

Doble Grado en Ingeniería Informática y Matemáticas
Curso 2021/2022
Trabajo de Fin de Grado

Fractales y Geometría Fractal

Fractales, geometría fractal y aplicaciones en la ciencia. Visualización de fractales con Ray-Tracing

Autor: Juan Antonio Villegas Recio

Autor: Juan Antonio Villegas Recio
Tutor de Matemáticas: Manuel Ruiz Galán, Catedrático de Universidad
Departamento de Matemática Aplicada, Universidad de Granada
Tutor de Informática: Carlos Ureña Almagro, Profesor Titular de Universidad
Departamento de Lenguajes y Sistemas Informáticos, Universidad de Granada

 AUTORÍA

I hereby affirm that this Master thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text. This work has not been submitted for any other degree or professional qualification except as specified; nor has it been published.

City, date



Juan Antonio Villegas Recio



ABSTRACT

(no more than 250-300 words)

Background

Describe background shortly

Aim

Describe the aim of your study

Method

Describe your methods

Results

Describe the main results of your study

Conclusion

State your conclusion

Keywords:

No more than six keywords, preferably MeSH terms

ÍNDICE GENERAL

Lista de Abreviaturas	II
Lista de Imágenes	IV
Lista de Tablas	V
1. El concepto de <i>fractal</i>	2
1.1. Ejemplos clásicos	3
1.1.1. El conjunto de Cantor	3
1.1.2. El triángulo de Sierpinski	4
1.1.3. La alfombra de Sierpinski y la esponja de Menger	5
1.1.4. La curva de Koch	5
1.1.5. El copo de nieve de Koch	6
1.2. Conceptos de dimensión fractal	7
1.2.1. Dimensión por cajas	8
1.2.2. Medida y dimensión de Hausdorff	11
1.2.3. Dimensión topológica	12
1.2.4. Relación entre los distintos tipos de dimensión fractal	13
2. Iteración	15
2.1. Iteración de funciones	15
2.1.1. Convergencia a un punto fijo	16
2.1.2. Velocidad de convergencia	17
2.2. El método de Newton y cuencas de atracción	18
2.2.1. Autosimilaridad	21
3. Conjuntos de Julia y Mandelbrot	23
3.1. Iteración convergente y no convergente	24
3.2. Conjuntos de Julia	25
3.2.1. Representación gráfica de los conjuntos de Julia	25
3.3. Distinción entre conjuntos de Julia conexos y polvaredas	27
3.4. El conjunto de Mandelbrot	28
3.4.1. Representación gráfica del conjunto de Mandelbrot	28
3.5. Autosimilaridad de los conjuntos de Julia y Mandelbrot	30
3.5.1. Autosimilaridad en conjuntos de Julia	30
3.5.2. Autosimilaridad en el conjunto de Mandelbrot	31

3.6. Conjuntos de Julia y Mandelbrot generalizados	31
3.6.1. Familia $\{z^N + c\}_{c \in \mathbb{C}}$	31
3.6.2. Conjuntos de Julia con funciones no polinómicas	33
4. Sistemas de Funciones Iteradas	36
4.1. Transformaciones afines en el plano euclídeo y SFI	36
4.2. Convergencia de SFI	39
4.2.1. El Espacio de Fractales y la Métrica de Hausdorff	39
4.2.2. Aplicaciones contractivas en el espacio de fractales	41
4.2.3. El espacio de fractales y los SFI	42
4.3. SFI y conjuntos autosimilares	44
4.4. El problema inverso	45
5. Introducción a las herramientas de renderización	49
5.1. Componentes de WebGL	50
5.1.1. Contexto de WebGL	50
5.1.2. El programa <i>Shader</i>	50
5.1.3. Los <i>Buffer</i>	52
5.2. Primera imagen renderizada	53
6. Visualización de fractales en 2D	56
6.1. Objetivo	56
6.2. Estructurando el código	57
6.3. El fragment shader	58
6.3.1. La función $f(z) = z^N + c$	59
6.3.2. Asignación de colores	60
6.3.3. Renderizando conjuntos de Julia	62
6.3.4. Renderizando conjuntos de Mandelbrot	62
6.3.5. Alternando conjuntos de Julia y Mandelbrot	64
7. Introducción al <i>Ray-Tracing</i>	65
7.1. Definición de Ray-Tracing	65
7.2. Creación del rayo	67
7.3. El background	70
7.4. Visualizando una escena sencilla	71
7.4.1. Renderizando una esfera	71
7.4.2. Renderizando varias esferas	74
7.4.3. Renderizando un plano con textura de ajedrez	75
7.5. Configurando la cámara	78
7.6. Modelo de iluminación de Phong	83
7.6.1. Componente ambiental	84
7.6.2. Componente difusa	84
7.6.3. Componente especular	85
7.6.4. Evaluación del modelo de iluminación	86
7.6.5. Implementación del modelo de Phong	88
8. Visualización de fractales en 3D	92
8.1. El algoritmo Ray-Marching	92
8.1.1. Implementación de Ray-Marching en GLSL	95
8.1.2. Comentarios sobre Ray-Marching	97
8.2. Signed Distance Functions (SDFs)	99

8.3.	Visualización tridimensional de conjuntos de Julia	100
8.3.1.	Aproximando la normal	103
	Método 1: Gradiente de la SDF	103
	Método 2: Técnica del tetraedro	103
8.3.2.	Implementación en GLSL	104
8.4.	Visualización tridimensional del conjunto de Mandelbrot	107
8.5.	El conjunto de Mandelbub	108
8.6.	Efectos realistas en la escena	112
8.6.1.	Sombras arrojadas	112
8.6.2.	Antialiasing	115
8.7.	Posibles optimizaciones	117
8.7.1.	Esferas englobantes	118
8.7.2.	Limitación de las sombras	119
Appendices		124
A. Documentación del código JavaScript		124
B. Another Appendix		125

LISTA DE ABREVIATURAS

- API: Application Programming Interface (Interfaz de Programación de Aplicaciones)
- CPU: Central Processing Unit (Unidad Central de Procesamiento)
- GPU: Graphics Processing Unit (Unidad de Procesamiento Gráfico)
- SDF: Signed Distance Function
- SFI: Sistema de Funciones Iteradas
- RT: Ray-Tracing

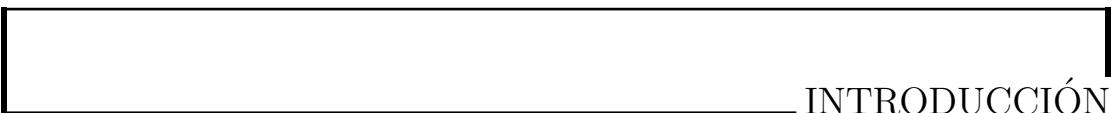
LISTA DE IMÁGENES

1.1.	Objetos de la naturaleza con estructura fractal	2
1.2.	Iteraciones del proceso de generación del conjunto de Cantor	4
1.3.	Generación del triángulo de Sierpinski	5
1.4.	Generación de la alfombra de Sierpinski	5
1.5.	Esponja de Menger	6
1.6.	Iteraciones del proceso de generación de la curva de Koch	6
1.7.	Generación del copo de nieve de Koch	7
1.8.	Curvas de Koch que componen el copo de Koch	7
1.9.	Possible movimiento de un punto en posibles objetos de \mathbb{R}^n	8
1.10.	Segmento, cuadrado y cubo recubiertos por objetos de lado $\frac{1}{2}$	9
1.11.	Una forma de calcular la dimensión por cajas de la curva de Koch .	10
1.12.	Figuras representativas de los ejemplos	13
2.1.	Representación de dos órbitas en \mathbb{C}	16
2.2.	Cuencas de atracción de $f(z) = z^2 - 1$	19
2.3.	Cuencas de atracción de distintas funciones coloreadas.	20
2.4.	Evaluación de la velocidad de convergencia en cada punto	21
2.5.	Cuencas de atracción de $f(z) = z^3 - 1$	21
2.6.	Diferentes regiones ampliadas de la figura 2.5	22
3.1.	Primeras imágenes de conjuntos de Julia	23
3.2.	Primeras imágenes del conjunto de Mandelbrot	24
3.3.	Conjuntos de Julia graficados con Mathematica	27
3.4.	Resultados de la orden ‘JuliaSetPlot’	27
3.5.	Representación de \mathcal{M} y detalle en $[-0.65, -0.4] \times [0.47, 0.72]$	30
3.6.	Diferentes regiones ampliadas de la figura 3.3 (b)	31
3.7.	Detalles autosimilares de algunos conjuntos de Julia	32
3.8.	Detalles autosimilares de \mathcal{M}	33
3.9.	Ampliaciones del bulbo principal de \mathcal{M}	34
4.1.	Ejemplos de aplicaciones de transformaciones lineales	38
4.2.	Representación gráfica de T y $w(T)$	39
4.3.	Otra posible semilla para iterar el SFI	39
4.4.	Resultado de iterar 8 veces w con distintas figuras iniciales	40
4.5.	Contraejemplo a la distancia entre conjuntos	40
4.6.	Atractores de los SFI definidos en las tablas 4.1 y 4.2.	44
4.7.	Imagen cuyo SFI debemos determinar	47

5.1.	<i>Clip space de WebGL</i>	51
5.2.	Cuadrado de colores renderizado con WebGL	54
5.3.	Componentes de WebGL e interacciones entre ellos	55
6.1.	Paleta de colores elegida para la visualización de fractales 2D	61
6.2.	Gradiente generado por la paleta de colores 6.1	61
6.3.	Renderizado de algunos conjuntos de Julia con WebGL	63
6.4.	Renderizado de algunos conjuntos de Mandelbrot con WebGL	64
7.1.	Esquema básico del funcionamiento del Ray-Tracing	66
7.2.	Elementos que participan el RT	68
7.3.	Gradiente utilizado para el fondo de la escena	70
7.4.	Primera escena vacía renderizada	71
7.5.	Escena con una esfera	74
7.6.	Escena con varias esferas	76
7.7.	Textura de ajedrez a partir de las partes enteras	77
7.8.	Escena compuesta por esferas y un plano	79
7.9.	Field Of View	80
7.10.	Vectores que forman una base del plano de proyección	80
7.11.	Representación gráfica de los campos de ‘Camera’	81
7.12.	Escena 7.8 desde otros puntos de vista	83
7.13.	La calima: un ejemplo del efecto de la componente ambiental	85
7.14.	Esquema de los vectores utilizados en la componente difusa	85
7.15.	Esquema de los vectores utilizados en la componente especular	86
7.16.	Punto donde no se aplica componente difusa ni especular	87
7.17.	Componentes aisladas del modelo de Phong	87
7.18.	Acción de todas las componentes	87
7.19.	Imagen final del capítulo 7	91
8.1.	Situación a la que aplicar Ray-Marching	93
8.2.	Dos primeras iteraciones de Ray Marching	93
8.3.	Posibles estados finales del algoritmo Ray-Marching	94
8.4.	Escena tras implementar Ray-Marching	97
8.5.	Escena utilizando vectores no unitarios en los rayos	98
8.6.	Escena renderizada utilizando $\varepsilon = 0.1$	98
8.7.	Escena renderizada utilizando MAX_STEPS=100	99
8.8.	Ejemplos de puntos con distintos valores de la SDF de una esfera	100
8.9.	Representación de una montaña con curvas de nivel	102
8.10.	Conjuntos de Julia 3D para distintos $c \in \mathbb{H}$	107
8.11.	Detalles del conjunto de Mandelbrot generalizado	109
8.12.	Conjunto de Mandelbulb	111
8.13.	Primera forma de representar sombras	113
8.14.	Sombra arrojada con suavizado	114
8.15.	Sombras con el parámetro $k = 2, 32$	114
8.16.	Sombras arrojadas por varias fuentes de luz	115
8.17.	Suelo antes de aplicar antialiasing	115
8.18.	Representación de los rayos que se lanzan a una pantalla	116
8.19.	Suelo tras de aplicar antialiasing	117
8.20.	Conjunto de Julia aplicando antialiasing	118

ÍNDICE DE TABLAS

4.1.	SFI para el árbol pitagórico	44
4.2.	SFI para el helecho de Barnsley	44
4.3.	SFI para la curva de Koch	45
4.4.	SFI para la imagen 4.7	48



INTRODUCCIÓN

TODO

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? ¿Por qué es necesaria una geometría 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 trozos que recuerdan el objeto original, mientras que estos pequeños trozos 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», evidenciando el movimiento rígido y queriendo en ambos casos referirnos a este concepto.

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 segmento de línea recta aplicando el siguiente proceso iterativo:

1. Partimos del segmento de recta compuesto por el intervalo cerrado $[0, 1]$, aunque realmente es indiferente qué intervalo se escoja, pues el resultado final será el mismo salvo homotecia. Dividimos dicho segmento en tres segmentos iguales y extraemos el intervalo central, manteniendo los extremos. Es decir, extraemos el intervalo abierto $(\frac{1}{3}, \frac{2}{3})$ y mantenemos el segmento $[0, \frac{1}{3}]$ y el $[\frac{2}{3}, 1]$. Nótese que obtenemos $2 = 2^1$ segmentos, cada uno a escala $\frac{1}{3} = (\frac{1}{3})^1$ del original.
2. Aplicamos el mismo proceso a los segmentos $[0, \frac{1}{3}]$ 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 $4 = 2^2$ segmentos iguales, cada uno de ellos a escala $\frac{1}{3}$ 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 obtendrían 2^n segmentos de recta a escala $(\frac{1}{3})^n$. Denotemos como C_n al conjunto unión de los 2^n segmentos de recta 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 \mathbf{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

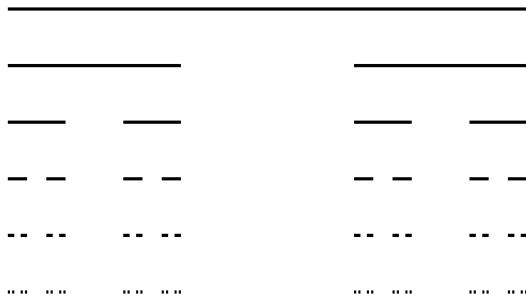


Imagen 1.2: Iteraciones del proceso de generación del conjunto de Cantor

sabemos gracias al *teorema 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 segmento de longitud $\frac{1}{3}$, en la segunda iteración se eliminan 2 segmentos de longitud $(\frac{1}{3})^2$ y en la n -ésima iteración extraemos 2^{n-1} segmentos de longitud $(\frac{1}{3})^n$. Si sumamos las longitudes de todos los segmentos que son eliminados en cada paso se obtiene:

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

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 que la longitud eliminada es igual a la longitud original, es decir, la longitud de \mathbf{C} es nula y aún así tenemos 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*.

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 $\frac{1}{2} = (\frac{1}{2})^1$ 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 $\frac{1}{4} = (\frac{1}{2})^2$ 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 $(\frac{1}{2})^n$ del original.



Imagen 1.3: Generación del triángulo de Sierpinski

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

Si llamamos A al área del triángulo inicial, 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:

$$A \sum_{n=1}^{\infty} 3^{n-1} \left(\frac{1}{4}\right)^n = \frac{A}{4} \sum_{n=0}^{\infty} 3^n \left(\frac{1}{4}\right)^n = \frac{A}{4} \sum_{n=0}^{\infty} \left(\frac{3}{4}\right)^n = \frac{A}{4} \left(\frac{1}{1 - \frac{3}{4}}\right) = A,$$

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 es 0 y seguimos teniendo 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 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).

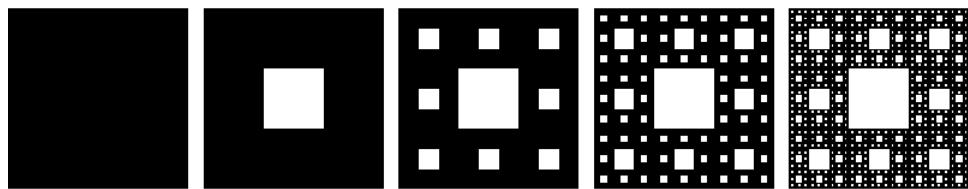


Imagen 1.4: Generación de la alfombra de Sierpinski

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.

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):

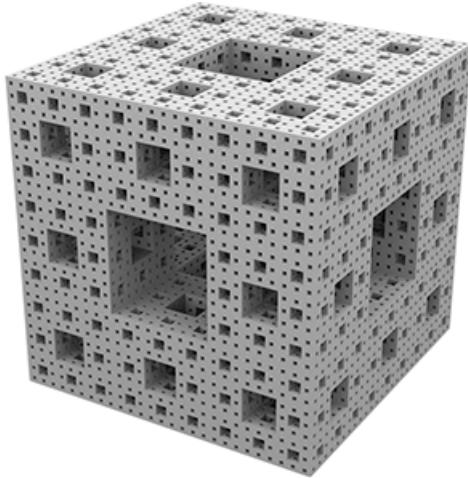


Imagen 1.5: Esponja de Menger

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**.

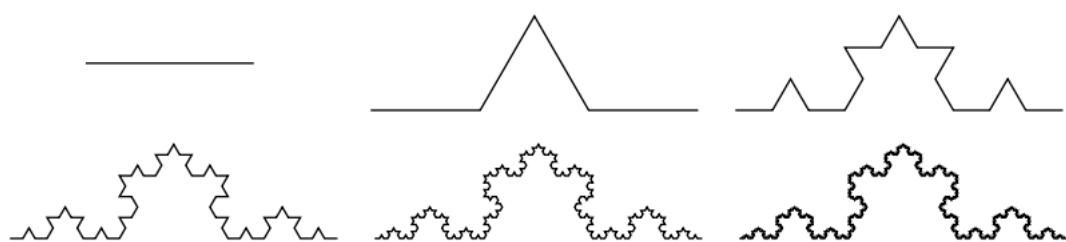


Imagen 1.6: Iteraciones del proceso de generación de la curva de Koch

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.

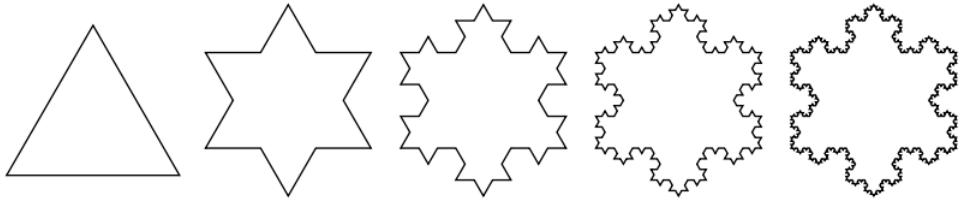


Imagen 1.7: Generación del copo de nieve de Koch

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.

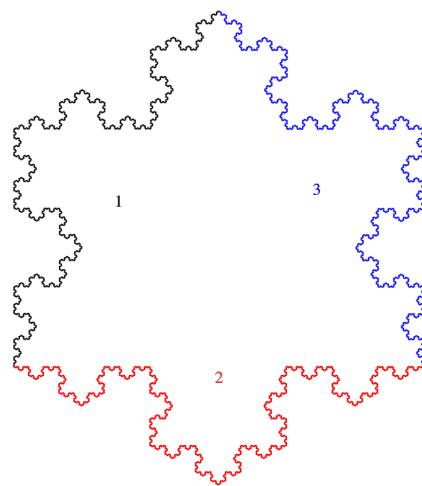


Imagen 1.8: Curvas de Koch que componen el copo de Koch

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 clara 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 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 $a_1, \dots, a_n \in K$ siendo 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)).

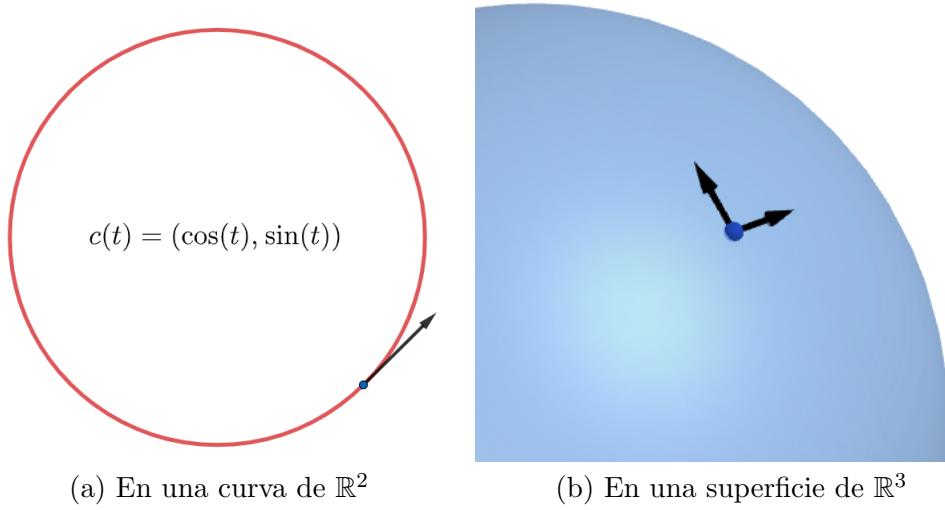


Imagen 1.9: Posible movimiento de un punto en posibles objetos de \mathbb{R}^n

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 S tenga dimensión 1 por la movilidad, pero tampoco dimensión 2 porque tiene área 0, pero 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 $\frac{1}{2}$.

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:

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

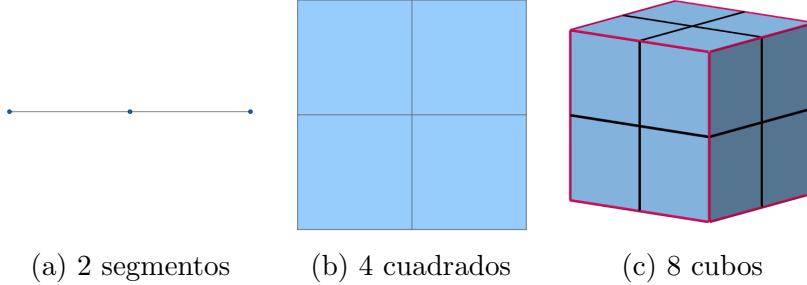


Imagen 1.10: Segmento, cuadrado y cubo recubiertos por objetos de lado $\frac{1}{2}$

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. Dado 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

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

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 tomamos en \mathbb{R}^n una cuadrícula de lado δ , i.e., 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 ‘cajas’, de diámetro $\delta\sqrt{n}$ y contando el número de cajas que recubren a A , obtenemos el mismo resultado, puede comprobarse en [4, sección 3.1].

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

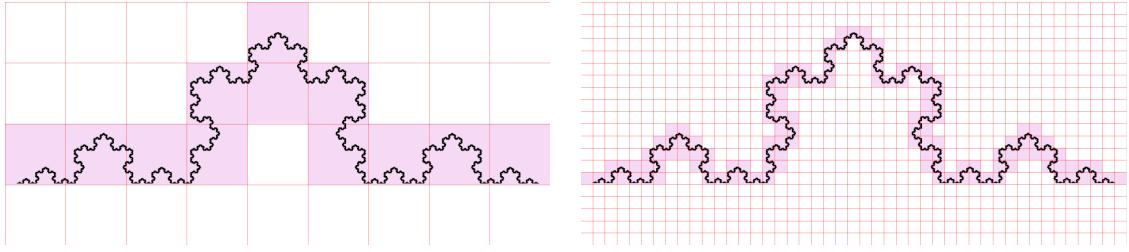
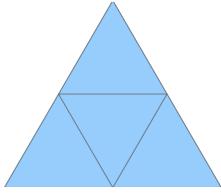


Imagen 1.11: Una forma de calcular la dimensión por cajas de la curva de Koch



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

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

Concluimos entonces que la dimensión por cajas de un triángulo es 2, lo cual tampoco nos sorprende.

De manera similar podemos ya calcular 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. Por tanto, la dimensión por cajas del triángulo de Sierpinski es

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

Efectivamente y tal y como discutimos en el inicio de esta sección, es un valor situado entre 1 y 2.

Por su parte, el conjunto de Cantor **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 **C** con $N_\delta(\mathbf{C}) = 2^n$ segmentos de recta (véase sección 1.1.1). Por lo que podemos calcular la dimensión por cajas del conjunto de Cantor como

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

que es un valor situado entre 0 y 1.

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 [5].

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 posiblemente el infinito como posible valor del límite. Definimos así

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

En [4, 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

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

□

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 $s = \dim_H(A)$ satisface que:*

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

Demostración. Si tomamos $0 < \varepsilon < 1$ y $t > s$ dos valores reales, por la proposición 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 $\mathcal{H}^t(A) = 0$. 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 $\dim_T(X) \leq n - 1$

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 $p \in U \subseteq V$. Por tanto concluimos que X tiene dimensión 0.

