Geometría Fractal’. A partir de este momento los científicos con ayuda de los primeros computadores consiguieron visualizar lo que los pioneros ya intuyeron.

Desde entonces, científicos como *Michael Barnsley* o *Kenneth Falconer* desarrollan teorías como los sistemas de funciones iteradas, los cuales con ayuda del ordenador suponen una revolución en la geometría fractal, con aportes como el *teorema del collage*, que tiene aplicaciones en ámbitos de la matemática más allá de los fractales.

Más tarde apareció la Informática Gráfica (la rama de la Informática que trata, entre otras cosas, sobre la generación de imágenes por ordenador) y posteriormente se popularizó el uso de las Unidades de Procesamiento Gráfico (GPUs, por Graphics Processing Units): un componente hardware caracterizado por su alta capacidad de cálculo paralelo en coma flotante, que se encuentra disponible en todos los ordenadores y dispositivos móviles en la actualidad. El

método más primitivo de visualizar escenas tridimensionales en pantallas, que son objetos de dos dimensiones, es el conocido como ‘rasterización’. Este método consiste en considerar cada objeto de la escena y calcular y colorear los píxeles donde se proyecta.

No fue hasta la década de 1980 cuando apareció el concepto de ‘emisión de rayos’ en la informática gráfica. Esta sería la semilla de la costosa, pero productiva técnica de generación de imágenes que conocemos como ‘Ray-Tracing’, en la cual, al contrario de la rasterización, consideramos cada píxel de la pantalla e identificamos los objetos que se proyectan en dicho pixel. Gracias a esta técnica, la visualización por ordenador de fractales incluso en tres dimensiones llega a su auge, con trabajos como los de *John Hart*, *Keenan Crane* o *Íñigo Quilez* en las décadas de 1990 y 2000.

A día de hoy, con la revolución tecnológica en la que nos vemos sumidos en la actualidad, toda la teoría fractal está en explotación, tanto en el ámbito informático como en el puramente matemático.

Existen revistas, catalogadas como ‘JCR’ (Journal Citation Reports), que tienen un alto prestigio entre las revistas dedicadas al mundo de la investigación. Por ejemplo, en el ámbito meramente fractal, la revista *Fractals*. Por su parte, en el ámbito de problemas inversos encontramos la revista *Inverse Problems*.

La geometría fractal y el teorema del collage también han inspirado el desarrollo de herramientas de trabajo en otras áreas de las matemáticas. Ejemplo de ello es el artículo titulado *Solving Parameter Identification Problems using the Collage Distance and Entropy* [20], con fecha de 2021. En la revista ‘Fractals’ podemos también encontrar un artículo de 2019 llamado *Self-similarity of solutions to integral and differential equations with respect to a fractal measure* [19], que relaciona geometría fractal y ecuaciones diferenciales.

Por su parte, la revolución informática afecta en particular a las GPUs. En 2020 NVIDIA lanzó al mercado la serie ‘GeForce RTX 30’. Por ejemplo, la NVIDIA GeForce RTX 3050, que es de los últimos modelos, ofrece núcleos de Ray Tracing dedicados y emplea tecnologías de inteligencia artificial.

Sobre esto, la última tendencia en informática gráfica, y particularmente en ray-tracing, es aplicar ‘Deep Learning’ en la síntesis de imágenes por ordenador. En contraste con la tradicional forma de programar potentes algoritmos para graficar imágenes lo más realistas posibles, el uso de redes neuronales en muchas situaciones ofrece resultados incluso mejores. Es por esto que este es un campo abierto a día de hoy en informática gráfica. Una explicación más detallada a la par que dinámica se puede encontrar en [56].

Vemos que el ray tracing está muy a la orden del día, pero sólo hace unos 3 años que existen herramientas propiamente dedicadas al ray-tracing, entre las que destacan *Vulkan con su extensión KHR*. El principal defecto que tienen es que solo pueden ser ejecutadas en GPUs muy caras y además sólo trabajan con mallas de polígonos. La alternativa a ellas es simular el ray-tracing en herramientas de visualización estándar.

Debemos también mencionar el evento *Siggraph*, que es un evento anual en el que se exponen nuevas técnicas y avances en el mundo de la informática gráfica. En su web podemos encontrar muchos papers y artículos relacionados con la generación de imágenes. Por ejemplo en la siguiente encontramos muchos textos con fecha de 2021: <https://kesen.realtimerendering.com/sig2021.html>.

Como hemos podido comprobar, la teoría de la geometría fractal es una disciplina muy

estudiada e investigada incluso en la actualidad. No obstante, es muy desconocida en general, tanto para estudiantes como para profesionales de las matemáticas y la informática, a pesar de tener una muy estrecha relación con ambas ciencias. Por ello, queremos aportar a esta disciplina nuestro granito de arena para que se dé a conocer a todo el público que lo desee.

Para ello, en este documento se presenta desde el punto de vista matemático los conceptos y resultados principales de la geometría fractal, utilizando no sólo desarrollos teóricos y texto, sino combinándolos también con ejemplos, imágenes y códigos para graficarlas. De esta forma se ofrece un contenido fácil de seguir y atractivo para el lector.

Aprovechando también el hardware y software moderno junto con conocimientos en informática gráfica, se crea como producto software una web interactiva en la que visualizar fractales, tanto 2D como 3D. Esta web ofrece la posibilidad de modificar dinámicamente parámetros tanto de los propios fractales como de los algoritmos utilizados para graficarlos. Se permite también movimiento por la escena, pudiendo observar detalles de los fractales incluso a escalas pequeñas. Con esta web se desarrolla un producto que es accesible para cualquier persona que tenga acceso a internet, acercándola al mundo de los fractales. La URL de la misma es <https://jantoniov.github.io/Geometria-Fractal/>.

Esta visualización se ha implementado utilizando técnicas avanzadas en informática gráfica, las cuales están minuciosamente introducidas y explicadas en esta memoria. Así además introducimos al lector varios conceptos y métodos de la informática gráfica, particularizando su uso a la síntesis de imágenes fractales.

En conclusión, con el objetivo de familiarizar al lector con la disciplina fractal se redacta una introducción teórica a la misma acompañada de imágenes, códigos y una web interactiva en la que poder ver desde cerca por uno mismo los fractales.

Recomendamos que el lector cuente con ciertos conocimientos básicos en algunas áreas de la informática y las matemáticas. Con respecto a las matemáticas, aunque la mayor parte de la materia está explicada, es deseable que el lector tenga nociones de las materias que describimos a continuación:

- Análisis matemático básico: En la mayoría de los desarrollos matemáticos más elaborados se parte del concepto de ‘espacio métrico’, haciendo uso exhaustivo de la distancia. También se emplea mucho el concepto de sucesión y convergencia (o divergencia) de sucesiones. Es igualmente necesario conocer los conceptos de continuidad de aplicaciones.
- Teoría de la medida: Aunque no se entre en detalle en estos aspectos, viene bien tener claro la medida usual de Lebesgue en \mathbb{R}^n y las bases de teoría de la medida en general.
- Variable compleja: Es prácticamente indispensable que el lector maneje bien los números complejos, propiedades del módulo y potencias enteras de números complejos. Esto, junto con alguna pincelada sobre funciones complejas sería suficiente para abordar el texto desde este punto de vista.
- Geometría afín euclídea: Se utilizan conceptos como transformaciones afines, tanto en 2D como en 3D, sistemas de referencia, movimientos rígidos, vectores, etc.
- Matemática aplicada: Utilizamos la iteración y el ordenador para programar los algoritmos de visualización de fractales, aplicando al mundo computacional los procedimientos explicados teóricamente para obtener algo tan tangible como son las imágenes.

- Topología: Aunque en realidad en ningún momento se utiliza una topología distinta de la usual en \mathbb{R}^n , es conveniente controlar conceptos como el de conjunto abierto, conjunto cerrado, entorno, compacidad o conexión.

Por su parte, para el desarrollo del producto software se han empleado intensamente técnicas de informática gráfica. A modo de base, es recomendable que el lector conozca aproximadamente la arquitectura hardware de un ordenador, aunque bastaría con saber diferenciar entre la unidad central de procesamiento (CPU, el procesador) y la unidad de procesamiento gráfico (GPU, la gráfica). También sería útil el conocimiento de conceptos estándar de informática gráfica como ‘rasterización’, ‘ray-tracing’, los distintos tipos de coordenadas (de mundo, de cámara y de dispositivo), las transformaciones entre estas coordenadas mediante matrices de vista y proyección, modelos de iluminación, texturas, etc. Todos estos conceptos además suponen un complemento a la geometría euclídea anteriormente descrita.

Otro aspecto deseable es el conocimiento o uso previo de alguna herramienta de rasterización y sus componentes, y en concreto OpenGL/WebGL. Específicamente, sería útil conocer el concepto de shader y su programación en un lenguaje específico para la GPU (‘shading language’). Aunque estos detalles no son indispensables pues se explican estas utilidades y su uso en la propia memoria.

Para el desarrollo de la web interactiva se ha utilizado naturalmente HTML5 y CSS3 (ayudado también por el framework Bootstrap). Pero la mayor parte del código del software está escrito en JavaScript (utilizando también la biblioteca jQuery) ya que es utilizado para la gestión del DOM y de los eventos en el documento HTML. Además, como herramienta de visualización se ha utilizado WebGL, que precisamente utiliza JavaScript para su gestión, de manera que lo usamos como intermediario entre el documento HTML y la gestión dinámica de eventos, pero a su vez también lo utilizamos como lenguaje para interactuar con las herramientas de WebGL.

Se utiliza el lenguaje GLSL (GL Shader Language) para la programación de código destinado a ejecutarse en la GPU. A las componentes de este código se les denomina ‘shaders’. El uso de este lenguaje es intensivo, pero tiene una sintaxis muy similar a la de C, por lo que el lector no debe preocuparse si no ha tratado nunca con GLSL, ya que entre la propia documentación oficial y lo descrito aquí será sencillo entender todo el código shader.

Para graficar escenas 3D hemos utilizado la técnica conocida como ‘ray-tracing’, identificando cada objeto con una función distancia (SDF) y aplicando el algoritmo ‘sphere-tracing’. Con estos elementos se ha implementado en GLSL el shader necesario para graficar las escenas en las que se pueden observar fractales 3D.

Partiendo de estas bases, comenzaremos esta memoria explicando de forma genérica e intuitiva el concepto de fractal, para posteriormente introducir importantes conceptos como autosimilaridad y distintos tipos de dimensión fractal. Seguidamente introduciremos la iteración de funciones complejas como herramienta para la generación de imágenes fractales mediante la coloración del plano complejo utilizando el método de Newton.

A continuación, y sin alejarnos de la iteración de funciones complejas, presentaremos la familia de funciones $P_c(z) = z^2 + c$, con la cual aprenderemos a visualizar y a interpretar el significado de los conjuntos de Julia y Mandelbrot, muy conocidos en el ámbito de los fractales. También presentaremos algunas generalizaciones de estos conjuntos mediante el cambio en el

exponente de P_c , iterando por tanto la función $P_{c,m} = z^m + c$, y mediante la iteración de funciones no polinómicas.

Continuaremos nuestro camino introduciéndonos en la teoría de los sistemas de funciones iteradas, que se basan también en iterar funciones, pero esta vez sobre conjuntos de un espacio métrico completo, con especial interés en \mathbb{R}^2 . Desarrollaremos una introducción a los mismos y demostraremos la convergencia de esta particular sucesión de iteradas a objetos de naturaleza fractal.

Seguidamente introduciremos el uso de herramientas de rasterización con el objetivo de conseguir visualizar fractales tanto 2D y 3D. Describiremos conceptos genéricos de informática gráfica y APIs de rasterización para el que lo necesite y en particular introduciremos el uso y los componentes de una aplicación gráfica basada en WebGL. Tras esta breve introducción, explicaremos la planificación, presupuesto, análisis de requisitos y análisis de casos de uso del producto software que se desarrollará en los siguientes capítulos. Tras esto, modificaremos la estructura y programaremos el shader para visualizar conjuntos de Julia y Mandelbrot, estándar y generalizados, aprovechando la teoría que introdujimos en capítulos anteriores.

Con los fractales en dos dimensiones ya bastante asimilados, es momento de dar el salto a las 3 dimensiones. Para ello, aprenderemos los fundamentos básicos del ray-tracing, con el objetivo primitivo de programar un programa ray-tracer que únicamente visualice una escena sencilla con varias esferas y un suelo con textura de ajedrez. Aplicaremos también un modelo de iluminación local para dotar de luces, sombras y realismo a la escena.

Una vez aprendidas las bases del ray-tracing, modificaremos el programa para visualizar fractales tridimensionales, aplicando el algoritmo ‘sphere-tracing’ para obtener imágenes de conjuntos de Julia y Mandelbrot generalizados a tres dimensiones.

Para el desarrollo de esta memoria las fuentes fundamentales consultadas han sido *Iteración y Fractales (con Mathematica)* de Gustavo N. Rubiano [27], un libro en el que se abordan los principios de la geometría fractal con Mathematica, aunque de una manera muy superficial y poco rigurosa, por ello ha sido completado con más bibliografía. Particularmente, el capítulo 4 sobre sistemas de funciones iteradas ha sido fundamentalmente basado en la teoría descrita por Michael Barnsley en *Fractals everywhere* [3].

En temas relacionados con la visualización de fractales con ray-tracing se ha adaptado el código C++ de *Ray Tracing in One Weekend*, de Peter Shirley [45] a código GLSL ejecutado en la GPU. Para graficar fractales se han consultado principalmente los artículos de Íñigo Quilez, disponibles en su web personal <https://iquilezles.org/articles/>, los artículos de John Hart de 1989 [14] y 1995 [15] y el artículo [6] de Keenan Crane.

Sin mucho más que añadir, el humilde autor de este trabajo desea una experiencia agradable durante su lectura y el disfrute de cada una de las secciones.

OBJETIVOS

Los objetivos iniciales de este trabajo son muy similares a los finalmente alcanzados, aunque algunos finalmente han tenido que ser descartados.

En el ámbito de las matemáticas, el objetivo inicial consistía en realizar un estudio de los fundamentos básicos de la Geometría Fractal. Para ello, se realizaría una recopilación de algunos de los conceptos y resultados necesarios, insistiendo en la teoría de la medida y el teorema del punto fijo de Banach. Se desarrollarían los elementos esenciales de la Geometría Fractal, especialmente conceptos de dimensión, como la de Hausdorff, sistemas de funciones iteradas, autosimilitud o el teorema del collage. Finalmente, se presentarían diversas aplicaciones de la Geometría Fractal, especialmente vinculadas con la naturaleza.

En efecto, se han recopilado los resultados más importantes de la geometría fractal. La autosimilitud o autosimilaridad se expone en la definición 1.0.1 y para clarificarla se exponen numerosos ejemplos de fractales clásicos. Además, se referencia continuamente a este término al ser, junto con la dimensión, una de las características fundamentales que componen la definición de *fractal* según *Benoit Mandelbrot* (definición 1.2.3).

Sobre teoría de la medida, en el capítulo 1 se comprueba la nulidad de la medida de Lebesgue de algunos objetos fractales como el conjunto de Cantor en la sección 1.1.1 o el triángulo de Sierpinski en la 1.1.2. Se presenta la medida de Hausdorff en la sección 1.2.2, la cual nos lleva directamente al concepto de dimensión de Hausdorff. Sobre dimensión, además de una introducción básica, se definen conceptos como la dimensión por cajas en el apartado 1.2.1, la ya mencionada dimensión de Hausdorff en el 1.2.2 y la dimensión topológica en el 1.2.3, para finalmente relacionar entre sí todas las definiciones en la sección 1.2.4.

Por su parte, el teorema del punto fijo de Banach (teorema 2.1.1) está enunciado y demostrado en la sección 2.1.1 y desde ese momento se convierte en una herramienta fundamental en toda la materia y en especial en el capítulo 4, el cual trata de Sistemas de Funciones Iteradas (SFI), concepto que también era objetivo inicial de este trabajo. Los SFI y su convergencia están muy basados en el teorema del punto fijo de Banach y el teorema del collage (teorema 4.4.1) es de hecho una consecuencia del mismo.

Sin embargo, aunque se hacen continuas referencias a la naturaleza con objetos como un romanescu o un rayo, como en las imágenes 1.1 , o se hayan descrito SFIs para modelar un árbol o un helecho en las tablas 4.1 y 4.2, cuyos resultados se pueden ver en las imágenes 4.6; lo cierto es que no hay ningún énfasis en aplicaciones de la geometría fractal en la naturaleza.

Podemos encontrar en el capítulo 4 algunas aplicaciones de la geometría fractal y del teorema del collage, pero realmente no existen explicaciones minuciosas. Esto se debe a que estos cuatro capítulos desarrollados se han considerado suficientes para completar los créditos y los objetivos fundamentales de un proyecto de estas características. No obstante, si el lector lo desea puede encontrar en [21] una gran cantidad de relaciones entre la naturaleza y la geometría fractal.

En relación a los objetivos y contenidos de informática, los objetivos iniciales consistían en estudiar diversos algoritmos conocidos para visualización de fractales en 2D y 3D, especialmente aquellos basados en Ray-tracing en 3D. Tras esto, se realizaría un análisis de sus características, con énfasis en la eficiencia en tiempo. Estos algoritmos se diseñarían e implementarían usando hardware gráfico moderno, y se evaluarían los resultados obtenidos en cuanto a tiempo de cálculo y funcionalidad ofrecida.

Hemos conseguido programar varios algoritmos para visualizar distintos fractales, partiendo de los que se describieron en el capítulo 3: los algoritmos 1 y 2. A partir de ellos describimos en el capítulo 7 cómo graficar fractales 2D utilizando WebGL. Seguidamente empezamos a codificar una aplicación que utilice ray-tracing para visualizar una escena 3D sencilla, para una vez completada implementar algoritmos de visualización de fractales con ray-tracing. Esta tarea fue muy compleja y costosa, prueba de ello es que tan solo los capítulos 8 y 9, correspondientes a introducción al ray-tracing y a visualización de fractales 3D, ocupan aproximadamente 70 páginas. La bibliografía sobre esta materia es bastante escasa y en muchas ocasiones ambigua, además de la dificultad que supone depurar código shader, por lo que fue muy costoso completar esta tarea.

Los objetivos relativos a la eficiencia en tiempo también se han logrado alcanzar. Aunque queda pendiente un análisis más profundo de la comparativa existente entre una ejecución de este código en GPU respecto al uso de una versión equivalente totalmente ejecutada en CPU, lo cierto es que con el código desarrollado es posible visualizar fractales 2D y 3D en tiempos del orden de milisegundos, los cuales son imposibles de conseguir con código ejecutado exclusivamente en la CPU.

Con estos párrafos damos por finalizadas las secciones introductorias del proyecto, comenzando en las siguientes páginas con el contenido concreto del TFG.

CAPÍTULO 1

EL CONCEPTO DE *FRACTAL*

Las primeras preguntas que se pueden plantear son ¿qué es un fractal? ¿Qué tienen de especial estas figuras? ¿Qué las diferencia de un objeto no fractal? Trataremos de responder a cada una de estas preguntas a lo largo de este capítulo, comenzando por la primera de ellas. En realidad, hay distintas definiciones de *fractal*, pero todas utilizan dos conceptos como base: la **autosimilaridad** y la **dimensión**. La primera de ellas es más cercana para nosotros de lo que en un principio podemos pensar, fijémonos en los ejemplos de la imagen 1.1.



(a) Romanescu



(b) Rayo

Imagen 1.1: Objetos de la naturaleza con estructura fractal

Observemos que el romanescu, que es un tipo de coliflor, pareciera que está formado de pequeños fragmentos que recuerdan el objeto original, mientras que estos a su vez también están formados de pequeños trozos similares al objeto inicial, y así sucesivamente. Por su parte, el rayo se compone de un destello principal del que salen ramificaciones de las que a su vez se originan otras divisiones, formando pequeños rayos semejantes al rayo primitivo.

Esta idea de objetos prácticamente iguales al original salvo cambios de escala es la subyacente al concepto de autosimilaridad.

Definición 1.0.1 (Autosimilaridad). Una figura o subconjunto A de \mathbb{R}^n es **autosimilar** si está compuesto por copias de sí mismo reducidas mediante un factor de escala y desplazadas por un movimiento rígido. Es decir,

$$A = \bigcup_{i=1}^n f_i \circ h_i(A),$$

donde cada $h_i, i = 1, \dots, n$, es una homotecia de razón menor que 1 y $f_i, i = 1, \dots, n$, es un movimiento rígido.

En futuras ocasiones se utilizarán indistintamente los términos de «reducción por un factor de escala» e «imagen vía una homotecia», eludiendo mencionar el movimiento rígido por simplicidad expositiva, si bien entendemos implícitamente que hay siempre uno asociado a la homotecia.

Para afianzar y formalizar conceptos y con el objetivo de introducir una definición de la dimensión, estudiaremos algunos ejemplos clásicos de objetos fractales.

1.1. Ejemplos clásicos

En adelante, y salvo que se indique lo contrario, cuando hablamos en términos topológicos de \mathbb{R}^n o subconjuntos suyos nos estaremos refiriendo al espacio topológico \mathbb{R}^n dotado de la topología usual o la topología inducida por la usual en el caso de subconjuntos de \mathbb{R}^n .

1.1.1. El conjunto de Cantor

Creado por el célebre matemático *George Cantor*, este fractal se construye a partir de un intervalo aplicando el siguiente proceso iterativo:

1. Partimos del intervalo cerrado $[0, 1]$, aunque realmente es indiferente cuál se escoja, pues el resultado final será el mismo salvo una homotecia. Dividimos dicho intervalo en tres intervalos iguales y extraemos el intervalo central, manteniendo los extremos. Es decir, extraemos el intervalo abierto $(\frac{1}{3}, \frac{2}{3})$ y mantenemos $[0, \frac{1}{3}]$ y $[\frac{2}{3}, 1]$. Nótese que obtenemos $2 = 2^1$ intervalos, cada uno a escala $\frac{1}{3} = (\frac{1}{3})^1$ del original.
2. Aplicamos el mismo proceso a los intervalos $[0, \frac{1}{3}]$ y $[\frac{2}{3}, 1]$. Esto es, se dividen ambos en tres partes iguales y se extrae el intervalo abierto central de cada uno de ellos, manteniendo los extremos. En este caso obtendríamos $4 = 2^2$ intervalos iguales, cada uno de ellos a escala $\frac{1}{9}$ de los dos obtenidos en el primer paso y a escala $\frac{1}{9} = (\frac{1}{3})^2$ del original.
3. Repetimos este proceso de manera indefinida, de manera que en el n -ésimo paso se obtendrán 2^n intervalos a escala $(\frac{1}{3})^n$ del original. Denotemos como C_n al conjunto unión de los 2^n intervalos que se generan en el paso n del proceso.

Los puntos del intervalo inicial $[0, 1]$ que restan tras las infinitas iteraciones son los que conforman el *conjunto de Cantor*, que denotamos con **C**, de forma que

$$\mathbf{C} = \bigcap_{n \in \mathbb{N}} C_n.$$

El conjunto de Cantor es además un conjunto compacto, pues cada C_n es una unión finita de intervalos cerrados y acotados de \mathbb{R} , y por tanto compactos, como sabemos gracias al *teorema*

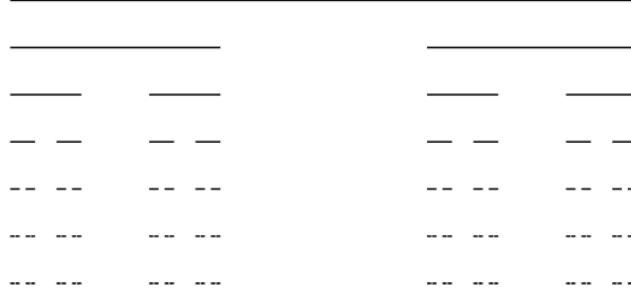


Imagen 1.2: Iteraciones del proceso de generación del conjunto de Cantor

de Heine-Borel. Sabiendo que la intersección arbitraria de conjuntos cerrados es cerrada, y que cada C_n está acotado, tenemos pues que \mathbf{C} es un conjunto compacto.

Observemos ahora que en la primera iteración eliminamos 1 intervalo de longitud $\frac{1}{3}$, en la segunda iteración se eliminan 2 intervalos de longitud $(\frac{1}{3})^2$ y en la n -ésima iteración extraemos 2^{n-1} intervalos de longitud $(\frac{1}{3})^n$. Si sumamos las longitudes de todos los intervalos que son eliminados en cada paso se obtiene:

$$\begin{aligned} \sum_{n=1}^{\infty} 2^{n-1} \left(\frac{1}{3}\right)^n &= \frac{1}{3} \sum_{n=0}^{\infty} 2^n \left(\frac{1}{3}\right)^n \\ &= \frac{1}{3} \sum_{n=0}^{\infty} \left(\frac{2}{3}\right)^n \\ &= \frac{1}{3} \left(\frac{1}{1 - \frac{2}{3}} \right) \\ &= 1, \end{aligned}$$

donde hemos usado que la suma de una serie geométrica de razón $|q| < 1$ es

$$\sum_{n=0}^{\infty} q^n = \frac{1}{1-q}.$$

Esto nos lleva a concluir, teniendo en cuenta que para los subconjuntos de \mathbb{R} considerados su longitud no es más que su medida usual de Lebesgue, que la longitud eliminada es igual a la longitud original, es decir, la longitud de \mathbf{C} es nula y aún así tenemos infinitos puntos, por ejemplo los extremos de los intervalos que se van generando. Es decir, los puntos de \mathbf{C} no están agrupados, sino que forman una especie de *polvareda*.

Mencionamos finalmente que el conjunto de Cantor posee importantes propiedades desde un punto de vista topológico, como por ejemplo el conocido teorema de Alexandroff-Hausdorff, que establece que cualquier espacio topológico compacto es imagen continua de \mathbf{C} . Puede encontrarse una prueba de este resultado en [8]. Este resultado tiene además algunas aplicaciones en otros campos de las matemáticas, por ejemplo en curvas y en conjuntos convexos. Si se desea profundizar en estas aplicaciones recomendamos consultar [4].

1.1.2. El triángulo de Sierpinski

Esta figura, original del polaco *Waclaw Sierpinski*, es creada de una manera que evoca al conjunto de Cantor, pero en dos dimensiones. Veamos detalladamente el proceso (ver imagen

1.3):

1. Se parte de un triángulo equilátero de lado 1 (de nuevo la longitud inicial es irrelevante). Uniendo los puntos medios de cada lado obtenemos una partición del triángulo inicial en 4 triángulos equiláteros, del cual extraemos el interior del triángulo central. Tenemos por tanto $3 = 3^1$ triángulos a escala $\frac{1}{2} = \left(\frac{1}{2}\right)^1$ del original.
2. En cada uno de estos tres triángulos equiláteros se repite la operación anterior, obteniendo por tanto $9 = 3^2$ triángulos, cada uno a escala $\frac{1}{4} = \left(\frac{1}{2}\right)^2$ del original.
3. Repetimos este proceso indefinidamente, de forma que en el paso n -ésimo se tienen 3^n triángulos, cada uno de ellos a escala $\left(\frac{1}{2}\right)^n$ del original.

La figura a la que converge este proceso infinito se conoce como triángulo **S** de Sierpinski.



Imagen 1.3: Generación del triángulo de Sierpinski

Si llamamos A al área del triángulo inicial, que es de hecho su medida usual de Lebesgue, sabemos que en la primera iteración eliminamos un área de $\frac{1}{4}A$, en el segundo eliminamos $3\left(\frac{1}{4}\right)^2 A$ y en el n -ésimo $3^{n-1}\left(\frac{1}{4}\right)^n A$, de forma que si sumamos todo el área que eliminamos en cada paso obtenemos:

$$\begin{aligned} A \sum_{n=1}^{\infty} 3^{n-1} \left(\frac{1}{4}\right)^n &= \frac{A}{4} \sum_{n=0}^{\infty} 3^n \left(\frac{1}{4}\right)^n \\ &= \frac{A}{4} \sum_{n=0}^{\infty} \left(\frac{3}{4}\right)^n \\ &= \frac{A}{4} \left(\frac{1}{1 - \frac{3}{4}} \right) \\ &= A. \end{aligned}$$

En este caso ocurre algo parecido a lo que vimos que sucedía con el conjunto de Cantor en la sección 1.1.1. El área eliminada es igual al área total, es decir, su área, su medida usual de Lebesgue, es 0 y seguimos teniendo infinitos puntos (por ejemplo los vértices de los triángulos originados en cada iteración). Es decir, los puntos que forman **S** no están agrupados formando un área.

1.1.3. La alfombra de Sierpinski y la esponja de Menger

El propio Sierpinski se dio cuenta que con el mismo patrón utilizado para generar **S** se pueden obtener otras formas. Por ejemplo, pensemos que en lugar de comenzar con un triángulo equilátero partimos de un cuadrado, lo subdividimos en 9 cuadrados de lado $\frac{1}{3}$ y extraemos el

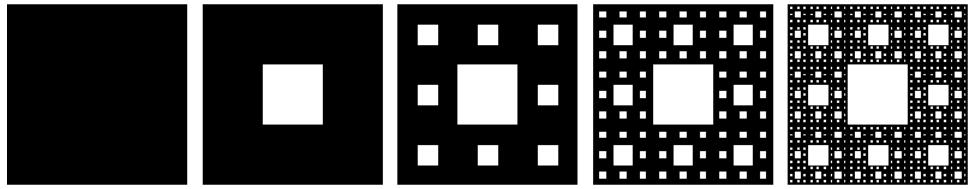


Imagen 1.4: Generación de la alfombra de Sierpinski

cuadrado central. Repitiendo este proceso indefinidamente con cada uno de los cuadrados que se generan finalmente se obtiene la llamada alfombra de Sierpinski (ver imagen 1.4).

Este proceso también se puede modelar en 3D, obteniendo así la conocida como esponja de Menger o cubo de Magritte, que es una generalización en tres dimensiones de la alfombra de Sierpinski, la cual podemos ver en la imagen 1.5.

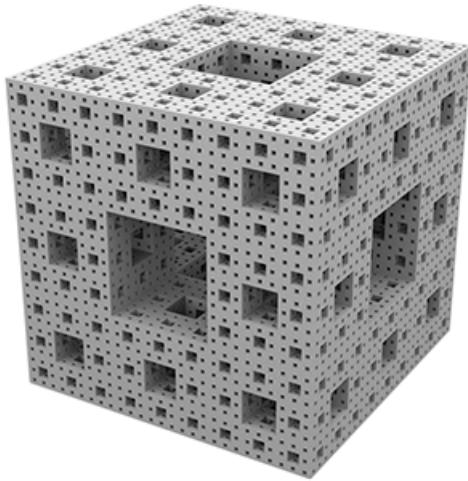


Imagen 1.5: Esponja de Menger

1.1.4. La curva de Koch

Esta figura fractal, creada por el sueco *N. F. Helge von Koch* sigue un proceso de construcción iterativo al igual que el conjunto de Cantor, pero en lugar de eliminar segmentos, se añaden de la siguiente manera (ver imagen 1.6):

1. Partiendo de un segmento de recta de longitud 1 (al igual que en el conjunto de Cantor, la longitud del segmento inicial es irrelevante, pues la figura final es la misma salvo homotecia), se divide en tres partes iguales de longitud $\frac{1}{3}$ y la parte central se sustituye por un triángulo equilátero al que se le elimina la base. Esto da lugar a $4 = 4^1$ segmentos de recta de longitud $\frac{1}{3} = \left(\frac{1}{3}\right)^1$.
2. Repetimos este proceso en cada uno de los segmentos de recta obtenidos, colocando el triángulo siempre por encima de la recta, obteniendo así $16 = 4^2$ segmentos de recta de longitud $\frac{1}{9} = \left(\frac{1}{3}\right)^2$.
3. Aplicamos este proceso indefinidamente, obteniendo en el paso n -ésimo 4^n segmentos de longitud $\left(\frac{1}{3}\right)^n$.

El resultado final del proceso es lo que llamamos la *curva de Koch*, que denotamos como **K**.

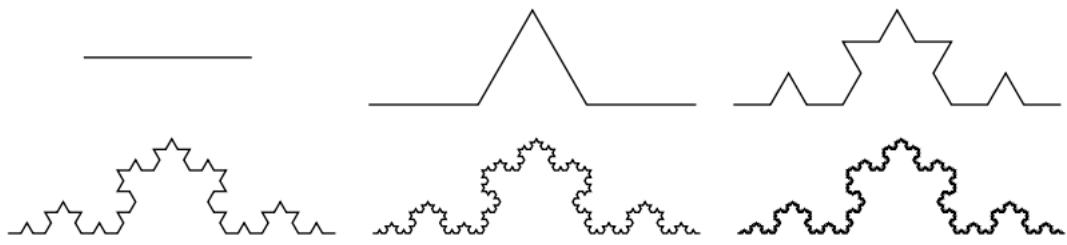


Imagen 1.6: Iteraciones del proceso de generación de la curva de Koch

1.1.5. El copo de nieve de Koch

A partir de la curva de Koch podemos generar un objeto matemático muy particular: el copo de nieve de Koch. Para crearlo, basta aplicar el proceso iterativo descrito para generar la curva de Koch a cada uno de los segmentos que componen un triángulo equilátero, de forma que los triángulos que se generan apunten hacia el exterior.

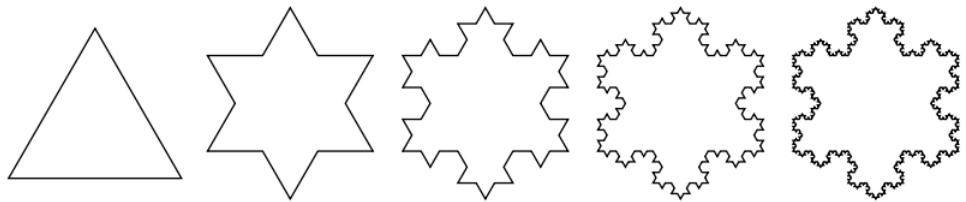


Imagen 1.7: Generación del copo de nieve de Koch

Esta curva posee la particularidad de tener longitud infinita y a su vez encerrar un área finita. Realmente el copo de Koch no es exactamente un fractal, pues no es totalmente autosimilar, aunque se compone de tres partes idénticas las cuales sí son autosimilares, como podemos ver en la imagen 1.8.

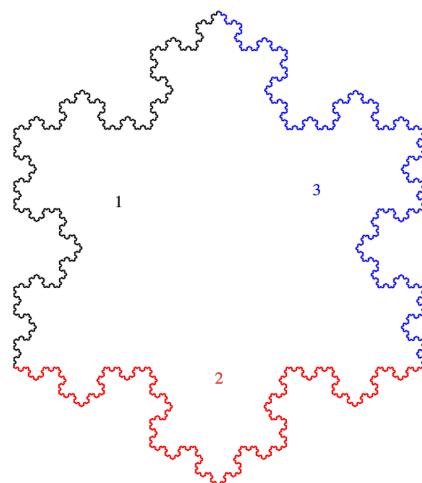


Imagen 1.8: Curvas de Koch que componen el copo de Koch

1.2. Conceptos de dimensión fractal

Al iniciar este capítulo mencionamos que las distintas definiciones de fractal utilizaban los conceptos de autosimilaridad y dimensión. Con los distintos ejemplos hemos entendido el primero de ellos, por lo que es momento de abordar el concepto de dimensión.

El concepto de dimensión más claro que tenemos es el de dimensión algebraica, esto es, la dimensión de un espacio vectorial. Sabemos que un espacio vectorial V se dice que tiene $\dim(V) = n$, con $n \in \mathbb{N}$, si, y solo si existe una base de V constituida por n vectores, de forma que cualquier elemento de V puede ser expresado de forma única como una combinación lineal de los n vectores de la base. Otra manera de ver esto es que para construir los vectores de V tenemos hasta n parámetros de libertad, esto es, $v = a_1v_1 + \dots + a_nv_n \quad \forall v \in V$, donde $\{v_1, \dots, v_n\}$ es una base de V y $a_1, \dots, a_n \in \mathbb{K}$ siendo \mathbb{K} el cuerpo sobre el que está construido el espacio vectorial.

En el caso de subconjuntos de \mathbb{R}^n como puede ser una curva parametrizada, si seguimos la analogía del número de parámetros que define un punto en este caso de una curva, podemos decir que su dimensión es 1, ya que un punto de una curva parametrizada puede expresarse en función de un único parámetro. La variación de este parámetro nos daría otro punto de la curva, por lo que podemos decir que un punto puede moverse por la curva con un grado de libertad (ver imagen 1.9 (a)). Por su lado, una superficie regular de \mathbb{R}^3 puede localmente ser expresada como la imagen de una parametrización que depende de dos variables y la variación de estas mediante dicha parametrización resulta en otro punto de la superficie, pudiendo expresar esto como que un punto de una superficie regular tiene dos grados de libertad, lo que intuitivamente permite afirmar que una superficie regular tiene dimensión 2 (ver imagen 1.9 (b)).

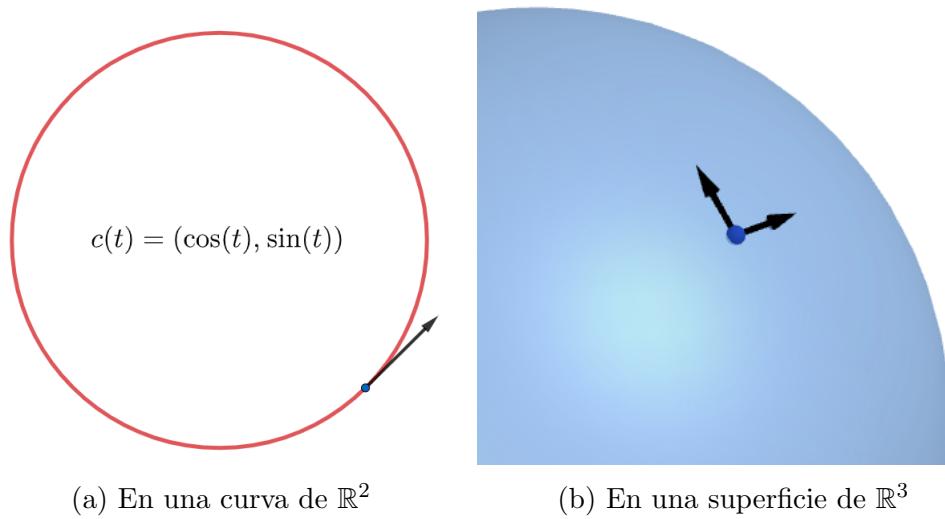


Imagen 1.9: Posible movimiento de un punto en posibles objetos de \mathbb{R}^n

Un posible enfoque para definir la dimensión puede ser el recién presentado, el cual nos sugiere pensar en el número de parámetros que definen la libertad de movimiento de un punto. Sin embargo, pensemos ahora en el triángulo de Sierpinski y en su dimensión. Comprobamos en la sección 1.1.2 que su área 2-dimensional es nula, pero en el objeto final pareciera que un punto se pudiera mover en varias direcciones. No se puede afirmar que \mathbf{S} tenga dimensión 1 por

la movilidad, pero tampoco dimensión 2 porque tiene área 0, luego sería un valor situado entre estos dos enteros.

1.2.1. Dimensión por cajas

Pensemos ahora en un segmento de recta, un cuadrado y un cubo, que son objetos indudablemente de 1, 2 y 3 dimensiones respectivamente. Ahora dividamos los lados de cada objeto en 2 tal y como indica la imagen 1.10. Entonces vemos que podemos recubrir el segmento con $N(2) := 2 = 2^1$ segmentos de longitud $\frac{1}{2}$, el cuadrado se puede cubrir con $N(2) := 4 = 2^2$ cuadrados de lado $\frac{1}{2}$ y el cubo con $N(2) := 8 = 2^3$ cubos, de nuevo cada uno de ellos de lado $\frac{1}{2}$.

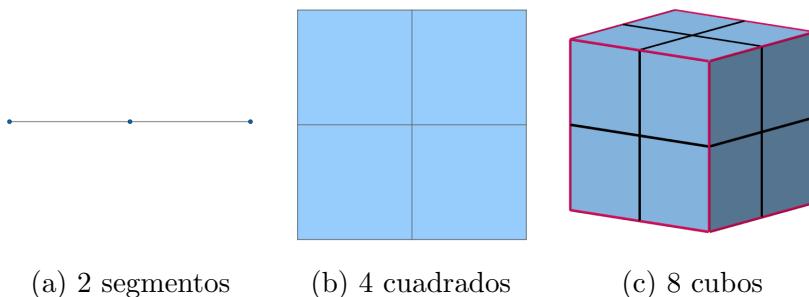


Imagen 1.10: Segmento, cuadrado y cubo recubiertos por objetos de lado $\frac{1}{2}$

Si en lugar de 2 tomamos cualquier número natural $k \geq 1$, los recubrimientos serían de $N(k) = k^1$ segmentos de recta, $N(k) = k^2$ cuadrados y $N(k) = k^3$ cubos, en todos los casos de lado $r = \frac{1}{k}$. Estas igualdades se pueden reescribir como:

$$\frac{N(k)}{k^1} = 1 \quad \frac{N(k)}{k^2} = 1 \quad \frac{N(k)}{k^3} = 1 \quad (1.1)$$

En este sentido, vemos que la dimensión de cada objeto es el *exponente* al que habría que elevar la longitud del lado $1/k$ para obtener la relación (1.1). Por lo que si llamamos d a este valor y lo despejamos, nos quedaría

$$d = \frac{\log(N(k))}{\log(k)}.$$

De manera análoga al segmento, el cuadrado y el cubo podemos tomar cualquier figura o subconjunto $X \subseteq \mathbb{R}^n$ que pueda ser recubierto por otros conjuntos de \mathbb{R}^n más pequeños.

Definición 1.2.1. Para un subconjunto U de \mathbb{R}^n , definimos su **diámetro** como:

$$\text{diam}(U) = \sup \{d(x, y) : x, y \in U\}.$$

Dado un conjunto $A \subset \mathbb{R}^n$ no vacío y acotado, sea $N_\delta(A)$ el mínimo número de conjuntos de diámetro a lo sumo δ necesario para recubrir A . La *dimensión por cajas superior* y la *dimensión por cajas inferior* se definen, respectivamente como

$$\begin{aligned} \underline{\dim}_B(A) &:= \liminf_{\delta \rightarrow 0} \frac{\log(N_\delta(A))}{\log(1/\delta)}, \\ \overline{\dim}_B(A) &:= \limsup_{\delta \rightarrow 0} \frac{\log(N_\delta(A))}{\log(1/\delta)}. \end{aligned}$$

¹Nótese la analogía con la notación de los límites superior e inferior como $\underline{\lim}$ y $\overline{\lim}$.

En caso de coincidir, se denomina *dimensión por cajas* de A al valor

$$\dim_B(A) := \lim_{\delta \rightarrow 0} \frac{\log(N_\delta(A))}{\log(1/\delta)}. \quad (1.2)$$

También es conocida en literatura como *dimensión por conteo de cajas* o *dimensión de Minkowski-Bouligand*.

Hay varias definiciones equivalentes, generalmente más sencillas de utilizar. Por ejemplo, si dividimos \mathbb{R}^n con conjuntos de la forma

$$[m_1\delta, (m_1 + 1)\delta] \times \cdots \times [m_n\delta, (m_n + 1)\delta]$$

donde m_1, \dots, m_n son enteros, tendríamos \mathbb{R}^n dividido en cuadrados de \mathbb{R}^n de lado δ , comúnmente conocidos como ‘cajas’, de diámetro $\delta\sqrt{n}$ y contando el número de cajas que recubren a A , obtenemos el mismo resultado, puede comprobarse en [12, sección 3.1].

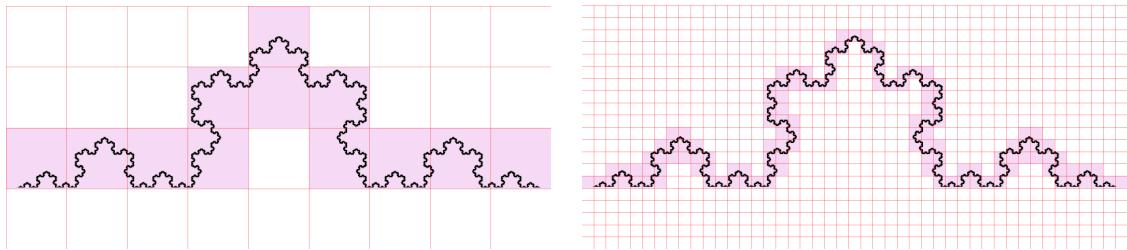


Imagen 1.11: Una forma de calcular la dimensión por cajas de la curva de Koch

Un ejemplo clarificador podría ser el de un triángulo equilátero T de lado 1, el cual podemos recubrir con 4 triángulos equiláteros de lado $\frac{1}{2}$. A su vez podríamos recubrir estos 4 triángulos con otros 4 triángulos de lado $\frac{1}{4}$, obteniendo así un recubrimiento de T con $16 = 2^{2 \cdot 2}$ triángulos de lado $\frac{1}{4} = \frac{1}{2^2}$. Si repetimos este proceso y teniendo en cuenta que el diámetro de un triángulo equilátero es la longitud de sus lados, podemos tomar $\delta = \frac{1}{2^n}$ y recubrir T con $N_\delta(T) = 2^{2n}$ triángulos equiláteros. Aplicando (1.2), tenemos por tanto que

$$\lim_{n \rightarrow \infty} \frac{\log(2^{2n})}{\log(2^n)} = \lim_{n \rightarrow \infty} \frac{2n \log(2)}{n \log(2)} = 2.$$

Pero esto no nos asegura que esta sea la dimensión por cajas del triángulo, pues en ningún momento hemos probado que $\underline{\dim}_B(T) = \overline{\dim}_B(T)$, en cuyo caso podríamos utilizar la ecuación (1.2). Este cálculo únicamente nos dice que en caso de existir $\dim_B(T)$, ésta vale 2. En la sección 4.3 del capítulo 4 comprobaremos mediante una serie de resultados que efectivamente $\dim_B(T) = 2$.

De manera similar podemos intuir qué valor tendría la dimensión por cajas de algunos fractales clásicos. Retomemos ahora el triángulo **S** de Sierpinski, que por su génesis (véase sección 1.1.2) puede ser inicialmente cubierto por un triángulo equilátero de lado 1, el cual si lo dividimos en 4 triángulos equiláteros y sustraemos el central sigue recubriendo a **S**, de forma que tenemos un recubrimiento de 3 triángulos de lado $\frac{1}{2}$. Si repetimos esta operación con cada uno de los tres triángulos podríamos recubrir el triángulo de Sierpinski con 9 triángulos equiláteros de lado $\frac{1}{4}$, y así sucesivamente. Por tanto, tomando $\delta = \frac{1}{2^n}$ obtenemos un recubrimiento de $N_\delta(\mathbf{S}) = 3^n$ triángulos, y podemos hacer el cálculo:

$$\lim_{n \rightarrow \infty} \frac{\log(3^n)}{\log(2^n)} = \lim_{n \rightarrow \infty} \frac{n \log(3)}{n \log(2)} = \frac{\log(3)}{\log(2)} \approx 1.58496.$$

Por lo que, de existir, la dimensión por cajas del triángulo de Sierpinski es $\frac{\log(3)}{\log(2)}$. Efectivamente y tal y como discutimos en el inicio de esta sección, sería un valor situado entre 1 y 2. En el capítulo 4 confirmaremos que $\dim_B(\mathbf{S}) = \frac{\log(3)}{\log(2)}$.

Por su parte, el conjunto de Cantor \mathbf{C} puede ser recubierto utilizando segmentos de recta, cuyo diámetro es precisamente la longitud de dicho segmento. Tomando $\delta = \frac{1}{3^n}$ podemos recubrir a \mathbf{C} con $N_\delta(\mathbf{C}) = 2^n$ segmentos de recta (véase sección 1.1.1). Por lo que, de existir, la dimensión por cajas del conjunto de Cantor sería

$$\lim_{n \rightarrow \infty} \frac{\log(2^n)}{\log(3^n)} = \lim_{n \rightarrow \infty} \frac{n \log(2)}{n \log(3)} = \frac{\log(2)}{\log(3)} \approx 0.63093,$$

que es un valor situado entre 0 y 1. También comprobaremos que $\dim_B(\mathbf{C}) = \frac{\log(2)}{\log(3)}$.

1.2.2. Medida y dimensión de Hausdorff

La dimensión por cajas tiene el inconveniente de que no tenemos garantizada la existencia de límite en la ecuación (1.2) o hipotéticos problemas con conjuntos más generales, por ejemplo conjuntos densos. Así, $F = \mathbb{Q} \cap [0, 1] \subset \mathbb{R}$ es un conjunto en el que cada punto por separado tiene obviamente dimensión por cajas igual a 0, pero al ser \mathbb{Q} un conjunto denso en \mathbb{R} la dimensión por cajas de F es 1. Por tanto, no se cumple en general que para una familia $\{F_i\}$ de conjuntos $\dim_B \bigcup_{i=1}^{\infty} F_i = \sup_i \dim_B F_i$. Para solucionar estos problemas *Felix Hausdorff* publicó en 1919 un artículo que cambiaría la teoría de la medida tal y como la conocíamos [16].

Si ahora tomamos un conjunto A de \mathbb{R}^n , un valor $\varepsilon > 0$ y una familia numerable de conjuntos $\{U_i\}_{i \in \mathbb{N}}$ tales que

$$A \subseteq \bigcup_{i \in \mathbb{N}} U_i, \quad 0 \leq \text{diam}(U_i) \leq \varepsilon \quad \forall i \in \mathbb{N}$$

se dice que la familia $\{U_i\}_{i \in \mathbb{N}}$ es un ε -recubrimiento de A . Si ahora consideramos un valor $s > 0$, definimos

$$\mathcal{H}_\varepsilon^s(A) := \inf \left\{ \sum_{n \in \mathbb{N}} \text{diam}(U_i)^s : \{U_i\}_{i \in \mathbb{N}} \text{ es un } \varepsilon\text{-recubrimiento de } A \right\}.$$

Si reducimos el valor de ε el número de posibles recubrimientos disminuye, por lo que $\mathcal{H}_\varepsilon^s(A)$ aumenta. Por esto nos planteamos cuál será el límite cuando ε tienda a cero, aceptando el infinito como posible valor del límite. Definimos así

$$\mathcal{H}^s(A) := \lim_{\varepsilon \rightarrow 0} \mathcal{H}_\varepsilon^s(A).$$

En [12, Secciones 5.2 y 5.4] se comprueba que $\mathcal{H}^s(A)$ es una medida definida en la σ -álgebra de Borel que llamamos *medida s-dimensional de Hausdorff* del conjunto A .

Veamos el comportamiento de $\mathcal{H}^s(A)$ como función de s . Es claro que siempre que $\varepsilon < 1$, $\mathcal{H}_\varepsilon^s(A)$ decrece conforme s aumenta, por tanto $\mathcal{H}^s(A)$ también es decreciente. Podemos de hecho probar el siguiente resultado.

Proposición 1.2.1. Sean $A \subset \mathbb{R}^n$, $t > s$, $0 < \varepsilon < 1$ y $\{U_i\}$ un ε -recubrimiento de A . Entonces

$$\mathcal{H}_\varepsilon^t(A) \leq \varepsilon^{t-s} \mathcal{H}_\varepsilon^s(A)$$

Demostración. Sabemos que $\text{diam}(U_i)^{t-s} \leq \varepsilon^{t-s} \forall i \in \mathbb{N}$ y del hecho de que $\sum_{i \in \mathbb{N}} \text{diam}(U_i)^t = \sum_{i \in \mathbb{N}} \text{diam}(U_i)^{t-s} \text{diam}(U_i)^s$ deducimos que

$$\sum_{i \in \mathbb{N}} \text{diam}(U_i)^t \leq \varepsilon^{t-s} \sum_{i \in \mathbb{N}} \text{diam}(U_i)^s.$$

Tomando el ínfimo en la anterior desigualdad tenemos que

$$\mathcal{H}_\varepsilon^t(A) \leq \sum_{i \in \mathbb{N}} \text{diam}(U_i)^t \leq \varepsilon^{t-s} \sum_{i \in \mathbb{N}} \text{diam}(U_i)^s,$$

por lo que, tomando el ínfimo en el segundo miembro,

$$\mathcal{H}_\varepsilon^t(A) \leq \varepsilon^{t-s} \mathcal{H}_\varepsilon^s(A).$$

□

Esta desigualdad nos será muy útil para probar el siguiente teorema que nos da la definición definitiva de lo que llamaremos dimensión de Hausdorff.

Teorema 1.2.1. *Sea $A \subset \mathbb{R}^n$. Entonces existe un único valor de s para el cual $\mathcal{H}^s(A)$ no es ni 0 ni ∞ . Este valor $s_0 = \dim_H(A)$ satisface que:*

$$\mathcal{H}^s(A) = \begin{cases} \infty & \text{si } s < \dim_H(A) \\ 0 & \text{si } s > \dim_H(A) \end{cases} \quad (1.3)$$

Demostración. Si tomamos $0 < \varepsilon < 1$ y $t > s$ dos valores reales, por la proposicion 1.2.1 tenemos que $\mathcal{H}_\varepsilon^t(A) \leq \varepsilon^{t-s} \mathcal{H}_\varepsilon^s(A)$. Por tanto, al hacer a ε tender a 0, siempre que $\mathcal{H}^s(A)$ sea finito necesariamente $\mathcal{H}^t(A) = 0$ para todo $t > s$. Si aplicamos el contrarrecíproco ocurre que si $\mathcal{H}^t(A) > 0$ entonces $\mathcal{H}^s(A) = \infty$.

Por lo que necesariamente debe de existir un valor $s_0 \in [0, \infty]$ tal que $\mathcal{H}^s(A) = 0 \forall s < s_0$ y $\mathcal{H}^s(A) = \infty \forall s > s_0$. □

Definición 1.2.2 (Dimensión de Hausdorff). Llamamos **dimensión de Hausdorff** de un conjunto A al único valor $\dim_H(A)$ que satisface las condiciones del teorema 1.2.1.

1.2.3. Dimensión topológica

Por último estudiaremos un concepto de dimensión aplicable a espacios topológicos. La **dimensión topológica** se define inductivamente de la siguiente forma:

1. La dimensión del conjunto vacío es $\dim_T(\emptyset) := -1$.
2. Un espacio topológico X tiene dimensión 0 ($\dim_T(X) = 0$) si para cualquier $x \in V$ con V abierto en X existe un abierto U cuya frontera $\partial(U)$ es vacía y se verifica que $x \in U \subseteq V$.
3. Un espacio topológico X tiene dimensión menor o igual que n ($\dim_T(X) \leq n$) si para cualquier $x \in X$ y cualquier V abierto que contiene a x existe un abierto U tal que $\dim_T(\partial(U)) \leq n-1$ y se verifica que $x \in U \subseteq V$.
4. X tiene dimensión n ($\dim_T(X) = n$) si se verifica que $\dim_T(X) \leq n$ pero es falso que $\dim_T(X) \leq n-1$.

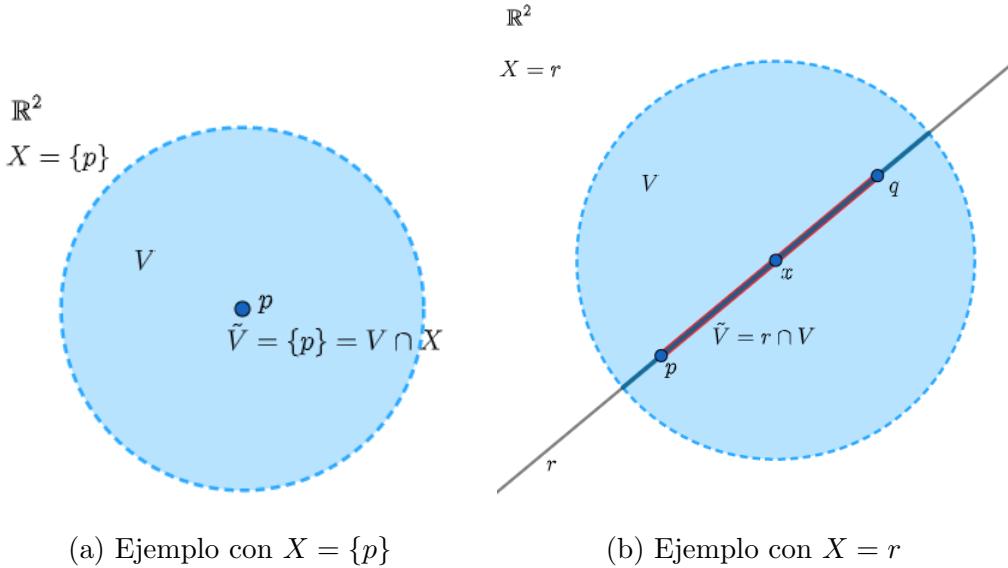


Imagen 1.12: Figuras representativas de los ejemplos

Ejemplo 1.2.1. (Véase imagen 1.12 (a)) En \mathbb{R}^2 dotado de la topología usual consideramos un punto p cualquiera y el espacio topológico $X = \{p\}$. Por tanto la topología inducida por la topología usual τ en X es $\tau_X = \{X, \emptyset\}$. Sea V un abierto de \mathbb{R}^2 que contenga a p , por tanto $\tilde{V} = V \cap X = \{p\}$ es un abierto en X que contiene a p . Podemos encontrar entonces un abierto $U = \{p\}$ de X tal que su frontera $\partial(U) = \emptyset$ y además $p \in U \subseteq \tilde{V}$. Por tanto concluimos que X tiene dimensión 0.

Ejemplo 1.2.2. (Véase imagen 1.12 (b)) En \mathbb{R}^2 dotado de la topología usual, consideramos una recta cualquiera, llamémosla r , y el espacio topológico $X = \{r\}$. Sea $x \in X$ y V un abierto de \mathbb{R}^2 que contenga a x , de forma que $\tilde{V} = r \cap V$ es un abierto de X . Podemos entonces tomar un abierto U dentro de \tilde{V} (un segmento abierto de recta), de forma que $x \in U \subseteq \tilde{V}$. Por otro lado, $\partial(U) = \{p, q\}$, y podemos comprobar fácilmente (con ayuda del ejemplo 1.2.1) que $\dim_T(\partial(U)) = 0$. Por todo esto podemos concluir que $X = r$ es un espacio topológico de dimensión 1.

1.2.4. Relación entre los distintos tipos de dimensión fractal

La dimensión por cajas, aunque útil en la práctica, al comienzo de la sección 1.2.2 hemos comprobado que tiene algunos problemas. No obstante, para muchos fractales, y en particular para los que cumplen la *condición de conjunto abierto* (véase sección 4.3), resulta más sencillo computacionalmente calcular su dimensión por cajas frente a su dimensión de Hausdorff.

En general, y como se puede comprobar en [12, Sección 3.1], ocurre que la dimensión por cajas acota superiormente a la dimensión de Hausdorff, es más, dado $A \subset \mathbb{R}^n$:

$$\dim_H(A) \leq \underline{\dim}_B(A) \leq \overline{\dim}_B(A),$$

donde, insistimos, es posible que se dé la igualdad.

Sobre la dimensión topológica, se tiene que si X es un espacio topológico separable², entonces

$$\dim_T(X) \leq \dim_H(X).$$

²Recordamos que un espacio topológico es *separable* si contiene un conjunto denso numerable

Este resultado fue originalmente probado por *Edward Szpilrajn*, véase [18, Capítulo VII].

Por otro lado, si nos restringimos a conjuntos totalmente autosimilares, existe un resultado que relaciona la dimensión de Hausdorff y la dimensión por cajas para tales conjuntos: el teorema de Moran (véase teorema 4.3.1). Este resultado nos asegura que en estos casos la dimensión por cajas y la dimensión de Hausdorff coinciden.

En conclusión, para un conjunto no vacío y acotado $A \subseteq \mathbb{R}^n$, se tiene que:

$$\dim_T(A) \leq \dim_H(A) \leq \dim_B(A) \leq n. \quad (1.4)$$

A partir de esta cadena de desigualdades, en la que notamos que la dimensión fractal, entendiendo por esta la dimensión de Hausdorff que es la más general, y que siempre excede o iguala a la dimensión topológica, podemos enunciar nuestra primera definición de fractal, que fue formulada por *Benoit Mandelbrot* en [21].

Definición 1.2.3 (Fractal). Un **fractal** es un subconjunto de \mathbb{R}^n que es autosimilar y cuya dimensión fractal excede a su dimensión topológica.

CAPÍTULO 2

ITERACIÓN

Como hemos podido comprobar en el capítulo anterior, muchos de los fractales clásicos son generados repitiendo indefinidamente un proceso. Iteración es el proceso de repetir una y otra vez un método, ocasionalmente sobre el resultado de la aplicación anterior. Este procedimiento es muy útil en muchas disciplinas matemáticas. Por ejemplo, existen métodos numéricos basados en la iteración como el método de *Jacobi* y *Gauss-Seidel* para resolución de sistemas de ecuaciones lineales, el método de *Newton-Raphson* para encontrar soluciones de ecuaciones, el método de *Runge-Kutta* para resolución numérica de ecuaciones diferenciales, etc. Incluso en otras disciplinas como el aprendizaje automático los algoritmos de *K-Means* para “clustering” o los métodos de generación de árboles de decisión en problemas de clasificación hacen uso de procesos iterativos. Esta metodología aplicada sobre el plano complejo y sobre ciertas funciones complejas será la que nos proporcionará nuestros primeros ejemplos de imágenes fractales.

2.1. Iteración de funciones

Definición 2.1.1. Consideramos una función $f : \mathbb{C} \rightarrow \mathbb{C}$ y un punto $z \in \mathbb{C}$. La aplicación sucesiva de f a z – i.e., $z, f(z), f(f(z)), f(f(f(z))), \dots$ – produce las **iteradas** de la función f en el punto z . Al conjunto de dichas iteradas se le denomina **órbita** $O_f(z)$ de f en z .

$$O_f(z) := \{z, f(z), f^2(z), \dots, f^n(z), \dots\},$$

donde f^n denota a $f \circ f^{n-1}$.

Lo siguiente es plantearse la posible convergencia de la sucesión $\{f^n(z)\}$. Para ello, y a partir de este momento nos ayudaremos del software **Mathematica** en su versión 12 (concretamente la versión 12.1)¹. El comando `NestList[f, z, n]` itera una función `f`, comenzando en el punto `z` un total de `n` veces y devuelve una lista con los `n` valores.

Para intuir qué ocurre a largo plazo podemos iterar un número grande de veces, fijémonos lo que ocurre si utilizamos $f(z) := z^2$ comenzando por $z_0 = 0.9$.

¹Los códigos completos que generan las imágenes fractales creadas con Mathematica que se observan en este documento se pueden encontrar en <https://github.com/JAntonioVR/Geometria-Fractal/tree/main/Mathematica>.

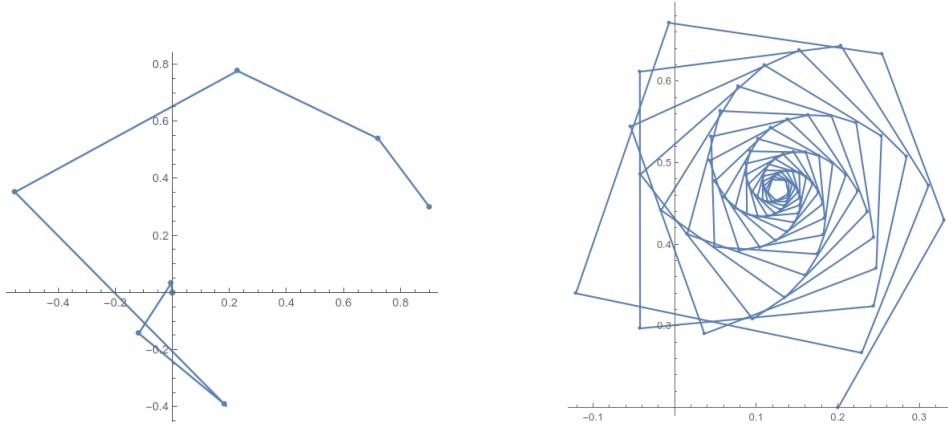
```

In[1]:= f[z_] := z^2;
NestList[f, 0.9, 10]

Out[1]= {0.9, 0.81, 0.6561, 0.430467, 0.185302, 0.0343368,
0.00117902, 1.39008*10^-6, 1.93233*10^-12, 3.73392*10^-24,
1.39421*10^-47}

```

Como se puede observar, en cada iteración se acerca cada vez más a 0, lo que parece indicar que $\{f^n(0.9)\} \rightarrow 0$. En este caso todos los valores eran reales, pero aprovechando la correspondencia de \mathbb{C} con \mathbb{R}^2 , de forma que un número complejo $z = x + y \cdot i \in \mathbb{C}$ se identifica con el par $(x, y) \in \mathbb{R}^2$, podemos representar en el plano, de forma visual, la tendencia de dichas sucesiones, uniendo con un segmento cada punto con su consecutivo. Observemos los ejemplos de las imágenes 2.1.



(a) $f(z) = z^2$ en $z_0 = 0.9 + 0.3i$ (b) $f(z) = z^2 + 0.33 + 0.35i$ en $z_0 = 0.2 + 0.2i$

Imagen 2.1: Representación de dos órbitas en \mathbb{C}

Los ejemplos expuestos hasta ahora son todos convergentes, pero esto no es siempre así. Nuestro objetivo ahora es poder conocer el comportamiento a largo plazo de la órbita de una función y un punto dados.

