Universidad Zaragoza

1542

Trabajo Fin de Grado

Modelado de geometría mediante Signed Distance Functions y desarrollo de algoritmos para su intersección.

Modeling of geometry by Signed Distance Functions and development of algorithms for its intersection.

Autor

Néstor Monzón González

Directores

Adolfo Muñoz Orbañanos

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2021

RESUMEN

Aunque la representación más común de la geometría en entornos de *ray tracing* sean las mallas de triángulos, existen otras formas de representar geometrías complejas. Una de estas representaciones alternativas son las SDFs (Signed Distance Functions) (capítulo 2). Estas funciones $f : \mathbb{R}^3 \rightarrow \mathbb{R}$ devuelven la distancia con signo a la superficie (para cada punto $\mathbf{p}$, $f(\mathbf{p}) = 0$ en la superficie, $f(\mathbf{p}) < 0$ en el interior y $f(\mathbf{p}) > 0$ en el exterior). Su naturaleza geométrica permite una manipulación de la geometría mucho más directa que la basada en polígonos: se modifica directamente la ecuación implícita de los volúmenes, en lugar de modificar individualmente los vértices de los polígonos.

En este trabajo, presentamos una librería para la definición y manipulación de geometría mediante *SDFs*. Mostramos distintos ejemplos de estas funciones y sus capacidades (capítulo 4). El código implementado se ha integrado con el renderer *Mitsuba2* para sintetizar escenas avanzadas con estas geometrías.

Además, se han implementado tanto los algoritmos más populares de intersección de estos volúmenes (*Sphere Tracing*, sección 2.2.3, y *Enhanced Sphere Tracing*, 2.2.4), como otros métodos nuevos diseñados para este proyecto (*Newton Marching*, en la sección 3.1, y *Forward Newton Marching*, sección 3.2), basados en el método de Newton para intersectar más rápidamente superficies planas. Se han comparado todos los métodos experimentalmente, y se han analizado las capacidades y limitaciones de los nuevos algoritmos (sección 5.1). Se muestra también la posibilidad que ofrecen los nuevos algoritmos de intersectar geometría definida por otro tipo de funciones derivables (no necesariamente *SDFs*).

Índice

1. Introducción **3**

2. Fundamentos teóricos y trabajo previo **5**

2.1. Representación de geometría . . . 6

2.2. Algoritmos de *marching* previos . . . 8

2.2.1. Ray Marching de paso fijo . . . 8

2.2.2. Regular marching . . . 9

2.2.3. Sphere Tracing . . . 10

2.2.4. Relaxed y Enhanced Sphere Tracing . . . 12

3. Contribución: Nuevos algoritmos de *Marching* **15**

3.1. Newton Marching . . . 15

3.2. Forward Newton Marching . . . 18

4. SDFs **21**

4.1. Primitivas . . . 21

4.1.1. Esfera . . . 21

4.1.2. Prisma rectangular . . . 22

4.1.3. Plano . . . 23

4.1.4. Línea . . . 23

4.1.5. Tetraedro . . . 24

4.1.6. Ruido de Perlin . . . 24

4.2. Modificaciones . . . 25

4.2.1. Transformaciones y escalado . . . 26

4.2.2. Simetrías . . . 26

4.2.3. Repetición . . . 27

4.2.4. Redondeado . . . 27

4.3. Combinaciones de *SDFs* . . . 28

4.3.1. Unión . . . 28

4.3.2. Intersección . . . 28

4.3.3. Diferencia 29
4.3.4. Desplazamiento 29
4.4. L-Systems, árboles y reflexiones 31
4.5. Diseño Software 33
4.5.1. Diseño de clases 33
4.5.2. PImpl 34
4.5.3. Integración con *Mitsuba* 35

5. Resultados **37**
5.1. Análisis de algoritmos 37
5.1.1. Experimento: Prisma redondeado 37
5.1.2. Experimento: túnel 39
5.1.3. Experimento: Cambios en la derivada 41
5.1.4. Análisis: conclusiones 41
5.2. Aplicaciones 43
5.2.1. Delimitación de materiales 44
5.2.2. Mapa de normales 45
5.2.3. Rascacielos 46
5.2.4. Tetraedro de Sierpinski 46

6. Conclusiones **49**

7. Bibliografía **51**

Lista de Figuras **53**

Lista de Tablas **55**

Capítulo 1

Introducción

Generalmente, tanto en animación como en videojuegos y en todos los ámbitos de la informática gráfica, la geometría de las escenas tridimensionales se define mediante mallas de triángulos. Aunque durante los años se han desarrollado y mejorado técnicas y algoritmos para hacer muy eficiente esta representación, existen otras opciones que cuentan con distintas ventajas.

Una de estas representaciones alternativas es las SDFs (Signed Distance Functions), funciones que, para cada punto en el espacio, devuelven la mínima distancia a la superficie del objeto. La principal ventaja de esta representación es que permite la manipulación (repetición, duplicación, unión suavizada, intersección, substracción, etc.) de la geometría de forma mucho más simple e intuitiva que las alternativas. Concretamente, permiten la manipulación del espacio de forma directa, en lugar de a través de la modificación de los vértices y normales tradicionales.

Otra de las principales ventajas de la representación de geometría mediante *SDFs* definidas proceduralmente, frente a la representación tradicional, es su capacidad de representar volúmenes muy complejos de forma compacta; con muy bajo coste en memoria [1].

Por otra parte, este grado de libertad introducido en la geometría puede hacer imposible la resolución analítica de la intersección de la geometría con los rayos, parte fundamental del proceso de renderizado de las escenas. Así, se deben recurrir a otros métodos computacionalmente más costosos.

El algoritmo más popular para su intersección se trata del *Sphere Tracing* [2]. Aunque destaca por su robustez y simplicidad, presenta ciertas limitaciones, como su ineficiencia en los casos en los que un rayo se aproxima a superficies planas. Se han propuesto modificaciones que abordan estos problemas, como el *Enhanced Sphere Tracing* (Keinert et al. [3]).

En este trabajo, proponemos varios nuevos algoritmos basados en el método de Newton, y los comparamos experimentalmente con los métodos previos. Además,

presentamos código en C++ para la definición, manipulación y renderización de geometría de este tipo utilizando un diseño moderno.

Por otra parte, se presentarán diversos ejemplos de funciones *SDF* como muestra de su expresividad para la definición, manipulación y combinación de geometrías.

Finalmente, la integración del código desarrollado con el renderer de código abierto *Mitsuba2* permitirá presentar muestras de escenas con este tipo de geometría. Además, se presentarán dos aplicaciones prácticas de las *SDFs* al margen de la representación de la geometría: la mezcla de materiales delimitada mediante *SDFs* (sección 5.2.1), y la representación implícita de un mapa de normales (sección 5.2.2).

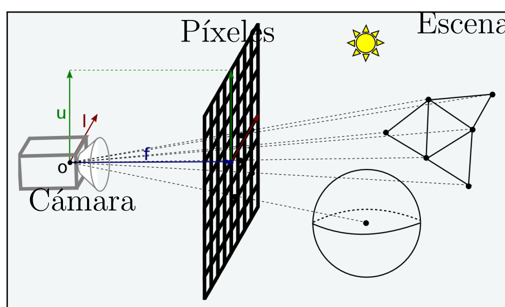
Capítulo 2

Fundamentos teóricos y trabajo previo

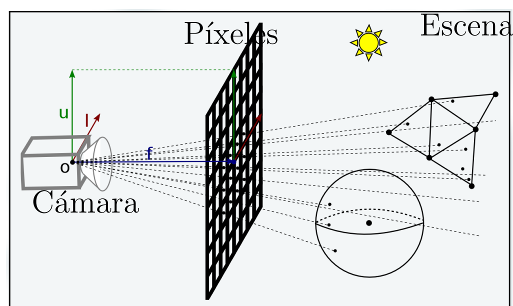
En informática gráfica existen dos grupos principales de técnicas para resolver el problema de trasladar la geometría tridimensional de una escena a una pantalla bidimensional. El primero es la rasterización, consistente en proyectar los vértices de los polígonos que definen la escena al espacio de pantalla (figura 2.1a). El segundo grupo es el de las técnicas basadas en el trazado de rayos (*ray tracing*), que imita el comportamiento de la luz lanzando rayos desde la cámara y comprobando sus intersecciones con las superficies de la escena (figura 2.1b).

La rasterización ha sido más popular históricamente en aplicaciones de tiempo real por su velocidad, aunque esta ventaja ha perdido importancia con las mejoras de hardware de los últimos años. Por otra parte, los algoritmos basados en *ray tracing* producen una apariencia mucho más realista, ya que permiten una simulación más físicamente correcta de la interacción de la luz con la escena.

Los algoritmos desarrollados en este trabajo se engloban en el segundo grupo de técnicas.



(a) Rasterización de una escena en los píxeles de la imagen, desde la perspectiva de la cámara.



(b) *Ray Tracing* de una escena desde la perspectiva de la cámara.

Figura 2.1: Rasterización y *Ray Tracing*.

2.1. Representación de geometría

Aunque también existen las representaciones paramétricas de las superficies, en este trabajo nos centramos en su representación implícita. Dada una función $f : \mathbb{R}^3 \to \mathbb{R}$, se define la superficie como el conjunto de puntos que satisfacen la ecuación:

$$f(x, y, z) = 0 \tag{2.1}$$

Es decir, estas funciones, para cada punto *x, y, z* en el espacio, $f(x, y, z) = 0$ si este pertenece a la superficie. Formalmente, la superficie es, por tanto, la función $f^{-1}(0)$. Por ejemplo, la ecuación de la superficie de una esfera centrada en el origen y de radio r es la siguiente:

$$x^2 + y^2 + z^2 - r = 0 \tag{2.2}$$

Así, en métodos como el *ray tracing*, se obtiene un sistema de ecuaciones combinando esta fórmula junto con las ecuaciones del rayo ($\mathbf{o} + t\mathbf{d} = 0$, $t > 0$ para un rayo con origen $\mathbf{o}$ y dirección $\mathbf{d}$), permitiendo obtener las intersecciones con la superficie al resolverlo. Este sistema es, para $\mathbf{p} = (x, y, z)$:

$$\begin{aligned} f(\mathbf{p}) &= 0 \\ \mathbf{p} &= \mathbf{o} + t\mathbf{d} \end{aligned} \tag{2.3}$$

Estos sistemas, para algunas funciones *f*, presentan soluciones analíticas, que tienen la ventaja de que se pueden resolver en tiempo constante. Gran cantidad de esfuerzo se ha dedicado durante los años a encontrar nuevos algoritmos e implementaciones para intersecciones con primitivas como los triángulos (por su flexibilidad para representar otras geometrías).