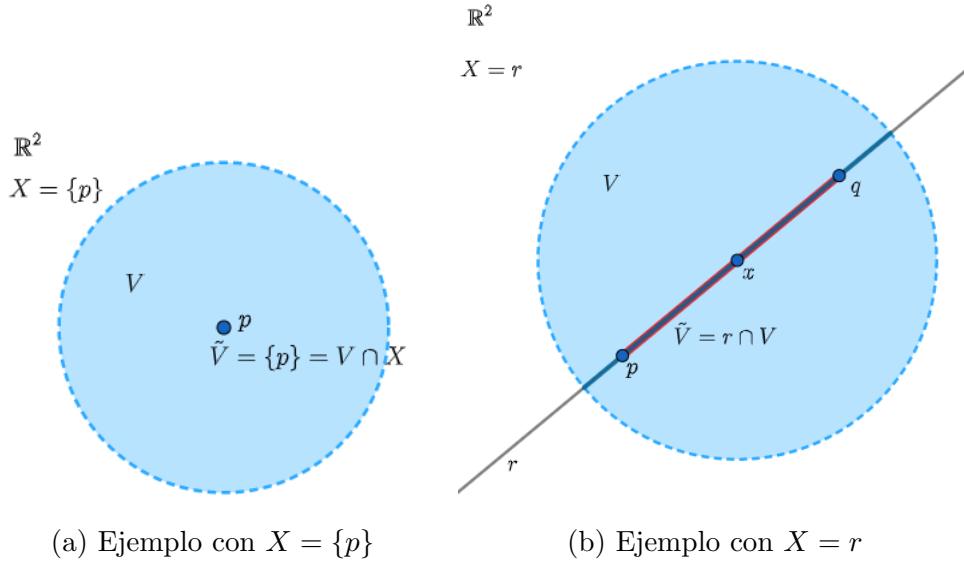


Imagen 1.12: Figuras representativas de los ejemplos

Ejemplo 1.2.2. (Véase imagen 1.12 (b)) En \mathbb{R}^2 dotado de la topología usual, consideramos una recta cualquiera, llámémosla 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 [4, Sección 3.1], ocurre que la dimensión por cajas acota superiormente a la dimensión de Hausdorff, esto es, 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

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

Este resultado fue originalmente probado por *Edward Szpilrajn*, véase [6, 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 conjuntos totalmente autosimilares: el teorema de Moran (véase teorema 4.3.1).

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

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 [3].

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

¹Los códigos completos que generan cada una de las imágenes que se observan en este documento se pueden encontrar en <https://github.com/JAntonioVR/Geometria-Fractal/tree/main/Iteracion-y-Fractales>

```
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 cual denotamos con $\{f^n(0.9)\} \rightarrow 0$. 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 corresponde con el par $(x, y) \in \mathbb{R}^2$ podemos representar en el plano y de forma visual la tendencia de dichas sucesiones. Observemos los ejemplos de las imágenes 2.1.

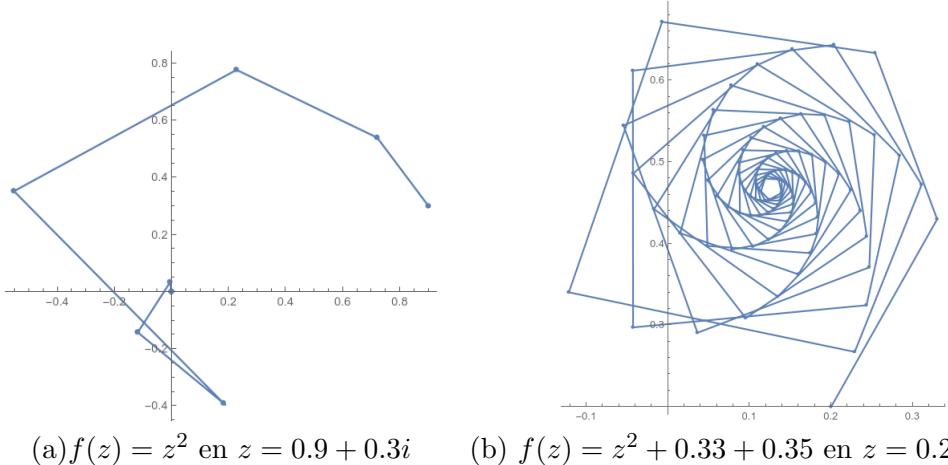


Imagen 2.1: Representación de dos órbitas en \mathbb{C}

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

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

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 $\{x_{n+1}\} = \{f(x_n)\} = \{f^n(x_0)\} \rightarrow x \in X$. Pero f es continua, por lo que si $\{x_n\} \rightarrow x$, 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

Si recordamos el teorema del punto fijo de Banach (teorema 2.1.1), que nos decía 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 alcance 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 métodos numéricos 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 adivinar con cierta precisión donde se anula una función derivable (al menos en un entorno de la raíz), pudiendo así utilizarlo como herramienta para la resolución 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 [12]).

²Más información en la documentación oficial <https://reference.wolfram.com/language/ref/FixedPointList.html>

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 [10, Capítulo 7] y [11, 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. 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[iteracionNR[f, #] &, z, 50]]];
DensityPlot[newtonArgumento[x + I*y], {x, -2, 2}, {y, -2, 2},
PlotPoints -> 100, Mesh -> False, ColorFunction -> "Rainbow"]
```

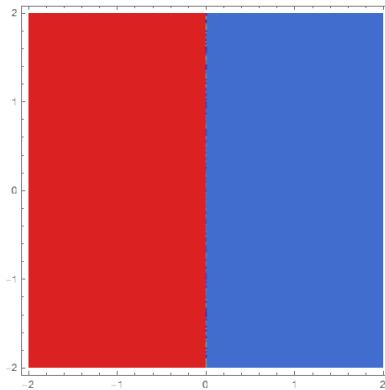


Imagen 2.2: Cuenca de atracción de $f(z) = z^2 - 1$.

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.

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 valor 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.

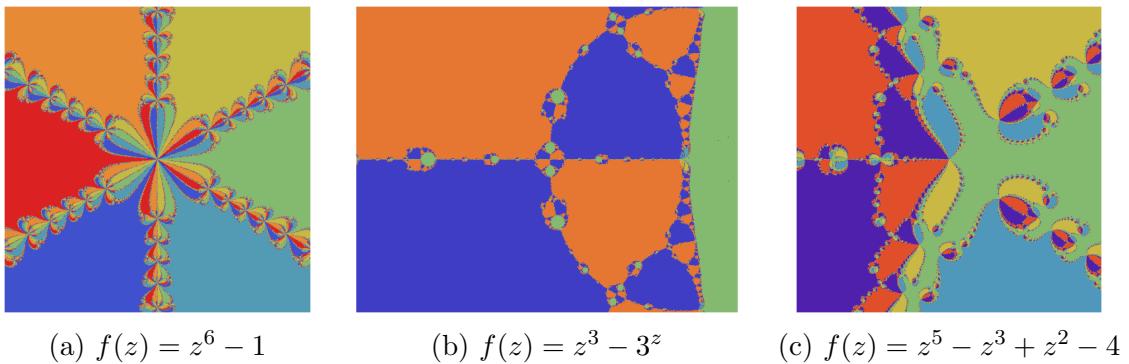


Imagen 2.3: Cuencas de atracción de distintas funciones coloreadas.

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 *caos matemático*: pequeñas variaciones en las condiciones iniciales conducen a comportamientos muy diferentes. Además, 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 converger a 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]
```

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.

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.

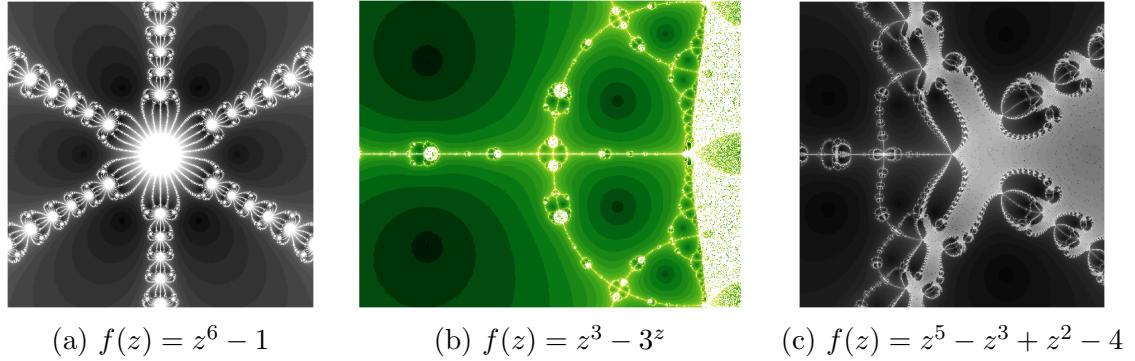


Imagen 2.4: Evaluación de la velocidad de convergencia en cada punto

Podemos jugar con estos parámetros como son la función a representar f , la región del plano que queremos visualizar $\{x, -2, 2\}, \{y, -2, 2\}$ o el valor del argumento `ColorFunction` para obtener distintas imágenes de diferentes colores.

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

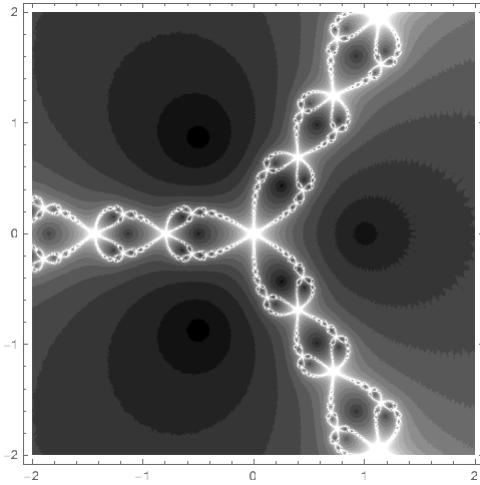


Imagen 2.5: Cuencas de atracción de $f(z) = z^3 - 1$

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 ver como esa estructura se repite en todas las escalas.

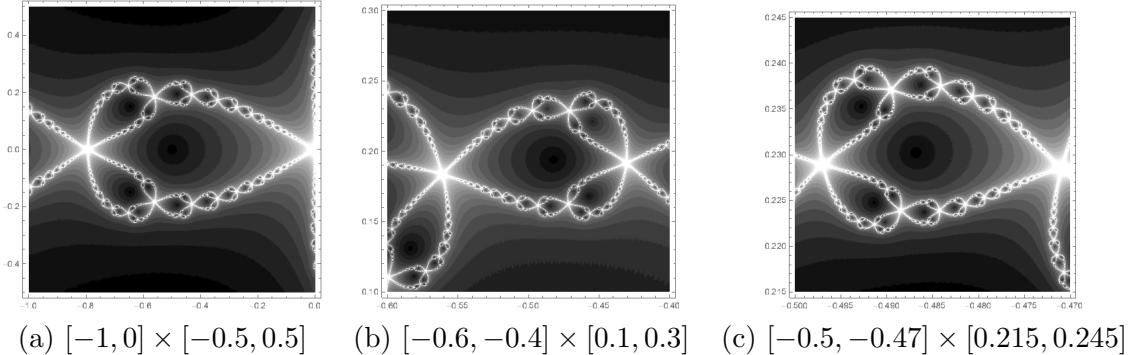


Imagen 2.6: Diferentes regiones ampliadas de la figura 2.5

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 observar este hecho en las imágenes 3.1, que son nuestros primeros dos ejemplos de conjuntos de Julia.

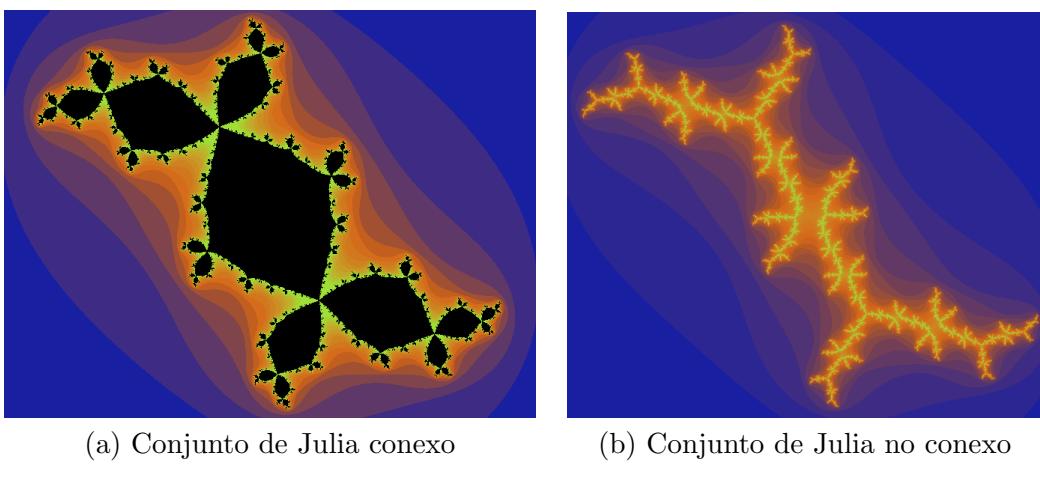
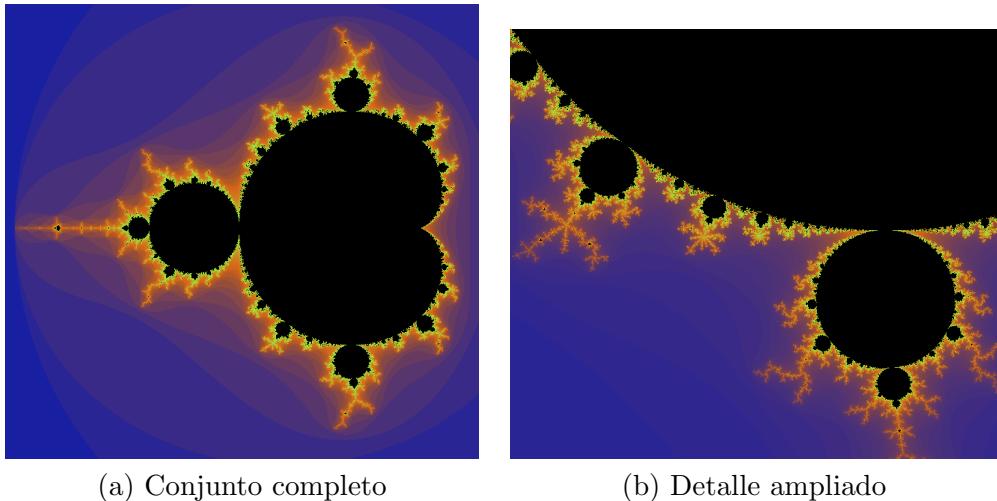


Imagen 3.1: Primeras imágenes de conjuntos de Julia

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 comprobamos 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, $O_g(1) = \{1, i, -1, -i, 1, \dots\}$.

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, fíjemonos 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 converga 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 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 J_c a la frontera de E_c . Es en este conjunto donde las iteradas tienen un comportamiento caótico. Se denomina **conjunto de Fatou** al complemento del conjunto de Julia.

Ejemplo 3.2.1. En el caso $c = 0$, es decir, $P_0(z) = z^2$, sabemos ya que $J_c = S^1$, 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 utilizaban 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:

$$\begin{aligned} |P_c(z_0)| = |z_0^2 + c| &\geq |z_0^2| - |c| \quad (\text{Propiedades del m\'odulo}) \\ &= |z_0|^2 - |c| \\ &\geq |z_0|^2 - |z_0| \quad (\text{Porque } |z_0| > |c|) \\ &= (|z_0| - 1)|z_0| \\ &= (1 + \varepsilon)|z_0| \end{aligned}$$

Por tanto tenemos que $|P_c(z_0)| \geq (1 + \varepsilon)|z_0|$, por lo que en cada iteración el m\'odulo 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 m\'aximo de iteraciones que se aplicar\'an en cada punto antes de decidir si el punto es prisionero o de escape. Si pasadas esas M iteraciones el m\'odulo de z_0 no ha alcanzado el n\'umero de escape e_c entonces consideramos que z_0 es un punto prisionero. En caso contrario, en el momento que se alcance el n\'umero de escape cesar\'an las iteraciones y se etiquetar\'a el punto como prisionero. El valor de M puede ser alto si queremos aumentar la precisi\'on a cambio de mayor tiempo de c\'omputo, y viceversa. Es posible que algunos de los puntos sean de escape pero alcancen e_c despu\'es de las M iteraciones, pero tomando un valor suficientemente alto el resultado es pr\'acticamente 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 funci\'on de las iteraciones necesarias antes de alcanzar el n\'umero de escape. Por ejemplo, si queremos representar en *Mathematica* el conjunto de la figura 3.1 (a), que era $\mathcal{J}_{-0.12+0.75i}$ utilizar\'iamos el siguiente c\'odigo:

```
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[#]] &)]
```

Fij\'emonos en que hemos utilizado el argumento `SameTest` para especificar cuando dejar de iterar, especificando que se haga cuando se alcance e_c . Hemos utilizado tambi\'en $M = 50$ como m\'aximo n\'umero de iteraciones. Podemos variar el segundo argumento de la llamada a la funci\'on `Julia`, el rango de valores, el argumento `PlotPoints` o el n\'umero m\'aximo de iteraciones M para graficar distintos conjuntos y detalles de los mismos. Algunos ejemplos de resultados de estos c\'odigos se encuentran en la imagen 3.3.

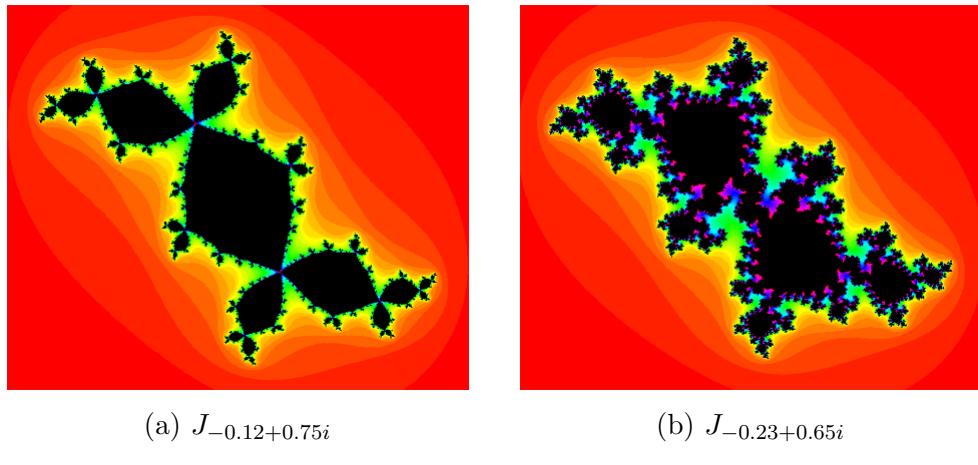


Imagen 3.3: Conjuntos de Julia graficados con Mathematica

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.

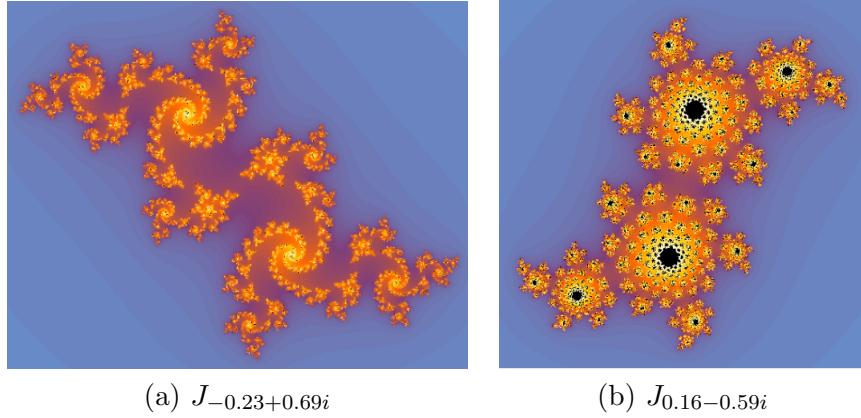


Imagen 3.4: Resultados de la orden ‘JuliaSetPlot’

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 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

¹Más información en la documentación oficial: <https://reference.wolfram.com/language/ref/JuliaSetPlot.html?q=JuliaSetPlot>

en [13, Theorem 9.5].

Teorema 3.3.1 (Teorema de Fatou-Julia). *Si P_c contiene todos los puntos críticos de $P_c(z)$, entonces $\mathcal{J}_c = \partial P_c$ es conexo. Si al menos un punto crítico de $P_c(z)$ pertenece a 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 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 $0 \in P_c$ (resp. $0 \in E_c$).*

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.

Por resumir, 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.

*Demuestra*ción. 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:

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

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 .

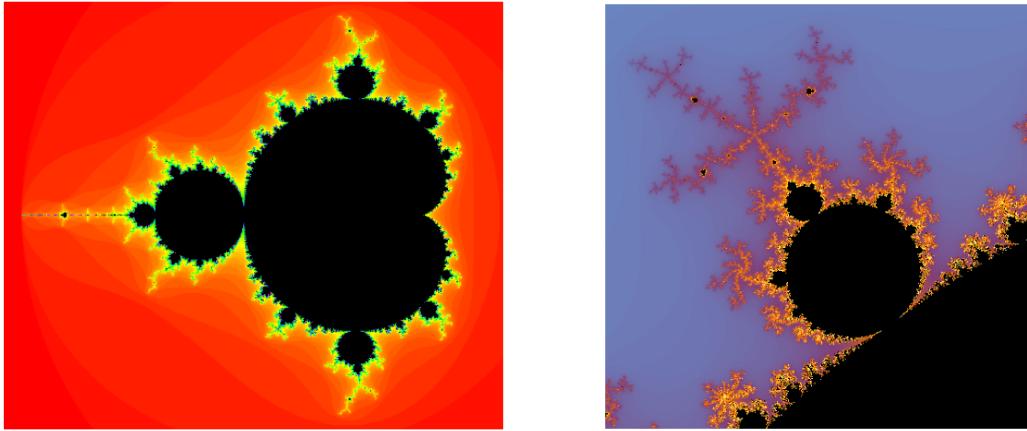
Además, en el momento que una iterada se sitúe fuera de $\bar{D}(0, 2)$, podemos asegurar que esa sucesión va a diverger, por lo que podemos dejar de iterar. Para concluir que la sucesión no diverge, al igual que para graficar conjuntos de Julia, fijamos un número máximo de iteraciones $M \in \mathbb{N}$, 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.

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 podemos ver 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.

²Más información en la documentación oficial <https://reference.wolfram.com/language/ref/MandelbrotSetPlot.html>

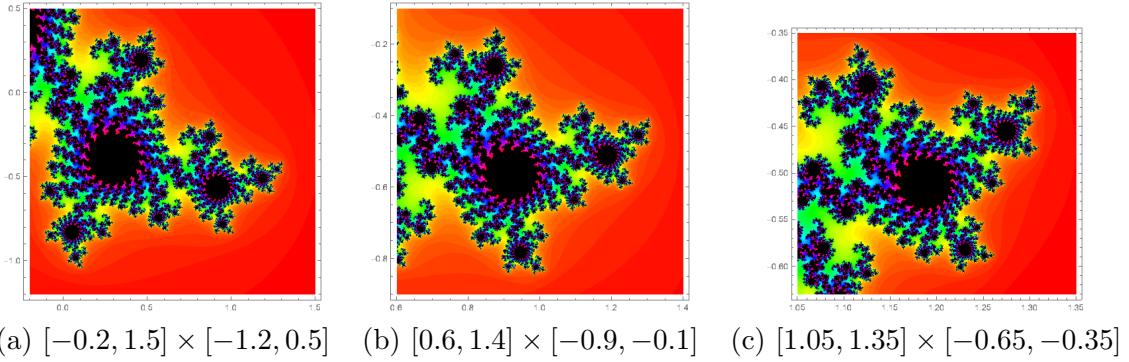


Imagen 3.6: Diferentes regiones ampliadas de la figura 3.3 (b)

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)).

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.

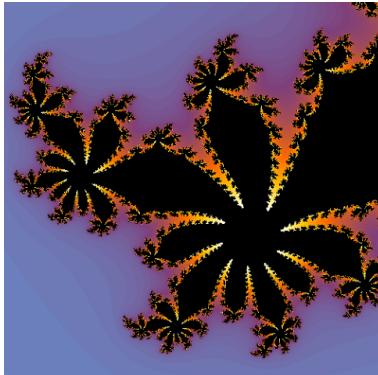
3.6. Conjuntos de Julia y Mandelbrot generalizados

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.

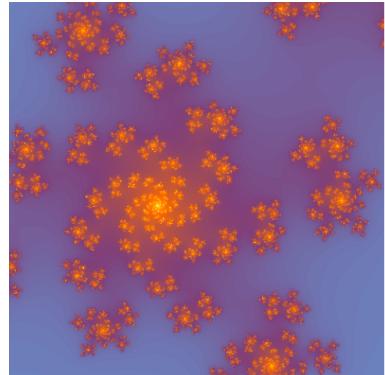
3.6.1. Familia $\{z^N + c\}_{c \in \mathbb{C}}$

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 $P_{c,N}(z) = z^N + c$, con $N \geq 2$ 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 ??.

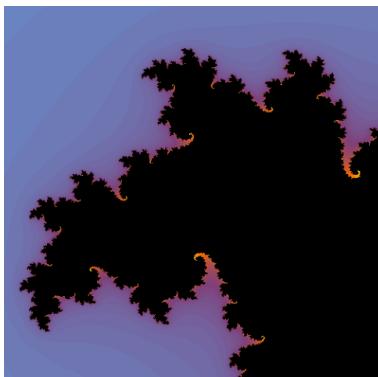
Fijémonos en que hemos fijado $c = -0.55 + 0.48i$ y hemos variado el exponente de la función polinómica. Llama la atención lo distintos que son los conjuntos entre sí cuando realmente el c fijado es el mismo en todos los casos.



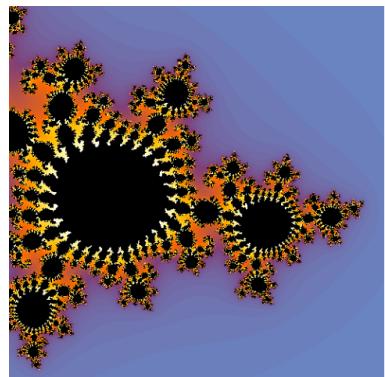
(a) $\mathcal{J}_{0.33-0.41i}$ en $[-1, -0.3] \times [-1.1, -0.4]$



(b) $\mathcal{J}_{0.45-0.62i}$ en $[-0.8, 0.2] \times [-0.8, 0.2]$



(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 $[0, 1.5] \times [-1, 0.5]$

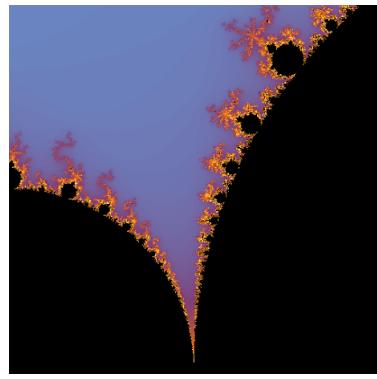
Imagen 3.7: Detalles autosimilares de algunos conjuntos de Julia

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 $\{P_{c,N}(z)\}_{c \in \mathbb{C}}$. 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 $\{P_{c,N}^n(0)\}$.

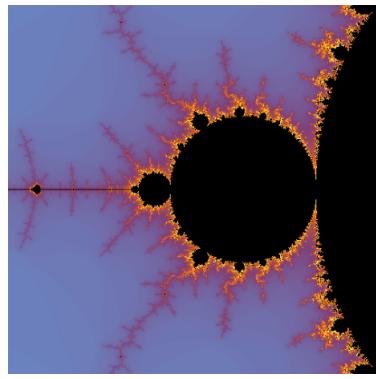
De esta forma, podemos hablar por tanto de conjuntos de Mandelbrot generalizados como representación gráfica de qué puntos tienen un conjunto de Julia conexo o polvareda utilizando la función $P_{c,N}(z)$, el cual denotamos como \mathcal{M}_N . De la misma forma que el resultado 3.2.1 es válido en la 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 ???. Por su parte, la función de *Mathematica* `MandelbrotSetPlot` admite un argumento `n` que representa el exponente de $P_{c,N}(z)$ a la hora de iterar.

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

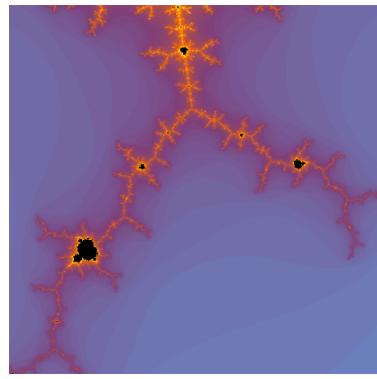
Finalmente, invitamos al lector a visitar una web interactiva en la que poder visualizar tantos conjuntos de Julia y Mandelbrot estándar y generalizados como desee. La web forma parte del proyecto y su url es <https://jantoniov.github.io>.



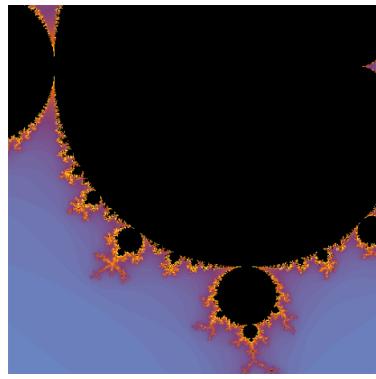
(a) $[-1, -0.5] \times [0, 0.5]$



(b) $[-1.5, -1.2] \times [-0.15, 0.15]$



(c) $[-0.2, 0] \times [-1.1, -0.9]$



(d) $[-0.9, 0.3] \times [-1, 0.2]$

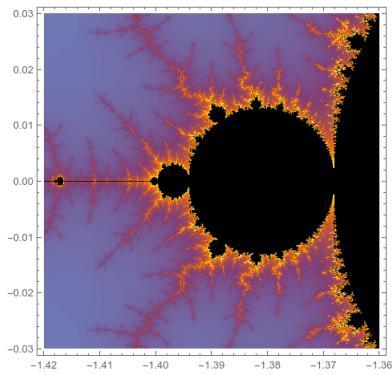
Imagen 3.8: Detalles autosimilares de \mathcal{M}

[io/Geometria-Fractal/\(WIP\)](#).

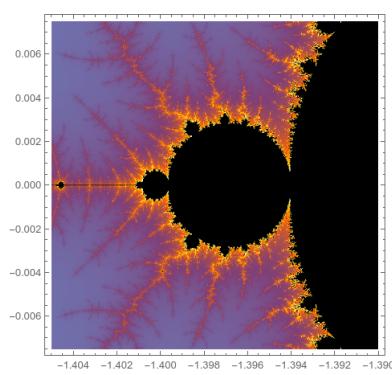
3.6.2. Conjuntos de Julia con funciones no polinómicas

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 $f_c(z) = c \cdot \cos(z)$:



(a) $[-1.42, -1.36] \times [-0.03, 0.03]$



(b) $[-1.405, -1.39] \times [-0.0075, 0.0075]$

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

- 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*[14] de *Michael Barnsley*. Por su parte, la geometría fractal nació en 1977 con la publicación de *The fractal geometry of nature*[3] 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 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 más adelante.

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 $b = \begin{pmatrix} e \\ f \end{pmatrix}$ y expresar w como:

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

La matriz A también puede ser expresada 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 $b = \begin{pmatrix} e \\ f \end{pmatrix}$.

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 $w(x) = Ax + b$ es una composición de la aplicación lineal representada por A , la cual transforma el espacio relativo al origen, y de la *traslación* de vector $b = \begin{pmatrix} e \\ f \end{pmatrix}$. A continuación definimos un caso concreto de transformación afín más familiar.

Definición 4.1.2. Una transformación afín de \mathbb{R}^2 $w(x) = Ax + b$, para cualquier $b \in \mathbb{R}^2$ 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

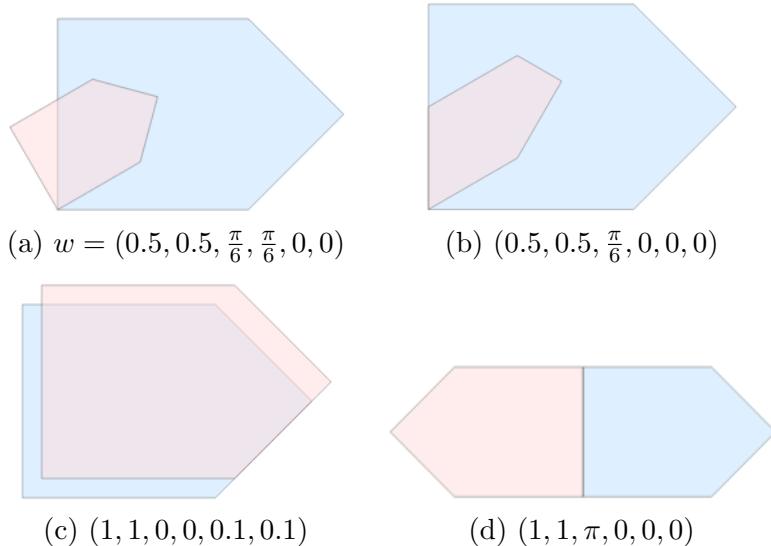


Imagen 4.1: Ejemplos de aplicaciones de transformaciones lineales

una traslación mediante el vector $b = (0.1, 0.1)^t$. Por último, en (d) vemos un caso de reflexión respecto al eje Y .

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 $w_i \quad \forall i = 1, \dots, n$.

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 .

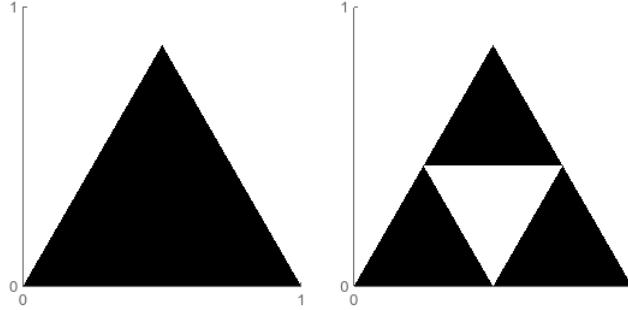


Imagen 4.2: Representación gráfica de T y $w(T)$

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).

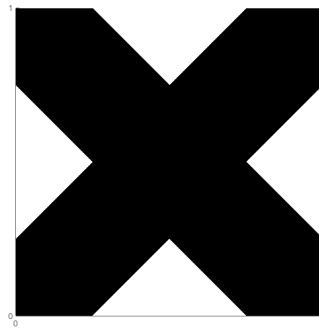


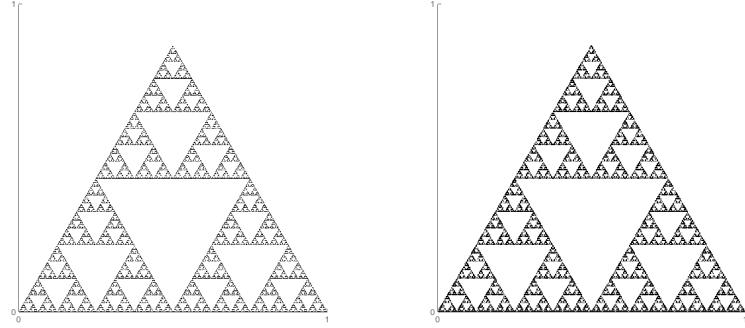
Imagen 4.3: Otra posible semilla para iterar el SFI

4.2.1. El Espacio de Fractales y la Métrica de Hausdorff

Consideramos un espacio métrico completo (X, d) y sea $\mathcal{H}(X) \subset \mathcal{P}(X)$ 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

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



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

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

donde el ínfimo es un mínimo porque A es compacto. 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 commutativa. Véase el contraejemplo de la figura 4.5.

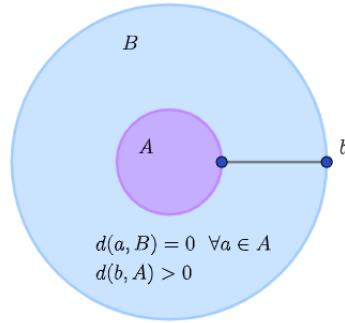


Imagen 4.5: Contraejemplo a la distancia entre conjuntos

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 [14, Sección 2.6] que, en efecto, h es una distancia. Además, en [14, 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 y $\mathcal{H}(X)$ un espacio métrico completo.

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

$$w(\mathcal{H}(X)) = \{w(A) : A \in \mathcal{H}(X)\} \subseteq \mathcal{H}(X),$$

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 :

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

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 $n \leq 1$. 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 que 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 $\max\{|r|, |s|\} \leq 1$, 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 $s(x_1, x_2) = (r \cdot x_1, s \cdot x_2)$ verifica $K = \max\{|r|, |s|\} \leq 1$. Entonces, para cualesquiera $x = (x_1, x_2), y = (y_1, y_2) \in \mathbb{R}^2$:

$$\begin{aligned} \|s(x) - s(y)\| &= \|(s_1 x_1 - s_1 y_1, s_2 x_2 - s_2 y_2)\| \\ &= \|(s_1(x_1 - y_1), s_2(x_2 - y_2))\| \\ &\leq K \|x - y\| \end{aligned}$$

Por tanto s 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 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.

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

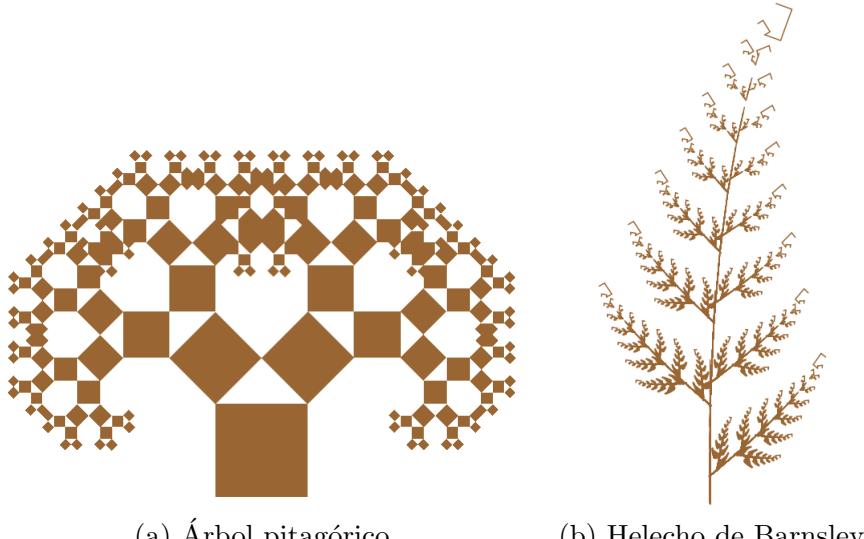


Imagen 4.6: Atractores de los SFI definidos en las tablas 4.1 y 4.2.

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 $\emptyset \neq U \subseteq X$ 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. *Sean (X, d) un espacio métrico completo, 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 [7].

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 $\dim_B(\mathbf{S}) = \frac{\log(3)}{\log(2)}$, como ya calculamos en la sección 1.2.1.

Es por tanto el momento de calcular 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 4.3. 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 4.3: SFI para la curva de Koch

4.4. El problema inverso

Como ya hemos podido ver, los SFI 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

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

donde A es 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 probamos 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 $x \in X$, la función distancia $d(x, \cdot)$ es continua, por tanto:

$$\begin{aligned} d(x, x^*) &= d(x, \lim_{n \rightarrow \infty} f^n(x)) = \lim_{n \rightarrow \infty} d(x, f^n(x)) \\ &\leq \lim_{n \rightarrow \infty} \sum_{i=1}^n d(f^{i-1}(x), f^i(x)) \quad (\text{desigualdad triangular}) \\ &\leq \lim_{n \rightarrow \infty} d(x, f(x)) \sum_{i=1}^n s^{i-1} \\ &= (1-s)^{-1} d(x, f(x)) \end{aligned}$$

donde en la última igualdad hemos utilizado que la suma de los elementos de una serie geométrica de razón $0 \leq \alpha < 1$ es $(1-\alpha)^{-1}$. \square

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).

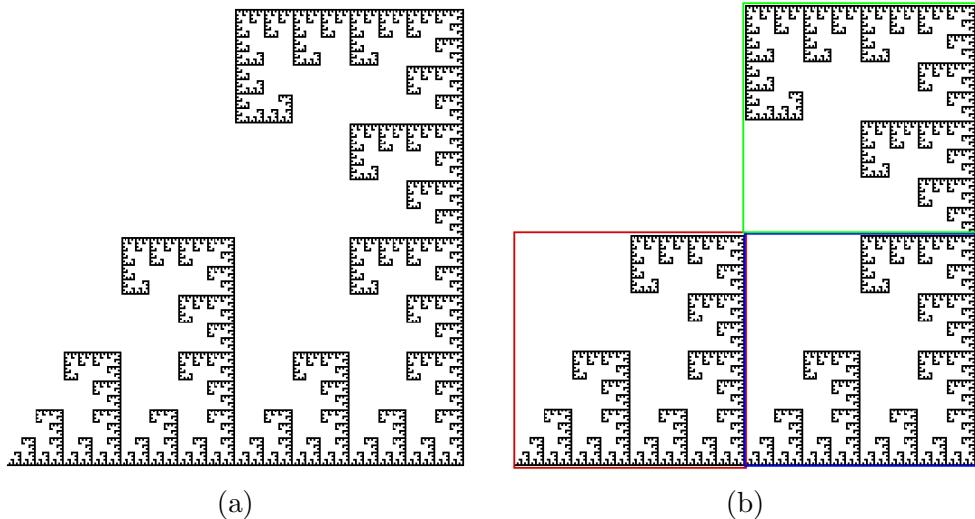


Imagen 4.7: Imagen cuyo SFI debemos determinar

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.

- 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 siguiente SFI:

Para una mayor cantidad y variedad de ejemplos invitamos al lector a consultar [14, Sección 3.10], donde puede encontrar gran cantidad de ejercicios y explicaciones prácticas y más complejas.

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 4.4: SFI para la imagen 4.7

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 teoría 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 RENDERIZACIÓN

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 el renderizado 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 3 dimensiones.

Sin embargo, y como cabría esperar, existen herramientas que nos ayudan a visualizar las directivas geométricas que nosotros mismos queramos, y además podremos hacerlo a tiempo real. En nuestro caso, utilizaremos la librería **WebGL**, que es una API de JavaScript para el renderizado de gráficos 2D y 3D dentro de cualquier navegador web. Precisamente este último aspecto es el que nos permitirá desplegar nuestro proyecto en una página web interactiva y accesible desde cualquier dispositivo cuya GPU se lo permita¹.

WebGL permite que las aplicaciones web utilicen una API basada en OpenGL ES 2.0 para renderizar imágenes 3D en un elemento `<canvas>` de HTML en los navegadores y sistemas que lo soporten sin necesidad de plug-ins. El principal defecto de herramientas como WebGL es la dificultad a la hora de querer comenzar a visualizar imágenes, pues tiene varios componentes complejos de enlazar en un principio. A continuación explicaremos el flujo de trabajo que utiliza WebGL y cómo podemos comenzar a utilizar la herramienta para visualizar fractales.

¹Esto se puede comprobar gracias a páginas dedicadas a ello como esta página de testeo WebGL o WebGL Report

5.1. 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>
```

5.1.1. 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 // Initialize the GL context
3 const gl = canvas.getContext("webgl2");
4
5 // Only continue if WebGL is available and working
6 if (gl === null) {
7   alert("Unable to initialize WebGL. Your browser or machine
8       may not support it.");
9 }
```

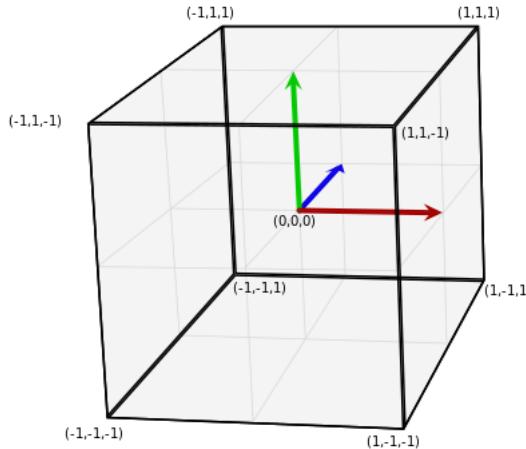
En caso de éxito, que se tiene en la mayoría de las ocasiones, debemos conservar este objeto, pues será necesario para utilizar la gran mayoría de directivas que implementa WebGL. Los siguientes elementos son los auténticos encargados de los objetos que se renderizan en el canvas.

5.1.2. 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. 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: Vertex Shader Source Code
2 // fsSource: Fragment Shader Source Code
3
4 const vertexShader = loadShader(gl, gl.VERTEX_SHADER,
5                               vsSource);
6 const fragmentShader = loadShader(gl, gl.FRAGMENT_SHADER,
7                                 fsSource);
8
9 // Create the shader program
10
11 const shaderProgram = gl.createProgram();
12 gl.attachShader(shaderProgram, vertexShader);
13 gl.attachShader(shaderProgram, fragmentShader);
14 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.

5.1.3. 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,    // white
3   1.0, 0.0, 0.0, 1.0,    // red
4   0.0, 1.0, 0.0, 1.0,    // green
5   0.0, 0.0, 1.0, 1.0,    // blue
6 ];
7 // Buffer creation
8 const colorBuffer = gl.createBuffer();
9 // We select the buffer for the future buffer operations
10 gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
11 // We assign the 'colors' array data to the 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 // Select the 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 renderizar la escena completa con la función `drawArrays`.

```

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

```

5.2. Primera imagen renderizada

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 5.1.3. Servimos al shader de valores a partir de los buffer y el resultado es el mostrado en la imagen 5.2.

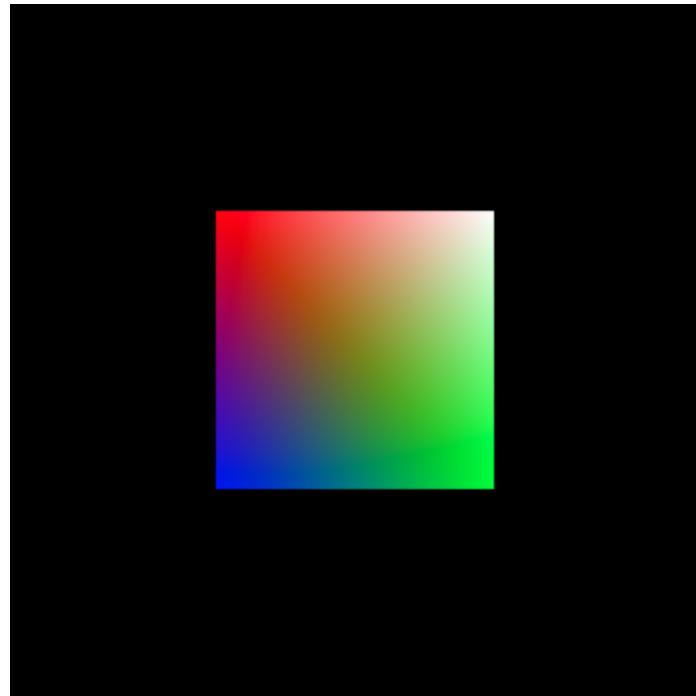


Imagen 5.2: Cuadrado de colores renderizado con WebGL

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 shader* y *fragment shader*.

Este ejemplo se corresponde al expuesto en [20], cuya adaptación podemos encontrar en el repositorio de este proyecto, 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.

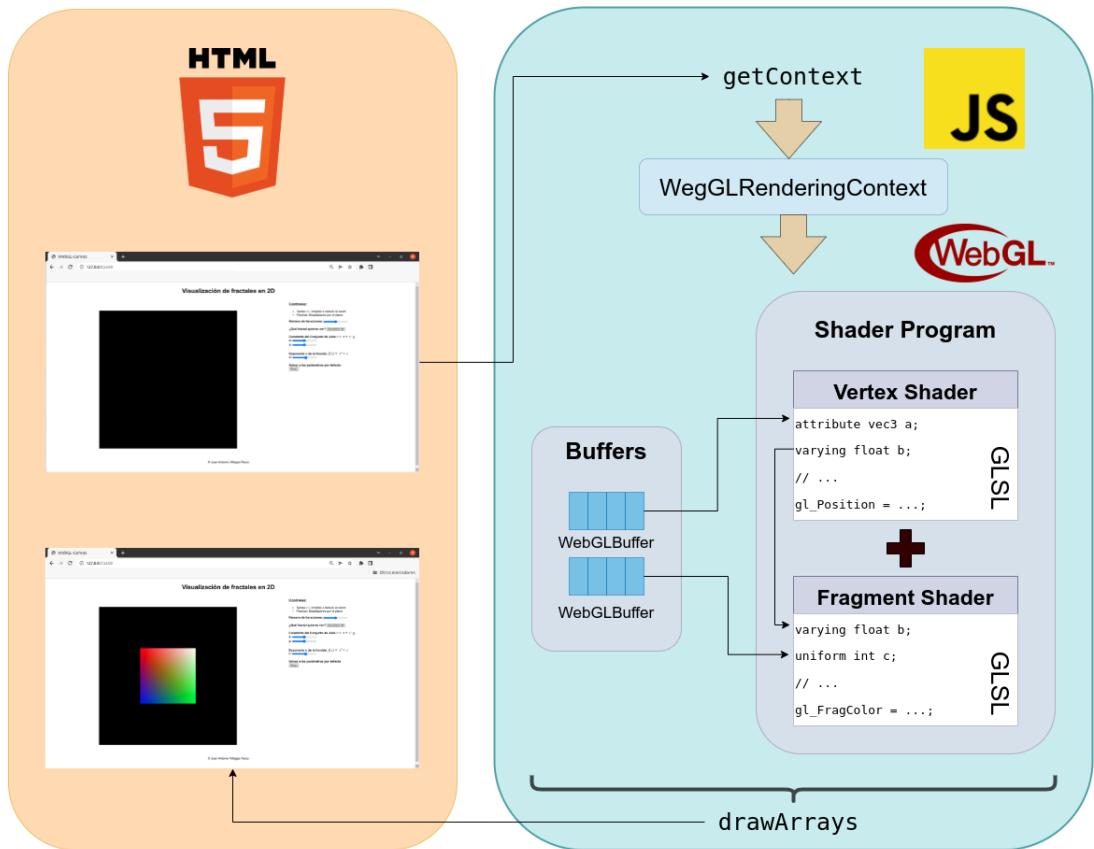


Imagen 5.3: Componentes de WebGL e interacciones entre ellos

CAPÍTULO 6

VISUALIZACIÓN DE FRACTALES EN 2D

En el capítulo 5 introdujimos el uso de WebGL como herramienta de renderizado 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. 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 renderizado se utiliza el convenio de 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.

6.1. Objetivo

Procedemos a explicar el objetivo principal de este objetivo y para el cual programaremos cada línea de código: Queremos desarrollar una página web interactiva, que cuente con un canvas donde se renderice el fractal que deseemos y, además, haya una serie de parámetros que se puedan controlar dinámicamente, de forma que conforme se cambia un parámetro el canvas modifica la imagen que está renderizando.

Queremos visualizar conjuntos de Julia \mathcal{J}_c para distintos $c \in \mathbb{C}$, el conjunto de Mandelbrot, y las generalizaciones de los conjuntos de Julia y Mandelbrot ocasionadas si se itera la función $P_{c,N}(z) = z^N + c$ para distintos valores de $N \in \mathbb{N}$. Además, si revisitamos el algoritmo que utilizamos en la sección 3.2.1 para graficar en *Mathematica* conjuntos de Julia y el que utilizamos en la sección 3.4.1 para visualizar el conjunto de Mandelbrot, podremos recordar que para aproximar qué puntos del plano complejo eran prisioneros o de escape fijábamos un número máximo de iteraciones M , tras las cuales se consideraba que un número $z_0 \in \mathbb{C}$ era prisionero si la sucesión de los módulos de sus iteradas $\{P_{c,N}^n(z_0)\}$ no superaba el número de escape $e_c = \max\{2, |c|\}$. Este valor M también podría ser un parámetro modificable, para así poder ver dinámicamente cómo cambia la resolución cuando se cambia el número máximo de iteraciones.

Además, conviene añadir la posibilidad de desplazarse y hacer zoom en distintas regiones del plano, para así poder explorar en detalle las regiones del fractal, permitiendo observar en detalle las autosimilaridades que nos ofrecen, como ya vimos en la sección 3.5.

6.2. Estructurando el código

Podemos usar como base el código utilizado para visualizar el cuadrado de colores, ya que nos puede venir bien su estructura para adaptar la misma a la renderización de fractales. Sin embargo, tiene una estructura muy procedural. Podemos 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 y siguiendo los principios SOLID.

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

- **Scene2D**: Inicializa y gestiona todos los componentes de WebGL necesarios para renderizar 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 **draw** que renderiza la escena a partir de los parámetros actuales.
- **ShaderType**:
un enumerado que puede tomar los valores `ShaderType.vertexShader` o `ShaderType.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 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 `draw` y así modificar dinámicamente la visualización.

El código completo se puede encontrar en GitHub: <https://github.com/JAntonioVR/Geometria-Fractal/tree/main/static>. La funcionalidad completa y detallada de cada clase puede consultarse en el apéndice A.

6.3. El fragment shader

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.

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 queremos inicialmente visualizar la región $[-2, 2] \times [-2, 2]$ y que las coordenadas de dispositivo son $(x, y) \in [0, 720] \times [0, 720]$, las cuales están precisamente en la región $[0, 720] \times [0, 720]$ porque las dimensiones del canvas son 720×720 píxeles. Entonces la transformación lineal que necesitamos es

$$\begin{aligned}\phi : [0, 720] \times [0, 720] &\longrightarrow [-2, 2] \times [-2, 2] \\ (x, y) &\longmapsto \frac{4}{720}(x, y) - (2, 2)\end{aligned}\tag{6.1}$$

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 $[-2, 2] \times [-2, 2]$ queremos representar cualquier otra, pero aún centrada en el origen $(0, 0)$. Podemos generalizar la transformación ϕ para cualquier otro intervalo, pero en lugar de ello y para mantener las proporciones introduciremos una variable que represente el *zoom* que se aplica a la imagen, de tal forma que tan solo habría que multiplicar el resultado de la transformación 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

$$\begin{aligned}\phi : [0, 720] \times [0, 720] &\longrightarrow [-2, 2] \times [-2, 2] \\ (x, y) &\longmapsto \lambda \left(\frac{4}{720}(x, y) - (2, 2) \right)\end{aligned}\tag{6.2}$$

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 (6.2) nos devuelve coordenadas en regiones centradas simplemente tenemos que sumar este par al resultado, de forma que nos queda

$$\begin{aligned}\phi : [0, 720] \times [0, 720] &\longrightarrow [-2, 2] \times [-2, 2] \\ (x, y) &\longmapsto (x_0, y_0) + \lambda \left(\frac{4}{720} (x, y) - (2, 2) \right)\end{aligned}\tag{6.3}$$

donde 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, de forma que queda el siguiente fragmento de código:

```

1 // Zoom: constante lambda que define el tamaño de la region a
   representar
2 uniform float u_zoomSize;
3 // Zoom center: Punto del plano situado en el centro del
   canvas
4 uniform vec2 u_zoomCenter;
5
6 // ...
7
8 vec2 get_world_coordinates() {
9     // Coordenadas de dispositivo normalizadas en [0,1]x[0,1]
10    vec2 uv = gl_FragCoord.xy / vec2(720.0, 720.0);
11    // Punto del plano complejo al que corresponde el pixel
12    return u_zoomCenter + (uv * 4.0 - vec2(2.0)) * u_zoomSize;
13 }
```

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 $P_{c,N}$ y en función del número de iteraciones necesarias para diverger (o no), asignar un color.

6.3.1. La función $f(z) = z^N + c$

Necesitamos código para la función $P_{c,N}$, 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) N 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

$$\begin{aligned}z^n &= z^{n-1} \cdot z = (\operatorname{Re} z^{n-1} + i \cdot \operatorname{Im} z^{n-1}) \cdot (\operatorname{Re} z + i \cdot \operatorname{Im} z) \\ &= (\operatorname{Re} z^{n-1} \cdot \operatorname{Re} z - \operatorname{Im} z^{n-1} \cdot \operatorname{Im} z) + (\operatorname{Re} z^{n-1} \cdot \operatorname{Im} z + \operatorname{Im} z^{n-1} \cdot \operatorname{Re} z) \cdot i\end{aligned}\tag{6.4}$$

y esto es válido para cualquier $n \in \mathbb{N}$. Podemos utilizar iterativamente la ecuación (6.4) 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;
6         current_pow = vec2( z_ant.x*z.x - z_ant.y*z.y,
7                             z_ant.x*z.y + z_ant.y*z.x);
8         if(i >= n) break;
9     }
10    return current_pow;
11 }
```

Recordamos, para el que no esté acostumbrado al código GLSL, que 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 $P_{c,N}$ aprovechando la aritmética preprogramada para los objetos vector en GLSL.

```

1 vec2 P(vec2 z, vec2 c, int n) {
2     return complex_pow(z,n) + 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 N 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^N + c
2 uniform vec2 u_juliaSetConstant;
3 // Valor N fijo en la funcion z^N + c
4 uniform int u_order;
```

6.3.2. Asignación de colores

Con la función $P_{c,N}$ ya programada, recordamos los algoritmos que utilizamos en *Mathematica* para graficar conjuntos de Julia y Mandelbrot, reprogramándolos ahora en GLSL para poder renderizar dichos conjuntos. Estos consistían en iterar la función $P_{c,N}$, 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 ($\text{rgb}(0, 0, 0)=\#000000$) para los puntos prisioneros (los que no divergen tras las M iteraciones) y para los que escapan utilizar un gradiente entre los siguientes colores:



Imagen 6.1: Paleta de colores elegida para la visualización de fractales 2D

La cual nos proporciona el siguiente gradiente, que proximamente 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.

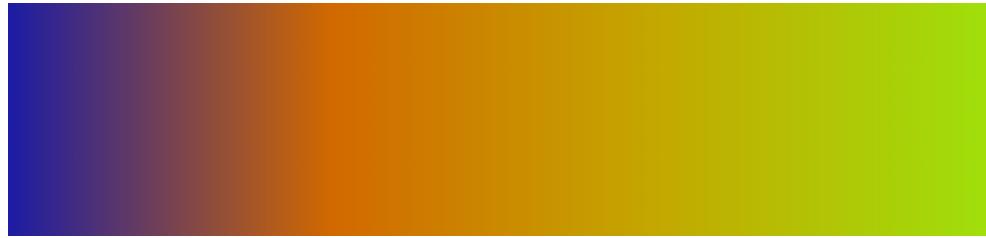


Imagen 6.2: Gradiente generado por la paleta de colores 6.1

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 asigna a la variable
16 // gl_FragColor el color que le corresponde.
17 void assignColor(bool escaped, int iterations) {
18     gl_FragColor = escaped ? vec4(palette(
19         3.0*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) // #A0E00B
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.

6.3.3. Renderizando conjuntos de Julia

Tenemos entonces todos los ingredientes para programar el renderizado, tan solo falta decidir si el punto que representa el píxel es prisionero o escapa mediante la iteración de $P_{c,N}$. Tras obtener las coordenadas de mundo que se le asignan al píxel mediante el método descrito al inicio de esta sección, debemos iterar la función $P_{c,N}$ almacenando el número de iteraciones dadas. 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. Atendiendo a esta decisión y al número de iteraciones dadas se asigna el color correspondiente como comentamos en la sección 6.3.2. El código GLSL sería el siguiente:

```

1 // A partir del valor de la constante c y el exponente n
2 // iteramos la función P_{c,N} y asignamos el color.
3 void Julia(vec2 c, int n) {
4     vec2 z0 = get_world_coordinates();
5     int iterations;
6     vec2 z = z0;
7     bool escaped = false;
8     float length_c = length(c);
9     for(int i = 0; i < 10000; i++) {
10         if(i > u_maxIterations) break;
11         iterations = i;
12         z = P(z, c, n);
13         if (length(z) > 2.0){
14             escaped = true;
15             break;
16         }
17     }
18     assignColor(escaped, iterations);
20 }

```

Ya desde la función `main`, llamamos a esta función recién presentada y obtenemos los resultados que se presentan en las imágenes 6.3.

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

6.3.4. Renderizando 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}_N 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}_N se componía de aquellos elementos

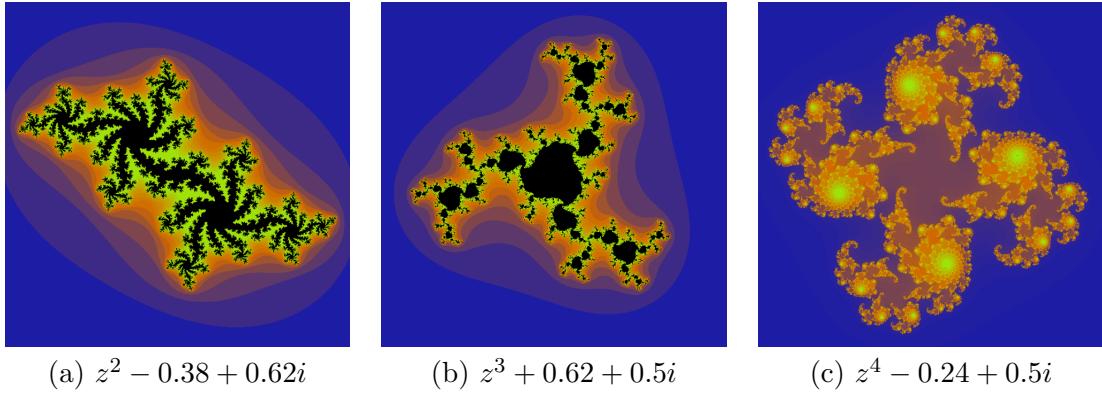


Imagen 6.3: Renderizado de algunos conjuntos de Julia con WebGL

$c \in \mathbb{C}$ tales que el conjunto de Julia \mathcal{J}_c es conexo, o equivalentemente, de aquellos cuya sucesión $\{P_{c,N}^n(0)\}$ no diverge.

La metodología consiste por tanto en obtener las coordenadas de mundo del píxel, iterar la función $P_{c,N}$ tomando $z_0 = 0$ como semilla y observar qué sucede. 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}_N . 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:

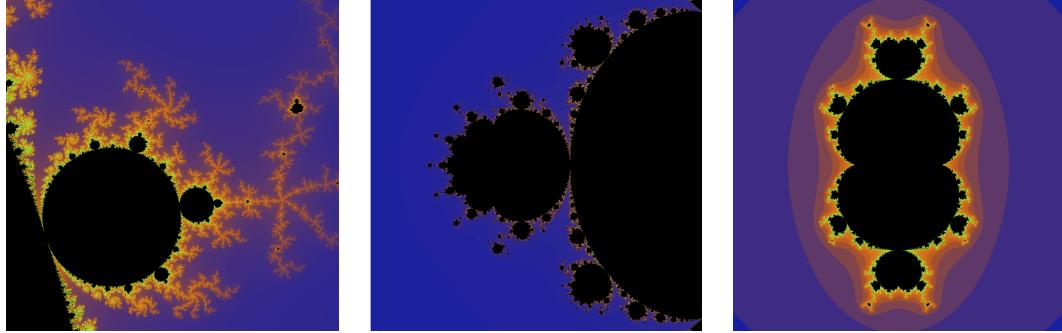
```

1 void Mandelbrot(int n) {
2     vec2 c = get_world_coordinates();
3     int iterations;
4     vec2 z = vec2(0.0);
5     bool escaped = false;
6     for (int i = 0; i < 10000; i++) {
7         if (i > u_maxIterations) break;
8         iterations = i;
9         z = P(z, c, n);
10        if (length(z) > 2.0) {
11            escaped = true;
12            break;
13        }
14    }
15    assignColor(escaped, iterations);
16 }
```

Vemos que es muy parecida a la función `Julia` presentada en la sección 6.3.3. 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` 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 | Mandelbrot(u_order);
```



(a) Detalle de \mathcal{M}_2

(b) Detalle de \mathcal{M}_8

(c) \mathcal{M}_3

Imagen 6.4: Renderizado de algunos conjuntos de Mandelbrot con WebGL

6.3.5. 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 // Renderizar conjunto de Julia o de Mandelbrot
2 uniform int u_fractal;
3
4 // ...
5
6 void main() {
7     if(u_fractal == 0)
8         Mandelbrot(u_order);
9     else
10        Julia(u_juliaSetConstant, u_order);
11 }
```

CAPÍTULO 7

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 renderizar fractales en tridimensionales.

7.1. Definición de Ray-Tracing

En el mundo de la informática gráfica existen dos formas fundamentales de proceder al renderizado 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 renderizarlo es identificar para cada objeto (también llamado primitiva) qué píxeles ocupan y asignar color a cada píxel, aplicando posibles modelos de iluminación y texturas. Esta es la técnica conocida como **rasterización**.
- Otra forma es, para cada pixel, calcular qué primitivas ocupan posiciones del espacio asociadas a dicho píxel y colorearlo dependiendo de la primitiva más cercana al observador. Esta es la base del **Ray-Tracing**.

Más en profundidad, la idea es situar al espectador en una cierta posición de la escena y colocar frente a él un plano, el cual estará dividido en tantos píxeles como tenga el canvas, de tal manera que se ‘trazan rayos’ que salen desde la posición del espectador 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 7.1 se puede observar un esquema aclarativo del funcionamiento de este método.

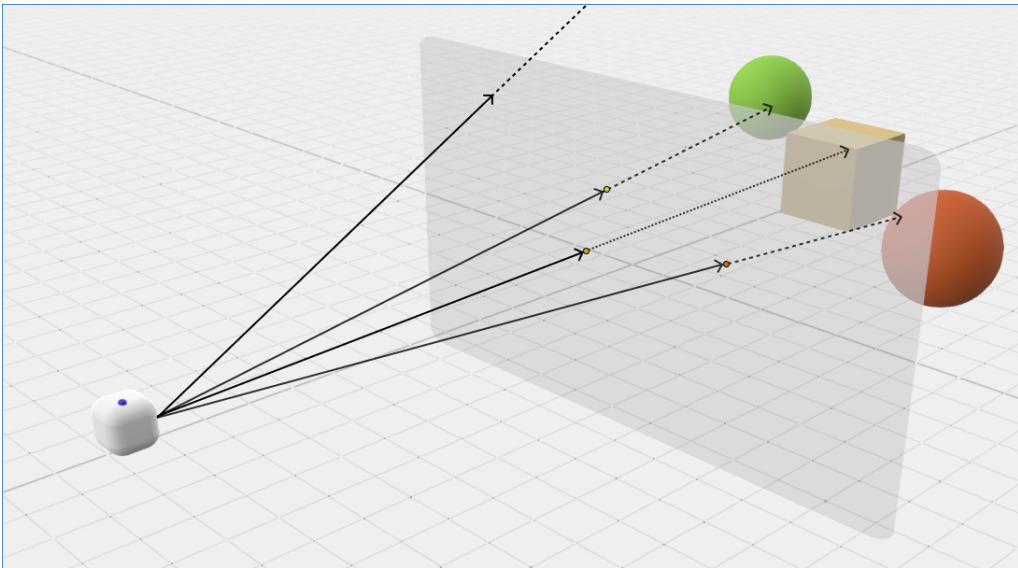


Imagen 7.1: Esquema básico del funcionamiento del Ray-Tracing

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, recordemos que las figuras fractales que perseguimos no es posible dibujarlas mediante un conjunto de vértices o aristas al no ser objetos de la geometría euclídea, por lo que sería muy difícil expresarlos como primitivas rasterizables. Lo más natural es lanzar rayos y, en caso de detectar intersección con el fractal estimar la normal en dicho punto y aplicar un modelo de iluminación. Es por esta serie de razones por las que para nuestro objetivo necesitamos programar un *ray-tracer* inicialmente básico con el objetivo futuro de renderizar fractales 3D.

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 tarjetas gráficas, comenzó a desarrollar algoritmos para poder utilizar RT en sus tarjetas 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).

Sin embargo, a pesar de dicha desventaja veremos que nuestro código puede ejecutarse en tiempo real, teniendo en cuenta siempre que debemos evitar el uso de funciones que comprometan la velocidad de ejecución. Por ejemplo, la función `pow` es muy lenta, por lo que debemos evitar usarla cuando se usen potencias enteras.

En este 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 6. 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 uniformes 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: `ray-tracer.js`. Recomendamos revisitar la sección 6.2 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.

7.2. Creación del rayo

Recordamos que cuando se trataba de fractales 2D utilizamos una transformación lineal para identificar cada píxel con un punto del plano complejo (inicio de la sección 6.3). En este caso debemos identificar cada píxel con un punto no del plano complejo, sino del que llamaremos *plano de proyección*, 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. En este caso utilizamos un canvas de 1280×720 píxeles, que guarda una proporción de 16 : 9, que es de hecho la más habitual.

```
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* al punto desde el que se observa, es decir, en el que se sitúa el observador; y *lookat* al punto hacia el que mira, que en este caso sería el $(0, 0, -1)$.

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;           // Ray's origin
3 |     vec3 dir;            // Ray's direction
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 7.2.1. 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 un rayo 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 $lookfrom = (0, 0, 0)$, llamamos df a la distancia focal, $PP_{width} = 2$ a la altura del plano de proyección, PP_{width} a la anchura del plano de proyección, $AR = 16/9$ (*aspect ratio*) a la proporción $AR = \frac{PP_{width}}{PP_{height}}$, de forma que se verifica $PP_{width} = AR \cdot PP_{height}$. Con estas variables, llamemos *LLC* (*Lower Left Corner*) al punto situado en la esquina inferior izquierda del plano de proyección, entonces:

$$LLC = lookfrom - \frac{PP_{width}}{2} \cdot (1, 0, 0) - \frac{PP_{height}}{2} \cdot (0, 1, 0) - df \cdot (0, 0, 1) \quad (7.1)$$

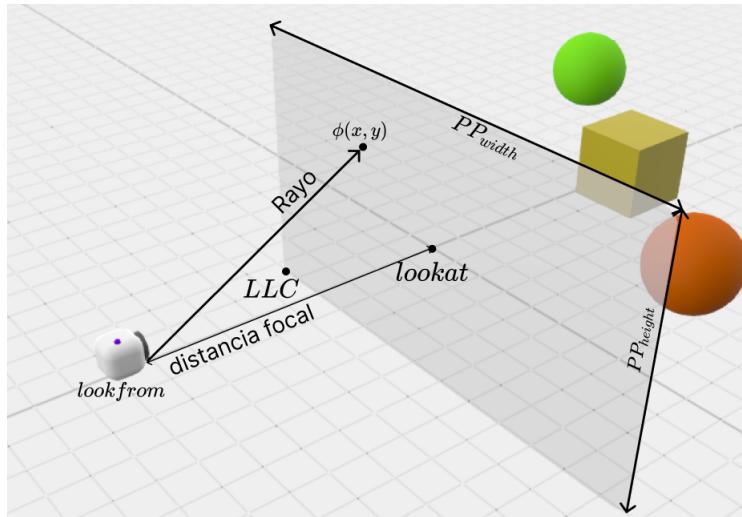


Imagen 7.2: Elementos que participan el RT

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 6.3, pero en este caso nos debemos llevar la región $[0, 1280] \times [0, 720]$

a $[0, PP_{width}] \times [0, PP_{height}]$. La transformación por tanto en este caso es

$$\begin{aligned}\phi : [0, 1280] \times [0, 720] &\longrightarrow [0, PP_{width}] \times [0, PP_{height}] \\ (x, y) &\longmapsto \left(\frac{PP_{width} \cdot x}{1280}, \frac{PP_{height} \cdot y}{720} \right)\end{aligned}\quad (7.2)$$

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`. Y 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 identificamos el píxel:

$$destiny := LLC + \phi(x, y)$$

Y por tanto, la dirección del rayo sería

$$\vec{v} = LLC + \phi(x, y) - lookfrom$$

y el origen sería obviamente el punto *lookfrom*.

El código GLSL que hemos utilizado hasta este punto 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 = float(16.0) / float(9.0);
24 float image_width = 1280.0;
25 float image_height = image_width / aspect_ratio;
26
27 // Coordenadas de dispositivo normalizadas [0,1]
28 vec2 uv = gl_FragCoord.xy / vec2(image_width, image_height);
29 float u = uv.x;
30 float v = uv.y;
31
32 // Dimensiones del plano de proyección
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 punto correspondiente al píxel en el plano de proyección.

7.3. 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 proyecto.

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 does not hit any surface
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
6                 (0.5,0.7,1.0), 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 7.3 podemos ver el gradiente utilizado.

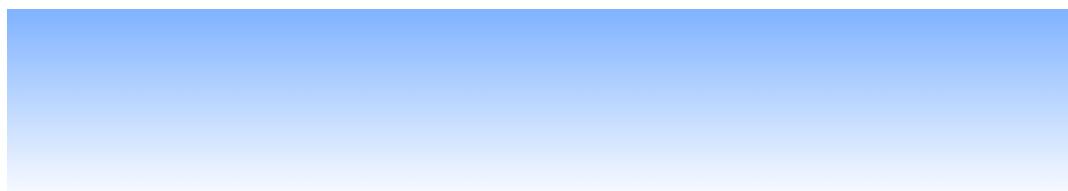


Imagen 7.3: Gradiente utilizado para el fondo de la escena

Y en la imagen 7.4 podemos ver el resultado de efectuar la llamada a `ray_color` una vez creado el rayo. Esta es la primera escena 3D renderizada utilizando RT en WebGL. Nótese que en la imagen no aparecen todos los colores del gradiente de

la imagen 7.3, 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 7.5), 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’.

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



Imagen 7.4: Primera escena vacía renderizada

7.4. 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 intersecar. 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.

7.4.1. Renderizando 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.

$$\begin{aligned}
 (R(t) - c) \cdot (R(t) - c) &= r^2 \\
 (p_0 + \vec{v}t - c) \cdot (p_0 + \vec{v}t - c) &= r^2 \\
 (\vec{v}t + (p_0 - c)) \cdot (\vec{v}t + (p_0 - c)) &= r^2 \\
 (\vec{v} \cdot \vec{v})t^2 + 2(\vec{v} \cdot (p_0 - c))t + (p_0 - c) \cdot (p_0 - c) - r^2 &= 0
 \end{aligned} \tag{7.3}$$

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 a la posición del espectador. 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;
3     float radius;
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 // Calculates the intersection between a Ray and a
2 // Sphere and stores the hit information in a Hit_record
3 // struct.
4
5 Hit_record hit_sphere(Sphere S, Ray R,
6     float t_min, float t_max) {
7
8     Hit_record result;
9     vec3 oc = R.orig - S.center;
10    float a = dot(R.dir, R.dir);
11    float b = 2.0 * dot(oc, R.dir);
12    float c = dot(oc, oc) - S.radius*S.radius;
13    float discriminant = b*b - 4.0*a*c;
14    if (discriminant < 0.0){
15        result.hit = false;
16        return result;
17    }

```

```

18     float sqrnd = sqrt(discriminant);
19     float root = (-b - sqrt(discriminant))/(2.0*a); // First
20         root
21     if (root < t_min || t_max < root){
22         // The first root is out of range
23         root = (-b + sqrt(discriminant))/(2.0*a); // The other
24             root
25     if (root < t_min || t_max < root){
26         // Both roots are out of range
27         result.hit = false;
28         return result;
29     }
30     result.hit = true;
31     result.t = root;
32     result.p = ray_at(R, result.t);
33     result.normal = normalize((result.p - S.center) / S.radius
34         );
35     return result;
36 }
```

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 (7.3) 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 7.6, 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 hits the sphere?
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(normalize(hr.normal), 1.0);
14    }
15
16    // R does not hit any surface
17    vec3 unit_direction = normalize(R.dir);
```

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

```

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
20         (0.5,0.7,1.0), 1.0);
21 }
22 // ...
23 // R is the Ray
24
25 Sphere S;
26 S.center = vec3(0.0, 0.0, -1.0); S.radius = 0.5;
27 gl_FragColor = ray_color(R, S);

```

El resultado que obtenemos es el que podemos observar en la imagen 7.5. Obsérvese la variedad de colores que ofrece la esfera, concordante con la variedad de normales que posee.

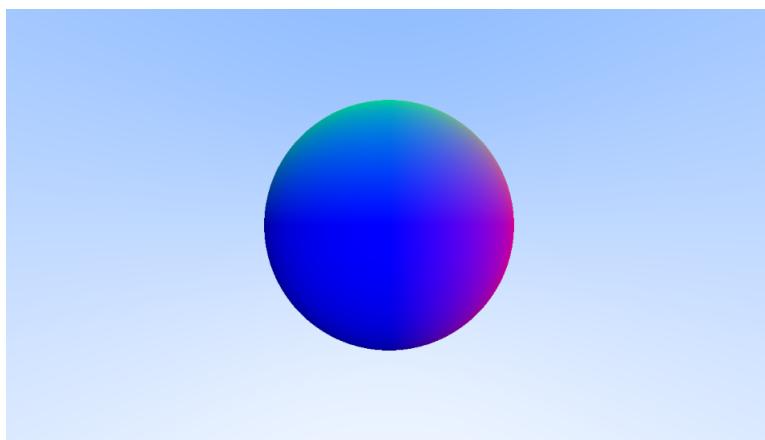


Imagen 7.5: Escena con una esfera

7.4.2. Renderizando 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    float closest_t = t_max;
11    for(int i = 0; i < ARRAY_TAM; i++){
12        if(i == size) break;
13        // Buscamos tan lejos como la ultima interseccion
14        tmp = hit_sphere(spheres[i], R, t_min, closest_t);
15        if(tmp.hit){ // Hay una interseccion mas cercana
16            closest_t = tmp.t;
17            result = tmp;
18        }
19    }
20    return result;
21 }
```

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 hits any sphere?
4     Hit_record hr = hit_spheres_list(world, size, R, 0.0,
5         MAX_DIST);
6     if(hr.hit)
7         return vec4(normalize(hr.normal), 1.0);
8
9     // R does not hit any surface
10    // Background code ...
11 }
12 // ...
13 // R is the Ray
14
15 // Spheres
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 7.6, en la que podemos ver varias esferas de distintos tamaños y con distintas posiciones.