2.1.1. Convergencia a un punto fijo

Fijémonos que en el ejemplo anterior si en lugar de tomar $z_0 = 0.9$ hubiésemos tomado cualquier valor con $|z_0| < 1$ la convergencia habría sido igualmente a 0, pues elevamos al cuadrado cada vez números más pequeños. Justo al contrario ocurre si $|z_0| > 1$, en cuyo caso la sucesión diverge. Por último, en caso de que $|z_0| = 1$, esto es, $z_0 \in S^1$, la sucesión $\{f^n(z_0)\}$ nunca saldrá de S^1 , siendo este el conjunto que delimita la frontera entre el conjunto de puntos cuya sucesión converge o diverge. En particular, $f^n(-1) = f^n(1) = 1 \quad \forall n \in \mathbb{N}$, por lo que $z = 1$ es un punto fijo de f , es decir, $f(z) = z$.

Nos interesa particularmente saber qué sucesiones de iteradas convergen y a qué elemento convergen. En este sentido *Stephan Banach* demostró uno de los resultados más útiles y vigentes del análisis funcional:

Teorema 2.1.1 (Punto fijo de Banach). *Sea X un espacio métrico completo y $f : X \rightarrow X$ una aplicación contractiva. Entonces f tiene un único punto fijo. Además, la sucesión de iteradas $\{f^n(x_0)\}$ converge a dicho punto fijo para cualquier $x_0 \in X$.*

Demostración. Probaremos primero la existencia:

Sea $x_0 \in X$ y sea la sucesión de iteradas $\{x_n\} = \{f^n(x_0)\}$. Por ser f contractiva, llamamos $K < 1$ a su constante de Lipschitz. Probaremos por inducción que $d(x_n, x_{n+1}) \leq K^n d(x_0, x_1) \quad \forall n \in \mathbb{N}$. El caso base es cierto pues $d(x_1, x_2) = d(f(x_0), f(x_1)) \leq Kd(x_0, x_1)$. Si suponemos que la hipótesis es cierta para cierto n , tenemos que

$$\begin{aligned} d(x_{n+1}, x_{n+2}) &= d(f(x_n), f(x_{n+1})) \\ &\leq Kd(x_n, x_{n+1}) \\ &\leq K^{n+1}d(x_0, x_1). \end{aligned}$$

Ahora, para $n, r \in \mathbb{N}$:

$$\begin{aligned} d(x_n, x_{n+r}) &\leq \sum_{j=0}^{r-1} d(x_{n+j}, x_{n+j+1}) \\ &\leq d(x_1, x_0) \sum_{j=0}^{r-1} K^{n+j} \\ &\leq K^n d(x_0, x_1) \sum_{j=0}^{\infty} K^j \\ &= \frac{d(x_0, x_1)K^n}{1 - K}. \end{aligned}$$

Dado $\varepsilon > 0$, como $\{K^n\} \rightarrow 0$, existe un $m \in \mathbb{N}$ tal que para $n > m$ se tiene que $d(x_0, x_1)K^n < \varepsilon(1 - K)$. Si ahora tomamos dos naturales $p, q \geq m$ con $p < q$, aplicando la desigualdad anterior tomando $n = p$ y $r = q - p$ obtenemos que:

$$d(x_p, x_q) = d(x_n, x_{n+r}) \leq \frac{d(x_0, x_1)K^n}{1 - K} < \varepsilon,$$

de donde deducimos que $\{x_n\}$ es una sucesión de Cauchy, y como X es completo, tenemos que existe $x \in X$ tal que $\{x_{n+1}\} = \{f(x_n)\} = \{f^n(x_0)\} \rightarrow x$. Pero f es continua, por lo que como $\{x_n\} \rightarrow x$, necesariamente $\{f(x_n)\} \rightarrow f(x)$, por lo que $f(x) = x$.

Para probar la unicidad, suponemos que $y \in X$ es otro punto fijo de f , por lo que $d(x, y) = d(f(x), f(y)) \leq Kd(x, y)$. Tomando el primero y el tercer miembro de la desigualdad deducimos que $(1 - K)d(x, y) \leq 0$, luego $d(x, y) = 0$. \square

Este teorema unido a que sabemos que \mathbb{C} , y por tanto todos sus subconjuntos cerrados, es completo, nos permite asegurar convergencia de las sucesiones $f^n(x_0)$ a un punto fijo para cualquier x_0 siempre que dicha función sea contractiva.

2.1.2. Velocidad de convergencia

Así pues, el teorema del punto fijo de Banach (teorema 2.1.1), dice que las funciones contractivas en un espacio métrico completo admiten un único punto fijo, el cual se puede hallar iterando la función. Recordemos que dada una función $f : \mathbb{C} \rightarrow \mathbb{C}$ contractiva, esta cumple que $|f(z_1) - f(z_2)| \leq K|z_1 - z_2| \quad \forall z_1, z_2 \in \mathbb{C}$, siendo $0 < K < 1$ la constante de Lipschitz de f , nombrada en este caso la constante de contractividad. Fijémonos que, si iteramos n veces

la función f

$$\begin{aligned}|f^n(z_0) - f^{n-1}(z_0)| &\leq K|f^{n-1}(z_0) - f^{n-2}(z_0)| \\ &\leq K^2|f^{n-2}(z_0) - f^{n-3}(z_0)| \\ &\leq \dots \leq K^{n-1}|f(z_0) - z_0|,\end{aligned}$$

por lo que en realidad la constante K define de forma genérica la velocidad de convergencia en términos de la función. Cuanto más cercana a 0, más rápida será la convergencia al punto fijo. Sin embargo, vemos que realmente también depende de la condición inicial fijada, y es este el aspecto que explotaremos para obtener imágenes fractales.

En los casos que tengamos asegurada la convergencia, es interesante saber cuántas iteraciones son necesarias en cada punto para saber si hemos alcanzado el valor al que converge la sucesión. Este número de iteraciones se toma como aproximado, pues generalmente nunca se llega a alcanzar el punto fijo, tan solo podemos reducir la diferencia tanto como queramos. En este sentido, *Mathematica* tiene dos comandos útiles:

- `FixedPointList[f,expr]` genera una lista de valores resultantes de aplicar `f` a `expr` repetidamente hasta que los dos últimos valores no cambian. Para parametrizar la precisión en la proximidad de los valores podemos utilizar el argumento opcional `SameTest`².
- `FixedPoint[f,expr]` hace lo mismo pero produce como salida únicamente el valor último producido.

El siguiente código muestra un ejemplo de uso:

```
In[2]:= f[z_] = z/2 + 1/z;
FixedPointList[f, 0.75 + 0.1 I, SameTest -> (Abs[#1 - #2] < 10^-4 &)]
FixedPoint[f, 0.75 + 0.1 I]

Out[2]= {0.75 + 0.1 I, 1.68504 - 0.124672 I, 1.43275 - 0.0186668 I,
1.41421 - 0.000241452 I, 1.41421 - 2.45537*10^-10 I,
1.41421 + 3.57849*10^-18 I}

Out[3]= 1.41421 + 0. I
```

Y a partir de la longitud de la lista que nos devuelve `FixedPointList`, es decir, con el sencillo comando `Length[FixedPoint[f,x]]` podemos medir la velocidad de convergencia de cada punto.

2.2. El método de Newton y cuencas de atracción

Gracias a la iteración y al estudio de la convergencia de las sucesiones que definen las órbitas, podemos definir muchos algoritmos que nos ayuden a resolver numéricamente problemas que de forma teórica o analítica serían difíciles de abordar. Entre estos está el conocido **método de Newton**. Este es un procedimiento iterativo que nos permite aproximar con cierta precisión donde se anula una función derivable, pudiendo así utilizarlo como herramienta para la resolución

²Más información en la [documentación oficial](#).

de ecuaciones. Existen varias y distintas versiones del método de Newton, siendo la más sencilla la que busca las raíces de una función real de variable real $f : \mathbb{R} \rightarrow \mathbb{R}$, conociéndose también en este caso como *método de Newton-Raphson*. Además, hay una generalización para aplicaciones $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$, la cual se apoya en la inversa del operador diferencial. Sin embargo, nosotros estamos trabajando con funciones complejas $f : \mathbb{C} \rightarrow \mathbb{C}$, que aunque \mathbb{C} y \mathbb{R}^2 sean isomorfos, la dinámica es distinta, pues si la función compleja es suficientemente buena, podemos usar su derivada $f'(z)$ y no el operador diferencial (véase [9]).

Recordemos que dada una ecuación $f(z) = 0$ para cierta función compleja y analítica $f : \mathbb{C} \rightarrow \mathbb{C}$, el método de Newton itera la función

$$N_f(z) = z - \frac{f(z)}{f'(z)}, \quad (2.1)$$

comenzando por un punto z_0 cercano a la raíz. Es decir, calcula la sucesión $\{z_n\}$ definida como $z_{n+1} = N_f(z_n) \forall n \in \mathbb{N}$, la cual aplicando el teorema 2.1.1 podemos deducir que converge a un punto $a \in \mathbb{C}$ que verifica $f(a) = 0$ siempre que $f'(a) \neq 0$. Para mayor detalle acerca de la convergencia local ver [24, Capítulo 7] y [1, Sección 5.4].

Sin embargo, en muchas ecuaciones, comenzando por las polinómicas de grado mayor que 1, existen varias soluciones distintas, pero el método de Newton converge sólo a una de ellas, dependiendo de qué z_0 fijemos. Nuestro objetivo ahora se sitúa en discernir, para cada punto z_0 del plano, a qué solución de la ecuación $f(z) = 0$ converge la sucesión $\{z_n\}$ dada por el método de Newton utilizando a z_0 como semilla, es decir, como iteración inicial. En las siguientes secciones veremos algunos ejemplos de esta distinción y su utilidad, llegando a las primeras imágenes fractales generadas por iteración.

Ejemplo 2.2.1. Consideramos la función compleja $f : \mathbb{C} \rightarrow \mathbb{C}$ dada por $f(z) = z^2 - 1$, la cual tiene dos raíces: 1 y -1 , las dos raíces cuadradas de 1. Una forma sencilla de comprobar a qué raíz converge cada sucesión utilizando como semilla cada $z_0 \in \mathbb{C}$ es asociando un color a cada punto del plano dependiendo de la raíz a la que converja y pidiendo a *Mathematica* que coloree el plano complejo siguiendo este criterio.

```
In[4]:= iteracionN = #2 - #1[#2]/Derivative[1][#1][#2] &;
newtonArgumento =
Compile[{{z, _Complex}},
Arg[FixedPoint[iteracionN[f, #] &, z, 50]]];
DensityPlot[newtonArgumento[x + I*y], {x, -2, 2}, {y, -2, 2},
PlotPoints -> 100, Mesh -> False, ColorFunction -> "Rainbow"]
```

Producido como resultado la imagen 2.2. La forma de deducir a qué raíz converge cada sucesión consiste en evaluar los argumentos de los valores a los que se acerca la misma. Como podemos ver, y teniendo en cuenta que la imagen solo grafica el intervalo $[-2, 2] \times [-2, 2]$ y un número finito de puntos³ la sucesión cuya semilla es un punto perteneciente al semiplano abierto de la derecha converge a la raíz 1. Por otro lado, si la sucesión comienza con un complejo del semiplano abierto de la izquierda, entonces esta converge a la raíz -1 . Apoyándonos en este ejemplo, definimos el siguiente concepto.

³El número de puntos que se grafica se puede parametrizar con el argumento opcional “PlotPoints” de las funciones “Plot”. A mayor valor mayor calidad de imagen y resolución, pero mayor tiempo de ejecución.

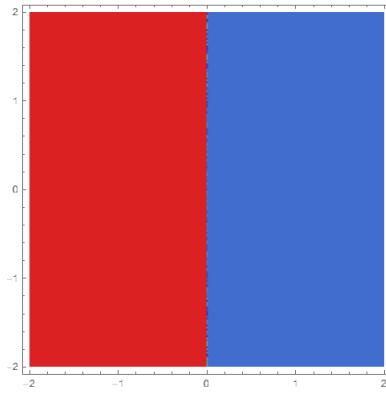


Imagen 2.2: Cuencas de atracción de $f(z) = z^2 - 1$.

Definición 2.2.1 (Cuenca de atracción). Definimos como **cuenca de atracción** de una raíz $a \in \mathbb{C}$ de una función compleja $f : \mathbb{C} \rightarrow \mathbb{C}$ (*i.e.* $f(a) = 0$), y denotamos como $A(a)$, al conjunto de puntos $z_0 \in \mathbb{C}$ tales que la sucesión $\{z_n\}$ dada por $z_{n+1} = N_f(z_n)$ converge a a utilizando a z_0 como primer término de la sucesión.

En muchas de las funciones que trataremos ocurre que existen distintas cuencas de atracción y la distinción sobre a cuál de ellas pertenece cada punto del plano complejo es la que nos permitirá graficar imágenes fractales. Si aplicamos el mismo procedimiento antes descrito en el ejemplo 2.2.1 con otras funciones encontramos las imágenes 2.3.

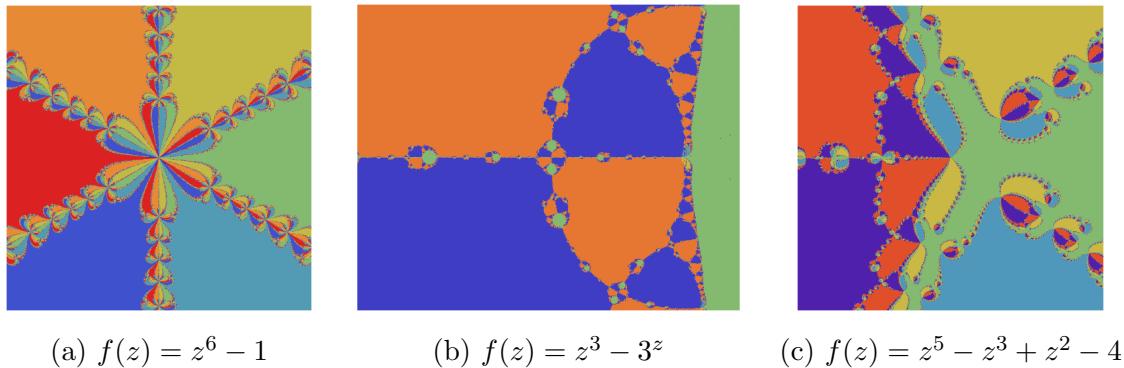


Imagen 2.3: Cuencas de atracción de distintas funciones coloreadas.

En estas imágenes podemos ahora sí ver los primeros ejemplos de estructuras fractales producidos por la iteración. Observemos que las sucesiones de dos puntos muy próximos pueden converger a raíces distintas, por lo que este es un ejemplo de inestabilidad o *caos matemático*: pequeñas variaciones en las condiciones iniciales conducen a comportamientos muy diferentes.

Por otro lado, en lugar de colorear el plano según la raíz a la que converja la sucesión también podemos fijarnos en la velocidad de convergencia. Esto es, asociar a cada punto del plano un color dependiendo del número de iteraciones necesarias para situarse a menos de cierta distancia de la raíz. En este sentido, podemos revisitar las imágenes 2.3 y en función de la velocidad de convergencia de las sucesiones utilizar un color distinto. El código necesario es el siguiente y los resultados se pueden observar en las imágenes 2.4:

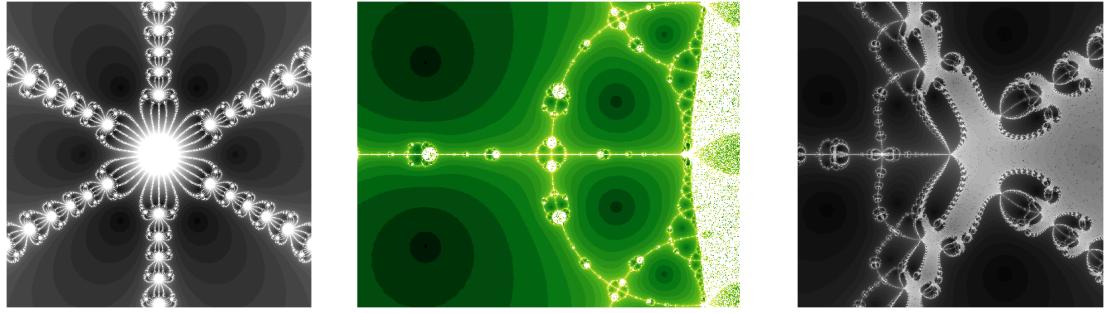
```
In[5]:= newtonVelocidad = Compile[{{z, _Complex}},
Length[FixedPointList[iteracionN[f, #] &, z, 50]]];
```

```

DensityPlot[newtonVelocidad[x + I*y], {x, -2, 2}, {y, -2, 2},
PlotPoints -> 200, Mesh -> False, ColorFunction -> GrayLevel]

```

donde se puede cambiar el valor de f , la región del plano $\{x, -2, 2\}$, $\{y, -2, 2\}$ o el valor del argumento `ColorFunction` para obtener distintas imágenes.



(a) $f(z) = z^6 - 1$ (b) $f(z) = z^3 - 3z$ (c) $f(z) = z^5 - z^3 + z^2 - 4$

Imagen 2.4: Evaluación de la velocidad de convergencia en cada punto

2.2.1. Autosimilaridad

Consideramos ahora la representación de las cuencas de atracción de la función compleja $f(z) = z^3 - 1$, que podemos ver en la imagen 2.5.

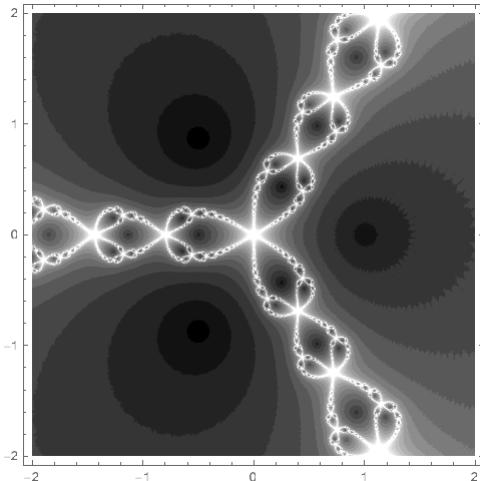


Imagen 2.5: Cuencas de atracción de $f(z) = z^3 - 1$

Fijémonos además que si hacemos *zoom* en ciertas partes de la imagen, es decir, representamos una región más pequeña, encontramos estructuras que son iguales independientemente del zoom que se aplique. En la figura 2.6 podemos ver distintos detalles de la figura 2.5, cada una es una ampliación de la anterior, y podemos observar como esa estructura se repite en todas las escalas.

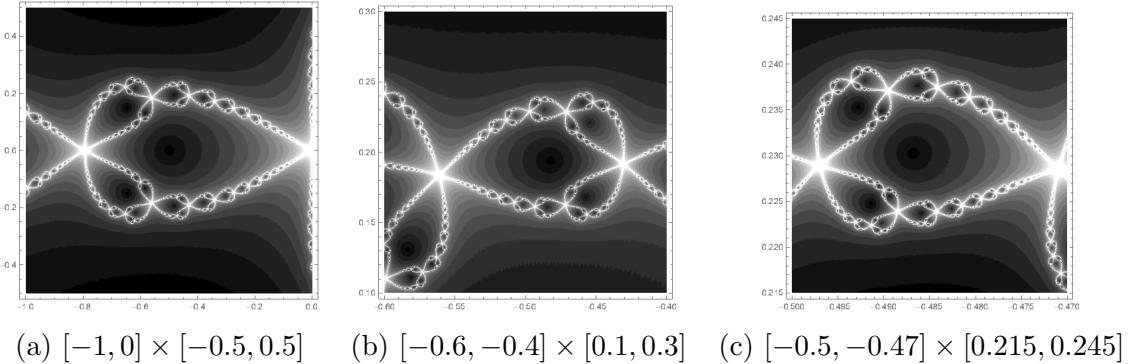


Imagen 2.6: Diferentes regiones ampliadas de la figura 2.5

Este aspecto es el que realmente nos define la naturaleza fractal de las cuencas de atracción, pues aunque las imágenes no son objetos totalmente autosimilares en el sentido de la definición 1.0.1 si que contienen regiones que sí lo son, como es el caso que acabamos de describir.

CAPÍTULO 3

CONJUNTOS DE JULIA Y MANDELBROT

Terminamos el capítulo 2 fijándonos en la autosimilaridad de las imágenes que nos proporcionaba la aplicación del método de Newton en ciertas funciones complejas para deducir las cuencas de atracción de las distintas funciones. Pudimos comprobar que a pesar de ser imágenes que no son totalmente autosimilares, por lo que no son fractales en el sentido estricto, contienen fragmentos que sí lo son. Este mismo hecho se produce en los conjuntos de Julia y en el conjunto de Mandelbrot. Podemos comprobarlo en las imágenes 3.1, que son nuestros primeros dos ejemplos de conjuntos de Julia.

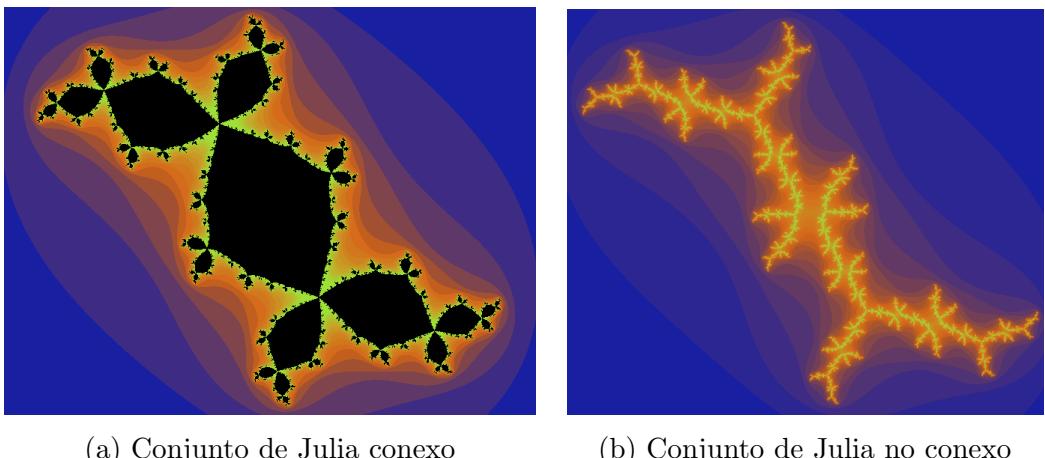
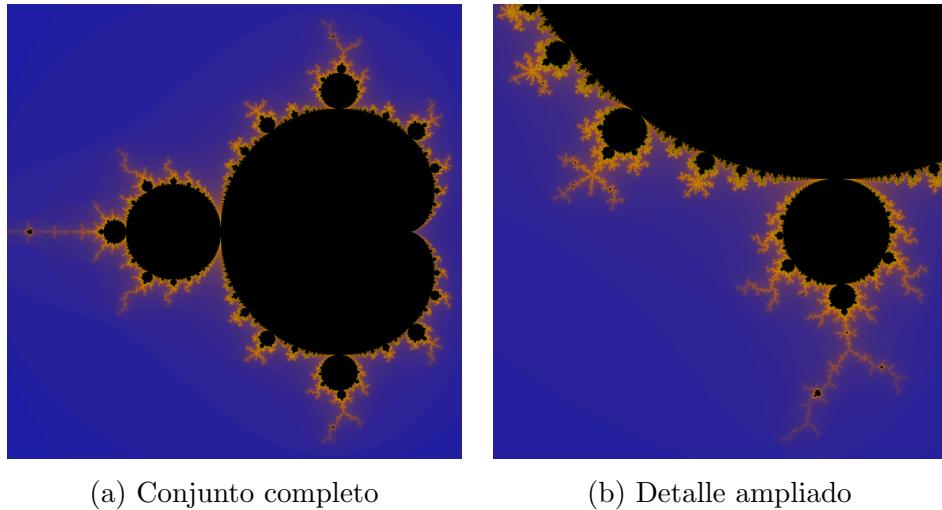


Imagen 3.1: Primeras imágenes de conjuntos de Julia

Una notable diferencia entre las imágenes 3.1 se encuentra en que mientras en la imagen (a) se puede percibir cierta conexión en el conjunto, esta desaparece en el caso de la imagen (b). Precisamente en esa distinción reside la génesis del conocido *conjunto de Mandelbrot*, que podemos también ver por primera vez, junto con algunas de sus autosimilaridades, en las imágenes 3.2.

En este capítulo aprenderemos qué elementos componen estos conjuntos, y cómo llegar a visualizar estas imágenes tan llamativas.



(a) Conjunto completo

(b) Detalle ampliado

Imagen 3.2: Primeras imágenes del conjunto de Mandelbrot

3.1. Iteración convergente y no convergente

Recordamos que en el capítulo 2, a partir de una función analítica $f : \mathbb{C} \rightarrow \mathbb{C}$, se le aplicaba una transformación $N_f(z)$ de forma que en muchos casos la iteración de dicha función era convergente independientemente del término $z_0 \in \mathbb{C}$ inicial. Sin embargo recordemos que la iteración de una función cualquiera f no siempre es convergente, como pudimos comprobar en el ejemplo situado al comienzo de la sección 2.1.1, en el que recordamos que las iteradas de la función $f(z) = z^2$ divergen siempre que $|z_0| > 1$, convergen a 0 si $|z_0| < 1$ y quedan encerradas en S^1 en caso de que $|z_0| = 1$. Otro posible comportamiento es el cíclico, como el que tienen las iteradas de la función $g(z) = z \cdot i$ en $z_0 = 1$, que si nos fijamos, son $O_g(1) = \{1, i, -1, -i, 1, \dots\}$.

Un último caso de posible comportamiento de una órbita son las órbitas caóticas, en las cuales no se percibe ningún patrón y además es muy sensible a las condiciones iniciales, fijémonos en lo que ocurre en el caso de la función $h(z) = z^2 - 1.9$ si miramos sus órbitas en $z_0 = 0, 0.1$:

```
In[6]:= h[z_] := z^2 - 1.9;
NestList[h, 0.1, 20]
NestList[h, 0.0, 20]

Out[6]= {0.1, -1.89, 1.6721, 0.895918, -1.09733, -0.695866,
-1.41577, 0.104404, -1.8891, 1.6687, 0.884552, -1.11757,
-0.651043, -1.47614, 0.279, -1.82216, 1.42026, 0.117148,
-1.88628, 1.65804, 0.849092}

Out[7]= {0., -1.9, 1.71, 1.0241, -0.851219, -1.17543, -0.518374,
-1.63129, 0.761102, -1.32072, -0.155688, -1.87576, 1.61848,
0.719477, -1.38235, 0.0109007, -1.89988, 1.70955, 1.02256,
-0.854379, -1.17004}
```

No se observa ningún patrón de convergencia y además, a pesar de ser semillas muy cercanas, las órbitas son muy diferentes.

La dicotomía existente entre qué z_0 iniciales hacen que las iteradas de una función converja, o no, restringida a cierta familia de funciones, es la que define a los distintos conjuntos de Julia.

Presentamos por tanto, para cada $c \in \mathbb{C}$, la familia de funciones

$$P_c(z) = z^2 + c \quad \forall z \in \mathbb{C}. \quad (3.1)$$

Nuestro objetivo es entonces clasificar para qué $z_0 \in \mathbb{C}$, las iteradas $\{P_c^n(z_0)\}$ convergen, divergen, ciclan, o tienen posiblemente un comportamiento caótico.

3.2. Conjuntos de Julia

Sin dejar de tener en cuenta la familia de funciones $\{P_c(z)\}_{c \in \mathbb{C}}$ introducimos la siguiente definición.

Definición 3.2.1. Dado un número complejo $c \in \mathbb{C}$ fijo consideramos $P_c(z) = z^2 + c$. Entonces:

- Se denomina **conjunto de puntos de escape**, y denotamos como E_c , al conjunto de puntos cuyas iteradas divergen, es decir:

$$E_c = \{z_0 \in \mathbb{C} : \{|P_c^n(z_0)|\} \rightarrow \infty\}$$

- Se denomina **conjunto de puntos prisioneros**, y denotamos como P_c al conjunto de puntos cuyas iteradas no divergen, por lo que es el complemento de E_c .

A partir de estas dos definiciones, que insisten en clasificar exhaustivamente los puntos del plano complejo entre de escape o prisioneros según su órbita, podemos introducir la definición que esperábamos.

Definición 3.2.2 (Conjunto de Julia). Dado un número $c \in \mathbb{C}$, se define su **conjunto de Julia**, y se denota como \mathcal{J}_c , a la frontera de E_c . Se denomina **conjunto de Fatou** al complemento del conjunto de Julia.

Es precisamente en los complejos que pertenecen al conjunto de Julia donde las iteradas tienen un comportamiento caótico.

Ejemplo 3.2.1. En el caso $c = 0$, es decir, $P_0(z) = z^2$, sabemos ya que $\mathcal{J}_c = S^1$, pues precisamente es S^1 la frontera entre los puntos cuyas iteradas divergen o convergen a 0.

Observación 3.2.1. Fijémonos por tanto que hay tantos conjuntos de Julia como números complejos, al poder asociar a cada número complejo un conjunto de puntos prisioneros, de escape, y por tanto un conjunto de Julia.

3.2.1. Representación gráfica de los conjuntos de Julia

Tenemos entonces una definición de los conjuntos de Julia, pero aparentemente está muy alejada de las imágenes 3.1 que presentamos en la introducción. La forma de llegar a ellas es similar a la que utilizamos para graficar las imágenes que emplean la velocidad de convergencia en el capítulo 2. Sin embargo, y al contrario que al utilizar el método de Newton, ahora no tenemos ningún tipo de convergencia asegurada, por lo que el método de aplicar iteradas hasta encontrar un patrón no es el más correcto. Debemos por tanto encontrar una manera eficiente de clasificar cada punto del plano como prisionero o de escape.

Para ello podemos fijarnos en que la operación de elevar z_n al cuadrado prima sobre la de sumar una constante c siempre que el módulo $|z_n|$ sea ‘suficientemente grande’. Procedemos entonces a enunciar el siguiente resultado:

Teorema 3.2.1. Dado un $c \in \mathbb{C}$, consideramos la función $P_c(z) = z^2 + c$. Si un número $z_0 \in \mathbb{C}$ verifica que $|z_0| > \max\{|c|, 2\}$, entonces z_0 es un punto de escape. Al número $e_c = \max\{|c|, 2\}$ se le denomina número de escape.

Demostración. Supongamos que $|z_0| > e_c = \max\{|c|, 2\}$, por tanto necesariamente debe existir un número $\varepsilon > 0$ tal que $|z_0| = 2 + \varepsilon$. Aplicamos entonces la cadena de desigualdades siguiente:

$$\begin{aligned} |P_c(z_0)| = |z_0^2 + c| &\geq |z_0^2| - |c| \text{ (propiedades del módulo)} \\ &= |z_0|^2 - |c| \\ &\geq |z_0|^2 - |z_0| \text{ (porque } |z_0| > |c|\text{)} \\ &= (|z_0| - 1)|z_0| \\ &= (1 + \varepsilon)|z_0|. \end{aligned}$$

Por tanto tenemos que $|P_c(z_0)| \geq (1 + \varepsilon)|z_0|$, por lo que en cada iteración el módulo aumenta al menos $1 + \varepsilon$ unidades, que es mayor que uno, es decir, $|P_c^k(z_0)| \geq (1 + \varepsilon)^k|z_0|$, por lo que la sucesión diverge y z_0 es un punto de escape. \square

Podemos aplicar este teorema para programar un algoritmo que grafique conjuntos de Julia. Para ello, fijamos un número $M \in \mathbb{N}$ que será el máximo de iteraciones que se aplicarán en cada punto antes de decidir si el punto es prisionero o de escape. Si pasadas esas M iteraciones el módulo de z_0 no ha alcanzado el número de escape e_c entonces consideramos que z_0 es un punto prisionero. En caso contrario, en el momento que se alcance el número de escape cesarán las iteraciones y se etiquetará el punto como prisionero. El valor de M puede ser alto si queremos aumentar la precisión a cambio de mayor tiempo de cómputo, y viceversa. Es posible que algunos de los puntos sean de escape pero alcancen e_c después de las M iteraciones, pero tomando un valor suficientemente alto el resultado es prácticamente el mismo.

Para graficar un conjunto de Julia podemos asignar un color fijo a los puntos prisioneros y a los puntos de escape asignarle otro en función de las iteraciones necesarias antes de alcanzar el número de escape. El pseudocódigo del algoritmo descrito para graficar conjuntos de Julia sería el que podemos ver en el algoritmo 1.

Por ejemplo, si queremos representar en *Mathematica* el conjunto de la figura 3.1 (a), que era $\mathcal{J}_{-0.12+0.75i}$ utilizaríamos el siguiente código:

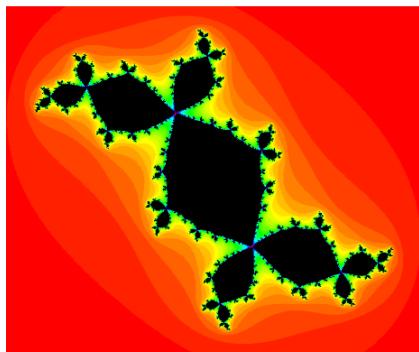
```
In[8]:= M = 50;
Julia[z_, c_] := Length[FixedPointList[#^2 + c &, z, M,
SameTest -> (Abs[#] > Max[2.0, Abs[c]] &)]];
DensityPlot[
Julia[x + I y, -0.12 + 0.75 I], {x, -1.5, 1.5},
{y, -1.25, 1.25}, PlotPoints->200, AspectRatio->Automatic,
ColorFunction -> (If[# >= 1, RGBColor[0, 0, 0], Hue[#]] &)]
```

Fijémonos en que hemos utilizado el argumento `SameTest` para especificar cuando dejar de iterar, especificando que se haga cuando se alcance e_c . Hemos utilizado también $M = 50$ como máximo número de iteraciones. Podemos variar el segundo argumento de la llamada a la función `Julia`, el rango de valores, el argumento `PlotPoints` o el número máximo de iteraciones M .

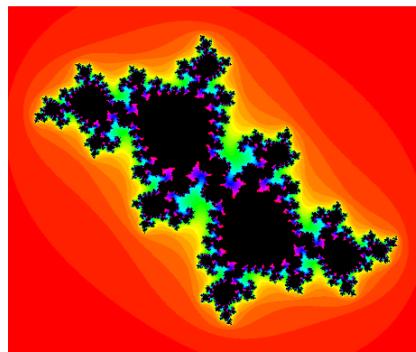
Algoritmo 1 Conjuntos de Julia

Para cada $z_0 \in \mathbb{C}$:

```
i ← 0
p ← z0
while i < M do
    p ← Pc(p)
    if |p| > ec = máx{|c|, 2} then
        break;
    end if
    i ++
end while
if i = M then
    z0 es un punto prisionero
    return color de puntos prisioneros
else
    z0 es un punto de escape
    return color(i)
end if
```



(a) $J_{-0.12+0.75i}$



(b) $J_{-0.23+0.65i}$

Imagen 3.3: Conjuntos de Julia graficados con Mathematica

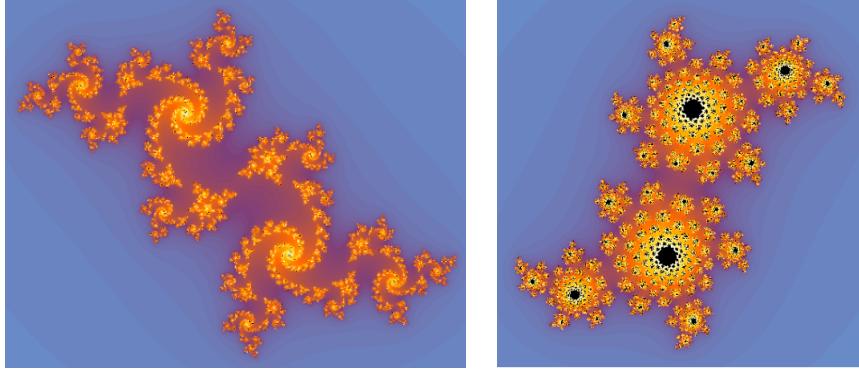
para graficar distintos conjuntos y detalles de los mismos. Algunos ejemplos de resultados de estos códigos se encuentran en la imagen 3.3.

Por sí solo, Mathematica incluye una función `JuliaSetPlot[c]` que muestra una imagen del conjunto \mathcal{J}_c . Esta función permite modificar los mismos parámetros que el método que acabamos de programar por nuestra cuenta¹, véanse las imágenes 3.4.

3.3. Distinción entre conjuntos de Julia conexos y polvaredas

Ya en la introducción de este capítulo y tras presentar las imágenes 3.1 prestamos atención a un detalle que, ahora que tenemos más ejemplos e imágenes de distintos conjuntos de Julia, se vuelve más evidente. Mientras algunos conjuntos de Julia se presentan conexos, como los de las

¹Consúltese la documentación oficial de esta función en <https://reference.wolfram.com/language/ref/JuliaSetPlot.html?q=JuliaSetPlot> si se desea informar detalladamente de los parámetros modificables.



(a) $J_{-0.23+0.69i}$

(b) $J_{0.16-0.59i}$

Imagen 3.4: Resultados de la orden ‘JuliaSetPlot’

imágenes 3.3, otros parecen no ser conexos y estar formados por trozos más pequeños, como los de las imágenes 3.4. Cabe recordar que en realidad los conjuntos de Julia son la frontera entre los puntos que se representan en negro y los que se representan de colores.

Para saber qué conjuntos de Julia son conexos y cuales no *Gaston Julia* y *Pierre Fatou* demostraron en 1918 el siguiente teorema, cuya prueba se puede encontrar en [22, Theorem 9.5].

Teorema 3.3.1 (Teorema de Fatou-Julia). *Si \mathbb{P}_c contiene todos los puntos críticos de $P_c(z)$, entonces $\mathcal{J}_c = \partial\mathbb{P}_c$ es conexo. Si al menos un punto crítico de $P_c(z)$ pertenece a \mathbb{E}_c , entonces \mathcal{J}_c es isomorfo al conjunto de Cantor, es decir, tiene un conjunto no numerable de componentes conexas.*

En este último caso el conjunto de Julia se conoce como «polvo de Fatou». En realidad el teorema en su versión original está enunciado para polinomios de grado mayor o igual que 2, entendiendo las extensiones naturales del conjunto de puntos de escape y prisioneros. Si nos restringimos entonces a la familia $\{P_c(z)\}_{c \in \mathbb{C}}$, es muy sencillo caracterizar qué conjuntos de Julia son conexos y cuales son polvaredas.

Corolario 3.3.1. *Dado $c \in \mathbb{C}$, el conjunto de Julia \mathcal{J}_c es conexo (resp. polvareda) si, y solo si, la sucesión de iteradas $\{P_c^n(0)\}$ no diverge (resp. diverge), es decir, si $0 \in \mathbb{P}_c$ (resp. $0 \in \mathbb{E}_c$).*

Demostración. Es claro que $P'_c(z) = (z^2 + c)' = 2z$, por lo que los puntos críticos de $P_c(z)$ se alcanzan cuando $z = 0$, por lo que aplicando el teorema 3.3.1 tenemos el resultado. \square

Sobre la dicotomía entre qué conjuntos de Julia son conexos y cuales son polvareda surge el conocido **conjunto de Mandelbrot**, el cual adelantamos que está formado por los números complejos c tales que su conjunto de Julia \mathcal{J}_c es conexo.

3.4. El conjunto de Mandelbrot

Como ya veníamos anunciando al final de la sección 3.3, el conjunto de Mandelbrot está formado por los $c \in \mathbb{C}$ tales que \mathcal{J}_c es conexo. La idea inicial de *Benoit Mandelbrot* para graficar el conjunto que denotaremos a partir de ahora como \mathcal{M} fue pintar de negro los puntos del plano cuyo conjunto de Julia fuese conexo y de blanco el resto. De entrada parece una tarea titánica graficar, para cada punto del plano (aunque realmente sería solo un subconjunto suficientemente

representativo), su conjunto de Julia y decidir si éste es o no conexo, pues en algunos casos la decisión se torna muy complicada.

Afortunadamente, gracias al teorema 3.3.1 y al corolario 3.3.1 esta tarea se vuelve mucho más sencilla, tan solo habría que tomar las iteradas en el origen, es decir

$$\{P_c^n(0)\} = \{c, c^2 + c, (c^2 + c)^2 + c, \dots\}$$

y decidir si la sucesión diverge o no mediante algún método similar al utilizado para graficar conjuntos de Julia.

Resumiendo, de momento sabemos que:

$$\begin{aligned}\mathcal{M} &= \{c \in \mathbb{C} : \mathcal{J}_c \text{ es conexo}\} \\ &= \{c \in \mathbb{C} : 0 \in P_c\} \\ &= \{c \in \mathbb{C} : \{P_c^n(0)\} \not\rightarrow \infty\}\end{aligned}$$

3.4.1. Representación gráfica del conjunto de Mandelbrot

Buscamos ahora obtener alguna figura similar a la imagen 3.2 a partir del conocimiento que tenemos de \mathcal{M} . Como ya viene siendo costumbre, la idea es dividir el plano en una cantidad finita de píxeles, asignando a cada uno un número complejo c y evaluar en cada uno si la órbita en $z = 0$ converge o diverge. Para tomar esta decisión nos podemos apoyar en el teorema 3.2.1 para probar este resultado.

Proposición 3.4.1. Dado un número complejo $c \in \mathbb{C}$, si $|c| > 2$ entonces la sucesión de iteradas $\{P_c^n(0)\}$ es divergente.

Demostración. Consideramos la sucesión de iteradas $z_0 = 0, z_n = P_c^n(0)$. Partimos de que $|c| > 2$, por lo que buscamos encontrar un $m \in \mathbb{N}$ tal que $|z_m| > e_c = \max\{|c|, 2\} = |c|$.

Tenemos que $z_1 = P_c(0) = c$, pero $|z_1| = |c| \not> |c|$.

Si iteramos una vez más, $z_2 = P_c^2(0) = P_c(c) = c^2 + c$. Ayudándonos de una propiedad del módulo tenemos que:

$$\begin{aligned}|c^2 + c| &\geq ||c^2| - |c|| \\ &= ||c|^2 - |c|| \\ &= |c|^2 - |c| \quad (\text{Pues al ser } |c| > 2, |c|^2 > |c|) \\ &= (|c| - 1)|c|.\end{aligned}$$

Y como $|c| > 2 \Leftrightarrow |c| - 1 > 1$, concluimos que

$$|z_2| = |c^2 + c| > |c| = \max\{|c|, 2\} = e_c.$$

Por lo que aplicando el teorema 3.2.1, la sucesión $\{z_n\}$ es divergente. \square

Este resultado nos facilita mucho la elaboración de un algoritmo que grafique a \mathcal{M} , pues de entrada nos afirma que todo número complejo cuyo módulo sea superior a 2 no pertenece a \mathcal{M} . O dicho de otra forma,

$$\mathcal{M} \subseteq \{c \in \mathbb{C} : |c| \leq 2\} = \bar{D}(0, 2),$$

donde $D(z, r)$ denota el disco abierto de centro z y radio r mientras que $\bar{D}(z, r)$ denota el cierre del disco abierto, es decir, el disco cerrado de centro z y radio r .

Además, en el momento que una iterada se sitúe fuera de $\bar{D}(0, 2)$, podemos asegurar que esa sucesión va a diverger, por lo que podemos dejar de iterar. Para concluir que la sucesión no diverge, al igual que para graficar conjuntos de Julia, fijamos un número máximo de iteraciones $M \in \mathbb{N}$, de forma que si al calcular la M -ésima iterada el módulo del elemento $P_c^M(0)$ no ha excedido a 2, podemos considerar que el número c pertenece a \mathcal{M} . En conclusión, el algoritmo consiste en, para cada píxel identificado con un punto del plano complejo $c \in \mathbb{C}$, calcular sus iteradas hasta un máximo de M iteraciones, en caso de exceder en módulo a 2 guardamos el mínimo valor $m \in \mathbb{N}$ tal que $|P_c^m(0)| > 2$ y asignamos un color en función; en caso de no exceder a 2 asignar un color fijo. Presentamos por tanto en el algoritmo 2 el procedimiento para graficar el conjunto de Mandelbrot.

Algoritmo 2 Conjunto de Mandelbrot

Para cada $c \in \mathbb{C}$:

```

 $i \leftarrow 0$ 
 $p \leftarrow c$ 
while  $i < M$  do
     $p \leftarrow P_c(p)$ 
    if  $|p| > 2$  then
        break;
    end if
     $i++$ 
end while
if  $i = M$  then
     $c \in \mathcal{M}$ 
    return color de puntos de  $\mathcal{M}$ 
else
     $c \notin \mathcal{M}$ 
    return color( $i$ )
end if
```

El código en *Mathematica* utilizado es por tanto el siguiente, muy similar al utilizado para graficar conjuntos de Julia:

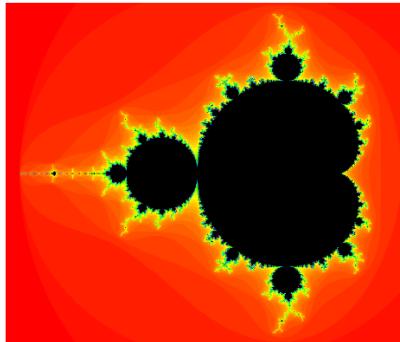
```

In[9]:= M = 100;
Mandelbrot = Compile[{{c, _Complex}},
  Length[FixedPointList[#^2 + c &, 0, M,
  SameTest -> (Abs[#] > 2.0 &)]]];

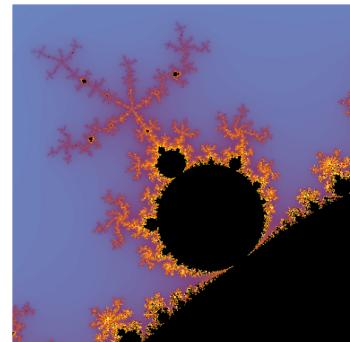
DensityPlot[Mandelbrot[x + I y], {x, -2.1, .7}, {y, -1.2, 1.2},
 Mesh -> False, Frame -> False, PlotPoints -> 200,
 AspectRatio -> Automatic,
 ColorFunction -> (If[# >= 1, Hue[0, 0, 0], Hue[#]] &)]
```

Al igual que en el caso de los conjuntos de Julia, *Mathematica* tiene una función preprogramada que grafica el conjunto de Mandelbrot, llamada `MandelbrotSetPlot`, la cual

admite varios argumentos opcionales, como la región a graficar, el máximo de iteraciones, resolución, etc.².



(a) Salida del código *Mathematica*



(b) Salida de la orden ‘MandelbrotSetPlot’

Imagen 3.5: Representación de \mathcal{M} y detalle en $[-0.65, -0.4] \times [0.47, 0.72]$

El conjunto de Mandelbrot es considerado por muchos como el objeto más complejo de la matemática. Además de la belleza que muestra por sí solo, los detalles de \mathcal{M} esconden bonitas estructuras: bulbos, valles, antenas, copias reducidas del propio \mathcal{M} ...

3.5. Autosimilaridad de los conjuntos de Julia y Mandelbrot

Ya en el comienzo de este capítulo mencionamos, y en las propias imágenes presentadas se puede comprobar, que los conjuntos de Julia y el conjunto de Mandelbrot no son objetos autosimilares, no al menos en el sentido de la definición 1.0.1. Sin embargo, sí contienen regiones y detalles que son autosimilares. En esta sección presentaremos algunas de las mismas.

3.5.1. Autosimilaridad en conjuntos de Julia

Recordemos el conjunto de Julia $\mathcal{J}_{-0.23+0.65}$, que podemos ver en la imagen 3.3 (b). En las imágenes 3.6 observamos distintos detalles, de forma que la primera es una ampliación de la original y las siguientes son cada una un ‘zoom’ de la anterior.

Obsérvese cómo efectivamente las imágenes, salvo giro, son prácticamente iguales, pudiendo ver así una de las regiones autosimilares de $\mathcal{J}_{-0.23+0.65}$. En las imágenes 3.7 podemos ver algunas regiones ampliadas de ciertos conjuntos de Julia, pudiendo observar regiones autosimilares.

²Consúltese la documentación oficial de esta función en <https://reference.wolfram.com/language/ref/MandelbrotSetPlot.html> si se desea informar detalladamente de los parámetros modificables.

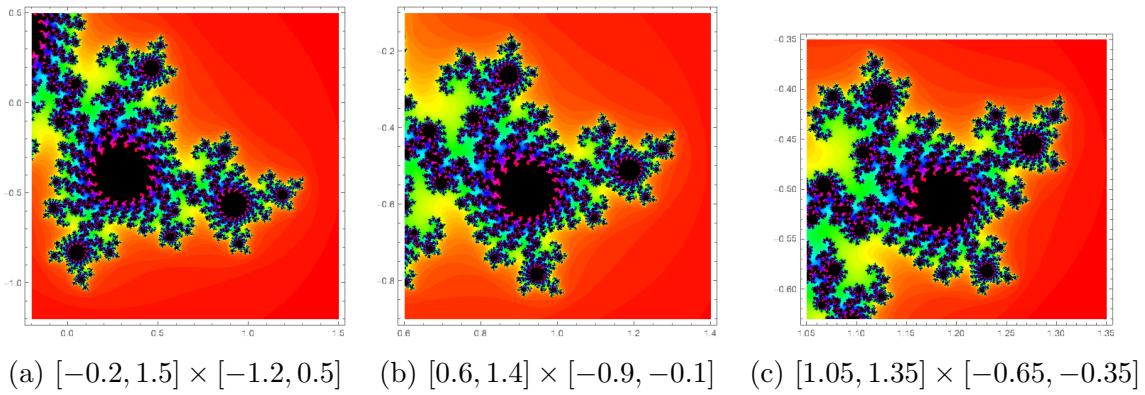
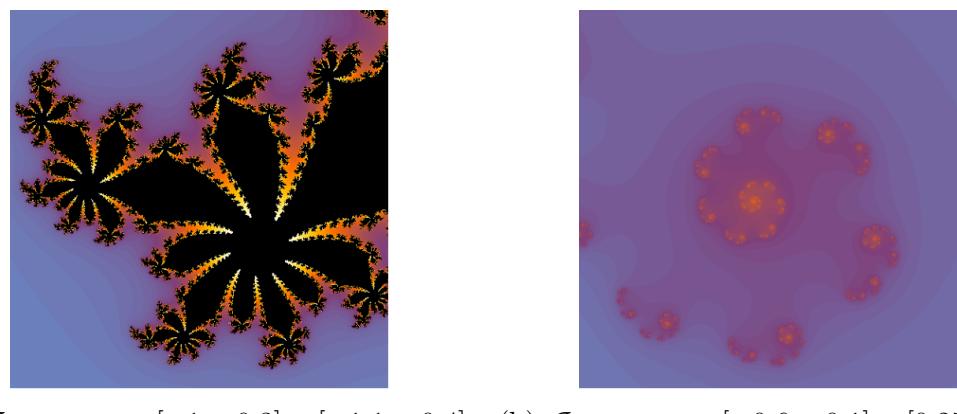


Imagen 3.6: Diferentes regiones ampliadas de la figura 3.3 (b)



(a) $\mathcal{J}_{0.33-0.41i}$ en $[-1, -0.3] \times [-1.1, -0.4]$ (b) $\mathcal{J}_{0.48-0.13i}$ en $[-0.6, -0.1] \times [0.25, 0.75]$



(c) $\mathcal{J}_{0.23+0.51i}$ en $[-1.1, -0.4] \times [0.4, 1.1]$ (d) $\mathcal{J}_{-0.52+0.51i}$ en $[0, 1.5] \times [-1, 0.5]$

Imagen 3.7: Detalles autosimilares de algunos conjuntos de Julia

3.5.2. Autosimilaridad en el conjunto de Mandelbrot

El conjunto de Mandelbrot, el cual ya conocemos, está compuesto fundamentalmente de un cuerpo principal con forma de cardioide con gran cantidad de bulbos adosados al mismo (imagen 3.8 (a), (b) y (d)), siendo cada uno de estos autosimilar y terminando en una antena que se bifurca (imagen 3.8 (b) y (c)). Además, en el propio \mathcal{M} hay minúsculas copias de sí mismo, en las cuales se vuelve a repetir su estructura (imagen 3.8 (c)).

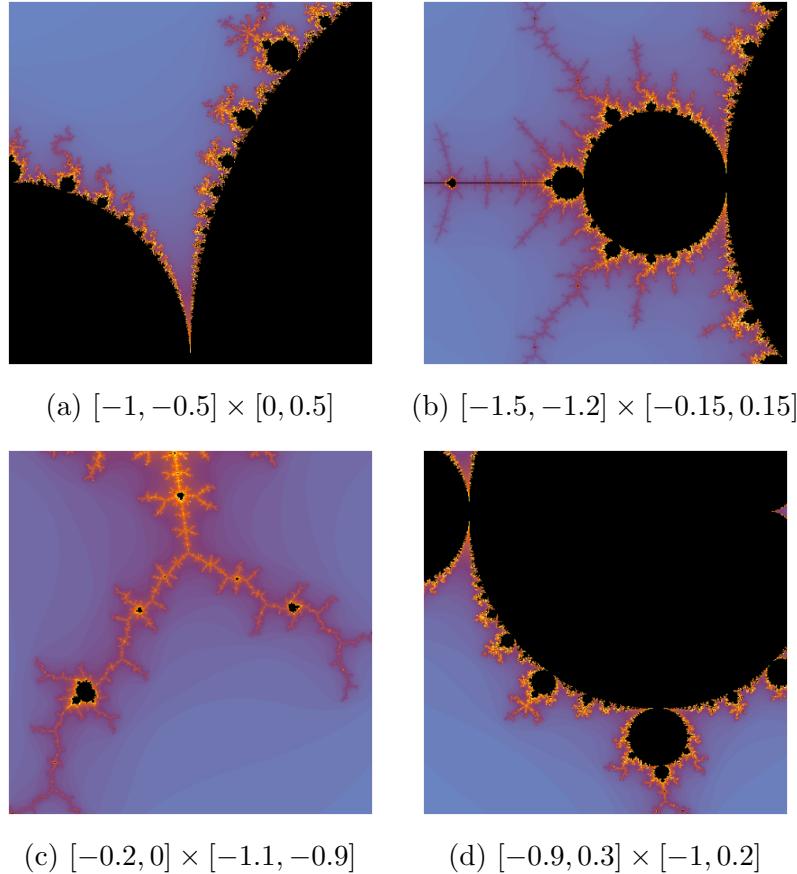


Imagen 3.8: Detalles autosimilares de \mathcal{M}

En la imagen 3.8 (b) podemos ver una representación del que se conoce como ‘bulbo principal’, pues es el más grande de todos los que están pegados al cuerpo principal. Podemos comprobar, observando los detalles que ofrecen las imágenes 3.9, que dicho bulbo muestra autosimilaridad.

3.6. Conjuntos de Julia y Mandelbrot generalizados

Hasta el momento todo lo explicado y todas las imágenes obtenidas se han basado en la familia de funciones $\{P_c(z)\}_{c \in \mathbb{C}} = \{z^2 + c\}_{c \in \mathbb{C}}$. Sin embargo, y como cabe esperar, la definición de los conjuntos de Julia como conjuntos frontera entre el conjunto de puntos prisioneros y de escape es completamente válido para las iteradas de cualquier función, o mejor dicho, para cualquier familia de funciones. En esta sección trataremos de extender los conocimientos obtenidos hasta ahora a otras funciones.

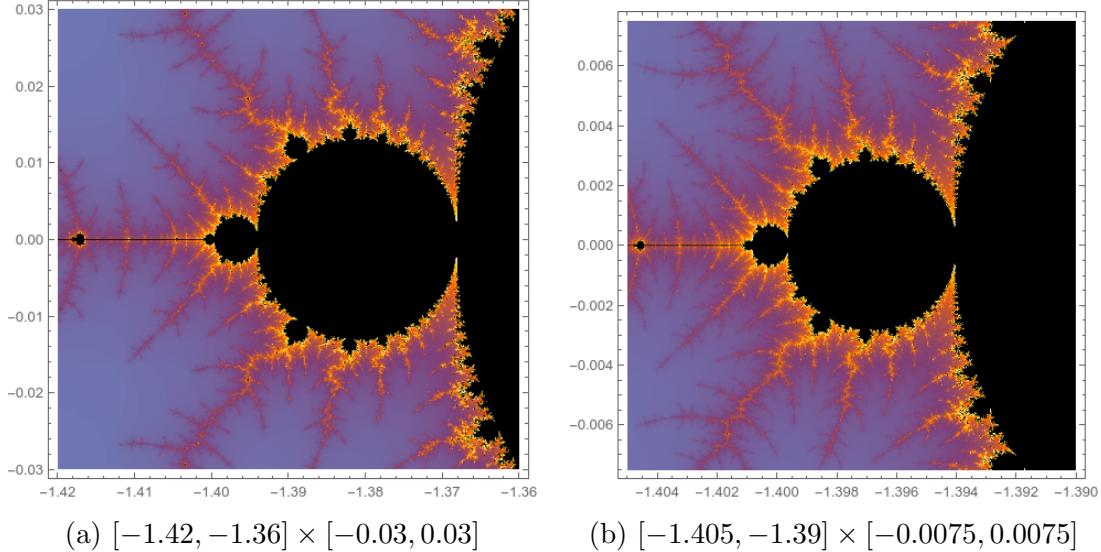


Imagen 3.9: Ampliaciones del bulbo principal de \mathcal{M}

3.6.1. Familia $\{z^m + c\}_{c \in \mathbb{C}}$

La extensión más natural y la que nos proporcionará resultados más llamativos consiste en cambiar el exponente de la función polinómica $P_c(z)$, a la cual ahora denotaremos como $P_{c,m}(z) = z^m + c$, con $m \geq 2$ natural para enfatizar el exponente. Es fácil comprobar que el teorema 3.2.1 es igual de válido con esta familia de funciones independientemente del exponente, por lo que el algoritmo sigue siendo el mismo, solo que en la función **Julia** debemos cambiar el exponente. Además, la orden **JuliaSetPlot** ya presentada en la sección 3.2.1 admite la posibilidad de indicarle una función arbitraria. Es esta la forma en la que hemos graficado las imágenes 3.10.

Fijémonos en que hemos fijado $c = -0.55 + 0.48i$ y hemos variado el exponente de la función polinómica. Llama la atención lo distintos que son los conjuntos entre sí cuando realmente el c fijado es el mismo en todos los casos.

Como ya comentamos en la sección 3.3, el teorema 3.3.1 (teorema de la dicotomía de Fatou-Julia) es válido para cualquier polinomio de grado mayor o igual que 2, en particular es válido para la familia de funciones $\{P_{c,m}(z)\}_{c \in \mathbb{C}}$. Tenemos por tanto que de la misma forma es válido el corolario, por lo que la distinción entre conjuntos de Julia conexos y polvaredas se vuelve a basar en buscar la convergencia o divergencia de la sucesión $\{P_{c,m}^n(0)\}$.

De esta forma, podemos hablar por tanto de conjuntos de Mandelbrot generalizados como representación gráfica de qué puntos tienen un conjunto de Julia conexo o polvareda utilizando la función $P_{c,m}(z)$, el cual denotamos como \mathcal{M}_m . De la misma forma que el resultado 3.2.1 es válido en la generalización, la proposición 3.4.1 también lo es, por lo que podemos también reutilizar el código que empleamos para representar el conjunto de Mandelbrot con tan solo variar el exponente, ver imágenes 3.11. Por su parte, la función de *Mathematica* **MandelbrotSetPlot** admite un argumento **n** que representa el exponente de $P_{c,m}(z)$. a la hora de iterar.

Obsérvese que a pesar de encontrar cierta similaridad en la formas de los distintos conjuntos de Mandelbrot generalizados, podemos determinar puntos que para unos exponentes se encuentran dentro y en otros casos fuera de su respectivo \mathcal{M}_m . Este hecho explica lo ocurrido con $c = -0.55 + 0.48i$, que para ciertos exponentes el conjunto de Julia es conexo y para otros

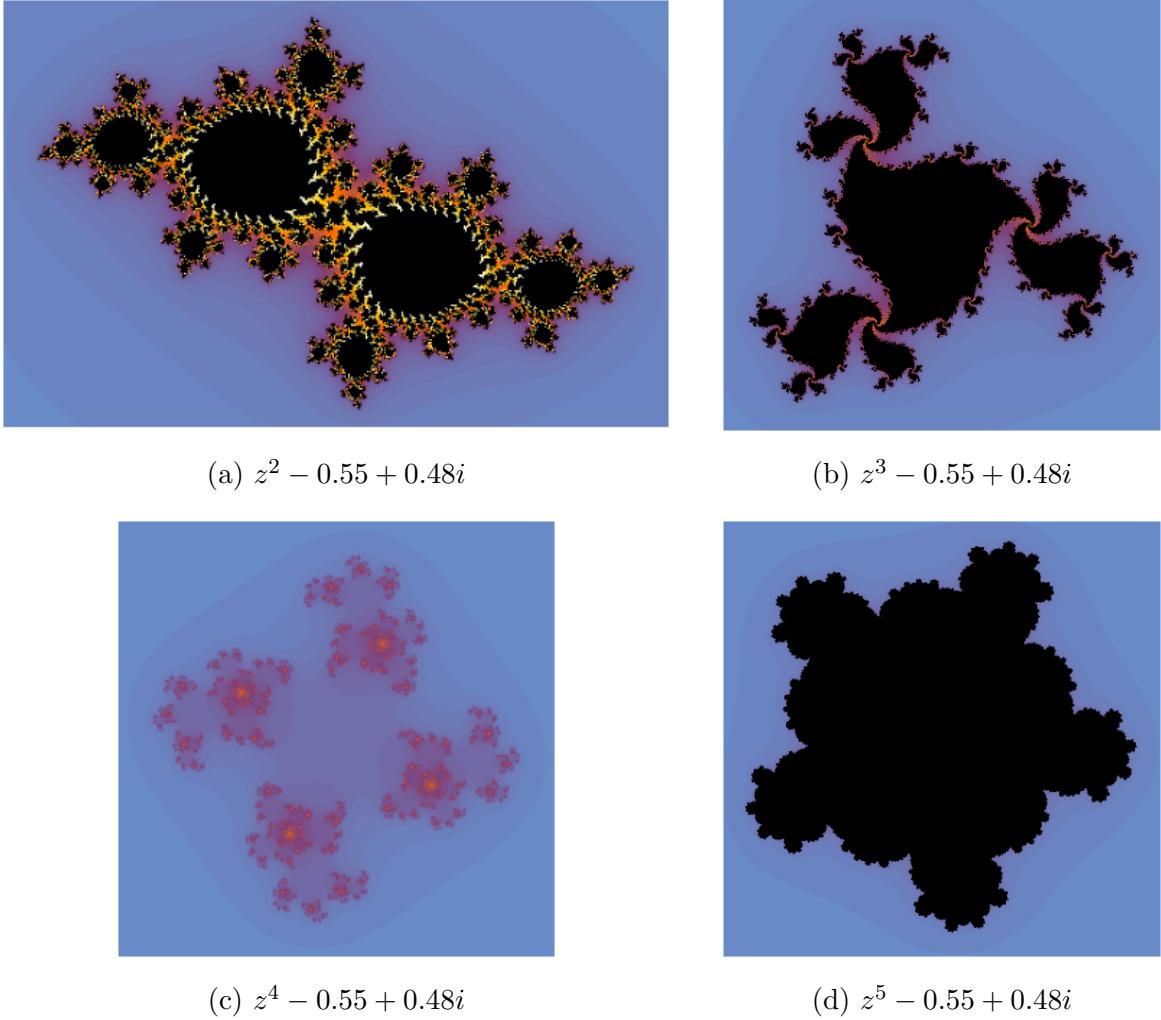


Imagen 3.10: Conjuntos de Julia con $P_{-0.55+0.48i,m}$ para distintos valores de m .

es polvareda.

Finalmente, invitamos al lector a visitar una web interactiva que hemos desarrollado como parte del TFG, la cual se describe minuciosamente su diseño e implementación a partir del capítulo 5. En esta web es posible visualizar tantos conjuntos de Julia y Mandelbrot estándar y generalizados como desee, además de modificar dinámicamente parámetros como la constante c de los conjuntos de Julia o hacer ‘zoom’ a las distintas regiones para poder observar los detalles y entresijos que nos ofrecen los conjuntos de Julia y de Mandelbrot. Su url es <https://jantoniovjr.github.io/Geometria-Fractal/>.