Por otra parte, para otras geometrías, su correspondiente función f podría no tener una solución analítica.

Así, se debe encontrar una solución numérica. En este contexto, las *SDFs* (*Signed Distance Functions*, funciones de distancia con signo) son un subconjunto de las funciones f que permiten aplicar métodos numéricos como los que se explicarán en el apartado 2.2 para resolver el sistema y hallar una intersección.

Las *SDFs* también definen una superficie como los puntos $\mathbf{p}$ que cumplen $f_{SDF}(\mathbf{p}) = 0$. La diferencia es que las distancias devueltas por $f_{SDF}(\mathbf{p})$ son distancias euclídeas, geométricas, en lugar de algebraicas [2]. Por ejemplo, la misma esfera definida algebraicamente en 2.2, sería 2.4:

$$\|\mathbf{p}\| - r = 0 \tag{2.4}$$

Como se ha comentado, en este caso, se sigue cumpliendo que $f_{SDF}(\mathbf{p}) = 0$ para todo $\mathbf{p}$ en la superficie, pero además, $f_{SDF}(\mathbf{p}) < 0$ si se encuentra dentro, y $f_{SDF}(\mathbf{p}) > 0$ si está en el exterior (véase 2.2). El valor de $f_{SDF}(\mathbf{p})$ es la distancia, con signo, a la superficie.

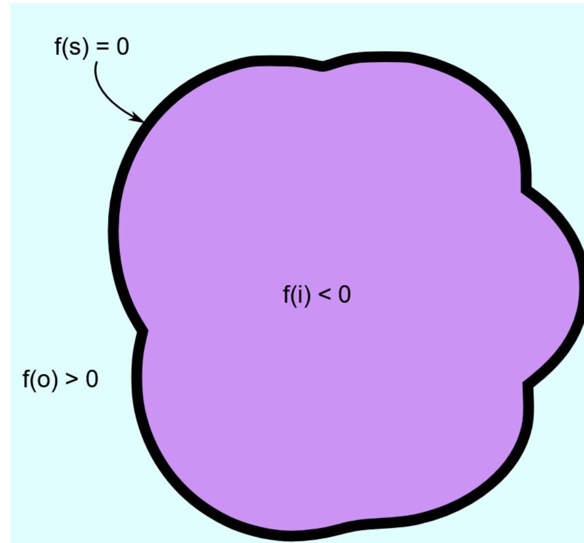


Figura 2.2: Esquema de un volumen definido por una función f *SDF*. En azul, el exterior del volumen (puntos $\mathbf{o}$). En violeta (puntos $\mathbf{i}$), el interior. En negro ($\mathbf{s}$), la superficie.

Utilizar esta representación basada en distancias euclídeas presenta varias ventajas. En primer lugar, su definición es muchas veces más intuitiva, en este caso, por ejemplo, la distancia a la superficie de una esfera es la distancia a su centro (que es el módulo del punto si el centro es el origen), menos el radio de la esfera. Esto permite definir nuevas formas de forma directa. Por ejemplo, un cilindro vertical infinito también centrado en el origen se representa de forma similar, ignorando el eje y:

$$f_{cilindro}(\mathbf{p}) = \|(p_x, 0, p_y)\| - r \tag{2.5}$$

Además, la naturaleza geométrica de las *SDFs* posibilita la manipulación directa de estas figuras mediante transformaciones al espacio, combinaciones de funciones y manipulación de las distancias devueltas por las funciones. Uno de los casos más sencillos pero muy común es la unión de distintas *SDFs* en una sola, que se define simplemente como:

$$f_{union}(\mathbf{p}) = min(f_{sdf1}(\mathbf{p}), f_{sdf2}(\mathbf{p})) \tag{2.6}$$

Ya que, por definición, la mínima distancia a un cierto número de primitivas será el mínimo de las distancias a cada una de ellas.

Todo esto ha convertido a estas representaciones en las favoritas de comunidades de artistas procedurales como *Shadertoy* [4], ya que permiten al programador un control mucho mayor que el que se obtendría en entornos clásicos de gráficos por ordenador.

2.2. Algoritmos de *marching* previos

En todos los casos, para intersectar estas funciones se requiere la búsqueda gradual en el espacio de la primera raíz (el punto en el que la función *SDF* vale 0), ya que al no conocer a priori la ecuación, no disponemos de una solución analítica como en el caso de las representaciones implícitas algebraicas más comunes. Esto se implementa mediante *ray marching*, que para determinar si un rayo intersecta una figura, recorre el rayo desde su origen en adelante siguiendo distintas estrategias.

En este trabajo, se han implementado varios algoritmos de *ray marching* existentes:

- *Regular marching*, sección 2.2.2.
- *Sphere tracing*, 2.2.3.
- *Enhanced Sphere Tracing*, 2.2.4.

Además, se proponen dos nuevos métodos que se detallarán en el capítulo 3. A continuación, se detallarán los fundamentos de los algoritmos existentes implementados.

2.2.1. Ray Marching de paso fijo

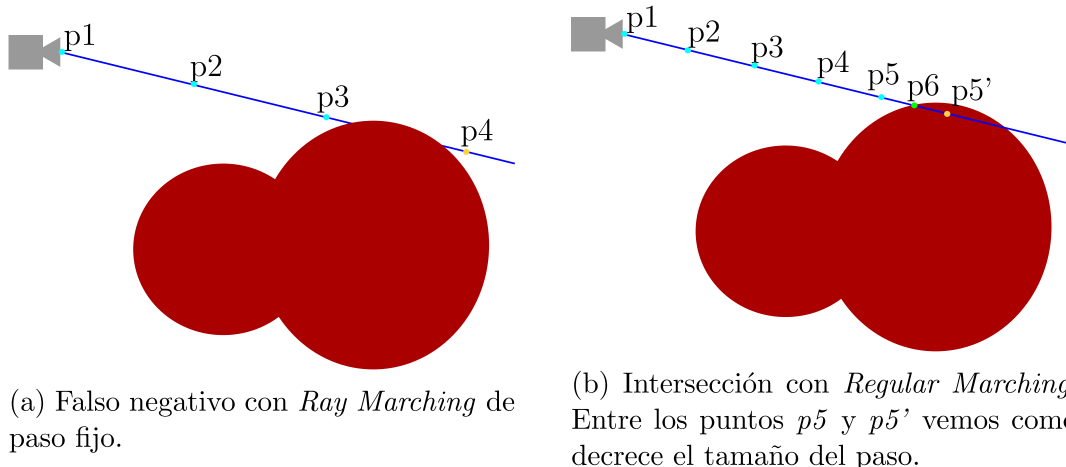
La versión más simple de este método es el *ray marching* de paso fijo, que avanza una distancia fija en cada paso hasta llegar a una distancia *suficientemente pequeña* a la superficie (o no detectarse una intersección, en la práctica al llegar a un máximo número de pasos o de distancia del origen).

Siendo *origen* el punto de origen del rayo, *d* su dirección, *tam_paso* el tamaño de cada paso y *max_pasos* el máximo número de pasos permitidos, el algoritmo que devuelve la distancia a la primera intersección con la figura definida por la función *sdf* es el siguiente:

Algorithm 1: Fixed Ray Marching

```
paso = 0;
while paso < max_pasos do
    p = origen + d * tam_paso * paso;
    distancia = sdf(p);
    if distancia < minDistancia then
        return tam_paso * paso;
    end
    paso = paso + 1;
end
return infinito;
```

Un problema de este algoritmo, aparte de su ineficiencia al ser un acercamiento de fuerza bruta, es que tanto el tamaño del paso como el máximo numero de pasos ideales



(a) Falso negativo con *Ray Marching* de paso fijo.

(b) Intersección con *Regular Marching*. Entre los puntos *p5* y *p5'* vemos como decrece el tamaño del paso.

Figura 2.3: Intersecciones con *Ray Marching* de paso fijo y *Regular Marching*. El volumen a intersectar, definido por una *SDF*, es el rojo. Se muestra un rayo de la cámara en azul, y los distintos puntos correspondientes a cada iteración de los algoritmos.

dependerán de la naturaleza de la *SDF* a intersectar. Por ejemplo, si el tamaño de paso es demasiado grande, es probable que el algoritmo no detecte algunas intersecciones al saltarse las raíces, como ocurre en el diagrama 2.3a. Como se puede observar, al llegar al punto *p4*, de la cuarta iteración, se ha atravesado la geometría sin detectarlo.

2.2.2. Regular marching

En este trabajo, se ha implementado un algoritmo similar al anterior (2.2.1) pero que utiliza el teorema de Bolzano para mayor precisión. Al principio también se avanza con pasos fijos, pero se mantienen dos puntos simultáneamente (p_i y p_{i+1}, pa y pb en el código) y si se detecta que la segunda distancia es negativa ($f(p_{i+1}) < 0$) se divide el rango por la mitad para encontrar la superficie con mayor precisión. Así, el algoritmo es el siguiente:

En la figura 2.3b se observa como, cuando el $pa = p5$ y $pb = p5'$, se detecta mediante Bolzano que hay una raíz entre ambos, por lo que el tamaño de paso se reduce a la mitad y se recalcula pb (ahora *p6*) sin modificar pa. Finalmente, se detecta que pb está suficientemente cerca de la superficie y se devuelve la intersección.

Aunque este algoritmo es más preciso que el algoritmo de paso fijo, también es muy poco eficiente. Por ejemplo, los rayos que no intersectan con la geometría siempre realizarán el máximo número de pasos permitido. Por otra parte, con el suficiente número de pasos es muy consistente incluso con funciones de distancia inexactas, ya que no asume que sus distancias serán exactas como los siguientes algoritmos (2.2.3,

Algorithm 2: Regular marching

```
paso = 0;
step = max_t / max_pasos;
ta = 0 tb = step pa = origen pb = origen + tb * d;
while paso < max_pasos do
    if sdf(pa) * sdf(pb) <= 0 then
        step = step / 2;
    else
        pa = p1;
        ta = t1;
    end
    if |sdf(pa)| < minDistancia then
        return ta;
    end
    if ta > max_dist then
        return infinito;
    end
    tb = ta + step;
    p1 = origen + d * tb;
    paso = paso + 1;
end
return infinito;
```

2.2.4).

2.2.3. Sphere Tracing

Por otra parte, el *Sphere Tracing* es el algoritmo más popular para la intersección con *SDFs*, por su simplicidad, eficiencia y robustez, desde su introducción en 1995 por Hart [2].