7.4.3. Renderizando 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

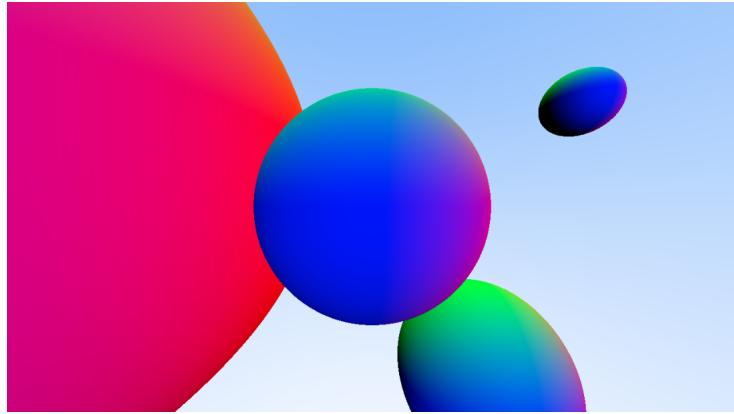


Imagen 7.6: Escena con varias esferas

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:

$$\begin{aligned} R(t) \cdot \vec{N} &= D \\ (p_0 + \vec{v}t) \cdot \vec{N} &= D \\ t &= \frac{D - p_0 \cdot \vec{N}}{\vec{v} \cdot \vec{N}} \end{aligned} \tag{7.4}$$

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;          // Término independiente
4 | };

```

Y a partir de la ecuación (7.4) 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 | {
3 |     Hit_record result;
4 |     float oc = dot(P.normal, R.dir);
5 |     if(oc == 0.0){ // No hay intersección
6 |         result.hit = false;
7 |         return result;
8 |     }
9 |     float t = (P.D - dot(P.normal, R.orig))/oc;
10 |    if (t < t_min || t > t_max)
11 |        result.hit = false;
12 |    else{
13 |        result.hit = true;
14 |        result.t = t;
15 |        result.p = ray_at(R, result.t);
16 |        result.normal = normalize(P.normal);
17 |    }
18 |    return result;
}

```

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 $\vec{N} = (0, 1, 0)$, $D = -2$. Los puntos de P son de la forma $(x, -2, z)$, con $x, z \in \mathbb{R}$, y a partir de los pares (x, z) definiremos qué puntos colorear de negro y cuáles de blanco. 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 7.7 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.

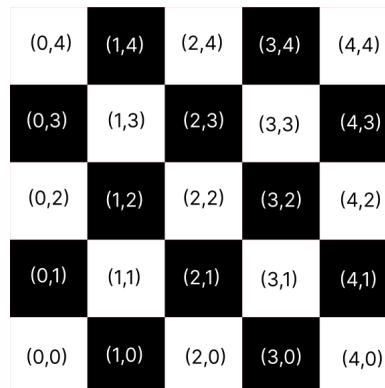


Imagen 7.7: Textura de ajedrez a partir de las partes enteras

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     float t_closest = MAX_DIST;
4     // r hits any sphere?
5     vec4 tmp_color;
6     Hit_record hr = hit_spheres_list(world, size, r, 0.0,
7         t_closest);
8     if(hr.hit){
9         t_closest = hr.t;
10        tmp_color = vec4((hr.normal + vec3(1.0))/float(2.0),
11            1.0);
12    }
13
14    // r hits the plane?
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));
25        if(modulus == 0) // Suma par
26            tmp_color = vec4(1.0, 1.0, 1.0, 1.0);
27        else // Suma impar
28            tmp_color = vec4(0.0, 0.0, 0.0, 1.0);
29    }
30    // If r hits any surface
31    if(t_closest < MAX_DIST) return tmp_color;
32    // r does not hit any surface
33    // Background code ...
34 }
```

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 7.4

```

1 // ...
2 // R is the Ray
3 // world is the Sphere array
4
5 // Plane
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);
```

7.5. 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 al principio 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 el observador se mantiene fijo carece de interés

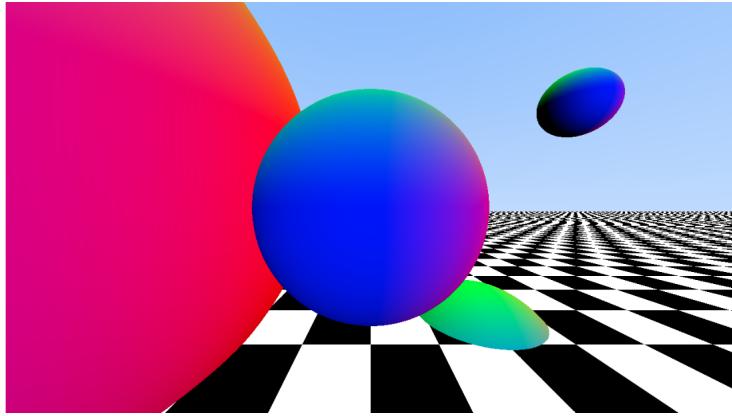


Imagen 7.8: Escena compuesta por esferas y un plano

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, es decir, la posición del espectador; y *lookat* al punto sobre el cual se fija la mirada. Nótese que realmente el punto *lookat* puede ser cualquiera situado en la semirecta que une al observador y el centro del plano, no tiene por qué estar incluido como tal en el 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 7.9. Como nuestro plano de proyección no es cuadrado, este ángulo es distinto en vertical y en horizontal, pero del ratio $16 : 9$ 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 7.9. Esto implica, asumiendo una distancia focal de 1, que la altura del plano de proyección (*viewport_height*) es $PP_{height} = 2h$, por lo que la anchura es $PP_{width} = AR \cdot PP_{height}$.

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 (vup)*, el cual define ese ‘arriba’ para la cámara. Normalmente, por convención se hace la elección $vup = (0, 1, 0)$. Mediante el vector $VD = lookat - lookfrom$ (*view direction*) y el vector *vup* podemos crear un sistema de referencia ortonormal del plano ortogonal a *VD* que pasa por el punto *lookfrom* 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

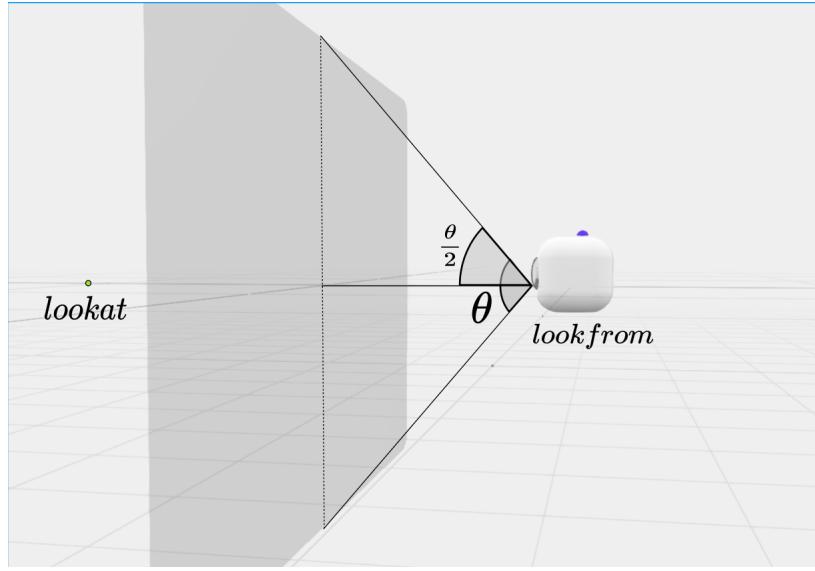


Imagen 7.9: Field Of View

vectores

$$w = \frac{lookat - lookfrom}{\|lookat - lookfrom\|},$$

$$u = \frac{vup \times w}{vup \times w},$$

$$v = w \times u,$$

de forma que los vectores u, v son, por su propia definición ortogonales a VD 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 7.2, mientras que w por su parte toma el relevo del vector $-(0, 0, 1)$. Fíjese para mejor comprensión en la imagen 7.10.

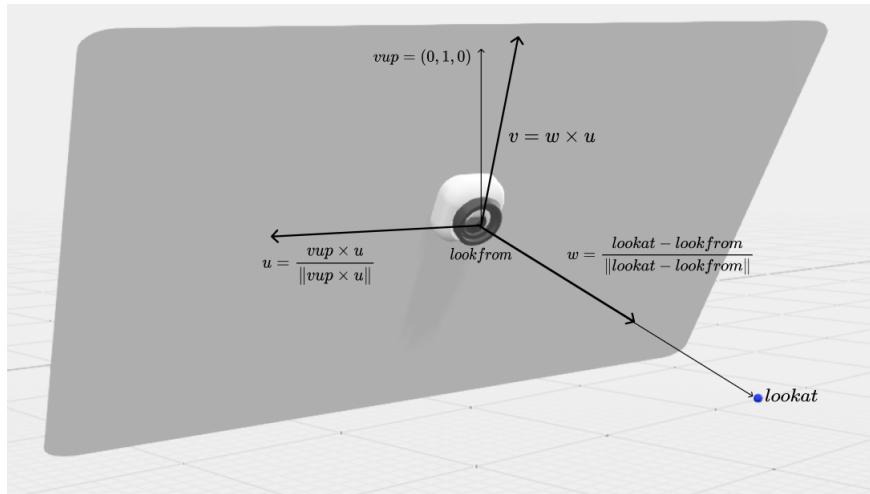


Imagen 7.10: Vectores que forman una base del plano de proyección

Ahora recordamos que para crear el rayo en la sección 7.2 calculamos la esquina inferior izquierda, transformamos las coordenadas de dispositivo en coordenadas de

mundo del plano de proyección en $[0, PP_{width}] \times [0, PP_{height}]$ 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 *lookfrom*, el punto *lookat*, el vector *vup* (usualmente $(0,1,0)$), el ángulo *fov* y el ratio 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
4             PP_width
5     vec3 vertical;        // Vector vertical de modulo
6             PP_height
7     vec3 lower_left_corner; // Punto situado en la esquina
8 };

```

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 7.11.

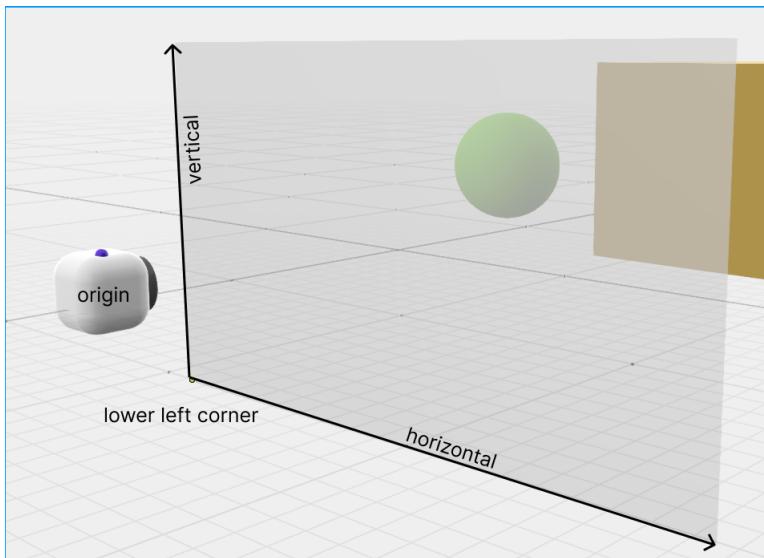


Imagen 7.11: Representación gráfica de los campos de ‘Camera’

Es claro que el origen es el punto *lookfrom*, el vector horizontal es $PP_{width} \cdot u$ y el vector vertical es $PP_{height} \cdot v$. Por tanto, la esquina inferior izquierda sería

$$LLC = lookfrom - \frac{PP_{width}}{2} \cdot u - \frac{PP_{height}}{2} \cdot v - w.$$

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); // Vertical FOV
8     float h = tan(theta/2.0);
9     float viewport_height = 2.0*h*focal_length;
10    float viewport_width = aspect_ratio * viewport_height;
11
12    vec3 w = normalize(lookfrom - lookat);
13    vec3 u = normalize(cross(vup,w));
14    vec3 v = cross(w,u);
15
16    cam.origin = lookfrom;
17    cam.horizontal = viewport_width * u;
18    cam.vertical = viewport_height * v;
19    cam.lower_left_corner = cam.origin
20        - cam.horizontal/float(2.0)
21        - cam.vertical/float(2.0)
22        - w;
23
24    return cam;
25 }

```

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 7.2 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 $fov = 90^\circ$, el vector $vup = (0, 1, 0)$ y el ratio 16:9, pero como nos gustaría poder movernos con libertad por la escena, declararemos dos variables `uniform` que representen el punto `lookfrom` y el punto `lookat`, a las cuales podremos asignarle variables dinámicamente desde JavaScript y dar la sensación de movimiento.

```

1
2 uniform vec3 u_lookfrom;      // lookfrom point
3 uniform vec3 u_lookat;       // lookat point
4
5 // ...
6 // aspect_ratio = 16/9
7 // image_width = 1280 pixels
8 // image_height = 720 pixels

```

```

9
10 // CAMERA
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 // [0,1] normalized device coordinates
17 vec2 uv = gl_FragCoord.xy / vec2(image_width, image_height);
18 float u = uv.x;
19 float v = uv.y;
20
21 Ray r = get_ray(cam, u, v);

```

Por ejemplo, si fijamos $lookfrom = (1, 1, 1)$ y $lookat = (0, 0, 0)$ obtenemos la imagen 7.12 (a), y con $lookfrom = (5, 5, -5)$, $lookat = (1, 2, -5)$ la imagen 7.12 (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’.

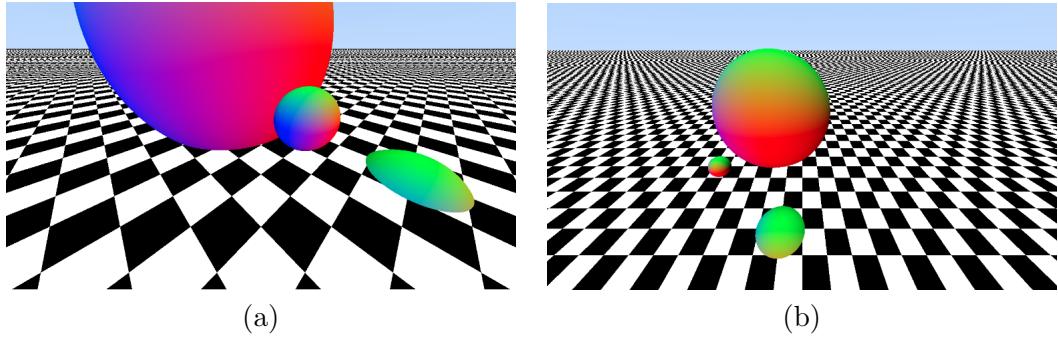


Imagen 7.12: Escena 7.8 desde otros puntos de vista

7.6. 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 finalizar este capítulo de introducción al Ray-Tracing, dotaremos a la escena de fuentes de luz y añadiremos a los objetos que componen la escena un material que nos permita observar sus irregularidades más allá del material normal que definimos en la sección 7.4.1 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 tripla 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.

7.6.1. 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.

$$I_a = I_{la}k_a, \quad (7.5)$$

7.6.2. 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



Imagen 7.13: La calima: un ejemplo del efecto de la componente ambiental

misma cantidad de luz que incide es la que se refleja en todas las direcciones, 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 7.14). Por tanto,

$$I_{dp} = I_p k_d (\vec{N} \cdot \vec{L}_p) \quad (7.6)$$

La componente difusa final es la suma de las componentes difusas calculadas para cada una de las fuentes de luz.

$$I_d = \sum_{p=1}^k I_p k_d (\vec{N} \cdot \vec{L}_p) \quad (7.7)$$

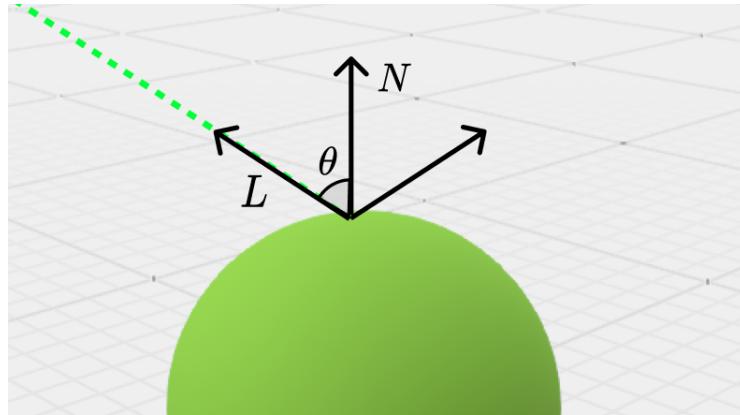


Imagen 7.14: Esquema de los vectores utilizados en la componente difusa

Fíjémonos que este cálculo es independiente de la posición de la cámara, solo depende de la superficie y la fuente de luz.

7.6.3. Componente especular

Esta componente representa a las reflexiones directas de la fuente de luz sobre un objeto con brillo. Es con esta componente mediante la que conseguimos efectos

brillantes y metalizados. La percepción de la iluminación especular depende ahora sí de la posición del observador 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 la posición del espectador, 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 7.15 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.

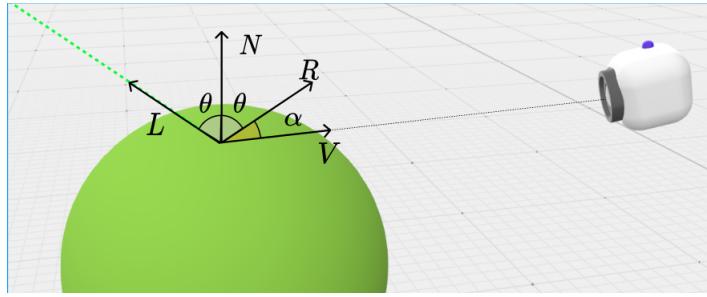


Imagen 7.15: Esquema de los vectores utilizados en la componente especular

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

$$I_{sp} = I_p k_s (\vec{R}_p \cdot \vec{V})^n \quad (7.8)$$

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.

$$I_s = \sum_{p=1}^k I_p k_s (\vec{R}_p \cdot \vec{V})^n \quad (7.9)$$

7.6.4. Evaluación del modelo de iluminación

Una vez se han explicado las tres componentes y cómo se calculan podemos evaluar el modelo de iluminación completo con solo sumar las tres intensidades calculadas.

$$\begin{aligned} I &= I_a + I_d + I_p \\ &= \frac{1}{k} \sum_{p=1}^k I_p k_a + \sum_{p=1}^k I_p k_d (\vec{N} \cdot \vec{L}_p) + \sum_{p=1}^k I_p k_s (\vec{R}_p \cdot \vec{V})^n \\ &= \sum_{p=1}^k \left(\frac{1}{k} I_p k_a + I_p k_d (\vec{N} \cdot \vec{L}_p) + I_p k_s (\vec{R}_p \cdot \vec{V})^n \right) \end{aligned} \quad (7.10)$$

Sin embargo, 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. Esto se traduce en que el coseno del ángulo θ sea negativo, es decir, que $(\vec{N} \cdot \vec{L}_p) < 0$. Al no existir luz incidente el material no puede irradiar ningún

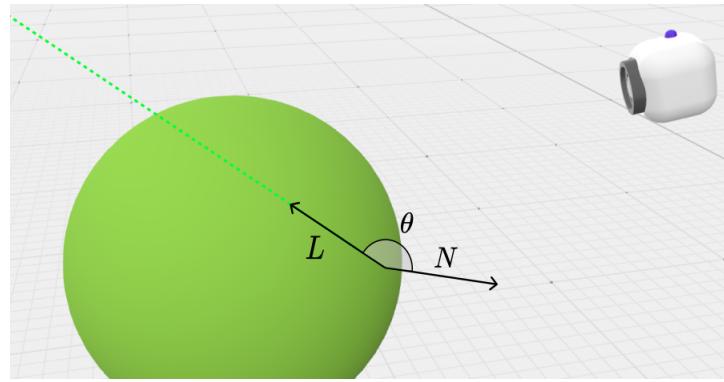


Imagen 7.16: Punto donde no se aplica componente difusa ni especular

tipo de luz, pues no hay luz que incida, produciendo efectos de sombras, donde únicamente afecta la componente ambiental. Fíjese en la imagen 7.16

A modo de adelanto, pero también con el objetivo de aclarar conceptos, obsérvese cómo afecta cada componente del modelo. En las imágenes 7.17 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.

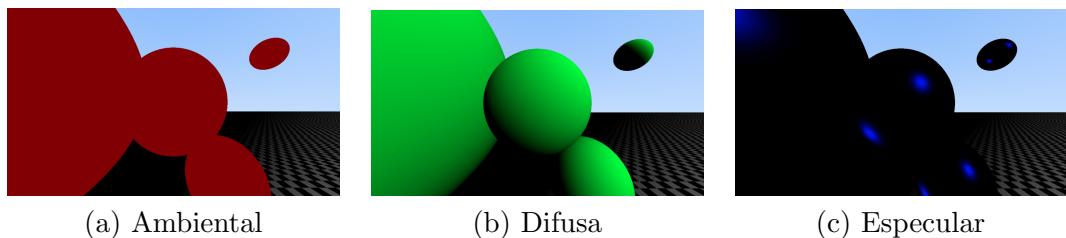


Imagen 7.17: Componentes aisladas del modelo de Phong

Y ahora observemos una imagen en la que se aprecian las tres componentes a la vez: la imagen 7.18. 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.

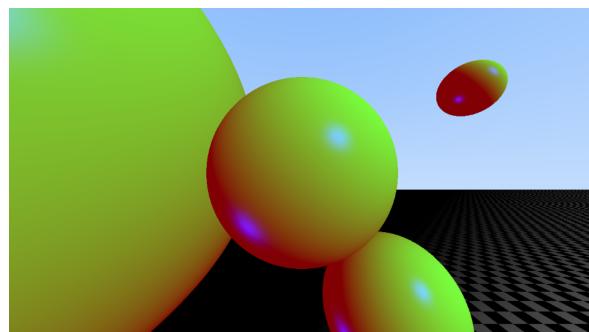


Imagen 7.18: Acción de todas las componentes

7.6.5. Implementación del modelo de Phong

Veamos ahora cómo a partir de la teoría introducida podemos obtener resultados como los de las imágenes 7.17 y 7.18. El elemento básico de un modelo de iluminación es, como no podría ser de otro modo, la fuente de luz. Una fuente de luz ya dijimos que se compone de un vector director y una tripla 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;    // Direccion 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 tripletas RGB. 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;      // Reflecitividad ambiental
3 |     vec4 kd;      // Reflecitividad difusa
4 |     vec4 ks;      // Reflecitividad 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.

Con estas variaciones ya podríamos implementar una función que evalue el modelo de iluminación completo a partir de la ecuación (7.10). 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 y la posición del espectador. Toda esta información puede ser recogida en un array de objetos `Directional_light`, en la estructura `Hit_record` que almacena la información del choque y en la variable global `u_lookfrom`. Presentamos por tanto el código de dicha función.