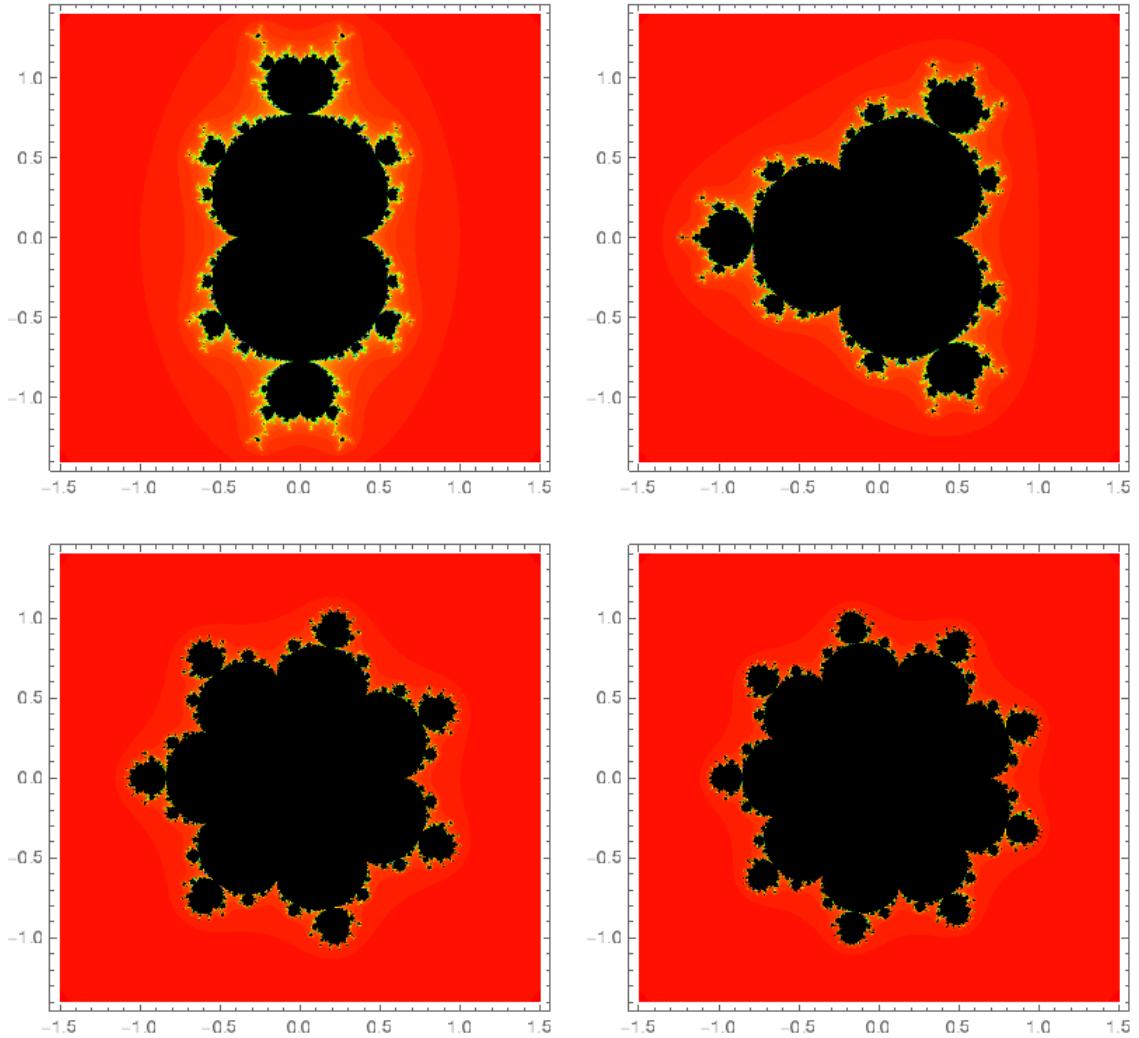
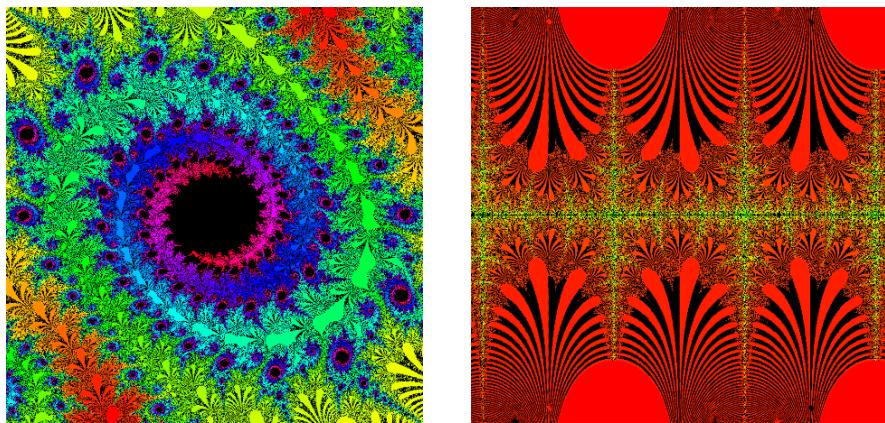


Imagen 3.11: Conjuntos de Mandelbrot \mathcal{M}_m para $m = 3, 4, 8, 10$

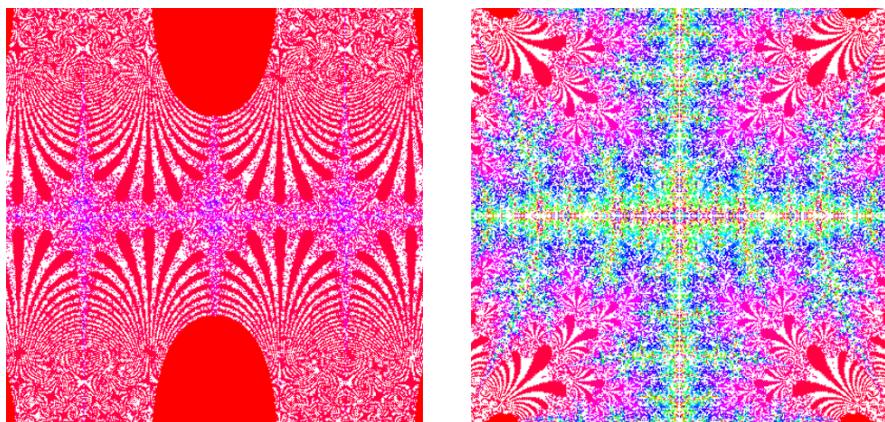
3.6.2. Conjuntos de Julia con funciones no polinómicas

Además de todos los conjuntos de Julia ya tratados y visualizados, utilizando otras funciones no polinómicas como pueden ser las trigonométricas y las exponenciales (complejas) también se pueden graficar algunos conjuntos con formas distintas a las ya vistas y con hermosas propiedades. A continuación mostraremos tan solo algunos ejemplos:

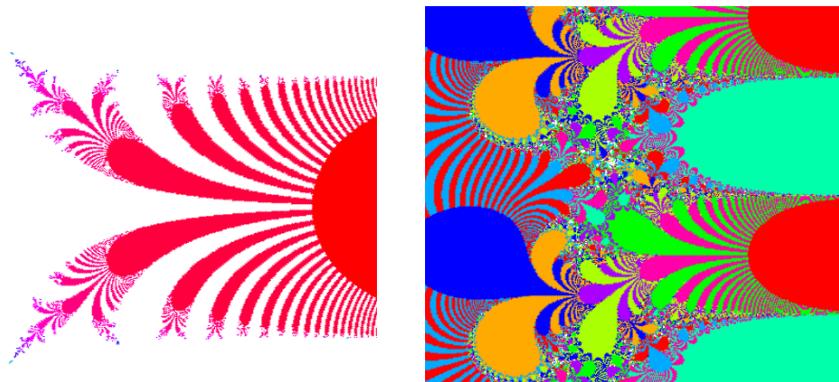
- Con la familia $f_c(z) = c \cdot \sin(z)$:



- Con la familia $f_c(z) = c \cdot \cos(z)$:



- Con la familia $f_c(z) = c \cdot e^z$:



CAPÍTULO 4

SISTEMAS DE FUNCIONES ITERADAS

Como venimos viendo ya en los dos últimos capítulos, la iteración es una poderosa herramienta en la generación de imágenes fractales. Sin embargo, hasta ahora siempre nos estamos basando en buscar convergencia y velocidad de convergencia de sucesiones de iteradas de funciones complejas, las cuales evalúan un número complejo $z \in \mathbb{C}$ para devolver otro número complejo $f(z) \in \mathbb{C}$. Si recuperamos la identificación $z = x + y \cdot i \simeq (x, y) \in \mathbb{R}^2$ podemos ver las funciones complejas como campos vectoriales de \mathbb{R}^2 , y si en lugar de aplicar una función a un único punto la aplicamos a un conjunto de puntos llegamos a la base de los *Sistemas de Funciones Iteradas*, en adelante SFI.

La matemática que explica los SFI se puede encontrar en el clásico libro *Fractals Everywhere* [3] de *Michael Barnsley*. Por su parte, la geometría fractal nació en 1977 con la publicación de *The fractal geometry of nature* [21] por parte de *Benoit Mandelbrot*. En general, gracias a la geometría fractal y ayudándonos de los SFI podemos recrear resultados de imágenes y objetos con un nivel de detalle que la geometría euclídea no puede conseguir. Sin embargo, el problema inverso también es interesante: ¿es posible, a partir de un objeto, describirlo matemáticamente mediante un SFI? Este es un área de la matemática aún abierta y en la que a día de hoy se continua trabajando. Uno de los resultados más conocidos en este ámbito es el *teorema del collage*, del cual hablaremos en la sección 4.4.

4.1. Transformaciones afines en el plano euclídeo y SFI

Si nos restringimos al plano euclídeo \mathbb{R}^2 visto como espacio afín, previo a la definición formal de SFI necesitamos unas nociones sobre transformaciones afines y maneras de representarlas, ya que estas serán las que compongan fundamentalmente los SFI.

Definición 4.1.1 (Transformación afín). Una transformación $w : \mathbb{R}^2 \rightarrow \mathbb{R}^2$ de la forma

$$w(x_1, x_2) = (ax_1 + bx_2 + e, cx_1 + dx_2 + f) \quad \forall (x_1, x_2) \in \mathbb{R}^2 \quad (4.1)$$

donde las constantes a, b, c, d, e, f son números reales es denominada una **transformación afín** del plano euclídeo.

Una forma equivalente de denotar w matricialmente es tomando la matriz $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \in \mathcal{M}_2(\mathbb{R})$ y el vector $\mathbf{b} = \begin{pmatrix} e \\ f \end{pmatrix}$ y expresar w como:

$$w(x) = w \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = Ax + \mathbf{b} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix} \quad \forall x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \in \mathbb{R}^2. \quad (4.2)$$

Hasta aquí, se trata de un concepto afín que además es bien conocido. Sin embargo, podemos visualizarlo de una forma equivalente, dando un sentido geométrico a la matriz A , pudiendo expresarla de la siguiente forma:

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} r \cos \alpha & -s \sin \beta \\ r \sin \alpha & s \cos \beta \end{pmatrix} \quad (4.3)$$

donde el par (r, α) son las coordenadas polares de (a, c) y $(s, \beta + \frac{\pi}{2})$ son las coordenadas polares de (b, d) . De esta forma, una transformación afín puede verse representada por 6 números reales $r, s, \alpha, \beta, e, f$, de forma que:

- r, s representan una homotecia o escalado de razón r en el eje X y razón s en el eje Y .
- α, β denotan una rotación de α radianes en la componente X y β en la componente Y .
- e, f simbolizan una traslación de vector $\mathbf{b} = \begin{pmatrix} e \\ f \end{pmatrix}$.

Nótese que la transformación lineal $x \mapsto Ax$ en \mathbb{R}^2 lleva un paralelogramo con un extremo en el origen en otro paralelogramo con un extremo en el origen, como consecuencia de la linealidad de las aplicaciones *homotecia* y *rotación*. Por lo que la transformación afín $w(x) = Ax + \mathbf{b}$ es una composición de la aplicación lineal representada por A , la cual transforma el espacio relativo al origen, y de la *traslación* de vector $\mathbf{b} = (e, f)^t$. A continuación definimos un caso concreto de transformación afín más familiar.

Definición 4.1.2. Una transformación afín de \mathbb{R}^2 $w(x) = Ax + \mathbf{b}$, con $\mathbf{b} \in \mathbb{R}^2$, se denomina **similitud** si la matriz A tiene alguna de las siguientes formas:

$$A = \begin{pmatrix} r \cos \alpha & -r \sin \alpha \\ r \sin \alpha & r \cos \alpha \end{pmatrix}, \quad A = \begin{pmatrix} r \cos \alpha & r \sin \alpha \\ r \sin \alpha & -r \cos \alpha \end{pmatrix} \quad (4.4)$$

para $r \neq 0$, $0 \leq \alpha < 2\pi$. A r se le llama **razón de la homotecia** o factor de escala y a α se le llama **ángulo de rotación**.

Nótese que en caso $\alpha = \pi, \beta = 0$ se consigue una reflexión respecto al eje Y , y viceversa.

Para aclarar todos estos conceptos en la figura 4.1 podemos ver cómo actúan distintas transformaciones lineales sobre una figura simple: el polígono formado al unir los vértices $(0, 0), (1, 0), (1.5, 0.5), (1, 1)$ y $(0, 1)$. Las transformaciones lineales vienen representadas en cada caso por una sextupla $w = (r, s, \alpha, \beta, e, f)$, teniendo cada elemento el significado ya definido.

En el caso de (a) observamos una similitud con $r = 0.5$ y $\alpha = \frac{\pi}{6}$. En (b) además de un escalado uniforme de razón $r = s = 0.5$ podemos ver el efecto que tiene una rotación (no uniforme) en X de razón $\alpha = \frac{\pi}{6}$. Por su parte, (c) simplemente aplica una traslación mediante el vector $\mathbf{b} = (0.1, 0.1)^t$. Por último, en (d) vemos un caso de reflexión respecto al eje Y .

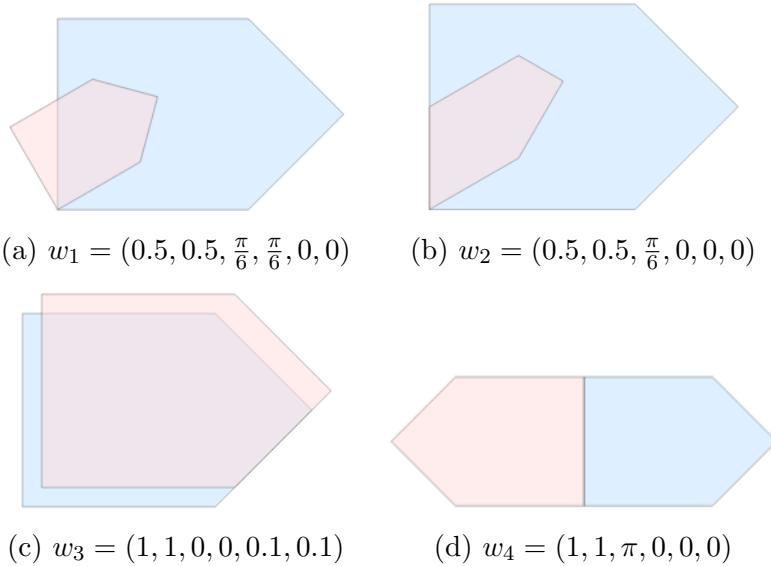


Imagen 4.1: Ejemplos de aplicaciones de transformaciones lineales

Un último detalle para la creación de SFI es la necesidad de que las transformaciones afines utilizadas sean *aplicaciones contractivas*. Procedemos por tanto a definir un SFI:

Definición 4.1.3 (Sistema de Funciones Iteradas). Un **Sistema de Funciones Iteradas** se compone de un espacio métrico completo (X, d) y de un conjunto finito de aplicaciones contractivas $w = \{w_i : i = 1, \dots, n\}$.

Se denomina *constante de contractividad* del SFI a la mayor de las constantes de contractividad de las aplicaciones que lo forman, $s = \max\{s_i : i = 1, \dots, n\}$, siendo s_i la constante de contractividad de $w_i \forall i = 1, \dots, n$.

Dado un subconjunto $A \subseteq X$, la imagen de A por medio de w es definida como

$$w(A) = \bigcup_{i=1}^n w_i(A).$$

Podemos utilizar como espacio métrico completo el plano euclídeo \mathbb{R}^2 y un conjunto finito de transformaciones lineales contractivas.

Ejemplo 4.1.1. Supongamos que tenemos un triángulo equilátero T cuyos vértices son los puntos $(0, 0), (1, 0), (\frac{1}{2}, \frac{\sqrt{3}}{2})$ y las transformaciones lineales

$$\begin{aligned} w_1 &= (0.5, 0.5, 0, 0, 0, 0) \\ w_2 &= \left(0.5, 0.5, 0, 0, \frac{1}{2}, 0\right) \\ w_3 &= \left(0.5, 0.5, 0, 0, \frac{1}{4}, \frac{\sqrt{3}}{4}\right) \end{aligned}$$

Entonces en la imagen 4.2 podemos ver tanto T como el resultado de aplicar el SFI $w = \{w_1, w_2, w_3\}$ a T .

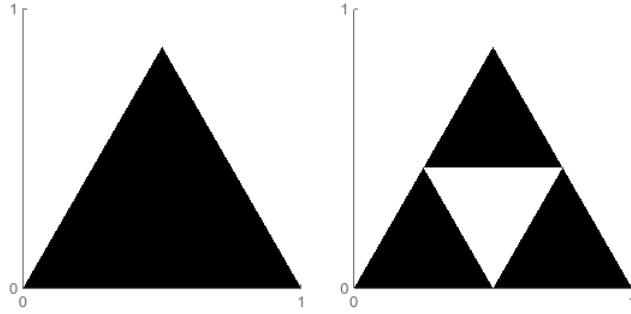


Imagen 4.2: Representación gráfica de T y $w(T)$

4.2. Convergencia de SFI

Probablemente el resultado de aplicar el SFI w a un triángulo equilátero que vemos en el ejemplo 4.1.1 le recuerde al primer paso al generar el triángulo de Sierpinski, el cual vimos en la sección 1.1.2. De hecho, podemos volver a aplicar w a $w(T)$, a $w(w(T))$, y así sucesivamente, de forma que iterando infinitamente w en T , el resultado final que obtenemos es efectivamente el triángulo de Sierpinski **S**, véase de nuevo la imagen 1.3.

Este es sólo un ejemplo, pero a lo largo de esta sección veremos que todo SFI converge a una figura, que denominamos el atractor del sistema, independientemente de la figura inicial. Véase en la figura 4.4 cómo incluso tomando como semilla una figura totalmente distinta al triángulo equilátero el resultado de la iteración es el mismo. Esto es de hecho una consecuencia del Teorema del punto fijo de Banach (teorema 2.1.1).

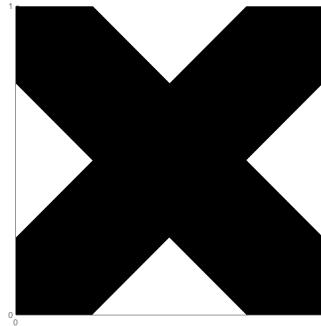
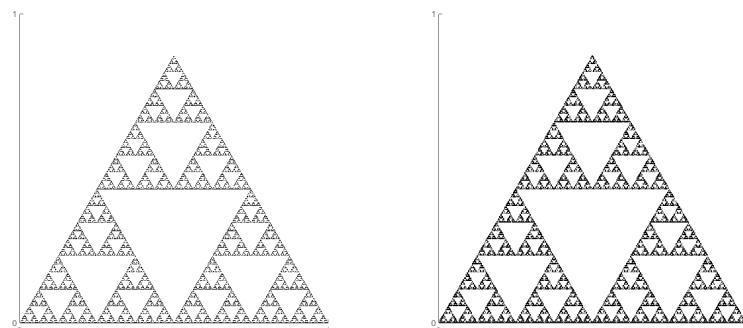


Imagen 4.3: Otra posible semilla para iterar el SFI



(a) Iterando un triángulo equilátero (b) Iterando la figura 4.3

Imagen 4.4: Resultado de iterar 8 veces w con distintas figuras iniciales

4.2.1. El Espacio de Fractales y la Métrica de Hausdorff

Consideramos un espacio métrico completo (X, d) y sea $\mathcal{H}(X) \subset 2^X$ el conjunto de todos los subconjuntos compactos de X , el cual también se denomina *espacio de fractales* de X . El objetivo ahora es dotar a $\mathcal{H}(X)$ de una distancia, la cual nos permita cuantificar la similaridad entre conjuntos compactos.

Dado un punto $x \in X$ y un subconjunto $A \in \mathcal{H}(X)$ la distancia de un punto a un conjunto es definida como

$$d(x, A) := \inf\{d(x, a) : a \in A\} = \min\{d(x, a) : a \in A\},$$

donde el ínfimo es un mínimo porque A es compacto y la función $d(x, \cdot)$ es continua. Si tomamos dos compactos $A, B \in \mathcal{H}(X)$, entonces la distancia de A a B se define como:

$$d(A, B) = \max\{d(a, B) : a \in A\}$$

El problema es que esta definición no nos basta para definir una distancia entre conjuntos, pues si $A \subset B$ tenemos que $d(A, B) = 0$, pero si $b \in B \setminus A$, entonces $d(b, A) > 0$, por lo que necesariamente $d(B, A) > 0$. Y como sabemos, una auténtica distancia es simétrica. Véase el contraejemplo de la figura 4.5.

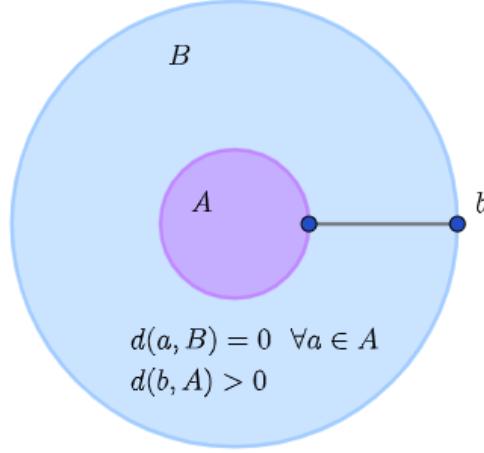


Imagen 4.5: Contraejemplo a la distancia entre conjuntos

Por suerte, a partir de estas definiciones no es difícil encontrar una definición que cumpla las propiedades de una distancia.

Definición 4.2.1. Dado un espacio métrico completo (X, d) , en el espacio de fractales $\mathcal{H}(X)$ se define la **distancia de Hausdorff** o **métrica de Hausdorff** como:

$$h(d)(A, B) = \max\{d(A, B), d(B, A)\} \quad \forall A, B \in \mathcal{H}(X).$$

En adelante denotaremos únicamente $h(\cdot, \cdot)$, omitiendo la dependencia de la distancia del espacio original.

Podemos comprobar en [3, Sección 2.6] que, en efecto, h es una distancia. Además, en [3, Sección 2.7] se prueba que el espacio métrico $(\mathcal{H}(X), h(d))$ es completo.

4.2.2. Aplicaciones contractivas en el espacio de fractales

Consideramos un espacio métrico completo (X, d) y su espacio de fractales dotado de la métrica de Hausdorff $(\mathcal{H}(X), h(d))$, que también es completo, y tomamos una aplicación contractiva $w : X \rightarrow X$. Buscamos averiguar qué ocurre al iterar w sobre $\mathcal{H}(X)$, siendo esta contractiva sobre X .

Lema 4.2.1. Sea (X, d) un espacio métrico y $w : X \rightarrow X$ una aplicación contractiva, entonces

$$A \in \mathcal{H}(X) \Rightarrow w(A) \in \mathcal{H}(X),$$

es decir, la imagen por w de todo compacto de X es un conjunto compacto de X .

Demostración. Sabemos que w es continua en X , pues toda aplicación contractiva es lipschitziana y por tanto continua. Como la imagen de un conjunto compacto por una aplicación continua es un conjunto compacto, podemos afirmar que w lleva elementos de $\mathcal{H}(X)$ a elementos de $\mathcal{H}(X)$. \square

Ahora necesitamos alguna forma de construir aplicaciones contractivas en el espacio $(\mathcal{H}(X), h)$ a partir de aplicaciones contractivas en (X, d) . Gracias al siguiente lema comprobamos que la forma más natural es suficiente.

Lema 4.2.2. Sea (X, d) un espacio métrico y $w : X \rightarrow X$ una aplicación contractiva con constante de Lipschitz s . Entonces $w : \mathcal{H}(X) \rightarrow \mathcal{H}(X)$, definida naturalmente como

$$w(A) = \{w(a) : a \in A\} \quad \forall A \in \mathcal{H}(X)$$

es una aplicación contractiva en $(\mathcal{H}(X), h)$.

Demostración. Si utilizamos el lema 4.2.1 podemos afirmar que la aplicación $w : \mathcal{H}(X) \rightarrow \mathcal{H}(X)$ está bien definida, por lo que falta probar que en efecto es contractiva. Sean $A, B \in \mathcal{H}(X)$, tenemos que

$$\begin{aligned} d(w(A), w(B)) &= \max \{ \min \{ d(w(a), w(b)) : b \in B \} : a \in A \} \\ &\leq \max \{ \min \{ s \cdot d(a, b) : b \in B \} : a \in A \} \\ &= s \cdot d(A, B). \end{aligned}$$

Análogamente, $d(w(B), w(A)) \leq s \cdot d(B, A)$. Por tanto

$$\begin{aligned} h(w(A), w(B)) &= \max \{ d(w(A), w(B)), d(w(B), w(A)) \} \\ &\leq s \max \{ d(A, B), d(B, A) \} \\ &= s \cdot h(A, B). \end{aligned}$$

Por lo que tenemos que w es contractiva sobre $\mathcal{H}(X)$. \square

Enunciamos una propiedad de la métrica de Hausdorff h que nos hará falta dentro de poco.

Lema 4.2.3. Sean $A, B, C, D \in \mathcal{H}(X)$, entonces

$$h(A \cup B, C \cup D) \leq \max \{ h(A, C), h(B, D) \}.$$

Demostración. La prueba se sigue de otra propiedad de la distancia d :

$$\begin{aligned} d(A \cup B, C) &= \max\{d(x, C) : x \in A \cup B\} \\ &= \max\{\max\{d(x, C) : x \in A\}, \max\{d(x, C) : x \in B\}\} \\ &= \max\{d(A, C), d(B, C)\}. \end{aligned}$$

Y de esta igualdad se deduce la desigualdad pedida. \square

Seguidamente presentamos un resultado que nos ayuda a construir aplicaciones contractivas en $\mathcal{H}(X)$ a partir de no sólo una sino de varias aplicaciones contractivas en X , a través del cual enlazaremos, esta vez en un contexto más teórico y formal, con la definición de SFI como conjunto de aplicaciones contractivas en un espacio métrico completo.

Proposición 4.2.1. Sea (X, d) un espacio métrico y sea $\{w_i : i = 1, \dots, n\}$ un conjunto finito de aplicaciones contractivas $w_i : \mathcal{H}(X) \rightarrow \mathcal{H}(X)$, cada una con constante de contractividad s_i . Definimos la aplicación $W : \mathcal{H}(X) \rightarrow \mathcal{H}(X)$ como

$$W(A) := \bigcup_{i=1}^n w_i(A) \quad \forall A \in \mathcal{H}(X). \quad (4.5)$$

Entonces W es una aplicación contractiva en $\mathcal{H}(X)$ y su constante de contractividad es $s = \max\{s_i : i = 1, \dots, n\}$.

Demostración. Demostraremos el caso $n = 2$, de forma que por inducción sería cierto para cualquier $n \geq 1$. Sean $A, B \in \mathcal{H}(X)$, tenemos que

$$\begin{aligned} h(W(A), W(B)) &= h(w_1(A) \cup w_2(A), w_1(B) \cup w_2(B)) \\ &\leq \max\{h(w_1(A), w_1(B)), h(w_2(A), w_2(B))\} \text{ (lema 4.2.3)} \\ &\leq \max\{s_1 \cdot h(A, B), s_2 \cdot h(A, B)\} \\ &\leq \max\{s_1, s_2\} h(A, B) \\ &= s \cdot h(A, B) \end{aligned}$$

Lo cual completa la demostración. \square

4.2.3. El espacio de fractales y los SFI

Hasta ahora hemos probado varios resultados que nos han servido para construir aplicaciones contractivas en el espacio de fractales de un espacio métrico a partir de un conjunto finito de aplicaciones contractivas. Si añadimos la hipótesis de la complitud, tendríamos con ese conjunto de aplicaciones contractivas un SFI, el cual podemos iterar bajo la seguridad de que no perdemos dicha contractividad. Por lo tanto, y recuperando la definición 4.1.3, ya podemos recopilar toda la información que tenemos y enunciar el siguiente teorema:

Teorema 4.2.1. Consideramos el SFI formado por el espacio métrico completo (X, d) y el conjunto $\{w_i : i = 1, \dots, N\}$ de aplicaciones contractivas, siendo s la constante de contractividad del SFI. Consideramos también el espacio de fractales $(\mathcal{H}(X), h)$, donde definimos la aplicación $W : \mathcal{H}(X) \rightarrow \mathcal{H}(X)$ como en (4.5):

$$W(A) := \bigcup_{i=1}^N w_i(A) \quad \forall A \in \mathcal{H}(X).$$

Entonces W es una aplicación contractiva de constante s y admite un único punto fijo $A^* \in \mathcal{H}(X)$, el cual está dado por

$$A^* = \lim_{n \rightarrow \infty} W^n(B) \quad \forall B \in \mathcal{H}(X).$$

Demostración. Gracias a los resultados anteriores solo nos quedaría probar que W admite un único punto fijo dado por la iteración infinita de W e independientemente de qué $B \in \mathcal{H}(X)$ inicial se tome. Sin embargo, teniendo en cuenta que W es contractiva y que $\mathcal{H}(X)$ es un espacio métrico completo, simplemente aplicando el teorema del punto fijo de Banach (teorema 2.1.1) se tiene el resultado completo \square

En resumen, tenemos que todo SFI admite un único punto fijo al que converge la iteración sucesiva de la aplicación W construida a partir de su conjunto de aplicaciones contractivas. Además, esta convergencia está asegurada sea cual sea el conjunto inicial en el que se comienzan las iteradas. Como veníamos anunciando al inicio de esta sección, podemos ponerle nombre al punto fijo del SFI:

Definición 4.2.2 (Atractor). Dado un SFI, definimos su único punto fijo como el **atractor** del SFI.

Recordamos ahora el SFI del ejemplo 4.1.1 y las imágenes 4.4, que nos permitirían comprobar que el triángulo de Sierpinski es el atractor del SFI y que independientemente de la figura inicial (siempre y cuando sea un conjunto compacto de \mathbb{R}^2) la iteración conjunta de $\{w_1, w_2, w_3\}$ converge a \mathbf{S} como resultado.

Ahora podemos aplicar esta teoría al espacio métrico completo \mathbb{R}^2 y a las transformaciones afines, que para asegurar la convergencia necesitamos que sean contracciones. Afortunadamente, es sencillo probar esta condición.

Proposición 4.2.2. Sea $w = (r, s, \alpha, \beta, e, f)$ una transformación afín de \mathbb{R}^2 . Si $\alpha = \beta$ y $\max\{|r|, |s|\} < 1$, entonces w es una aplicación contractiva.

Demostración. Por un lado, las rotaciones uniformes y las traslaciones son movimientos rígidos, en particular aplicaciones lipschitzianas de constante de Lipschitz igual a 1. Por tanto, por composición, para comprobar que w es contractiva, debemos probar que un escalado (uniforme o no) es una aplicación contractiva si $\max\{|r|, |s|\} < 1$. Supongamos que la aplicación $h(x_1, x_2) = (r \cdot x_1, s \cdot x_2)$ verifica $K = \max\{|r|, |s|\} < 1$. Entonces, para cualesquiera $x = (x_1, x_2), y = (y_1, y_2) \in \mathbb{R}^2$:

$$\begin{aligned} \|h(x) - h(y)\| &= \|(rx_1 - ry_1, sx_2 - sy_2)\| \\ &= \|(r(x_1 - y_1), s(x_2 - y_2))\| \\ &< K\|x - y\| \end{aligned}$$

Por tanto h es contractiva. \square

Y una vez que sabemos cómo hacer que una transformación afín sea contractiva, podemos, sin importar el conjunto de partida, obtener imágenes fractales a partir únicamente de una colección finita de sextuplas, cada una representando una transformación afín contractiva. Como posibles ejemplos, y para que se observe la diversidad que ofrece este método, en las tablas 4.1 y 4.2

w	r	s	α	β	e	f
1	1	1	0	0	0	0
2	$1/\sqrt{2}$	$1/\sqrt{2}$	$\pi/4$	$\pi/4$	0	1
3	$1/\sqrt{2}$	$1/\sqrt{2}$	$-\pi/4$	$-\pi/4$	$1/2$	$3/2$

Tabla 4.1: SFI para el árbol pitagórico

w	r	s	α	β	e	f
1	0.85	0.85	$-\pi/72$	$-\pi/72$	0	1.6
2	0.3	0.34	$49\pi/180$	$49\pi/180$	0	1.6
3	0.3	0.34	$2\pi/3$	$2\pi/3$	0	0.44
4	0	0.3	0	0	0	0

Tabla 4.2: SFI para el helecho de Barnsley

se pueden observar los SFI cuyos atractores son el árbol pitagórico y el helecho de Barnsley (imágenes 4.6 (a) y (b) respectivamente), siendo este último una imagen cuya representación es ya bastante realista.

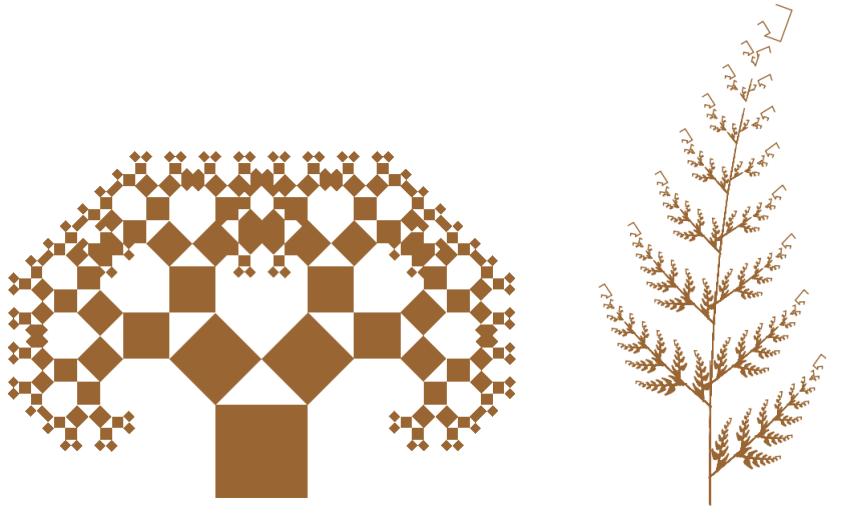


Imagen 4.6: Atractores de los SFI definidos en las tablas 4.1 y 4.2.

4.3. SFI y conjuntos autosimilares

En el capítulo 1 y concretamente en la sección 1.2.4 presentamos algunas relaciones entre los distintos tipos de dimensión, y en la sección 1.2.1 hablamos de la dimensión por cajas de un conjunto. Si nos restringimos a conjuntos autosimilares el cálculo de la dimensión por cajas se simplifica gracias a algunos resultados que le debemos a la teoría de los SFI. Para ello, previamente introducimos la siguiente definición.

Definición 4.3.1. Sean (X, d) un espacio métrico completo y un SFI $w = \{w_i : i = 1, \dots, n\}$ en X . w satisface la **condición de conjunto abierto** si existe un conjunto abierto no vacío $U \subseteq X$ tal que las imágenes $w_1(U), \dots, w_n(U) \subseteq U$ y son disjuntas.

Intuitivamente, esta condición nos afirma que podemos controlar el ‘tamaño’ del resultado de aplicar el SFI y que además las imágenes de cada aplicación que compone el SFI no se solapan unas con otras. Gracias a esta condición podemos enunciar el siguiente teorema, que es el que nos proporcionará las herramientas esenciales para el cálculo de la dimensión en estos casos.

Teorema 4.3.1 (Moran). *Sean (X, d) un espacio métrico completo. En X se define un SFI $w = \{w_i : i = 1, \dots, n\}$ en X que satisface la condición de conjunto abierto y $\{c_i : i = 1, \dots, n\}$ las constantes de contractividad de las aplicaciones w_i . Entonces la dimensión por cajas del atractor de w es el único valor real positivo d que satisface la ecuación $1 = c_1^d + \dots + c_n^d$ y esta coincide con la dimensión de Hausdorff.*

Podemos encontrar la prueba de este teorema en [23].

Corolario 4.3.1. *En caso de que todas las constantes de contractividad sean iguales a cierta constante $0 < c < 1$, entonces la dimensión d es*

$$d = \frac{\log(n)}{\log(1/c)}.$$

Observación 4.3.1. En \mathbb{R}^2 , a partir de un conjunto autosimilar según la definición 1.0.1 podemos construir un SFI cuyo atractor sea el propio conjunto autosimilar.

Mediante el teorema de Moran y viendo ciertos fractales como atractores de SFI podemos rápidamente calcular su dimensión por cajas. Por ejemplo, el triángulo de Sierpinski es el atractor del SFI presentado en el ejemplo 4.1.1. El SFI cumple la condición de conjunto abierto tomando como U el interior del propio triángulo equilátero inicial y viendo, como podemos observar en la imagen 4.2 que las imágenes por cada una de las aplicaciones no salen del triángulo y además son disjuntas. En este caso tenemos tres similitudes, en todas ellas la constante de contractividad es $\frac{1}{2}$, por lo que aplicando el corolario su dimensión por cajas es

$$\dim_B(\mathbf{S}) = \frac{\log(3)}{\log(2)} \approx 1.585,$$

confirmando la suposición que hicimos en la sección 1.2.1.

Por su parte, el conjunto de Cantor es el atractor del SFI de la tabla 4.3, que cumple la condición de conjunto abierto tomando el intervalo $U = (0, 1)$. La constante de contractividad es $\frac{1}{3}$, luego podemos asegurar que

$$\dim_B(\mathbf{C}) = \frac{\log(2)}{\log(3)} \approx 0.6309.$$

w	r	s	α	β	e	f
1	$1/3$	$1/3$	0	0	0	0
2	$1/3$	$1/3$	0	0	$1/3$	0

Tabla 4.3: SFI para el conjunto de Cantor

Para calcular la dimensión por cajas del triángulo equilátero T de la sección 1.2.1 podemos tomar el mismo SFI que hemos utilizado para el triángulo de Sierpinski en el ejemplo 4.1.1 pero

añadiéndole además la transformación afín $w_4 = (0.5, 0.5, \pi/3, \pi/3, 0.5, 0)$, en cuyo caso $n = 4$ y la constante de contractividad es $c = 1/2$, por lo que

$$\dim_B(T) = \frac{\log(4)}{\log(2)} = 2,$$

pudiendo finalmente confirmar que la dimensión por cajas de un triángulo equilátero es 2.

Es por tanto el momento de hallar la dimensión por cajas de la curva de Koch \mathbf{K} , la cual no hemos calculado aún. La curva de Koch es el atractor del SFI definido por las transformaciones especificadas en la tabla 4.4. Fijémonos que son 4 transformaciones afines, todas ellas similitudes cuya homotecia tiene la misma razón: $\frac{1}{3}$. Por tanto, aplicando el corolario, vemos que

$$\dim_B(\mathbf{K}) = \frac{\log(4)}{\log(3)} \approx 1.2618,$$

que es un valor situado entre 1 y 2.

w	r	s	α	β	e	f
1	1/3	1/3	0	0	0	0
2	1/3	1/3	$\pi/3$	$\pi/3$	1/3	0
3	1/3	1/3	$-\pi/3$	$-\pi/3$	1/2	$\sqrt{3}/6$
4	1/3	1/3	0	0	2/3	0

Tabla 4.4: SFI para la curva de Koch

Para comprobar que los atractores de los SFIs que presentamos son los que realmente afirmamos que son puede ejecutarse el código *Mathematica* que se ha utilizado para graficar muchas de las imágenes de este capítulo. Puede encontrarse, junto con los códigos de los demás capítulos, en <https://github.com/JAntonioVR/Geometria-Fractal/tree/main/Mathematica>.

4.4. El problema inverso

Como ya hemos podido ver, los SFIs nos abren un mundo de posibilidades para crear imágenes fractales, ya que todos convergen a un atractor. Sin embargo, podemos también plantearnos el problema inverso. A partir de una imagen, ¿es posible determinar un SFI tal que su atractor sea la imagen inicial? Este es el punto clave de la teoría de la compresión por imágenes fractales. Pensemos que es mucho más ligero almacenar en memoria un conjunto finito de números en coma flotante que una imagen (viéndose esta como una colección de píxeles).

En este contexto, la mejor herramienta que tenemos es el conocido como *teorema del collage*, que enunciamos a continuación.

Teorema 4.4.1 (El teorema del collage). *Sean (X, d) un espacio métrico completo, $L \in \mathcal{H}(X)$ y $\varepsilon \geq 0$. Si $\{w_i : i = 1, \dots, N\}$ es un SFI en X con constante de contractividad $0 \leq s < 1$ cumpliendo que*

$$h\left(L, \bigcup_{i=1}^N w_i(L)\right) \leq \varepsilon,$$

donde $h(d)$ es la métrica de Hausdorff, entonces

$$h(L, A) \leq \frac{\varepsilon}{1-s}$$

siendo A el atractor del SFI. Equivalentemente,

$$h(L, A) \leq (1-s)^{-1} h\left(L, \bigcup_{i=1}^N w_i(L)\right) \quad \forall L \in \mathcal{H}(X).$$

Para probar el teorema del collage primero establecemos el siguiente lema, que nos deja la mayor parte del trabajo hecho.

Lema 4.4.1. Sea (X, d) un espacio métrico completo. Sea $f : X \rightarrow X$ una aplicación contractiva con constante de contractividad $0 \leq s < 1$ y consideramos $x^* \in X$ el punto fijo de f . Entonces

$$d(x, x^*) \leq (1-s)^{-1} d(x, f(x)) \quad \forall x \in X.$$

Demostración. Fijado cualquier $x \in X$, veamos primeramente la siguiente cadena de desigualdades

$$\begin{aligned} d(x, x^*) &\leq d(x, f(x)) + d(f(x), x^*) \text{ (por la desigualdad triangular)} \\ &= d(x, f(x)) + d(f(x), f(x^*)) \quad (f(x^*) = x^*) \\ &\leq d(x, f(x)) + s \cdot d(x, x^*). \quad (f \text{ es contractiva}) \end{aligned}$$

Por lo que agrupando, tenemos que $d(x, x^*)(1-s) \leq d(x, f(x))$, lo cual equivale a

$$d(x, x^*) \leq (1-s)^{-1} d(x, f(x)).$$

□

Aplicando el lema 4.4.1 al espacio métrico $(\mathcal{H}(X), h(d))$ y a la aplicación contractiva W generada por $\{w_i : i = 1, \dots, N\}$ como en (4.5) tenemos probado el teorema del collage.

Intuitivamente, este teorema nos dice que para encontrar un SFI cuyo atractor es similar a un conjunto dado, debemos buscar un conjunto finito de aplicaciones contractivas (transformaciones afines contractivas en \mathbb{R}^2) tales que la unión (el collage) formada por las imágenes del conjunto dado vía dichas aplicaciones se parezca a sí mismo. La forma de medir esta similaridad de manera cuantitativa es mediante la métrica de Hausdorff.

Ejemplo 4.4.1. Vamos a aplicar el teorema del collage para encontrar un SFI que nos ayude a replicar la imagen 4.7 (a).

Para ello, siguiendo lo que nos sugiere el teorema del collage debemos encontrar transformaciones afines tales que la unión de las imágenes de este conjunto sea lo más parecida posible al conjunto original. Llamemos L al subconjunto de \mathbb{R}^2 que representa este fractal.

Fijémonos, tal y como se puede ver en la imagen 4.7 (b) que L está compuesto por tres copias reducidas de sí mismo. Por lo que trataremos de buscar tres transformaciones afines tales que cada una genere una de las copias del propio L , de forma que al unirlas obtendríamos de nuevo el mismo conjunto.

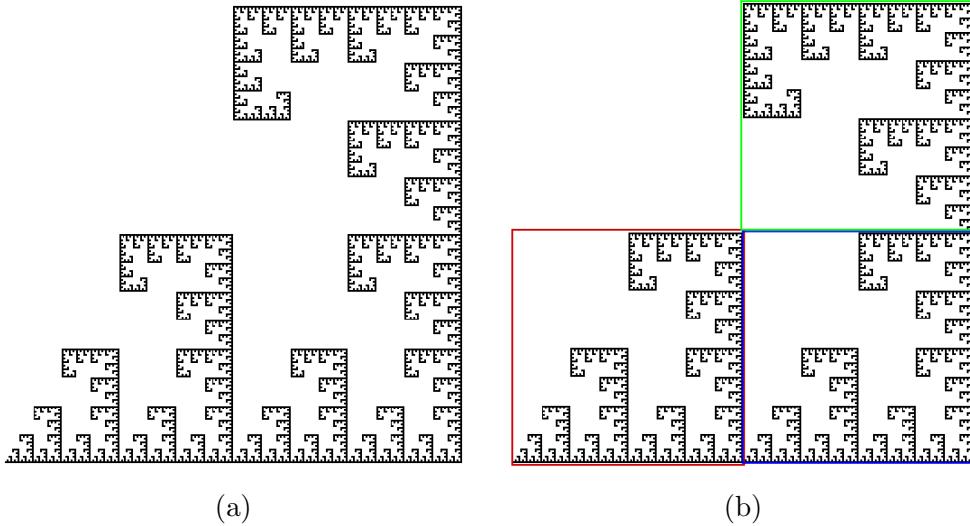


Imagen 4.7: Imagen cuyo SFI debemos determinar

- La copia **roja** es simplemente el propio L contraído a la mitad en ambos ejes, por lo que utilizamos la aplicación $w_1 = (0.5, 0.5, 0, 0, 0, 0)$.
- La copia **azul** es, al igual que la **roja**, el resultado de aplicar una homotecia de razón 0.5, pero esta vez también se desplaza 0.5 unidades a la derecha. La aplicación que buscamos es por tanto $w_2 = (0.5, 0.5, 0, 0, 0.5, 0)$.
- Por último, la copia **verde** debemos fijarnos que además de escalada y desplazada, está girada. Por lo que es el resultado de aplicar una similitud de razón de homotecia 0.5 y ángulo de rotación $\frac{\pi}{2}$, para posteriormente colocar este resultado encima de la copia **azul**. Es decir, usamos la transformación $w_3 = (0.5, 0.5, \frac{\pi}{2}, \frac{\pi}{2}, 1, 0.5)$.

Por tanto, L es el atractor del SFI que representamos en la tabla 4.5.

w	r	s	α	β	e	f
1	0.5	0.5	0	0	0	0
2	0.5	0.5	0	0	0.5	0
3	0.5	0.5	$\frac{\pi}{2}$	$\frac{\pi}{2}$	1	0.5

Tabla 4.5: SFI para la imagen 4.7

Para una mayor cantidad y variedad de ejemplos invitamos al lector a consultar [3, Sección 3.10], donde puede encontrar ejercicios y explicaciones prácticas y más complejas.

Como comentario final, decir que el teorema del collage es de hecho una potente herramienta que trasciende al ámbito fractal. Por ejemplo, se ha probado su utilidad en problemas integrales o diferenciales de estimación de parámetros. Véase [20] para tener una visión de ello.

En conclusión, los SFI en \mathbb{R}^2 a pesar de ser herramientas muy simples como lo son las transformaciones afines, nos permiten originar complejas y bellas imágenes basadas en la iteración. Además, gracias al teorema del collage podemos aproximar una imagen simplemente a partir de las transformaciones afines que definen un SFI que converge a dicha aproximación. Toda esta disciplina está además basada en una densa teoría que gira en torno al teorema del punto fijo de Banach.

CAPÍTULO 5

INTRODUCCIÓN A LAS HERRAMIENTAS DE VISUALIZACIÓN

Durante los capítulos anteriores hemos tratado los fractales desde un punto de vista teórico y matemático, ayudándonos del software *Mathematica* para visualizar imágenes de naturaleza fractal, desde los primeros ejemplos clásicos como el triángulo de Sierpinski o el copo de Koch (imágenes 1.3 y 1.7), representaciones de cuencas de atracción en el capítulo 2, conjuntos de Julia y Mandelbrot en el capítulo 3 y atractores de sistemas de funciones iteradas en el capítulo 4.

Queda claro que *Mathematica* es un software de cálculo muy útil, pero también es lento, ya que realmente no está orientado a la síntesis de imágenes. En este sentido, obsérvese que sólo hemos utilizado *Mathematica* para visualizar fractales 2D, que se presentan como subconjuntos del plano euclídeo \mathbb{R}^2 o coloreando el plano complejo \mathbb{C} . Esto se debe no solo a la simplicidad algorítmica que nos proporciona limitar los razonamientos a 2 dimensiones, sino a que el software es realmente lento en cálculos complejos 3D.

5.1. Síntesis de imágenes en GPUs. WebGL

Afortunadamente, contamos con herramientas que están dedicadas a la producción de imágenes por ordenador. Éste es el objetivo principal de la *informática gráfica*. De forma genérica, la informática gráfica, en ocasiones también llamada *computación gráfica*, es la rama de las ciencias de la computación que aprovecha los recursos hardware y software de un computador para sintetizar y manipular digitalmente contenido visual. Concretamente, un computador puede utilizar durante el procesado tanto su CPU (unidad central de procesamiento, el procesador) como opcionalmente la GPU (unidad de procesamiento gráfico, la gráfica). La GPU es un microprocesador cuya función es realizar tareas gráficas y operaciones en coma flotante para así aligerar la carga del procesador central en aplicaciones gráficas, por ejemplo videojuegos. Así, mientras lo relacionado con gráficos e imágenes se ejecuta en la gráfica, el procesador puede dedicarse a otras cosas.

En lo que a arquitectura hardware se refiere, la CPU y la GPU presentan varias diferencias. La CPU está formada por pocos núcleos de procesamiento pero muy potentes, por lo que está principalmente orientada a la ejecución secuencial de las tareas. Por contra, la GPU está formada por cientos, y a veces incluso miles de ALUs (unidad aritmético-lógica) especializadas

en operaciones en coma flotante, por lo que está muy preparada para la ejecución paralela de tareas. Esto nos lleva a la principal ventaja del uso de la GPU. Gracias a esa gran cantidad de unidades de procesamiento es posible en cada instante procesar miles de píxeles, vértices o colores a la vez, provocando una destacable mejora de las prestaciones en lo que a tiempos se refiere.

La idea general de la computación gráfica consiste en llevar una escena, que puede ser en 2D o en 3D, compuesta por un conjunto de objetos, a una pantalla formada por un número alto pero finito de píxeles. La manera más común de hacer esto es dividir cada elemento de la escena en un conjunto de caras planas llamadas *primitivas*, generalmente triángulos, y una vez divididos calcular qué píxeles de la pantalla ocupa cada primitiva. Este procedimiento se denomina *rasterización*. La complejidad en el proceso de rasterización puede aumentar muy rápidamente, por lo que merece la pena separar qué partes de una aplicación querremos cambiar de una aplicación gráfica a otra (p. ej. vértices, objetos o colores) y cuáles es preferible encapsular y optimizar en una interfaz (p. ej. motor gráfico o código de bajo nivel). Así es como surgen las llamadas APIs de rasterización, entre las que destacan [OpenGL](#), [DirectX](#), [Metal](#) o [Vulkan](#).



Para usar la GPU, debemos especificarle a la API un fragmento de código programado para ejecutarse específicamente en la misma. Se suelen denominar *shaders* a estos trozos de código fuente que están escritos para ser ejecutados exclusivamente en la GPU. El resto de código necesario para que la API funcione se suele escribir en C/C++, pero también en Python o en JavaScript y está destinado a ejecutarse en la CPU. Estos shaders en general están escritos en su propio lenguaje: GLSL (GL Shading Language) en el caso de OpenGL y Vulkan, HLSL (High-Level Shading Language) de Microsoft para DirectX y MSL (Metal Shading Language) para Metal.

Por su parte, **WebGL** es una adaptación de OpenGL a JavaScript. Es decir, en lugar de C/C++ se emplea JavaScript como lenguaje cuyo código se ejecuta en CPU. Al igual que el propio OpenGL, utiliza GLSL para la programación de shaders. El rasgo más característico de WebGL es que, al utilizar JavaScript, es posible ejecutarlo en cualquier navegador web. Esta es la base de la gran ventaja de WebGL: su portabilidad. Una aplicación web programada en WebGL puede ser usada desde cualquier navegador con soporte, sin que el usuario tenga que instalar ningún tipo de dependencia y de forma totalmente independiente al sistema operativo que utilice el dispositivo o del fabricante de la GPU. Además, el propio JavaScript cuenta con soporte para manipular dinámicamente el DOM de un documento HTML y para la gestión de eventos. Esto, unido a la portabilidad, hace que WebGL sea la mejor opción que podemos elegir para este proyecto.

Las aplicaciones web basadas en WebGL se utilizan para la visualización de imágenes en un elemento <canvas> de HTML en los navegadores y sistemas que lo soporten¹ sin necesidad de plug-ins. El principal defecto tanto de WebGL como de otras APIs es la dificultad a la hora de querer comenzar a visualizar imágenes, pues tiene varios componentes complejos de enlazar

¹Esto se puede comprobar gracias a páginas dedicadas a ello como [esta página de testeo WebGL](#) o [WebGL Report](#)

en un principio. A continuación explicaremos el flujo de trabajo que utiliza WebGL y cómo podemos comenzar a utilizar la herramienta para visualizar fractales.

5.2. Componentes de WebGL

A grandes rasgos, los programas que utilizan WebGL se componen de código de JavaScript que interactúa con la propia biblioteca junto con código GLSL que se ejecuta en la GPU. Para dar nuestros primeros pasos con WebGL debemos, en nuestro documento HTML, utilizar un elemento `<canvas>` al cual especificamos sus dimensiones en píxeles. En el siguiente ejemplo, el canvas sería cuadrado de 720×720 píxeles. Evidentemente, a mayor número de píxeles mayor resolución, pero también mayor tiempo de cómputo.

```
1 <main>
2   <canvas id="glCanvas" width="720" height="720"></canvas>
3 </main>
```

5.2.1. Contexto de WebGL

Por su parte, en JavaScript podemos acceder mediante el DOM al elemento `<canvas>` y extraer un objeto que será lo que a partir de ahora denominemos **contexto de WebGL** (`WebGLRenderingContext`). Este objeto nos proporciona una interfaz a un contexto de OpenGL ES 2.0 para dibujar en la superficie del canvas, de forma que se pueden invocar muchas de las funciones utilizadas en OpenGL. La forma de extraer este contexto es mediante la función `getContext`:

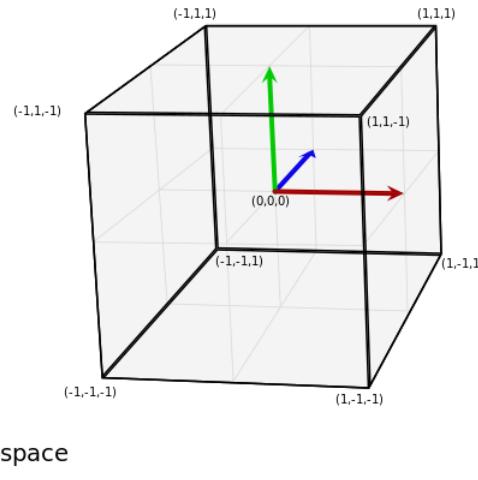
```
1 const canvas = document.querySelector("#glCanvas");
2 // Inicializamos el contexto de WebGL
3 const gl = canvas.getContext("webgl2");
4
5 // Solo continuamos si WebGL esta disponible y funciona
6 if (gl === null) {
7   alert("No se ha podido inicializar WebGL. Es posible que su
8       navegador o dispositivo no lo permitan");
9 }
```

En caso de éxito, que se tiene en la mayoría de las ocasiones, debemos conservar este objeto, pues será necesario para utilizar la gran mayoría de directivas que implementa WebGL. Los siguientes elementos son los auténticos encargados de los objetos que se representan en el canvas.

5.2.2. El programa *Shader*

El ‘**shader**’ es un programa escrito en el llamado **OpenGL ES Shading Language**, más comúnmente conocido como **GLSL**, que a partir de la información sobre los vértices que forman una figura genera un color para cada píxel, para así dibujar dicha figura en el canvas. Hay dos tipos de *shader*: el **vertex shader** y el **fragment shader**. Ambos se escriben en GLSL, de forma que se le especifica el código GLSL a WebGL y este se ejecuta en la GPU. A continuación se explicarán las principales diferencias entre estos dos componentes.

El *vertex shader* se ejecuta una vez por cada vértice de la figura, su misión es transformar la coordenada de mundo de dicho vértice en coordenadas normalizadas en el intervalo $[-1, 1]$, rango utilizado por WebGL en su *clip space* (Imagen 5.1). Tras realizar estos cálculos y ajustes, se almacena el valor de salida en la variable `gl_Position`. También podemos utilizar el vertex shader para otros cometidos como calcular la coordenada de textura de un objeto, calcular la normal a un objeto en dicho vértice para posteriormente aplicar algún modelo de iluminación, o cualquier otro procesado que podamos hacer en un vértice con la idea de posteriormente pasarlo dicho valor al *fragment shader*.



Clipspace

Imagen 5.1: *Clip space de WebGL*

Por su parte, el *fragment shader*, que es en el que nos centraremos mayormente, es un programa cuyo código se ejecuta una vez por cada píxel y siempre después de que se ejecute el *vertex shader*. Su objetivo es determinar el color del píxel en cuestión en función de la escena que queramos dibujar, posiblemente aplicando un modelo de iluminación a los objetos que la componen.

El conjunto formado por el *vertex shader* y el *fragment shader* es conocido como **shader program**, que comúnmente se refiere al mismo únicamente como *shader*. A partir del código fuente de ambos *shaders*, cada uno se crea y compila por separado, para seguidamente unirse en único programa. En el siguiente código se puede ver este procedimiento, siendo el objeto `shaderProgram` el objeto que representa el *programa shader*.

```

1 // vsSource: Código fuente del Vertex Shader
2 // fsSource: Código fuente del Fragment Shader
3
4 const vertexShader = loadShader(gl, gl.VERTEX_SHADER,
5                               vsSource);
6 const fragmentShader = loadShader(gl, gl.FRAGMENT_SHADER,
7                                   fsSource);
8
9 // Creamos el programa shader
10 const shaderProgram = gl.createProgram();
11 gl.attachShader(shaderProgram, vertexShader);
12 gl.attachShader(shaderProgram, fragmentShader);
```

```
13 | gl.linkProgram(shaderProgram);
```

Además, GLSL utiliza tres tipos especiales de variables además de las propias variables locales que se definen en el programa, cada una con su cometido, que procedemos a explicar.

- **Variables ‘attribute’**: Sólo están disponibles en el *vertex shader* y en el código de JavaScript, desde el cual se les da valor. Se suelen utilizar para almacenar información de color, coordenadas de textura, o en general cualquier información que deba ser compartida entre el código de JavaScript y el *vertex shader*.
- **Variables ‘varying’**: Son declaradas por el *vertex shader* y son utilizadas para enviar información desde el *vertex* para el *fragment shader*, de manera que la información se interpola. Por ejemplo, si el vertex shader asocia color negro a un vértice en una variable *varying* y blanco a otro vértice en la misma variable, entonces los píxeles situados entre estos dos vértices tendrán en esa variable un tono de gris.
- **Variables ‘uniform’**: Estas variables se definen por el código de JavaScript y se puede acceder a ellas tanto en el *vertex* como en el *fragment shader*. Se usan para especificarle a los shaders valores que no cambian independientemente del vértice o del píxel que se esté ejecutando. Por ejemplo, el color de un material o el zoom que se está aplicando a la escena.

Vemos por tanto que mediante estas variables podemos intercambiar información entre el código GLSL que se ejecuta en GPU y el código de JavaScript que comanda el uso de WebGL. Sin embargo, claro está que debe de haber alguna forma de transferir esa información desde JavaScript hasta la GPU. De eso mismo se encargan las estructuras conocidas como *buffers*, que procedemos a explicar.

5.2.3. Los *Buffer*

En el sentido más general de la palabra, un *buffer* es una memoria de almacenamiento temporal de información que permite transferir los datos entre unidades funcionales con características de transferencia diferentes. En nuestro contexto, existen estructuras que almacenan las variables definidas en JavaScript y se transfieren como variables **attribute** o **uniform** al programa *shader*. Por ejemplo, en el siguiente código creamos un buffer que almacena los valores que componen un array de JavaScript de 16 elementos agrupados de 4 en 4, cada grupo representando un color RGBA para un vértice.

```
1 | const colors = [
2 |   1.0, 1.0, 1.0, 1.0,      // blanco
3 |   1.0, 0.0, 0.0, 1.0,      // rojo
4 |   0.0, 1.0, 0.0, 1.0,      // verde
5 |   0.0, 0.0, 1.0, 1.0,      // azul
6 | ];
7 | // Creacion del Buffer
8 | const colorBuffer = gl.createBuffer();
9 | // Seleccionamos el buffer para las proximas operaciones
10 | gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
11 | // Asignamos el array 'colors' al buffer
```

```

12 | gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors),
|     gl.STATIC_DRAW);

```

Y una vez creados los buffers y el *shader* por separado, debemos especificarle al buffer que transfiera la información que contiene a las posiciones de memoria que tiene el *shader* reservadas para las variables que espera. Supongamos que hemos creado en un *vertex shader* una variable `attribute vec4 aVertexColor;` de forma que queremos asociar los colores del código anterior a esta variable por cada uno de los 4 vértices. Mostramos a continuación la forma de hacerlo, pudiendo encontrar más información clarificadora en la documentación de la clase [WebGLRenderingContext](#).

```

1 const location = gl.getAttribLocation(
2     shaderProgram, 'aVertexColor');
3 const numComponents = 4;      // Number of vertex
4 const type = gl.FLOAT;        // GLSL type of the vars
5 const normalize = false;      // Do not normalize
6 const stride = 0;            // Stride
7 const offset = 0;             // offset
8
9 // Seleccionamos el buffer
10 gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
11 gl.vertexAttribPointer(
12     location,
13     numComponents,
14     type,
15     normalize,
16     stride,
17     offset);
18 gl.enableVertexAttribArray(location);

```

De forma muy similar, pero con las funciones correspondientes, se procede desde JavaScript para darle valor a las variables tipo `uniform`.

Tras crear y compilar el programa *shader* y dotarlo de valores para sus variables `attribute` y `uniform`, procedemos a graficar la escena completa con la función `drawArrays`.

```

1 const offset = 0;
2 const vertexCount = 4;
3 gl.drawArrays(gl.TRIANGLE_STRIP, offset, vertexCount);

```

5.3. Primera imagen generada

Una vez conocemos los componentes principales de WebGL, es momento de utilizarlos para crear alguna imagen.

A modo de ejemplo, y aprovechando las situaciones concretas que se han explicado a lo largo del capítulo, supongamos que queremos dibujar en el canvas un cuadrado de colores. Para ello, necesitamos 4 vértices, y para cada vértice su posición y un color, de forma que necesitamos variables `attribute` en el *vertex shader* que representen posición y color. Además, dicho color queremos que se interpole en cada píxel a partir del color de los vértices, para ello usaremos una variable `varying` a la que le asociaremos el color del vértice en el *vertex shader* y recibirá

el color del pixel en el *fragment shader* (es decir, recibirá del vertex shader precisamente lo que tiene que devolver).

```
1 | attribute vec4 aVertexPosition;  
2 | attribute vec4 aVertexColor;  
3 |  
4 | varying lowp vec4 vColor;
```

Crearemos un buffer para almacenar las posiciones de los cuatro vértices y otro para sus colores. Éste último es el que se ha mostrado en el código de ejemplo de la sección 5.2.3. Servimos al shader de valores a partir de los buffer y el resultado es el mostrado en la imagen 5.2.

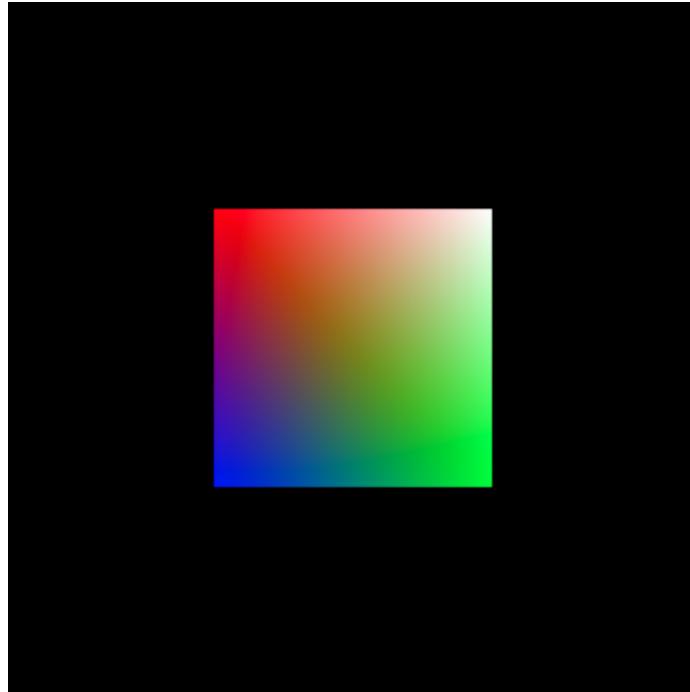


Imagen 5.2: Cuadrado de colores graficado con WebGL

Fíjese cómo hay un vértice blanco, uno rojo, uno verde y uno azul, de forma que en los píxeles intermedios hay colores intermedios entre estos, fruto de la interpolación que se realiza entre *vertex* y *fragment shader*.

Este ejemplo se corresponde al expuesto en [30], cuya adaptación podemos encontrar en el repositorio de este trabajo, concretamente en <https://github.com/JAntonioVR/Geometria-Fractal/blob/main/static/js/canvas.js>. Podemos observar en la imagen 5.3 un esquema de los componentes de WebGL y cómo se relacionan entre ellos hasta finalmente visualizar la imagen deseada.