Consigue ser mucho más rápido que los algoritmos anteriores al reducir en gran medida el número de pasos en muchos casos. Esto lo logra utilizando un tamaño de paso flexible, aprovechando la definición de las funciones *SDF*.

Como el *Regular Marching*, se trata de una variante del *Ray Marching*. Es similar al primero, pero, como se ha mencionado, en lugar de dar pasos de tamaño fijo, en cada iteración el tamaño del paso se determina con el valor devuelto por la *SDF*, ya que asegura que el tamaño del paso no se saltará ninguna raíz. Como, por definición, es la mínima distancia a la superficie desde el punto actual, al avanzar esa distancia en cualquier dirección el nuevo punto estará o fuera de la figura, o en su superficie.

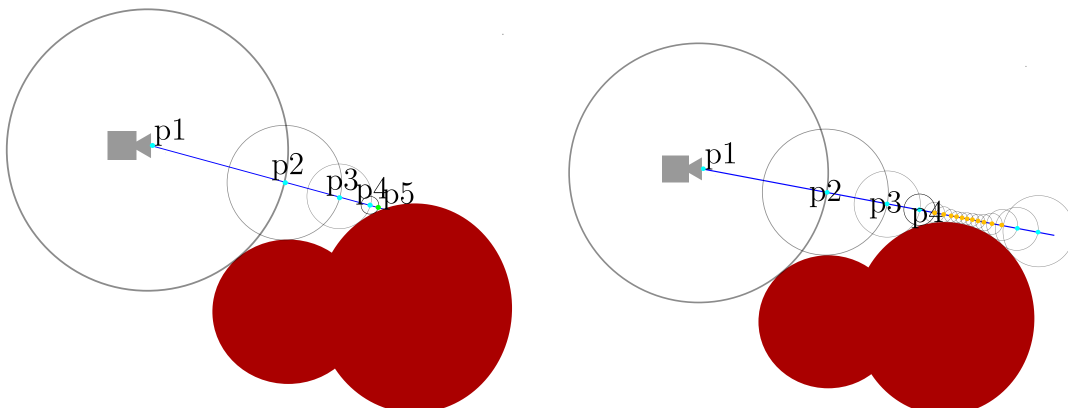
Este proceso se repite de forma similar al anterior algoritmo, o bien hasta detectar la superficie al llegar a una cierta distancia de esta, o bien hasta decidir que no ha habido una intersección al llegar al máximo de distancia o de número de pasos. Sea t

la distancia recorrida por el rayo desde su origen, y *max_dist* el máximo de distancia:

Algorithm 3: Sphere Tracing

```
paso = 0;
t = 0;
while paso < max_pasos & t < max_dist do
    p = origen + d * t;
    distancia = sdf(p);
    if distancia < minDistancia then
        return t;
    end
    t = t + distancia;
end
return infinito;
```

Por ejemplo, el siguiente diagrama (2.4a) muestra una intersección con el algoritmo explicado. Se ha reducido a dos dimensiones por simplicidad, ya que el proceso es idéntico en tres dimensiones. La geometría de la escena son los dos círculos rojos, y se quiere hallar la intersección del rayo azul originado en la cámara. Como se ha descrito, el rayo se va recorriendo paso a paso, comenzando en su origen (*p1*). En cada iteración, al obtener la distancia a la superficie evaluando la *SDF* de la escena, se obtiene efectivamente el círculo (o esfera en tres dimensiones) que rodea al punto y no contiene la superficie buscada. Esto permite avanzar hasta el borde de la circunferencia, en la dirección del rayo, sin sobrepasar la superficie. Al repetir este proceso observamos cómo se generan los sucesivos puntos a partir del primero, hasta que *p5* se encuentra en el rango de distancia buscado, devolviendo la intersección.



(a) Una intersección con *Sphere Tracing* en 2D.

(b) Caso costoso con *Sphere Tracing*.

Figura 2.4: Casos del algoritmo de *Sphere Tracing*.

Limitaciones:

Aunque el algoritmo de *Sphere Tracing* es muy robusto (con el suficiente número de pasos intersectará cualquier superficie definida por una *SDF*), puede resultar costoso en algunos casos.

Uno de los peores casos se da cuando un rayo pasa muy cerca de una superficie, de forma paralela, pero sin llegar a intersectar (2.4b). El tamaño de paso se hace muy pequeño, llevando a dar un gran número de pasos en la zona tangente del rayo (marcados en amarillo en diagrama).

Además, cabe destacar que el *Sphere Marching* requiere que las funciones de distancia sean o bien exactas (*SDFs* reales) o estimadores $f_{estimador}$ que siempre devuelvan valores mayores o iguales a la distancia real en el exterior de los volúmenes (es decir, $f_{SDF}(\mathbf{p}) > 0 \rightarrow f_{estimador}(\mathbf{p}) \geq f_{SDF}(\mathbf{p}) \forall \mathbf{p} \in \mathbb{R}^3$). Formalmente, estos estimadores corresponden a la división de la *SDF* entre su constante de *Lipschitz* [1]. En la sección 4.3.4 se mostrará un ejemplo de una de estas funciones no exactas.

Aunque esto es previsible dado que se trata de un algoritmo específico para la intersección de este tipo de funciones, otros *marchers* como los dos anteriores (2.3a, 2.3b) o los nuevos algoritmos propuestos en las secciones 3.1 y 3.2 no cuentan con esta limitación.

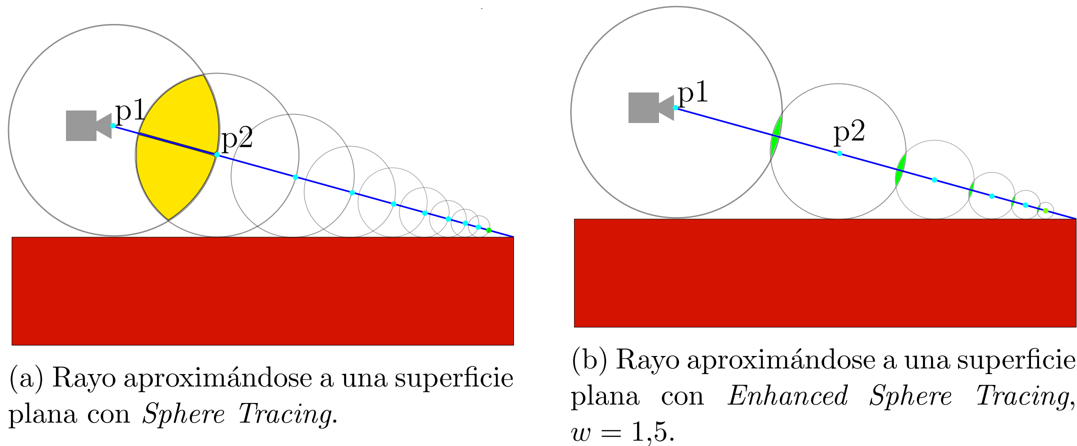
2.2.4. Relaxed y Enhanced Sphere Tracing

En esta sección describiremos dos mejoras similares al algoritmo anterior. Parten de la observación de la ineficiencia del primer algoritmo en los rayos que se aproximan a superficies planas (o casi planas), como en el caso anterior, 2.4b, o en el siguiente, 2.5a.

La primera fue propuesta en 2014 por Keinert et al. [3]. La segunda, propuesta por Bálint et al. [5], es una extensión ligeramente mejorada de la primera, y es la versión que hemos implementado en este trabajo. A continuación, describiremos en primer lugar la primera aproximación, ya que sirve como introducción a la segunda.

En la figura 2.5a observamos como una intersección que debería ser sencilla, mediante *Sphere Tracing* se resuelve con un número de pasos que crece exponencialmente, solo limitado por la distancia mínima ϵ proporcionada al algoritmo.

En el *Enhanced Sphere Tracing* se observa que en ocasiones como esta el algoritmo original es muy conservador: por ejemplo, los círculos que rodean los puntos *p1* y *p2* presentan un gran área de superposición. Sabemos que este área no puede contener la superficie, por lo que se podría aumentar el tamaño del paso más allá del radio hallado en *p1*. Este aumento viene dado por el hiperparámetro de relajación w, de forma que



(a) Rayo aproximándose a una superficie plana con *Sphere Tracing*.

(b) Rayo aproximándose a una superficie plana con *Enhanced Sphere Tracing*, $w = 1{,}5$.

Figura 2.5: Comparación de *Sphere Tracing* y *Enhanced Sphere Tracing* para un mismo caso.

$paso_i = r_i \cdot w$. Por ejemplo, con $w = 1{,}5$, obtenemos el siguiente diagrama para el mismo caso 2.5b.

Este procedimiento sirve para aproximar una sola superficie plana, pero para geometrías más complejas se debe hacer una comprobación adicional. Si en una iteración se detecta que dos círculos consecutivos no intersectan (punto *p3* en la figura 2.6) se debe retroceder, dando el paso de *Sphere Tracing* normal en su lugar (punto *p4*).

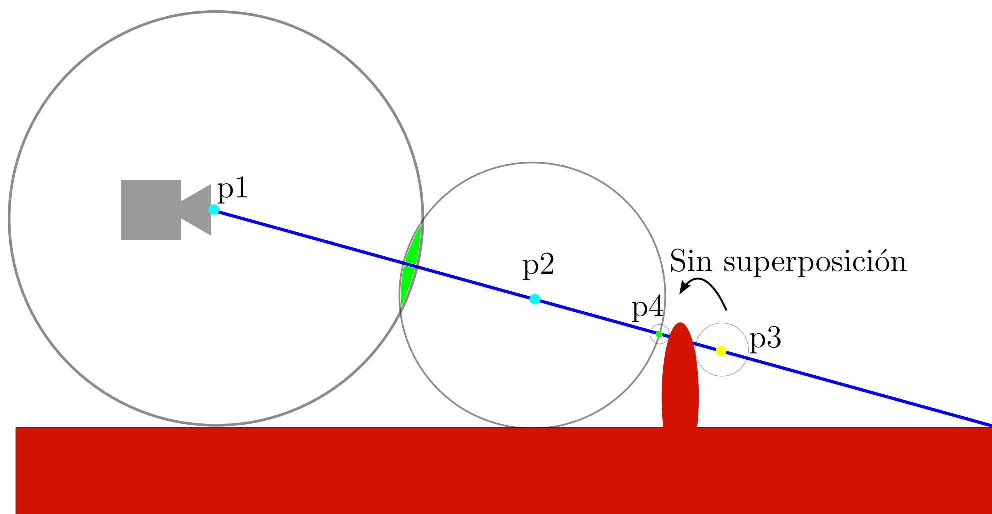


Figura 2.6: Detección de un posible error en *Enhanced Sphere Tracing*, $w = 1{,}5$.