```

1 | vec4 evaluate_lighting_model(
2 |     Directional_light lights[ARRAY_TAM],
3 |     int num_lights, Hit_record hr ) {
4 |
5 |     vec4 color_average = vec4(0.0, 0.0, 0.0, 1.0);
6 |     Material mat = hr.mat; // Material
7 |     Directional_light light;
8 |     vec3 light_dir;

```

```

9 |     vec3 view_dir = normalize(u_lookfrom - hr.p); // V
10|     vec3 normal = normalize(hr.normal); // N
11|
12|     vec4 ambient, diffuse, specular;
13|     ambient = vec4(0.0, 0.0, 0.0, 1.0);
14|     diffuse = vec4(0.0, 0.0, 0.0, 1.0);
15|     specular = vec4(0.0, 0.0, 0.0, 1.0);
16|
17|     if(num_lights > 0){
18|         for(int i = 0; i < ARRAY_TAM; i++){
19|             if(i == num_lights) break;
20|             light = lights[i];
21|             color_average += light.color; // Suma de las
22|                                         intensidades
23|             light_dir = normalize(light.dir); // L_p
24|             float cos_theta = max(0.0, dot(normal, light_dir))
25|                                         ;
26|
27|             // Solo si la luz es visible desde ese punto
28|             if(cos_theta > 0.0) {
29|                 // R_p
30|                 vec3 reflection_dir =
31|                     reflect(-light_dir, normal);
32|
33|                 diffuse += mat.kd * light.color * cos_theta;
34|                 specular += mat.ks * light.color * pow(
35|                     max(0.0, dot(reflection_dir, view_dir)), mat.sh);
36|             }
37|             color_average /= float(num_lights);
38|             ambient = mat.ka * color_average;
39|         }
40|         return ambient + diffuse + specular;
41|     }

```

Lo siguiente es editar, aunque muy brevemente, 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 la reflectancia especular de ese material, asignándole 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 hits any sphere?
6|     // ...
7|     if(hr.hit){
8|         // ...
9|         tmp_color = evaluate_lighting_model(lights,

```

```

10         num_lights, hr);
11     }
12
13     // r hits the plane?
14     // ...
15     if(hr.hit){
16         // ...
17         if(modulus == 0)
18             hr.mat.ks = vec4(0.7, 0.7, 0.7, 1.0);
19         else
20             hr.mat.ks = vec4(0.0, 0.0, 0.0, 1.0);
21         tmp_color = evaluate_lighting_model(lights,
22             num_lights, hr);
23     }
24     // ...
25 }
```

Y ya por último tan solo tenemos que inicializar las luces y los materiales. En nuestros ejemplos estamos utilizando una luz blanca cuya dirección es $(1, 1, 1)$ y una luz azul cuya dirección es $(-1, -1, 0)$. Respecto a los materiales, el del plano únicamente tiene la componente 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 // ...
8
9 // Materials
10 Material mat, ground_material;
11 mat.ka = u_ka;
12 mat.kd = u_kd;
13 mat.ks = u_ks;
14 mat.sh = u_sh;
15
16 ground_material.ka = vec4(0.0, 0.0, 0.0, 1.0);
17 ground_material.ks = vec4(0.0, 0.0, 0.0, 1.0);
18 ground_material.sh = 1.0;
19
20 // Spheres
21 // ...
22 S1.mat = mat; S2.mat = mat;
23 S3.mat = mat; S4.mat = mat;
24 // ...
25
26 // Plane
27 // ...
28 P.mat = ground_material;
29 // ...
30
31 // Lights
32 Directional_light lights[ARRAY_TAM];
33 int num_lights = 2;
```

```

34 | Directional_light l1, l2;
35 | l1.color = vec4(1.0, 1.0, 1.0, 1.0);
36 | l2.color = vec4(0.0, 0.0, 1.0, 1.0);
37 | l1.dir = vec3(1.0, 1.0, 1.0);
38 | l2.dir = vec3(-1.0, -1.0, 0.0);
39 | lights[0] = l1; lights[1] = l2;
40 | // ...
41 |
42 | gl_FragColor = ray_color(r, world, num_spheres,
43 |                           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 ejemplos tenemos las imágenes que se han presentado al inicio de esta sección, aunque 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 e iluminación.

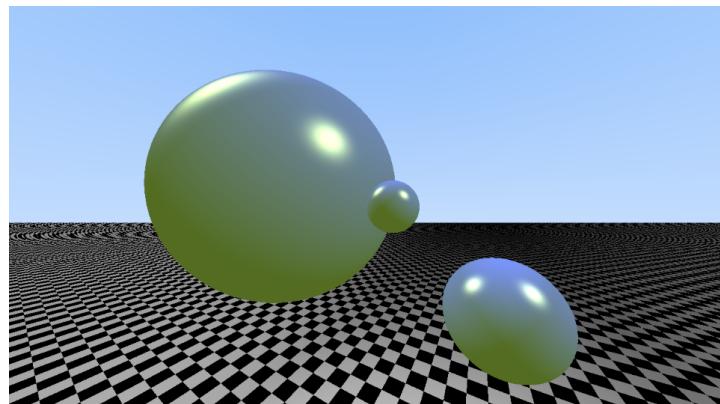


Imagen 7.19: Imagen final del capítulo 7

CAPÍTULO 8

VISUALIZACIÓN DE FRACTALES EN 3D

En el capítulo 6 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 de los tres capítulos anteriores es aplicar técnicas avanzadas de renderizado para poder visualizar fractales en tres dimensiones. Recordamos que explicamos en la introducción del capítulo 7 que la mejor forma de renderizar 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 7 para graficar fractales tridimensionales.

Para ello, antes aún debemos asentar una serie de conceptos que son los que nos ayudarán a cumplir nuestro objetivo: el algoritmo *Ray-Marching* y las conocidas como *Signed Distance Functions* (SDFs). Con estos dos elementos combinados conseguiremos ver hermosas figuras fractales.

8.1. El algoritmo Ray-Marching

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 7.4.1 y 7.4.3. 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. Aunque no tengamos una forma de calcular analíticamente la intersección, si contamos con funciones que nos estiman, fijado un punto, a qué distancia se halla este punto del fractal mediante una cota, las conocidas como SDFs (sección 8.2). De igual manera, si fijamos un punto $p \in \mathbb{R}^3$, podemos calcular a qué distancia se encuentra dicho punto de una esfera o de un plano.

El algoritmo *Ray-Marching*, también conocido como *Sphere-Tracing* es una técnica utilizada en Ray-Tracing y que se basa en estimar la distancia a la que se encuentra un punto de cada una de las superficies que componen la escena y avanzar en el rayo una distancia correspondiente a la mínima de estas distancias,

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 es lo suficientemente pequeña como para considerar que el punto interseca con la superficie. Los fundamentos matemáticos de este método y las propiedades que deben cumplir las funciones estimadoras se pueden encontrar en [31].

Pongamos un ejemplo. Supongamos que tenemos una escena con una esfera tangente al plano horizontal $y = 0$, un rayo R fijo y queremos aplicar Ray-Marching para encontrar la intersección.

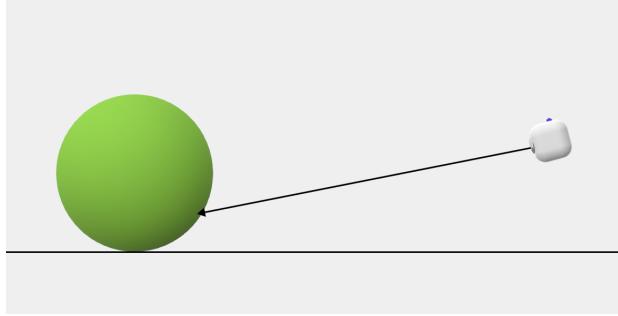
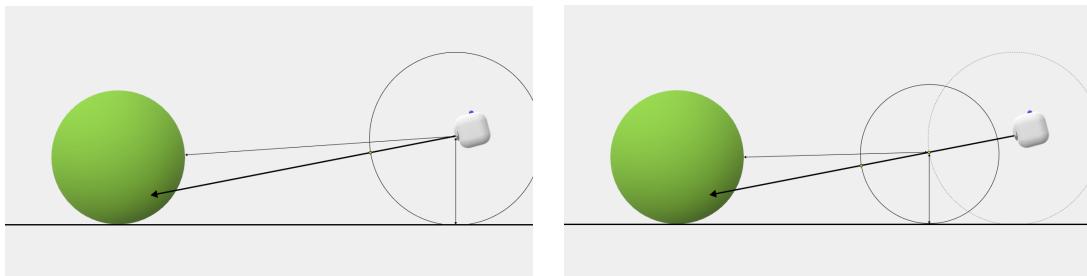


Imagen 8.1: Situación a la que aplicar Ray-Marching

La distancia de un punto p a una esfera S de centro c y radio r es

$$d(p, S) = \|p - c\| - r \quad (8.1)$$

y la distancia de un punto al plano $y = 0$ es simplemente su componente y . Por tanto, partiendo del origen, calculamos la distancia a la esfera y la distancia al plano, vemos que es menor la distancia al plano, por lo que avanzamos en el rayo una distancia igual a la distancia que había al plano (imagen 8.2 (a)). Seguidamente repetimos la operación: volvemos a calcular la distancia a la esfera y al plano. De nuevo es menor la distancia al plano, por lo que avanzamos en el rayo la misma distancia que había en el plano (imagen 8.2 (b)).



(a) Primera iteración

(b) Segunda iteración

Imagen 8.2: Dos primeras iteraciones de Ray Marching

Si repetimos este proceso indefinidamente, llegará el momento en el que la distancia a la esfera será tan pequeña que consideraremos que el punto está en la esfera y habremos encontrado la intersección (imagen 8.3 (a)). En caso de que no exista ninguna intersección, el algoritmo seguirá avanzando en el rayo, pero al no intersecar con ninguna superficie se avanzará indefinidamente (imagen 8.3 (b)),

por lo que hay que fijar una distancia máxima recorrida o un número máximo de iteraciones como parámetro del algoritmo.

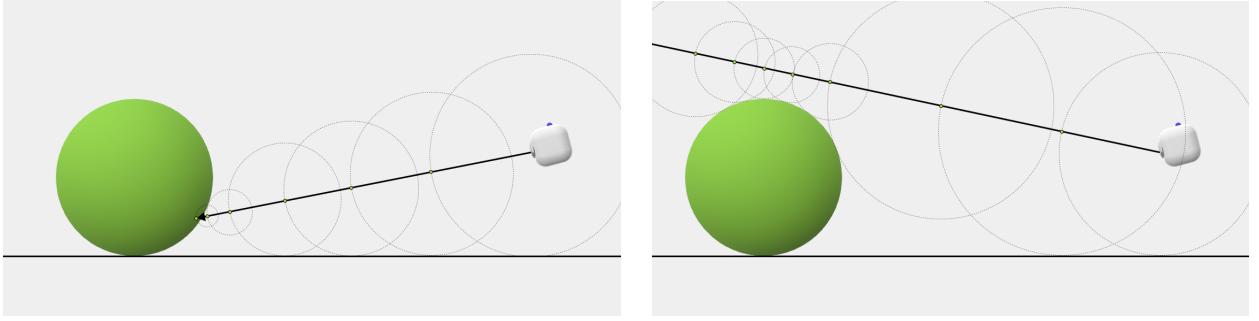


Imagen 8.3: Posibles estados finales del algoritmo Ray-Marching

Por tanto, el algoritmo, si consideramos un conjunto de *objetos*, un rato $R(t) = p_0 + \vec{v}t$ y un número máximo de avances en el rayo antes de decidir que no existe intersección *MAX_STEPS*, se describiría como indica el algoritmo 1.

Algorithm 1 Ray-Marching

```

 $p \leftarrow lookfrom$ 
 $steps \leftarrow 0$ 
while  $steps < MAX\_STEPS$  do     $\triangleright$  También se puede fijar una distancia máxima
     $min\_dist \leftarrow MAX\_DIST$ 
     $i \leftarrow 0$ 
    while  $i < num\_objects$  do           $\triangleright$  Cálculo de la mínima distancia
         $dist \leftarrow distancia(p, objects[i])$ 
        if  $dist < min\_dist$  then
             $min\_dist \leftarrow dist$ 
             $index \leftarrow i$ 
        end if
         $i ++$ 
    end while
    if  $min\_dist < \varepsilon$  then
        Hay intersección con  $objects[index]$ 
    else
         $p \leftarrow p + \vec{v} \cdot min\_dist$            $\triangleright$  Avanzamos en el rayo
    end if
     $steps ++$ 
end while
No existe intersección

```

Observación 8.1.1. 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 $|v| \neq 1$ se avanzaría una distancia distinta y el algoritmo no funcionaría correctamente.

8.1.1. Implementación de Ray-Marching 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 Ray-Marching para calcular las intersecciones. Realmente el resultado de estas modificaciones debería ser el mismo que en el final del capítulo anterior, pero ahora calcularemos las intersecciones de forma distinta.

Lo primero que necesitaremos es una función que calcule la distancia de un punto a una esfera S . Esto es tan sencillo como implementar una función que calcule la fórmula 8.1.

```

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, podríamos aprovechar que nuestro plano es el $y = -2$ y simplemente devolver la componente y del punto más dos. Sin embargo, no cuesta tanto calcular la distancia de un punto a un plano arbitrario $Ax + By + Cz = D$, siendo $\vec{N} = (A, B, C)$ el vector normal al plano. El punto 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.

$$\begin{aligned}\vec{N} \cdot S(t) &= D \\ \vec{N} \cdot (p + \vec{N}t) &= D \\ t &= \frac{D - \vec{N} \cdot p}{|\vec{N}|^2}\end{aligned}\tag{8.2}$$

Por tanto la distancia entre punto y plano es la longitud del vector que une a p con $S(t)$, siendo este t el recién calculado. 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(
5         P.normal, P.normal);
6     vec3 closest_point = p + t_interseccion * P.normal;
7     return length(p - closest_point);
```

Seguidamente necesitamos fijar dos constantes: el número máximo de iteraciones que se darán en el algoritmo 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.

```

1 uniform float u_epsilon;
2 // ...
3 #define MAX_STEPS 1000
```

Y ya tenemos todo preparado para implementar el Ray-Marching. La función que dado un rayo calcula posibles intersecciones con los cuerpos y asigna un color en función 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 7.4.3 aplicaremos Ray-Marching. 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; // Hit_record structure
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    // Ray Marching
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
42            hr.hit = true;
43            hr.t = current_t;
44            hr.p = ray_at(r, hr.t);
45
46            if(object_index == num_spheres){ // Suelo

```

```

47         // Codigo de interseccion con el suelo ...
48     }
49
50     else {      // Una de las esferas
51         // Codigo de interseccion con S_hit
52     }
53 }
54
55 // Si no hay interseccion, avanzamos en el rayo
56 current_t += closest_dist;
57 p = ray_at(r, current_t);
58
59 // Si estamos muy lejos acabamos el bucle
60 if(current_t >= MAX_DIST) break;
61 }
62 // No hay interseccion
63 // Background code ...
64 }
```

Fíjese que concuerda con la estructura que describe el algoritmo 1. Hecha esta modificación, el resultado fijando ciertos parámetros dinámicamente es el mostrado en la imagen 8.4. Tal y como adelantamos, el resultado es igual que cuando calculábamos las intersecciones exactas, ya que realmente lo único que cambia es la forma de calcular las intersecciones, nada estrictamente visual.

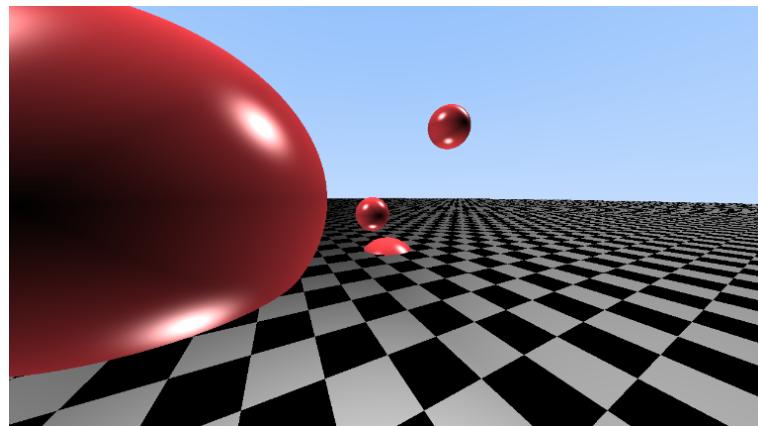


Imagen 8.4: Escena tras implementar Ray-Marching

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 8.5 los resultados obtenidos al no normalizar el vector director al crear el rayo.

8.1.2. Comentarios sobre Ray-Marching

Una vez hemos introducido e implementado el algoritmo Ray-Marching, 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 ray-marching 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

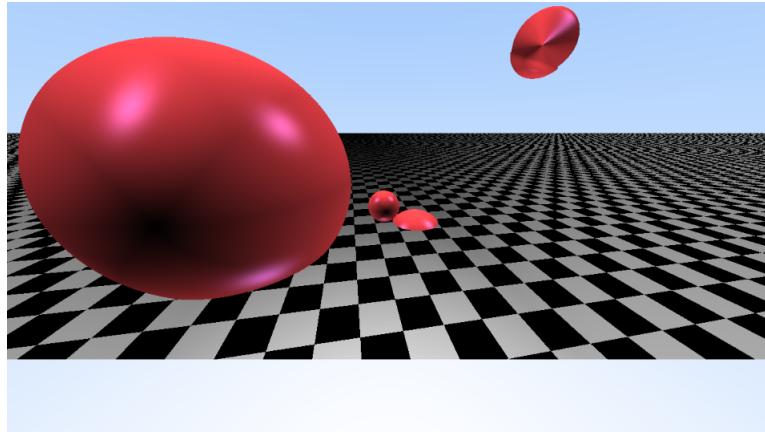


Imagen 8.5: Escena utilizando vectores no unitarios en los rayos

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, ray-marching sólo necesita, para cada cuerpo, una función que aproxime suficientemente bien la distancia a un conjunto. A partir de un rayo y dichas funciones se puede aplicar ray-marching, 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 ray-marching 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 8.4 se ha utilizado $\varepsilon = 0.001$ y nos da un resultado muy bueno, pero obsérvese en la imagen 8.6 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. Sin embargo 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.

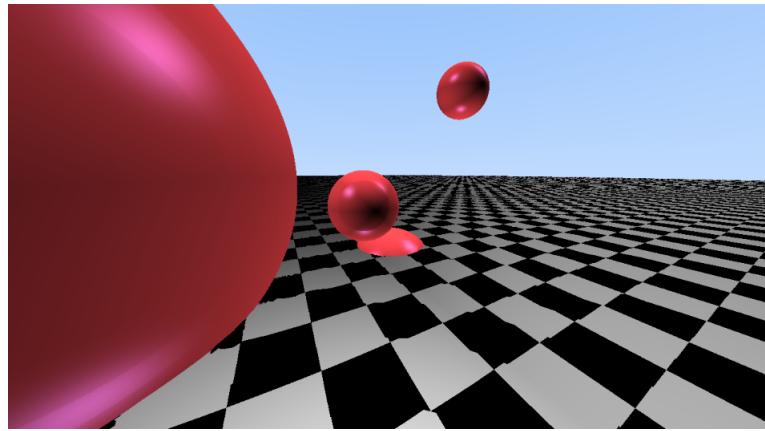


Imagen 8.6: Escena renderizada utilizando $\varepsilon = 0.1$

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 8.7, donde se ha utilizado `MAX_STEPS=100`. Fíjese en las fronteras entre el horizonte y las esferas, o en los bordes de las esferas.

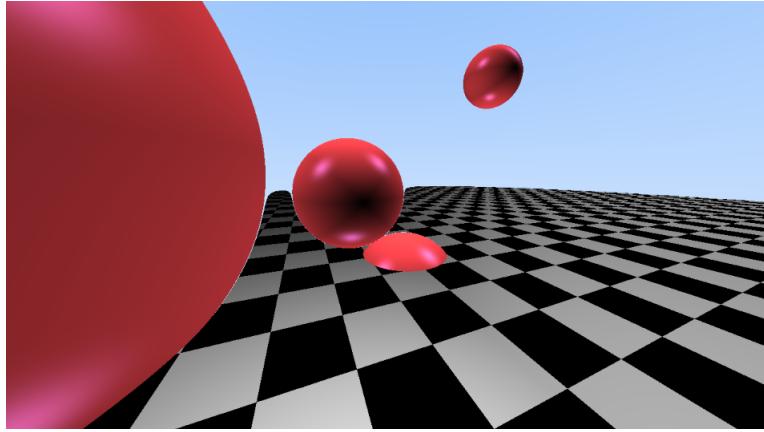


Imagen 8.7: Escena renderizada utilizando `MAX_STEPS=100`

De igual forma, menos iteraciones implica más rapidez, pero peores resultados, por lo que es necesario fijar un valor correcto.

8.2. Signed Distance Functions (SDFs)

Durante toda la sección anterior hemos hablado de que para aplicar ray-marching es necesario que cada objeto que componga la escena cuente con una función que estime la distancia de un punto cualquiera de \mathbb{R}^3 a su superficie. En esta sección introduciremos el concepto de SDF y pondremos un ejemplo sencillo con una esfera.

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 ($f(x) < 0 \quad \forall x \in A$), 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 8.2.1 (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

$$|f(x)| \leq d(x, f^{-1}(0)) \quad \forall x \in \mathbb{R}^3 \tag{8.3}$$

Si se tiene la igualdad en (8.3), entonces f es una ‘**Signed Distance Function**’ (función distancia con signo).

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 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 8.8). En [31, Table 1] podemos encontrar una lista de primitivas con referencias a sus correspondientes SDFs.

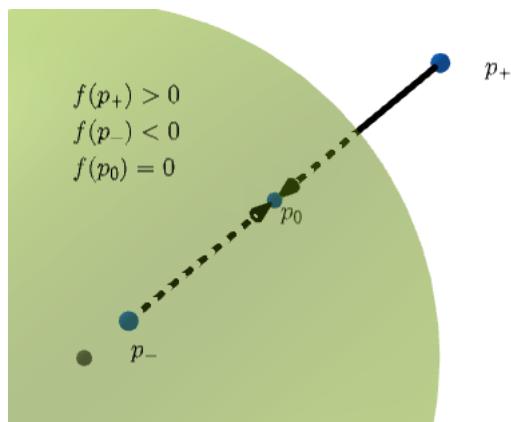


Imagen 8.8: Ejemplos de puntos con distintos valores de la SDF de una esfera

8.3. Visualización tridimensional de conjuntos de Julia

Una vez tenemos definido el algoritmo que usaremos para la visualización de fractales, que es el algoritmo de ray-marching 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 6, 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

$$E_c = \{z_0 \in \mathbb{C} : \{|P_c^n(z_0)|\} \rightarrow \infty\} \quad (8.4)$$

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 [33], 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:

$$q = x \cdot i + y \cdot j + z \cdot k + w \cong (x, y, z, w) \in \mathbb{R}^4. \quad (8.5)$$

Nótese que estamos denotando como componente real a w , la última de la terna (x, y, z, w) . Las unidades imaginarias se multiplican entre ellas siguiendo las siguientes reglas:

$$\begin{aligned} ij &= k & jk &= i & ki &= j \\ ji &= -k & kj &= -i & ik &= -j \\ i^2 &= j^2 = k^2 = -1. \end{aligned} \quad (8.6)$$

Lo cual nos da a entender que los cuaternios tienen estructura de anillo no conmutativo. Una forma de calcular el producto de dos cuaternios arbitrarios $q = q_x \cdot i + q_y \cdot j + q_z \cdot k + q_w \cong (q_x, q_y, q_z, q_w)$ y $q' = q'_x \cdot i + q'_y \cdot j + q'_z \cdot k + q'_w \cong (q'_x, q'_y, q'_z, q'_w)$ es expandiendo la expresión

$$qq' = (q_x \cdot i + q_y \cdot j + q_z \cdot k + q_w)(q'_x \cdot i + q'_y \cdot j + q'_z \cdot k + q'_w) \quad (8.7)$$

Si utilizamos su expresión como ternas de \mathbb{R}^4 , denotando $q = (q_{xyz}, q_w), q' = (q'_{xyz}, q'_w)$, agrupando términos y aplicando las relaciones 8.6 el resultado puede expresarse mediante productos vectoriales y escalares:

$$\begin{aligned} (qq')_{xyz} &= q_{xyz} \times q'_{xyz} + q_w q'_{xyz} + q'_w q_{xyz} \\ (qq)_w &= q_w q'_w - (q_{xyz} \cdot q'_{xyz}) \end{aligned} \quad (8.8)$$

Aplicando estas relaciones podemos deducir el cuadrado de un cuaternion con tan solo multiplicar un cuaternion q por sí mismo, obteniendo

$$\begin{aligned} (q^2)_{xyz} &= 2q_w q_{xyz} \\ (q^2)_w &= q_w^2 - q_{xyz} \cdot q_{xyz} \end{aligned} \quad (8.9)$$

que realmente es una simplificación de las ecuaciones (8.8) 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 $c \in \mathbb{H}$:

$$\begin{aligned} P_c : \mathbb{H} &\longrightarrow \mathbb{H} \\ q &\longmapsto q^2 + c \end{aligned} \quad (8.10)$$

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

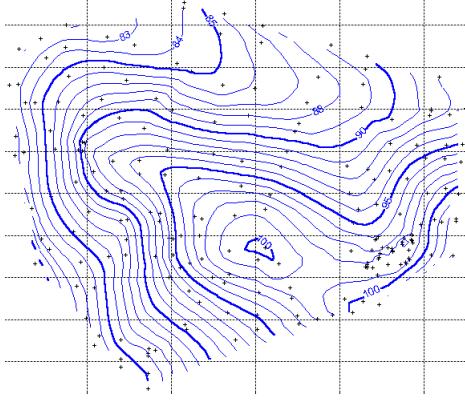


Imagen 8.9: Representación de una montaña con curvas de nivel

en topografía (imagen 8.9), los cuales fijando una altura 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, pudiendo modificar después la dimensión que hemos fijado.

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 manera más natural es identificar x con la parte real y la segunda y tercera componente con las dos primeras componentes imaginarias. Es decir,

$$p = (x, y, z) \cong q = x + yi + zj \cong (y, z, 0, x) \quad (8.11)$$

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 nunca ocurra durante el proceso de ray-marching, pues se evaluarán puntos que están en el conjunto de puntos de escape, en búsqueda de esta frontera. Es decir, iteramos el punto con una amplia posibilidad de que antes o después la n -ésima iterada tenga un módulo grande. Denotamos como $q_n = P_c^n(q)$ a esta n -ésima iterada. A partir de este valor, definimos la SDF del conjunto de Julia como

$$d_c : \mathbb{R}^3 \longrightarrow \mathbb{R}$$

$$d_c(p) = \frac{|q_n| \log |q_n|}{2|q'_n|} \quad (8.12)$$

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 $|q'_{i+1}| = 2|q_i||q'_i|$.

Intuitivamente, fíjese que realmente es una forma iterativa de aplicar la derivada de (8.10)

Es difícil explicar y entender la génesis de esta fórmula, aunque los detalles matemáticos concretos pueden encontrarse en [36, Capítulo 8].

Es decir, iterando el punto inicial y aplicando la transformación (8.12) podemos estimar a qué distancia se sitúa el punto del conjunto de Julia \mathcal{J}_c .

8.3.1. 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. No existen formas de calcular de forma exacta esta normal, pero en realidad nosotros estamos fijando una resolución que limita el nivel de detalle de la figura, 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 ray-marching. 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.

Algorithm 2 Cálculo de la normal mediante el gradiente de la SDF