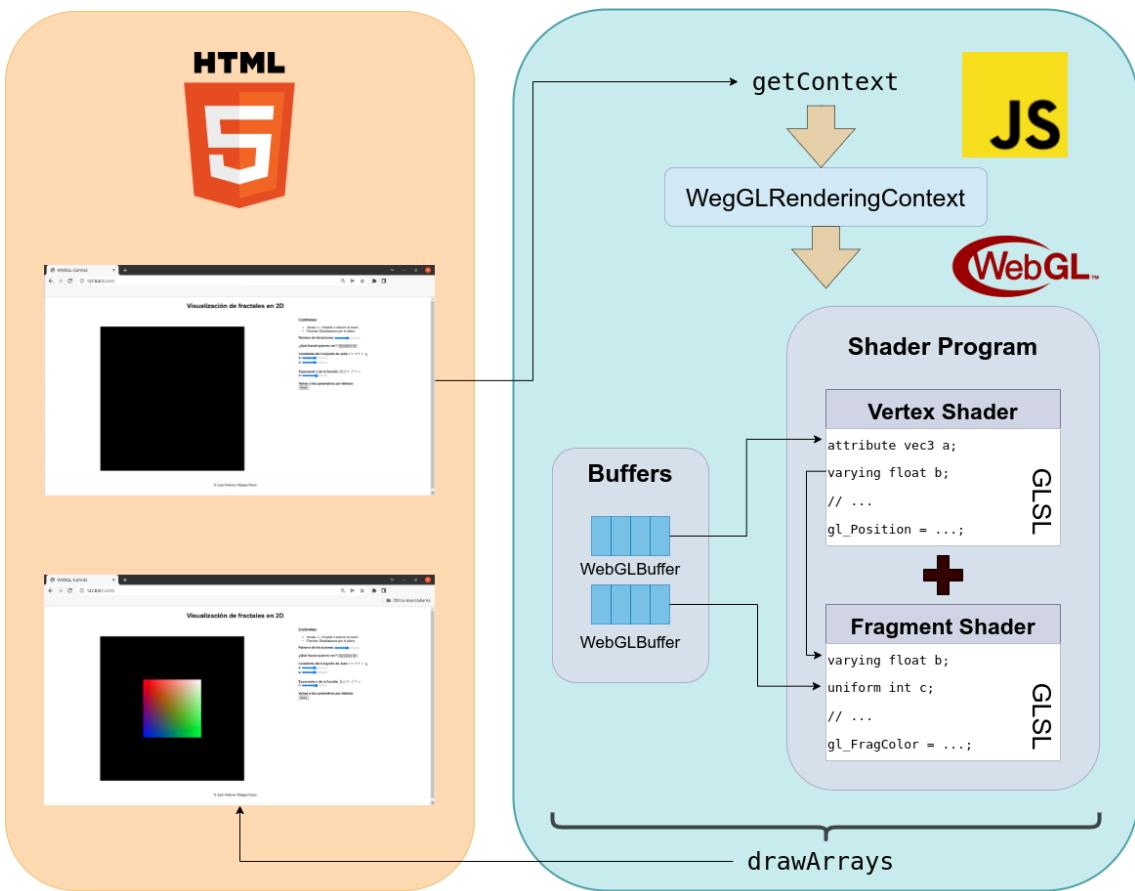


Imagen 5.3: Componentes de WebGL e interacciones entre ellos

CAPÍTULO 6

PLANIFICACIÓN Y ANÁLISIS DEL SOFTWARE

Hasta el momento hemos introducido la materia matemática básica de esta disciplina y las herramientas gráficas utilizadas en informática gráfica, con énfasis en WebGL. Con el objetivo de familiarizar al usuario de internet con los conjuntos de Julia y Mandelbrot en dos dimensiones, estudiados en el capítulo 3, y con el de conseguir nuestras primeras visualizaciones de fractales tridimensionales es momento de comenzar el desarrollo del producto software. Como se ha dicho en la introducción y en otras partes de esta memoria, este producto consiste en una web interactiva en la que poder graficar distintos fractales, tanto 2D como 3D, permitiendo modificar algunos de sus parámetros de manera dinámica.

6.1. Planificación

A continuación describiremos la planificación temporal por fases de este producto software. Debe tenerse en cuenta que con cada avance hay asociado un tiempo de codificación de la funcionalidad, pero también un tiempo buscando información, consultando bibliografía y redactando la documentación del software. El desarrollo comienza a la par que el periodo lectivo correspondiente al segundo cuatrimestre del curso universitario 2021–2022, por lo que dividiremos el tiempo que transcurre desde el 7 de febrero de 2022 hasta la fecha de entrega, que es el 20 de junio de 2022, en 20 semanas, cada una identificada por la fecha del lunes de dicha semana. Utilizaremos estas unidades y fechas para describir los tiempos empleados en cada fase de desarrollo, las cuales procedemos a describir en las siguientes subsecciones.

6.1.1. Infraestructura básica

La primera fase de desarrollo comienza en la segunda semana de febrero de 2022, que da comienzo el día 7 de este mismo mes. El primer paso es conseguir una web vacía con un elemento tipo ‘canvas’ en el cual visualicemos algo sencillo, como por ejemplo un cuadrado de colores, que es el que encontramos en la figura 5.2.

Esta tarea es sencilla, ya que hay muchos tutoriales, es el ‘hello world’ de las aplicaciones gráficas. Debe estar lista en esta misma semana, es decir, antes de acabar la **semana 1**, la cual transcurre desde el 7 hasta el 13 de febrero. Con esto, ya tenemos la infraestructura básica del

producto software.

6.1.2. Visualización de fractales 2D

Con el código hasta ahora, debemos modificar la estructura para que el software grafique los primeros fractales 2D de forma estática. Primero bastará con graficarlos en blanco y negro para posteriormente añadir colores.

Seguidamente, debemos preparar el documento web para el soporte de interactividad y gestión de eventos. Es decir, añadir formularios, botones, deslizadores... en principio sin funcionalidad alguna. Hecho esto, añadimos definitivamente la interactividad, permitiendo que a partir del documento web se pueda interactuar con el canvas y la escena que se está visualizando. Con cada avance en el desarrollo debe cuidarse la apariencia y el estilo de la web.

En este punto, el producto consiste en una web interactiva para visualizar fractales en 2D. Esta tarea no es tampoco demasiado compleja al ser sencillo identificar un canvas 2D con una región del plano, pero requiere mucha funcionalidad muy distinta, la cual es la primera vez que se programa, y teniendo en cuenta que es una parte importante del proyecto que servirá de base para el resto de fases, estipulamos que debemos dar como margen hasta la **semana 8**, es decir, entre el lunes 7 de febrero y el domingo 3 de abril de 2022.

6.1.3. Construcción de un programa ray-tracer

Comenzando con la parte tridimensional del software, aprovechamos gran parte del código ya desarrollado para la escena 2D, pues realmente el documento web puede tener la misma forma y la interacción con sus elementos es similar, aunque adaptada a los distintos parámetros que admite una escena 3D como son la posición o la iluminación.

El primer paso para tener una web en la que visualizar una escena con fractales es empezar graficando cuerpos sencillos utilizando ray-tracing. Por tanto, lo primero que haremos será un pequeño programa que mediante ray-tracing visualice una escena sencilla con esferas y un plano (a modo de suelo) que nos permita movernos libremente por el espacio. Aplicaremos también un modelo de iluminación local sencillo para aportar un poco de realismo a la escena.

A la par, debemos seguir cuidando tanto el estilo del documento web como la posibilidad de interactuar con la página. Por suerte, a partir del código ya desarrollado en la anterior fase, esta tarea se facilita considerablemente respecto a las primeras veces.

Este programa requiere programar el propio ray-tracing, formas de calcular las intersecciones de los rayos con los objetos, llenar de objetos la escena, parametrizar la posición de la cámara e implementar el modelo de iluminación. La tarea es generalmente compleja, pero está muy basada en el código ya implementado para fractales 2D, por lo que estimamos que debe estar programada para la **semana 12**, que finaliza el domingo 1 de mayo de 2022.

6.1.4. Visualización de fractales en 3D

Esta aunque no es la última fase del desarrollo del producto, probablemente sí sea la más importante y la más compleja. Debemos modificar el programa ray-tracer del apartado anterior con el objetivo de visualizar en la escena imágenes de fractales 3D.

Esta fase requiere, además de código, mucho tiempo de búsqueda y consulta bibliográfica. Una vez adquiridos conocimientos necesarios haremos las modificaciones oportunas a las regiones

de código necesarias al programa que tenemos hasta ahora con el objetivo de visualizar los fractales en 3D. Al igual que en todas las fases, debemos ir añadiendo parámetros modificables a la web y cuidar el estilo.

Una vez realizado esto, contaría con una web en la que interactuar con fractales 2D o 3D, pudiendo visualizar de forma concreta y sencilla estas singulares figuras. Esto finalizaría la parte más compleja y también la más necesaria del proyecto, pues es la que cumple el objetivo fundamental. Estipulamos por tanto que debe estar lista en la **semana 15**, es decir, antes del domingo 22 de mayo.

6.1.5. Realismo y optimización en la escena 3D

Aunque la funcionalidad está completa y el objetivo cumplido, es bueno pulir algunos detalles relacionados con el realismo. Por ejemplo, tanto en las imágenes 2D como 3D podemos aplicar la técnica conocida como ‘Super Sampling Antialiasing’ para suavizar los bordes y conseguir una mayor calidad de imagen.

También podemos añadir al modelo de iluminación de la escena 3D la visualización de sombras arrojadas, consiguiendo efectos más realistas.

El problema es que estas dos adiciones son muy costosas en tiempo, lo cual compromete la velocidad de ejecución, por lo que añadiremos a la web la posibilidad de elegir si se desean aplicar o no.

Seguidamente, podemos añadir a la escena 3D esferas englobantes como técnica de optimización, al menos en escenas donde el fractal esté relativamente lejano.

Estas modificaciones completarían el desarrollo software de la visualización e interacción de la escena y en lo que a ‘backend’ se refiere completa el software. Debe estar lista durante la **semana 18**, que finaliza el domingo 12 de junio.

6.1.6. Añadir estilo y UX a la web

Desde el inicio, al estar en fases de desarrollo, se suele descuidar bastante la apariencia de la página en lo que a secciones, fuentes y colores se refiere. Por tanto, añadiremos estilo y código CSS al documento para darle una apariencia más visual y amigable. Para ello nos ayudaremos del framework de CSS y JavaScript ‘Bootstrap’.

La última, pero no por ello menos importante fase del desarrollo debe estar finalizada la misma semana que se produce la entrega, es decir, la **semana 20** y por supuesto antes del día **20 de Junio de 2022**.

En conclusión, y a modo de resumen, presentamos en la imagen 6.1 el conocido como *diagrama de Gantt*, que sintetiza en una tabla el tiempo dedicado a cada actividad. Cada columna representa una semana, identificada por el día y el mes del lunes de dicha semana.

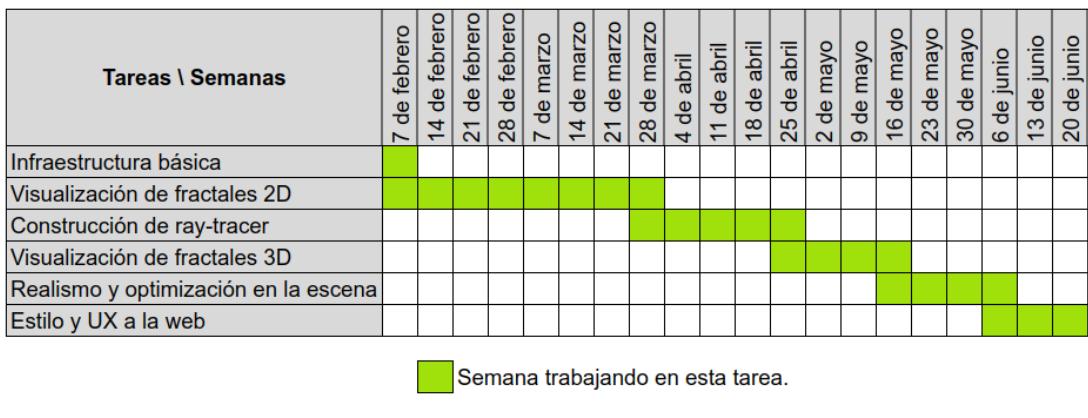


Imagen 6.1: Diagrama de Gantt de la planificación del desarrollo del producto software

6.2. Presupuesto del producto software

A continuación haremos una estimación del presupuesto de este producto. Para ello desglosaremos los distintos recursos utilizados en: recursos hardware, recursos software y recursos humanos. Estimaremos el coste de cada uno de ellos y presentaremos un presupuesto final.

6.2.1. Recursos hardware

Todo el software ha sido desarrollado en un ordenador portátil personal. Cualquier ordenador moderno con una GPU estándar a fecha de junio de 2022 sería capaz de ejecutar y programar el software con suficiente fluidez. Estimamos que unos **500€** son suficientes para cubrir las necesidades, aunque cuanto mejor sea el ordenador y su GPU (y por tanto más caros) mejor rendimiento nos proporcionará.

Buenas opciones serían este [Lenovo V15 G2 ALC AMD Ryzen 3 5300U/8GB/256GB SSD/15.6"](#) o este [HP 15S-fq2158ns Intel Core i3-1115G4/8GB/256GB SSD/15.6"](#).

6.2.2. Recursos Software

A continuación enumeraremos los recursos software utilizados en la codificación y en la documentación.

- **Visual Studio Code:** Editor de código utilizado tanto para redactar todo el código fuente necesario como la memoria, gracias a sus múltiples extensiones que lo convierten en un entorno de desarrollo muy versátil.
- **Google Chrome:** Navegador web, ampliamente utilizado para visualizar la web que se ha desarrollado en cada una de sus fases. También se ha empleado para consultar bibliografía, buscar información o realizar muchas de las imágenes utilizadas. En este último aspecto, los sitios más visitados han sido [Vectary](#) y [Figma](#) para las imágenes explicativas sobre ray-tracing junto con [app.diagrams.net](#) y [Visual Paradigm Online](#) para hacer los diagramas y algunas imágenes.
- **Git y Github:** Sistema de control de versiones, gracias a git se ha permitido llevar un histórico de los avances que se han hecho en el trabajo. Además, con ‘Github pages’ se ha podido desplegar la web en internet gratuitamente.

- **Bootstrap**: Framework de CSS y JavaScript útil para la programación del estilo de la página web.
- **jQuery**: Biblioteca de JavaScript que simplifica la sintaxis, la gestión de eventos y la manipulación del DOM.

Todo esto junto con las aplicaciones tradicionales como el navegador de archivos, la terminal, etc.

Afortunadamente, todos los recursos software son gratuitos, por lo que el presupuesto de los recursos software se reduce a **0€**.

6.2.3. Recursos humanos

Sobre recursos humanos, el producto ha sido en su totalidad desarrollado por el alumno y autor de este texto: Juan Antonio Villegas Recio. Siempre bajo la supervisión y ayuda del tutor de informática: D. Carlos Ureña Almagro.

Si hacemos una estimación de unas 20 horas semanales de media, teniendo en cuenta que el desarrollo ha durado 20 semanas, lo cual supone un total de 400 horas, a régimen de 15€ por cada hora de trabajo, en total el gasto en recursos humanos supone **6.000€**.

6.2.4. Presupuesto final

Presentamos entonces el desglose final del presupuesto de este proyecto en la tabla 6.1.

Recurso	Precio
Recursos Hardware	500€
Recursos Software	0€
Recursos Humanos	6000€
Coste total	6500€

Tabla 6.1: Desglose por recursos del presupuesto final

Por tanto, el presupuesto de este producto software es de **6500€**.

6.3. Análisis de requisitos

Una vez estimada la planificación temporal y el presupuesto del proyecto, es momento de definir correctamente los requisitos y la funcionalidad que ofrecerá el producto. Para ello, haremos un análisis de requisitos del problema, estableceremos sus casos de uso y veremos mediante un diagrama las interacciones entre los mismos.

Tras un estudio profundo del problema y evaluando la viabilidad de los mismos, introducimos los requisitos, funcionales y no funcionales, de este producto:

6.3.1. Requisitos funcionales

En caso de la visualización de fractales 2D:

- **RF1.1**: Visualización de conjuntos de Julia y Mandelbrot 2D.

- **RF1.2:** Posibilidad de alternar qué fractal ver en el canvas: Julia o Mandelbrot.
- **RF1.3:** Posibilidad de cambiar el exponente m de la función $P_{c,m}(z) = z^m + c$ para poder visualizar conjuntos de Julia y Mandelbrot generalizados.
- **RF1.4:** Posibilidad de cambiar la constante c de la función $P_{c,m}(z) = z^m + c$ en el caso de los conjuntos de Julia para poder visualizar diferentes conjuntos de Julia \mathcal{J}_c para distintos $c \in \mathbb{C}$.
- **RF1.5:** Poder movernos libremente por el plano.
- **RF1.6:** Hacer zoom de forma que podamos observar las infinitas irregularidades que presentan los fractales.
- **RF1.7:** Modificar los parámetros que acepten los algoritmos utilizados para visualizar los fractales para ver cómo cambia la imagen al cambiar dichos parámetros.

En caso de fractales 3D:

- **RF2.1:** Visualización de generalizaciones tridimensionales de los conjuntos de Julia y Mandelbrot.
- **RF2.2:** Posibilidad de distinguir dos modos de uso debidos a la complejidad de la generación de imágenes 3D: redibujado continuo (a tiempo real) y redibujado a demanda con un botón.
- **RF2.3:** Posibilidad de alternar qué fractal 3D ver en el canvas en cada momento.
- **RF2.4:** Posibilidad de modificar parámetros del modelo de iluminación para así observar distintos materiales y coloraciones en los fractales.
- **RF2.5:** Modificar parámetros de los algoritmos de visualización que empleemos.
- **RF2.6:** Posibilidad de moverse libremente por la escena utilizando una cámara orbital.
- **RF2.7:** Hacer zoom en la escena para poder observar de cerca las superficies de los fractales.

De forma genérica:

- **RF3:** Botón de reseteo para restablecer los valores de los parámetros por defecto.
- **RF4:** Posibilidad de cambiar rápidamente entre visualización 2D y 3D.

6.3.2. Requisitos no funcionales

- **RNF1:** El tiempo de procesado no debe ser demasiado largo. A ser posible se ejecutará en tiempo real.
- **RNF2:** Deben utilizarse herramientas portables, es decir, que puedan ejecutarse en cualquier navegador sin necesidad de instalar nada.

- **RNF3:** La interfaz de usuario tiene que ser sencilla, para que pueda utilizarla cualquier persona.
- **RNF4:** El software debe poder ser ejecutado en cualquier ordenador estándar. Por tanto, las técnicas utilizadas no pueden requerir un hardware demasiado avanzado ni demasiado caro.
- **RNF5:** Es necesario incluir cierta documentación en la web para que el usuario con pocos conocimientos sepa identificar el uso de cada parámetro.
- **RNF6:** Deben usarse colores e interfaces que cuiden la experiencia de usuario.
- **RNF7:** No será posible elegir valores de parámetros imposibles o que requieran una cantidad inviable de cálculo.

6.3.3. Requisitos de datos

No existen requisitos de datos, ya que no hay realmente ninguna base de datos ni es necesaria, pues no es una aplicación que almacene datos, sino que realiza cálculos para graficar imágenes.

6.4. Análisis de casos de uso

A continuación describiremos los posibles casos de uso del software a desarrollar:

- **CU1:** Primer acceso a la página. El usuario accede por primera vez a la página.
- **CU2:** Modificar un parámetro. El usuario cambia uno de los distintos parámetros modificables.
- **CU3:** Comprobar si un nuevo parámetro es incorrecto o inviable. Ante el cambio de un parámetro, es necesario comprobar que no se ha introducido un valor imposible o computacionalmente inviable. Por ejemplo, un número negativo o demasiado grande de iteraciones.
- **CU4:** Truncar o modificar valores de los parámetros. Ante la introducción de valores incorrectos, el sistema por sí solo modifica estos valores estableciendo unos límites o valores por defecto.
- **CU5:** Establecer valores por defecto. Cambiar los parámetros de la escena a los que se toman por defecto.
- **CU6:** Dibujar la escena. Visualizar la escena a partir de los parámetros actuales, estando seguros de que son correctos.

Ante estos casos de uso, en la figura 6.2 vemos el diagrama de casos de uso, en el cual vemos a cuales de ellos puede acceder el usuario y qué interdependencias hay entre los mismos.

Por último, con el objetivo de clarificar las distintas actividades que puede realizar tanto el software como el usuario, presentamos en la imagen 6.3 un diagrama de actividad, en el cual podemos observar cómo tras la generación de una imagen el usuario puede continuar indefinidamente variando parámetros hasta que finalmente decida cerrar la página.

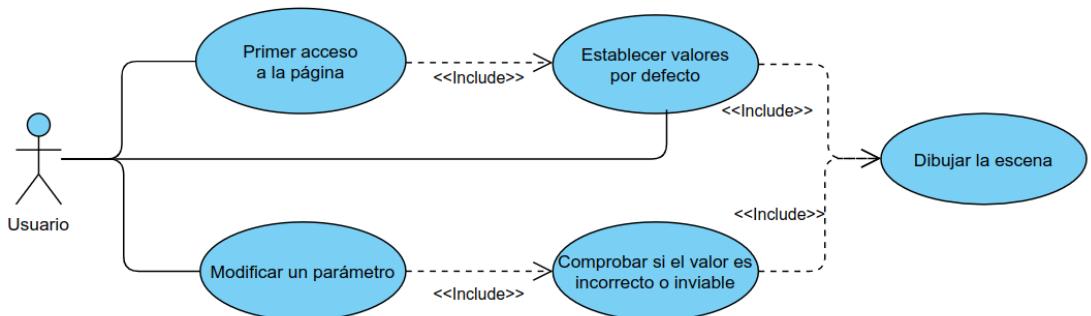


Imagen 6.2: Diagrama de Casos de Uso

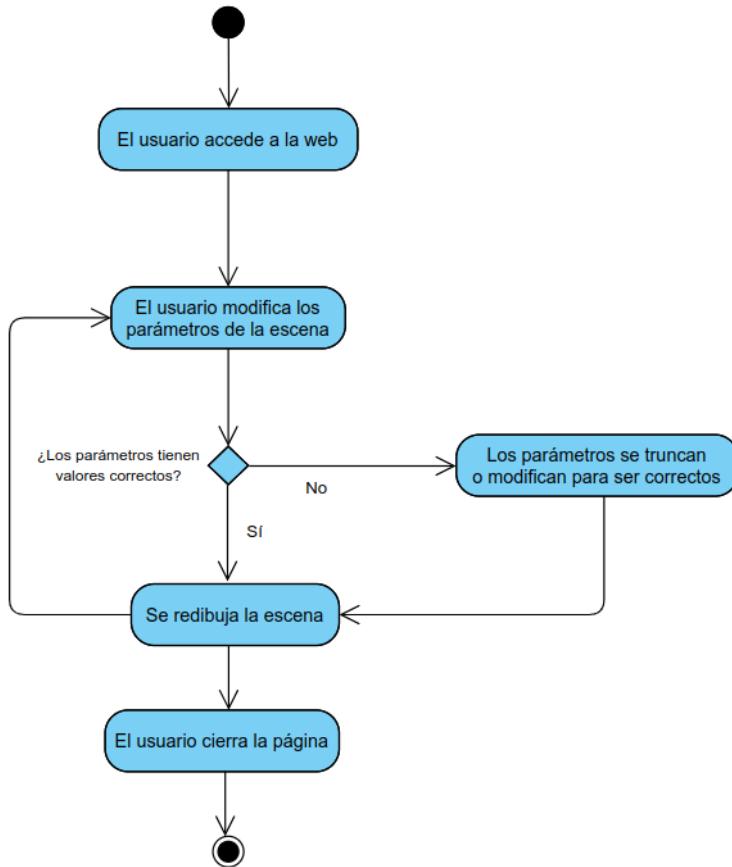


Imagen 6.3: Diagrama de Actividad

6.5. Bocetos de la web

A continuación presentaremos los bocetos básicos, a veces llamados ‘wireframes’, de la apariencia que tendrá la web. Son únicamente tres pantallas, y dos de ellas son prácticamente iguales en lo que a nivel de boceto se refiere. La pantalla en la que se visualizan fractales 2D y 3D en un canvas (imagen 6.5) es prácticamente igual en ambos casos, por lo que sólo hemos incluido una de ellas entendiendo que los enlaces y los títulos son intercambiables en cada versión.

Como vemos en la imagen 6.4, al acceder a la web nos aparecería la posibilidad de elegir si

queremos explorar el mundo de los fractales 2D o 3D. Debajo del menú seleccionable aparecería una breve introducción al mundo de los fractales desde un punto de vista entendible por casi cualquier persona con conocimientos básicos.

Sobre la pantalla de la imagen 6.5, aparecería a la izquierda el canvas donde se visualizarían los fractales y los distintos controles y parámetros modificables estarían en el lado derecho. Debajo de esta sección de la pantalla también se incluiría un texto documentando el sentido de cada parámetro para que el usuario que no conozca demasiado el trasfondo matemático ni informático pueda interactuar sin problemas.

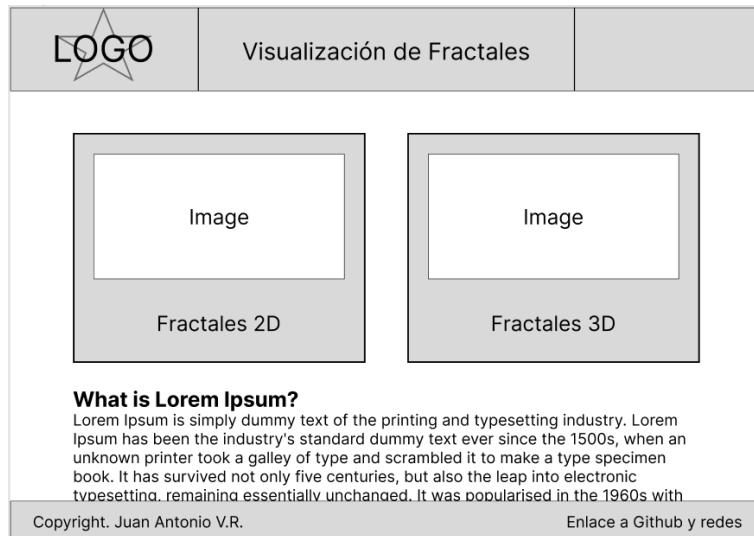


Imagen 6.4: Wireframe de la portada de la página

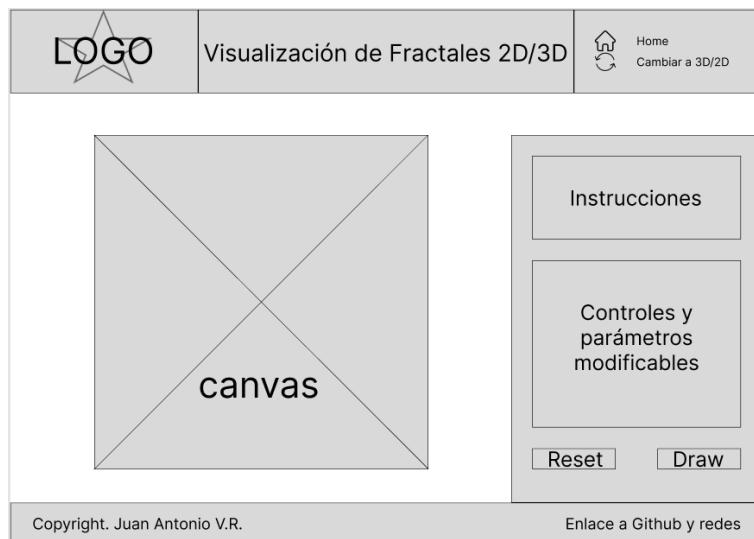


Imagen 6.5: Wireframe de la pantalla en la que se visualizan fractales

CAPÍTULO 7

VISUALIZACIÓN DE FRACTALES EN 2D

En el capítulo 5 introdujimos el uso de WebGL como herramienta de generación de imágenes y estudiamos sus componentes, sin embargo, no olvidemos que nuestro objetivo es la visualización de fractales, donde la característica principal de los mismos es que no se pueden expresar a partir de un conjunto de vértices o líneas, sino que son curvas o superficies totalmente irregulares. Entonces, ¿cómo podemos usar WebGL para graficar imágenes fractales?

A nivel conceptual, una imagen realmente es un conjunto de píxeles con un color asociado. Si identificamos cada píxel con la dupla de números naturales (x, y) formada por el número de fila y de columna que ocupa el píxel, entonces a partir de una función f que a cada pareja le asigne un color, esto es, una terna (r, g, b) , podemos obtener una imagen. Fíjese en el ejemplo de la imagen 7.1, donde generamos una imagen de 4×2 píxeles a partir de una función f . Si queremos implementar esta síntesis de imágenes mediante una función f , podemos programar esta función f y ejecutarla en CPU píxel a píxel de manera secuencial, pero evidentemente sería más rápido y eficiente paralelizar el cálculo de f aprovechando la GPU, de forma que en un mismo instante se pueden estar procesando miles de píxeles en lugar de uno.

$f(0, 1)$	$f(1, 1)$	$f(2, 1)$	$f(3, 1)$
$f(0, 0)$	$f(1, 0)$	$f(2, 0)$	$f(3, 0)$

Imagen 7.1: Síntesis de una imagen vía una función

Sin embargo, WebGL está orientado a rasterización, es decir, dadas unas primitivas colorea del color adecuado los píxeles que ocupen estas primitivas, dividiendo las mismas en triángulos. La forma que tenemos por tanto de ejecutar f una vez por cada píxel es rasterizando dos triángulos que ocupen toda la superficie del canvas. Al existir dos primitivas que en conjunto cubren toda la imagen, el fragment shader se ejecutará una vez por cada píxel. Desde el fragment shader podemos acceder a las coordenadas del píxel, y utilizando estas coordenadas y los valores asignados a las variables `varying` y `uniform`, calcular el valor que devuelve la función f ,

asignando así un color a cada píxel, que es precisamente el cometido del fragment shader.

Por tanto, a efectos prácticos, nuestro vertex shader tomará como entrada los vértices $(-1, -1)$, $(1, -1)$, $(1, 1)$ y $(-1, 1)$ y no aplicará ninguna transformación, pues ya están normalizados en el clip space (teniendo en cuenta que estaríamos visualizando un fragmento del plano $z = 0$). Cabe en este momento aclarar que en el ámbito de herramientas de visualización por convenio se suele utilizar la coordenada Y para la altura y la coordenada Z para la profundidad.

A partir de estos cuatro vértices, en el canvas se visualizarán dos triángulos que completarán la superficie completa del mismo. El vertex shader a partir de ahora será totalmente trivial, pues solo devolverá en la variable `gl_Position` la misma posición que obtiene del buffer de posición.

```
1 | attribute vec2 a_Position;
2 | void main() {
3 |     gl_Position = vec4(a_Position.x, a_Position.y, 0.0, 1.0);
4 | }
```

Mientras que, por su parte, el fragment shader podrá acceder a la posición (en coordenadas de dispositivo) del píxel que se está ejecutando mediante la variable `gl_FragCoord` y a partir de estas coordenadas devolver un color en la variable `gl_FragColor`. Es decir, estamos dibujando una escena completa, próximamente un fractal, en dos triángulos. Por ejemplo, las imágenes 3.1 y 3.2 son el resultado de esta metodología.

7.1. Estructurando el código

Con el objetivo de utilizar WebGL para generar imágenes fractales, podemos usar como base el código utilizado para visualizar el cuadrado de colores del capítulo 5, ya que nos puede venir bien su estructura para adaptar la misma a la visualización de fractales. Sin embargo, esta estructura es muy procedural. Es posible mantener la misma arquitectura de forma que cambiando los elementos que sean necesarios y el código de los shaders podamos ver los fractales que deseemos, pero en ese caso la depuración se complicaría, el código es más difícil de leer y cuesta mucho añadir interactividad. Por este motivo, adaptaremos el código a un paradigma orientado a objetos, modularizando los distintos componentes, creando abstracciones de las herramientas que proporciona WebGL.

En concreto, para el código de JavaScript hemos creado las siguientes clases:

- **Scene2D**: Inicializa y gestiona todos los componentes de WebGL necesarios para representar una escena: el contexto WebGL, los parámetros, el shader, las variables del shader y los buffers. Cuenta además con métodos getter y setters de cada uno de los parámetros de la escena y un método `drawScene` que grafica la escena a partir de los parámetros actuales.
- **ShaderType**: Es un enumerado que puede tomar los valores `vertexShader` o `fragmentShader`.
- **Shader**: Representa una abstracción de lo que sería un ‘vertex shader’ o un ‘fragment shader’. A partir del código fuente y del tipo de shader, se compila y almacena el shader de WebGL en un atributo.

- **ShaderProgram**: A partir de dos objetos de la clase **Shader**, que serían el vertex shader y el fragment shader, crea el programa shader final.
- **Buffer**: Construye un buffer de WebGL a partir de un array de JavaScript que se le pasa como parámetro.

Además, el fichero **fractals-2D.js** utiliza un objeto de la clase **Scene2D** para interactuar con el DOM, gestionar los eventos y así poder modificar dinámicamente los parámetros, de tal forma que cada vez que se interactúa con la página se registra un evento con una función manejadora asociada. Esta función manejadora utiliza los setters correspondientes del objeto **Scene2D** y hace las modificaciones correspondientes para finalmente llamar a **drawScene** y así cambiar dinámicamente la visualización.

El código completo correspondiente a la funcionalidad desarrollada en este capítulo y en los siguientes se puede encontrar en GitHub: <https://github.com/JAntonioVR/Geometria-Fractal/tree/main/static>. La web interactiva ya terminada correspondiente a la visualización de fractales en 2D puede visitarse en <https://jantoniov.github.io/Geometria-Fractal/2D-fractals.html>. Y la funcionalidad completa de JavaScript detallada de cada clase creada para éste y próximos capítulos puede consultarse en el apéndice A.

Una vez estructurado el código en JavaScript, es bueno recordar que, como dijimos al inicio de este capítulo, nuestro vertex shader es trivial, simplemente asigna a **gl_Position** las mismas coordenadas que se introducen desde JavaScript al buffer de posiciones. Es en el fragment shader donde se realiza el grueso de la programación necesaria para poder visualizar los distintos fractales. Por este motivo, las siguientes secciones estarán implícitamente dedicadas a explicar la implementación en GLSL del fragment shader, el cual se puede encontrar en el fichero **fragment-shader-2D-fractals.js**, disponible en <https://github.com/JAntonioVR/Geometria-Fractal/blob/main/static/glsl/fragment-shader-2D-fractals.js>.

7.2. Identificando la pantalla con el plano

Recordemos que el fragment shader se ejecuta una vez por cada píxel, de manera que podemos identificar la superficie completa del canvas con una región $[x_1, x_2] \times [y_1, y_2] \subseteq \mathbb{R}^2 \cong \mathbb{C}$ del plano complejo y en particular cada píxel con un número complejo. Para ello, necesitamos transformar las coordenadas de dispositivo que el shader encuentra en la variable **gl_FragCoord**. Supongamos que tenemos un canvas de $n_x \times n_y$ píxeles, y en el mismo queremos graficar una región del plano centrada en el origen de w unidades de ancho y h de alto, con w, h fijos. es decir, la región $[-\frac{w}{2}, \frac{w}{2}] \times [-\frac{h}{2}, \frac{h}{2}]$. Es muy recomendable que se guarde la proporción $\frac{n_x}{n_y} = \frac{w}{h}$ para así evitar posibles deformaciones.

Las coordenadas de dispositivo son por tanto $(x, y) \in [0, n_x] \times [0, n_y]$, las cuales están precisamente en la región $[0, n_x] \times [0, n_y]$ porque las dimensiones del canvas son $n_x \times n_y$ píxeles. Entonces la transformación lineal que necesitamos es

$$\begin{aligned}\phi : [0, n_x] \times [0, n_y] &\longrightarrow \left[-\frac{w}{2}, \frac{w}{2}\right] \times \left[-\frac{h}{2}, \frac{h}{2}\right] \\ (x, y) &\longmapsto \left(\frac{w}{n_x}x - \frac{w}{2}, \frac{h}{n_y}y - \frac{h}{2}\right)\end{aligned}\tag{7.1}$$

De esta forma, a partir de las coordenadas de dispositivo obtenemos un punto del plano complejo. Supongamos ahora que en lugar de querer visualizar la región $[-\frac{w}{2}, \frac{w}{2}] \times [-\frac{h}{2}, \frac{h}{2}]$ con w, h fijos queremos representar cualquier otra, pero aún centrada en el origen $(0, 0)$. Podemos variar los valores de w y h en la transformación (7.1) y construir la transformación según queramos, pero en lugar de ello y para mantener las proporciones introduciremos una variable que represente el *zoom* que se aplica a la imagen. De esta forma, tan solo habría que multiplicar el resultado de la transformación (7.1) por una constante λ . Esta constante será menor que 1 si se desea acercar la región o mayor que 1 si se desea alejar. Por tanto la nueva transformación será

$$\begin{aligned}\phi : [0, n_x] \times [0, n_y] &\longrightarrow \left[-\lambda \frac{w}{2}, \lambda \frac{w}{2} \right] \times \left[-\lambda \frac{h}{2}, \lambda \frac{h}{2} \right] \\ (x, y) &\longmapsto \lambda \left(\frac{w}{n_x} x - \frac{w}{2}, \frac{h}{n_y} y - \frac{h}{2} \right)\end{aligned}\tag{7.2}$$

Y por último, procedemos a buscar la forma de representar cualquier parte del plano centrada o no. Tenemos entonces que fijar un par (x_0, y_0) que sea el centro de la región en la que se hace este posible zoom, que hasta este momento hemos asumido que es el $(0, 0)$, pero a partir de ahora queremos que tome cualquier valor. Aprovechando que la transformación (7.2) nos devuelve coordenadas en regiones centradas simplemente tenemos que sumar este par al resultado, de forma que nos queda

$$\begin{aligned}\phi : [0, n_x] \times [0, n_y] &\longrightarrow \left[x_0 - \lambda \frac{w}{2}, x_0 + \lambda \frac{w}{2} \right] \times \left[y_0 - \lambda \frac{h}{2}, y_0 + \lambda \frac{h}{2} \right] \\ (x, y) &\longmapsto (x_0, y_0) + \lambda \left(\frac{w}{n_x} x - \frac{w}{2}, \frac{h}{n_y} y - \frac{h}{2} \right)\end{aligned}\tag{7.3}$$

En nuestro ejemplo concreto, usaremos un canvas de 1280×720 píxeles, manteniendo así un ratio $r = 1280/720 = 16/9$, que es uno de los más estándar. La región del plano a representar será inicialmente la $[-3.56, 3.56] \times [-2, 2]$, esto es, $h = 4, w = r \cdot h = 7.11$. Sobre esta región inicial aplicaremos zoom y desplazamientos.

En muchos contextos, la transformación lineal (7.3) se suele expresar en términos de una matriz 4×4 , identificando cada coordenada de dispositivo (x, y) con un vector de 4 dimensiones $(x, y, 0, 1)^t$ y cada punto $p = (p_1, p_2) \in \mathbb{R}^2$ con el vector $(p_1, p_2, 0, 1)^t$, de forma que:

$$\phi(x, y) = \begin{pmatrix} \lambda \frac{w}{n_x} & 0 & 0 & -\lambda \frac{w}{2} + x_0 \\ 0 & \lambda \frac{h}{n_y} & 0 & -\lambda \frac{h}{2} + y_0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 0 \\ 1 \end{pmatrix}\tag{7.4}$$

En rasterización, se denomina matriz de vista a aquella que transforma las coordenadas de mundo (WC) de los objetos que componen la escena a coordenadas de cámara, mientras que la matriz de proyección es la que calcula las coordenadas de dispositivo (DC) finales a partir de las coordenadas de cámara. La composición de estas dos matrices es la matriz de vista y proyección M , que convierte las coordenadas de mundo en coordenadas de dispositivo. La transformación ϕ convierte las coordenadas de dispositivo en coordenadas de mundo en el plano complejo, por lo que la matriz de la transformación (7.4) es la inversa de la matriz de vista y proyección.

Las constantes de zoom λ y el centro (x_0, y_0) pueden ser parametrizables para así poder visualizar cualquier región del plano en el canvas. La forma de darle distintos valores a estas constantes es mediante una variable `uniform` para cada caso. Otro detalle a tener en cuenta es que, aunque en todo momento hemos hablado de coordenadas de dispositivo como valores enteros, lo estándar es tomar reales con parte fraccionaria igual a $\frac{1}{2}$, de forma que identificamos el píxel con el punto de \mathbb{R}^2 que se encuentra en el centro del píxel. En caso de mantener la coordenada entera estaríamos identificándolo con la esquina inferior izquierda. Queda por tanto el siguiente fragmento de código:

```

1 // Zoom: constante lambda que define el tamano de la region a
2     representar
3 uniform float u_zoomSize;
4 uniform vec2 u_zoomCenter;
5
6 // ...
7
8 vec2 get_world_coordinates() {
9     int nx = 1280, ny = 720; // Tamano del canvas (pixels)
10    float r = 16.0/9.0;        // Ratio ancho/alto = 16/9
11    float h = 4.0, w = r * h; // Tamano de imagen (unidades)
12    // Coordenadas de dispositivo normalizadas en [0,1]x[0,1]
13    vec2 uv = (gl_FragCoord.xy + vec2(0.5))
14        / vec2(float(nx), float(ny));
15    float u = uv.x;
16    float v = uv.y;
17    // Punto del plano complejo al que corresponde el pixel
18    return u_zoomCenter + vec2(u*w - w/2.0, v*h - h/2.0) *
19        u_zoomSize;
20 }
```

Y ya tenemos en el shader una función que calcula el punto del plano al cual corresponde el píxel. A partir de esto, y al igual que en el capítulo 3, tan solo tenemos que iterar la función $P_{c,m}$ y en función del número de iteraciones necesarias para diverger (o no), asignar un color.

7.3. La función $P_{c,m}(z) = z^m + c$

Necesitamos código para la función $P_{c,m}$, pero esta requiere a su vez la programación de potencias de números complejos. Tal y como se ha evidenciado en el código recientemente presentado, y de manera natural, representaremos un número complejo $z = x + i \cdot y \cong (x, y) \in \mathbb{R}^2$ mediante una variable del tipo `vec2`. Por lo que debemos implementar una función que, de forma iterativa, multiplique (usando el producto de números complejos) m veces por sí mismo una variable `vec2`. Como ya sabemos,

$$z^2 = (\operatorname{Re} z + i \cdot \operatorname{Im} z)(\operatorname{Re} z + i \cdot \operatorname{Im} z) = ((\operatorname{Re} z)^2 - (\operatorname{Im} z)^2) + 2 \cdot \operatorname{Re} z \cdot \operatorname{Im} z \cdot i$$

y como $z^n = z^{n-1} \cdot z$, entonces

$$\begin{aligned} z^n &= z^{n-1} \cdot z = (\operatorname{Re} z^{n-1} + \operatorname{Im} z^{n-1} \cdot i) \cdot (\operatorname{Re} z + \operatorname{Im} z \cdot i) \\ &= (\operatorname{Re} z^{n-1} \cdot \operatorname{Re} z - \operatorname{Im} z^{n-1} \cdot \operatorname{Im} z) + (\operatorname{Re} z^{n-1} \cdot \operatorname{Im} z + \operatorname{Im} z^{n-1} \cdot \operatorname{Re} z) \cdot i \end{aligned} \quad (7.5)$$

y esto es válido para cualquier $n \in \mathbb{N}$. Podemos utilizar iterativamente la ecuación (7.5) para programar un método que calcule potencias complejas:

```

1 // Potencias complejas
2 vec2 complex_pow(vec2 z, int n) {
3     vec2 current_pow = vec2(1.0, 0.0);
4     for (int i = 1; i < 100; i++) {
5         vec2 z_ant = current_pow; // z^i
6         // Calculamos z^{i+1} = z^i * z
7         current_pow = vec2( z_ant.x*z.x - z_ant.y*z.y,
8                             z_ant.x*z.y + z_ant.y*z.x );
9         if(i >= n) break;
10    }
11    return current_pow;
12 }
```

Para el que no esté acostumbrado al código GLSL, este lenguaje no permite iterar bucles utilizando variables, por lo que necesitamos fijar un máximo de iteraciones (en este caso 100) y salir del bucle al alcanzar las `n` iteraciones.

Una vez tenemos esta función, es muy sencillo programar la función $P_{c,m}$ aprovechando la aritmética preprogramada para los objetos vector en GLSL.

```

1 vec2 P(vec2 z, vec2 c, int m) {
2     return complex_pow(z,m) + c;
3 }
```

En el caso de los conjuntos de Julia debemos fijar una constante c a la cual calcularle el conjunto \mathcal{J}_c . Naturalmente esta constante es común a todos los píxeles, por lo que necesitamos una variable `uniform` para desde JavaScript enviar al shader qué conjunto de Julia queremos graficar. Además, también debemos fijar el exponente m que también es común a todos los píxeles, por lo que hacemos uso de otra variable `uniform` al uso.

```

1 // Valor c fijo en la funcion z^m + c
2 uniform vec2 u_juliaSetConstant;
3 // Valor m fijo en la funcion z^m + c
4 uniform int u_order;
```

7.4. Asignación de colores

Con la función $P_{c,m}$ ya programada, recordamos los algoritmos que utilizamos en *Mathematica* para graficar conjuntos de Julia y Mandelbrot, reprogramándolos ahora en GLSL para poder visualizar dichos conjuntos. Estos consistían en iterar la función $P_{c,m}$, cada uno a su manera, fijando un número máximo de iteraciones M tras las cuales se decidía qué puntos son prisioneros o de escape y almacenando en una variable cuántas iteraciones se han necesitado

para tomar esta decisión. En función del valor de dicha variable asignamos un color. Por tanto, necesitamos una forma de, a partir de este valor, asignar un color.

Lo primero de todo, este número máximo de iteraciones M es clave en esta asignación de colores. Queremos además que sea parametrizable, por lo que declaramos en el shader una variable `uniform` cuyo valor le pasaremos al shader con JavaScript.

```
1 // Numero de iteraciones maximo para decidir que elementos son
2 // prisioneros o escapan
3 uniform int u_maxIterations;
```

Seguidamente diseñaremos una paleta de colores y mediante interpolación lineal calcularemos qué color asignar a cada píxel. Podemos tomar tantos colores como queramos y elegir qué colores. Tras varias pruebas hemos decidido asignar el color negro (`rgb(0, 0, 0)=#000000`) para los puntos prisioneros (los que no divergen tras las M iteraciones) y para los que escapan utilizar un gradiente entre los colores de la imagen 7.2.



Imagen 7.2: Paleta de colores elegida para la visualización de fractales 2D

La paleta de colores 7.2 nos proporciona el gradiente de la imagen 7.3, que próximamente podremos ver en nuestros fractales, de manera que los puntos que diverjan antes se acercarán más al azul y los que tarden más se colorearán de un color más parecido al verde.

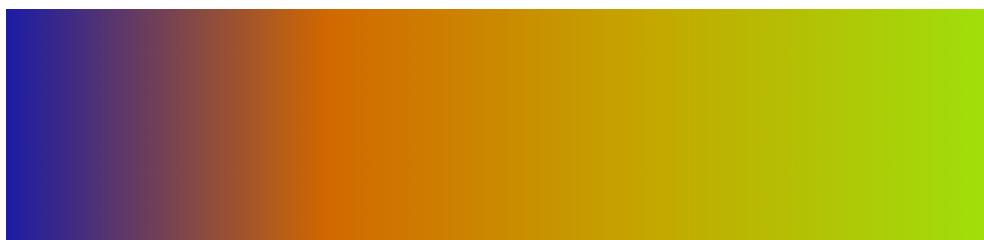


Imagen 7.3: Gradiente generado por la paleta de colores 7.2

Con esta decisión tomada, presentamos el código utilizado

```
1 // Paleta de colores, a partir de un numero 0<=t<=1 y 4
2 // colores c1, c2, c3 y c4 devuelve el color
3 // correspondiente en el gradiente creado por los cuatro
4 // colores.
5 vec3 palette(float t, vec3 c1, vec3 c2, vec3 c3, vec3 c4) {
6     float x = 1.0 / 3.0;
```

```

7   if (t < x) return mix(c1, c2, t/x);
8   else if (t < 2.0 * x) return mix(c2, c3, (t - x)/x);
9   else if (t < 3.0 * x) return mix(c3, c4, (t - 2.0*x)/x);
10  return c4;
11 }
12
13 // Asignacion de colores, a partir de una variable que define
14 // si el punto es prisionero o de escape y el numero de
15 // iteraciones antes de escapar se calcula y devuelve un color
16 // que proximamente se asignara finalmente a gl_FragColor
17 vec4 computePixelColor(bool escaped, int iterations) {
18     return escaped ? vec4(palette(
19         float(iterations)/ float(u_maxIterations),
20         vec3(0.109,0.109,0.647), // #1C1CA5
21         vec3(0.823, 0.411, 0.0), // #D26900
22         vec3(0.769, 0.659, 0.0), // #C4A800
23         vec3(0.627,0.878,0.043) // #AOE00B
24     ),
25     1.0) : vec4(vec3(0.0,0.0,0.0), 1.0);
26 }
```

Como se puede observar, es una simple interpolación lineal entre los 4 colores que se acaban de presentar, que en GLSL deben codificarse como tripletas RGB donde cada componente está normalizada entre 0 y 1.

7.5. Visualizando conjuntos de Julia

Tenemos entonces todos los ingredientes para programar la visualización, tan solo falta decidir si el punto que representa el píxel es prisionero o escapa mediante la iteración de $P_{c,m}$. Tras obtener las coordenadas de mundo que se le asignan al píxel mediante el método descrito en la sección 7.2, debemos iterar la función $P_{c,m}$. Atendiendo al teorema 3.2.1, en caso de que el módulo supere el valor 2 se decide que el punto es de escape. Si esto nunca ocurre y se alcanza el número máximo de iteraciones sin que el módulo sea mayor que 2, entonces se decide que el punto es prisionero (recordemos el pseudocódigo del algoritmo 1). Atendiendo a esta decisión y al número de iteraciones dadas se asigna el color correspondiente como comentamos en la sección 7.4. El código GLSL sería el siguiente:

```

1 // A partir del valor de la semilla z0, la constante c
2 // y el exponente m iteramos la funcion P_{c,m}.
3 void iterateJulia(vec2 z0, vec2 c, int m,
4     out bool escaped, out int iterations) {
5     vec2 z = z0;
6     escaped = false;
7     for(int i = 0; i < 10000; i++) {
8         if(i == u_maxIterations) break;
9         iterations = i;
10        z = P(z, c, m);
11        if (length(z) > 2.0){
12            escaped = true;
```

```

13     break;
14 }
15 }
16 }
17
18
19 // A partir del valor de la constante c y el exponente m
20 // se calcula el punto asociado al pixel, se itera P_{c,m}
21 // y se devuelve el color en función de dicha iteración.
22 vec4 Julia(vec2 c, int m) {
23     vec2 z0 = get_world_coordinates();
24     bool escaped;
25     int iterations;
26     iterateJulia(z0, c, m, escaped, iterations);
27     return computePixelColor(escaped, iterations);
28 }
```

Nótese el uso de los parámetros `out`, que son variables las cuales mantienen el valor que se les haya asignado en la función después del retorno. En este caso, se itera $P_{c,m}$ y se asigna `true` a la variable `escaped` si y solo si la sucesión de iteradas diverge. En caso de que la sucesión diverja, la variable `iterations` almacenaría el número de iteraciones calculadas hasta que el módulo de la iterada supere a 2.

Por último, desde la función `main`, llamamos a esta función recién presentada, le asignamos el color que devuelve a `gl_FragColor` y obtenemos los resultados que se presentan en las imágenes 7.4.

```
1 | gl_FragColor = Julia(u_juliaSetConstant, u_order);
```

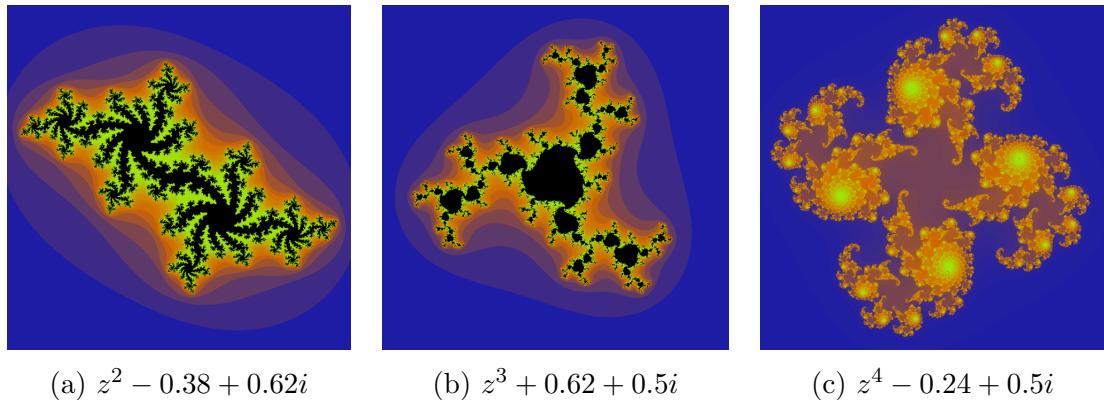


Imagen 7.4: Generación de algunos conjuntos de Julia con WebGL

7.6. Visualizando conjuntos de Mandelbrot

Ya hemos visto la metodología a utilizar si queremos visualizar conjuntos de Julia y, como es de esperar, la metodología para graficar conjuntos de Mandelbrot \mathcal{M}_m no es muy distinta. Recordamos que, tal y como afirmamos en las secciones 3.4 y 3.6.1 el conjunto de Mandelbrot \mathcal{M}_m se componía de aquellos elementos $c \in \mathbb{C}$ tales que el conjunto de Julia \mathcal{J}_c es conexo, o equivalentemente, de aquellos cuya sucesión $\{P_{c,m}^n(0)\}$ no diverge.

La idea consiste por tanto en obtener las coordenadas de mundo del píxel, iterar la función $P_{c,m}$ tomando $z_0 = 0$ como semilla y observar qué sucede, recordemos el algoritmo 2. Por la proposición 3.4.1 afirmamos que en el momento que una iterada supere en módulo a 2, entonces la sucesión de iteradas es divergente. Por tanto, almacenamos en una variable el número de iteradas que se han calculado antes de que el módulo de la sucesión supere a 2 en caso de que lo haga. Si esto no ocurre, se etiqueta al punto como prisionero y por tanto el punto pertenece a \mathcal{M}_m . De nuevo, la forma de asignar un color al píxel depende de si la sucesión diverge o no y del número de iteraciones hasta decidir que diverge en dicho caso. El código GLSL por tanto es el siguiente:

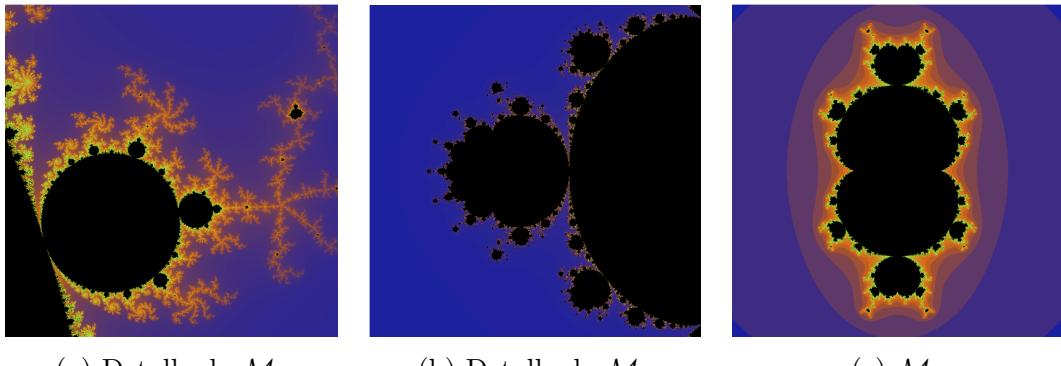
```

1 // A partir de la constante c, el exponente m y
2 // tomando como semilla z=0 se itera P_{c,m}
3 void iterateMandelbrot(vec2 c, int m,
4     out bool escaped, out int iterations) {
5
6     vec2 z = vec2(0.0);
7     escaped = false;
8     for (int i = 0; i < 10000; i++) {
9         if (i == u_maxIterations) break;
10        iterations = i;
11        z = P(z, c, m);
12        if (length(z) > 2.0) {
13            escaped = true;
14            break;
15        }
16    }
17 }
18
19 // A partir del exponente m se calcula el punto asociado al
20 // pixel, se itera y se devuelve el color asociado.
21 vec4 Mandelbrot(int m) {
22     vec2 c = get_world_coordinates();
23     bool escaped;
24     int iterations;
25     iterateMandelbrot(c, m, escaped, iterations);
26     return computePixelColor(escaped, iterations);
27 }
```

Vemos la analogía entre las funciones `iterateJulia` y `Julia` con las funciones `iterateMandelbrot` y `Mandelbrot`, estando las primeras presentadas en la sección 7.5. La diferencia fundamental se encuentra en que mientras en los conjuntos de Julia fijamos un $c \in \mathbb{C}$ y usamos el complejo asociado al píxel como semilla en el caso del conjunto de Mandelbrot usamos siempre $z_0 = 0$ como semilla y tomamos como c el complejo que representa el píxel.

Por tanto, ya solo falta llamar a esta función desde `main`, asignarle el valor que devuelve a `gl_FragColor` y podremos ver el conjunto de Mandelbrot en detalle. Presentamos algunas imágenes de detalles del mismo, a la vez que animamos a revisitar las imágenes 3.2, las cuales también se han obtenido mediante WebGL.

```
1 | gl_FragColor = Mandelbrot(u_order);
```



(a) Detalle de \mathcal{M}_2

(b) Detalle de \mathcal{M}_8

(c) \mathcal{M}_3

Imagen 7.5: Representación de algunos conjuntos de Mandelbrot con WebGL

7.7. Alternando conjuntos de Julia y Mandelbrot

Finalmente, queremos dotar a nuestra web de la posibilidad de alternar qué fractal visualizar (Julia o Mandelbrot) y poder ajustar los parámetros a nuestro gusto. Para ello introducimos en el fragment Shader una variable `uniform` entera que, en caso de valer 0 se visualizaría el conjunto de Mandelbrot y en caso de valer 1 se visualizaría el conjunto de Julia.

```

1 // Graficar conjunto de Julia o de Mandelbrot
2 uniform int u_fractal;
3
4 // ...
5
6 void main() {
7     vec4 color;
8     if(u_fractal == 0){
9         color = Mandelbrot(u_order);
10    }
11    else{
12        color = Julia(u_juliaSetConstant, u_order);
13    }
14    gl_FragColor = color;
15 }
```

7.8. Supersampling Antialiasing

Para terminar este capítulo, introduciremos una técnica que aportará realismo y calidad a nuestras imágenes. El aliasing es un efecto que ocurre en muchas aplicaciones gráficas debido a la discretización que se realiza de una imagen real en un subconjunto finito de píxeles. Por ejemplo, a no ser que una línea recta sea totalmente vertical u horizontal, se presentará con sobresaltos impropios de una línea recta (véase imagen 8.22). Por su parte, el *antialiasing* es un conjunto de técnicas que nos permite evitar este efecto mediante el suavizado de bordes. Por ejemplo, fíjese en la imagen 7.6, de $\mathcal{J}_{-0.53+0.53i}$. En ella se pueden observar irregularidades, como por ejemplo pequeños puntos negros o píxeles consecutivos de colores muy distintos.

Una forma simple de corregir estos efectos se consigue teniendo en cuenta que se está identificando cada píxel con un único punto del plano, cuando realmente en cada píxel hay

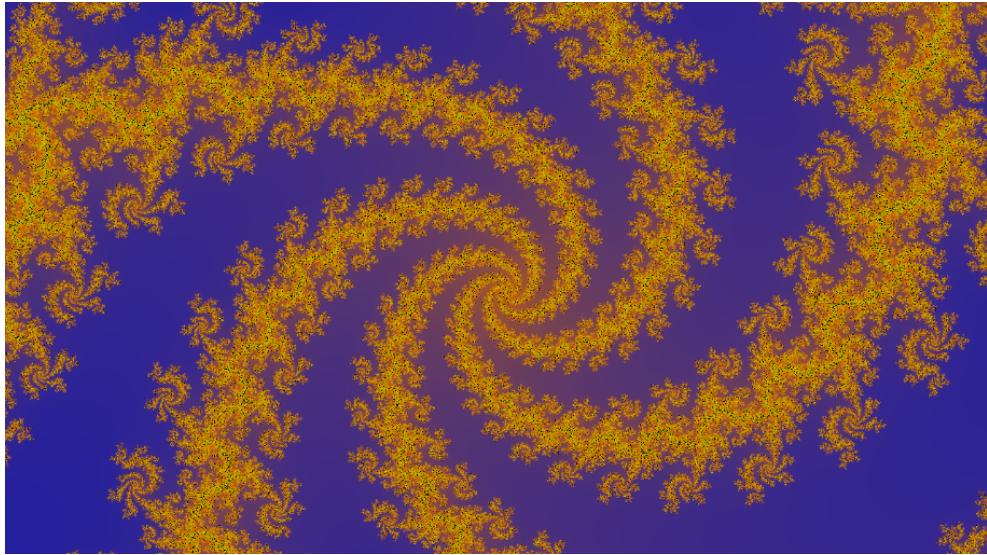


Imagen 7.6: Detalle de $\mathcal{J}_{-0.53+0.53i}$ antes de aplicar antialiasing

infinitos puntos. Podría incluso ocurrir que el conjunto de Julia atravesara el píxel de forma que la mayor parte de sus puntos sean de escape pero el que evalúa el algoritmo es prisionero, coloreando el píxel de color negro cuando realmente no todos los puntos que lo componen son prisioneros. Una solución sencilla a este impedimento es aplicar la técnica popularmente conocida como *Supersampling Antialiasing* (SSAA). El SSAA consiste en calcular más de un punto por píxel, de forma que se cubra una región más representativa del mismo y se promedie el color que devuelve el algoritmo en cada uno de los puntos.

Fijémonos en la figura 7.7 (a), en la cual representamos una pantalla de 5×3 píxeles y cada punto señalado sería un punto del plano complejo que iteraríamos y calcularíamos el color asociado según las técnicas implementadas hasta el momento. Para calcular más de un punto, fijamos un número natural $n \in \mathbb{N}$ y programaremos la forma de calcular n^2 puntos en cada píxel, tal y como vemos en la imagen 7.7 (b) con $n = 2$.

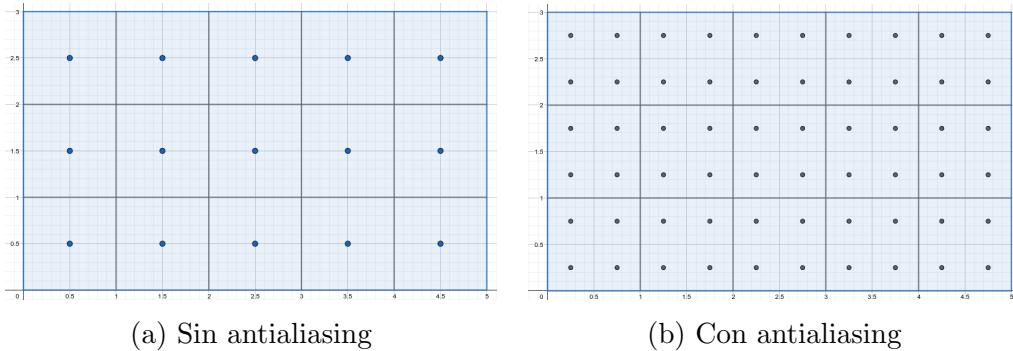


Imagen 7.7: Representación de los rayos que se lanzan a una pantalla

Por tanto, ahora, en lugar de calcular las “coordenadas de mundo de cada píxel”, a las cuales nos referímos como las coordenadas de mundo del punto situado en el centro del píxel, ahora debemos calcular, por cada píxel, n^2 coordenadas.

Recordamos, del código de la función `get_world_coordinates` presentado en la sección 7.2, que utilizamos las variables `u, v` como las coordenadas de dispositivo normalizadas en $[0, 1]$,

es decir:

```

1 // ...
2 vec2 uv = (gl_FragCoord.xy + vec2(0.5)) /
3         vec2(float(nx), float(ny));
4 float u = uv.x;
5 float v = uv.y;

```

Si ahora en lugar de utilizar un punto por píxel situado en centro del mismo deseamos tener n^2 puntos uniformemente distribuidos por la superficie del píxel, tenemos que dividir su ancho en n intervalos, análogamente el alto. Por tanto, dividimos cada píxel en una cuadrícula $n \times n$ donde cada fragmento mide $h_w := \frac{1}{n_x \cdot n}$ en anchura y de $h_h := \frac{1}{n_y \cdot n}$ en altura. Tras esto, deberíamos sumar $\frac{1}{2}h_w$ en ancho y $\frac{1}{2}h_h$ en alto, para así tener dicha cuadrícula centrada en el píxel, véase la imagen 7.8.