Así, el parámetro de relajación w debe depender de la convexidad de la escena: cuanto más plana sea mayor puede ser el parámetro. Si el parámetro es demasiado grande para la escena, se sobrestimará un gran número de pasos, haciendo este método más costoso que el *Sphere Tracing* original.

El segundo método (Bálint et al. [5]) es casi idéntico al primero, con la diferencia de que aproxima de forma óptima superficies planas. Consiste en encontrar mediante trigonometría el siguiente punto a explorar a partir de los dos anteriores de forma que la esfera que lo rodea sea tangente a la anterior. En la práctica, esto equivale a encontrar el parámetro de relajación geométricamente de forma dinámica. Esta versión es la que se ha decidido implementar para compararla con los otros métodos, y a la que nos referiremos cuando hablemos de *Enhanced Sphere Tracing* en adelante.

Formalmente, el algoritmo es el siguiente.

Algorithm 4: Enhanced Sphere Tracing

$r_{i-1} = r_i = d_i = t = 0$;
$r_{i+1} = \infty$;
paso = 0;
t = 0;
while $paso < max_pasos$ & $t < max_dist$ **do**
  $d_i = r_i + w + r_i * \dfrac{d_i - r_{i-1} + r_i}{d_i + r_{i-1} - r_i}$;
  $r_{i+1} = sdf(origen + d * (t + d_i)$;
  if $d_i > r_i + r_{i+1}$ **then**
    $d_i = r_i$;
    $r_{i+1} = sdf(origen + d * (t + d_i)$;
  end
  $t = t + d_i$;
  if $r_{i+1} < minDistancia$ **then**
    return t;
  end
  $r_{i-1} = r_i$;
  $r_i = r_{i+1}$;
end
return ∞;

Capítulo 3

Contribución: Nuevos algoritmos de *Marching*

Dada la limitación que imponen a las *SDFs* (descrita en la sección 2.2.3) los métodos basados en *Sphere Tracing*, así como el elevado número de pasos que requieren para alcanzar muchas superficies, especialmente las planas (véase la sección 2.2.3), hemos desarrollado una serie de métodos que, utilizando las derivadas de la función *SDF* en cuestión, pueden alcanzar antes los ceros de la ecuación.

Al utilizarse las derivadas en cada iteración del algoritmo, se ha considerado importante una implementación analítica de las derivadas (en lugar de la solución numérica mediante derivadas parciales que es más común). Así, se han implementado estas derivadas analíticas en todas las funciones que ha sido posible (para más detalle, véanse las secciones de primitivas, 4.1, y de diseño software, 4.5).

Además, como se ha mencionado brevemente en la sección de *Sphere Tracing* (2.2.3), estos algoritmos pueden servir para hallar intersecciones con superficies definidas por ecuaciones implícitas al margen de las *SDFs*, ya que no requieren una distancia exacta como los anteriores.

3.1. Newton Marching

El **método de Newton** es un algoritmo de búsqueda de raíces de funciones reales. Geométricamente, su funcionamiento es el siguiente (figura 3.1). Dada la función *f(x)*, en rojo, y el punto inicial *p1* $(x1, f(x1))$, se toma la derivada en ese punto, $f'(x1)$, tangente a la función y se intersecta con el eje x para obtener *x2*. Este proceso se repite hasta llegar a un x_i tal que $|f(x_i)| < \epsilon$ (*p4* en la figura).

Matemáticamente, el punto x_{i+1} se calcula a partir de x_i siguiendo la fórmula [6]:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{3.1}$$

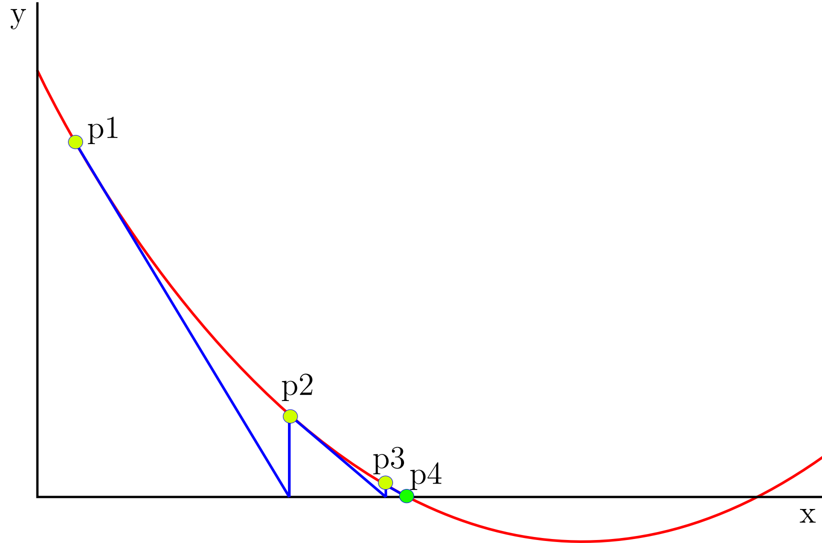


Figura 3.1: Método de Newton.

Este proceso se repite hasta llegar a un máximo número de iteraciones o alcanzar una distancia ϵ suficientemente cercana a cero (encontrar un x_i tal que $|f(x_i)| < \epsilon$).

Newton Marching:

Así, el primer método propuesto, que hemos denominado *Newton Marching*, consiste en la aplicación casi directa del método de Newton descrito para la función *SDF* de la escena a lo largo de la trayectoria del rayo. Es decir, siendo t la distancia recorrida desde el origen o en la dirección d, buscamos el primer cero de la función:

$$f(t) = f_{sdf}(o + d \cdot t) \tag{3.2}$$

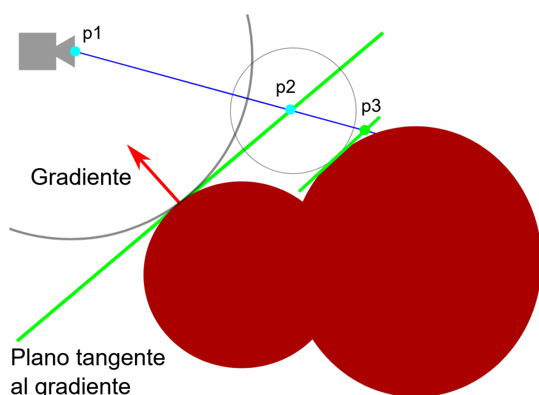
La única diferencia es que si en un paso se detecta que el gradiente actual llevaría más atrás del origen ($dist/ddist > t$ en el algoritmo 5), se realiza el paso correspondiente a *Sphere Marching* en su lugar. El algoritmo 5 detalla este proceso.

Visualmente, el algoritmo funciona como en la figura 3.2a. Dado un punto p_i, el siguiente punto p_{i+1} se encuentra asumiendo que la derivada será constante en el recorrido del rayo. Es decir, encontrará la intersección con superficies planas en una sola iteración (figura 3.2b). Geométricamente, esto equivale a hallar la intersección con el plano tangente a la figura en el punto de su superficie más cercano a p_i.

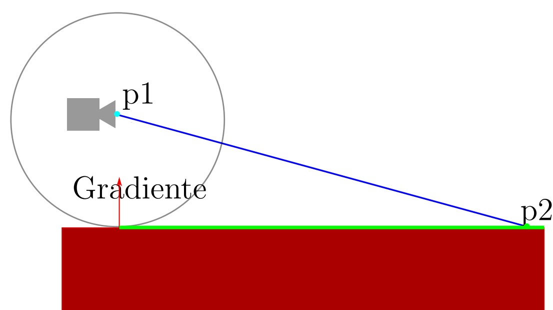
El problema principal de este método es que puede no llegar a converger en tres casos, haciendo que llegue al máximo número de pasos sin encontrar una solución. En primer lugar, el método puede atascarse en un mínimo local. En la figura 3.3a podemos ver un ejemplo de esto, en el que el algoritmo alternará indefinidamente entre *p1* y *p2*. Utilizará al máximo número de pasos tanto si hay una raíz más adelante como en caso contrario.

Algorithm 5: Newton Marching

```
paso = t = 0;
while paso < max_pasos do
    p = origen + d * t;
    dist = sdf(p);
    ddist = ∂sdf(origen + d * t) / ∂t;
    if ddist == 0 then
        return ∞;
    end
    if dist/ddist > t then
        t = t + |dist|;
    else
        t = t - dist/ddist;
    end
    if dist < minDistancia then
        return t;
    end
    paso = paso + 1;
end
return ∞;
```

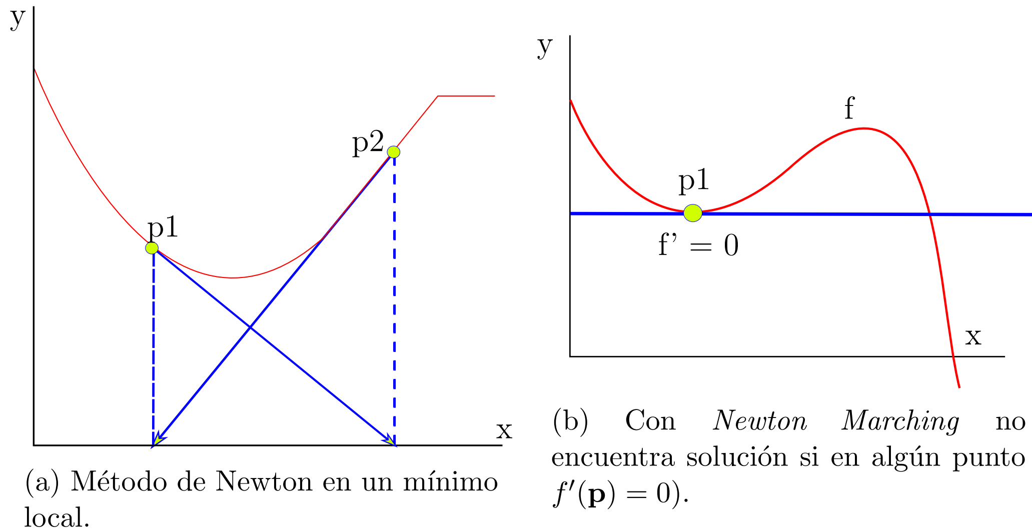


(a) *Newton Marching* intersectando la unión de dos círculos.



(b) *Newton Marching* encuentra la intersección con superficies planas en una iteración.

Figura 3.2: Dos ejemplos de *Newton Marching* en dos dimensiones.



(a) Método de Newton en un mínimo local.