```

 $h \leftarrow \varepsilon$ 
 $g_{x+} \leftarrow p + (h, 0, 0)$ 
 $g_{x-} \leftarrow p - (h, 0, 0)$ 
 $g_{y+} \leftarrow p + (0, h, 0)$ 
 $g_{y-} \leftarrow p - (0, h, 0)$ 
 $g_{z+} \leftarrow p + (0, 0, h)$ 
 $g_{z-} \leftarrow p - (0, 0, h)$ 
 $\nabla d_{c_x} \leftarrow d_c(g_{x+}) - d_c(g_{x-})$ 
 $\nabla d_{c_y} \leftarrow d_c(g_{y+}) - d_c(g_{y-})$ 
 $\nabla d_{c_z} \leftarrow d_c(g_{z+}) - d_c(g_{z-})$ 
return  $\frac{(\nabla d_{c_x}, \nabla d_{c_y}, \nabla d_{c_z})}{\|(\nabla d_{c_x}, \nabla d_{c_y}, \nabla d_{c_z})\|}$ 
```

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.

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 la normal a la superficie en el punto p es la normalización del vector $\sum_{i=0}^3 k_i d_c(p_i)$. Consultese [37] si se desea tener una justificación del funcionamiento de este método.

Algorithm 3 Técnica del tetraedro para el cálculo de normales

```

 $h \leftarrow \varepsilon$ 
 $p_0 \leftarrow p + h(1, -1, -1)$ 
 $p_1 \leftarrow p + h(-1, -1, 1)$ 
 $p_2 \leftarrow p + h(-1, 1, -1)$ 
 $p_3 \leftarrow p + h(1, 1, 1)$ 
 $N \leftarrow \sum_{i=0}^3 k_i d_c(p_i)$ 
return  $\frac{N}{\|N\|}$ 
```

8.3.2. 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 cuaternio 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 cuaternio.

```

1 | vec4 q; // Representa un cuaternion (x,y,z,w)
2 | // q = xi + yj + zk + w
```

Si ahora seguimos la ecuación (8.9) podemos rápidamente implementar una función que dado un cuaternion calcule su cuadrado.

```

1 | // Dado un cuaternion q, calcula su cuadrado q^2
2 | vec4 quat_square(vec4 q){
3 |     vec3 xyz = 2.0*q.w*q.xyz;
4 |     float w = q.w*q.w - dot(q.xyz, q.xyz);
5 |     return vec4(xyz, w);
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 cuaternion 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 éste mantiene su valor tras el retorno de la función. El código de la función es

```

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 | // 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 < MAX_STEPS; i++) {
7 |         dq = 2.0 * length(q) * dq;
8 |         q = quat_square(q) + c; // P_c(q) = q^2 + c
9 |         if(dot(q, q) > 256.0) break; // |q| es grande
10 |    }
```

```
11 | }
```

Fíjese que esta función itera un cuaternio mediante la función $P_c(q)$ y a la vez calcula la sucesión recurrente $|q'_{i+1}| = 2|q_n||q'_n|$. 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 cuaternio $q \in \mathbb{H}$, iterar q y $|q'|$ teniendo en cuenta que la iteración de q' comienza en 1 y aplicar la función (8.12).

```
1 float get_dist_julia(vec3 p, vec4 c) {
2     float dist;
3     vec4 q = vec4(p.y, p.z, 0.0, p.x);
4     float dq = 1.0; // q'_0 = 1
5     iterate_julia(q, dq, c);
6     float length_q = length(q);
7     return 0.5*length_q * log(length_q) / dq;
8 }
```

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 2 y 3.

```
1 // Calcula la normal al conjunto J_c en el punto p
2 vec3 calculate_normal_julia(vec3 p, vec4 c) {
3     vec3 N;
4     float h = u_epsilon;
5
6     #if NORMAL == 0
7         // Gradiente de la SDF
8         vec4 qp = vec4(p.y, p.z, 0.0, p.x);
9         float gradX, gradY, gradZ;
10        vec3 gx1 = (qp - vec4( 0.0, 0.0, 0.0, h )).wxy;
11        vec3 gx2 = (qp + vec4( 0.0, 0.0, 0.0, h )).wxy;
12        vec3 gy1 = (qp - vec4( h, 0.0, 0.0, 0.0 )).wxy;
13        vec3 gy2 = (qp + vec4( h, 0.0, 0.0, 0.0 )).wxy;
14        vec3 gz1 = (qp - vec4( 0.0, h, 0.0, 0.0 )).wxy;
15        vec3 gz2 = (qp + vec4( 0.0, h, 0.0, 0.0 )).wxy;
16
17        gradX = (get_dist_julia(gx2,c) - get_dist_julia(gx1,c))
18            /(2.0*h);
19        gradY = (get_dist_julia(gy2,c) - get_dist_julia(gy1,c))
20            /(2.0*h);
21        gradZ = (get_dist_julia(gz2,c) - get_dist_julia(gz1,c))
22            /(2.0*h);
23        N = normalize(vec3( gradX, gradY, gradZ ));
24
25    #else
26        // Método del tetraedro
27        const vec2 k = vec2(1,-1);
28        N = normalize( k.xyy*get_dist_julia( p + k.xyy*h, c ) +
29                        k.yyx*get_dist_julia( p + k.yyx*h, c ) +
30                        k.yxy*get_dist_julia( p + k.yxy*h, c ) +
31                        k.hxx*get_dist_julia( p + k.hxx*h, c ) );
32    #endif
33    return N;
34 }
```

Y ya únicamente resta sustituir en el ray-marching las SDFs y el método para

calcular normales de las esferas por el código que corresponde a los conjuntos de Julia. 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.

```

1 // Cuaterniono c fijo. Visualizaremos el conjunto J_c
2 uniform vec4 u_julia_set_constant;
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: Ground, 1: Julia
12
13    // Ray Marching
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_julia_set_constant);
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 hits the ground
34                // ...
35            else { // r hits Julia
36                hr.mat = fractal_material;
37                hr.normal = calculate_normal_julia(
38                    hr.p, u_julia_set_constant);
39                return evaluate_lighting_model(lights, num_lights,
40                    hr);
41            }
42            current_t += closest_dist;
43            p = ray_at(r, current_t);
44            if(current_t >= MAX_DIST) break;
45        }
46        // r does not hit nothing
47        // ...
48    }

```

Y con esta modificación de `ray_color` podemos ya visualizar conjuntos de Julia tridimensionales. Podemos también cambiar los parámetros del material para

observar distintas apariencias.

En las imágenes 8.10 podemos ver algunos de los resultados que, después de este largo camino, hemos podido obtener modificando los parámetros.

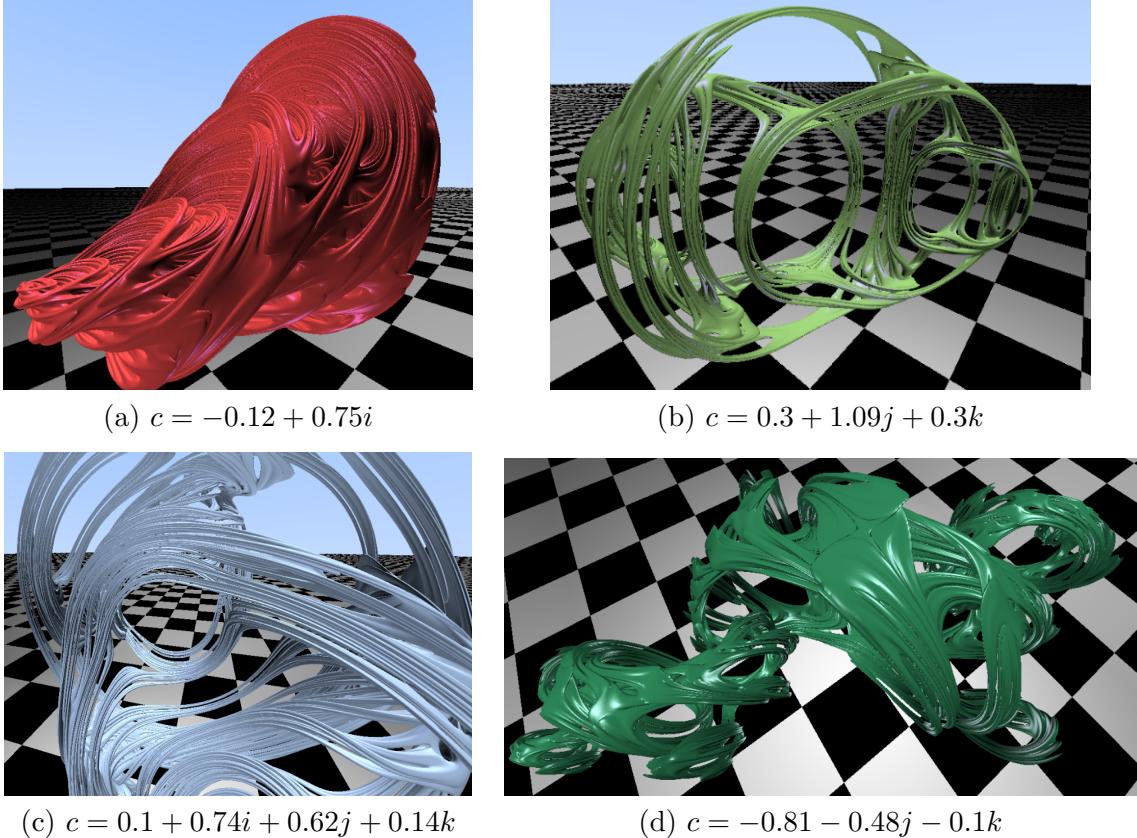


Imagen 8.10: Conjuntos de Julia 3D para distintos $c \in \mathbb{H}$

8.4. 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

$$\mathcal{M} = \{c \in \mathbb{C} : \{P_c^n(0)\} \not\rightarrow \infty\}$$

Por tanto, la manera de iterar q y q' cambia ligeramente. En lugar de utilizar q 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 $c = q$ inicial. La iteración de $|q'|$ también cambia, ya que intuitivamente la derivada de la función que iteramos es $2qq' + 1$, por lo que ahora la sucesión sería $|q'_{i+1}| = 2|q_i||q'_i| + 1$, con $q'_0 = 0$. Puede consultarse una explicación más detallada de esta diferencia en [38]. Finalmente, tras suficientes iteraciones, se obtienen los valores q_n y q'_n necesarios en la SDF.

```
1 // Itera la funcion P_c y obtiene tanto q_n como |q'_n|
```

```

2 // q: Cuaternion que se itera
3 // dq: Sucesion |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 < MAX_STEPS; i++) {
8         dq = 2.0 * length(q) * dq + 1.0;
9         q = quat_square(q) + c;
10        if(dot(q, q) > 256.0) 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 8.3.1 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 6.3.5 explicamos que añadiendo una variable `uniform` la cual en función de su valor se renderizaba 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 8.11.

Hay que reconocer que el resultado es un poco decepcionante, parecería un cuerpo generado por la revolución de la frontera del conjunto de Mandelbrot en torno al eje X .

8.5. El conjunto de Mandelbub

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

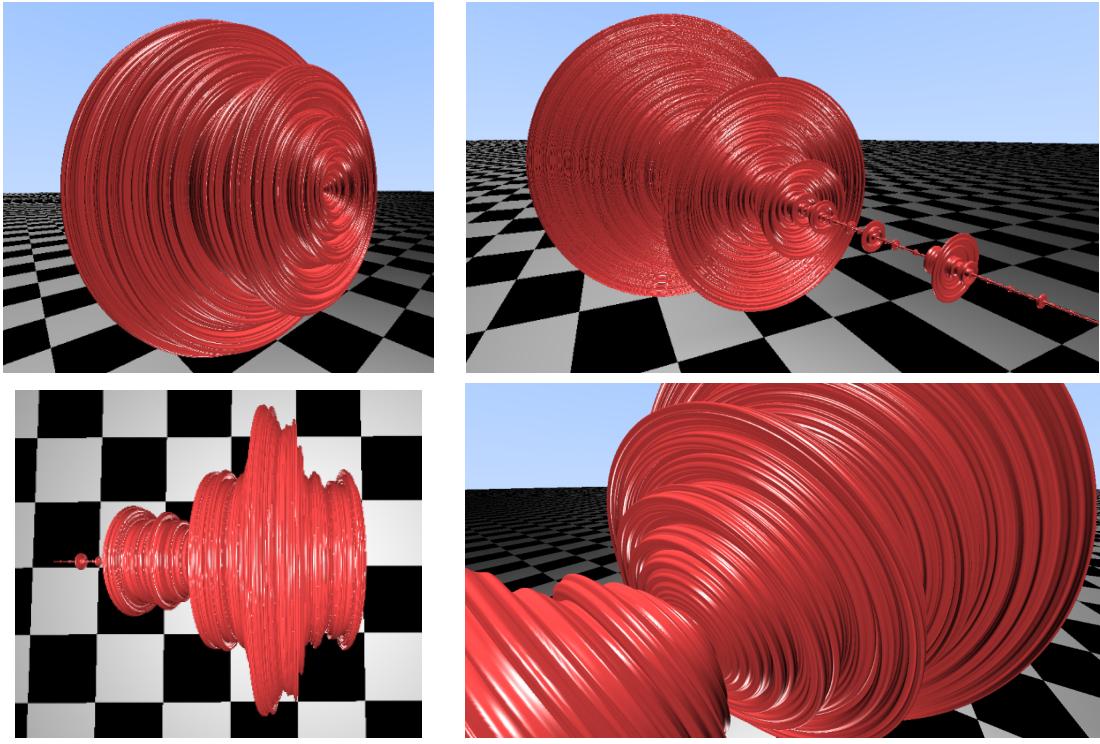


Imagen 8.11: Detalles del conjunto de Mandelbrot generalizado

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 X ϕ y el ángulo que forma con el plano $y = 0$ θ , podemos extender esta operación. El procedimiento sería pasar el punto (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

$$(x, y, z) \mapsto (r, \phi, \theta) \mapsto (r^2, 2\phi, 2\theta) \mapsto (x', y', z')$$

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

$$f = (x, y, z) \mapsto (r, \phi, \theta) \mapsto (r^8, 8\phi, 8\theta) \mapsto (x', y', z') \mapsto c + (x', y', z') \quad (8.13)$$

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 (8.13). 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

$$\begin{aligned} w_{i+1} &= f(w_i), w_0 = w \\ |w'_{i+1}| &= 8|w|^7|w'| + 1, w'_0 = 1 \end{aligned} \quad (8.14)$$

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 8.12. Consultese [39] 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 8.3.1 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 8.13), una función que itere un punto concreto y la sucesión de ‘derivadas’, y una que invoque a la iteración y calcule la estimación de la distancia y una 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 wr = sqrt(dot(w,w));
4 |     float wo = acos(w.y/wr);
5 |     float wi = atan(w.x,w.z);
6 |     // Escalado y rotación
7 |     wr = pow( wr, 8.0 );
8 |     wo = wo * 8.0;
9 |     wi = wi * 8.0;
10 |    // Vuelta a coordenadas cartesianas
11 |    w.x = wr * sin(wo)*sin(wi);
12 |    w.y = wr * cos(wo);
13 |    w.z = wr * sin(wo)*cos(wi);
14 |    return c + w;
15 |

```

Seguidamente, una función que invoque a `f_Mandelbub` para iterar `w` y `dw`. Puede llamar la atención que en este caso se rompa el bucle cuando la longitud supera el valor 2 en contraste con los 16 que se utilizan en los conjuntos de Julia y Mandelbrot, pero se ha comprobado empíricamente que de esta forma los resultados son óptimos.

```

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 < MAX_STEPS; 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 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 renderice el conjunto de Mandelbub según el valor de `u_fractal`, ya podemos visualizar el conjunto de Mandelbub.

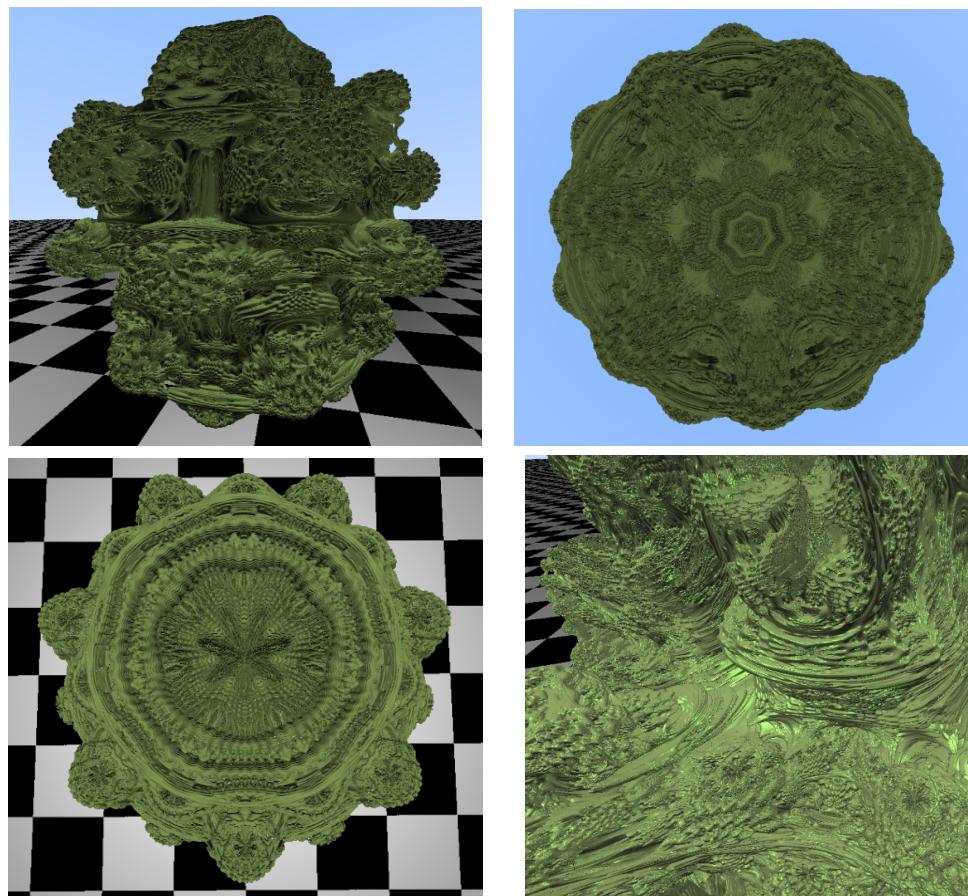


Imagen 8.12: Conjunto de Mandelbub

Nótese como efectivamente mejora por mucho el resultado que nos ofrecía el conjunto de Mandelbrot generalizado a cuaternios de las imágenes 8.11. 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.

8.6. Efectos realistas en la escena

Una vez hemos conseguido visualizar objetos de naturaleza fractal en una escena 3D, debemos cuidar el realismo y las apariencias de la escena. Esto lo conseguiremos añadiendo sombras arrojadas en el suelo y la técnica conocida como antialiasing.

8.6.1. Sombras arrojadas

En la sección 7.6 dotamos a nuestra escena de un modelo de iluminación y de dos fuentes de luz que alumbran los objetos que componen la escena, pero no nos preocupamos de la posible sombra que causen estas fuentes de luz en el plano que consideramos como suelo. Es momento por tanto de graficar estas posibles sombras.

Pensemos primero en cómo hacer notar el efecto de una única fuente de luz direccional sobre el plano. Aprovecharemos el algoritmo ray-marching para esto. El método consiste en, una vez detectada una intersección rayo-plano y conocer el punto p del plano en el que se produce dicha intersección, crear un rayo en la dirección de la fuente de luz y mediante ray-marching detectar una posible intersección con el conjunto que se esté visualizando en ese momento. En caso de encontrarla, consideraremos que el punto está en la sombra provocada por dicha fuente de luz, por lo que multiplicaríamos por una constante cercana a 0 el color que calcula el modelo de iluminación, de tal forma que el color se oscurece, dando esa sensación de sombra. Crearemos entonces una función que devuelva una constante igual a 0 si el punto está a la sombra e igual a 1 si no. Para abbreviar el código, llamamos en el siguiente código `get_dist` a la SDF del objeto que se esté visualizando, la cual depende de la variable `u_fractal`.

```

1 float one_light_shadow(vec3 p, Directional_light light) {
2     Ray R;
3     R.orig = p; R.dir = light.dir;
4     float t = 0.0;
5     float h;
6     for(int i = 0; i < MAX_STEPS; i++ ) {
7         h = get_dist(ray_at(R, t));
8         if(h < u_epsilon)
9             return 0.0;
10        t += h;
11        if(t >= MAX_DIST) break;
12    }
13    return 1.0;
14 }
```

Sin embargo, si multiplicamos por 0 el valor que devuelva el modelo de iluminación el resultado sería una sombra completamente negra, como en la imagen 8.13 (a). Para evitarlo, fijaremos una constante α tal que ese será el valor mínimo por el cual se multiplicaría el color. Por ejemplo, si fijamos $\alpha = 0.3$ y el punto está a la sombra multiplicaríamos el color que calcula el modelo de Phong por

0.3, de forma que se oscurecería el píxel pero no sería completamente negro, como en la imagen 8.13 (b). Para ello necesitamos una transformación lineal que transforme reales en $[0, 1]$ a valores en $[\alpha, 1]$. La transformación deseada sería por tanto $t \mapsto \frac{t}{1-\alpha} + \alpha$.

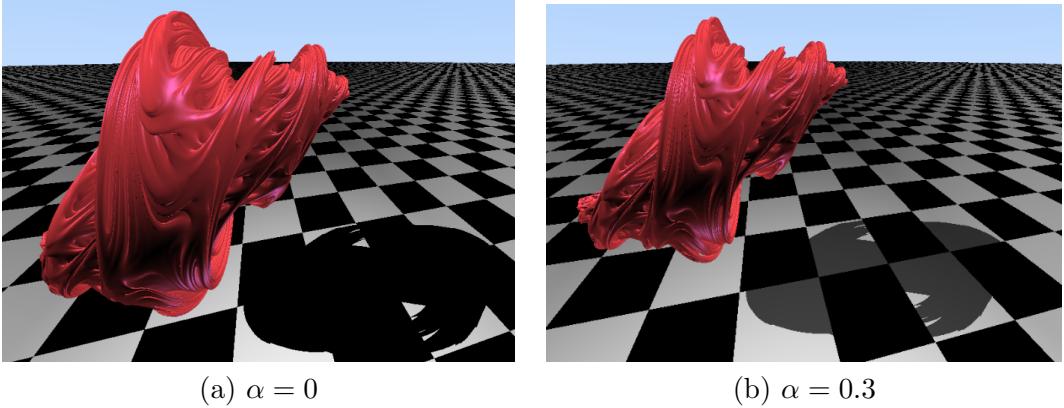


Imagen 8.13: Primera forma de representar sombras

Aún así, 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 a la sombra del conjunto, estén cerca. El truco consiste en pensar en los rayos que, aunque no impacten con el conjunto, pasen muy cerca de él. En estos casos pondríamos los puntos del plano 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 factor de penumbra en cada iteración de ray-marching y nos quedaremos con el menor de ellos. Tan solo tenemos que añadir una línea de código.