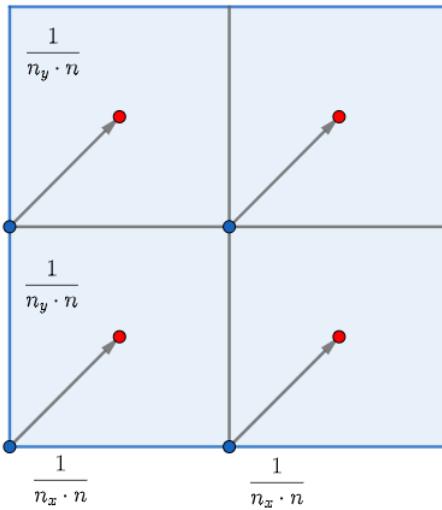


Imagen 7.8: Construcción de los puntos por píxel en SSAA

Seguidamente, calculamos los n^2 colores, uno por cada punto que utilicemos. Por cada punto, iteramos la función $P_{c,m}$ igual que en las secciones anteriores y que en los algoritmos 1 y 2 y calculamos el color que le corresponde, de forma que sumamos los n^2 colores que se calculan. Finalmente, dividimos esta suma entre n^2 , obteniendo así el color promedio del píxel, siendo este el valor que se asignaría finalmente a `gl_FragColor`.

La forma de hacer este cálculo es iterando un índice $i = 0, \dots, n^2 - 1$. Para a partir del índice i obtener la coordenada correspondiente podemos dividir el conjunto $\sigma_{n^2} = \{0, 1, \dots, n^2 - 1\}$ en n conjuntos de n elementos cada uno, de forma que

$$\sigma_{n^2} = \bigcup_{k=0}^{n-1} \{nk, nk+1, \dots, nk+n-1\}.$$

Cada subconjunto de σ_{n^2} representa una fila de puntos del píxel. Por tanto, aplicando la división entera, i/n será la fila e $i \% n$ la columna. Por lo que habría que sumar $(i/n)h_w + \frac{1}{2}h_w$ a la coordenada anchura inicial e $(i \% n)h_h + \frac{1}{2}h_h$ a la altura para obtener la coordenada del punto i -ésimo.

Para ello, modificamos la función `get_world_coordinates` para que acepte el índice i (`int i`) y el valor n (`int nSamples`) como parámetros y calcule las coordenadas de mundo

del punto i -ésimo del píxel.

```

1 vec2 get_world_coordinates(int i, int nSamples) {
2
3     int nx = 1280, ny = 720; // Canvas size (pixels)
4     float r = 16.0/9.0;      // Ratio width/height = 16/9
5     float h = 4.0, w = r * h; // Image size (units)
6
7     // Incrementos
8     float hw = 1.0 / (float(nx * nSamples)),
9          hh = 1.0 / (float(ny * nSamples));
10
11    int x = i/nSamples;
12    int y = i - nSamples*x;
13
14    // Coordenadas de dispositivo normalizadas en [0,1]x[0,1]
15    vec2 uv = (gl_FragCoord.xy) /
16        vec2(float(nx), float(ny));
17    float u = uv.x + float(x) * hw + 0.5 * hw,
18        v = uv.y + float(y) * hh + 0.5 * hh;
19
20    return u_zoomCenter + vec2(u*w - w/2.0, v*h - h/2.0) * u_zoomSize;
21 }
```

Finalmente, en las funciones `Julia` y `Mandelbrot`, en lugar de hacer una única llamada a la función `get_world_coordinates()`, hacemos n^2 llamadas a `get_world_coordinates(i, nSamples)`, fijando previamente el valor de n (`nSamples`). Sumamos los n^2 colores que se calculan en cada muestra y finalmente se devuelve el promedio. Mostramos el código de la función `Julia`, ya que el de `Mandelbrot` es totalmente análogo.

```

1 vec4 Julia(vec2 c, int n) {
2     vec4 sum_colors = vec4(0.0, 0.0, 0.0, 1.0);
3     int nSamples = 3; // Se calcularian 3^2 = 9 colores
4     for(int i = 0; i < 1000; i++) {
5         if(i == nSamples*nSamples) break;
6         vec2 z0 = get_world_coordinates(i, nSamples);
7         bool escaped;
8         int iterations;
9         iterateJulia(z0, c, n, escaped, iterations);
10        sum_colors += computePixelColor(escaped, iterations);
11    }
12    return sum_colors/float(nSamples*nSamples);
13 }
```

El resultado tras aplicar esta modificación en $\mathcal{J}_{-0.53+0.53i}$, que es el conjunto que aparece en la imagen 7.6, es la imagen 7.9. Fijémonos como la sensación de irregularidad desaparece, dando apariencia más suavizada, mejorando notablemente la calidad de la imagen.

El principal y mayor inconveniente que presenta esta técnica es que, como se puede imaginar, es n^2 veces más costosa que utilizar un único punto por píxel, por lo que se ralentiza mucho la ejecución. Sin embargo, si se desea una imagen concreta con parámetros muy claros y no tanto la interacción es una buena herramienta, pues proporciona un nivel de detalle mucho mayor.

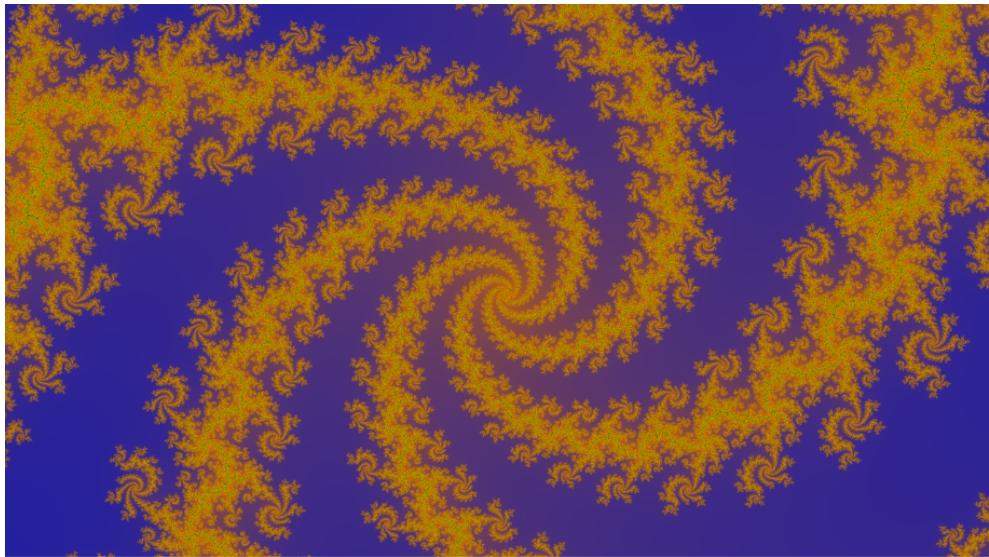


Imagen 7.9: Detalle de $J_{-0.53+0.53i}$ tras aplicar SSAA

Precisamente por este costo, se incluye en el software la posibilidad de decidir si se desea o no aplicar SSAA, y en caso afirmativo, cuántos rayos por píxel se desean trazar. Esto se consigue mediante dos variables `uniform`. Una de ellas es booleana, su nombre es `u_antialiasing`, que en caso de valer `true` se aplicaría antialiasing y en caso contrario se utilizaría un único punto por píxel. Para ello utilizamos la otra variable `uniform`, que es entera y su nombre es `u_nSamples`, que es la análoga a la `n` que hemos utilizado en las explicaciones. En caso de que `u_antialiasing` tenga el valor `false`, `u_nSamples` tendrá el valor 1, en caso contrario tendrá el valor que decida el usuario.

```
1 uniform bool u_antialiasing;
2 uniform int u_nSamples;
3 // ...
4 vec4 Julia(vec2 c, int n) {
5     vec4 sum_colors = vec4(0.0, 0.0, 0.0, 1.0);
6     // int nSamples = 3;
7     int nSamples = u_antialiasing ? u_nSamples : 1;
8     // ...
9 }
```


CAPÍTULO 8

INTRODUCCIÓN AL RAY-TRACING

En los anteriores capítulos hemos visto cómo asignar colores a los píxeles utilizando el fragment shader y cómo dibujar hermosos fractales en la superficie que ocupan dos triángulos. Sin embargo, no hemos podido movernos de las dos dimensiones. Esto se debe a la dificultad que de por sí supone que en 2D un píxel puede representar un único punto del plano mientras que en 3D un píxel representa infinitos puntos del espacio. Nuestro objetivo ahora es poder visualizar escenas simples en 3D para posteriormente generar fractales tridimensionales. El código del fragment shader que se emplea en este capítulo se puede encontrar en el fichero `fragment-shader-ray-tracer.js`, que se puede encontrar en <https://github.com/JAntonioVR/Geometria-Fractal/blob/main/static/glsl/fragment-shader-ray-tracer.js>.

8.1. Definición de Ray-Tracing

En el mundo de la informática gráfica existen dos formas fundamentales de proceder a la representación de escenas. Supongamos que tenemos una escena 3D con varios objetos, como pueden ser por ejemplo un cubo y un par de esferas y queremos visualizar la misma en un canvas 2D.

- Una de las formas de representarla es la **rasterización**, ya introducida al inicio del capítulo 5. Esta técnica se basa en modelos de fronteras de objetos 3D. Esto es, se representan objetos constituidos a partir de conjuntos de caras planas poligonales (típicamente triángulos), y se denominan primitivas a dichas caras. Pensemos por ejemplo en un cono como una pirámide cuya base es un polígono regular de muchos lados. El método consiste en identificar para cada primitiva que compone la escena qué píxeles ocupa y asignar color a cada píxel, aplicando posibles modelos de iluminación y texturas.
- Otra forma es, para cada pixel, calcular qué primitivas de la escena se proyectan sobre dicho píxel y colorearlo dependiendo de la primitiva más cercana al observador. Esta es la base del **Ray-Tracing**. En este caso los objetos son representados mediante un algoritmo de cálculo de intersección entre una semirrecta (rayo) y la superficie del objeto en cuestión. Esto engloba los modelos de fronteras utilizados en rasterización, pero además

abre posibilidades como objetos con extensión infinita o los propios fractales, que son nuestro principal objetivo.

Más en profundidad, la idea es situar al observador en una cierta posición de la escena y colocar frente a él un plano, que llamaremos *plano de proyección*, el cual estará dividido en tantos píxeles como tenga el canvas, de tal manera que se ‘trazan rayos’ que salen desde la posición del observador (en adelante también llamada ‘punto de vista’ o ‘foco de la proyección’) en dirección a cada píxel, por lo que se lanzan tantos rayos como píxeles haya. El rayo atraviesa el plano y avanza por la escena hasta que encuentra la intersección con un objeto. En este caso, se calcula en qué punto se ha alcanzado la intersección y se evalúa un modelo de iluminación o se asigna un color dependiendo de las características de la escena. También es posible que el rayo no alcance ningún objeto, en cuyo caso habría que saber qué hacer con los píxeles cuyos rayos se pierden en el infinito. En la imagen 8.1 se puede observar un esquema aclarativo del funcionamiento de este método.

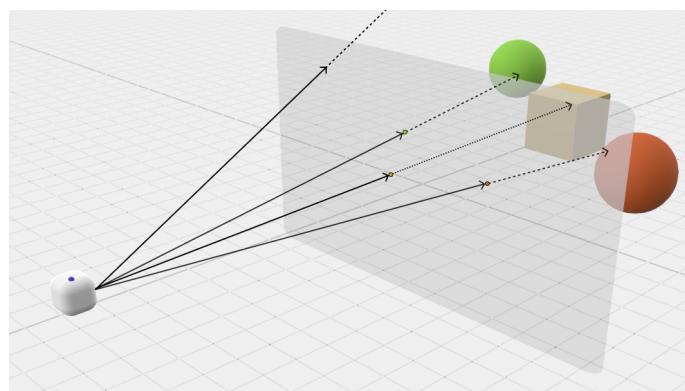


Imagen 8.1: Esquema básico del funcionamiento del Ray-Tracing

La principal ventaja que tiene el Ray-Tracing (en adelante RT) sobre la rasterización es que este trazado de rayos nos permite facilitar el cálculo de los reflejos y sombras creados por las iluminaciones del entorno, consiguiendo así efectos mucho más realistas en las escenas. Además, las figuras fractales como objetos matemáticos ideales carecen de una superficie o frontera que sea una variedad bidimensional o superficie regular continua, ni siquiera continua a trozos, por lo que no es posible representarlos idealmente utilizando modelos de fronteras como los que utiliza la rasterización.

No obstante, al igual que en la mayoría de problemas de computación gráfica, los fractales 3D se pueden convertir a modelos de fronteras asumiendo un cierto error de discretización, el cual es totalmente inevitable. Por tanto, partiendo de este hecho, es posible visualizar esos modelos con rasterización o con ray-tracing. Sin embargo, esa conversión es lenta y produce modelos de gran tamaño en memoria.

Por tanto, es mucho más sencillo usar RT, el cual, si bien también es algo aproximado, es muchísimo más eficiente en memoria, ya que no requiere de ninguna estructura de datos, sino simplemente de un algoritmo de intersección aproximado entre una semirrecta y el fractal. Además no emplea tiempo en la conversión a modelo de fronteras.

Otra gran ventaja del ray-tracing sobre la rasterización es la dificultad de elegir una conversión a modelos de fronteras con una resolución adecuada. Si la discretización es demasiado

fina el tiempo de computación crece mientras que se consigue una mayor resolución, y si no es lo suficientemente fina se pierde mucha resolución.

Sin embargo, este algoritmo tiene una principal desventaja, y es que es un proceso muy costoso, y más aún cuanto más detalle queramos conseguir. Es por esto que ha sido muy difícil dar el salto al ray tracing en tiempo real. Por ejemplo, en el mundo de los videojuegos NVIDIA, que es una famosa empresa desarrolladora de GPUs, comenzó a desarrollar algoritmos para poder utilizar RT en sus GPUs hace varios años, pero hasta finales de 2018 no salieron al mercado las primeras gráficas con estas características. Por su parte, los juegos también deben soportar el algoritmo, por lo que aún son una minoría de videojuegos los que en la actualidad soportan Ray Tracing. En un futuro no muy lejano es posible que haya una tendencia a un desarrollo masivo de juegos que utilicen RT, pero de momento solo tenemos algunos ejemplos como Battlefield V, Minecraft, Metro Exodus, Watch Dogs y Call of Duty: Modern Warfare (2019).

En resumen, y por todas las ventajas y a pesar de las desventajas que hemos descrito de RT sobre la rasterización y los modelos de fronteras en nuestro problema concreto, tomamos la decisión de programar un *ray-tracer* inicialmente básico con el objetivo futuro de generar fractales 3D. La idea por tanto es lanzar rayos a cada píxel y en caso de encontrar intersección con el objeto evaluar un modelo de iluminación para asignar un color concreto.

8.2. Elementos y estructura de un Ray-Tracer

Hasta ahora hemos visto y explicado en qué consiste el algoritmo RT desde un punto de vista básico. Veamos ahora qué componentes tiene un programa que emplee ray-tracing y conceptos generales, pero más concretos, acerca del cálculo de intersecciones entre semirrectas y objetos 3D. Tres de los componentes más esenciales en un *ray-tracer* son:

- **Generación de rayos primarios:** Por cada píxel se crea un rayo (llamado *rayo primario* o *rayo de cámara*).
- **Cálculo de intersecciones:** Dado un punto $o \in \mathbb{R}^3$ y un vector $\vec{v} \in \mathbb{R}^3$, se trata de encontrar, si existe, el mínimo valor positivo de t tal que el punto $p(t) = o + \vec{v}t$ se encuentra en la superficie o frontera de algún objeto de la escena.
- **Modelo de Iluminación Local:** Abreviado como MIL, calcula el color que se asigna a cada píxel cuyo rayo asociado ha intersecado un objeto en función del material asignado, las fuentes de luz y el punto de intersección rayo-objeto. RT además en este caso nos proporciona una poderosa ventaja, pues trazando un rayo desde cualquier punto de la escena en dirección a una fuente de luz podemos conocer si dicho punto está o no en la sombra arrojada de algún objeto sin más que evaluar si el rayo encuentra alguna intersección en su camino hacia la fuente de luz.

Con estos tres elementos, una primera descripción en pseudocódigo del algoritmo fundamental de Ray-Tracing sería el del algoritmo 3.

Sobre el cálculo de intersecciones, en caso de que se utilicen modelos de fronteras es necesario a su vez utilizar indexación espacial. Esto es, dividir jerárquicamente el espacio en zonas de forma que sea más sencillo localizar un objeto. Esto se debe a que, en caso de encontrarse una

Algoritmo 3 Ray Tracing

$o \leftarrow$ posición del observador en WC.
for cada píxel (i, j) del canvas **do**
 $q \leftarrow$ punto central del píxel (i, j) en WC
 $\vec{v} \leftarrow$ vector desde o hasta q
 $O \leftarrow$ primer objeto visible desde o en la dirección \vec{v}
 if no existe objeto visible **then**
 $rad \leftarrow$ color de fondo para \vec{v}
 else
 $p \leftarrow$ punto de intersección en O
 $rad \leftarrow$ evaluarMIL(O, p).
 end if
 Asignar el color rad al píxel (i, j)
end for

intersección con un objeto, para saber de cual se trata podemos hacer una búsqueda secuencial, pero sería muy ineficiente en escenas muy complejas. Por ello, se divide en regiones el espacio y se busca en orden de complejidad logarítmica.

Por su parte, para intersecciones de rayos con objetos no definidos mediante modelos de fronteras debemos calcular el primer cero de determinado campo escalar a lo largo del rayo. Para ello hay dos posibilidades:

- **Usando expresiones analíticas.** Por ejemplo, una esfera de centro $c \in \mathbb{R}^3$ y radio $r > 0$ encuentra una intersección con el rayo $o + \vec{v}t$ en el punto que se verifica la ecuación

$$\|o + \vec{v}t - c\| - r = 0$$

Este tema se aborda con detalle en la sección 8.5.1.

- **Con un método iterativo.** En ocasiones la resolución de estas ecuaciones es demasiado compleja, por lo que se abordan soluciones iterativas, como pueden ser técnicas estándar de resolución de ecuaciones como el método de Newton. Dentro de esta posibilidad, podemos también contar con una función que estima la distancia a la superficie del objeto y aplicar el algoritmo conocido como ‘ray marching’ o ‘sphere tracing’, el cual explicaremos detenidamente en la sección 9.1, pero adelantamos que fue introducido por Hart en [15].

A partir de ahora, veremos como implementar un *ray-tracer* básico utilizando WebGL. Utilizaremos objetos sencillos como planos y esferas, cuyas intersecciones con rayos se pueden calcular fácilmente de forma analítica sin recurrir a métodos iterativos ni a modelos de fronteras.

En nuestro caso, y de manera similar a la efectuada con los fractales 2D, el vertex shader utilizado es trivial, de hecho es exactamente el mismo que se puede encontrar al inicio del capítulo 7. Por tanto el grueso de la programación se hará de nuevo en el fragment shader, que es el que se ejecuta una vez por píxel. Por su parte, usaremos JavaScript para pasarle variables uniform al shader y para añadir interactividad a la escena. Usaremos de nuevo las clases **Shader** y **Buffer**, pero por las diferencias que existen entre los parámetros de una escena en 2D y una escena en 3D se ha implementado una nueva clase **Scene3D**, que realmente hace lo mismo que **Scene2D** pero

utilizando parámetros que requiere una escena 3D. También para gestionar la interacción usuario-escena se ha utilizado un nuevo fichero de JavaScript: `fractals-3D.js`, llamado así porque es el mismo fichero que en el futuro utilizaremos para interactuar con fractales. Recomendamos revisitar la sección 7.1 para recordar el papel que realiza la escena y el fichero que gestiona la interacción y los eventos. Puede consultar la documentación del código de JavaScript en el apéndice A.

Las siguientes secciones irán dedicadas a explicar la programación del fragment shader, ya que es el elemento que mayor complejidad y lógica contiene.

8.3. Creación de rayos primarios

Recordamos que cuando se trataba de fractales 2D utilizamos una transformación lineal para identificar cada píxel con un punto del plano complejo (sección 7.2). En este caso debemos identificar cada píxel con un punto no del plano complejo, sino del *plano de proyección* (P), que es el plano que colocamos frente al espectador dividido en tantos píxeles como tenga el canvas, de forma que se trazan rayos desde el espectador y hacia dichos píxeles. Supongamos a partir de ahora que queremos representar la escena en un canvas de $n_x \times n_y$ píxeles.

Para cada píxel debemos calcular su posición central en coordenadas de mundo a partir de las coordenadas de dispositivo, que son dos valores enteros (x', y') , aunque para identificar cada píxel con su centro tomaremos como coordenadas de dispositivo $(x, y) = (x', y') + (\frac{1}{2}, \frac{1}{2})$. Este proceso es justamente el contrario al que se utiliza en rasterización, donde se necesita convertir las coordenadas de mundo de las primitivas en coordenadas de dispositivo. Para ello, se utiliza una matriz M que es la composición de la matriz de vista 3D, la cual transforma coordenadas de mundo (WC) a coordenadas de cámara junto con la matriz de proyección, que calcula las coordenadas de dispositivo (DC) a partir de las coordenadas de cámara (véase la analogía con los párrafos que describen la transformación 7.4). Por tanto, en cierto sentido lo que buscamos es buscar una transformación lineal asociada a la inversa de la matriz M .

En nuestro caso concreto, y al igual que en el caso 2-dimensional, utilizaremos un canvas de 1280×720 píxeles, que guarda una proporción de 16 : 9.

```
1 | <canvas id="glCanvas" width="1280" height="720"></canvas>
```

Inicialmente, supongamos que el observador se sitúa en el punto $(0, 0, 0)$ y que el plano de proyección se sitúa a distancia 1 en el eje negativo Z . Esta distancia del observador al plano de proyección es la conocida como la *distancia focal*. Se denomina *lookfrom* u origen y denotamos como o_c al punto desde el que se observa, es decir, en el que se sitúa el observador; y *lookat* o punto de atención a_t al punto hacia el que mira, que en este caso sería el $(0, 0, -1)$. En ocasiones nos referiremos al punto o_c como posición del observador o punto de vista.

Como convención asumimos que a la derecha se sitúa el eje positivo X , hacia arriba el eje positivo Y y el ‘interior de la pantalla’ es el eje negativo Z . Asumimos ahora también que el plano de proyección tiene 2 unidades de alto, lo cual implica que si el ratio ancho/alto es 16/9 entonces el ancho es de 3.55 unidades, aunque ese es un valor que calcularemos y almacenaremos en una variable y no importará realmente cual sea.

Un rayo es en realidad simplemente una semirrecta de \mathbb{R}^3 , las cuales están únicamente determinadas por un punto p del espacio afín \mathbb{R}^3 al cual llamaremos *origen* y por un vector \vec{v}

del espacio vectorial \mathbb{R}^3 que denominamos *dirección*. De esta forma, un rayo puede ser expresado como la imagen de la función

$$R(t) = p + t \cdot \vec{v} \quad \forall t \in \mathbb{R}_0^+,$$

de forma que para cualquier punto p_0 del rayo R existe un único $t_0 \in \mathbb{R}_0^+$ tal que $R(t_0) = p_0$.

A nivel de código GLSL, la manera de representar un rayo será utilizando una estructura (**struct**) que denominaremos **Ray**. Las estructuras en GLSL son muy parecidas en sintaxis y también en uso a las del lenguaje C.

```
1 struct Ray {
2     vec3 orig;           // Origen del rayo
3     vec3 dir;            // Dirección del rayo
4 };
```

Y si dado un rayo R queremos calcular qué punto de \mathbb{R}^3 le corresponde a cierto $t \in \mathbb{R}_0^+$, utilizamos la siguiente función.

```
1 vec3 ray_at(Ray R, float t){
2     return R.orig + t*R.dir;
3 }
```

Esta función nos será útil a la hora de calcular el punto exacto en el que se produce una intersección rayo-objeto cuando solo se conoce la distancia t a la que se produce el impacto.

Observación 8.3.1. Realmente se pueden utilizar valores de $t \in \mathbb{R}$ positivos o negativos, pero pensemos que los valores negativos corresponden a puntos del rayo situados detrás del observador, que no se pueden ver, por lo que es preferible restringirnos a valores no negativos.

Una vez tenemos determinada una forma de representar un rayo, es momento de crear uno cuyo origen sea la posición del observador y su dirección sea el vector que tiene como origen el observador y como destino el punto del plano de proyección que identificamos con el píxel. En la situación hipotética que hemos planteado antes en la cual $o_c = (0, 0, 0)$, llamamos d_f a la distancia focal, $P_h = 2$ a la altura del plano de proyección, P_w a la anchura del plano de proyección, $r = 16/9$ (*aspect ratio*) a la proporción $r = \frac{P_w}{P_h}$, de forma que se verifica $P_w = r \cdot P_h$. Con estas variables, llamemos LLC (*Lower Left Corner*) al punto situado en la esquina inferior izquierda del plano de proyección, entonces:

$$\text{LLC} = o_c - \frac{P_w}{2} \cdot (1, 0, 0) - \frac{P_h}{2} \cdot (0, 1, 0) - d_f \cdot (0, 0, 1) \quad (8.1)$$

Una vez conocemos las coordenadas de mundo de la esquina superior izquierda, la cual identificamos con el píxel inferior izquierdo del canvas, debemos recuperar la transformación 7.3, pero en este caso nos debemos llevar la región $[0, n_x] \times [0, n_y]$ a $[0, P_w] \times [0, P_h] \times \{-1\}$ (recordemos que estamos situando el plano de proyección en el plano $z = -1$). La transformación por tanto en este caso es

$$\begin{aligned} \phi : [0, n_x] \times [0, n_y] &\longrightarrow [0, P_w] \times [0, P_h] \times \{-1\} \\ (x, y) &\longmapsto \left(\frac{P_w \cdot x}{n_x}, \frac{P_h \cdot y}{n_y}, -1 \right) \end{aligned} \quad (8.2)$$

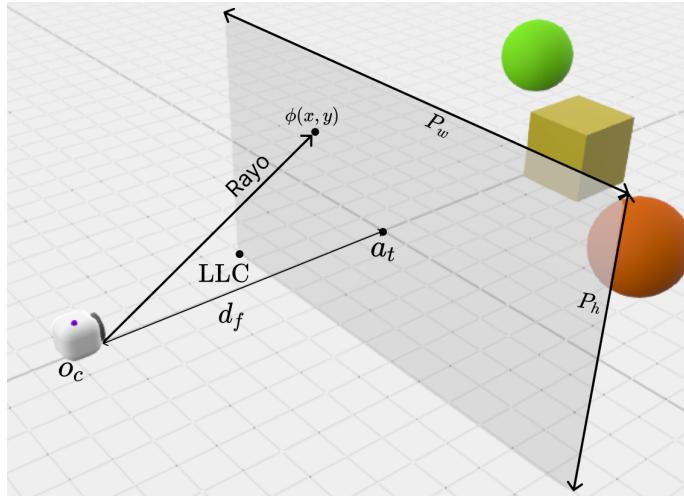


Imagen 8.2: Elementos que participan en RT

donde (x, y) son las coordenadas de dispositivo en términos de píxeles a las que puede acceder el fragment shader a través de `gl_FragCoord`, a las cuales recordamos que se les suele añadir $\frac{1}{2}$ en cada componente para así hacer la identificación con el punto que se encuentra en el centro del píxel, y no la esquina inferior izquierda. Así identificamos el alto y ancho del canvas con el alto y ancho del plano de proyección, de manera que a partir de estos valores y conociendo qué punto se sitúa en la esquina inferior izquierda podemos definitivamente calcular con qué punto del plano de proyección (P) identificamos el píxel:

$$\begin{aligned} \phi : [0, n_x] \times [0, n_y] &\longrightarrow P \\ (x, y) &\longmapsto LLC + \left(\frac{P_w}{n_x} x, \frac{P_h}{n_y} y, -1 \right) \end{aligned} \quad (8.3)$$

Y por tanto, la dirección del rayo sería

$$\vec{v} = \phi(x, y) - o_c$$

y el origen sería obviamente el punto o_c .

Como mencionamos anteriormente, la transformación ϕ es la que corresponde a la inversa de la matriz M de vista y proyección, pues transforma las coordenadas de dispositivo en coordenadas de mundo del plano de proyección. Suponiendo que identificamos las coordenadas de dispositivo (x, y) con el vector $(x, y, 0, 1)^t$ y un punto $p = (p_1, p_2, p_3) \in \mathbb{R}^3$ con el vector $(p_1, p_2, p_3, 1)^t$, la transformación (8.3) puede expresarse matricialmente como:

$$\phi(x, y) = \left(\begin{array}{ccc|c} \frac{P_w}{n_x} & 0 & 0 & 0 \\ 0 & \frac{P_h}{n_y} & 0 & 0 \\ 0 & 0 & 0 & -1 \\ \hline 0 & 0 & 0 & 1 \end{array} \right) \begin{pmatrix} x \\ y \\ 0 \\ 1 \end{pmatrix} \quad (8.4)$$

El código GLSL que hemos utilizado hasta este punto, en el que hemos calculado `LLC` e implementado la transformación (8.3), sería el siguiente:

```
1 | 
2 | Ray get_ray(vec3 lookfrom, vec3 lookat,
```

```

3     float viewport_width, float viewport_height,
4     float u, float v) {
5
6     Ray R;
7     R.orig = lookfrom;
8     float df = length(lookat - lookfrom); // Distancia focal
9     vec3 lower_left_corner = lookfrom
10        - viewport_width/float(2.0)*vec3(1.0, 0.0, 0.0)
11        - viewport_height/float(2.0)*vec3(0.0, 1.0, 0.0)
12        - df*vec3(0.0, 0.0, 1.0);
13     R.dir = lower_left_corner
14        + u*vec3(viewport_width, 0.0, 0.0)
15        + v*vec3(0.0, viewport_height, 0.0)
16        - lookfrom;
17     return R;
18 }
19
20 // ...
21
22 // Dimensiones del canvas
23 float aspect_ratio = 16.0/9.0;
24 float nx = 1280.0;
25 float ny = nx / aspect_ratio;
26
27 // Coordenadas de dispositivo normalizadas [0,1]
28 vec2 uv = (gl_FragCoord.xy + vec2(0.5)) / vec2(nx, ny);
29 float u = uv.x;
30 float v = uv.y;
31
32 // Dimensiones del plano de proyeccion
33 float viewport_height = float(2.0);
34 float viewport_width = viewport_height * aspect_ratio;
35
36 // lookfrom y lookat
37 vec3 lookfrom = vec3(0.0, 0.0, 0.0);
38 vec3 lookat = vec3(0.0, 0.0, -1.0);
39
40 // Ray
41 Ray R = get_ray(lookfrom, lookat,
42     viewport_width, viewport_height,
43     u, v);

```

Y de esta forma el fragment shader obtiene un rayo por cada píxel que tiene como origen la posición del observador y que interseca con el plano de proyección en el punto correspondiente al píxel.

8.4. El background

Una vez tenemos creado el rayo asignado a un píxel debemos asignar un color. La función que dado un rayo R devuelve el color del cual colorearemos el píxel la llamaremos `ray_color`. Esta función se verá sometida a muchos cambios, sobre todo en sus argumentos dependiendo de los elementos que compongan la escena, pero de momento asumimos una escena vacía. Que la escena esté vacía supone que ningún rayo interseca ninguna superficie, pero independientemente de ello hay que asignar un color al píxel. Esta asignación de color a un rayo que no interseca ninguna superficie determina el fondo (*background*) de la escena, y esta decisión que tomaremos ahora sobre cómo colorear el fondo nos acompañará durante el resto del desarrollo.

En concreto, hemos decidido simular algo parecido al cielo mediante un degradado vertical de un azul `rgb(127, 178, 255)` a blanco `rgb(255, 255, 255)`, definiendo el color a partir de la componente y del vector director normalizado. Mostramos el código correspondiente también para clarificar esta descripción.

```
1 | vec4 ray_color(Ray R) {  
2 |     // R no interseca ninguna superficie  
3 |     vec3 unit_direction = normalize(R.dir);  
4 |     float t = 0.5*(unit_direction.y + 1.0);  
5 |     return vec4((1.0-t)*vec3(1.0,1.0,1.0) + t*vec3(0.5,0.7,1.0),  
6 |                 1.0);  
 }
```

De esta forma, si el rayo apunta hacia arriba el color del píxel será más azul y si apunta hacia abajo más blanco. En la imagen 8.3 podemos ver el gradiente utilizado.

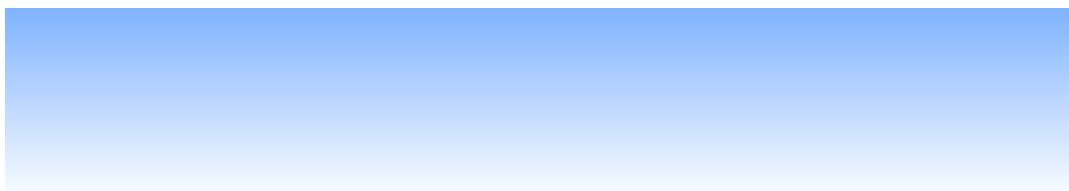


Imagen 8.3: Gradiente utilizado para el fondo de la escena

Y en la imagen 8.4 podemos ver el resultado de efectuar la llamada a `ray_color` una vez creado el rayo. Esta es la primera escena 3D generada utilizando RT en WebGL.

```
1 | gl_FragColor = ray_color(R);
```

Nótese que en la imagen no aparecen todos los colores del gradiente de la imagen 8.3, y esto se debe a que los rayos que se trazan en esta situación inicial no cubren todas las alturas posibles. Por ejemplo, no se traza un rayo totalmente vertical hacia arriba ni hacia abajo, solo se trazan las alturas necesarias para cubrir el plano de proyección. Esto nos da una manera de orientarnos en la altura una vez parametrizamos la posición y orientación de la cámara (sección 8.6), de tal manera que si vemos el fondo muy azul podemos asumir que se está mirando ‘al cielo’ y si el color es blanco se estará mirando ‘al vacío’.



Imagen 8.4: Primera escena vacía visualizada

8.5. Visualizando una escena sencilla

Hasta este momento hemos diseñado la estructura del ray tracer como un observador situado en la posición $(0, 0, 0)$ y proyectando una escena vacía en un plano de 2 unidades de alto y guardando un ratio de 16:9. Sin embargo, aún queda lo más importante, que es añadir cuerpos a la escena con los que los rayos puedan intersecciar. El objetivo de esta sección es describir la metodología y el código GLSL necesario para dibujar una escena con una o varias esferas y un plano con textura de tablero de ajedrez.

Para ello, y debido a la simplicidad que requiere el cálculo de intersecciones rayo-esfera y rayo-plano, calcularemos analíticamente las intersecciones, explicando detalladamente cómo hallarlas en las secciones 8.5.1 y 8.5.3 respectivamente. En el capítulo 9 sustituimos este cálculo analítico por un cálculo iterativo utilizando el algoritmo sphere-tracing, pero a modo introductorio es más sencillo de momento calcular directamente las intersecciones.

Tras esta visualización sencilla, en la sección 8.7 emplearemos además un modelo de iluminación local sencillo pero muy utilizado: el modelo de Phong, el cual se describe con detalle a nivel teórico y a nivel de código. Con esto completaríamos los tres ítems esenciales que mencionamos al inicio de la sección 8.2: rayos primarios, intersecciones y MIL.

8.5.1. Visualizando una esfera

Primero introduciremos código para visualizar una esfera. Por simple geometría euclídea sabemos que, fijado un centro $c = (c_x, c_y, c_z) \in \mathbb{R}^3$ y un radio $r \in \mathbb{R}^+$, una esfera S se define como aquellos puntos de \mathbb{R}^3 tales que su distancia a c es r , es decir:

$$S = \{p \in \mathbb{R}^3 : \|p - c\| = r\} = \{p \in \mathbb{R}^3 : (p - c) \cdot (p - c) = r^2\}.$$

donde en esta definición el operador \cdot denota el producto escalar de \mathbb{R}^3 . Si recordamos que los puntos que componen un rayo R se pueden expresar como $R(t) = p_0 + \vec{v}t$ (donde p_0 es el origen del rayo y el vector \vec{v} su dirección) con valores de t reales no negativos, podemos calcular la intersección rayo-esfera sin más que resolver la ecuación

$$(R(t) - c) \cdot (R(t) - c) = r^2$$

de tal manera que, si resolvemos la ecuación en t podremos saber en qué valor de t golpea el rayo la esfera, si es que efectivamente lo interseca.

$$\begin{aligned}
 (R(t) - c) \cdot (R(t) - c) &= r^2 \\
 (p_0 + \vec{v}t - c) \cdot (p_0 + \vec{v}t - c) &= r^2 \\
 (\vec{v}t + (p_0 - c)) \cdot (\vec{v}t + (p_0 - c)) &= r^2 \\
 (\vec{v} \cdot \vec{v})t^2 + 2(\vec{v} \cdot (p_0 - c))t + (p_0 - c) \cdot (p_0 - c) - r^2 &= 0
 \end{aligned} \tag{8.5}$$

Y esto es una ecuación de segundo grado en t . Esto nos dice que el rayo puede intersecar dos veces con la esfera (secante), una única vez si el discriminante se anula (tangente) o ninguna si el discriminante es negativo (el rayo no interseca con la esfera). En caso de que exista intersección debemos quedarnos con el valor de t más pequeño, que es el más cercano al punto de vista. También debemos quedarnos únicamente con valores de t positivos, pues si es negativo significa que la esfera está detrás del observador, en cuyo caso no es visible.

A nivel de código podemos codificar una esfera como un **struct** cuyos elementos sean su centro y su radio

```

1 struct Sphere{
2     vec3 center;      // Centro de la esfera
3     float radius;    // Radio de la esfera
4 };

```

Además, para el futuro necesitaremos una estructura que almacene información sobre un impacto rayo-superficie. Información como el punto en el que se produce, en qué t , la normal a la superficie en ese punto, etc. En estas fases tan tempranas igual no es tan necesario pero pronto encontraremos su utilidad.

```

1 struct Hit_record {
2     vec3 p;           // Punto donde se produce el impacto
3     vec3 normal;     // Normal a la superficie en el punto p
4     float t;          // Valor de t para el que el rayo impacta
5     bool hit;         // True si se golpea alguna superficie
6 };

```

Claro que los tres primeros campos solo tendrán sentido si el campo **hit** es verdadero, si es falso no tiene sentido calcular ni consultar los demás. A continuación presentamos el código GLSL para visualizar una esfera utilizando estas estructuras.

```

1 // Calcula la intersección entre un rayo y una esfera
2 // y almacena la información del impacto en una
3 // estructura Hit_record
4 Hit_record hit_sphere(Sphere S, Ray R,
5     float t_min, float t_max) {
6
7     Hit_record result;
8     vec3 oc = R.orig - S.center;
9     float a = dot(R.dir, R.dir);
10    float b = 2.0 * dot(oc, R.dir);
11    float c = dot(oc, oc) - S.radius*S.radius;
12    float discriminant = b*b - 4.0*a*c;

```

```

13     if (discriminant < 0.0){
14         result.hit = false;
15         return result;
16     }
17     float sqrtD = sqrt(discriminant);
18     float root = (-b - sqrt(discriminant))/(2.0*a); // Primera raiz
19     if (root < t_min || t_max < root){
20         // La primera raiz esta fuera del rango que nos interesa
21         root = (-b + sqrt(discriminant))/(2.0*a); // La otra raiz
22         if (root < t_min || t_max < root){
23             // Las dos raices estan fuera del rango,
24             // consideramos que no existe impacto
25             result.hit = false;
26             return result;
27         }
28     }
29     result.hit = true;
30     result.t = root;
31     result.p = ray_at(R, result.t);
32     result.normal = normalize((result.p - S.center) / S.radius);
33     return result;
34 }
```

Fijémonos en que hemos introducido unas variables `t_min` y `t_max` de forma que sólo nos interesamos por los puntos del rayo $R(t) = p_0 + \vec{v}t$ para valores de t situados entre un valor mínimo y un máximo. Esto sirve para fijar una distancia mínima y máxima en la que buscar intersecciones, evitando así valores de t negativos o demasiado grandes. El código implementa la ecuación (8.5) de forma que en caso de no haber impacto asigna `false` al campo `hit` de la estructura `Hit_record` y `true` en caso contrario. Además, solo calcula el punto de intersección más cercano.

A partir de la estructura `Hit_record` y la información que contiene podemos asignar el color que deseemos. Lo ideal es evaluar un modelo de iluminación, como haremos en la sección 8.7, pero de momento aprovecharemos la normal en cada punto para mapear una componente de \mathbb{R}^3 normalizada en una terna RGB. Esto es una forma de simular un material conocido popularmente como ‘*normal material*’¹, llamado así por utilizar la normal en un punto para calcular un color.

Realizamos por tanto la primera modificación del código de `ray_color`, que ahora acepta como argumento una esfera.

```

1 // Fijamos una distancia maxima
2 #define MAX_DIST 100.0
3
4 // ...
5
6 vec4 ray_color(Ray R, Sphere S) {
7
8     // R interseca la esfera?
```

¹Consultar por ejemplo la implementación en Three.js para más información <https://threejs.org/docs/#api/en/materials/MeshNormalMaterial>

```

9     Hit_record hr = hit_sphere(S, R, 0.0, MAX_DIST);
10    if(hr.hit){
11        // Nos llevamos las componentes a [0,1]
12        vec3 color = (hr.normal + vec3(1.0))/float(2.0);
13        return vec4(color, 1.0);
14    }
15
16    // R no interseca ninguna superficie
17    vec3 unit_direction = normalize(R.dir);
18    float t = 0.5*(unit_direction.y + 1.0);
19    return vec4((1.0-t)*vec3(1.0,1.0,1.0) + t*vec3(0.5,0.7,1.0),
20                1.0);
21}
22// ...
23// R es el rayo (Ray)
24
25Sphere S;
26S.center = vec3(0.0, 0.0, -1.0); S.radius = 0.5;
27gl_FragColor = ray_color(R, S);

```

El resultado que obtenemos es el que podemos observar en la imagen 8.5. Obsérvese la variedad de colores que ofrece la esfera, concordante con la variedad de normales que posee.

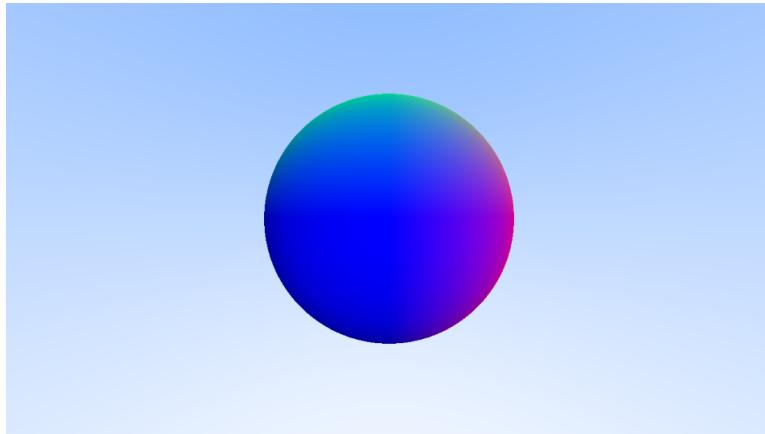


Imagen 8.5: Escena con una esfera

8.5.2. Visualizando varias esferas

Veamos ahora cómo poder visualizar varias esferas, aunque el procedimiento una vez se ha conseguido visualizar una es bastante natural. Podemos declarar en lugar de una un array de esferas, cada una con sus parámetros y crear una función que itere llamando a la función `hit_sphere` con cada esfera. Sin embargo, un rayo puede intersecar varias esferas, pero consideramos que únicamente hemos golpeado la más cercana a efectos prácticos. Este problema se extiende en general a escenas con varios objetos, no sólo esferas, pero la solución es la misma, evaluar únicamente la más cercana de las intersecciones. Aquí es también donde ponemos en valor el parámetro `t_max` de la función `hit_sphere`, pues una vez hemos encontrado una intersección

con una de las esferas no debemos considerar impactos más lejanos.

Por tanto, buscamos implementar una función que dado un array de esferas devuelva en una estructura `Hit_record` la información sobre la intersección rayo-esfera con el valor de t más pequeño. El método consiste en llamar a `hit_sphere` almacenando la información de la intersección pero sólo mantenemos la información de la más próxima, buscando en cada iteración únicamente intersecciones más cercanas que las anteriores (en caso de haberlas).

```
1 // Tamano de los arrays
2 #define ARRAY_TAM 100
3
4 // ...
5
6 Hit_record hit_spheres_list(Sphere spheres[ARRAY_TAM] ,
7     int size, Ray R, float t_min, float t_max) {
8
9     Hit_record result, tmp;
10    result.hit = false;
11    float closest_t = t_max;
12    for(int i = 0; i < ARRAY_TAM; i++){
13        if(i == size) break;
14        // Buscamos tan lejos como la ultima interseccion
15        tmp = hit_sphere(spheres[i], R, t_min, closest_t);
16        if(tmp.hit){ // Hay una interseccion mas cercana
17            closest_t = tmp.t;
18            result = tmp;
19        }
20    }
21    return result;
22 }
```

Y en la función `main` inicializamos algunas esferas, editamos `ray_color` para que acepte como argumento un array de esferas y desde ahí hacemos llamada a `hit_spheres_list`.

```
1 vec4 ray_color(Ray R, Sphere world[ARRAY_TAM], int size) {
2
3     // R interseca alguna superficie?
4     Hit_record hr = hit_spheres_list(world, size, R, 0.0, MAX_DIST);
5     if(hr.hit)
6         return vec4((hr.normal+vec3(1.0))/2.0, 1.0);
7
8     // R no interseca ninguna superficie
9     // Codigo del background ...
10 }
11
12 // ...
13 // R es el rayo (Ray)
14
15 // Esferas
16 int num_spheres = 4;
17 Sphere world[ARRAY_TAM];
18 Sphere S1, S2, S3, S4;
```

```

19 S1.center = vec3(0.0, 0.0, -1.0);    S1.radius = 0.5;
20 S2.center = vec3(-5, 0.5, -3.0);    S2.radius = 4.0;
21 S3.center = vec3(2.0, -3, -4.0);    S3.radius = 1.5;
22 S4.center = vec3(20.0, 10, -20.0);  S4.radius = 3.0;
23 world[0] = S1; world[1] = S2; world[2] = S3; world[3] = S4;
24
25 gl_FragColor = ray_color(R, world, num_spheres);

```

Y así obtenemos la imagen 8.6, en la que podemos ver varias esferas de distintos tamaños y con distintas posiciones.

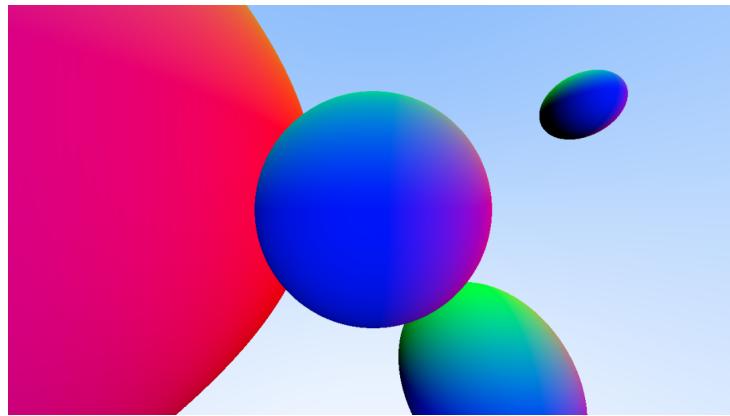


Imagen 8.6: Escena con varias esferas

8.5.3. Visualizando un plano con textura de ajedrez

Para orientarnos en la escena cuando podamos modificar la posición de la cámara dinámicamente y hacernos también a la idea del movimiento que estamos haciendo y a qué velocidad es útil utilizar un plano con textura similar a la de un tablero de ajedrez a modo de suelo. Necesitamos por tanto ahora implementar la intersección rayo-plano y una vez encontrada la intersección discernir entre colorear el punto negro o blanco.

En geometría euclídea una forma de caracterizar los puntos de un plano P es mediante una ecuación lineal

$$P = \{(x, y, z) \in \mathbb{R}^3 : Ax + By + Cz = D\}$$

donde el vector $\vec{N} = (A, B, C) \in \mathbb{R}^3$ es un vector normal al plano $(-(A, B, C)$ también lo sería) y $D \in \mathbb{R}$ es una constante que define la posición del plano. Podemos entonces ver un plano de la siguiente forma equivalente

$$P = \{p = (x, y, z) \in \mathbb{R}^3 : \vec{N} \cdot p = D\}$$

donde de nuevo el punto \cdot denota el producto escalar. Si ahora queremos calcular la intersección del plano P con un rayo $R(t) = p_0 + \vec{v}t$ tenemos que resolver una ecuación que además es lineal:

$$\begin{aligned}
 R(t) \cdot \vec{N} &= D \\
 (p_0 + \vec{v}t) \cdot \vec{N} &= D \\
 t &= \frac{D - p_0 \cdot \vec{N}}{\vec{v} \cdot \vec{N}}
 \end{aligned} \tag{8.6}$$

Lo cual nos dice en qué t se produce la intersección. Obsérvese que en el caso de que la dirección del rayo \vec{v} y la normal al plano \vec{N} sean perpendiculares (*i.e.* $\vec{v} \cdot \vec{N} = 0$), lo cual se traduce en que el rayo es paralelo al plano o está incluido en el mismo, no existe intersección.

Como ya hemos dicho, podemos identificar únicamente un plano a partir de su normal en cualquier punto y una constante $D \in \mathbb{R}$, por lo que mediante una estructura podemos representar en GLSL un plano.

```

1 struct Plane{
2     vec3 normal;      // Vector normal al plano
3     float D;          // Termino independiente
4 };

```

Y a partir de la ecuación (8.6) podemos implementar la función `hit_plane`, que hace lo correspondiente a `hit_sphere` pero acepta como argumento un plano y devuelve la intersección con el mismo.

```

1 Hit_record hit_plane(Plane P, Ray R, float t_min, float t_max){
2     Hit_record result;
3     float oc = dot(P.normal, R.dir);
4     if(oc == 0.0){ // No hay intersección
5         result.hit = false;
6         return result;
7     }
8     float t = (P.D - dot(P.normal, R.orig))/oc;
9     if (t < t_min || t > t_max)
10        result.hit = false;
11    else{
12        result.hit = true;
13        result.t = t;
14        result.p = ray_at(R, result.t);
15        result.normal = normalize(P.normal);
16    }
17    return result;
18 }

```

Y ahora desde `ray_color` debemos considerar la posibilidad de intersecar con el plano o con las esferas, pero solo debemos darle color a la intersección más próxima. Por eso debemos mantener una variable `t_closest` que represente el valor de t más pequeño en el que hemos detectado una intersección. Puede ocurrir que el rayo impacte primero con una esfera y después con el plano o al revés, primero el plano y después una esfera; en estos casos se daría color atendiendo la intersección con la esfera y con el plano respectivamente.

Como es natural, utilizaremos como suelo un plano horizontal, como puede ser el plano $P = \{(x, y, z) \in \mathbb{R}^3 : y = -2\}$, en cuyo caso $\vec{N} = (0, 1, 0)$, $D = -2$. Véamos entonces cómo aplicarle una textura a este plano. Normalmente las texturas se suelen almacenar en imágenes, pero por sencillez en este caso la generaremos proceduralmente desde el fragment shader. Es decir, se evalúa el color de la textura en función del punto en la superficie del objeto, de forma que a cada punto del plano le asignamos un color (blanco o negro).

Por tanto, se trata de un algoritmo que a partir de una coordenada 2D de un punto en el suelo (u, v) se calcula una terna RGB de color, que es el que se asigna finalmente al píxel. Los

puntos de P son de la forma $(x, -2, z)$, con $x, z \in \mathbb{R}$, y utilizaremos como coordenadas (u, v) a los pares (x, z) extraídos de las coordenadas de mundo (x, y, z) de los puntos que forman el plano. Muchas veces, sobre todo en superficies que no son planas o que son irregulares, el mapeo $(x, y, z) \mapsto (u, v)$ es mucho más complejo, pero en este caso es bastante sencillo.

Lo primero es quedarnos con la parte entera de ambos valores, de forma que dividimos el plano en cuadrados. Si la suma de las partes enteras es un número par, colorearemos el píxel de blanco, y si es impar de negro. En la imagen 8.7 podemos ver cómo identificando cada punto con las partes enteras de cada componente y coloreando de negro las partes enteras cuya suma sea impar se obtiene una textura de ajedrez.

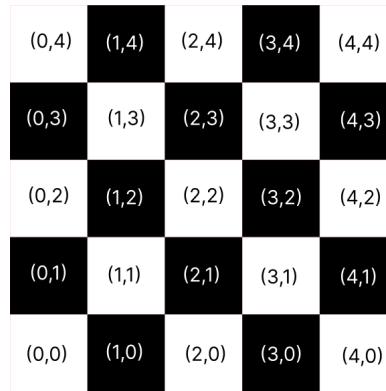


Imagen 8.7: Textura de ajedrez a partir de las partes enteras

Debemos, por tanto, en `ray_color` añadir esta funcionalidad.

```

1 vec4 ray_color(Ray r, Sphere world[ARRAY_TAM] ,
2     int size, Plane P) {
3
4     float t_closest = MAX_DIST;
5     // r interseca alguna esfera?
6     vec4 tmp_color;
7     Hit_record hr = hit_spheres_list(world, size, r, 0.0, t_closest)
8         ;
9     if(hr.hit){
10         t_closest = hr.t;
11         tmp_color = vec4((hr.normal + vec3(1.0))/2.0, 1.0);
12     }
13
14     // r interseca el plano?
15     // Solo intersecciones mas cercanas
16     hr = hit_plane(P, r, 0.0, t_closest);
17     if(hr.hit){
18         t_closest = hr.t;
19         vec3 p = hr.p;
20         int x_int = int(floor(p.x)), // Parte entera de x
21             z_int = int(floor(p.z)), // Parte entera de z
22             sum = x_int + z_int;    // Suma de partes enteras
23         // Modulo 2
24         int modulus = sum - (2*int(sum/2));

```

```

24     if(modulus == 0)      // Suma par
25         tmp_color = vec4(1.0, 1.0, 1.0, 1.0);
26     else                  // Suma impar
27         tmp_color = vec4(0.0,0.0,0.0, 1.0);
28 }
29 // Si r interseca alguna superficie
30 if(t_closest < MAX_DIST) return tmp_color;
31 // r no interseca ninguna superficie
32 // Codigo del background ...
33 }
```

Y tras declarar el plano $y = -2$, las esferas y realizar la llamada a `ray_color`, al fin podemos visualizar una escena completa compuesta de varias esferas y un plano, tal y como nos propusimos al inicio de esta sección 8.5

```

1 // ...
2 // R es el rayo (Ray)
3 // world es el array de objetos 'Sphere'
4
5 // Plano
6 Plane P;
7 P.normal = vec3(0.0, 1.0, 0.0);
8 P.D = -2.0;           // y = -2.0
9
10 gl_FragColor = ray_color(r, world, num_spheres, P);
```

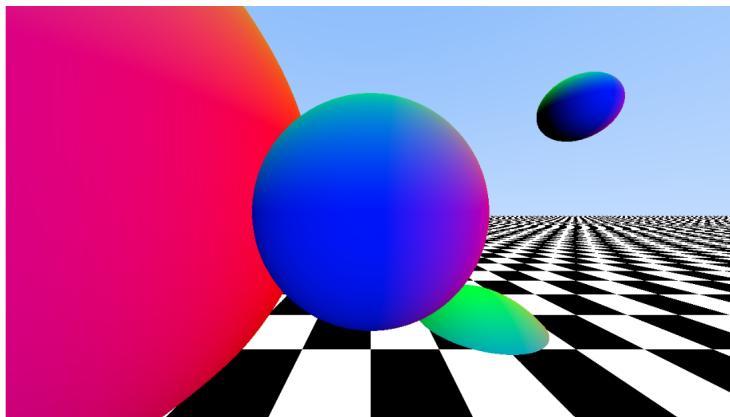


Imagen 8.8: Escena compuesta por esferas y un plano

8.6. Configurando la cámara

Hasta ahora hemos visto cómo componer una escena sencilla y visualizarla, pero desde el principio nos hemos visto limitados por la decisión que tomamos de situar al observador en el punto $(0, 0, 0)$, mirar hacia el $(0, 0, -1)$ y proyectar la escena en un plano de altura 2 y ratio 16 : 9. Evidentemente una aplicación gráfica en la que tanto el observador como la escena se mantienen fijos carece de interés alguno. Debemos añadir la posibilidad de modificar la posición del observador y la dirección en la que se mira, para así poder disfrutar de distintos puntos de vista en una misma escena.

Lo primero es fijar una serie de conceptos. Recordamos que denominábamos *lookfrom* al punto desde el cual se observa (o_c), es decir, el punto de vista; y *lookat* al punto sobre el cual se fija la mirada (a_t). Nótese que realmente el punto a_t puede ser cualquiera situado en la semirecta que une al observador y el centro del plano de proyección, no tiene por qué estar incluido como tal en el propio plano de proyección; pensemos en alguna vez que hemos pensado que alguien nos saludaba y realmente saludaba a alguien que está detrás nuestra.

Se denomina *field of view (FOV)* al ángulo total observable, que corresponde al ángulo θ en la imagen 8.9. Como nuestro plano de proyección no es cuadrado, este ángulo es distinto en vertical y en horizontal, pero del ratio $16 : 9$ ($r = 16/9$) se deduce el uno del otro. En nuestro caso fijaremos $\theta = 90^\circ = \pi/2\text{rad}$, sea $h = \tan(\frac{\theta}{2}) = 1$ el ratio constante que van a mantener la semialtura del plano de proyección y la distancia focal, tal y como se puede observar en la imagen 8.9. Nosotros mantendremos entonces fijas la altura del plano de proyección $P_h = 2 = 2h$ y la distancia focal $d_f = 1 = h$.

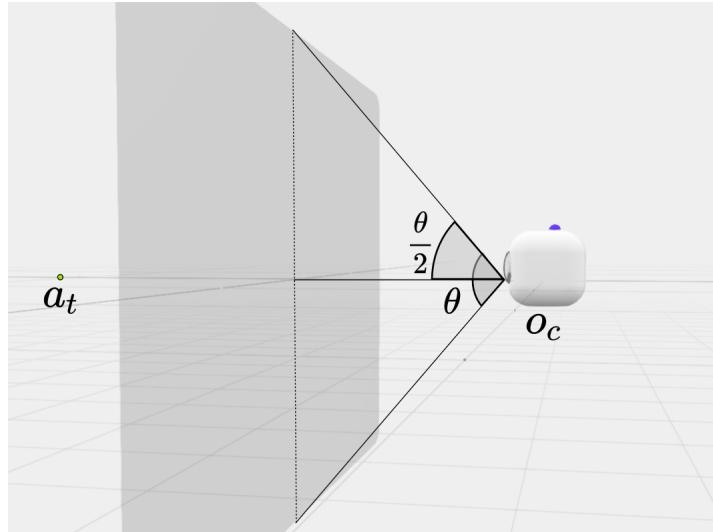


Imagen 8.9: Field Of View

Por otro lado, aunque definamos un punto desde el que mirar y un punto al que mirar, no está todo dicho sobre la orientación de la cámara. Piense que mientras usted lee este documento puede girar la cabeza hacia un lado y seguir mirando desde y hacia la misma posición. Por tanto necesitamos alguna forma de expresarle a la cámara que se mantenga vertical. Esto se hace mediante un vector que se denomina *view up* (\vec{vup}), el cual define ese ‘arriba’ para la cámara. Normalmente, por convención se hace la elección $\vec{vup} = (0, 1, 0)$. Mediante el vector $\vec{v}_d = o_c - a_t$ (*view direction*) y el vector \vec{vup} podemos crear un sistema de referencia ortonormal del plano ortogonal a \vec{v}_d que pasa por el punto o_c y que define la orientación de la cámara. No confundamos este plano con el plano de proyección, pues aunque vectorialmente son iguales, y son por tanto paralelos desde el punto de vista afín, son planos distintos. Precisamente de este hecho nos aprovecharemos más tarde. Presentamos entonces los siguientes vectores

$$\begin{aligned}\vec{w} &= \frac{o_c - a_t}{\|o_c - a_t\|}, \\ \vec{u} &= \frac{\vec{vup} \times \vec{w}}{\|\vec{vup} \times \vec{w}\|}, \\ \vec{v} &= \vec{w} \times \vec{u},\end{aligned}$$

de forma que los vectores \vec{u}, \vec{v} son, por su propia definición ortogonales a \vec{v}_d y ortogonales entre sí. Además están normalizados, constituyendo así una base ortonormal del plano vectorial que define el plano de proyección, por lo que hacen las veces de los vectores $(1, 0, 0)$ y $(0, 1, 0)$ en el código de `get_ray` en la sección 8.3, mientras que \vec{w} por su parte toma el relevo del vector $(0, 0, 1)$. Fíjese para mejor comprensión en la imagen 8.10. El conjunto $\{o_c; \vec{u}, \vec{v}, \vec{w}\}$ es por tanto un sistema de referencia del plano afín paralelo al plano de proyección que pasa por el punto o_c , al cual se le suele denominar *marco de coordenadas de vista*.

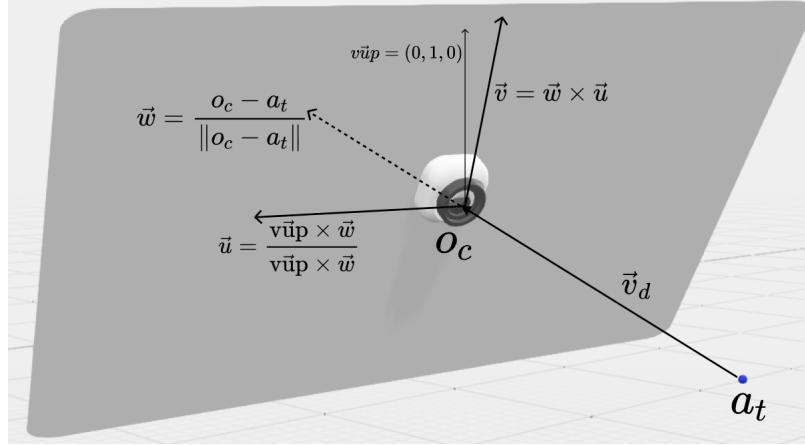


Imagen 8.10: Vectores que forman una base del plano de proyección

Ahora recordamos que para crear el rayo en la sección 8.3 calculamos la esquina inferior izquierda, transformamos las coordenadas de dispositivo en coordenadas de mundo del plano de proyección en $[0, P_w] \times [0, P_h] \times \{-1\}$ y con ello calculamos el punto destino. En el caso generalizado la metodología es la misma, pero necesitamos aclarar, a partir de los datos de entrada como son el punto o_c , el punto a_t , el vector \overrightarrow{vup} (usualmente $(0, 1, 0)$), el ángulo $\widehat{\text{fov}}$ y el ratio r ancho/alto (16/9 en nuestro caso), cuáles son las dimensiones del plano de proyección y la esquina inferior izquierda. Para ello introducimos una estructura que representará una cámara, que almacena los siguientes campos:

```

1 struct Camera{
2     vec3 origin;           // Punto desde el que se observa
3     vec3 horizontal;       // Vector horizontal de modulo P_w
4     vec3 vertical;         // Vector vertical de modulo P_h
5     vec3 lower_left_corner; // Punto situado en la esquina
6 };
```

donde `origin` son las coordenadas de mundo del punto en el que se sitúa la cámara, `horizontal` es un vector cuyo módulo es el ancho del plano de proyección y su dirección el vector u recién presentado, análogamente `vertical` es un vector cuyo módulo es la altura del plano de proyección y su vector director es v , por último `lower_left_corner` son las coordenadas de mundo del punto situado en la esquina inferior izquierda del plano de proyección. Fíjese en la imagen 8.11.