(b) Con *Newton Marching* no encuentra solución si en algún punto $f'(\mathbf{p}) = 0$).

Figura 3.3: Los dos problemas principales del *Newton Marching*.

Finalmente, también fallará si el gradiente es nulo en algún paso (geométricamente, el rayo es paralelo a la figura). Véase la figura 3.3b.

3.2. Forward Newton Marching

Para solucionar los problemas de *Newton Marching* expuestos se ha desarrollado una mejora consistente en la evaluación de $g = f((t_{max} - t)/t)$ mediante *Newton* en lugar de la evaluación directa de la *SDF*.

De esta forma, efectivamente se añade una raíz a la máxima distancia. Esto hace que, si no existía una raíz en el recorrido de un rayo, este tienda a avanzar al final, reduciendo el problema de los mínimos locales expuesto en 3.3a, especialmente para los rayos sin raíces.

Además, si se detecta que la geometría se encuentra tras el origen, se utiliza el paso de *Sphere Tracing* en su lugar, como con el primer algoritmo.

Así, en lugar de *f* y su derivada *f’*, en *Forward Newton Marching* utilizamos *g* y *g’*, que obtenemos a partir de *f* mediante la regla de la cadena. Formalmente, este proceso es el algoritmo 6.

Limitaciones

Estos métodos ya no requieren necesariamente que las funciones intersectadas sean *SDFs* exactas (no es necesario que devuelvan distancias euclídeas), pero sí que requieren que las funciones sean derivables.

Además, pueden tener problemas con modificaciones de la derivada a lo largo del recorrido del rayo, ya que no tienen en cuenta posibles puntos de inflexión de la

función. Esto podría solucionarse con una versión de los métodos que tuvieran en cuenta derivadas de mayor orden.

En cuanto a la eficiencia, esta dependerá en gran medida de que la derivada sea analítica y no numérica. Para hallar la derivada numérica de una *SDF* arbitraria, computando sus derivadas parciales, son necesarias cuatro evaluaciones de la función de distancia.

Algorithm 6: Forward Newton Marching

paso = t = 0;
while $paso < max_pasos$ **do**
  $p = origen + d * t$;
  $dist = sdf(p) * \frac{t - max_dist}{t}$;
  $ddist = \frac{\partial sdf(origen + d * t)}{\partial t}$;
  $ddist = ddist * \frac{t - max_dist}{t} + sdf(p) * \frac{1}{t} - \frac{t - max_dist}{eps^2}$;
  if $ddist == 0$ **then**
    return ∞;
  end
  if $dist/ddist > t$ **then**
    $t = t + |dist|$;
  else
    $t = t - dist/ddist$;
  end
  if $dist < minDistancia$ **then**
    return t;
  end
  if $dist > maxDistancia$ **then**
    return ∞;
  end
  $paso = paso + 1$;
end
return ∞;

Capítulo 4

SDFs

La representación de la geometría mediante *SDFs* permite una gran expresividad más allá de las capacidades de las mallas de polígonos típicas. En esta sección, detallaremos distintas de estas opciones, que hemos implementado en una estructura de clases para su facilidad de uso: permite abstraer los detalles geométricos de la implementación de la interfaz de alto nivel. Se mostrarán ejemplos de las más importantes gracias a la integración de la librería con Mitsuba, que permite introducir todas estas figuras de forma sencilla desde el fichero *xml* de la escena (más detalle en la sección 4.5).

Las *SDFs* implementadas se pueden clasificar en tres tipos:

- Primitivas (sección 4.1). Representan una figura básica (por ejemplo una esfera, un plano, etc.).
- Modificaciones (4.2) a una *SDF*, como rotaciones, escalados o duplicaciones.
- Combinaciones (4.3) de *SDFs*, como uniones, intersecciones o desplazamientos.

En las siguientes secciones se detallarán algunas de estas *SDFs* implementadas.

4.1. Primitivas

4.1.1. Esfera

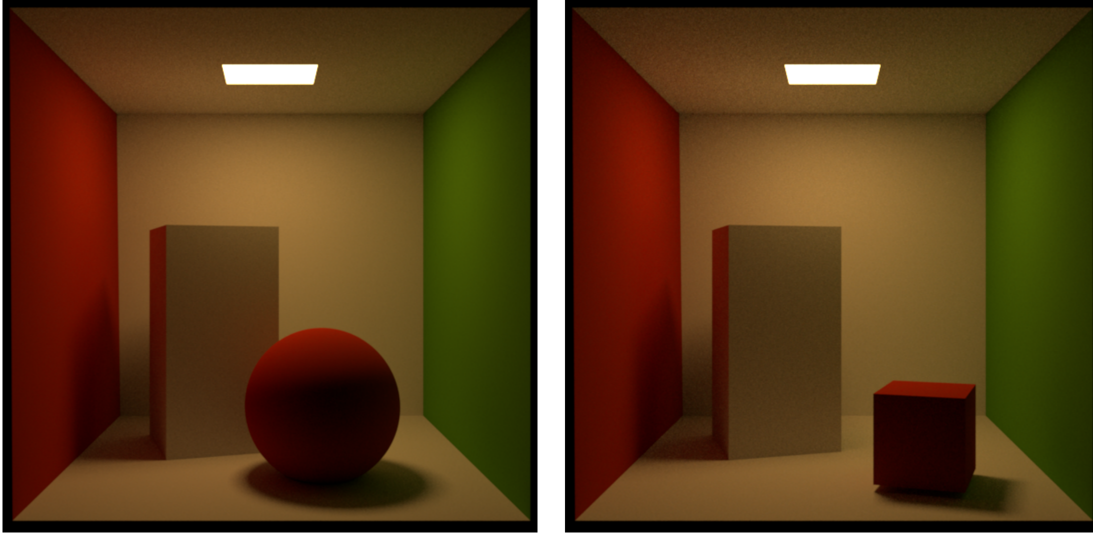
Como se ha comentado en el capítulo de fundamentos 2, una esfera centrada en el origen y con radio r se define mediante la *SDF*:

$$f_{esfera}(\mathbf{p}) = \|\mathbf{p}\| - r \tag{4.1}$$

Para permitir su localización en cualquier punto, se ha añadido el parámetro $\mathbf{c} = (x, y, z)$ que define el centro de la esfera, obteniendo:

$$f_{esfera}(\mathbf{p}) = \|\mathbf{p} - \mathbf{c}\| - r \tag{4.2}$$

Así, renderizando la figura resultante en una *Cornell Box* a través de Mitsuba, obtenemos 4.1a. Cabe destacar que si no se aplican modificaciones posteriores, no es práctico definir esferas mediante *SDFs*, ya que la intersección numérica mediante *Ray Marching* será mucho más lenta que la de *Ray Tracing*, que utiliza la solución analítica del sistema, como se ha explicado en 2.



(a) Esfera en una *Cornell Box*. (b) Prisma rojo en una *Cornell Box*.

Figura 4.1: Dos primitivas definidas por *SDFs* renderizadas en Mitsuba en una *Cornell Box*.

4.1.2. Prisma rectangular

La figura es un prisma rectangular centrado en el origen (su centroide es el origen) y alineado con los ejes de dimensiones $2\mathbf{b}$ ($2b_x, 2b_y, 2b_z$ en sus tres ejes). Para desplazarlo o escalarlo se utilizarán modificaciones (4.2).

La *SDF* del prisma se ha adaptado de la web de Iñigo Quilez [7]. Conceptualmente, divide el problema en sus tres ejes, y halla la distancia en una de sus dos direcciones en cada uno. Finalmente, utilizando el valor absoluto del punto en cada eje, se genera el lado opuesto del prisma en dicho eje. Este concepto es similar al operador de simetría, que se detallará más adelante (4.2.2).

Matemáticamente, la *SDF* del prisma es:

$$f_{prisma}(\mathbf{p}) = \|\mathbf{q_{pos}} + min(max(q_x, max(q_y, q_z)), 0)\|, \tag{4.3}$$

donde $\mathbf{q} = \mathbf{p_{abs}} - \mathbf{b}$, siendo $\mathbf{p_{abs}} = (|p_x|, |p_y|, |p_z|)$. Finalmente, $\mathbf{q_{pos}} = (max(q_x, 0), max(q_y, 0), max(q_z, 0))$.

De nuevo, renderizado en Mitsuba, obtenemos 4.1b.

4.1.3. Plano

El plano infinito, definido por una normal $\mathbf{n}$ y una distancia al origen d, se representa mediante la siguiente *SDF*:

$$f_{plano}(\mathbf{p}) = \mathbf{n} \cdot \mathbf{p} + d \tag{4.4}$$

En 4.2 vemos un ejemplo de un plano horizontal con un material conductor de oro, renderizado en *Mitsuba*. Como en el caso de la esfera, por sí solo no se debería definir mediante una *SDF*, ya que su intersección analítica es muy sencilla.

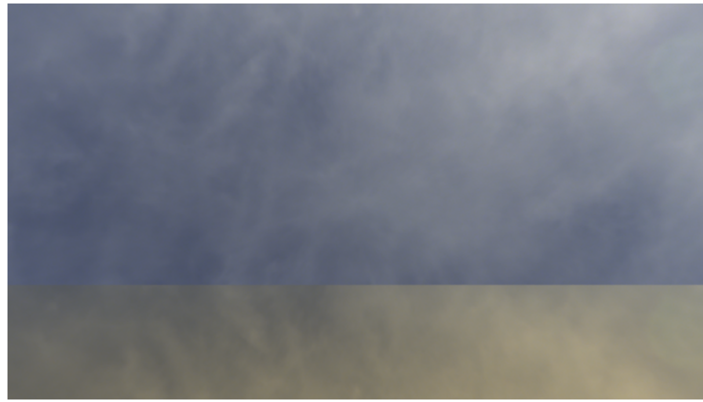


Figura 4.2: Plano horizontal definido por una *SDF* con un material dorado, rodeado de un mapa de entorno.