```

1 float one_light_shadow(vec3 p, Directional_light light) {
2     // ...
3     float res = 1.0; // Inicialmente 1
4     float k = 8.0; // Constante de suavizado
5     for(int i = 0; i < MAX_STEPS; i++) {
6         h = get_dist(ray_at(R, t));
7         if(h < u_epsilon)
8             return 0.0;
9         // Calculamos el nuevo factor de penumbra
10        res = min(res, k*h/t);
11        t += h;
12        if(t >= MAX_DIST) break;
13    }
14    float alpha = 0.3;           // Smooth constant
15    res = res/(1.0 - alpha) + alpha;
16    return res;
17 }
```

La constante k es un factor que indica si el suavizado es más o menos notable. El valor 8 es suficientemente bueno para nuestro cometido, véase el resultado final en la imagen 8.14, pero mostramos en las imágenes 8.15 la representación con los valores $k = 2, 32$. Nótese como en la imagen 8.15 (a) el sombreado es muy suave y en la imagen 8.15 (b) es bastante más marcado.

Una vez tenemos un método de representar las sombras provocadas por una única fuente de luz es momento de extender la técnica para poder representar

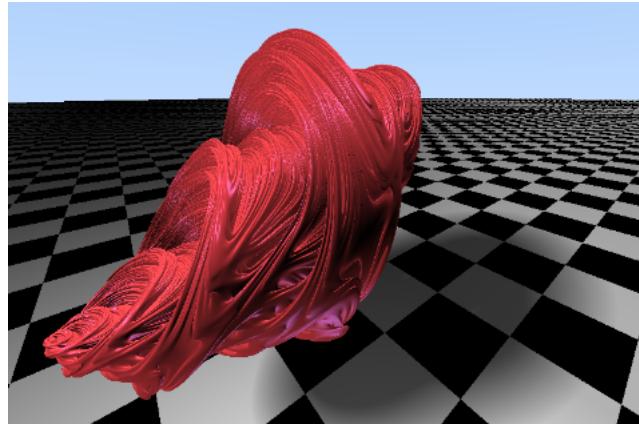
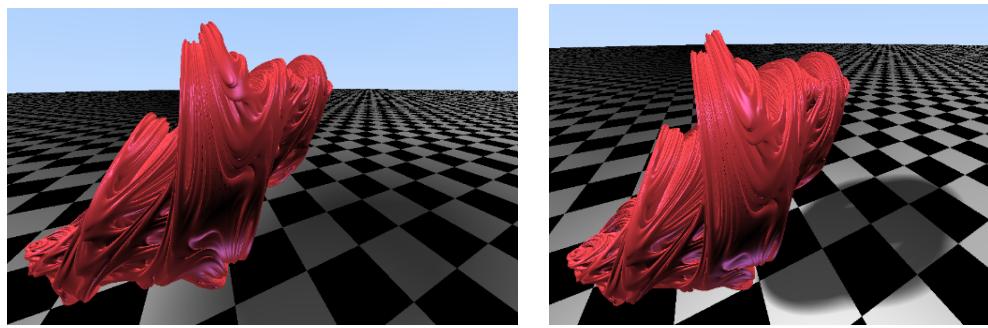


Imagen 8.14: Sombra arrojada con suavizado



(a) $k = 2$

(b) $k = 32$

Imagen 8.15: Sombras con el parámetro $k = 2, 32$

sombras con varias fuentes de luz. Pensemos en un objeto iluminado por varias fuentes de luz. Cada una provoca su propia sombra, pero un punto del suelo puede estar en la sombra provocada por una fuente de luz mientras está iluminado por otra, que hace la sombra de la primera menos notable. La forma de simular este efecto es promediar el efecto de las distintas luces. Por ejemplo, si suponemos dos fuentes de luz:

- Un punto que esté lejos del objeto y por tanto iluminado por las dos fuentes tendrá un factor de 1 en ambos casos, promediando 1, es decir, no hay ninguna sombra.
- Si el punto está en la sombra de una fuente pero iluminado por otra tendrá un factor de 0 por la primera y de 1 por la segunda, resultando en 0.5, es decir, una sombra moderada.
- En un último caso, puede estar en la sombra de ambas, resultando un factor de 0 en cada una de ellas, promediando 0.

Por tanto, el código sería el siguiente:

```

1 | float multi_light_shadow(vec3 p,
2 |     Directional_light[ARRAY_TAM] lights, int num_lights) {
3 |     float shadow = 0.0;
4 |     for(int i = 0; i < ARRAY_TAM; i++) {
5 |         if(i == num_lights) break;
6 |         shadow += one_light_shadow(p, lights[i]);

```

```

7     }
8     shadow /= float(lights_shading);
9     float alpha = 0.0;           // Smooth constant
10    shadow = shadow/(1.0 - alpha) + alpha;
11    return shadow;
12 }
```

En este caso hemos tomado $\alpha = 0$ porque con un valor más alto a penas se hacían notar las sombras. El resultado obtenido es el observable en la imagen 8.16, en la cual podemos ver el efecto suavizado provocado por la interacción entre las dos fuentes de luz.

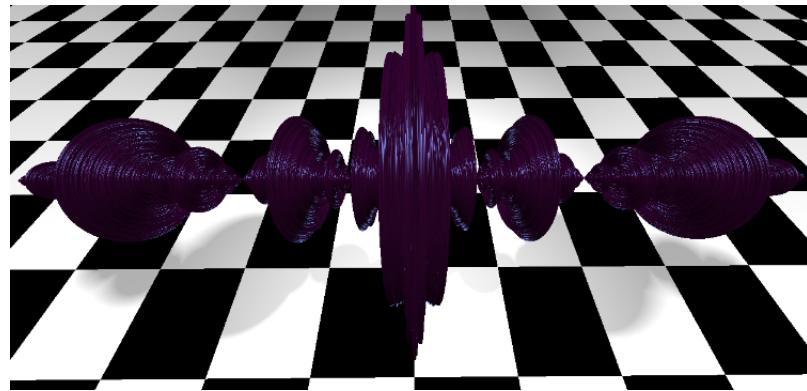


Imagen 8.16: Sombras arrojadas por varias fuentes de luz

8.6.2. Antialiasing

El *antialiasing* es una técnica que nos permite suavizar las imágenes presentadas por píxeles. Para ello, debemos implementar algunos métodos concretos para que la imagen se presente más real. Por ejemplo, fíjese en la imagen 8.17, en la cual se notan mucho los bordes entre los píxeles, de forma que si ampliamos la imagen se vería muy pixelada.

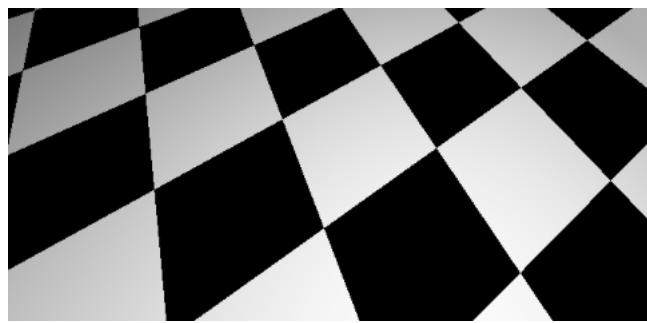


Imagen 8.17: Suelo antes de aplicar antialiasing

Para corregir este efecto, debemos tener en cuenta que al dividir el plano de proyección en píxeles lanzamos un único rayo por píxel, pero por la región del plano que corresponde puede pasar una recta diagonal como las de la imagen 8.17, lo cual puede derivar en que un pixel se dibuje totalmente negro cuando realmente no lo es. De hecho podría ocurrir que en su mayoría el píxel fuera blanco pero justo el rayo se lanza hacia un punto que se evalúa como negro, en cuyo caso el píxel se colorearía negro. Fijémonos en la figura 8.18 (a), en la cual representamos

una pantalla de 5×3 píxeles y cada punto representaría un punto del plano de proyección hacia el cual se lanzaría un rayo según las técnicas implementadas hasta el momento. La forma de solucionar este impedimento es lanzar más de un rayo por píxel, de forma que se cubra una región más representativa del píxel y se promedie el color que devuelve cada rayo en cada píxel. Para ello, fijamos un número natural $n \in \mathbb{N}$ y programaremos la forma de lanzar n^2 rayos en cada píxel, tal y como vemos en la imagen 8.18 (b) con $n = 2$.

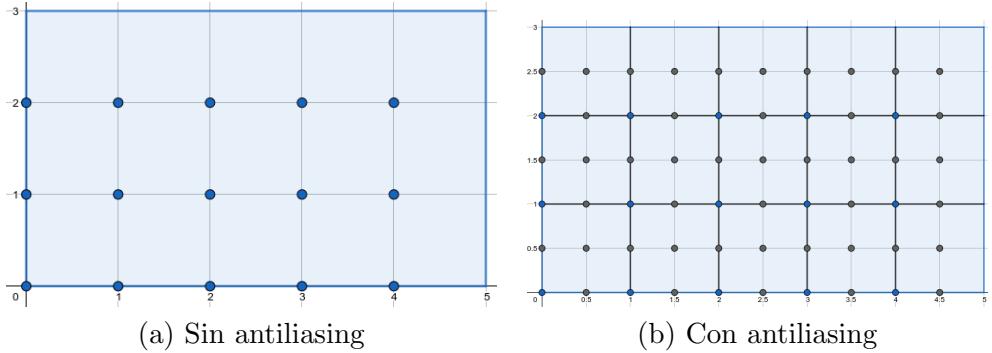


Imagen 8.18: Representación de los rayos que se lanzan a una pantalla

Por tanto, en `main`, cuando obtenemos \mathbf{u}, \mathbf{v} y creamos el rayo, debemos calcular n^2 coordenadas. Recordemos, de la sección 7.5, \mathbf{u}, \mathbf{v} son las coordenadas de dispositivo normalizadas en $[0, 1]$, es decir:

```

1 | vec2 uv = gl_FragCoord.xy / vec2(image_width, image_height);
2 | float u = uv.x;
3 | float v = uv.y;
```

Por lo que ahora debemos sumarle a estas variables pequeños incrementos hasta completar el píxel. Concretamente, ahora dividimos el ancho de un píxel en n intervalos, análogamente con el alto. Por tanto, el incremento sería de $h_w := \frac{1}{image_width \cdot n}$ en anchura y de $h_h := \frac{1}{image_height \cdot n}$ en altura. Pensemos que si normalizamos la pantalla a $[0, 1] \times [0, 1]$, entonces las dimensiones de un píxel son $\frac{1}{image_width} \times \frac{1}{image_height}$, y si cada dimensión la queremos dividir en n , hay que dividir cada dimensión por n , de ahí el valor del incremento.

Seguidamente, creamos un array de colores, el cual tendrá n^2 elementos, uno por cada rayo que lancemos. Por cada posición del array calculamos la coordenada de dispositivo normalizada del punto al cual queremos lanzar el rayo, llamamos a `get_ray`, a `ray_color` y almacenamos el valor de retorno de esta última en el array. Finalmente calculamos la media de todos los colores del array y ese es el valor que finalmente devolvemos.

La forma de, a partir del índice i del array, obtener la coordenada correspondiente es imaginar que dividimos el array en n grupos de n componentes cada uno. Cada grupo representa una fila de puntos del píxel. Por tanto, aplicando la división entera, i/n será la fila y $i \% n$ la columna. Por lo que habría que sumar $(i/n)h_w$ a la coordenada anchura inicial e $(i \% n)h_h$ para obtener la coordenada a la que lanzar el rayo.

El código por tanto sería el siguiente

```

1 | // Antialiasing
2 | vec2 uv = gl_FragCoord.xy / vec2(image_width, image_height);
3 | float u,v;
4 | int n_samples = 3; // 0 cualquier valor
```

```

5 | float hw = 1.0 / (float(image_width * n_samples)),
6 |     hh = 1.0 / (float(image_height * n_samples));
7 Ray r;
8 vec4 colors[ARRAY_TAM];
9 for(int i = 0; i < ARRAY_TAM; i++) {
10     if(i == n_samples*n_samples) break;
11     int x = i/n_samples;
12     int y = i - n_samples*x;
13     u = uv.x + float(x) * hw;
14     v = uv.y + float(y) * hh;
15     r = get_ray(cam, u, v);
16     colors[i] = ray_color(r, ground, lights, num_lights);
17 }
18 gl_FragColor = color_array_average(colors, n_samples*n_samples
    );

```

Donde la función `color_array_average`, tal y como su nombre indica, calcula la media de un array de colores.

Con esta modificación y fijando el valor que consideremos en `n_samples` se consiguen resultados como el de la figura 8.19. Como se puede ver, los bordes son más suaves y de mejor calidad

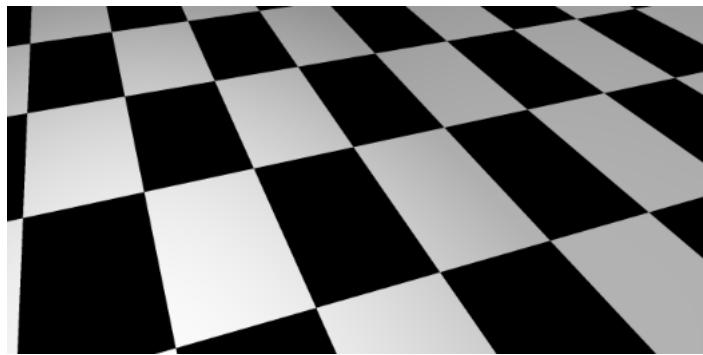


Imagen 8.19: Suelo tras de aplicar antialiasing

El principal y mayor inconveniente que presenta esta técnica es que, como se puede imaginar, es n^2 veces más costosa que lanzar un único rayo por píxel, por lo que ralentiza muchísimo la ejecución, comprometiendo incluso la posibilidad de hacer ejecuciones en tiempo real. 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 incluso para valores de ϵ muy pequeños, fíjese por ejemplo en la representación de la imagen 8.20, la cual tiene una gran calidad y un alto nivel de detalle a pesar de haber usado $\epsilon = 10^{-5}$.

8.7. 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 ray-marching, 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: las esferas englobantes y la limitación de las sombras.

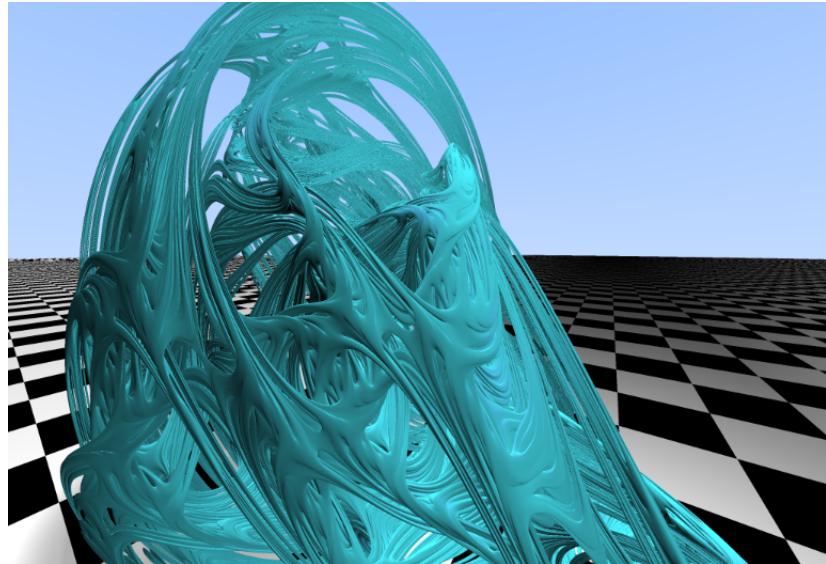


Imagen 8.20: Conjunto de Julia aplicando antialiasing

8.7.1. 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 dentro de la misma. Antes de aplicar ray-marching, reutilizaríamos el código que ya programamos en la sección 7.4.1 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, 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 Mandelbub, 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 // Returns true if R hits S, false otherwise
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(
18     bounding_sphere, r);
19
20 // Ray Marching
21 for(int i = 0; i < MAX_STEPS; i++) {
22     // Distancia al plano ...
23     // ...
24     // Distancia Julia/Mandelbrot/Mandelbulb
25     if(hits_bounding_sphere) {
26         dist = get_dist(p)
27         // ...
28     }
29     // ...
30 }
31 }

```

Y con esta simple técnica podemos optimizar bastante las escenas en las que el conjunto 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.

8.7.2. Limitación de las sombras

En ciertos píxeles hay que aplicar varias veces ray-marching, es el caso de los píxeles que corresponden a puntos del plano que consideramos como suelo, ya que como se ha explicado en la sección 8.6.1, la forma de añadir sombras a la escena es, a partir del punto del plano donde se produce la intersección con el rayo, aplicar ray-marching en la dirección de cada una de las fuentes de luz y buscar posibles intersecciones con el conjunto que se visualice en la escena. Aplicar varias veces ray-marching implica muchas ejecuciones de las SDFs, lo cual también ralentiza la ejecución.

Sin embargo, pensemos que un punto del plano que esté lejano al conjunto no va a estar en la sombra salvo que el ángulo incidente sea muy horizontal, caso que ignoraremos, pues estamos usando fuentes de luz con ángulos incidentes de $45^\circ = \pi/4$ rad y de $135^\circ = 3\pi/4$ rad. Entonces aplicaremos este posible shading únicamente en puntos que estén cercanos al conjunto. Es decir, calcularemos la primera vez la SDF y si esta nos devuelve un valor grande (mayor que 10, por ejemplo), dejaremos de ejecutar el método y devolveremos 1. Esto se puede hacer con una ligera modificación del método `one_light_shadow`.

```

1 float one_light_shadow(vec3 p, Directional_light light) {
2     // ...
3     for(int i = 0; i < MAX_STEPS; i++ ) {
4         h = get_dist(ray_at(R, t));
5         if(i == 0 && h > 10.0)
6             // The point is too far of the set
7             return 1.0;
8         if(h < u_epsilon)
9             return 0.0;

```

```
10 |           // Calculamos el nuevo factor de penumbra
11 |           // ...
12 |       }
13 |       // ...
14 | }
```

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.

BIBLIOGRAFÍA

- [1] Rubiano, G. (2020). *Iteración y fractales (con mathematica ®)*. Universidad Nacional de Colombia.
- [2] Edgar, G.A. (2008). *Measure, Topology, and Fractal Geometry* (2nd ed. 2008.). Springer New York. <https://doi.org/10.1007/978-0-387-74749-1>
- [3] Mandelbrot, B. (1983). *The Fractal geometry of nature*. Freeman.
- [4] Falconer, K. (1990). *Fractal geometry: mathematical foundations and applications*. John Wiley.
- [5] Hausdorff, F. *Dimension und ausseres Mass*. Mathematische Annalen 79 (1919): 157-179. <http://eudml.org/doc/158784>.
- [6] Hurewicz, W.; Wallman, H. (1948). *Dimension Theory*. Princeton University Press.
- [7] Moran, P.A. (1946). *Additive functions of intervals and Hausdorff measure*. In *Mathematical Proceedings of the Cambridge Philosophical Society* (Vol. 42, No. 1, pp. 15-23). Cambridge University Press.
- [8] Wagon, S. (2010). *Mathematica in Action*. Springer Publishing. p. 223
- [9] Payá, R (2008). *Apuntes de Análisis Matemático I...*
- [10] Ostrowski, A.M. (1973). *Solution of equations in Euclidean and Banach spaces*. (3rd ed.). Academic Press.
- [11] Atkinson, K; Han, W. (2009). *Theoretical Numerical Analysis: A Functional Analysis Framework* (3rd ed. 2009). Springer New York. <https://doi.org/10.1007/978-1-4419-0458-4>
- [12] Dubeau, F.; Gnang, C. (2018). Fixed Point and Newton's Methods in the Complex Plane. *Journal of Complex Analysis*, 2018, 1-11. <https://doi.org/10.1155/2018/7289092>
- [13] Milnor, J. (2006). *Dynamics in one complex variable*. (3rd ed.). Princeton University Press. <https://doi.org/10.1515/9781400835539>

- [14] Barnsley, M; Rising, H. (1993). *Fractals everywhere* (2nd ed.). Academic Press.
- [15] *Pythagorean Tree*. (s. f.). Agness Scott College. <https://larryriddle.agnesscott.org/ifs/pythagorean/pythTree.htm>
- [16] Snyder, S.S. (2006). Fractals and the Collage Theorem. *MAT Expository Papers*. 49. <https://digitalcommons.unl.edu/mathmidexppap/49>.
- [17] Dummit, E. (2015). *Dynamics, Chaos, and Fractals (part 4): Fractals*. Rochester MTH 215. https://web.northeastern.edu/dummit/docs/dynamics_4_fractals.pdf.
- [18] Bandt, C., Nguyen Viet Hung, Rao, H. (2006). On the Open Set Condition for Self-Similar Fractals. *Proceedings of the American Mathematical Society*, 134(5), 1369–1374. <http://www.jstor.org/stable/4097989>
- [19] Foroutan-pour, K., Dutilleul, P., Smith, D. (1999). Advances in the implementation of the box-counting method of fractal dimension estimation. *Applied Mathematics and Computation*, 105(2–3), 195–210. [https://doi.org/10.1016/s0096-3003\(98\)10096-6](https://doi.org/10.1016/s0096-3003(98)10096-6).
- [20] *Adding 2D content to a WebGL context - Web APIs — MDN*. (2022, 24 abril). MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Tutorial/Adding_2D_content_to_a_WebGL_context
- [21] colaboradores de Wikipedia. (2022, 16 febrero). *WebGL*. Wikipedia, la enciclopedia libre. <https://es.wikipedia.org/wiki/WebGL>
- [22] *Data in WebGL - Web APIs — MDN*. (2022, 14 marzo). MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Data
- [23] *Getting started with WebGL - Web APIs — MDN*. (2022, 24 abril). MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Tutorial/Getting_started_with_WebGL
- [24] *LearnOpenGL - OpenGL*. (s. f.). OpenGL. <https://learnopengl.com/Getting-started/OpenGL>
- [25] *OpenGL ES - The Standard for Embedded Accelerated 3D Graphics*. (2011, 19 julio). The Khronos Group. <https://www.khronos.org/api/opengles>
- [26] *WebGL: 2D and 3D graphics for the web - Web APIs — MDN*. (2022, 27 abril). MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API
- [27] *WebGL Fundamentals - HTML5 Rocks*. (s. f.). HTML5 Rocks - A Resource for Open Web HTML5 Developers. https://www.html5rocks.com/en/tutorials/webgl/webgl_fundamentals/
- [28] *WebGL model view projection - Web APIs — MDN*. (2022, 26 abril). MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/WebGL_model_view_projection#clip_space

- [29] *WebGLRenderingContext* - *Web APIs* — MDN. (2022, 20 enero). WebGLRenderingContext. <https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext>
- [30] López, P. (2020, 30 abril). *Ray Tracing: ¿Qué es y para qué sirve? - Definición*. GEEKNETIC. <https://www.geeknetic.es/Ray-Tracing/que-es-y-para-que-sirve>
- [31] Hart, J. (1995). Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces. *The Visual Computer*. <https://doi.org/10.1007/s003710050084>
- [32] Quilez, I. (s. f.). *3D SDFs*. Íñigo Quilez. Recuperado 14 de mayo de 2022, de <https://iquilezles.org/articles/distfunctions/>
- [33] Conway, J. H., Smith, D. A. (2003). *On quaternions and octonions*. 21. CRC Press LLC.
- [34] Pharr, M., Jakob, W., Humphreys, G. (2017). *Physically Based Rendering* (3rd ed.). 57-121. Morgan Kaufmann. <https://doi.org/10.1016/B978-0-12-800645-0.50002-6>
- [35] Crane, K. (2005). Ray Tracing Quaternion Julia Sets on the GPU. *University of Illinois at Urbana-Champaign*
- [36] Douady, A., Hubbard, J. (2009). *Exploring the Mandelbrot set*. *The Orsay Notes*.
- [37] Quilez, I. (s. f.-b). *Normals for an SDF*. Íñigo Quilez. Recuperado 15 de mayo de 2022, de <https://iquilezles.org/articles/normalsSDF/>
- [38] Quilez, I. (2004). *Distance to fractals*. Íñigo Quilez. Recuperado 16 de mayo de 2022, de <https://iquilezles.org/articles/distancefractals/>
- [39] Quilez, I. (2009). *Mandelbulb*. Íñigo Quilez. Recuperado 16 de mayo de 2022, de <https://iquilezles.org/articles/mandelbulb/>

APÉNDICE A

DOCUMENTACIÓN DEL CÓDIGO JAVASCRIPT

APÉNDICE B _____

_____ ANOTHER APPENDIX