Es claro que el origen es el punto *lookfrom* (o_c), el vector horizontal es $P_w \cdot \vec{u}$ y el vector vertical es $P_h \cdot \vec{v}$. Por tanto, asumiendo una distancia focal de 1, la esquina inferior izquierda sería

$$\text{LLC} = o_c - \frac{P_w}{2} \cdot \vec{u} - \frac{P_h}{2} \cdot \vec{v} - \vec{w}.$$

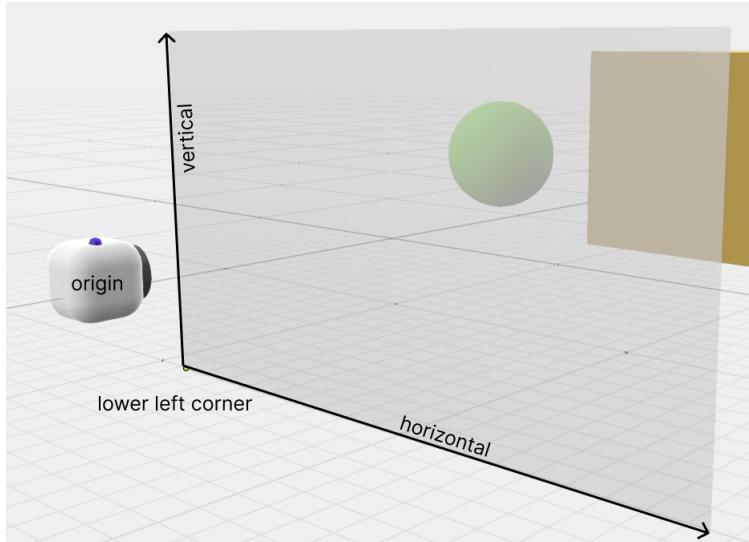


Imagen 8.11: Representación gráfica de los campos de ‘Camera’

En base a estos elementos, volvemos a redefinir la transformación ϕ , de forma que ahora el plano de proyección P puede tener cualquier posición y orientación. A partir de los puntos o_c y a_c y el vector \overrightarrow{vup} podríamos calcular los vectores $\vec{u}, \vec{v}, \vec{w}$, y como P es paralelo al plano de la imagen 8.10, tenemos que $\mathcal{R} = \{LLC; \vec{u}, \vec{v}, \vec{w}\}$ es un marco de referencia del plano de proyección, de forma que podemos obtener un punto del plano de proyección P a partir de las coordenadas de dispositivo aplicando la siguiente transformación.

$$\begin{aligned} \phi : [0, n_x] \times [0, n_y] &\longrightarrow P \\ (x, y) &\longmapsto LLC + \frac{P_w}{n_x} x \cdot \vec{u} + \frac{P_h}{n_y} y \cdot \vec{v}. \end{aligned} \tag{8.7}$$

También podemos expresar esta transformación matricialmente. En este caso, la inversa de la matriz M de vista y proyección se obtiene mediante el producto de la matriz de cambio de marco de referencia entre el marco usual y el marco \mathcal{R} y una matriz análoga a la de la transformación (8.4). Realmente esto es intuitivo, pues no es más que un cambio de posición y orientación respecto al caso base.

$$\phi(x, y) = \left(\begin{array}{ccc|c} \vec{u} & \vec{v} & \vec{w} & LLC \\ \hline 0 & 0 & 0 & 1 \end{array} \right) \left(\begin{array}{ccc|c} \frac{P_w}{n_x} & 0 & 0 & 0 \\ 0 & \frac{P_h}{n_y} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array} \right) \left(\begin{array}{c} x \\ y \\ 0 \\ 1 \end{array} \right) \tag{8.8}$$

Reflejamos estos cálculos y asignaciones en el siguiente código, que corresponde a una función que dados los parámetros de una escena, crea, inicializa y devuelve un objeto **Camera**.

```

1 | Camera init_camera (vec3 lookfrom, vec3 lookat,
2 |     vec3 vup, float vfov,
3 |     float aspect_ratio){
4 |
5 |     Camera cam;
6 |     float focal_length = 1.0;

```

```

7   float theta = degrees_to_radians(vfov); // FOV vertical
8   float h = tan(theta/2.0);
9   // Dimensiones del plano de proyección
10  float viewport_height = 2.0*h*focal_length;
11  float viewport_width = aspect_ratio * viewport_height;
12
13  vec3 w = normalize(lookfrom - lookat);
14  vec3 u = normalize(cross(vup,w));
15  vec3 v = cross(w,u);
16
17  cam.origin = lookfrom;
18  cam.horizontal = viewport_width * u;
19  cam.vertical = viewport_height * v;
20  cam.lower_left_corner = cam.origin
21      - cam.horizontal/float(2.0)
22      - cam.vertical/float(2.0)
23      - w;
24
25  return cam;
26 }
```

Y una vez tenemos estos parámetros almacenados en un objeto `Camera` la función `get_ray`, a la cual ahora en lugar de todos los parámetros con la cual la programamos primitivamente en la sección 8.3 la reprogramaremos para que acepte únicamente como argumentos la cámara y las coordenadas de dispositivo normalizadas $[0, 1]$. Tomará estas coordenadas $u, v \in [0, 1]$ y multiplicará por los vectores horizontal y vertical respectivamente, definiendo así el destino del rayo.

```

1 Ray get_ray(Camera cam, float u, float v){
2     Ray R;
3     R.orig = cam.origin;
4     R.dir = cam.lower_left_corner
5         + u*cam.horizontal + v*cam.vertical
6         - cam.origin;
7     return R;
8 }
```

Vemos que ahora son los vectores `cam.horizontal` y `cam.vertical` los que hacen la función que en la primera implementación hacían los vectores `vec3(viewport_width, 0.0, 0.0)` y `vec3(0.0, viewport_height, 0.0)`.

Con todo este código, tan solo tenemos que fijar los argumentos de `init_camera` y observar cómo cambia el punto de vista aunque la escena sea la misma. En principio vamos a mantener constantes el $\widehat{\text{fov}} = 90^\circ$, el vector $\overrightarrow{\text{vup}} = (0, 1, 0)$ y el ratio 16:9, pero como nos gustaría poder movernos con libertad por la escena, declararemos dos variables `uniform` que representen el punto o_c y el punto a_t , a las cuales podremos asignarle variables dinámicamente desde JavaScript y dar la sensación de movimiento.

```

1
2 uniform vec3 u_lookfrom;    // Punto lookfrom
3 uniform vec3 u_lookat;     // Punto lookat
```

```

4
5 // ...
6 // aspect_ratio = 16/9
7 // nx = 1280 pixels
8 // ny = 720 pixels
9
10 // CAMARA
11 vec3 vup = vec3(0.0, 1.0, 0.0);
12 float vfov = 90.0; // Vertical field of view in degrees
13 Camera cam = init_camera(u_lookfrom, u_lookat,
14     vup, vfov, aspect_ratio);
15
16 // Coordenadas de dispositivo normalizadas [0,1]
17 vec2 uv = (gl_FragCoord.xy + vec2(0.5)) / vec2(nx, ny);
18 float u = uv.x;
19 float v = uv.y;
20
21 Ray r = get_ray(cam, u, v);

```

Por ejemplo, si fijamos $o_c = (1, 1, 1)$ y $a_t = (0, 0, 0)$ obtenemos la imagen 8.12 (a), y con $o_c = (5, 5, -5)$, $a_t = (1, 2, -5)$ la imagen 8.12 (b). En este último caso podemos de hecho ver la escena desde el lado negativo del eje Z , lo que antes de parametrizar la cámara llamábamos ‘el interior de la pantalla’.

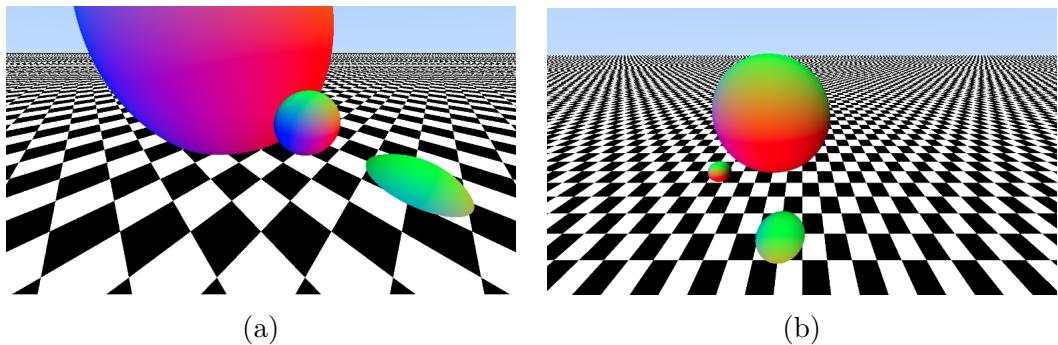


Imagen 8.12: Escena 8.8 desde otros puntos de vista

8.7. Modelo de iluminación de Phong

En este punto hemos creado una escena sencilla y podemos movernos libremente por ella, pero vendría bien una dosis de realismo a la escena. Para ello, dotaremos a la escena de fuentes de luz y añadiremos a los objetos que componen la escena un material que nos permita observar su aspecto más allá del material normal que definimos en la sección 8.5.1 a la hora de darle color a la esfera.

La iluminación real es imposible de simular porque cada punto de cada objeto irradia una cantidad de luz en todas las direcciones que es inviable de computar, por ello necesitamos de aproximaciones que simulen en mayor o menor medida dicha iluminación. Existen muchos y muy distintos modelos de iluminación, tan complejos y realistas como queramos. El conocido *modelo de Phong* no es ni el más realista, ni el más complejo, pero es suficiente para nuestro cometido,

pues nos permite darle colores a los materiales y apariencias mate además de posibles brillos que consiguen efectos metalizados.

Primero de todo, haremos una clasificación de los distintos modelos de iluminación que se suelen utilizar en informática gráfica en base a la luz que recibe un punto de una superficie, la cual puede deberse a:

- **Iluminación directa:** Provocada por una fuente de luz que incide directamente sobre dicho punto.
- **Iluminación indirecta:** Luz que llega a la superficie tras haber rebotado en otras superficies.

Como consecuencia de esta clasificación, los modelos de iluminación pueden clasificarse en:

- **Modelos locales:** Únicamente consideran la iluminación directa
- **Modelos globales:** Consideran tanto iluminación directa como indirecta.

El modelo de Phong es un modelo de iluminación local, es decir, tan solo consideraremos la posible acción directa de una fuente de luz sobre los objetos, lo cual es además computacionalmente más sencillo. Este modelo se basa en descomponer la luz que incide sobre un punto de un objeto en tres componentes RGB: ambiental, difusa y especular.

Consideraremos también únicamente el efecto de luces direccionales, las cuales se componen, tal y como su propio nombre indica, de un vector director $\vec{L}_p \in \mathbb{R}^3$ y de una triplete RGB que es la intensidad de la luz I_p . El rasgo principal de este tipo de fuentes de luz es que inciden sobre todos los puntos del objeto con el mismo vector, a diferencia de las fuentes de luz puntuales. Por conveniencia, consideraremos que el vector \vec{L}_p y todos los vectores que utilizaremos están normalizados. Sean entonces k fuentes de luz, cada una con su vector director \vec{L}_p y su intensidad I_p , $p = 1, \dots, k$.

En adelante fijaremos un punto cualquiera de una superficie el cual queremos calcular el color que asignarle al píxel correspondiente, calculando el mismo mediante las ecuaciones del modelo de Phong. A continuación describiremos cada una de las componentes y explicaremos su efecto.

8.7.1. Componente ambiental

La componente ambiental representa la luz del entorno y afecta uniformemente a todos los puntos del objeto independientemente de la forma de éste. Por ejemplo, en un día de calima todos los objetos en todos sus puntos tenían una componente ambiental naranja. Este valor I_a , que denominamos iluminación ambiental, se deduce del producto (componente a componente) de las siguientes tripletas:

- Intensidad de la luz ambiente ($I_{la} \in \mathbb{R}^3$): Es la media de las intensidades de las luces de la escena: $I_{la} = \frac{1}{k} \sum_{p=1}^k I_p$.
- Reflectividad ambiental del material ($k_a \in \mathbb{R}^3$): Depende únicamente del material de la superficie y expresa la respuesta del material a este tipo de iluminación.

$$I_a = I_{la}k_a, \quad (8.9)$$



Imagen 8.13: La calima: un ejemplo del efecto de la componente ambiental

8.7.2. Componente difusa

Cuando la luz impacta sobre un objeto, si este es opaco la luz es reflejada a muchas direcciones, si es translúcido entra en el objeto. Esta componente representa la cantidad de luz que es reflejada. Nosotros consideraremos que la misma cantidad de luz que incide es la que se refleja en todas las direcciones (los objetos son totalmente opacos), y esta depende tanto del ángulo de incidencia como de la normal a la superficie en dicho punto. A esta componente también se le llama reflexión de Lambert por su relación con el modelo de Lambert.

Fijada una fuente de luz, la iluminación difusa I_d se obtiene mediante el producto de la reflectividad difusa del material $k_d \in \mathbb{R}^3$, la intensidad de la luz I_p y el coseno del ángulo que forman la normal al punto (\vec{N}) y la dirección de la fuente de luz (\vec{L}_p), el cual, si \vec{N} y \vec{L}_p están normalizados se calcula simplemente como el producto escalar $\vec{N} \cdot \vec{L}_p$ (véase imagen 8.14). Por tanto,

$$I_{d_p} = I_p k_d (\vec{N} \cdot \vec{L}_p) \quad (8.10)$$

La componente difusa final es la suma de las componentes difusas calculadas para cada una de las fuentes de luz.

$$I_d = \sum_{p=1}^k I_p k_d (\vec{N} \cdot \vec{L}_p) \quad (8.11)$$

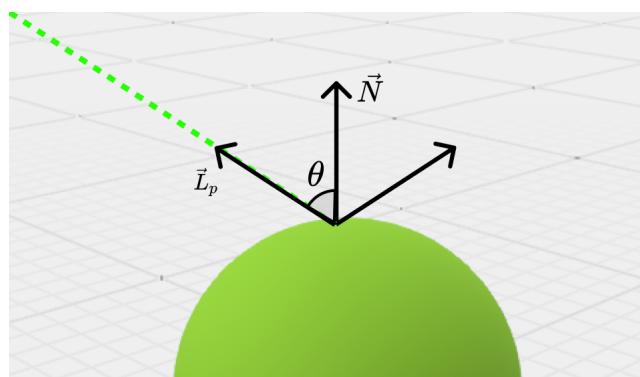


Imagen 8.14: Esquema de los vectores utilizados en la componente difusa

Fijémonos que este cálculo es independiente de la posición de la cámara, solo depende de la superficie y la fuente de luz.

8.7.3. Componente especular

Esta componente representa a las reflexiones directas de la fuente de luz sobre un objeto con brillo. Es la intensidad especular mediante la que conseguimos efectos brillantes y metalizados. La percepción de la iluminación especular depende ahora sí de la posición de la cámara respecto a la superficie. Concretamente, fijada una fuente de luz, la componente especular se ve afectada por el ángulo α que forman el vector que une el punto de la superficie que estamos evaluando con el punto de vista, llamémoslo \vec{V} , y la dirección que tomaría un vector reflejado por la superficie proveniente de la fuente de luz, \vec{R}_p . Véase la imagen 8.15 para mayor claridad. El tamaño del resplandor que causan estos brillos se regula con una constante real $n \in \mathbb{R}$ que es parte de las propiedades del material.

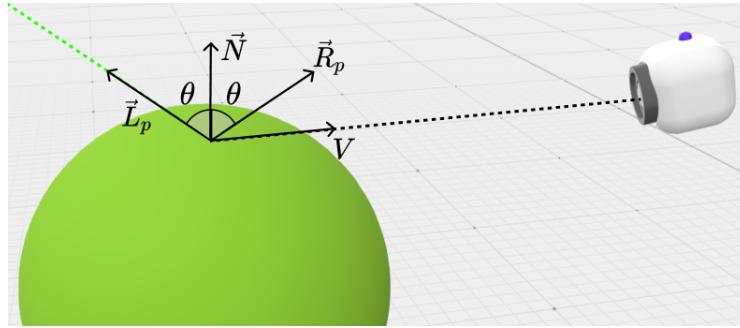


Imagen 8.15: Esquema de los vectores utilizados en la componente especular

A partir de toda esta información, la iluminación especular I_s se calcula con el producto de la reflectividad especular del material k_s , la intensidad de la luz I_p y el coseno del ángulo α elevado a n . Asumiendo que \vec{R}_p y \vec{V} son unitarios, $\cos \alpha = \vec{R}_p \cdot \vec{V}$, por lo que tenemos que

$$I_{sp} = I_p k_s (\vec{R}_p \cdot \vec{V})^n \quad (8.12)$$

Al igual que en el caso de la componente difusa, el resultado final es la suma de las componentes especulares generadas por todas las fuentes de luz.

$$I_s = \sum_{p=1}^k I_p k_s (\vec{R}_p \cdot \vec{V})^n \quad (8.13)$$

8.7.4. Sombras arrojadas

Hay algunas situaciones en las que no se aplica iluminación difusa ni especular, y es en el caso de que la luz no incida directamente sobre el punto. En este caso, al no incidir la luz se considera que el punto está en la sombra provocada por dicha luz. Hay varias formas de saber si un punto concreto está o no en la sombra. Pensemos primero en puntos que están ‘en la cara oculta’ de un objeto. Es decir, pertenecen a una zona de un objeto sobre la cual una fuente de luz concreta no incide. Para determinar esto se puede utilizar la normal a la superficie en dicho punto, \vec{N} , y el vector dirección de la fuente de luz, \vec{L}_p . Si el coseno del ángulo θ , que recordemos que es el vector formado por \vec{N} y \vec{L}_p , es negativo, entonces consideraremos que la luz no incide sobre el punto, véase la imagen 8.16.

El problema que esta condición es suficiente únicamente en casos de objetos convexos, como las esferas, ya que si la superficie tiene puntos tapados por el propio objeto el cálculo afirmaría

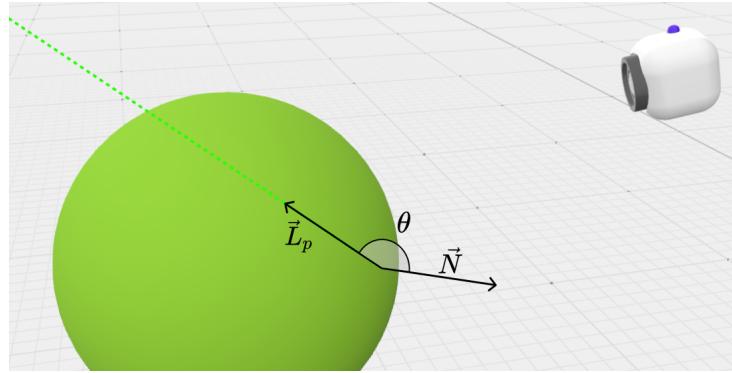


Imagen 8.16: Punto sobre el que no incide la luz

que la luz incide sobre el mismo cuando realmente no es así. Fíjese en el caso del tubo de la figura 8.17.

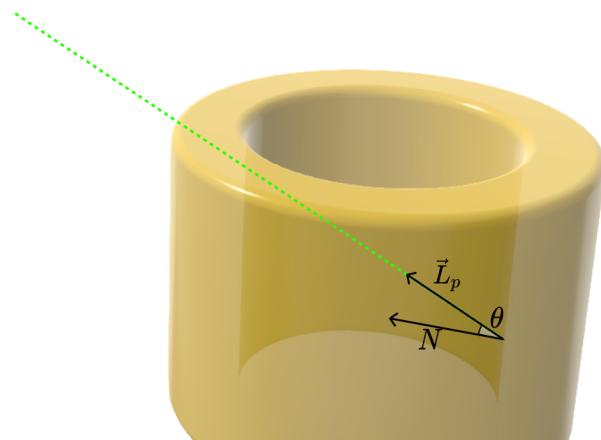


Imagen 8.17: Ejemplo de objeto no convexo y punto a la sombra

En estos casos, la forma de saber si el punto está o no a la sombra es lanzar un rayo en la misma dirección que la luz y averiguar si existe alguna intersección con algún objeto de la escena. En caso de que la haya multiplicamos por 0 el valor de la componente difusa y la especular (el punto está a la sombra). Si no existe impacto se establece que la luz incide directamente sobre el punto y se multiplicarían por 1 las componentes difusa y especular. Este proceso debe repetirse con cada una de las fuentes de luz que se utilicen.

Por ejemplo, fíjese en la imagen 8.18. Si se lanza un rayo desde el punto p_0 , este no encontrará intersección con ningún objeto, por lo que se considera que la luz incide sobre él. Por su parte, p_1 está en la parte del objeto sobre la cual no impacta la luz, y si además lanzamos un rayo en dirección a la fuente de luz intersecaría con la propia esfera. Y finalmente, al lanzar un rayo desde p_2 éste interseca con la esfera, por lo que se considera que está a la sombra.

Por cualquiera de las dos razones, un punto puede ocurrir que esté a la sombra, en cuyo caso no se calcularían o se anularían las componentes difusa y especular del modelo. Por tanto, las intensidades difusa y especular se redefinen multiplicando cada una de las dos por una constante binaria para cada luz, c_p , la cual vale 0 si el punto evaluado está a la sombra o 1 si la luz incide sobre el mismo.

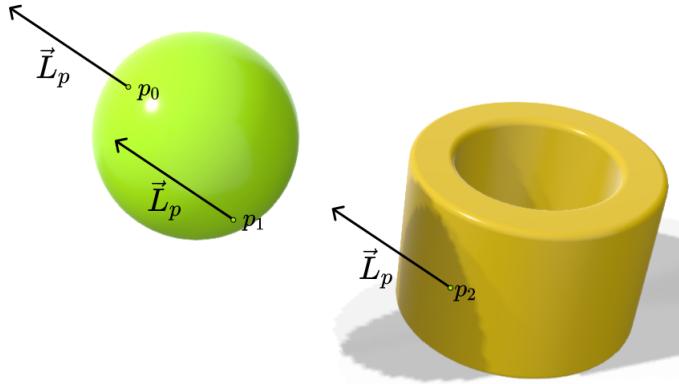


Imagen 8.18: Ejemplos de puntos sobre los que incide y no una fuente de luz

$$\begin{aligned}
 I_{dp} &= c_p I_p k_d (\vec{N} \cdot \vec{L}_p) & I_d &= \sum_{p=1}^k c_p I_p k_d (\vec{N} \cdot \vec{L}_p) \\
 I_{sp} &= c_p I_p k_s (\vec{R}_p \cdot \vec{V})^n & I_s &= \sum_{p=1}^k c_p I_p k_s (\vec{R}_p \cdot \vec{V})^n
 \end{aligned}$$

8.7.5. Evaluación del modelo de iluminación

Una vez se han explicado las tres componentes y cómo se calculan podemos evaluar el modelo de iluminación completo sumando las tres componentes.

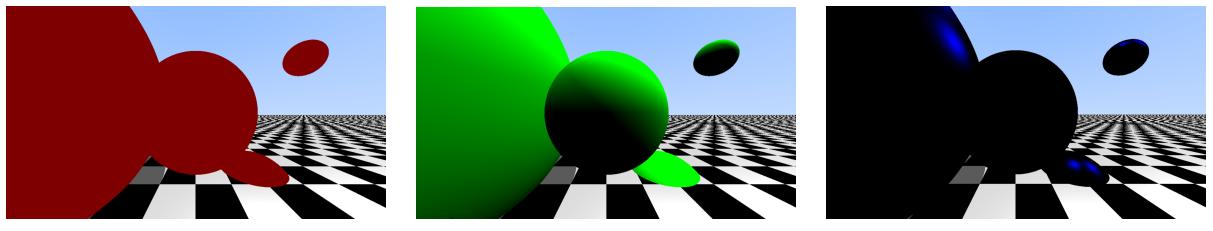
$$\begin{aligned}
 I &= I_a + I_d + I_p \\
 &= \frac{1}{k} \sum_{p=1}^k I_p k_a + \sum_{p=1}^k c_p I_p k_d (\vec{N} \cdot \vec{L}_p) + \sum_{p=1}^k c_p I_p k_s (\vec{R}_p \cdot \vec{V})^n \\
 &= \sum_{p=1}^k \left(\frac{1}{k} I_p k_a + c_p I_p k_d (\vec{N} \cdot \vec{L}_p) + c_p I_p k_s (\vec{R}_p \cdot \vec{V})^n \right)
 \end{aligned} \tag{8.14}$$

A modo de adelanto, pero también con el objetivo de aclarar conceptos, obsérvese cómo afecta cada componente del modelo. En las imágenes 8.19 vemos el efecto de cada tipo de iluminación por separado. En la imagen (a) observamos en rojo que el efecto de la parte ambiental afecta por igual a todos los puntos, de forma que realmente no da ninguna sensación de volumen. En (b) se puede ver cómo la componente difusa colorea de forma mate en color verde el objeto. Por último, en (c) se observan únicamente brillos especulares azules.

Y ahora observemos una imagen en la que se aprecian las tres componentes a la vez: la imagen 8.20. En las esferas predomina el verde, pero se aprecian brillos azules y sombras rojas, pudiendo apreciar perfectamente qué papel desarrolla cada componente del modelo. Fijémonos como efectivamente en los puntos que no incide la luz se ven rojos por el efecto de la componente ambiental.

8.7.6. Implementación del modelo de Phong

Veamos ahora cómo a partir de la teoría introducida podemos obtener resultados como los de las imágenes 8.19 y 8.20. El elemento básico de un modelo de iluminación es, como no podría ser



(a) Ambiental

(b) Difusa

(c) Especular

Imagen 8.19: Componentes aisladas del modelo de Phong

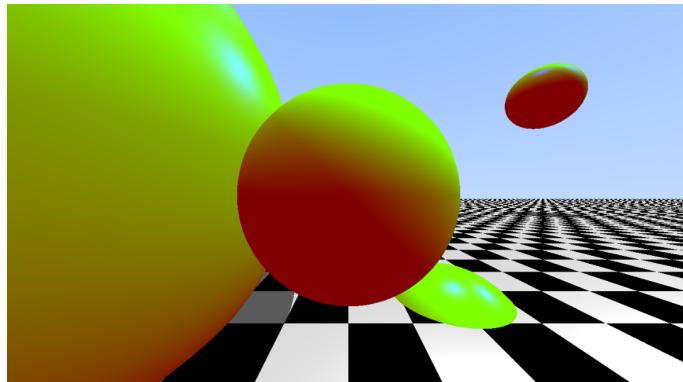


Imagen 8.20: Acción de todas las componentes

de otro modo, la fuente de luz. Una fuente de luz ya dijimos que se compone de un vector director y una triplete RGB que define su intensidad, pero en GLSL los colores vienen representados por 4 componentes, siendo la última la que corresponde a transparencia. Por simplicidad nosotros la consideraremos siempre igual a 1.0. Por tanto, representamos en GLSL una fuente de luz direccional mediante el siguiente struct.

```

1 | struct Directional_light{
2 |     vec3 dir;    // Dirección de la luz
3 |     vec4 color; // Intensidad RBGA de la luz
4 | };

```

Lo siguiente es fijarnos en que el material del objeto desempeña una labor fundamental, de hecho es el que define finalmente la apariencia del mismo. Este material tiene como elementos las reflectividades ambiental (k_a), difusa (k_d) y especular (k_s), las cuales son también tuplas RGBA. Además de las reflectividades, el exponente de brillo n también es una propiedad del material. De esta forma, podemos representar un material con un struct tal que así:

```

1 | struct Material {
2 |     vec4 ka;      // Reflectividad ambiental
3 |     vec4 kd;      // Reflectividad difusa
4 |     vec4 ks;      // Reflectividad especular
5 |     float sh;    // Exponente de brillo
6 | };

```

Como ya hemos comentado, cada objeto tiene un único material, por lo que añadimos a las estructuras `Sphere` y `Plane` un nuevo campo `Material mat`; de forma que a cada esfera y al plano se le puede asignar un material. Además, en caso de intersección almacenábamos

información en una estructura `Hit_record`, y esa información es la que nos servía para calcular el color. De igual manera que necesitábamos por ejemplo la normal ahora necesitamos saber el material del objeto con el que impacta el rayo, por lo que añadimos a `Hit_record` también un campo `Material mat`; al cual se le asignaría el material del objeto con el que se produce el choque.

Para calcular la constante c_p con la que conseguir el sombreado necesitamos ahora una función que, dado un punto, trace un rayo en dirección de una fuente de luz dada y devuelva 0 si hay intersección con algún objeto y 1 si no la hay. Fijémonos que debemos calcular intersecciones que estén a una pequeña distancia del punto, ya que en otro caso las funciones que calculan la intersección siempre calcularían que existe impacto (pues el punto que se evalúa está en una superficie), de forma que todos los puntos estarían a la sombra, cosa que no tiene sentido. Además, consideramos que solo las esferas arrojan sombras.

```

1 float light_is_visible(Directional_light light, vec3 p,
2     Sphere world[ARRAY_TAM], int size) {
3     Ray R;
4     R.dir = normalize(light.dir);
5     R.orig = p;
6     float t = 0.01;
7     float h;
8
9     return hit_spheres_list(world, size, R, t, MAX_DIST).hit ?
10        0.0 : 1.0;
11 }
```

Además, con el objetivo de añadir eficiencia e interactividad se ha añadido una variable `uniform` de tipo `bvec3`, es decir, una tripleta de booleanos, que sirve para especificar desde JavaScript qué luces queremos que proyecten sombras. Así podemos proyectar las sombras de la luz que queramos cuando busquemos realismo o desactivarlas cuando queramos mayor rapidez en la ejecución, bajo la limitación de que solo podremos utilizar hasta 3 fuentes de luz.

```
1 uniform bvec3 u_shadows;
```

Con estas variaciones ya podríamos implementar una función que evalúe el modelo de iluminación completo a partir de la ecuación (8.14). Esta función necesita como argumentos las fuentes de luz, el material, el punto en el que se evalúa el modelo, la normal a la superficie en dicho punto, los objetos que arrojan sombras y el punto o_c desde el que se observa. Toda esta información puede ser recogida en un array de objetos `Directional_light`, en la estructura `Hit_record` que almacena la información del choque, en la variable global `u_lookfrom` y el vector de esferas. Presentamos por tanto el código de dicha función.

```

1 vec4 evaluate_lighting_model(
2     Directional_light lights[ARRAY_TAM], int num_lights,
3     Hit_record hr,
4     Sphere world[ARRAY_TAM], int size ) {
5
6     Material mat = hr.mat;
7     Directional_light light;
8     vec3 light_dir;
9     vec3 view_dir = normalize(u_lookfrom - hr.p);
```

```

10    vec3 normal = normalize(hr.normal);
11    vec4 ambient, diffuse, specular;
12    float visibility;
13    vec4 L_in = vec4(0.0, 0.0, 0.0, 1.0);
14    if(num_lights > 0){
15        for(int i = 0; i < ARRAY_TAM; i++){
16            if(i == num_lights) break;
17            ambient = vec4(0.0, 0.0, 0.0, 1.0);
18            diffuse = vec4(0.0, 0.0, 0.0, 1.0);
19            specular = vec4(0.0, 0.0, 0.0, 1.0);
20            light = lights[i];
21            ambient += mat.ka*light.color;
22            if(num_lights <= 3 && !u_shadows[i]) visibility = 1.0;
23            else
24                visibility = light_is_visible(light, hr.p, world, size);
25            light_dir = normalize(light.dir);
26            float cos_theta = max(0.0,dot(normal,light_dir));
27            // Solo si la fuente de luz es visible desde el punto
28            if(cos_theta > 0.0 && visibility > 0.0) {
29                // Reflection direction
30                vec3 reflection_dir = reflect(-light_dir, normal);
31                diffuse = mat.kd * cos_theta;
32                specular = mat.ks * pow( max(0.0,
33                    dot(reflection_dir, view_dir)),
34                    mat.sh);
35            }
36            L_in += visibility * light.color * (diffuse + specular);
37        }
38    }
39    L_in += ambient/float(num_lights);
40    return vec4(L_in.xyz, 1.0);
41 }

```

Lo siguiente es editar, aunque muy levemente, las funciones `hit_sphere` y `hit_plane` para que en caso de existir intersección asignen a `result.mat` el material de la esfera `S.mat` y el del plano `P.mat` respectivamente. Por último debemos, una vez más, editar `ray_color` para que acepte como argumento el array de luces y hacer las llamadas correspondientes a la función `evaluate_lighting_model`.

Sin embargo, recordemos que el suelo tenía textura de tablero de ajedrez, por lo que no podemos asignar de manera uniforme un material al plano. Lo que sí podemos es editar los parámetros del material utilizado para el suelo asignando a sus componentes blanco o negro según corresponda, aunque a partir de este momento y por pura estética se utilizará un color gris claro `rgb(178,178,178)` en lugar del blanco.

```

1 vec4 ray_color(Ray r, Sphere world[ARRAY_TAM], int size,
2     Plane P, Directional_light lights[ARRAY_TAM],
3     int num_lights) {
4
5     // r interseca alguna esfera?
6     // ...

```

```

7  if(hr.hit){
8      // ...
9      tmp_color = evaluate_lighting_model(lights,
10         num_lights, hr, world, size);
11 }
12
13 // r interseca el plano?
14 // ...
15 if(hr.hit){
16     // ...
17     if(modulus == 0){
18         hr.mat.kd = vec4(0.5, 0.5, 0.5, 1.0);
19         hr.mat.ks = vec4(0.5, 0.5, 0.5, 1.0);
20         hr.mat.sh = 10.0;
21     }
22     else
23         hr.mat.kd = vec4(0.0,0.0,0.0, 1.0);
24     tmp_color = evaluate_lighting_model(lights,
25         num_lights, hr, world, size);
26 }
27 // ...
28 }
```

Y ya por último tan solo tenemos que inicializar las luces y los materiales. En el caso de las luces, hemos utilizado dos fuentes de luz y hemos añadido dos variables `uniform` para elegir los colores de ambas, aunque inicialmente serán blancas. Las direcciones de estas luces son $(1, 1, 0)$ y $(0, 1, 0)$. Respecto a los materiales, el del plano únicamente tiene la componente difusa y especular que se le asigna en `ray_color` y el de las esferas es editable por el usuario, de forma que tenemos 4 variables `uniform` que son editables dinámicamente y a partir de las cuales se inicializa el material de las esferas.

```

1 // Material parametrizable
2 uniform vec4 u_ka;
3 uniform vec4 u_kd;
4 uniform vec4 u_ks;
5 uniform float u_sh;
6
7 // Colores de las luces
8 uniform vec4 u_lightColor0;
9 uniform vec4 u_lightColor1;
10
11 // ...
12
13 // Materiales
14 Material mat, ground_material;
15 mat.ka = u_ka;
16 mat.kd = u_kd;
17 mat.ks = u_ks;
18 mat.sh = u_sh;
19
```

```

20 ground_material.ka = vec4(0.0, 0.0, 0.0, 1.0);
21 ground_material.ks = vec4(0.0, 0.0, 0.0, 1.0);
22 ground_material.sh = 1.0;
23
24 // Esferas
25 // ...
26 S1.mat = mat; S2.mat = mat;
27 S3.mat = mat; S4.mat = mat;
28 // ...
29
30 // Plano
31 // ...
32 P.mat = ground_material;
33 // ...
34
35 // Luces
36 Directional_light lights[ARRAY_TAM];
37 int num_lights = 2;
38 Directional_light l0, l1;
39 l0.color = u_lightColor0;
40 l1.color = u_lightColor1;
41 l0.dir = vec3(1.0, 1.0, 0.0);
42 l1.dir = vec3(0.0, 1.0, 0.0);
43 lights[0] = l1; lights[1] = l2;
44 // ...
45
46 gl_FragColor = ray_color(r, world, num_spheres,
    P, lights, num_lights);

```

Evidentemente todos estos parámetros se pueden modificar y poner los que queramos, para finalmente obtener las imágenes y la coloración que deseemos. Como ejemplo, obsérvese el resultado obtenido al modificar algunos de los parámetros en la imagen 8.21.

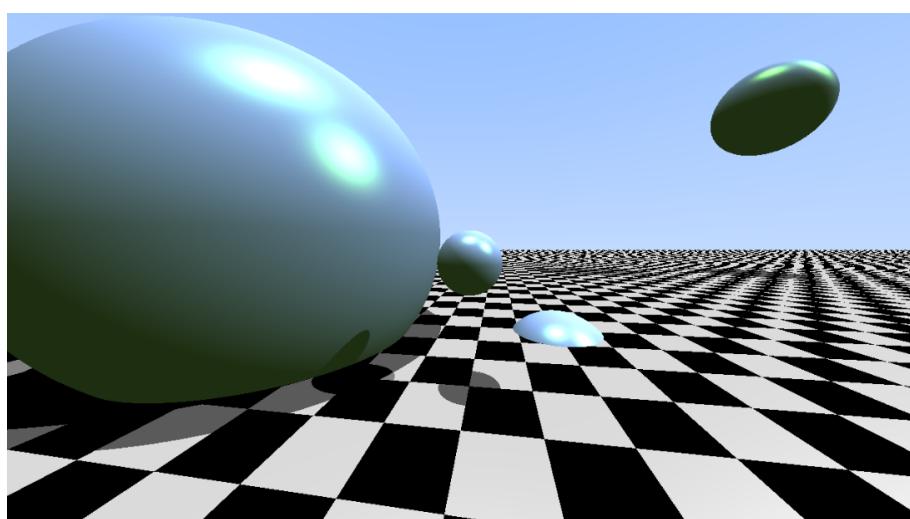


Imagen 8.21: Escena aplicando el modelo de Phong completo

8.8. SSAA en Ray-Tracing

En el capítulo 7, y concretamente en la sección 7.8 mostramos la posibilidad de utilizar más de un punto del plano por píxel, calcular el color asociado a cada uno y asignar un valor promedio a `gl_FragColor`. Esta modificación producía imágenes y bordes más suavizados, con un nivel más alto de realismo. En 3D, esta técnica es igualmente aplicable y necesaria de hecho, fíjese por ejemplo en la imagen 8.22, en la cual se aprecian las irregularidades en los bordes de las fichas negras y blancas y a distancias mayores la textura se distorsiona bastante.

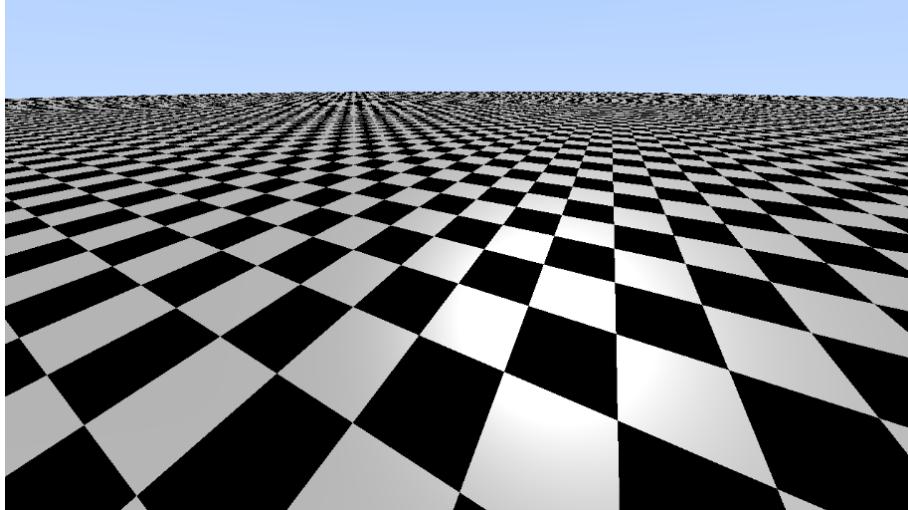


Imagen 8.22: Suelo antes de aplicar antialiasing

Por suerte, podemos aplicar SSAA también en 3D. De igual forma que calculamos n^2 puntos por píxel en la región del plano que representábamos en 2D, podemos calcular, por cada píxel, n^2 puntos del plano de proyección, lanzar un rayo en dirección de cada uno y promediar los colores calculados para cada rayo. La metodología es prácticamente análoga a la utilizada en la sección 7.8, teniendo en cuenta que en este caso calculamos puntos del plano de proyección para posteriormente crear rayos. De hecho, invitamos al lector que visite dicha sección si no entiende algunos de los razonamientos, los cuales hemos abreviado por su parecido a los ya explicados en la sección 7.8. Procedemos entonces a explicar la metodología.

En `main`, cuando obtenemos `u, v` y creamos el rayo, ahora debemos calcular n^2 coordenadas. Recordemos, de la sección 8.6, `u, v` son las coordenadas de dispositivo normalizadas en $[0, 1]$, es decir:

```
1 | vec2 uv = (gl_FragCoord.xy + vec2(0.5)) / vec2(nx, ny);  
2 | float u = uv.x;  
3 | float v = uv.y;
```

Si ahora en lugar de utilizar un rayo por píxel que se dirige al centro del mismo deseamos lanzar n^2 rayos uniformemente distribuidos por el la superficie del píxel, tenemos que dividir su ancho en n intervalos, análogamente el alto. Por tanto, dividimos cada píxel en una cuadrícula $n \times n$ donde cada fragmento mide $h_w := \frac{1}{n_x \cdot n}$ en anchura y de $h_h := \frac{1}{n_y \cdot n}$ en altura. Tras esto, centramos la cuadrícula sumando $\frac{1}{2}h_w$ en ancho y $\frac{1}{2}h_h$ en alto (revisitar imagen 7.8).

Seguidamente, calculamos los n^2 colores, uno por cada rayo que lancemos. Por cada rayo calculamos la coordenada de dispositivo normalizada del punto hacia el cual queremos lanzar

el rayo, llamamos a `get_ray`, a `ray_color` y sumamos el valor de retorno de esta última. Finalmente dividimos esta suma entre n^2 para calcular el color promedio y este es el que finalmente asignaríamos a `gl_FragColor`.

Para calcular la coordenada de dispositivo normalizada del punto i -ésimo iteramos un índice $i = 0, \dots, n^2$, de forma que i/n sería la fila e $i \% n$ la columna. Es decir, a la coordenada de dispositivo normalizada inicial habría que sumarle $(i/n)h_w + \frac{1}{2}h_w$ en anchura e $(i \% n)h_h + \frac{1}{2}h_h$ en altura para obtener la coordenada normalizada a la que lanzar el rayo. El código GLSL por tanto sería el siguiente

```

1 // Antialiasing
2 vec2 uv = (gl_FragCoord.xy) / vec2(nx, ny);
3 float u = uv.x;
4 float v = uv.y;
5
6 int nSamples = 3; // 0 cualquier otro numero natural
7 float hw = 1.0 / (float(nx * nSamples)),
8      hh = 1.0 / (float(ny * nSamples));
9 Ray r;
10 vec4 sum_colors = vec4(0.0, 0.0, 0.0, 1.0);
11
12 for(int i = 0; i < 1000; i++) {
13     if(i == nSamples*nSamples) break;
14     int x = i/nSamples;
15     int y = i - nSamples*x;
16     u = uv.x + float(x) * hw + 0.5 * hw;
17     v = uv.y + float(y) * hh + 0.5 * hh;
18     r = get_ray(cam, u, v);
19     sum_colors += ray_color(r, world, num_spheres, ground,
20                             lights, num_lights);
21 }
22
23 gl_FragColor = sum_colors / float(nSamples*nSamples);

```

Con esta modificación y fijando el valor que consideremos en n (`nSamples`) se consiguen resultados como el de la figura 8.23. Como se puede ver, los bordes son más suaves y de mejor calidad y las regiones más lejanas tienen mejor resolución.

De nuevo, por cuestiones de eficiencia e interactividad, se incluye en el software la posibilidad de decidir si se desea o no aplicar SSAA en 3D, y en caso afirmativo, cuántos rayos por píxel se desean trazar. Esto se consigue mediante las variables `uniform u_antialiasing` y `u_nSamples`, cuyas funciones son las mismas que las ya explicadas en la sección 7.8, aunque se puede deducir del código presentado a continuación.

```

1 uniform bool u_antialiasing;
2 uniform int u_nSamples;
3 // ...
4 void main() {
5     // ...
6     if(u_antialiasing) {
7         // Aplicar SSAA
8         int nSamples = u_nSamples;

```

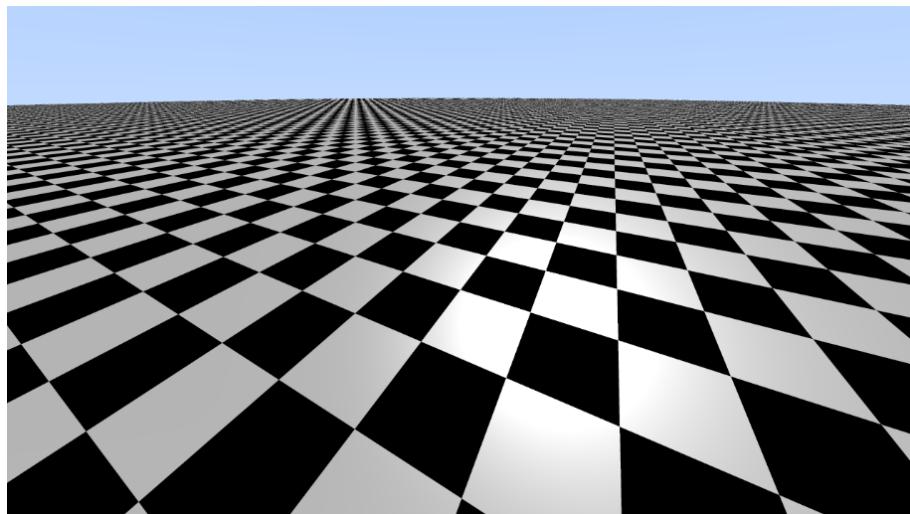


Imagen 8.23: Suelo tras aplicar antialiasing

```
9      // ...
10 }
11 else {
12     // No aplicar SSAA
13     // ...
14 }
15 }
```

Para terminar esta exposición y este capítulo mostraremos una última imagen que resume todo lo desarrollado: Ray-Tracing, intersecciones con distintos cuerpos, posicionamiento de cámara, iluminación, sombras y SSAA.

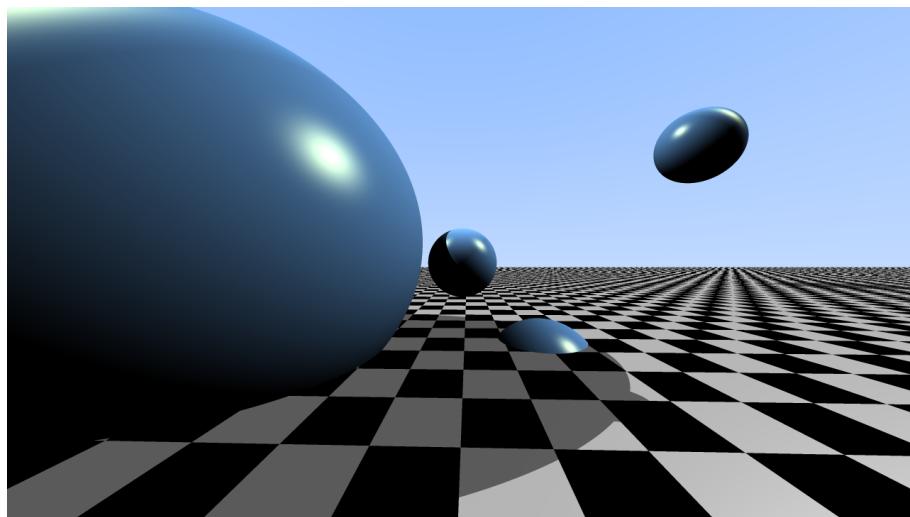


Imagen 8.24: Imagen final del capítulo 8

CAPÍTULO 9

VISUALIZACIÓN DE FRACTALES EN 3D

En el capítulo 7 introdujimos técnicas y código necesario para poder visualizar conjuntos de Julia y conjuntos de Mandelbrot 2-dimensionales en un canvas utilizando WebGL, todo ello apoyado en la teoría explicada en el capítulo 3. Sin embargo, el objetivo de éste y del capítulo anterior es aplicar técnicas avanzadas de generación de imágenes para poder visualizar fractales en tres dimensiones. Recordamos que explicamos en la introducción del capítulo 8 que la mejor forma de generar imágenes de fractales 3D es utilizando la técnica de ‘Ray Tracing’, que en dicho capítulo explicamos con detalle. A pesar de ello y de conseguir escenas con varios objetos, materiales y movimientos de cámara no visualizamos nada lejanamente parecido a un fractal. Será en este capítulo en el que aprovecharemos toda la infraestructura y todo el código implementado hasta el momento en el ‘ray-tracer’ del capítulo 8 para graficar fractales tridimensionales. El código del fragment shader utilizado en este capítulo se puede encontrar en el fichero `fragment-shader-3D-fractals.js`, que se puede consultar en <https://github.com/JAntonioVR/Geometria-Fractal/blob/main/static/glsl/fragment-shader-3D-fractals.js>.

Para ello, antes aún debemos asentar una serie de conceptos que son los que nos ayudarán a cumplir nuestro objetivo: el algoritmo *Sphere-Tracing* y las conocidas como *Signed Distance Functions* (SDFs). Con estos dos elementos combinados conseguiremos generar hermosas figuras fractales.

9.1. El algoritmo Sphere-Tracing

Hasta ahora siempre hemos calculado las intersecciones rayo-esfera de manera totalmente analítica, ya que es muy sencillo describir una esfera o un plano mediante una ecuación y solucionar esta ecuación en t , como vimos en las secciones 8.5.1 y 8.5.3. Sin embargo, no es tan sencillo encontrar ecuaciones que describan la superficie de los fractales. Es por esto que necesitamos otra manera de encontrar las intersecciones rayo-superficie. Recordemos entonces que en la sección 8.2 hablamos de que además de las expresiones analíticas las intersecciones se pueden calcular mediante métodos iterativos.

En literatura, se denomina *Ray-Marching* a cualquier método numérico iterativo utilizado para encontrar intersecciones mediante movimientos a lo largo de un rayo. Sobre este tema hay

muchísima bibliografía y es un tema de interés vigente. El método más sencillo es la búsqueda de raíces de una función mediante el método de Newton generalizado a \mathbb{R}^3 , pero existen técnicas más avanzadas.

El algoritmo *Sphere-Tracing* es una de estas técnicas de ray-marching descrita por *John C. Hart* en [15]. Se basa en estimar la distancia más corta de un punto en el rayo a cada una de las superficies que componen la escena, identificar la mínima de estas y avanzar en el rayo dicha distancia, pues se tiene la certeza de que al menos en una esfera de radio dicha distancia no hay ninguna intersección con ningún otro cuerpo. Una vez se ha avanzado en el rayo se repite esta operación y así sucesivamente hasta que la estimación de la distancia mínima es lo suficientemente pequeña como para considerar que el punto forma parte de una superficie.

Aclaramos que *Ray-Marching* y *Sphere-Tracing* no son sinónimos, aunque en ocasiones se abusa del lenguaje y se utilizan indistintamente. *Sphere-Tracing* es un tipo de técnica de Ray-Marching, siendo este último un concepto que engloba no solo a sphere-tracing, sino a toda técnica iterativa que busque intersecciones recorriendo el rayo.

Pongamos un ejemplo. Supongamos que tenemos una escena con una esfera tangente al plano horizontal $y = 0$, un rayo R fijo y queremos aplicar *Sphere-Tracing* para encontrar la intersección.

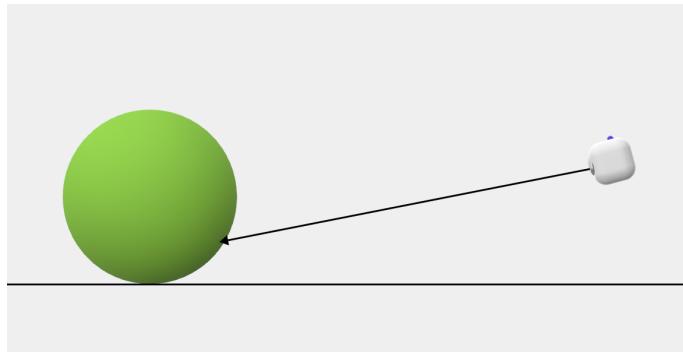


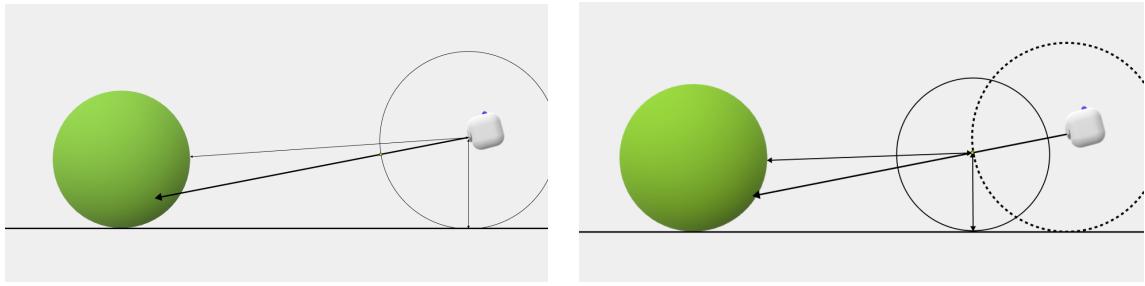
Imagen 9.1: Situación a la que aplicar *Sphere-Tracing*

La distancia de un punto p a una esfera S de centro c y radio r es

$$d(p, S) = \|p - c\| - r \quad (9.1)$$

y la distancia de un punto al plano $y = 0$ es simplemente su componente y . Por tanto, partiendo del origen del rayo, calculamos la distancia a la esfera y la distancia al plano, vemos que es menor la del plano, por lo que avanzamos en el rayo una distancia igual a la calculada para el plano (imagen 9.2 (a)). Seguidamente repetimos la operación: volvemos a calcular la distancia a la esfera y al plano. De nuevo el plano es el objeto más cercano, por lo que avanzamos en el rayo la misma distancia que había hasta el plano (imagen 9.2 (b)).

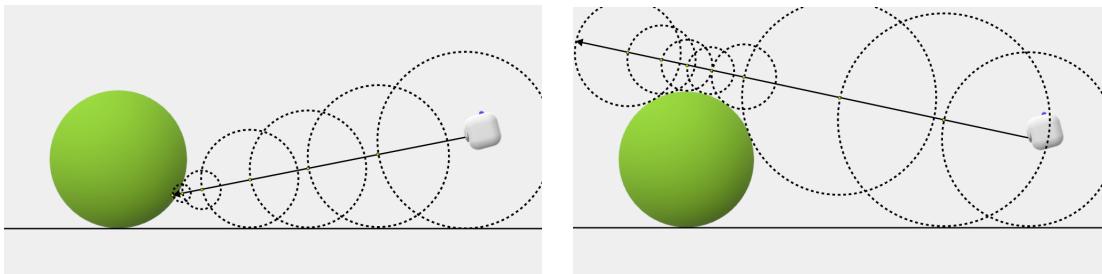
Si repetimos este proceso indefinidamente, llegará el momento en el que la distancia a la esfera será tan pequeña que consideraremos que el punto pertenece a la superficie de la misma y habremos encontrado la intersección (imagen 9.3 (a)). En caso de que no exista ninguna intersección, el algoritmo seguirá avanzando en el rayo, pero al no encontrar nunca una distancia suficientemente pequeña se avanzará indefinidamente (imagen 9.3 (b)), por lo que hay que fijar una distancia máxima recorrida y/o un número máximo de iteraciones como parámetro del algoritmo, estableciendo un criterio de parada.



(a) Primera iteración

(b) Segunda iteración

Imagen 9.2: Dos primeras iteraciones de Sphere-Tracing



(a) Encuentra intersección

(b) El rayo se pierde

Imagen 9.3: Posibles estados finales del algoritmo Sphere-Tracing

9.1.1. Signed Distance Functions (SDFs)

Como hemos visto, es indispensable para poder aplicar este algoritmo disponer para cada objeto que componga la escena de una función que estime la distancia de un punto cualquiera de \mathbb{R}^3 a su superficie. A grandes rasgos, son las conocidas como ‘Signed Distance Function’ (SDFs) las que para una superficie concreta calculan para cualquier punto la distancia estimada a dicha superficie.

Primero explicaremos brevemente el fundamento matemático. Sea $f : \mathbb{R}^n \rightarrow \mathbb{R}$ una función continua que define implícitamente el conjunto

$$A = \{x \in \mathbb{R}^n : f(x) \leq 0\}.$$

Por continuidad, $f(x) = 0 \quad \forall x \in \partial A$. Decimos que la frontera ∂A define la *superficie implícita* de f . De hecho, f es negativa en el interior de A ($f(x) < 0 \quad \forall x \in \mathring{A}$), por lo que podemos decir que dicha superficie implícita ∂A coincide precisamente con el conjunto $f^{-1}(0)$. A partir de una función real y continua podemos entonces definir una superficie en \mathbb{R}^n .

Definición 9.1.1 (SDF). Una función continua $f : \mathbb{R}^3 \rightarrow \mathbb{R}$ se conoce como una ‘**Signed Distance Bound**’ (cota de la distancia con signo) de su superficie implícita $f^{-1}(0)$ si, y solo si

$$|f(x)| \leq d(x, f^{-1}(0)) \quad \forall x \in \mathbb{R}^3 \tag{9.2}$$

Si se tiene la igualdad en (9.2), entonces f es una ‘**Signed Distance Function**’ (función distancia con signo).

El caso de las ‘signed distance bound’ también suele ser llamado ‘BDF’ (Bounding Distance Function).

Una forma sencilla de entender este concepto es concibiendo una superficie como los puntos en los que se anula la función distancia. Es decir, si un punto se sitúa a distancia 0 es porque pertenece a dicha superficie. En caso de que la distancia sea positiva en módulo el punto se sitúa fuera, tan lejos como diga dicho módulo.

Como ya hemos visto en la sección anterior, algunas primitivas, como las esferas, pueden ser fácilmente definidas por su SDF. Recordamos de la ecuación (9.1) que la SDF de una esfera es $f(x) = |x - c| - r$ siendo $c \in \mathbb{R}^3$ su centro y $r \in \mathbb{R}$ su radio. Entonces un punto exterior a la esfera tendrá un valor de la SDF positivo, uno interior tendrá un valor negativo y únicamente se anulará en el caso de que el punto pertenezca a la superficie de la esfera (ver imagen 9.4).

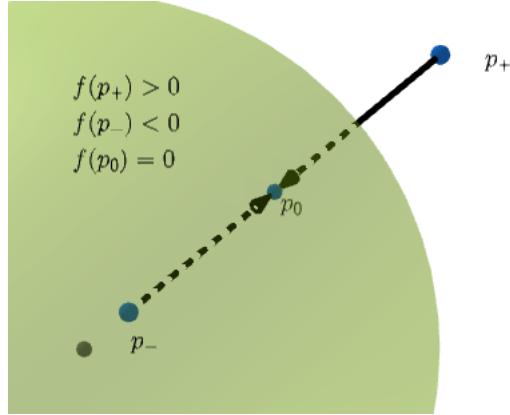


Imagen 9.4: Ejemplos de puntos con distintos valores de la SDF de una esfera

Otro ejemplo sencillo es el caso de la SDF de un plano. Sea entonces un plano arbitrario P definido por la ecuación $Ax + By + Cz = D$, siendo $\vec{N} = (A, B, C)$ el vector normal al plano. El punto p_0 perteneciente a un plano más cercano a otro punto p dado es aquel que interseca con la recta cuyo vector director es el normal al plano y que pasa por p . Es decir, el punto de la recta $S(t) = p + \vec{N}t$ que satisface la ecuación $(A, B, C) \cdot (x, y, z) = D$. Veámos para qué t se satisfacen las ecuaciones.

$$\begin{aligned} \vec{N} \cdot S(t) &= D \\ \vec{N} \cdot (p + \vec{N}t) &= D \\ t &= \frac{D - \vec{N} \cdot p}{|\vec{N}|^2} \end{aligned} \tag{9.3}$$

Por tanto la distancia entre punto y plano es la longitud del vector que une a p con p_0 , véase en la imagen 9.5.

En [15, Table 1] podemos encontrar una lista de primitivas con referencias a sus correspondientes SDFs.

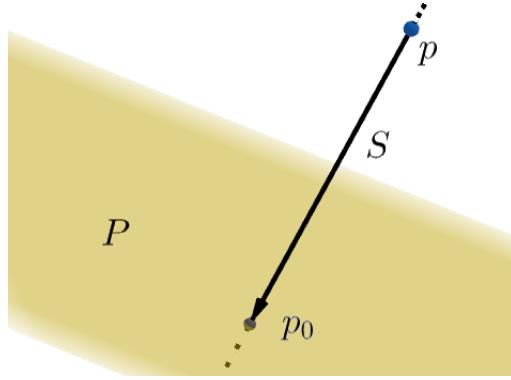


Imagen 9.5: Cálculo de la distancia de un punto a un plano

9.1.2. Implementación de Sphere-Tracing en pseudocódigo

Una vez conocemos el funcionamiento intuitivo del algoritmo y la necesidad de SDFs para cada objeto, podemos empezar a pensar en una primera implementación en pseudocódigo del algoritmo. Previamente, necesitamos fijar algunas constantes que sirven como parámetro al algoritmo. Estas constantes son:

- **MAX_STEPS:** Número máximo de iteraciones o avances en el rayo que se procesarán antes de decidir que el rayo no impacta con ninguna superficie. Es decir, alcanzado dicho número de iteraciones, el algoritmo concluye que el rayo se pierde en el infinito.
- **MAX_DIST:** Distancia máxima considerada. A distancia del origen del rayo mayor que ésta suponemos que no existe ningún objeto en la escena.
- ε : Mínima distancia tal que si en una iteración la distancia mínima calculada es menor que ε se considera que el punto evaluado pertenece a la superficie de un objeto. Claramente un valor pequeño de ε supone una resolución mayor a cambio de un mayor tiempo de cálculo.

Por tanto, el algoritmo, si consideramos un conjunto de *objetos* (cada uno identificado por un entero i , empezando en 0) y un rayo $R(t) = p_0 + \vec{v}t$, se describiría como indica el algoritmo 4.

Observación 9.1.1. Cabe destacar un importante detalle, y es que a partir de este momento es indispensable que en la creación del rayo el vector dirección esté normalizado, pues si queremos avanzar min_dist unidades en la dirección \vec{v} y $|\vec{v}| \neq 1$ se avanzaría una distancia distinta y el algoritmo no funcionaría correctamente, véase imagen 9.7.

Nótese cómo para el cálculo de la distancia utilizamos funciones distintas que dependen de cada objeto, de ahí el subíndice.

9.1.3. Implementación de Shere-Tracing en GLSL

Veámos ahora cómo podemos llevarnos todos estos nuevos conocimientos a la GPU con GLSL. El objetivo ahora es modificar el fragment shader eliminando la dependencia de `Hit_Sphere` y de `Hit_Plane` para utilizar Sphere-Tracing en el momento de calcular las intersecciones. Realmente el resultado de estas modificaciones debería ser el mismo que en el final del capítulo anterior, pero ahora encontraremos las intersecciones del rayo de forma distinta.

Algoritmo 4 Sphere-Tracing

```
procedure SPHERE-TRACING(objects[num_objects], R(t) = p0 +  $\vec{v}t$ )
    p ← p0
    steps ← 0
    while steps < MAX_STEPS do
        min_dist ← MAX_DIST
        i ← 0
        while i < num_objects do           ▷ Cálculo de la mínima distancia en cada iteración
            dist ← SDFobjects[i](p)
            if dist < min_dist then
                min_dist ← dist
                index ← i
            end if
            i ++
        end while
        if min_dist < ε then
            Hay intersección con objects[index]
        else
            p ← p +  $\vec{v} \cdot min\_dist$           ▷ Avanzamos en el rayo
        end if
        steps ++
    end while
    No existe intersección
end procedure
```

Lo primero que necesitaremos es una función que calcule la distancia de un punto a una esfera S y otra que calcule la distancia a un plano P . Respecto a la esfera, es tan sencillo como programar una función que implemente la fórmula (9.1).

```
1 // Dada una esfera S y un punto p, calcula la distancia
2 // del punto a la superficie de la esfera
3 float get_dist_sphere(vec3 p, Sphere S){
4     return length(S.center - p) - S.radius;
5 }
```

Por parte del plano, ya vimos en la sección 9.1.1 cómo calcular la distancia de un punto p a un plano dado por la ecuación $Ax + By + Cz = D$. Primero calculamos el valor de t dado por la ecuación 9.3 para así obtener el punto del plano más cercano a P , y a partir de ahí obtener la distancia. El código entonces sería:

```
1 // Dado un plano P y un punto p, calcula la distancia del
2 // punto p a la superficie del plano.
3 float get_dist_plane (vec3 p, Plane P) {
4     float t_interseccion = (P.D - dot(P.normal,p))/dot(P.normal,
5                             P.normal);
5     vec3 closest_point = p + t_interseccion * P.normal;
6     return length(p-closest_point);
```

7 | }

En realidad podríamos aprovechar que utilizamos como suelo el plano horizontal $y = -2$ y la SDF sería simplemente la componente y del punto p más dos, pero estaríamos limitando la función a este único plano.