4.1.4. Línea

Se ha definido una primitiva para las líneas (un cilindro con una hemiesfera del mismo radio en cada extremo, véase 4.3a). Su base se encuentra en el origen, y el cilindro tiene una altura h (en el eje y) y un radio r. La *SDF*, que también se ha adaptado de [7], es la siguiente:

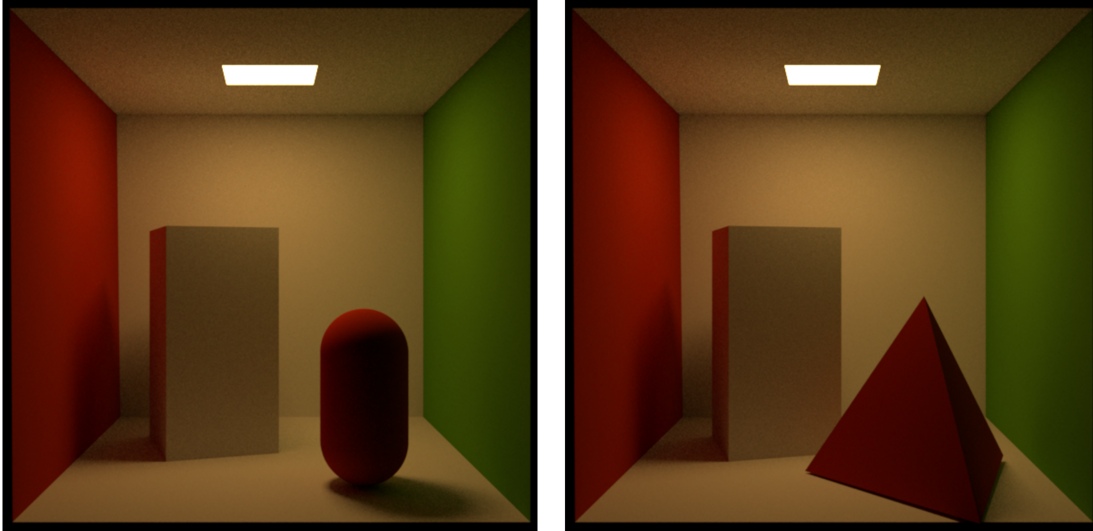
$$f_{linea}(\mathbf{p}) = \|\mathbf{q}\| - r \tag{4.5}$$

Donde $\mathbf{q} = (p_x, p_y clamp, p_z)$ y

$$p_y clamp = \begin{cases} p_y, & \text{si } p_y \leq 0 \\ p_y - h, & \text{si } p_y \geq h \\ 0, & \text{si no} \end{cases}$$

Nótese que 4.5 es muy similar a la *SDF* de la esfera (2.4). Esto se debe a que, matemáticamente, esta *SDF* consiste en una esfera alargada [7].

Finalmente, en la figura 4.3a observamos un ejemplo de esta primitiva en la *Cornell Box* de *Mitsuba*.



(a) Línea en una *Cornell Box*. (b) Tetraedro en una *Cornell Box*.

Figura 4.3: Línea y tetraedro, definidos por *SDFs*, renderizados en Mitsuba en una *Cornell Box*.

4.1.5. Tetraedro

Se ha implementado la *SDF* del tetraedro como base del tetraedro de Sierpinski que se mostrará más adelante (5.2.4). La *SDF* del tetraedro se ha adaptado del siguiente *shader* de *Shadertoy*[1].

Siendo $a(\mathbf{p}) = |p_x + p_y| - p_z$ y $b(\mathbf{p}) = |p_x - p_y| + p_z$, la *SDF* del tetraedro es:

$$f_{tetraedro}(\mathbf{p}) = \begin{cases} \frac{a(\mathbf{p})-1}{\sqrt{3}}, & \text{si } a(\mathbf{p}) \geq b(\mathbf{p}) \\ \frac{b(\mathbf{p})-1}{\sqrt{3}}, & \text{si no} \end{cases}$$

En la figura 4.3b observamos un render en Mitsuba de un tetraedro en la *Cornell Box*.

4.1.6. Ruido de Perlin

El ruido de Perlin es un método muy utilizado en diversas disciplinas para generar patrones similares a procesos naturales. Se puede generar ruido de Perlin de cualquier dimensión, pero en este trabajo se ha utilizado el bidimensional (funciones $\mathbb{R}^2 \rightarrow \mathbb{R}$) y el tridimensional ($\mathbb{R}^3 \rightarrow \mathbb{R}$). Su utilidad viene de la variación gradual del ruido en el espacio: si $f_{perlin}(\mathbf{p})$ devuelve un valor v, $f_{perlin}(\mathbf{p} + (\epsilon))$ devolverá un valor próximo a v para un vector (ϵ) suficientemente pequeño. Esta variación gradual es la clave para sus diversas aplicaciones, desde el modelado de montañas (5.7) hasta la generación de formas orgánicas.

[1] Shader con la *SDF* del tetraedro: https://www.shadertoy.com/view/Ws23zt

En cuanto a su implementación, el ruido de Perlin se obtiene generando números pseudoaleatorios para los gradientes del ruido final, y computando los valores a partir de estos gradientes. Además, el valor final se obtiene combinando distintas octavas de ruido (con mayor frecuencia y menor amplitud), permitiendo distintos niveles de detalle.

En este proyecto se ha utilizado la librería de ruido de Perlin [8], que implementa en C++ la segunda versión del algoritmo (*improved noise*), presentada en 2002 por Ken Perlin [9].

Para integrarlo con el resto del proyecto, se ha implementado como una *SDF*, lo que permite aplicarle las mismas operaciones y combinaciones que al resto de primitivas. En la figura 4.4 se observan tres secciones de un volumen definido directamente por este ruido en tres dimensiones. Cada punto se ha coloreado en función del valor de la *SDF*: rojizo si está dentro, azulado si está fuera. Se puede observar que las dos primeras secciones, con $z = 0$ y $z = 0{,}1$ respectivamente, son mucho más similares que la tercera, con $z = 1{,}5$

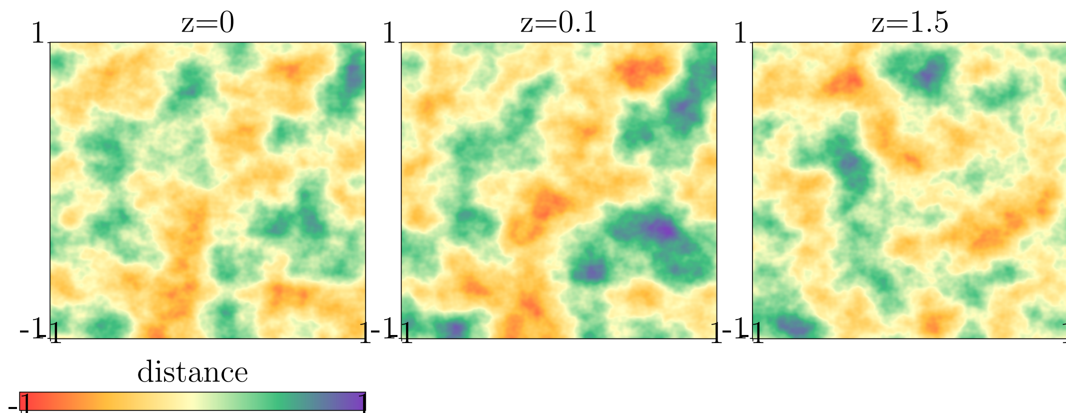


Figura 4.4: Tres secciones de una *SDF* definida por ruido de Perlin en tres dimensiones.

En este trabajo el ruido de Perlin se ha utilizado en diversas escenas que se detallarán más adelante (como 5.5, 5.7 o 4.4).

4.2. Modificaciones

Dada una figura cualquiera representada mediante una *SDF*, existen distintas operaciones que permiten modificarla. Pueden clasificarse en dos grupos principales, según el dato que alteran:

- Modificaciones del punto, antes de la evaluación de la *SDF*, funciones de tipo $f : \mathbb{R}^3 \to \mathbb{R}^3$.
- Modificaciones del valor de distancia devuelto por la *SDF*, de tipo $f : \mathbb{R} \to \mathbb{R}$.

4.2.1. Transformaciones y escalado

Las transformaciones (traslaciones y rotaciones) se agrupan en el primer conjunto, y se aplican a las figuras sencillamente transformando mediante una matriz de cambio de base los puntos antes de la evaluación de la *SDF*. Al devolver las normales se aplica la transformación opuesta.

$$Transformar(f_{SDF}, \mathbf{T}, \mathbf{p}) : f_{SDF}(\mathbf{T} \cdot \mathbf{p}) \tag{4.6}$$

El escalado de la geometría es un caso particular, ya que requiere modificar también la distancia devuelta por la *SDF* con la inversa de la escala aplicada. Sea s el escalar deseado:

$$Escalar(f_{SDF}, s, \mathbf{p}) : s \cdot f_{SDF}(\frac{\mathbf{p}}{s}) \tag{4.7}$$

4.2.2. Simetrías

Las simetrías son uno de los operadores más sencillos pero efectivos. Definidas mediante un plano (un punto y una normal), reflejan toda la geometría en el lado de la normal hacia el otro. Matemáticamente, esto se consigue modificando los puntos que se pasan a la función $f_{SDF}(\mathbf{p})$ de forma que, si se hallan en el lado opuesto a la normal, se reflejen al otro. Sea d_{plano} la distancia al plano:

$$d_{plano} = min((\mathbf{p} - \mathbf{p_{plano}}) \cdot \mathbf{n_{plano}}, 0), \tag{4.8}$$

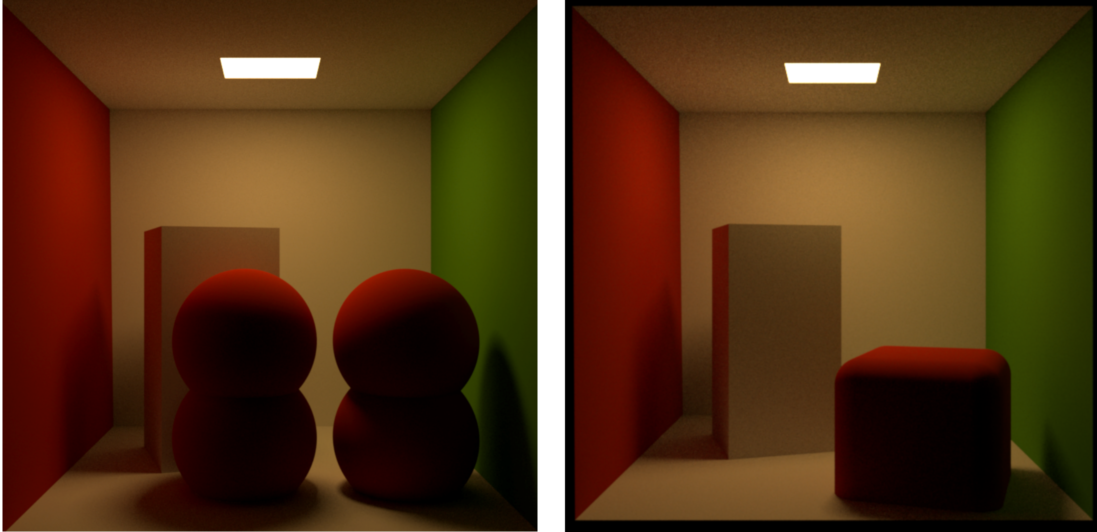
El nuevo punto $\mathbf{p}'$ es:

$$\mathbf{p}' = \mathbf{p} - 2d_{plano} \cdot \mathbf{n_{plano}} \tag{4.9}$$

donde $\mathbf{p_{plano}}$ es el punto del plano y $\mathbf{n_{plano}}$ su normal. Así, al evaluar la función original $f_{SDF}(\mathbf{p}')$, efectivamente se duplicará la geometría presente en el lado de la normal hacia el otro. Nótese que, gracias a la función *min*, los puntos que se encuentren en el lado de la normal no se verán afectados.