Seguidamente necesitamos fijar dos constantes: el número máximo de iteraciones que se darán en el algoritmo (**MAX_STEPS**) y ε , la distancia mínima por debajo de la misma se considera que un punto pertenece a la superficie. La primera de ellas la podemos tomar como una macro, al igual que hicimos con el tamaño de los arrays, pero la segunda interesa que sea parametrizable, para así poder observar los niveles de detalle y qué variaciones experimenta la escena al modificar este valor. Por ello, usamos una variable **uniform** a la que pasaremos valor vía JavaScript, que llamaremos **u_epsilon**.

```
1 uniform float u_epsilon;
2 // ...
3 #define MAX_STEPS 1000
```

Y ya tenemos todo preparado para implementar el sphere-tracing. La función que dado un rayo calcula posibles intersecciones con los cuerpos y asigna un color es **ray_color**, por lo que lo más natural es editar el código de esta función. En lugar de llamar a **hit_sphere_list** y **hit_plane** como hicimos en la sección 8.5.3 aplicaremos sphere-tracing. Como GLSL no nos permite mantener un array heterogéneo en el que almacenar todos los objetos independientemente de su tipo, tenemos que hacerlo de forma secuencial. Y como tampoco permite acceder a elementos de un array a partir de índices no constantes, utilizaremos una variable donde almacenar la esfera más cercana, para recuperarla en caso de intersección, véase el código.

```
1 vec4 ray_color(Ray r, Sphere S[ARRAY_TAM], int num_spheres,
2     Plane ground,
3     Directional_light lights[ARRAY_TAM], int num_lights) {
4
5     Hit_record hr; hr.hit = false; // Estructura Hit_record
6     float dist = MAX_DIST; // Distancia a cada objeto
7     vec3 p = r.orig; // Punto del rayo
8     float closest_dist = MAX_DIST; // Menor distancia
9     float current_t = 0.0; // p=orig+current_t*dir
10    vec4 tmp_color; // Color temporal
11    Sphere S_hit; // Esfera mas cercana
12
13    int object_index; // 0,...,num_spheres-1: esferas,
14                      // num_spheres: plano
15    // Sphere Tracing
16    for(int i = 0; i < MAX_STEPS; i++) {
17        // Calculamos el objeto mas cercano
18        closest_dist = MAX_DIST;
19
20        // Distancia a las esferas
21        for(int i = 0; i < ARRAY_TAM; i++) {
22            if(i == num_spheres) break;
23            dist = get_dist_sphere(p, S[i]);
```

```

24         if(dist < closest_dist) {
25             closest_dist = dist;
26             object_index = i;
27             S_hit = S[i];
28         }
29     }
30
31     // Distancia al suelo
32     dist = get_dist_plane(p, ground);
33     if(dist < closest_dist) {
34         closest_dist = dist;
35         object_index = num_spheres;
36     }
37
38     // closest_dist almacena la menor de las distancias
39     // y object_index el indice del objeto mas cercano
40     if(closest_dist < u_epsilon){ // Hay interseccion
41         hr.hit = true;
42         hr.t = current_t;
43         hr.p = ray_at(r, hr.t);
44
45         if(object_index == num_spheres){ // Suelo
46             // Codigo de interseccion con el suelo ...
47         }
48         else { // Una de las esferas
49             // Codigo de interseccion con S_hit
50         }
51     }
52     // Si no hay interseccion, avanzamos en el rayo
53     current_t += closest_dist;
54     p = ray_at(r, current_t);
55     // Si estamos muy lejos acabamos el bucle
56     if(current_t >= MAX_DIST) break;
57 }
58 // No hay interseccion
59 // Codigo del background ...
60 }
```

Fíjese que concuerda con la estructura que describe el algoritmo 4.

Una modificación muy necesaria para que el código funcione es la de la función `light_is_visible`, que también utilizaba trazado de rayos para saber qué puntos se situaban a la sombra. La idea es la misma que la aplicada en `ray_color`: sustituir la llamada a `hit_sphere_list` por un bucle que implemente sphere-tracing hasta encontrar una intersección.

```

1 float light_is_visible(Directional_light light, vec3 p,
2     Sphere world[ARRAY_TAM], int num_spheres) {
3     Ray R;
4     R.dir = normalize(light.dir);
5     R.orig = p;
6     float t = 2.0*u_epsilon; // Comenzamos a una pequena distancia
```

```

7   float dist, h;
8   for(int i = 0; i < MAX_STEPS; i++ ) {
9     h = MAX_DIST;
10    for(int i = 0; i < ARRAY_TAM; i++) {
11      if(i == num_spheres) break;
12      dist = get_dist_sphere(ray_at(R,t), world[i]);
13      h = min(h, dist);
14    }
15    if(h < u_epsilon) // El rayo interseca una esfera, p esta a la
16      sombra
17      return 0.0;
18    t += h;
19    if(t >= MAX_DIST) break;
20  }
21  return 1.0; // No se ha encontrado interseccion, la luz incide
               sobre p
21 }

```

Hecha esta modificación, el resultado fijando ciertos parámetros dinámicamente es el mostrado en la imagen 9.6. Tal y como adelantamos, el resultado es muy similar al obtenido cuando calculábamos las intersecciones exactas, ya que realmente lo único que cambia es la forma de calcular las intersecciones, nada estrictamente visual.

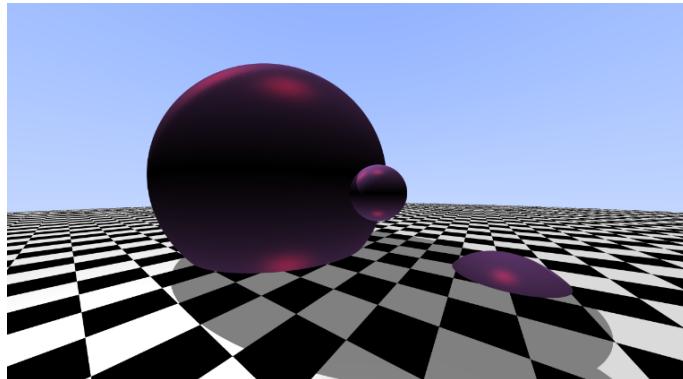


Imagen 9.6: Escena tras implementar Sphere Tracing

Recordamos e insistimos en que el vector director del rayo `r.dir` debe ser un vector normalizado, es decir, unitario. De otro modo los avances en el rayo no estarán controlados, obsérvese en la imagen 9.7 los resultados obtenidos al no normalizar el vector director al crear el rayo.

Aprovechando que el algoritmo sphere-tracing calcula en cada iteración la mínima distancia a objetos de la escena, podemos hacer una mejora en las sombras. Como se puede observar en las imágenes 8.19, 9.6 y en todas las que presentan sombras hasta ahora, los límites de las sombras están muy marcados, lo cual queda poco realista. La apariencia mejoraría si se suavizaran los bordes, asignándole valores intermedios entre 0 y 1 a puntos que, aunque no estén en la sombra de un objeto, estén cerca. El truco consiste en pensar en los rayos que, aunque no impacten con ningún objeto, pasen muy cerca de él. En estos casos pondríamos los puntos en cuestión en una especie de penumbra, de forma que cuanto más cerca pase del conjunto, más oscuro se representará, es decir, se devolverá un valor más cercano a 0. Por tanto, calcularemos un

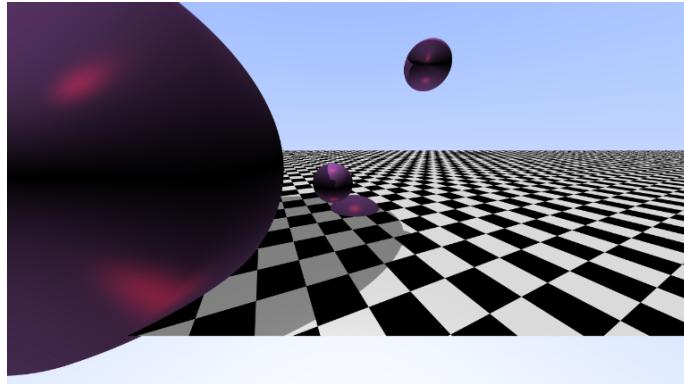


Imagen 9.7: Escena utilizando vectores no unitarios en los rayos

factor de penumbra en cada iteración de sphere-tracing y nos quedaremos con el menor de ellos. Tan solo tenemos que añadir algunas líneas de código a `light_is_visible`. Para una mayor comprensión del trasfondo de esta mejora, consúltese [40].

```

1 float light_is_visible(Directional_light light, vec3 p,
2   Sphere world[ARRAY_TAM], int num_spheres) {
3
4   Ray R;
5   R.dir = normalize(light.dir);
6   R.orig = p;
7   float k = 16.0; // Constante de suavizado de bordes
8   float res = 1.0; // Inicialmente 1
9   float t = 2.0*u_epsilon;
10  float dist;
11  float h = MAX_DIST;
12  for(int i = 0; i < MAX_STEPS; i++) {
13    h = MAX_DIST;
14    for(int i = 0; i < ARRAY_TAM; i++) {
15      if(i == num_spheres) break;
16      dist = get_dist_sphere(ray_at(R, t), world[i]);
17      if(dist < h)
18        h = dist;
19    }
20    if(h < u_epsilon)
21      return 0.0;
22    res = min(res, k*h/t); // Reducimos o no res
23    t += h;
24    if(t >= MAX_DIST) break;
25  }
26  return res;
27 }
```

La constante `k` es un factor que indica si el suavizado es más o menos notable. El valor 16 es suficientemente bueno para nuestro cometido, véase el resultado final en la imagen 9.8, pero mostramos en las imágenes 9.9 la representación con los valores $k = 2, 32$. Nótese como en la imagen 9.9 (a) el sombreado es muy difuminado y en la imagen 9.9 (b) es bastante más marcado.

La manera más óptima y correcta que hemos conseguido de iluminar la escena a partir

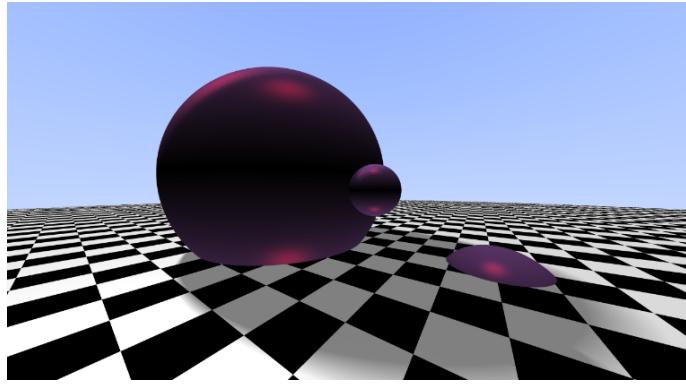


Imagen 9.8: Resultado final de la modificación

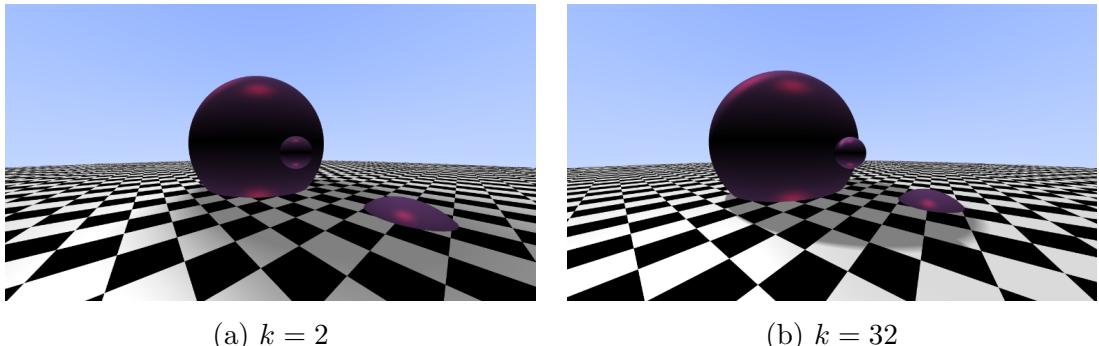


Imagen 9.9: Sombras con el parámetro $k = 2, 32$

de este momento, y con la mirada puesta en la visualización de fractales, es mediante dos luces direccionales cuya intensidad es parametrizable al igual que en capítulo anterior y cuyos vectores de dirección son $(-1, 1, 0)$ y $(1, 1, 0)$. Es decir, dos luces que iluminan ‘desde arriba y por ambos lados’. Sin embargo, esto provoca que la parte de abajo de los objetos siempre se vea oscura, por lo que hemos dotado a la escena de una tercera luz direccional blanca en la dirección $(0, -1, 0)$, con la particularidad de que esta no arroja sombras (siempre se le asigna `visibility=1.0`), simplemente aporta iluminación a las partes bajas.

9.1.4. Comentarios sobre Sphere-Tracing

Una vez hemos introducido e implementado el algoritmo Sphere-Tracing, el lector es posible que piense que es una forma menos exacta y además menos eficiente que calcular analíticamente las intersecciones con los cuerpos. Es cierto que en casos como el de esferas o planos, cuyas ecuaciones implícitas y sus intersecciones con rayos están muy bien definidos y son sencillas de calcular, aplicar sphere-tracing ralentiza el procesado. Sin embargo, muchas superficies, como es el caso de los fractales, no cuentan con una ecuación implícita que la defina. Otras sí cuentan con ecuaciones implícitas pero estas son muy difíciles de resolver. En este último caso se pueden aplicar métodos numéricos para aproximar las soluciones, que son métodos precisamente iterativos.

Sin embargo, sphere-tracing sólo necesita, para cada cuerpo, una función que aproxime suficientemente bien la distancia a un conjunto, o al menos una cota inferior, y esta cota disminuya conforme el rayo se acerca al conjunto. Sería suficiente una cota inferior porque esta nos aseguraría que al avanzar en el rayo no nos pasamos de largo. Lo único que debemos

garantizar es que la cota inferior debe tener exactamente los mismos ceros que la original (es decir, nunca es cero donde la original no lo es). De hecho, se puede usar esta metodología en casos donde incluso la función distancia exacta sea muy difícil o imposible de calcular.

En concreto, podemos usar cualquier función f que tenga exactamente los mismos ceros que la función distancia exacta y que sea lipschitziana, es decir, que existe una constante positiva K tal que

$$|f(p) - f(q)| \leq K\|p - q\| \quad \forall p, q \in \mathbb{R}^3$$

de tal forma que si llamamos p_0 al punto de la superficie más cercano a p , por hipótesis $f(p_0) = 0$, por lo que

$$|f(p)| \leq K\|p - p_0\|$$

Si tomamos la función $g = f/K$, directamente se tendría

$$|g(p)| \leq \|p - p_0\|$$

Convirtiendo a g en una cota inferior de la distancia a p_0 (una BDF), por lo que podemos utilizar a g para aplicar sphere tracing.

Sin embargo, si tomamos una cota inferior en lugar de la exacta, al dar pasos más pequeños, se tiene menos eficiencia. Dicho esto, queda claro que la función distancia exacta de un punto a la superficie es la mejor opción, pues es lipschitziana por definición y su constante es de Lipschtiz es 1, pero por dificultad en su cálculo se suelen usar cotas inferiores.

Por tanto, a partir de un rayo y dichas funciones se puede aplicar sphere-tracing, de forma que se converge finalmente al punto de intersección. Por contra, si buscamos intersecciones analíticas hay que resolver ecuaciones, en ocasiones para acabar aproximando las soluciones, y retener la más pequeña.

Un aspecto a destacar de sphere-tracing es la dependencia de sus parámetros. Es importante fijar un valor ε adecuado, ya que si es demasiado grande puede brindarnos resultados muy pobres. En la imagen 9.6 se ha utilizado $\varepsilon = 0.001$ y nos da un resultado muy bueno, pero obsérvese en la imagen 9.10 qué ocurre si se fija $\varepsilon = 0.1$. Los límites de los cuadrados del suelo se distorsionan y las apariencias de las esferas también se ven muy resentidas. No obstante, es cierto que a mayor valor de ε menos iteraciones y por tanto más velocidad, pero los resultados son peores. Es por tanto necesario utilizar valores de ε adecuados a la situación, siendo suficientemente pequeños como para obtener un buen resultado pero suficientemente grandes como para que sea computacionalmente viable.

Una reflexión similar se puede aplicar al parámetro que define el número máximo de iteraciones (**MAX_STEPS**). Pensemos en un rayo que pasa muy cerca de un objeto pero no llega a intersecarlo, sino que finalmente interseca al suelo. Durante las iteraciones que el rayo pase cerca del objeto quizás avance muy lentamente durante varias iteraciones, pues la distancia mínima es la distancia a dicho objeto que es pequeña. Este hecho puede provocar que se alcance el número máximo de iteraciones, considerando así que el rayo no golpea ningún objeto y asignándole el color de fondo cuando realmente con algunas iteraciones más podría alcanzarse la intersección con el suelo. Este hecho se puede observar en la imagen 9.11, donde se ha utilizado **MAX_STEPS=100**. Fíjese en las fronteras entre el horizonte y las esferas, o en los bordes de las esferas.

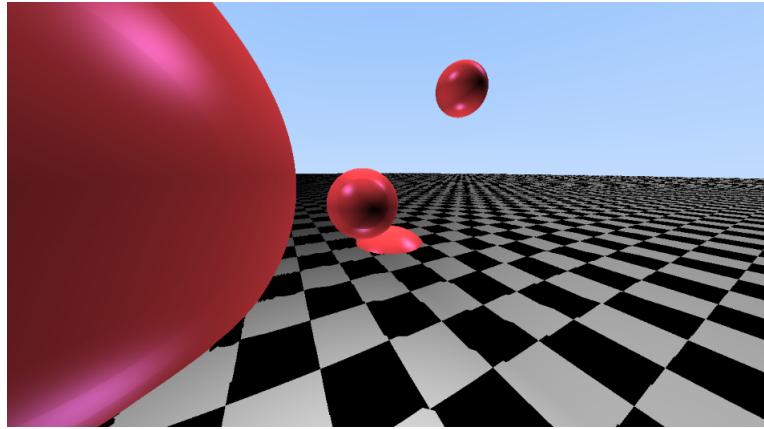


Imagen 9.10: Escena generada utilizando $\varepsilon = 0.1$

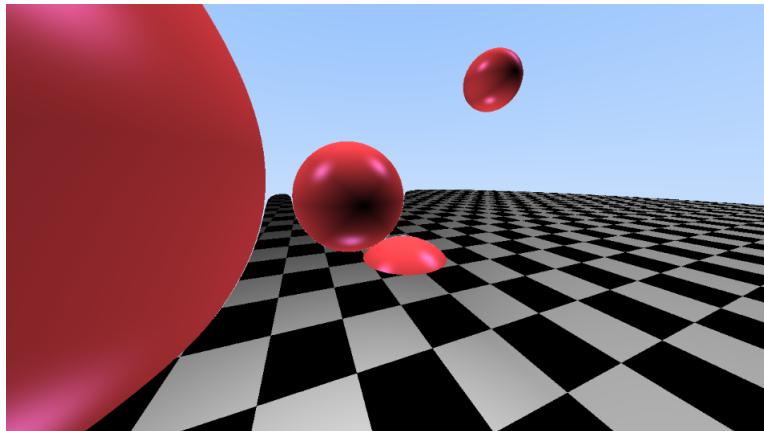


Imagen 9.11: Escena generada utilizando **MAX_STEPS=100**

De igual forma, menos iteraciones implica más rapidez, pero peores resultados, por lo que es necesario fijar un valor correcto.

9.2. Visualización tridimensional de conjuntos de Julia

Una vez tenemos definido el algoritmo que usaremos para la visualización de fractales, que es el algoritmo de sphere-tracing acompañado de una SDF concreta, es momento de trabajar en la visualización de fractales 3D. Comenzaremos, siguiendo el mismo orden que adoptamos en el capítulo 7, por los conjuntos de Julia en 3D.

Recordemos de la sección 3.2 que el conjunto de Julia \mathcal{J}_c de un número complejo fijo $c \in \mathbb{C}$ se define como la frontera entre el conjunto de puntos prisioneros y el conjunto de puntos de escape bajo la iteración de la función $P_c(z) = z^2 + c$. Es decir, $\mathcal{J}_c = \partial E_c$, donde

$$E_c = \{z_0 \in \mathbb{C} : \{|P_c^n(z_0)|\} \rightarrow \infty\} \quad (9.4)$$

Queda por tanto a la vista una clara dependencia de los números complejos en esta definición, los cuales podemos identificar únicamente con puntos del plano euclídeo \mathbb{R}^2 . Necesitamos por tanto una generalización tridimensional de los números complejos. Estrictamente, no existe como tal una generalización tridimensional que mantenga la aritmética compleja, pero sí que existe una en cuatro dimensiones: los cuaternios, usualmente denotados como \mathbb{H} .

Podemos encontrar información sobre los cuaternios en [5], pero introduciremos brevemente su aritmética. De igual forma que para los números complejos se utiliza la unidad imaginaria i , en los cuaternios se tienen tres unidades imaginarias: i, j, k , de tal manera que un cuaternion se compone de una parte real y tres imaginarias, pudiendo expresarlo como:

$$q = ai + bj + ck + d \cong (a, b, c, d) \in \mathbb{R}^4. \quad (9.5)$$

Nótese que estamos denotando como componente real a d , la última de la terna (a, b, c, d) . Las unidades imaginarias se multiplican entre ellas siguiendo las siguientes reglas:

$$\begin{aligned} ij &= k & jk &= i & ki &= j \\ ji &= -k & kj &= -i & ik &= -j \\ i^2 &= j^2 = k^2 = -1. \end{aligned} \quad (9.6)$$

Lo cual nos da a entender que los cuaternios tienen estructura de anillo no commutativo. Una forma de calcular el producto de dos cuaternios arbitrarios $q = ai + bj + ck + d \cong (a, b, c, d)$ y $q' = a'i + b'j + c'k + d' \cong (a', b', c', d')$ es expandiendo la expresión

$$qq' = (ai + bj + ck + d)(a'i + b'j + c'k + d') \quad (9.7)$$

Si utilizamos su expresión como ternas de \mathbb{R}^4 , denotando $q = (q_{abc}, q_d)$, $q' = (q'_{abc}, q'_d)$, agrupando términos y aplicando las relaciones 9.6 el resultado puede expresarse mediante productos vectoriales y escalares:

$$\begin{aligned} (qq')_{abc} &= q_{abc} \times q'_{abc} + q_d q'_{abc} + q'_d q_{abc} \\ (qq)_d &= q_d q'_d - (q_{abc} \cdot q'_{abc}) \end{aligned} \quad (9.8)$$

Aplicando estas relaciones podemos deducir el cuadrado de un cuaternion con tan solo multiplicarlo por sí mismo, obteniendo

$$\begin{aligned} (q^2)_{abc} &= 2q_d q_{abc} \\ (q^2)_d &= q_d^2 - q_{abc} \cdot q_{abc}, \end{aligned} \quad (9.9)$$

que realmente es una simplificación de las ecuaciones (9.8) teniendo en cuenta que el producto vectorial de un vector por sí mismo es 0.

Y una vez tenemos clara la aritmética básica de los cuaternios, podemos generalizar fácilmente los conjuntos de Julia sin más que extender la función $P_c(z)$ a \mathbb{H} , fijando un cuaternion $c \in \mathbb{H}$:

$$\begin{aligned} P_c : \mathbb{H} &\longrightarrow \mathbb{H} \\ q &\longmapsto q^2 + c \end{aligned} \quad (9.10)$$

De forma que ahora separamos el espacio \mathbb{H} en puntos prisioneros y de escape frente a la iteración de $P_c(q)$ y redefinimos los conjuntos de Julia en 4D como la frontera entre los puntos prisioneros y de escape.

Sin embargo, es claro que no podemos visualizar conjuntos en 4D de forma tan directa como lo hacemos en 2D, pero sí podemos generarlos y visualizar una proyección en 3D. Pensemos en los mapas de curvas de nivel que se usan en topografía (imagen 9.12), los cuales fijando una altura

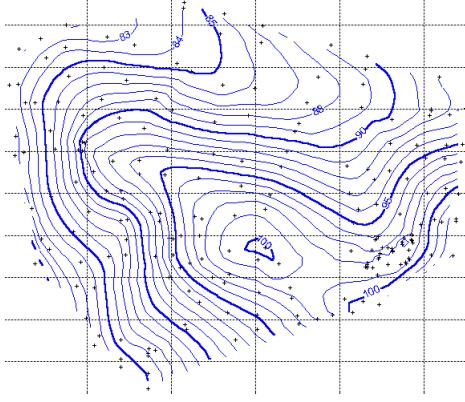


Imagen 9.12: Representación de una montaña con curvas de nivel

concreta representan la curva que dibujaría una montaña al intersecarla con un plano horizontal situado a esa altura, permitiéndonos hacernos una idea de la forma en 3D de la montaña a partir de un plano en 2D.

En nuestro contexto, podemos fijar una dimensión y visualizar el conjunto de Julia para las otras tres.

Es por tanto momento de estimar la distancia de un punto cualquiera $p \in \mathbb{R}^3$ a un conjunto de Julia \mathcal{J}_c , con $c \in \mathbb{H}$. Lo primero es indentificar un punto del espacio euclídeo $(x, y, z) \in \mathbb{R}^3$ con un cuaternio. La forma que escogeremos será identificando x con la parte real y la segunda y tercera componente con las dos primeras componentes imaginarias, fijando a 0 la tercera componente imaginaria (k). Es decir,

$$p = (x, y, z) \cong q = x + yi + zj \cong (y, z, 0, x) \quad (9.11)$$

Es posible que parezca antinatural identificar (x, y, z) con $(y, z, 0, x)$, pero si utilizamos la relación $(x, y, z) \cong xi + yj + zk$ anulando la parte real, los conjuntos de Julia que se calculen serían todos esferas concéntricas, lo cual carece de interés [14, Sección 2.3].

Tengamos en cuenta que en principio los rayos van a partir de un punto que está fuera del conjunto de Julia, es decir, al iterar la función P_c la sucesión va a ser divergente, por lo que la SDF nos debe de dar una distancia positiva. La idea es que cada vez nos dé un valor menor conforme nos acerquemos a \mathcal{J}_c hasta prácticamente anularse en caso de que el rayo interseque con el conjunto.

Debemos por tanto antes de nada iterar q para hacernos a la idea de esta distancia al conjunto, de forma que si tras cierto número M de iteraciones el módulo de la iterada $|P_c^M(q)|$ no es suficientemente grande consideraremos que el punto es convergente, aunque esto probablemente ocurra únicamente si se evalúa en un punto muy cercano a la superficie del conjunto. Es decir, iteramos el punto con una amplia posibilidad de que antes o después la n -ésima iterada tenga un módulo grande, y en caso contrario la iterada convergerá a algún valor tras las M iteraciones. Denotamos como $q_n = P_c^n(q)$ a esta n -ésima iterada. A partir de este cuaternio, definimos la SDF del conjunto de Julia como

$$d_c : \mathbb{R}^3 \longrightarrow \mathbb{R}$$

$$d_c(p) = \frac{|q_n| \log |q_n|}{2|q'_n|} \quad (9.12)$$

donde q'_n es otra sucesión iterativa que se puede calcular conjuntamente con q_n mediante $q'_{i+1} = 2q_i q'_i$, comenzando con $q'_0 = 1 \cong (0, 0, 0, 1)$, aunque como realmente sólo necesitamos su módulo podemos calcular $|q'_{i+1}| = 2|q_i||q'_i|^1$. Esta sucesión q'_n es la derivada de q_i respecto de q_0 , calculada iterativamente según la regla de la cadena, es decir

$$\begin{aligned} q'_{i+1} &= \frac{\partial P_c^{i+1}(q_0)}{\partial q_0} \\ &= \frac{\partial}{\partial q_0} ((P_c^i(q_0))^2 + c) \\ &= 2P_c^i(q_0) \frac{\partial P_c^i(q_0)}{\partial q_0} \\ &= 2q_i q'_i \end{aligned} \tag{9.13}$$

La función 9.12 es una aproximación de la distancia real, concretamente una cota inferior, la cual se basa en el potencial de Hubbard-Douady, tal y como se describe en [7, Capítulo 8]. Para ello se usa la posibilidad de usar BDFs que se mencionó en el apartado 9.1.4.

Sabemos hasta ahora que iterando el punto inicial y aplicando la transformación (9.12) podemos estimar a qué distancia se sitúa el punto del conjunto de Julia \mathcal{J}_c .

9.2.1. Aproximando la normal

Teniendo esta estimación de la distancia podemos calcular el posible punto de intersección de un rayo con \mathcal{J}_c , pero si queremos evaluar el modelo de iluminación en dicho punto, antes necesitamos conocer la normal a la superficie en dicho punto. Y aquí es donde tenemos que afrontar la dificultad que supone la propia belleza de los fractales, que es su irregularidad y que su superficie en general no es diferenciable. Por ello, no existe un plano tangente a la superficie ni un vector normal a la misma en ninguno de sus puntos. Sin embargo, nosotros estamos aproximando esa superficie ideal con otra distinta, muy similar, pero no igual, y esa otra superficie aproximada sí es diferenciable, por lo que podemos utilizar ciertos métodos para aproximar esta normal.

Método 1: Gradiente de la SDF

La manera más sencilla de obtener un vector normal a la superficie es tomando el gradiente de la SDF, el cual evidentemente no calculamos analíticamente sino numéricamente utilizando diferencias centradas. Para ello, fijado un punto $p \in \mathbb{R}^3$, calculamos 6 puntos muy próximos a él, sumando y restando en cada una de sus tres componentes un pequeño incremento, que por simplicidad consideraremos que será el propio ε que fijamos al hacer sphere-tracing. En cada uno de los 6 puntos evaluamos la SDF y calculamos para cada componente la diferencia, tomando como normal el vector formado por cada una de estas diferencias divididas normalizado. Puede consultarse un pseudocódigo de este método en el algoritmo 5.

En realidad debería tomarse en cada componente del gradiente la diferencia dividida entre $2h$, pero la propia normalización del vector se encarga de reescalar el vector adecuadamente.

¹El módulo de un cuaternio es $|xi + yj + zk + w| = \sqrt{x^2 + y^2 + z^2 + w^2}$

Algoritmo 5 Cálculo de la normal mediante el gradiente de la SDF

```
procedure CALCULARNORMALGRADIENTE( $p$ : punto de  $\mathbb{R}^3$ ,  $d_c$ : SDF del objeto)
     $h \leftarrow \varepsilon$ 
     $g_{x+} \leftarrow p + (h, 0, 0)$ 
     $g_{x-} \leftarrow p - (h, 0, 0)$ 
     $g_{y+} \leftarrow p + (0, h, 0)$ 
     $g_{y-} \leftarrow p - (0, h, 0)$ 
     $g_{z+} \leftarrow p + (0, 0, h)$ 
     $g_{z-} \leftarrow p - (0, 0, h)$ 
     $\nabla d_{c_x} \leftarrow d_c(g_{x+}) - d_c(g_{x-})$ 
     $\nabla d_{c_y} \leftarrow d_c(g_{y+}) - d_c(g_{y-})$ 
     $\nabla d_{c_z} \leftarrow d_c(g_{z+}) - d_c(g_{z-})$ 
    return  $\frac{(\nabla d_{c_x}, \nabla d_{c_y}, \nabla d_{c_z})}{\|(\nabla d_{c_x}, \nabla d_{c_y}, \nabla d_{c_z})\|}$ 
end procedure
```

Método 2: Técnica del tetraedro

Consideramos un tetraedro cuyos vértices son $k_0 = (1, -1, -1)$, $k_1 = (-1, -1, 1)$, $k_2 = (-1, 1, -1)$ y $k_3 = (1, 1, 1)$. Sean ahora los puntos $p_i := p + \varepsilon k_i$ $i = 0, \dots, 3$. Entonces una aproximación de la normal a la superficie en el punto p es la normalización del vector $\sum_{i=0}^3 k_i d_c(p_i)$. Puede verse un pseudocódigo de este método en el algoritmo 6. Consultese [42] si se desea tener una justificación del funcionamiento de este método.

Algoritmo 6 Técnica del tetraedro para el cálculo de normales

```
procedure CALCULARNORMALTETRAEDRO( $p$ : punto de  $\mathbb{R}^3$ ,  $d_c$ : SDF del objeto)
     $h \leftarrow \varepsilon$ 
     $p_0 \leftarrow p + h(1, -1, -1)$ 
     $p_1 \leftarrow p + h(-1, -1, 1)$ 
     $p_2 \leftarrow p + h(-1, 1, -1)$ 
     $p_3 \leftarrow p + h(1, 1, 1)$ 
     $N \leftarrow \sum_{i=0}^3 k_i d_c(p_i)$ 
    return  $\frac{N}{\|N\|}$ 
end procedure
```

9.2.2. Implementación en GLSL

Tenemos por tanto ya la forma de calcular tanto la distancia de un punto al conjunto de Julia de un $c \in \mathbb{H}$ fijo como un método para calcular la normal a la superficie del fractal en un punto, dos de hecho. Es por tanto momento de la implementación. Basta con programar funciones que calculen la SDF y sustituir las llamadas a `get_dist_sphere` por una llamada a dicha función, y una vez se conoce el punto de intersección calcular la normal y evaluar el modelo de iluminación.

De manera natural, igual que identificamos un cuaternionio con un elemento de \mathbb{R}^4 utilizaremos el tipo `vec4` para representar a los mismos en GLSL. Recordemos que en este caso es la última componente la parte real del cuaternionio.

```

1 | vec4 q; // Representa un cuaternionio (a,b,c,d)
2 | // q = ai + bj + cz + d

```

Si ahora seguimos la ecuación (9.9) podemos rápidamente implementar una función que dado un cuaternionio calcule su cuadrado.

```

1 | // Dado un cuaternionio q, calcula su cuadrado q^2
2 | vec4 quat_square(vec4 q){
3 |     vec3 abc = 2.0*q.w*q.xyz;
4 |     float d = q.w*q.w - dot(q.xyz, q.xyz);
5 |     return vec4(abc, d);
6 |

```

Por lo que gracias a esta función y a la aritmética ya programada es sencillo programar la función $P_c(q) = q^2 + c$. Utilizamos una función para iterar el cuaternionio asociado a un punto p , aprovechando la posibilidad que nos ofrece GLSL de simular el paso de parámetros por referencia mediante las variables `out` e `inout`. En el último caso, una función recibe un parámetro y en caso de que su valor cambie durante la ejecución éste mantiene su valor tras el retorno de la función. El código es

```

1 | // Itera la funcion P_c y obtiene tanto q_n como |q'_n|
2 | // q: Cuaternionio que se itera
3 | // dq: Sucesion |q'_n|
4 | // c: Constante fija de P_c asociada al conjunto de Julia
5 void iterate_julia(inout vec4 q, inout float dq, vec4 c) {
6     for(int i = 0; i < 50; i++) { // 50 son suficientes
7         dq = 2.0 * length(q) * dq;
8         q = quat_square(q) + c; // P_c(q) = q^2 + c
9         if(dot(q, q) > 4.0) break; // |q| es suficientemente grande
10    }
11 }

```

Fíjese que esta función itera un cuaternionio mediante la función $P_c(q)$ y a la vez calcula la sucesión recurrente $|q'_{i+1}| = 2|q_i||q'_i|$.

Por tanto, el cálculo de la distancia de un punto p al conjunto \mathcal{J}_c se reduce a identificar $p \in \mathbb{R}^3$ con un cuaternionio $q \in \mathbb{H}$, iterar q y $|q'|$ teniendo en cuenta que la iteración de q' comienza en 1 y aplicar la función (9.12).

```

1 | float get_dist_julia(vec3 p, vec4 c) {
2 |     vec4 q = vec4(p.y, p.z, 0.0, p.x);
3 |     float dq = 1.0; // q'_0 = 1
4 |     iterate_julia(q, dq, c);
5 |     float length_q = length(q);
6 |     return 0.5*length_q * log(length_q) / dq;
7 |

```

Y ya tenemos implementada la SDF de Julia. Únicamente resta un método para calcular normales. Como hemos presentado dos, implementaremos los dos, utilizando ahora compilación condicional mediante macros para decidir cuál emplear. Tan solo tenemos que implementar los algoritmos 5 y 6.

```

1 | #define NORMAL 0 // 0 1

```

```

2 // Calcula la normal al conjunto J_c en el punto p
3 vec3 calculate_normal_julia(vec3 p, vec4 c) {
4     vec3 N;
5     float h = u_epsilon;
6
7     #if NORMAL == 0
8         // Gradiente de la SDF
9         vec4 qp = vec4(p.y, p.z, 0.0, p.x);
10        float gradX, gradY, gradZ;
11        vec3 gx1 = (qp - vec4(0.0, 0.0, 0.0, h)).wxy;
12        vec3 gx2 = (qp + vec4(0.0, 0.0, 0.0, h)).wxy;
13        vec3 gy1 = (qp - vec4(h, 0.0, 0.0, 0.0)).wxy;
14        vec3 gy2 = (qp + vec4(h, 0.0, 0.0, 0.0)).wxy;
15        vec3 gz1 = (qp - vec4(0.0, h, 0.0, 0.0)).wxy;
16        vec3 gz2 = (qp + vec4(0.0, h, 0.0, 0.0)).wxy;
17
18        gradX = (get_dist_julia(gx2,c) - get_dist_julia(gx1,c))/(2.0*h);
19        gradY = (get_dist_julia(gy2,c) - get_dist_julia(gy1,c))/(2.0*h);
20        gradZ = (get_dist_julia(gz2,c) - get_dist_julia(gz1,c))/(2.0*h);
21        N = normalize(vec3(gradX, gradY, gradZ));
22
23    #else
24        // Metodo del tetraedro
25        const vec2 k = vec2(1,-1);
26        N = normalize( k.xyy*get_dist_julia( p + k.xyy*h, c ) +
27                        k.yyx*get_dist_julia( p + k.yyx*h, c ) +
28                        k.yxy*get_dist_julia( p + k.yxy*h, c ) +
29                        k.hxx*get_dist_julia( p + k.hxx*h, c ) );
30
31    #endif
32    return N;
33 }

```

Y ya únicamente resta sustituir en sphere-tracing las SDFs y el método para calcular normales de las esferas por el código que corresponde a los conjuntos de Julia, tanto en `ray_color` como en `light_is_visible`. Para poder dinamizar el cuaternio $c \in \mathbb{H}$ que se considera fijo en los cálculos declaramos una variable `uniform` al igual que se hizo a la hora de los conjuntos de Julia 2D. También creamos una variable global que es `Material fractal_material`; que como su propio nombre indica es el material que se le asignará al objeto fractal y para el cual utilizaremos las variables `uniform` que definimos en el capítulo anterior para el material parametrizable, permitiéndonos modificar la apariencia y el color del fractal que visualicemos.

```

1 // Cuaterniono c fijo. Visualizaremos el conjunto J_c
2 uniform vec4 u_juliaSetConstant;
3 // Material que se asigna al fractal
4 Material fractal_material
5 // ...
6
7 vec4 ray_color(Ray r, Plane ground,
8     Directional_light lights[ARRAY_TAM], int num_lights) {

```

```

9
10 // ...
11 int object_index; // 0: Suelo, 1: Julia
12
13 // Sphere Tracing
14 for(int i = 0; i < MAX_STEPS; i++) {
15     closest_dist = MAX_DIST;
16     // Distancia al plano
17     dist = get_dist_plane(p, ground);
18     if(dist < closest_dist) {
19         closest_dist = dist;
20         object_index = 0;
21     }
22     // Distancia a Julia
23     dist = get_dist_julia(p, u_juliaSetConstant);
24     if(dist < closest_dist){
25         closest_dist = dist;
26         object_index = 1;
27     }
28
29     if(closest_dist < u_epsilon){ // Hay intersección
30         hr.hit = true;
31         hr.t = current_t;
32         hr.p = ray_at(r, hr.t);
33         if(object_index == 0){ // r interseca el suelo
34             // ...
35         } else { // r hits Julia
36             hr.mat = fractal_material;
37             hr.normal = calculate_normal_julia(hr.p, u_juliaSetConstant)
38                 ;
39             return evaluate_lighting_model(lights, num_lights, hr);
40         }
41         current_t += closest_dist;
42         p = ray_at(r, current_t);
43         if(current_t >= MAX_DIST) break;
44     }
45     // r no interseca ninguna superficie
46     // ...
47 }

```

Y con esta modificación de `ray_color` y la análoga en `light_is_visible` podemos ya visualizar conjuntos de Julia tridimensionales. Podemos también cambiar los parámetros del material para observar distintas apariencias.

En las imágenes 9.13 podemos ver algunos de los resultados que, después de este largo camino, hemos podido obtener modificando los parámetros.

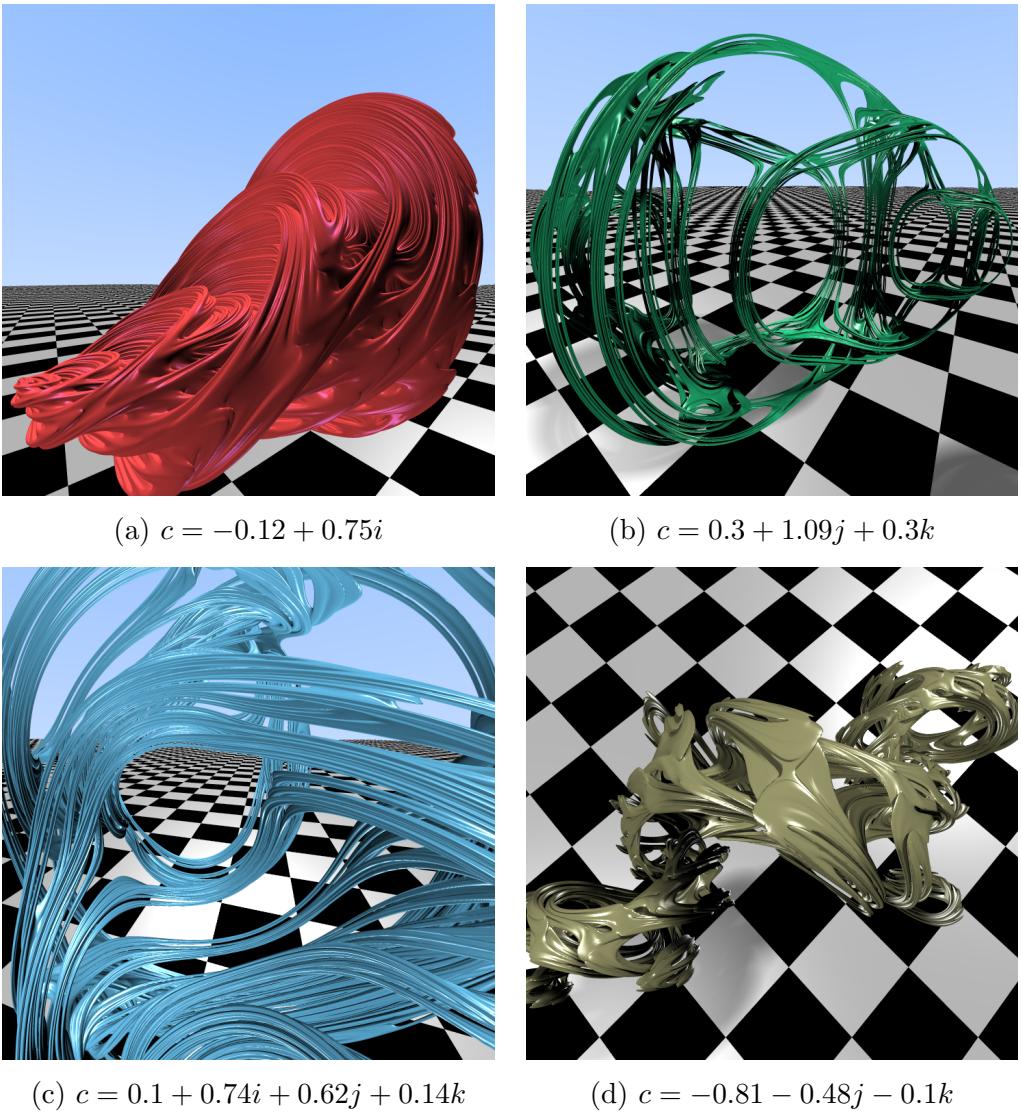


Imagen 9.13: Conjuntos de Julia 3D para distintos $c \in \mathbb{H}$

9.3. Visualización tridimensional del conjunto de Mandelbrot

La pregunta más natural tras visualizar conjuntos de Julia es si podemos aprovechar todo el código que tenemos para, bajo ligeras modificaciones, visualizar lo que sería el conjunto de Mandelbrot en 3D. Afortunadamente, la respuesta es sí. Recordemos ahora la sección 3.4, en la cual definimos, en \mathbb{C} , al conjunto de Mandelbrot \mathcal{M} de varias formas equivalentes, entre las que destacamos

$$\mathcal{M} = \{c \in \mathbb{C} : \{P_c^n(0)\} \not\nearrow \infty\}$$

Por tanto, la manera de iterar q y q' cambia levemente. En lugar de utilizar un cuaternio q_0 como semilla e iterar $q^2 + c$ con un $c \in \mathbb{H}$ fijo, utilizaremos a 0 como semilla e iteraremos $q^2 + c$ con $c = q_0$. La iteración de $|q'|$ también cambia, ya que ahora calculamos la sucesión

$q_{i+1} = q_i^2 + q_0$, por lo que ahora la sucesión sería $|q'_{i+1}| = 2|q_i||q'_i| + 1$, con $q'_0 = 0$, ya que

$$\begin{aligned}
 q'_{i+1} &= \frac{\partial P_c^{i+1}(0)}{\partial q_0} \\
 &= \frac{\partial}{\partial q_0} (q_i^2 + q_0) \\
 &= 2q_i \frac{\partial q_i}{\partial q_0} + 1 \\
 &= 2q_i q'_i + 1
 \end{aligned} \tag{9.14}$$

Puede consultarse una explicación más detallada de esta diferencia en [43]. Finalmente, tras suficientes iteraciones, se obtienen los valores q_n y q'_n necesarios en la SDF.

```

1 // Itera la función P_c y obtiene tanto q_n como |q'_n|
2 // q: Cuaternion que se itera
3 // dq: Sucesión |q'_n|
4 void iterate_mandelbrot(inout vec4 q, inout float dq) {
5     vec4 c = q;          // Constante inicialmente igual a q
6     q = vec4(0.0);      // Semilla inicial
7     for(int i = 0; i < 50; i++) {
8         dq = 2.0 * length(q) * dq + 1.0;
9         q = quat_square(q) + c;
10        if(dot(q, q) > 4.0) break;
11    }
12 }
```

Salvo esta diferencia, la función que calcula la distancia de un punto al conjunto de Mandelbrot generalizado es idéntica a la utilizada en conjuntos de Julia:

$$d(p) = \frac{|q_n| \log |q_n|}{2|q'_n|},$$

teniendo en cuenta las diferencias existentes entre estos q_n y q'_n con respecto a los utilizados en Julia.

```

1 float get_dist_mandelbrot(vec3 p) {
2     float dist;
3     vec4 q = vec4(p.y, p.z, 0.0, p.x);
4     float dq = 0.0;
5     iterate_mandelbrot(q, dq);
6     float length_q = length(q);
7     return 0.5*length_q * log(length_q) / dq;
8 }
```

Por su parte, los métodos para aproximar la normal descritos en la sección 9.2.1 son igualmente válidos no sólo para este caso sino realmente para cualquier superficie, ya que son considerados métodos estándar de cálculo de normales. Por tanto podemos reutilizarlos sin más que cambiar las llamadas a `get_dist_julia` por llamadas a `get_dist_mandelbrot`.

Y una vez tenemos estas funciones, podemos modificar `ray_color` para que en lugar de utilizar las funciones de Julia utilice las de Mandelbrot. Sin embargo, en lugar de cambiar la actual, añadiremos funcionalidad. De igual manera que en la sección 7.7 explicamos que añadiendo una variable `uniform` la cual en función de su valor se graficará un conjunto u otro,

en este caso haremos exactamente lo mismo, añadir una variable `uniform` llamada `u_fractal` y en función de su valor invocaremos la SDF y el método para calcular normales de Julia o de Mandelbrot.

```
1 | uniform int u_fractal; // 1: Julia, 2: Mandelbrot
```

Tras esta pequeña modificación y darle a `u_fractal` el valor 2 mediante JavaScript, mostramos el resultado de estas operaciones en las imágenes 9.14.

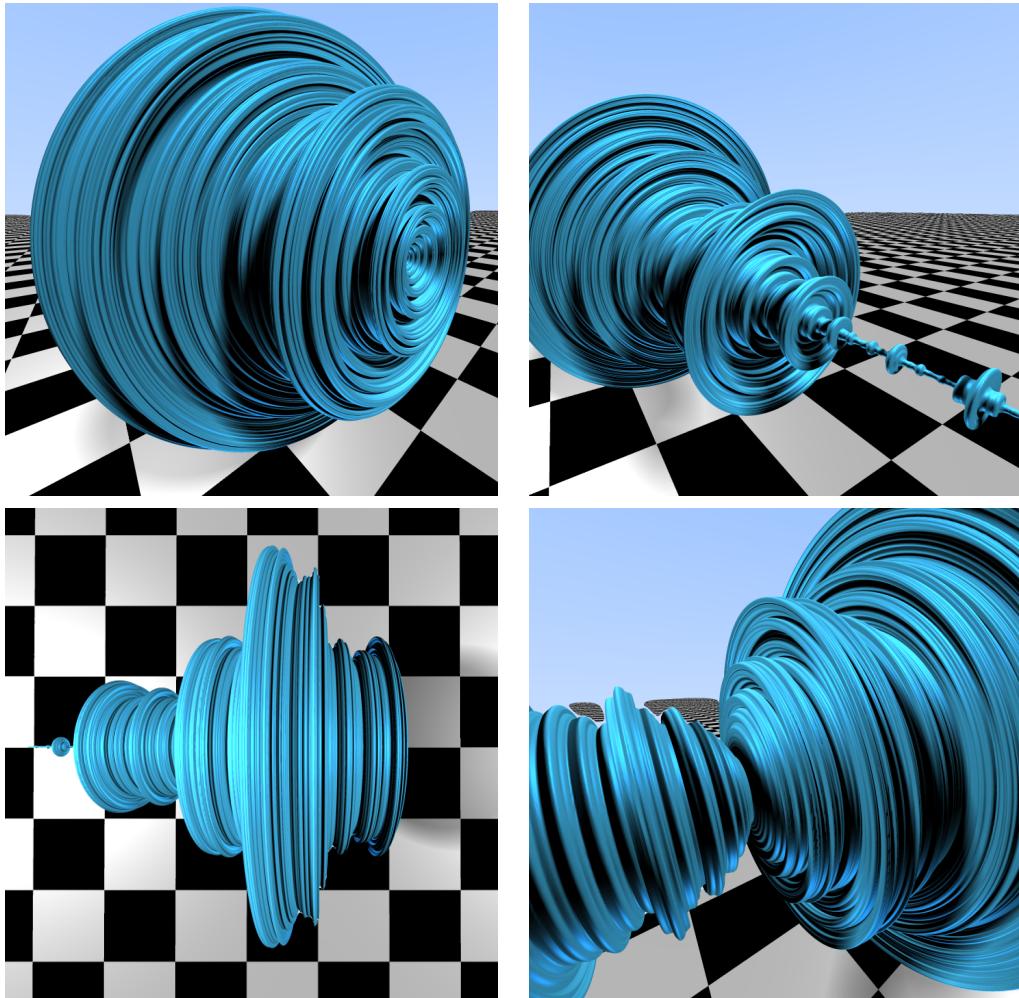


Imagen 9.14: Detalles del conjunto de Mandelbrot generalizado

Hay que reconocer que el resultado es un poco decepcionante, parecería un cuerpo generado por la revolución de la frontera del conjunto de Mandelbrot 2-dimensional en torno al eje X .

9.4. El conjunto de Mandelbulb

Gracias a los cuaternios hemos podido generalizar la definición de los números complejos a 4D y visualizar en 3D proyecciones de los conjuntos de Julia y el conjunto de Mandelbrot. Sin embargo, la apariencia del conjunto de Mandelbrot nos ha dejado un mal sabor de boca, y es que dista bastante de la belleza que nos ofrecía \mathcal{M} en dos dimensiones. En realidad, a día de hoy no se ha conseguido ninguna generalización matemáticamente correcta que posea la complejidad y la belleza del conjunto de Mandelbrot en 2D, aunque ha habido varios intentos. El conjunto

de Mandelbub es uno de estos intentos. Es una generalización tridimensional del conjunto de Mandelbrot alejada del álgebra que ofrecen cuaternios, hipercomplejos y otros tipos de álgebras, a veces inconsistentes.

Pensemos en $P_c(z) = z^2 + c$, esta vez con $z, c \in \mathbb{C}$ como la operación que, dado un complejo z multiplica su módulo por sí mismo, duplica su argumento y suma una cierta constante c . Si recurrimos en 3D a las coordenadas esféricas, las cuales identifican un punto $p \in \mathbb{R}^3$ mediante su módulo r , el ángulo que forma con el eje Y (φ) y el ángulo que formaría la proyección a \mathbb{R}^2 con el eje Z (θ), podemos extender esta operación. El procedimiento sería pasar el punto $p = (x, y, z)$ de coordenadas cartesianas a esféricas (r, φ, θ) , elevar al cuadrado r , duplicar φ y θ y devolver a coordenadas cartesianas. Es decir, tendríamos la composición

$$(x, y, z) \mapsto (r, \varphi, \theta) \mapsto (r^2, 2\varphi, 2\theta) \mapsto (x', y', z')$$

Realmente, el álgebra, las operaciones, las ‘derivadas’ utilizadas en este procedimiento y en el cálculo de la distancia al objeto no tiene ningún rigor ni fundamento matemático, pero el resultado que produce es mucho más satisfactorio.

De igual manera que utilizamos el 2 como exponente, podemos usar cualquier otro. En este contexto lo más usual es usar el 8, ya que para exponentes más altos se tiende a la simetría y a detalles menos marcados. De manera que la transformación que usaremos entonces es la función f que actúa sobre $p = (x, y, z)$ de la siguiente forma

$$f(p) = f(x, y, z) = \Phi(|p|^8, 8\varphi, 8\theta), \quad (9.15)$$

donde Φ es la función que pasa de coordenadas esféricas a cartesianas, es decir

$$\Phi(r, \varphi, \theta) = (r \sin(\varphi) \sin(\theta), r \cos(\varphi), r \sin(\varphi) \cos(\theta))$$

y donde $\varphi = \arccos(\frac{y}{|p|})$ y $\theta = \arctan(\frac{x}{z})$.

El procedimiento, en lo que se refiere a iterar un punto para posteriormente estimar la distancia al objeto es bien parecido al ya conocido, solo que en este caso en lugar de utilizar la potencia y suma de cuaternios utilizamos la transformación (9.15). Y claro está a la hora de calcular lo que entonces conocíamos como q' la iteración cambia al haber cambiado el exponente. En este caso sería $|w'_{i+1}| = 8|w|^7|w'| + 1$, de forma que, dado un punto $w \in \mathbb{R}^3$, debemos calcular las iteraciones

$$\begin{aligned} w_{i+1} &= f(w_i), & w_0 &= w \\ |w'_{i+1}| &= 8|w_i|^7|w'_i| + 1, & w'_0 &= 1 \end{aligned}$$

Tomando para f como c el valor de w inicial.

Una vez w y $|w'|$ han sido suficientemente iteradas, tenemos el punto w_n y el módulo $|w'_n|$, dispuestos a aplicarle la SDF que de nuevo es la transformación 9.12. Consultese [44] para más información sobre la SDF.

Como dijimos anteriormente, el método del gradiente de la SDF y el método del tetraedro descritos en la sección 9.2.1 son igualmente válidos para cualquier superficie supuesta conocida su SDF. Por lo que podemos también calcular las normales al conjunto de Mandelbub a partir de estos métodos.

La implementación necesaria para la visualización del conjunto de Mandelbub se basa en programar la función f (la transformación (9.15)), una función que itere un punto concreto y la sucesión de ‘derivadas’, una que invoque a la iteración y calcule la estimación de la distancia y otra que calcule las normales.

Respecto de la primera, podemos aprovechar que la GPU es muy rápida calculando funciones trigonométricas. Existen alternativas polinómicas que evitan el uso de trigonometría, la cual suele ser lenta en CPU, pero como en GPU suelen ser más rápidas, haremos uso de estas.

```

1 vec3 f_Mandelbub(vec3 w, vec3 c) {
2     // Coordenadas esféricas
3     float r = sqrt(dot(w,w));
4     float phi = acos(w.y/r);
5     float theta = atan(w.x,w.z);
6     // Escalado y rotación
7     r = pow(r, 8.0);
8     phi = phi * 8.0;
9     theta = theta * 8.0;
10    // Vuelta a coordenadas cartesianas
11    w.x = r * sin(phi)*sin(theta);
12    w.y = r * cos(phi);
13    w.z = r * sin(phi)*cos(theta);
14    return c + w;
15 }
```

Seguidamente, una función que invoque a `f_Mandelbub` para iterar `w` y `dw`. En este caso 10 iteraciones a lo sumo son suficientes, ya que elevar el módulo a 8 es una transformación que en general cambia muchísimo los valores por lo que la convergencia o divergencia es rápida.

```

1 // Itera un punto de R^3 segun la función f_Mandelbub y
2 // calcula tambien la sucesión de derivadas
3 void iterate_mandelbub(inout vec3 w, inout float dw){
4     float m;
5     vec3 c = w;
6     for(int i = 0; i < 10; i++) {
7         m = length(w);          // |z|
8         dw = 8.0*m*m*m*m*m*m*m*dw + 1.0; // 8*|z|^7*z' + 1.0
9         w = f_Mandelbub(w,c);      // w_n^8 + w_0
10        if(m > 2.0) break;       // |z| > 2
11    }
12 }
```

Por último, la función que definitivamente dado un punto $p \in \mathbb{R}^3$ devuelve la aproximación de la distancia al conjunto de Mandelbub.

```

1 float get_dist_mandelbub(vec3 p) {
2     vec3 w = p;
3     float dw = 1.0;
4     iterate_mandelbub(w,dw);      // Iteramos w y dw
5     float length_w = length(w);      // |w|
6     return 0.5*length_w * log(length_w) / dw;
7 }
```

Con este nuevo código junto con el necesario para calcular las normales y las modificaciones necesarias para que se represente el conjunto de Mandelbulb según el valor de `u_fractal`, ya podemos visualizar el mismo, véase la imagen 9.15.

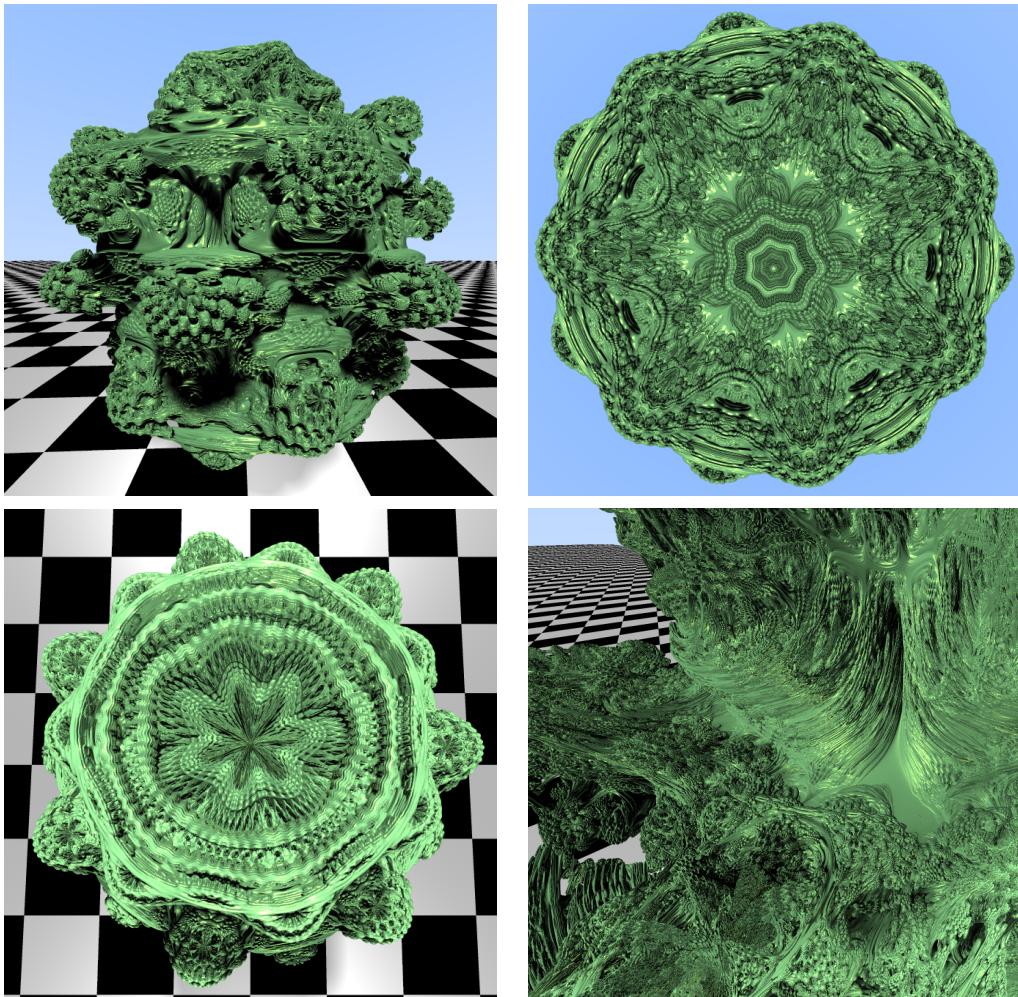


Imagen 9.15: Conjunto de Mandelbulb

Nótese como efectivamente mejora por mucho el resultado que nos ofrecía el conjunto de Mandelbrot generalizado a cuaternios de las imágenes 9.14. Sin embargo, recordamos que el álgebra y el rigor utilizado para visualizarlo está lejos de ser correcto, siendo aún así el resultado más llamativo hasta ahora. Este es el colofón a toda la teoría y todas las implementaciones que se han hecho desde que se inició con un cuadrado de colores en el capítulo 5, pudiendo al fin observar fractales tridimensionales desde distintas perspectivas, con distintos materiales y por tanto diferentes apariencias.

9.5. Comparación con los fractales 2D

En las secciones anteriores y a lo largo de todo el recorrido que hemos hecho desde el capítulo 5, el objetivo era conseguir generalizar a tres dimensiones los fractales del capítulo 3 y que con ayuda de WebGL pudimos revisualizar en el 7: conjuntos de Julia y Mandelbrot. Es por tanto momento de comparar los resultados obtenidos en 2D con los resultados obtenidos en 3D.

9.5.1. Comparación entre conjuntos de Julia 2D y 3D

Los conjuntos de Julia generalizados a tres dimensiones no son sino proyecciones a partir de un punto $p = (x, y, z) \in \mathbb{R}^3$ identificándolo con un cuaternio al cual se le anula su última componente imaginaria $q = x + yi + zj + 0k \in \mathbb{H}$. En particular, podemos tomar la tercera componente también como nula e identificar un punto $p' = (x, y) \in \mathbb{R}^2$ con un cuaternio $q' = x + yi + 0j + 0k \in \mathbb{H}$, y como además $i^2 = -1$, podemos restringir la aritmética de los cuaternios a la de los complejos. Esto, a nivel teórico, nos lleva a decir que si tomamos los puntos del plano $z = 0$, los identificamos con un cuaternio y los iteramos bajo $P_c(q) = q^2 + c$ el resultado es precisamente el conjunto de Julia 2-dimensional.

Es decir, la sección provocada por el plano $z = 0$ en los conjuntos de Julia 3D debe coincidir con los conjuntos de Julia 2D (no olvidemos que el conjunto en sí es la frontera entre conjuntos prisioneros y de escape). Esto se puede apreciar bastante bien si observamos el conjunto \mathcal{J}_{-1} en 3D, que a la vista queda su parecido con su análogo 2-dimensional, fíjese en la figura 9.16.

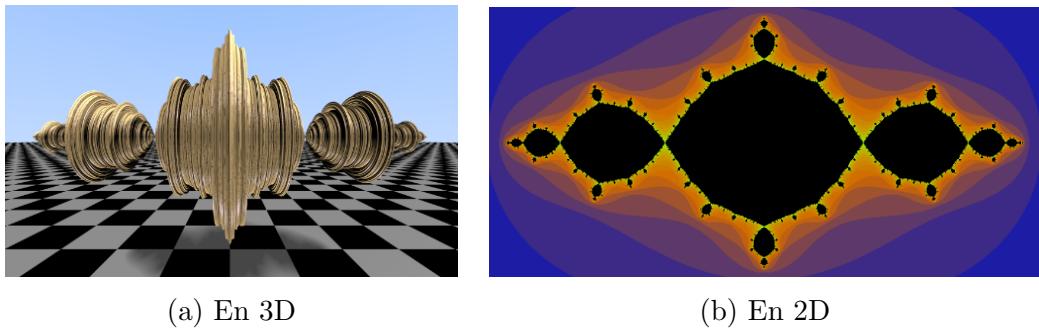


Imagen 9.16: Conjunto \mathcal{J}_{-1}

De hecho, la generalización tridimensional es la superficie generada por la revolución del conjunto en 2D en torno al eje X , fíjese en la vista ofrecida por la figura 9.17. Esto ocurre con todos los conjuntos de Julia de números puramente reales (se puede comprobar interactuando con la web <https://jantoniov.github.io/Geometria-Fractal/3D-fractals.html>).