Cabe destacar que tras esta transformación se obtiene otra *SDF*, con lo que se puede manipular de la misma forma que la figura original. Esto permite, entre otras cosas, encadenar múltiples reflexiones. Por ejemplo, en la siguiente escena (4.5a), se ha duplicado una esfera mediante dos planos de simetría (uno vertical y otro horizontal).

La principal ventaja del uso de las simetrías frente a la instanciación es que el cálculo necesario para la transformación del punto, que se evaluará en cada paso del *Ray Marcher* utilizado, suele ser mucho menos costoso computacionalmente que el que correspondería a la evaluación de una instancia adicional de una *SDF*.



(a) Esfera duplicada dos veces mediante simetrías.

(b) Prisma redondeado un 15 % de su volumen.

Figura 4.5: Dos modificaciones de *SDFs* renderizadas en *Mitsuba*.

4.2.3. Repetición

El operador de repetición o módulo es similar al de simetría, en el sentido de que sirve para replicar una geometría en el espacio de forma eficiente, sin instanciarla más de una vez. Se consigue obteniendo el módulo de cada uno de los componentes del punto según el vector de repetición **r**:

$$\mathbf{q_i} = \mathbf{p_i} \ \mathbf{mod} \ \mathbf{r_i}, \forall i \in x, y, z \tag{4.10}$$

donde el operador *mod* equivale al resto de la de la división correspondiente, redondeada hacia el cero. Es decir, para dos reales a y b,

$$a \ \mathbf{mod} \ b = a - \left\lfloor \frac{a}{b} \right\rfloor b \tag{4.11}$$

Así, si $\mathbf{r} = (10, 2, 10)$, la *SDF* original se verá repetida cada 10 unidades en los ejes x y z y cada 2 en ele eje y. Por ejemplo, en la siguiente escena (4.6) se puede ver aplicado a una esfera (en este caso, los tres componentes del vector son iguales).

4.2.4. Redondeado

El redondeado de un volumen definido por f_{SDF} se consigue simplemente restando un valor constante r a su *SDF*:

$$f_{redondeado}(\mathbf{p}) = f_{SDF}(\mathbf{p}) - r \tag{4.12}$$

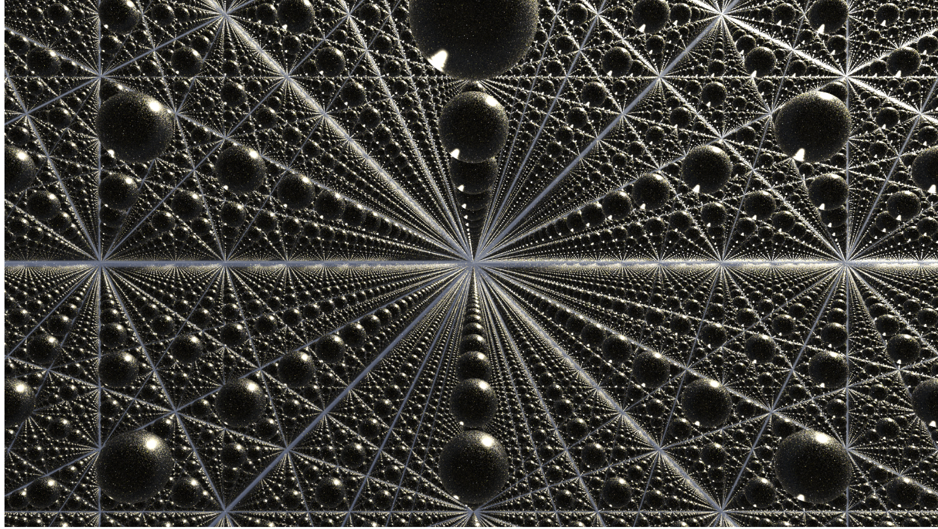


Figura 4.6: Render de una *SDF* de una esfera replicada infinitamente mediante el módulo.

Con esto, se recubre el volumen original con una capa de profundidad r. En 4.5b se puede observar un render en *Mitsuba* de un prisma redondeado.

4.3. Combinaciones de *SDFs*

4.3.1. Unión

La unión ya se ha mencionado en el capítulo de fundamentos 2 por su simplicidad y utilidad. Dadas las funciones *SDF* de dos figuras f_a y f_b, la *SDF* de su unión es:

$$f_{union}(\mathbf{p}) = min(f_a(\mathbf{p}), f_b(\mathbf{p})) \tag{4.13}$$

Ya que la unión resultante también es una *SDF*, este proceso se puede repetir indefinidamente para unir el número necesario de funciones.

Cabe destacar que, a diferencia de operaciones como las simetrías o las repeticiones, la unión requiere la evaluación de ambas funciones internas, por lo que puede ser computacionalmente costosa.

4.3.2. Intersección

Dados dos volúmenes definidos por las *SDFs* f_a y f_b, el volumen de su intersección, definido por $f_{interseccion}$, contendrá únicamente los puntos en el interior de ambos volúmenes. Es decir, para todo punto $\forall \mathbf{p} \in \mathbb{R}^3, f_a(\mathbf{p}) < 0 \wedge f_b(\mathbf{p}) < 0 \implies f_{interseccion}(\mathbf{p}) < 0$.

Esto se consigue simplemente con la siguiente fórmula:

$$f_{interseccion}(\mathbf{p}) = max(f_a(\mathbf{p}), f_b(\mathbf{p})) \tag{4.14}$$

En la figura 4.7a vemos un ejemplo de la intersección de dos esferas.

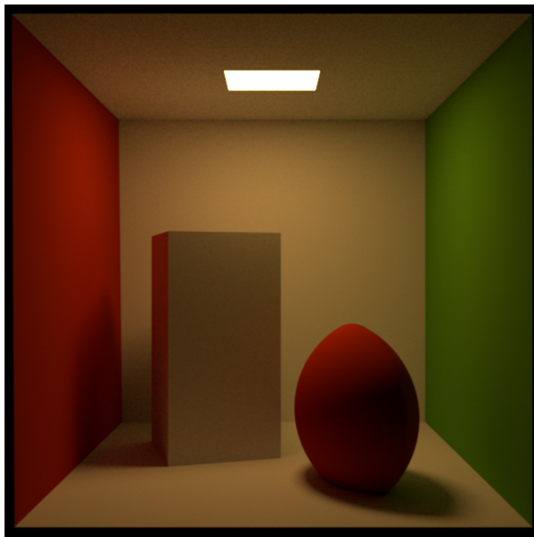
4.3.3. Diferencia

Otra combinación sencilla de *SDFs* es la diferencia. Dadas de nuevo las *SDFs* f_a y f_b, $f_{diferencia}$ equivale a f_a sin los puntos en f_b ($\forall \mathbf{p} \in \mathbb{R}^3, f_a(\mathbf{p}) < 0 \wedge f_b(\mathbf{p}) > 0 \implies f_{diferencia}(\mathbf{p}) < 0$).

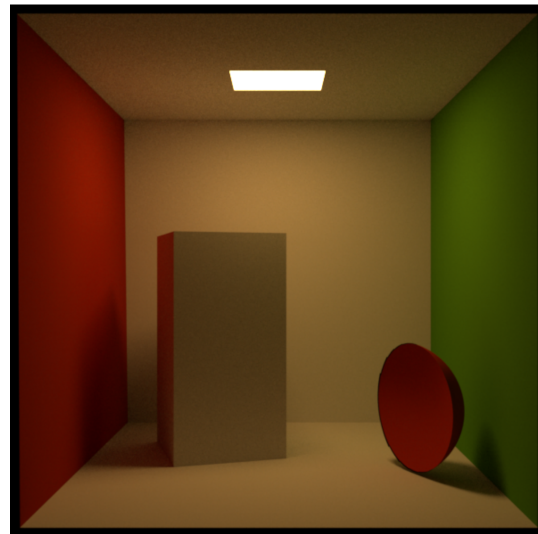
Como se puede observar, esto es equivalente a la intersección entre f_a y $-f_b$, por lo que su implementación es:

$$f_{diferencia}(\mathbf{p}) = max(f_a(\mathbf{p}), -f_b(\mathbf{p})) \tag{4.15}$$

En la figura 4.7b se puede ver la diferencia de las mismas *SDFs* de la intersección de 4.7a.



(a) *SDF* de la intersección de dos esferas.



(b) *SDF* de la diferencia de dos esferas.

Figura 4.7: Dos combinaciones distintas de dos esferas definidas por *SDFs* (intersección y diferencia), renderizadas en *Mitsuba*.

4.3.4. Desplazamiento

Una forma de conseguir detalles de relieve o deformar otro volumen es utilizar el operador de desplazamiento. Matemáticamente consiste en la suma de dos *SDFs*. Para las SDFs f_a y f_b, el desplazamiento es:

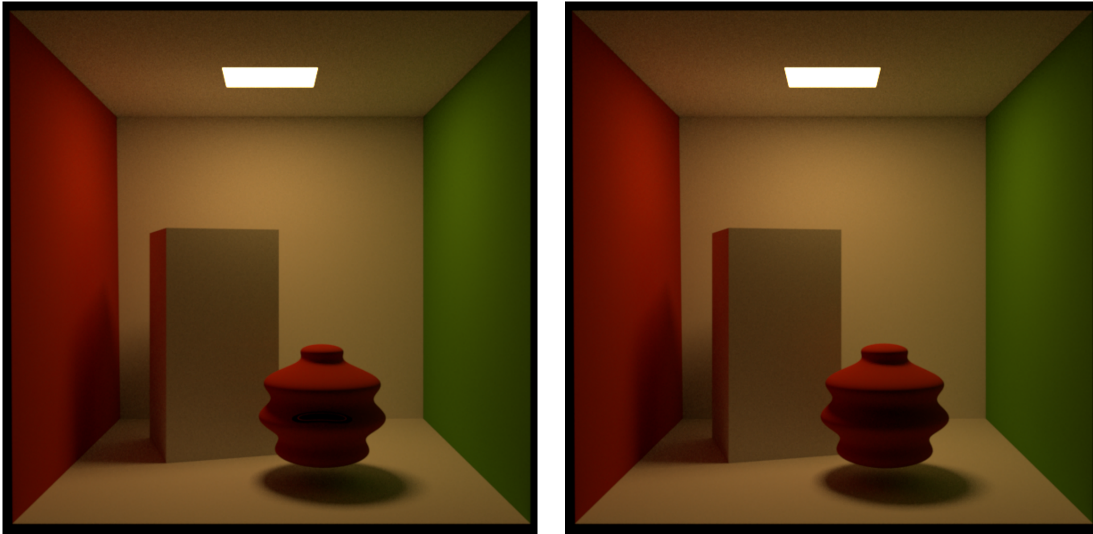
$$f_{desplazamiento}(\mathbf{p}) = f_a(\mathbf{p}) + f_b(\mathbf{p}) \tag{4.16}$$

Este operador puede ser problemático, ya que la función de distancia resultante no es exacta: $f_{desplazamiento}(\mathbf{p})$ puede devolver una distancia distinta de la real. Si la distancia devuelta es menor que la real esto hace que los algoritmos basados en *Sphere Tracing* sean más lentos. Si es mayor, puede llevar a que no se detecten algunas intersecciones, como en la figura 4.8a.