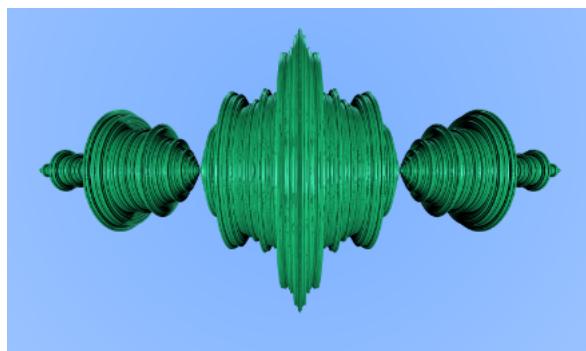
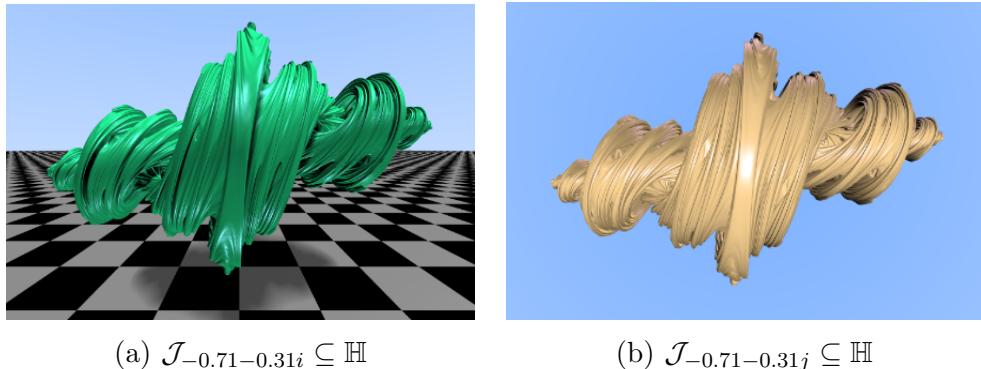


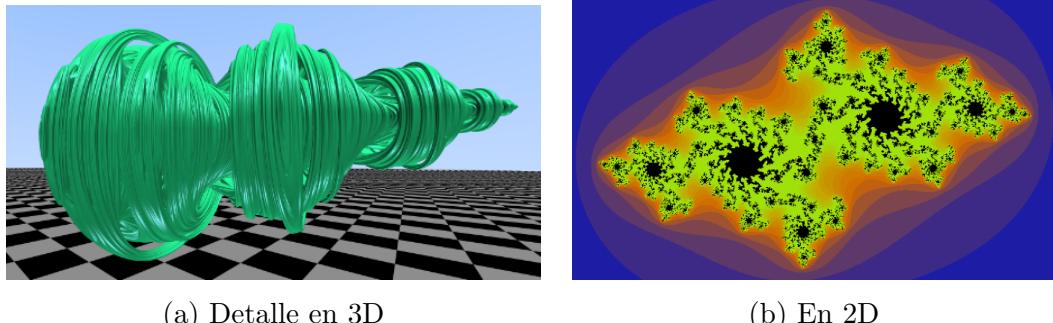
Imagen 9.17: \mathcal{J}_{-1} visto desde abajo

De igual manera que anulando la componente en z se obtiene en la sección del plano $z = 0$ el conjunto 2D, si se anula la componente en y y se libera la z pasa lo propio en el plano $y = 0$. Fíjese como la sección de $\mathcal{J}_{-0.71-0.31i}$ en el plano $z = 0$ (imagen 9.18 (a)) es la misma que la de $\mathcal{J}_{-0.71-0.31j}$ en el plano $y = 0$ (imagen 9.18 (b)).

Además, aunque los detalles 3D no nos dejan apreciar claramente las similitudes entre el

(a) $\mathcal{J}_{-0.71-0.31i} \subseteq \mathbb{H}$ (b) $\mathcal{J}_{-0.71-0.31j} \subseteq \mathbb{H}$ Imagen 9.18: Generalizaciones de $\mathcal{J}_{-0.71-0.31i} \subseteq \mathbb{C}$

conjunto $\mathcal{J}_{-0.71-0.31i}$ en \mathbb{C} y en \mathbb{H} , en la figura 9.19 podemos observar ciertos parecidos entre los bulbos que tiene el fractal en 3D a su izquierda y los que tiene el fractal en 2D.



(a) Detalle en 3D

(b) En 2D

Imagen 9.19: Conjunto $\mathcal{J}_{-0.71-0.31i}$

9.5.2. Comparación entre el conjunto de Mandelbrot 2D y 3D

De manera análoga a lo explicado para conjuntos de Julia basados en la iteración de cuaternios, un cuaternionio $q = x + yi + 0j + 0k$ puede ser identificado con un número complejo $x + yi$. Entonces, si un complejo $c = c_x + c_yi \in \mathbb{C}$ pertenece a \mathcal{M} , necesariamente un cuaternionio $q = c_x + c_yi + 0j + 0k$, el cual en 3D se identifica con un punto $(c_x, c_y, 0)$ perteneciente al plano $z = 0$, debe también pertenecer al conjunto de Mandelbrot en 3D. Entonces, el conjunto de Mandelbrot tridimensional debe dibujar al conjunto de Mandelbrot 2D en el plano $z = 0$. Análogamente, esto también ocurre en el plano $y = 0$ si anulamos la coordenada en y en lugar de la z . Nos remitimos a la sección 9.3 y a las imágenes 9.14 para recordar que la generalización 3D no era más que la superficie generada por la revolución de \mathcal{M} en torno al eje X , por lo que queda comprobada la propiedad que acabamos de describir.

No obstante, en las imágenes 9.20 presentamos algunas imágenes comparativas para que se pueda apreciar directamente la similitud entre \mathcal{M} en 2D y en 3D. Precisamente por la revolución dejan de observarse a simple vista detalles como los bulbos o la cardioide que constituye el cuerpo principal de \mathcal{M} .

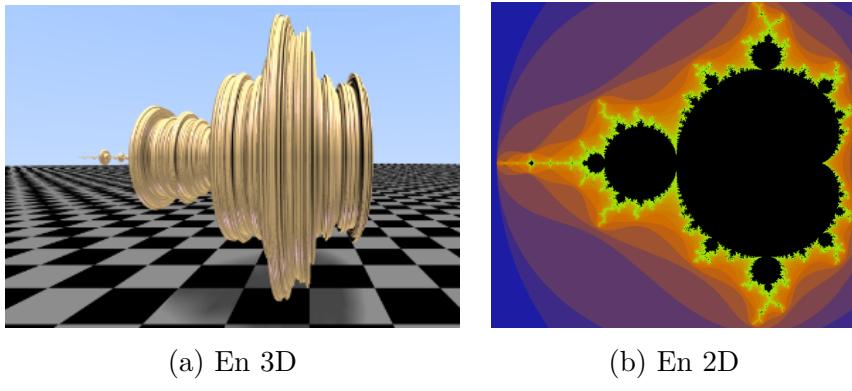


Imagen 9.20: Conjunto de Mandelbrot

9.5.3. Comparación entre el conjunto de Mandelbub y \mathcal{M}_8

En realidad el conjunto de Mandelbub no es sino una forma de generalizar, de forma matemáticamente poco precisa, el conjunto de Mandelbrot de orden 8, el cual nos remitimos a la sección 3.6.1 para recordar su génesis. En la imagen 9.21 (b) podemos recordar el aspecto que tenía \mathcal{M}_8 . Aparentemente, está dividido en lo que parecen 7 regiones, cada una con infinitos bulbos donde destaca uno que es más grande que los demás.

La parte de Mandelbub que más recuerda a \mathcal{M}_8 es la parte inferior (imagen 9.21 (a)), que aunque no es igual, también se aprecia que está dividida en 7 regiones iguales. A su vez, el propio cuerpo de Mandelbub tiene 7 salientes en cada uno de sus dos niveles (imagen 9.15), haciendo cierta analogía con \mathcal{M}_8 .

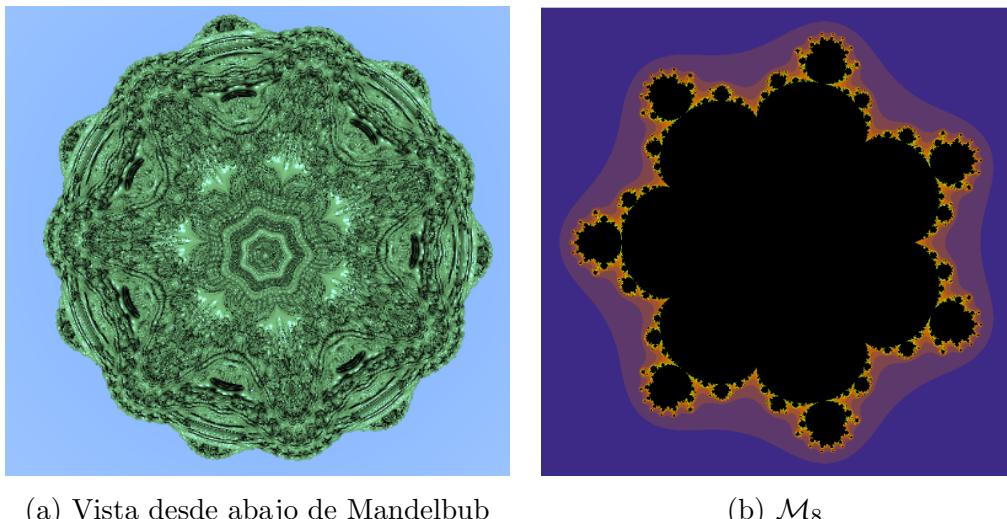


Imagen 9.21: Conjunto de Mandelbub y de Mandelbrot de orden 8

9.6. Posibles optimizaciones

El software, en lo que a funcionalidad se refiere está completo, sin embargo puede ser lento en algunos dispositivos dependiendo de la GPU que posean. En gran medida, lo que más ralentiza la ejecución es el hecho de evaluar la SDF correspondiente en cada iteración del sphere-tracing, ya que esto supone iterar el punto y aplicar transformaciones como el logaritmo que son más costosas. Por lo que trataremos de introducir algunas medidas que nos permitan ahorrarnos algunas ejecuciones de las SDFs, todas ellas basadas en una misma idea: las esferas englobantes.

9.6.1. Esferas englobantes

Si se lanza un rayo que está muy lejano al conjunto que se esté visualizando o que apunta directamente al cielo o al suelo, antes de converger al suelo o diverger al cielo hay que evaluar, sobre todo en este último caso, hasta mil veces la SDF cuando realmente casi desde el principio podría saberse que no va a intersecar al conjunto. Para ahorrarnos todas estas evaluaciones podemos utilizar lo que se denominan esferas englobantes (*bounding spheres*).

La técnica consiste en crear una esfera que ‘englobe’ el conjunto, en el sentido de que éste quepa completamente en el interior de la misma. Antes de aplicar sphere-tracing, reutilizaríamos el código que ya programamos en la sección 8.5.1 para calcular analíticamente la intersección de un rayo con una esfera, aunque esta vez sólo nos interesa saber si el rayo interseca la esfera englobante, no el punto ni el valor de t . Si se calcula que el rayo no golpea la esfera englobante, entonces no se calcula la distancia al conjunto en ninguna iteración, pues sabemos que no habrá intersección con el mismo, aunque aún puede haber intersección con el plano, pero al no tener que calcular la SDF del conjunto que se esté visualizando la ejecución es mucho más rápida.

Para implementar este método, hemos probado empíricamente qué radio de esfera se ajusta mejor a los conjuntos de Julia, al conjunto de Mandelbrot y al conjunto de Mandelbulb, obteniendo respectivamente 2.5, 2 y 1.5 respectivamente. Por tanto, ajustamos el radio de esta esfera y sólo calculamos las SDFs en caso de impacto.

```
1 // Devuelve true si R interseca a S, false en otro caso
2 bool hit_sphere_limits( Sphere S, Ray R ){
3     vec3 oc = R.orig - S.center;
4     float a = dot(R.dir, R.dir);
5     float b = 2.0 * dot(oc, R.dir);
6     float c = dot(oc, oc) - S.radius*S.radius;
7     float discriminant = b*b - 4.0*a*c;
8     return discriminant >= 0.0;
9 }
10
11 vec4 ray_color(Ray r, ...) {
12     // ...
13     Sphere bounding_sphere;
14     bounding_sphere.center = vec3(0.0, 0.0, 0.0);
15     bounding_sphere.radius = u_fractal == 1 ? 1.5 :
16     (u_fractal == 2 ? 2.5 : 2.0);
17     bool hits_bounding_sphere = hit_sphere_limits(bounding_sphere, r);
18
19     // Sphere Tracing
```

```

20  for(int i = 0; i < MAX_STEPS; i++) {
21      // Distancia al plano ...
22      // ...
23      // Distancia a Julia/Mandelbrot/Mandelbulb
24      if(hits_bounding_sphere) {
25          dist = get_dist(p)
26          // ...
27      }
28      // ...
29  }
30  // ...
31 }
```

Y con esta simple técnica podemos optimizar bastante las escenas en las que el fractal aparece más lejano. Sin embargo, si queremos ampliar detalles del conjunto no influiría, pues la gran mayoría de los rayos impactan con la esfera englobante. Aún así, en caso de estar cerca las iteraciones que se dan representan un número pequeño, así que mejoramos el rendimiento de una u otra forma.

9.6.2. Optimización de sombras con esferas englobantes

En ciertos píxeles hay que aplicar varias veces sphere-tracing, es el caso de los píxeles que corresponden a puntos del fractal o del plano que consideramos como suelo, ya que como se ha explicado en la sección 8.7.4, la forma de añadir sombras a la escena es, a partir del punto donde se produce la intersección, lanzar un rayo en la dirección de cada una de las fuentes de luz y buscar posibles intersecciones con el conjunto que se visualice en la escena, asignando así a cada fuente un coeficiente de visibilidad. Esto supone aplicar varias veces sphere-tracing, lo que a su vez implica muchas ejecuciones de las SDFs, lo cual también ralentiza la ejecución.

Sin embargo, al igual que con el sphere-tracing principal de `ray_color`, podemos desde el principio descartar todos los rayos que no intersequen la esfera englobante. Esta mejora se puede hacer con una ligera modificación del método `light_is_visible`.

```

1 float light_is_visible(Directional_light light, vec3 p) {
2     Ray R;
3     R.dir = normalize(light.dir);
4     R.orig = p;
5     // ...
6     if(!hit_sphere_limits(bounding_sphere, R))
7         return 1.0;
8     // Sphere Tracing
9     for(int i = 0; i < 50; i++ ) {
10         // ...
11     }
12     return res;
13 }
```

El inconveniente es que esta mejora sólo puede aplicarse a ocasiones en las que el canvas visualice puntos lejanos al conjunto, por lo que cuando se esté graficando un detalle de un fractal puede ser poco útil. No obstante, en estos casos al haber menos puntos del plano directamente

no tiene por qué ejecutarse en gran medida el código correspondiente a las sombras, por lo que hemos mejorado la eficiencia en los casos donde se ejecutaba dicho código más veces, es decir, en planos más abiertos y lejanos al conjunto.

CONCLUSIONES

Para finalizar la redacción de esta memoria, pondremos en valor algunas conclusiones extraídas durante el desarrollo de este trabajo. En primer lugar, nos gustaría resaltar que, salvo las bases, todo este trabajo ha sido sobre una materia totalmente nueva. No existe ninguna materia en el grado ni de informática ni de matemáticas que explique geometría fractal, a pesar de ser una disciplina no sólo muy hermosa visualmente, sino muy relacionada tanto con una como con otra.

De hecho, llama la atención el número de distintas áreas de la ciencia y de las matemáticas que confluyen en la geometría fractal, al menos en este TFG. Ya en la introducción vimos la cantidad de diferentes ámbitos que se recomendaba conocer: análisis, variable compleja, matemática aplicada, geometría euclídea... O por ejemplo, el artículo [7] del cual se extrae el ‘potencial de Hubbard-Douady’, que realmente es un concepto puramente matemático aparentemente poco relacionado con esta disciplina, pero finalmente sirve para calcular una SDF con la que visualizar imágenes fractales en 3D.

Además de no enseñarse en el grado, tampoco es un área de investigación en ningún departamento ni hay realmente profesores en esta universidad expertos en esta materia. Prueba de ello es lo difícil que fue encontrar un profesor dispuesto a tutorizar este tema en la parte de matemáticas. Por suerte, finalmente esta idea que surgió hace unos años ha podido ser realizada.

La geometría fractal es también muy utilizada en los efectos especiales de películas, como mínimo en la música, pues las ondas sonoras tienen estructura fractal. Más allá de la música, por ejemplo en la película *Limitless* (2011) se utiliza un zoom fractal en su inicio². Otro ejemplo son las ‘técnicas de renderizado fractal utilizadas’ en *Lucy* (2014), publicadas en SIGGRAPH 2014 (más información en [46]). Estos y otros muchos ejemplos de la presencia de fractales en películas pueden consultarse en [33]. Por su parte, los IFS también tienen un gran protagonismo en la industria de los videojuegos. De la misma forma que se puede construir un helecho con un SFI se pueden construir en 3D árboles, olas del mar o incluso montañas.

Si miramos al futuro, con las nuevas tecnologías y unidades de procesado cada vez más rápidas y eficientes esta ciencia llegará lejos y podremos hacer uso de ella con niveles de realismo y resoluciones cada vez mayores, a la par que contribuir a industrias como las del cine o los videojuegos entre otras. Esta es una materia en expansión y que vemos que puede ser muy rica en aportes a otras disciplinas, por lo que incitamos a quien lo desee que entre a descubrir la magia

²Se puede ver en https://www.youtube.com/watch?v=uy_NJjRT3zk

de estas extrañas figuras para poder no solo disfrutar de una experiencia placentera, sino quien sabe si poder aportar el día de mañana su granito de arena.

Muchas gracias al lector por su atención, deseo de todo corazón que haya disfrutado leyendo esta memoria como yo haciéndolo, siempre pensando en usted y en todo el que desee conocer la geometría fractal.

BIBLIOGRAFÍA

- [1] Atkinson, K y Han, W. (2009). *Theoretical Numerical Analysis: A Functional Analysis Framework*, 39. Springer New York, 3rd edition. Disponible en <https://doi.org/10.1007/978-1-4419-0458-4>.
- [2] Bandt, C., Viet Hung, N. y Rao, H. (2006). On the Open Set Condition for Self-Similar Fractals. *Proceedings of the American Mathematical Society*, 134(5):1369–1374. Disponible en <http://www.jstor.org/stable/4097989>.
- [3] Barnsley, M (1993). *Fractals everywhere*. Academic Press, 2nd edition.
- [4] Benyamini, Y. (1998). Applications of the Universal Surjectivity of the Cantor Set. *The American Mathematical Monthly*, 105(9):832–839. Disponible en <https://arxiv.org/pdf/1303.3810.pdf>.
- [5] Conway, J. H. y Smith, D. A. (2005). *On quaternions and octonions*. CRC Press LLC.
- [6] Crane, K. (2005). Ray Tracing Quaternion Julia Sets on the GPU. *University of Illinois at Urbana-Champaign*. Disponible en <https://www.cs.cmu.edu/~kmcrane/Projects/QuaternionJulia/paper.pdf>.
- [7] Douady, A. y Hubbard, J. (2009). *Exploring the Mandelbrot set. The Orsay Notes*. Cornell University, Ithaca, NY. Disponible en <https://pi.math.cornell.edu/~hubbard/OrsayEnglish.pdf>³.
- [8] Dreher, F. y Samuel, T. (2014). Continuous Images of Cantor’s Ternary Set. *The American Mathematical Monthly*, 121(7):640–643. Disponible en <https://doi.org/10.4169/amer.math.monthly.121.07.640>.
- [9] Dubeau, F. y Gnang, C. (2018). Fixed Point and Newton’s Methods in the Complex Plane. *Journal of Complex Analysis*, 2018:1–11. Disponible en <https://doi.org/10.1155/2018/7289092>.

³Esta referencia es la traducción a inglés de las notas en francés de un curso dado entre 1983 y 1984, donde se presentan resultados publicados en las referencias [DH1] y [Do1] (1982).

- [10] Dummit, E. (2015). *Dynamics, Chaos, and Fractals (part 4): Fractals*. Rochester MTH 215. Disponible en https://web.northeastern.edu/dummit/docs/dynamics_4_fractals.pdf.
- [11] Edgar, G.A. (2008). *Measure, Topology, and Fractal Geometry* Undergraduate Texts in Mathematics. Springer New York, 2nd edition. Disponible en <https://doi.org/10.1007/978-0-387-74749-1>.
- [12] Falconer, K. (1990). *Fractal geometry: mathematical foundations and applications*. John Wiley.
- [13] Foroutan-pour, K., Dutilleul, P. y Smith, D. (1999). Advances in the implementation of the box-counting method of fractal dimension estimation. *Applied Mathematics and Computation*, 105(2):195–210. Disponible en [https://doi.org/10.1016/s0096-3003\(98\)10096-6](https://doi.org/10.1016/s0096-3003(98)10096-6).
- [14] Hart, J., Sandin, D. y Kauffman, L. (1989). Ray tracing deterministic 3-D fractals. *ACM SIGGRAPH Computer Graphics*, 23(3):289–296. Disponible en <https://doi.org/10.1145/74334.74363>.
- [15] Hart, J. (1995). Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces. *The Visual Computer*, 12:527–545. Disponible en <https://doi.org/10.1007/s003710050084>.
- [16] Hausdorff, F. (1919). Dimension und ausseres Mass. *Mathematische Annalen*, 79: 157–179.
- [17] Hughes, J.F., Van Dam, A., McGuire, M., Sklar, D.F., Foley, J.D., Feiner, S.K. y Akeley, K. (2013). *Computer graphics: principles and practice*. Addison-Wesley Professional, Boston, MA, 3rd edition.
- [18] Hurewicz, W. y Wallman, H. (1948). *Dimension Theory*. Princeton mathematical series; 4. Princeton University Press.
- [19] Kunze, H., La Torre, D., Mendivil, F. y Vrscay, E.R. (2019). Self-similarity of solutions to integral and differential equations with respect to a fractal measure. *Fractals*, 7. Disponible en <https://doi.org/10.1142/S0218348X19500142>.
- [20] Kunze, H. y La Torre, D. (2021). Solving Parameter Identification Problems using the Collage Distance and Entropy. *Recent Developments in Mathematical, Statistical and Computational Sciences. AMMCS 2019. Springer Proceedings in Mathematics and Statistics*, 343:167–175, Cham. Springer International Publishing. Disponible en https://doi.org/10.1007/978-3-030-63591-6_16.
- [21] Mandelbrot, B. (1983). *The Fractal geometry of nature*. Freeman, New York.
- [22] Milnor, J. (2011). *Dynamics in One Complex Variable*. Princeton University Press, 3rd edition. Disponible en <https://doi.org/10.1515/9781400835539>.
- [23] Moran, P.A.P. (1946). Additive functions of intervals and Hausdorff measure. *Mathematical Proceedings of the Cambridge Philosophical Society* 42(1):15–23).

- [24] Ostrowski, A.M. (1973). *Solution of equations in Euclidean and Banach spaces*. Academic Press, 3rd edition.
- [25] Payá, R (2008). *Apuntes de Análisis Matemático I*. Disponibles en https://www.ugr.es/~rpaya/documentos/AnalisisI/2021-22/Apuntes_05.pdf.
- [26] Pharr, M., Jakob, W. y Humphreys, G. (2017). 02 - Geometry and transformations. En Pharr, M., Jakob, W. y Humphreys, G., *Physically Based Rendering* 57–121. Morgan Kaufmann, Boston, 3rd edition. Disponible en <https://doi.org/10.1016/B978-0-12-800645-0.50002-6>.
- [27] Rubiano, G.N. (2013). *Iteración y Fractales (con Mathematica ®)*. Universidad Nacional de Colombia.
- [28] Snyder, S. (2006). Fractals and the Collage Theorem. *MAT Expository Papers*, 49. Disponible en <https://digitalcommons.unl.edu/mathmidexpap/49>.
- [29] Wagon, S. (2010). *Mathematica in Action: Problem Solving Through Visualization and Computation*. Springer Publishing Company, Incorporated, 3rd edition.

Recursos web

- [30] *Adding 2D content to a WebGL context - Web APIs* — MDN. (2022, 24 abril). MDN Web Docs. Recuperado 5 de mayo de 2022, de <https://mzl.la/3x5V5TH>.
- [31] Colaboradores de Wikipedia. (2022, 16 febrero). *WebGL*. Wikipedia, la enciclopedia libre. Recuperado 5 de mayo de 2022, de <https://es.wikipedia.org/wiki/WebGL>.
- [32] *Data in WebGL - Web APIs* (2022, 14 marzo). MDN Web Docs. Recuperado 6 de mayo de 2022, de https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Data.
- [33] Doolan, D. C. (2018, 4 diciembre). *Examples of Graphics, Animation and Fractals in Film*. Dr. Daniel C. Doolan News and Photos. Recuperado 29 de mayo de 2022, de <https://dcdoolan.wordpress.com/2017/01/23/examples-of-graphics-animation-and-fractals-in-film/>.
- [34] *Getting started with WebGL - Web APIs*. (2022, 24 abril). MDN Web Docs. Recuperado 5 de mayo de 2022, de https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Tutorial/Getting_started_with_WebGL.
- [35] Glawion, A. (2022, 12 abril). *CPU vs. GPU Rendering – What's the difference and which should you choose?* CG Director. Recuperado 20 de mayo de 2022, de <https://bit.ly/3NnTVd9>.
- [36] *LearnOpenGL - OpenGL*. (s. f.). OpenGL. Recuperado 5 de mayo de 2022, de <https://learnopengl.com/Getting-started/OpenGL>.
- [37] López, P. (2020, 30 abril). *Ray Tracing: ¿Qué es y para qué sirve? - Definición*. GEEKNETIC. Recuperado 10 de mayo de 2022, de <https://www.geeknetic.es/Ray-Tracing/que-es-y-para-que-sirve>.

- [38] *OpenGL ES - The Standard for Embedded Accelerated 3D Graphics*. (2011, 19 julio). The Khronos Group. Recuperado 7 de mayo de 2022, de <https://www.khronos.org/api/opengles>.
- [39] *Pythagorean Tree*. (s. f.). Agness Scott College. Recuperado 2 de mayo de 2022, de <https://larryriddle.agnesscott.org/ifs/pythagorean/pythTree.htm>.
- [40] Quilez, I. (2010). *Soft shadows in ray-marched SDFs*. Íñigo Quilez. Recuperado 30 de mayo de 2022, de <https://iquilezles.org/articles/rmshadows/>.
- [41] Quilez, I. (s. f.). *3D SDFs*. Íñigo Quilez. Recuperado 14 de mayo de 2022, de <https://iquilezles.org/articles/distfunctions/>
- [42] Quilez, I. (s. f.). *Normals for an SDF*. Íñigo Quilez. Recuperado 15 de mayo de 2022, de <https://iquilezles.org/articles/normalsSDF/>.
- [43] Quilez, I. (2004). *Distance to fractals*. Íñigo Quilez. Recuperado 16 de mayo de 2022, de <https://iquilezles.org/articles/distancefractals/>.
- [44] Quilez, I. (2009). *Mandelbulb*. Íñigo Quilez. Recuperado 16 de mayo de 2022, de <https://iquilezles.org/articles/mandelbulb/>.
- [45] Shirley, P. (2020). *Ray Tracing in One Weekend*. Recuperado 28 de mayo de 2022, de <https://raytracing.github.io/books/RayTracingInOneWeekend.html>.
- [46] *SIGGRAPH 2014 News: ILM Reps Present New Fractal Rendering Technique*. (2017, 1 junio). Pluralsight. Recuperado 29 de mayo de 2022, de <https://www.pluralsight.com/blog/film-games/siggraph-2014-news-ilm-reps-present-new-fractal-rendering-technique>.
- [47] *WebGL: 2D and 3D graphics for the web - Web APIs*. (2022, 27 abril). MDN Web Docs. Recuperado 7 de mayo de 2022, de https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API.
- [48] *WebGL Fundamentals - HTML5 Rocks*. (s. f.). HTML5 Rocks - A Resource for Open Web HTML5 Developers. Recuperado 5 de mayo de 2022, de https://www.html5rocks.com/en/tutorials/webgl/webgl_fundamentals/.
- [49] *WebGL model view projection - Web APIs* (2022, 26 abril). MDN Web Docs. Recuperado 27 de mayo de 2022, de https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/WebGL_model_view_projection#clip_space.
- [50] *WebGLRenderingContext - Web APIs*. (2022, 20 enero). WebGLRenderingContext. Recuperado 7 de mayo de 2022, de <https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext>.

Multimedia

- [51] [Imagen] Curvas de nivel. Civilgeeks. Recuperado el 16 de mayo de 2022, de <https://civilgeeks.com/wp-content/uploads/2011/08/curvas-de-nivel2.png>.
- [52] [Imagen] Esponja de Menger. Matemáticas cercanas. Recuperado 27 de febrero de 2022, de https://i0.wp.com/matematicascercanas.com/wp-content/uploads/2014/08/esponja_menger.png?resize=300%2C300&ssl=1.
- [53] [Imagen] Paisaje con calima. Escestaticos. Recuperado 11 de mayo de 2022, de <https://bit.ly/3ziX220>.
- [54] [Imagen] Rayo. Meteored. Recuperdado 25 de abril de 2022, de <https://bit.ly/3NkzAFJ>.
- [55] [Imagen] Romanescu. Wikipedia, la enciclopedia libre. Recuperado 25 de febrero de 2022, de https://upload.wikimedia.org/wikipedia/commons/thumb/5/5e/Romanesco_broccoli_%28Brassica_oleracea%29.jpg/800px-Romanesco_broccoli_%28Brassica_oleracea%29.jpg.
- [56] [Vídeo] Two Minute Papers. (2022, 26 abril). *NVIDIA's Ray Tracing AI - This is The Next Level!*. YouTube. Disponible en <https://www.youtube.com/watch?v=y11jkmF7Xug>.

APÉNDICE A

DOCUMENTACIÓN DEL CÓDIGO JAVASCRIPT

En este apéndice documentaremos todas las clases, métodos y atributos que se han utilizado en el lenguaje JavaScript. Estas han sido utilizadas para gestionar desde un nivel más alto de abstracción los distintos componentes de WebGL explicados en la sección 5.2, modificar el DOM de los documentos HTML y darle valores a las variables `attribute` y, de forma dinámica, también a las variables `uniform` del fragment shader. Mencionar también el uso de la biblioteca [jQuery](#), la cual mediante una sintaxis simplificada de JavaScript nativo facilita en gran parte la programación en dicho lenguaje. Los ficheros que aquí documentamos pueden ser todos ellos encontrados en <https://github.com/JAntonioVR/Geometria-Fractal/tree/main/static/js>, donde además de encontrar las funciones aquí descritas con su código fuente, también encontraremos la misma cabecera descriptiva que incluimos en este apéndice.

Presentamos también en la sección A.4 un diagrama UML con las relaciones que tienen estas clases entre sí.

A.1. Componentes de WebGL comunes a 2D y 3D

Comenzaremos documentando las clases y métodos comunes a la visualización de fractales 2D y 3D. Por ejemplo, la estructura de los shaders es la misma, aunque cambia drásticamente el código fuente de un caso a otro. También se utiliza un mismo buffer de vértices inicial para el vertex shader y ciertas inicializaciones de la escena son exactamente las mismas.

A.1.1. Fichero `shader.js`

Fichero con el código relativo a una abstracción de un shader. Hay dos posibles tipos de Shader (`ShaderType`): Vertex y Fragment (clase `Shader`), que juntos forman lo que llamamos ‘Programa Shader’ (clase `ShaderProgram`).

Enumerado `ShaderType`

Enumerado inmutable, de forma que un elemento de la clase `Shader` sólo puede tener un tipo: `ShaderType.vertexShader` o `ShaderType.fragmentShader`.

Clase `Shader`

Clase que abstrae un shader de WebGL, entendiendo shader como el programa relativo a un vertex shader o a un fragment shader.

Atributos

- `program`: `WebGLShader`. Programa shader compilado.
- `source`: `string`. Código fuente.
- `type`: `ShaderType`. Tipo de shader. Vertex o fragment.

Métodos

- `constructor`: A partir del contexto de WebGL, del código fuente y del tipo de Shader, este método crea y compila el vertex/fragment shader.

Argumentos:

- `gl`: `WebGLContext`. Contexto de WebGL.
- `source`: `string`. Código fuente del shader.
- `type`: `ShaderType`. Tipo de Shader, vertex o fragment.

Devuelve: Un elemento de la clase `Shader` inicializado.

- `getProgram`: Getter del atributo `program`, de tipo `WebGLShader`.

Clase `ShaderProgram`

Clase que representa una abstracción de un programa shader, formado por el enlazado de un vertex shader y un fragment shader.

Atributos

- `vertexShader`: `Shader`. Objeto de la clase `shader` que corresponde a un vertex shader.
- `fragmentShader`: `Shader`. Objeto de la clase `shader` que corresponde a un fragment shader.
- `program`: `WebGLProgram`. Programa shader ya inicializado a partir de la conjunción de vertex y fragment shader.

Métodos

- `constructor`: Utiliza el contexto de WebGL y dos instancias de la clase `Shader`, uno de tipo vertex y otro de tipo fragment para inicializar el programa shader definitivo.

Argumentos

- `gl`: `WebGLContext`. Contexto de WebGL.
- `vs`: `Shader`. Vertex Shader.

- **fs**: `Shader`. Fragment Shader.

Devuelve: Un elemento de la clase `ShaderProgram` inicializado.

- `getShaderProgram`: Getter del atributo `program`, de la clase `WebGLProgram`.

A.1.2. Fichero `buffer.js`

Fichero que contiene una abstraccion de un buffer de datos de WebGL.

Clase `Buffer`

Clase que contiene un atributo del tipo `WebGLBuffer`, información sobre el mismo y encapsula el comportamiento relativo a la inicialización de un buffer de WebGL.

Atributos

- `buffer`: `WebGLBuffer`. Buffer de WebGL que contendrá la información requerida.
- `nElements`: `number`. Número de elementos (como pueden ser vértices, colores, coordenadas de textura...) almacenadas en el buffer.
- `nValuesPE`: `number`. Numero de valores por elemento. Número de valores consecutivos que representan a cada elemento. Por ejemplo, un vértice 2D serían dos valores, un color RGB serían tres valores, etc.

Métodos

- `constructor`: A partir del contexto de WebGL y de un array de JavaScript, construye y almacena en un atributo un elemento de la clase `WebGLBuffer`.

Argumentos

- `gl`: `WebGLContext`. Contexto de WebGL.
- `array`: `Array`. Array de JavaScript a partir del cual se crea el buffer.
- `nElements`: `number`. Número de elementos que se almacenan en el buffer.
- `nValuesPE`: `number`. Número de valores consecutivos que componen cada elemento.

Devuelve: Un elemento de la clase `Buffer` inicializado.

- `getBuffer`: Getter del atributo `buffer`, de la clase `WebGLBuffer`.
- `getNumberOfElements`: Getter del atributo `nElements`, de tipo `number`.
- `getNumberOfValuesPerElements`: Getter del atributo `nValuesPE`, de tipo `number`.

A.1.3. Fichero `scene.js`

Este fichero contiene los atributos y métodos de la clase ‘Scene’, una clase abstracta que representa una escena 2D o 3D.

Clase `Scene`

Es una clase abstracta que contiene el código común de una escena 2D y una 3D. No podremos por tanto declarar una variable que sea de tipo `Scene`, tan sólo podremos utilizar clases que hereden de esta.

Atributos

- `context`: `WebGLContext`. Contexto de WebGL.
- `shaderProgram`: `ShaderProgram`. Programa Shader utilizado para graficar la escena en el canvas.
- `buffer`: `Buffer`. Objeto de la clase Buffer que almacenará las posiciones de los vértices que utilizaremos para el vertex shader.

Métodos

- `constructor`: A partir del código fuente del vertex shader y del fragment shader inicializa el contexto de WebGL, el programa Shader y el buffer de posiciones de los vertices que toma como entrada el vertex shader.

Argumentos

- `vsSource`: `string`. Código fuente del vertex shader
- `fsSource`: `string`. Código fuente del fragment shader.

Devuelve: Nada, realmente este constructor no se puede llamar por si solo, únicamente tiene el código comun a escenas 2D y 3D.

- `initShaderProgram`: Crea y compila el vertex shader y el fragment shader. A partir de estos dos shaders, se crea el programa shader.

Argumentos

- `vsSource`: `string`. Código fuente del vertex shader
- `fsSource`: `string`. Código fuente del fragment shader.

Devuelve: El programa shader ya inicializado, de la clase `ShaderProgram`.

- `initBuffers`: Inicializa los buffer necesarios, en nuestro caso tan solo se trata del buffer de posiciones de los vértices que recibe el vertex shader.

Argumentos: No acepta.

Devuelve: Un objeto de la clase `Buffer` con informacion sobre el buffer de posiciones.

- `drawScene`: Método abstracto que en las clases que hereden de `Scene` tomará el shader, los buffer y los parámetros de la escena para visualizarla en el canvas.
- `checkGLError`: Método que comprueba si hay algún error de OpenGL, en cuyo caso lanza una excepción.

Argumentos: No acepta.

Devuelve: Nada, tan solo lanza la excepcion cuando sea necesario.

A.2. Visualización de fractales 2D

Con la sección anterior hemos aclarado que utilizaremos el mismo buffer de posiciones, las mismas clases para el programa shader y mucho código común para las escenas. Las diferencias fundamentales entre el código JavaScript dedicado al uso de WebGL para visualizar fractales 2D y 3D se encuentran en los parámetros de la escena y en las maneras de gestionar los eventos. En esta sección describiremos todos estos aspectos en lo relativo a escenas 2D.

Recordamos que durante todo el capítulo 7 estuvimos principalmente describiendo el código del fragment shader, pues es este el que se ejecuta en cada píxel, de forma que identificábamos cada píxel del canvas con un punto del plano complejo \mathbb{C} (o varios si se aplica [SSAA](#)) y calculábamos si éste pertenecía al conjunto de Mandelbrot o al conjunto de Julia que decidíramos. Para ayudarnos a visualizar distintas regiones del plano, distintos conjuntos de Julia fijando diferentes constantes $c \in \mathbb{C}$ o cambiar el exponente de la función $P_{c,m} = z^m + c$ utilizamos variables `uniform` que insistíamos que se asignaba su valor utilizando JavaScript. La clase que gestiona esa asignación de valores y el cambio de forma dinámica de los mismos es `Scene2D`, de ahí el valor de esta clase. En concreto, y aunque ahora explicaremos su significado en relación con los atributos de la clase, las distintas variables `uniform` que hemos utilizado en el shader (fichero ‘fragment-shader-2D-fractals.js’) son:

```
1 uniform vec2 u_zoomCenter;
2 uniform float u_zoomSize;
3 uniform int u_maxIterations;
4 uniform vec2 u_juliaSetConstant;
5 uniform int u_order;
6 uniform int u_fractal;
7 uniform bool u_antialiasing;
8 uniform int u_nSamples;
```

Recordando y teniendo estos detalles en mente, procedemos a documentar la clase `Scene2D`.

A.2.1. Fichero `scene2D.js`

Código respectivo a la clase `Scene2D`, que representa una escena en la que podemos visualizar fractales en 2D.

Clase `Scene2D`

Es una clase que hereda de `Scene` y que contiene el código relativo a la creación, parámetros, visualizado y gestión de una escena 2D en la cual representaremos fractales 2D en un canvas de WebGL.

Atributos Además de los heredados de `Scene`, contamos con los siguientes atributos.

- **parameters:** `Object`. Objeto (diccionario) de JavaScript con varios campos, estos campos guardan una estrecha relación con las variables `uniform` descritas anteriormente, ya que albergan los valores que posteriormente se enviarán a la variable homónima. Estos campos son:
 - **zoomCenter:** `Array`. Punto en WC que se encuentra en el centro del canvas (x_0, y_0).

- `zoomSize: number`. Valor que representa cuanto zoom hacemos en la escena (λ).
 - `LLC: Array`. Esquina inferior izquierda (*Lower Left Corner*) de la región del plano que se está representando en el canvas.
 - `URC: Array`. Esquina superior derecha (*Upper Right Corner*) de la región del plano que se está representando en el canvas.
 - `maxIterations: number`. Número máximo de iteraciones que se fijan en los algoritmos para graficar conjuntos de Julia y Mandelbrot.
 - `delta: number`. Incremento que se suma o resta a la hora de desplazarse por la escena.
 - `juliaSetConstant: Array`. Valor complejo c en la ecuación $P_{c,m}(z) = z^m + c$.
 - `order: number`. Exponente m en la ecuación $P_{c,m}(z) = z^m + c$.
 - `fractal: number`. Si este parámetro vale 0 se visualizará el conjunto de Mandelbrot. Si valiera 1 se graficaría el conjunto de Julia asociado al campo `juliaSetConstant`.
 - `antialiasing: boolean`. Si es true se aplicará SSAA a la escena. En caso contrario se identificará cada píxel con un único punto situado en el centro del mismo.
 - `nSamples: number`. Valor entero tal que en caso de que `antialiasing` sea true se utilizarán $nSamples^2$ puntos por píxel para calcular el color final.
- `programInfo: Object`. Objeto que almacena información genérica relativa al programa:
- `program: WebGLProgram`. Programa Shader ya inicializado.
 - `attribLocations: Object`. Localización en memoria de las variables `attribute`.
 - `uniformLocations: Object`. Localización en memoria de las variables `uniform`.
- `initialParameters: Object`. Objeto tipo diccionario que es una copia exacta de `parameters`, salvo que es inmutable, de forma que se permite restaurar los parámetros por defecto en caso de que se deseé.

Métodos

- `constructor`: A partir del código fuente del vertex y el fragment shader, llama al constructor de la clase 'Scene', el cual crea el programa shader y el buffer de posiciones de los vértices que maneja el vertex shader. Además inicializa el atributo `parameters`, con los parámetros iniciales que maneja la escena. Se crea y se le dan valores al atributo `programInfo` y se almacenan en el atributo `initialParameters` los parámetros iniciales de la escena.

Argumentos

- `vsSource: string`. Código fuente del vertex shader.
- `fsSource: string`. Código fuente del fragment shader.

Devuelve: Un objeto de la clase Scene2D con todos sus atributos inicializados.

- **drawScene**: A partir de los parámetros actuales que se encuentren en el atributo **parameters**, las variables **attribute** y **uniform**, se envían estos valores a WebGL y se visualiza la escena.

Argumentos: No acepta, toda la información que necesita está en los atributos de la clase.

Devuelve: No devuelve nada, visualiza la escena en el canvas.

- **zoomIn**: Se reduce el parámetro **zoomSize** para ampliar la escena. También se reduce **delta** para que los desplazamientos sean menos bruscos.
- **zoomOut**: Se aumenta el parámetro **zoomSize** para alejar la escena. También se aumenta **delta** para que los desplazamientos sean más notables.
- **moveLeft**: Se reduce la primera coordenada de **zoomCenter** para desplazar la escena a la izquierda y se modifican adecuadamente los valores de **LLC** y **URC**.
- **moveRight**: Se aumenta la primera coordenada de **zoomCenter** para desplazar la escena a la derecha y se modifican adecuadamente los valores de **LLC** y **URC**.
- **moveUp**: Se aumenta la segunda coordenada de **zoomCenter** para desplazar la escena hacia arriba y se modifican adecuadamente los valores de **LLC** y **URC**.
- **moveDown**: Se reduce la segunda coordenada de **zoomCenter** para desplazar la escena hacia abajo y se modifican adecuadamente los valores de **LLC** y **URC**.
- **changeAntialiasing**: Se cambia el valor del parámetro booleano **antialiasing**.
- **calculateZoom**: Recalcula los parámetros **zoomCenter** y **zoomSize** ante posibles cambios en **LLC** y **URC**.

Getters

- **getAntialiasing**: Getter del parámetro **antialiasing**, de tipo booleano.
- **getDelta**: Getter del parametro **delta**, de tipo number.
- **getFractal**: Getter del parámetro **fractal**, de tipo number.
- **getJuliaConstantX**: Getter de la primera componente del parámetro **juliaSetConstant**, de tipo number.
- **getJuliaConstantY**: Getter de la segunda componente del parámetro **juliaSetConstant**, de tipo number.
- **getLLC**: Getter del parámetro **LLC**, de tipo Array.
- **getMaxIterations**: Getter del parámetro **maxIterations**, de tipo number.
- **getNSamples**: Getter del parámetro **nSamples**, de tipo number.
- **getOrder**: Getter del parámetro **order**, de tipo number.

- **getURC**: Getter del parámetro URC, de tipo Array.

Setters

- **setFractal**: Setter del parámetro **fractal**.
- **setInitialParameters**: Método que restablece los parámetros del objeto **parameters** a los valores iniciales.
- **setJuliaConstantX**: Setter de la primera componente del parámetro **juliaSetConstant**.
- **setJuliaConstantY**: Setter de la segunda componente del parámetro **juliaSetConstant**.
- **setLLCX**: Setter de la primera componente del parámetro **LLC**.

Argumentos:

- **x: number**. Nuevo valor que se le asignará a **LLC[0]**.
 - **URCFixed: boolean**. Simboliza si el usuario ha decidido que la esquina superior derecha quede fija, en cuyo caso también habría que modificar el valor de **LLC[1]** para mantener la proporción 16:9.
- **setLLCY**: Setter de la segunda componente del parámetro **LLC**.

Argumentos:

- **y: number**. Nuevo valor que se le asignará a **LLC[1]**.
 - **URCFixed: boolean**. Simboliza si el usuario ha decidido que la esquina superior derecha quede fija, en cuyo caso también habría que modificar el valor de **LLC[0]** para mantener la proporción 16:9.
- **setMaxIterations**: Setter del parámetro **maxIterations**.
 - **setNSamples**: Setter del parámetro **nSamples**.
 - **setOrder**: Setter del parámetro **order**.
- **setURCX**: Setter de la primera componente del parámetro **URC**.

Argumentos:

- **x: number**. Nuevo valor que se le asignará a **URC[0]**.
 - **LLCFixed: boolean**. Simboliza si el usuario ha decidido que la esquina inferior izquierda quede fija, en cuyo caso también habría que modificar el valor de **URC[1]** para mantener la proporción 16:9.
- **setURCY**: Setter de la segunda componente del parámetro **URC**.

Argumentos:

- **y: number**. Nuevo valor que se le asignará a **URC[1]**.
- **LLCFixed: boolean**. Simboliza si el usuario ha decidido que la esquina inferior izquierda quede fija, en cuyo caso también habría que modificar el valor de **URC[0]** para mantener la proporción 16:9.

A.2.2. Fichero `fractals-2D.js`

Fichero javascript que gestiona los eventos y los indicadores HTML del documento `2D-fractals.html`, entendiendo estos últimos como los pequeños elementos del documento en los cuales escribimos el valor actual de un parámetro (véase el documento `2D-fractals.html` y cualquier etiqueta `input` para una mejor comprensión). En este archivo se declara y utiliza continuamente una variable global de la clase `Scene2D`, ya que es ésta la que lleva toda la interacción entre WebGL y los parámetros modificables. Este fichero, por su parte, sirve como intermediario entre el ‘front-end’ HTML y la escena.

No existen como tal clases, tan sólo métodos que se utilizan como gestores de eventos y una función `main`, la cual se ejecuta al cargar la página y contiene la inicialización de los indicadores HTML, la asignación de los manejadores de eventos y la primera visualización.

Por su parte, las funciones gestoras de eventos verifican y modifican los valores de los elementos y cierto HTML si es necesario, llaman al setter indicado para cambiar el parámetro con el nuevo valor y llaman a `drawScene` para redibujar la escena con los nuevos parámetros. Incluimos debajo un ejemplo en el caso del deslizador que permite cambiar el parámetro `maxIterations`:

```
1 // CAMBIAR NUMERO MAXIMO DE ITERACIONES
2 function changeMaxIterations(event){
3     let new_value = verifyMaxIterations(event.target.value);
4     $("#valorNIteraciones").val(new_value);
5     $("#nIteraciones").val(new_value);
6     theScene.setMaxIterations(new_value);
7     theScene.drawScene();
8 }
9
10 // Asegura que number este entre 10 y 1000, truncando si es
11 // necesario.
12 function verifyMaxIterations(number) {
13     return parseInt(Math.min(Math.max(10, number), 1000));
14 }
```

A.3. Ray-Tracing y fractales 3D

En el caso de escenas tridimensionales, en la memoria se puede apreciar que la complejidad se dispara, aunque realmente el grueso de la dificultad se haya en el fragment shader. En lo relativo a JavaScript es todo más sencillo, con la salvedad de que el número de parámetros sí que es mucho mayor en estos casos, pero la filosofía es exactamente la misma. Tenemos una clase `Scene3D` que gestiona el paso de parámetros dinámicamente a WebGL, los visualizados y los cambios en estos valores. A su vez también contamos con un fichero `fractals-3D.js` cuya función es bien parecida a la de `fractals-2D.js` pero con su contexto: la gestión de eventos y la correcta modificación de indicadores HTML.

Cuando visualizamos el espacio tridimensional usando técnicas como ray-tracing el fragment shader es mucho más complejo, aunque recordemos que en este momento lo único que nos interesa son las variables `uniform` y su identificación con los parámetros de la escena. Veámos

las variables que tenemos en el archivo `fragment-shader-3D-fractals.js`, el cual contiene el código fuente del fragment shader utilizado y además es realmente una extensión de `fragment-shader-ray-tracer.js`.

```

1 uniform vec3 u_lookfrom;
2 uniform vec3 u_lookat;
3
4 uniform vec4 u_ka;
5 uniform vec4 u_kd;
6 uniform vec4 u_ks;
7 uniform float u_sh;
8
9 uniform vec4 u_lightColor0;
10 uniform vec4 u_lightColor1;
11 uniform bvec3 u_shadows;
12
13 uniform float u_epsilon;
14
15 uniform int u_fractal;
16 uniform vec4 u_juliaSetConstant;
17 uniform bool u_antialiasing;
18 uniform int u_nSamples;
```

Como en la sección anterior, explicaremos el significado de cada una de ellas y su relación con los parámetros de la escena. Algunas de ellas, como `u_fractal` o `u_juliaSetConstant` tienen significados muy similares a los que tenían sus análogas en dos dimensiones, veámoslo.

A.3.1. Fichero `scene3D.js`

Contiene el código respectivo a la clase `Scene3D`, que representa una escena en la que podemos visualizar fractales en 3D.

Clase `Scene3D`

Es una clase que hereda de `Scene` y que contiene el código relativo a la creación, parámetros, visualizado y gestión de una escena 3D en la cual representaremos fractales 3D en un canvas de WebGL.

En esta clase se gestiona además una cámara orbital. Es decir, nos permite movernos alrededor de un punto fijo, que es el origen $(0, 0, 0)$, acercar y alejar nuestra posición. Para esto, se mantiene un parámetro que son las coordenadas esféricas del punto $lookfrom = (\rho, \phi, \theta)$, ya que con editar estos tres parámetros estaríamos moviéndonos hacia adelante o atrás, derecha o izquierda y arriba o abajo en torno al punto $(0, 0, 0)$.

También se cuenta con parámetros para gestionar el realismo de la escena mediante el parámetro ε que usamos en ray-marching; un material cuyas propiedades son totalmente editables; dos fuentes de luz situadas en las direcciones $(-0.5, 0.5, 0.0)$ (luz 0) y $(0.5, 0.5, 0.0)$ (luz 1) a las cuales se les puede cambiar el color y si deben o no arrojar sombras y la posibilidad de lanzar varios rayos por píxel. Aclaramos que todas estas mejoras dan unos resultados muy buenos pero también son computacionalmente muy costosas, por lo que puede ralentizarse mucho el rendimiento.

Dicho esto, presentamos los atributos que nos ayudarán a manejar toda esta funcionalidad.

Atributos Además de los heredados de **Scene**, contamos con los siguientes atributos.

- **parameters: Object**. Objeto (diccionario) de JavaScript con varios campos, estos campos guardan una estrecha relación con las variables **uniform** descritas anteriormente, ya que albergan los valores que posteriormente se enviarán a la variable homónima. Estos campos son:
 - **lookfrom: Array**. Punto en el cual se sitúa el espectador en coordenadas cartesianas de mundo.
 - **lookfromSpheric: Array**. Coordenadas esféricas del punto **lookfrom**.
 - **lookat: Array**. Punto hacia el que mira el espectador en coordenadas de mundo.
 - **ka: Array**. Tupla RGBA que representa la reflectividad ambiental del material parametrizable con el que se graficará el objeto central de la escena.
 - **kd: Array**. Tupla RGBA que representa la reflectividad difusa del material parametrizable con el que se graficará el objeto central de la escena.
 - **ks: Array**. Tupla RGBA que representa la reflectividad especular del material parametrizable con el que se graficará el objeto central de la escena.
 - **sh: number**. Exponente de brillo especular del material parametrizable.
 - **lightColor0: Array**. Tupla RGBA que será la intensidad de la luz derecha.
 - **lightColor1: Array**. Tupla RGBA que será la intensidad de la luz izquierda.
 - **shadows: Array**. Tripleta booleana tal que si **shadows[i]** es true la fuente de luz i-esima arrojará sombras.
 - **fractal: number**. Si este parámetro vale 1 se visualizará el conjunto de Mandelbulb. Si vale 2 se visualizará el conjunto de Julia asociado al campo **juliaSetConstant**. Si vale 3 se visualizará el conjunto de Mandelbrot 3D.
 - **juliaSetConstant: Array**. Cuaternio c en la ecuación $P_c(q) = q^2 + c$.
 - **epsilon: number**. Distancia mínima ε utilizada en sphere-tracing para decidir cuándo un punto está en la frontera del fractal.
 - **antialiasing: boolean**. Si es true se aplicará antialiasing a la escena. En caso contrario se trazarán un único rayo por píxel.
 - **nSamples: number**. Valor entero que en caso de que **antialiasing** sea true se lanzarán **nSamples**² rayos por píxel.
 - **delta: number**. Incremento que se suma o se resta a la hora de desplazarse por la escena.
- **programInfo: Object**. Objeto que almacena información genérica relativa al programa:
 - **program: WebGLProgram**. Programa Shader ya inicializado.
 - **attribLocations: Object**. Localización en memoria de las variables **attribute**.
 - **uniformLocations: Object**. Localización en memoria de las variables **uniform**.

Hay que aclarar que este objeto no es el mismo que el de `Scene2D`, pues aunque la estructura sea la misma el objeto `Shader` y las variables `uniform` (y por tanto sus localizaciones) no son las mismas.

- `initialParameters: Object`. Objeto tipo diccionario que es una copia exacta de `parameters`, salvo que es inmutable, de forma que se permite restaurar los parámetros por defecto en caso de que se deseé.

Métodos

- `constructor`: A partir del código fuente del vertex y el fragment shader, llama al constructor de la clase `Scene`, el cual crea el programa shader y el buffer de posiciones de los vértices que maneja el vertex shader. Además inicializa el atributo `parameters`, con los parámetros iniciales que maneja la escena. Se crea y se le dan valores al atributo `programInfo` y se almacenan en el atributo `initialParameters` los parámetros iniciales de la escena.

Argumentos

- `vsSource: string`. Código fuente del vertex shader.
- `fsSource: string`. Código fuente del fragment shader.

Devuelve: Un objeto de la clase `Scene3D` con todos sus atributos inicializados.

- `drawScene`: A partir de los parámetros actuales que se encuentren en el atributo `parameters`, las variables `attribute` y `uniform`, se envían estos valores a WebGL y se visualiza la escena.

Argumentos: No acepta, toda la información que necesita está en los atributos de la clase.

Devuelve: No devuelve nada, visualiza la escena en el canvas.

- `zoomIn`: Se reduce el modulo del vector que une el origen con el punto `lookfrom` para acercar la escena.
- `zoomOut`: Se aumenta el modulo del vector que une el origen con el punto `lookfrom` para alejar la escena.
- `moveLeft`: Se modifica el angulo ϕ de las coordenadas esféricas del punto `lookfrom` para desplazar la escena a la izquierda.
- `moveRight`: Se modifica el angulo ϕ de las coordenadas esféricas del punto `lookfrom` para desplazar la escena a la derecha.
- `moveX`: Dada una cantidad ‘`desp`’, desplaza la cámara a la derecha o a la izquierda `desp` unidades.

Argumentos:

- `desp: number`. Número entero (positivo o negativo) que indica el desplazamiento.

- **moveUp**: Se modifica el angulo θ de las coordenadas esféricas del punto `lookfrom` para desplazar la escena hacia arriba.
- **moveDown**: Se modifica el angulo θ de las coordenadas esféricas del punto `lookfrom` para desplazar la escena hacia abajo.
- **moveY**: Dada una cantidad ‘`desp`’, desplaza la cámara hacia arriba o hacia abajo `desp` unidades.

Argumentos:

- `desp: number`. Número entero (positivo o negativo) que indica el desplazamiento.
- **changeShadow**: Si la componente i -ésima del parámetro `shadow` esta a false la pone a true y viceversa. Es una forma de activar o desactivar las sombras arrojadas provocadas por una fuente de luz concreta.

Argumentos:

- `i: number`: Índice de la luz cuyas sombras queremos activar o desactivar.
- **changeAntialiasing**: Cambia el valor booleano del parámetro `antialiasing`. Es una forma de decirle al programa si queremos que se aplique o no antialiasing a los píxeles.
- **#rescaleAngles**: Cuando se modifican los ángulos de las coordenadas esféricas es posible que se salgan del rango $[0, 2\pi]$. Esta función ajusta esos valores para que siempre se sitúen en el intervalo correcto.

Getters

- **getAntialiasing**: Getter del parámetro `antialiasing`, de tipo booleano.
- **getEpsilon**: Getter del parámetro `epsilon`, de tipo number.
- **getFractal**: Getter del parámetro `fractal`, de tipo number.
- **getJuliaConstant**: Getter del parámetro `juliaSetConstant`, de tipo Array (cuaternio).
- **getKa**: Getter del parámetro `ka`, de tipo array (terna RGBA).
- **getKd**: Getter del parámetro `kd`, de tipo array (terna RGBA).
- **getKs**: Getter del parámetro `ks`, de tipo array (terna RGBA).
- **getLightColor**: Getter del parámetro `lightColorX` donde `X` puede ser 0 o 1 según el valor del argumento.

Argumentos:

- `i: number`. Si vale 0 se devuelve el parámetro `lightColor0`, en cualquier otro caso se devuelve `lightColor1`.
- **getNSamples**: Getter del parámetro `nSamples`, de tipo number.
- **getPosition**: Getter del parámetro `lookfrom`, la posición del observador.

- **getSh**: Getter del parámetro **sh**, tipo number.

Setters

- **setEpsilon**: Setter del parámetro **epsilon**.
- **setFractal**: Setter del parámetro **fractal**.
- **setInitialParameters**: Método que restablece los parámetros a los valores por defecto.
- **setJuliaConstant**: Setter del parámetro **juliaSetConstant**.
- **setKa**: Setter del parámetro **ka** (reflectividad ambiental).
- **setKd**: Setter del parámetro **kd** (reflectividad difusa).
- **setKs**: Setter del parámetro **ks** (reflectividad especular).
- **setLightColor**: Setter del parámetro **lightColorX**, donde **X** depende del valor del argumento **i**.

Argumentos:

- **i: number**. Si vale 0, se modifica la terna RGB de la luz 0, en otro caso se modifica la terna de la luz 1.
- **setNSamples**: Setter del parámetro **nSamples**.
- **setPosition**: Setter del parámetro **lookfrom**, la posición del observador.
- **setSh**: Setter del parámetro **sh** (exponente de brillo).

A.3.2. Fichero **fractals-3D.js**

Fichero javascript que gestiona los eventos y los indicadores HTML del documento **3D-fractals.html**. La metodología utilizada en este fichero es la misma que la empleada en el caso 2-dimensional, solo que al haber más parámetros modificables, también hay más elementos HTML, y por tanto más eventos que gestionar y más indicadores que editar. Esta vez utilizamos una variable global de la clase **Scene3D** para gestionar toda la interacción entre WebGL y los parámetros modificables.

Tampoco existen clases, únicamente funciones manejadoras de eventos y la función **main** que se ejecuta al cargar la página y contiene la inicialización de los elementos e indicadores HTML, la asignación de los manejadores de eventos y la primera visualización.

Para no alargar innecesariamente este apéndice y para que el propio lector observe la metodología por sí mismo, lo invitamos a visitar el fichero completo en <https://github.com/JAntonioVR/Geometria-Fractal/blob/main/static/js/fractals-3D.js>.

A.4. Diagrama de clases UML

En la imagen A.1 podemos ver un diagrama de clases del código recién explicado. Si se desea ver en mayor detalle en [este enlace](#) puede encontrar el archivo en formato ‘png’.

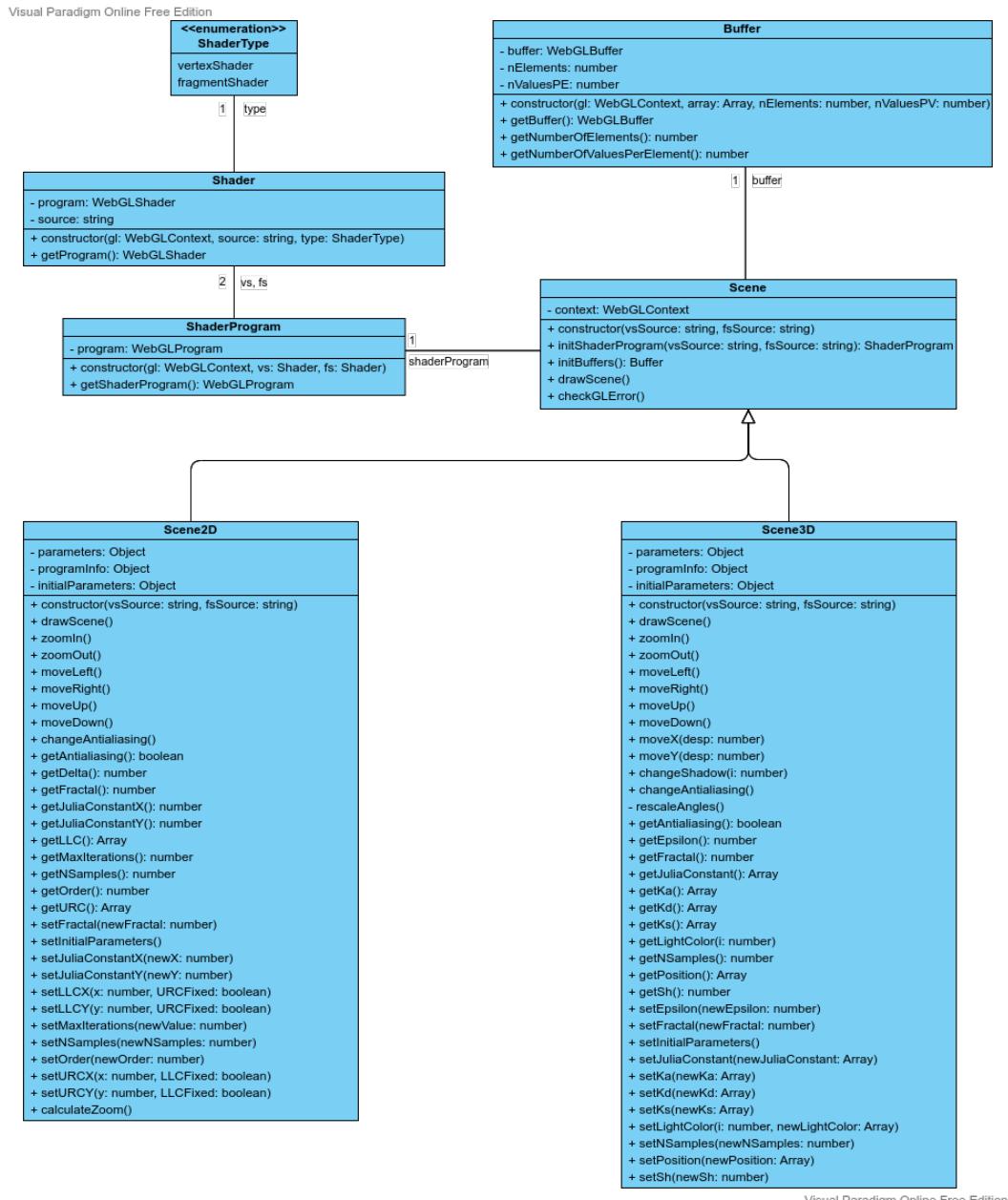


Imagen A.1: Diagrama de clases JavaScript utilizadas

APÉNDICE B

INSTALACIÓN DEL PRODUCTO SOFTWARE

Como se ha dicho en otras secciones de esta memoria, la web interactiva, que es el producto software desarrollado en este proyecto, puede consultarse y utilizarse desde cualquier navegador desde la dirección <https://jantoniovrv.github.io/Geometria-Fractal/>.

No obstante, por cuestiones de eficiencia recomendamos ejecutar el proyecto en local. Explicamos el procedimiento en los siguientes párrafos.

1. Primero de todo necesitamos descargar el código del repositorio del proyecto en Github, el cual se puede encontrar en <https://github.com/JAntonioVR/Geometria-Fractal>. Para ello, mediante una terminal nos situamos en el directorio que deseemos y clonamos el repositorio:

```
1 | git clone git@github.com:JAntonioVR/Geometria-Fractal.git  
2 | cd Geometria-Fractal
```

Si no se dispone de `git`, en la pestaña `Code` se puede encontrar la posibilidad de descargar un ZIP con el código.

2. Es necesario lanzar un servidor web local. Para esto, se ofrecen algunas de las alternativas más sencillas. Si se dispone de `Python`, bastaría con ejecutar el comando:

```
1 | python SimpleHTTPServer 8000 # Si se dispone de Python2
```

o

```
1 | python http.server 8000 # Si se dispone de Python3 (recomendado)
```

Tras esto, abra su navegador en la URL `http://localhost:8000/` y encontrará la portada de la web.

Otra alternativa para los usuarios del editor `Visual Studio Code` (VSCode) es la extensión `Live Server`, la cual con tan solo instalarla y pulsar el botón ‘Go Live’ que aparecerá en la esquina inferior derecha lanza un servidor web en el directorio que se tenga abierto en el momento y abre una ventana en el navegador.

Por último, en [este enlace](#) se explica la posibilidad de crear un servidor web utilizando `node.js`.

Nota: Con Python se utiliza el puerto 8000, con ‘Live Server’ el 5500 y con node.js el 8080. Sin embargo, es posible que por alguna razón esos puertos estén ocupados por otros procesos y aparezca algún error. Si esto ocurre simplemente utilice algún puerto distinto que esté disponible.

3. Independientemente de la opción que hayamos elegido para lanzar el servidor web, tan solo resta abrir el navegador que queramos e introducir la dirección `http://localhost:PPPP/`, siendo PPPP el puerto que hayamos elegido finalmente.