Por eso, se ha implementado un parámetro de seguridad s, con $0 < s < 1$ que reduce la distancia devuelta por esta función:

$$f_{desplazamientoSeguro}(\mathbf{p}) = s(f_a(\mathbf{p}) + f_b(\mathbf{p})) \tag{4.17}$$

Cabe destacar que esto no modifica los ceros de la función, por lo que la superficie es la misma. Con este arreglo, obtenemos figuras como 4.8b.



(a) Esfera desplazada con artefactos al no modificar el valor devuelto (3.7 min).

(b) Esfera desplazada con $s = 0{,}5$ (renderizada en 4.6 min).

Figura 4.8: Esferas (*SDFs*) desplazadas mediante una función seno vertical renderizadas en *Mitsuba*. Comparación del parámetro s de seguridad.

Este desplazamiento es un ejemplo de las funciones $f_{estimador}$ mencionadas en la sección 2.2.3, que no devuelven distancias exactas sino una aproximación.

Este parámetro de seguridad, para conseguir la máxima eficiencia, debería corresponderse con la mínima constante de *Lipschitz* de la función resultante para cada rayo, ya que esta expresa el máximo gradiente de la función en su recorrido, como Galin et al. exploran en profundidad en [1]. Este cálculo, no obstante, escapa al alcance de este trabajo.

4.4. L-Systems, árboles y reflexiones

Se ha implementado, como base para realizar comparaciones, un intérprete de L-Systems [10]. Los L-Systems son una técnica de construcción de geometrías complejas que divide el problema en dos partes. En primer lugar, toma una cadena de símbolos y una serie de reglas de sustitución, y realiza las sustituciones un número de veces. Así, se obtiene una nueva cadena más compleja y con estructura auto-similar. Por ejemplo, si utilizamos el siguiente axioma (cadena original), junto con la siguiente regla:

- **Raíz**: FX
- **Regla**: $X \longrightarrow [+FX] - FX$

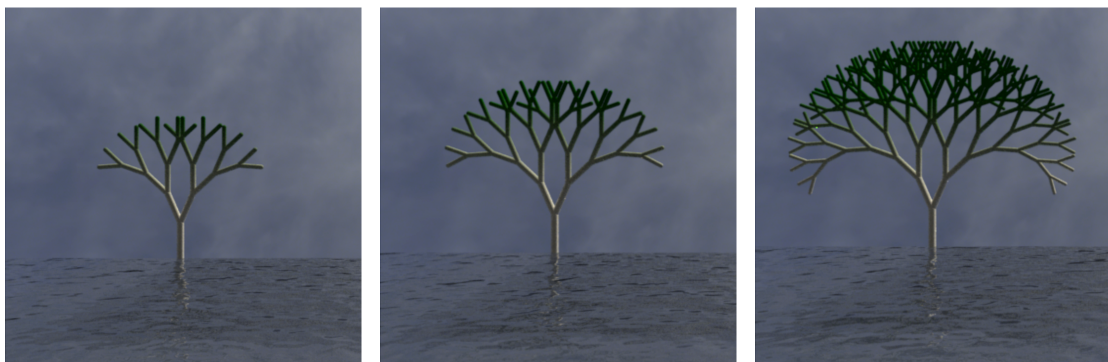
Las tres primeras iteraciones resultan en las cadenas:

1. F[+FX]-FX
2. F[+F[+FX]-FX]-F[+FX]-FX
3. F[+F[+F[+FX]-FX]-F[+FX]-FX]-F[+F[+FX]-FX]-F[+FX]-FX

Tras la generación de la cadena, se interpreta geométricamente para construir la forma. Por ejemplo, los símbolos F significan generar una línea desde la posición actual en adelante; *+* significa rotar hacia un lado, - hacia el contrario; *[* y *]* sirven para guardar y retomar la posición actual, respectivamente.

Así, interpretando el sistema anterior, se generan las figuras de 4.9, con cuatro, cinco y siete iteraciones, respectivamente.

Nótese que, de esta forma, el número de ramas crece exponencialmente con el número de iteraciones (con cada iteración se duplica el número de ramas generadas).



(a) Cuatro iteraciones. (b) Cinco iteraciones. (c) Siete iteraciones.

Figura 4.9: Árboles generados mediante LSystems. Con 4, 5 y 7 iteraciones, respectivamente.

Esto aumenta de forma similar el coste computacional de su intersección, así como su coste en memoria.

No obstante, con una técnica sencilla se puede representar de forma mucho más eficiente. Teniendo en cuenta la simetría de la forma, se puede utilizar la operación de reflexión del espacio explicada en 4.2.2. En lugar de generar cada rama individualmente, se expande una y se refleja en el lado opuesto. Así, el orden de primitivas se convierte en lineal con el número de iteraciones. Por ejemplo, el árbol de 4.11a, con cuarenta iteraciones y simetrías, se ha renderizado en menos de cuatro minutos, cuando 4.9c, con siete iteraciones pero sin simetrías, ha llevado algo más de nueve.

En el gráfico 4.10 se observa la tendencia de ambos métodos experimentalmente. Cada dato se ha obtenido renderizando una escena similar, variando solo el método y el número de iteraciones.

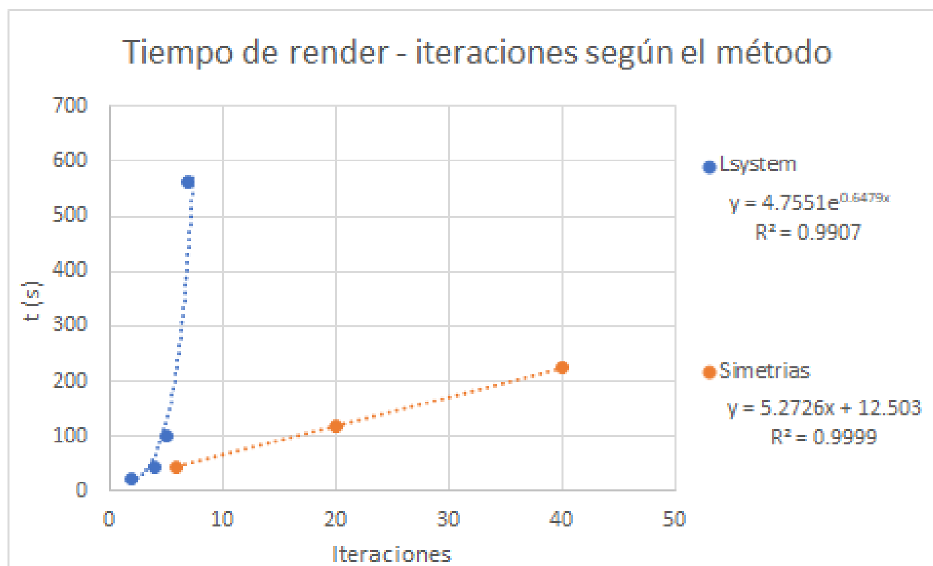
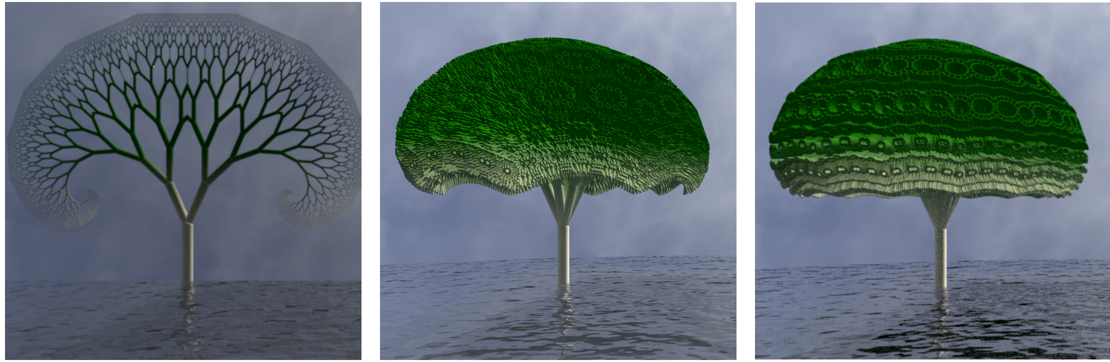


Figura 4.10: Tiempo de render de un árbol (en segundos) con cada método en función del número de iteraciones.

Además, se puede aplicar el mismo concepto en tres dimensiones, aplicando más planos de simetría alrededor del tronco del árbol en cada paso. Esto permite crear distintos árboles tridimensionales sin apenas coste adicional (ya que el cálculo del nuevo punto es mucho menos costoso que la evaluación de una *SDF* completa, como hemos visto en 4.2.2). Por ejemplo, con dos planos de simetría el número de ramas generadas en cada nodo pasa a ser cuatro en lugar de dos, con tres planos, ocho (4.11c), etc.



(a) Árbol generado mediante simetrías, 40 iteraciones.

(b) Árbol generado mediante cuatro planos de simetría.

(c) Árbol generado mediante ocho planos de simetría.

Figura 4.11: Árboles generados mediante Simetrías. Con 1, 4 y 8 planos, respectivamente.

4.5. Diseño Software

4.5.1. Diseño de clases

En alto nivel, el diseño Software seguido para implementar los conceptos explicados se resume en el diagrama 4.12.

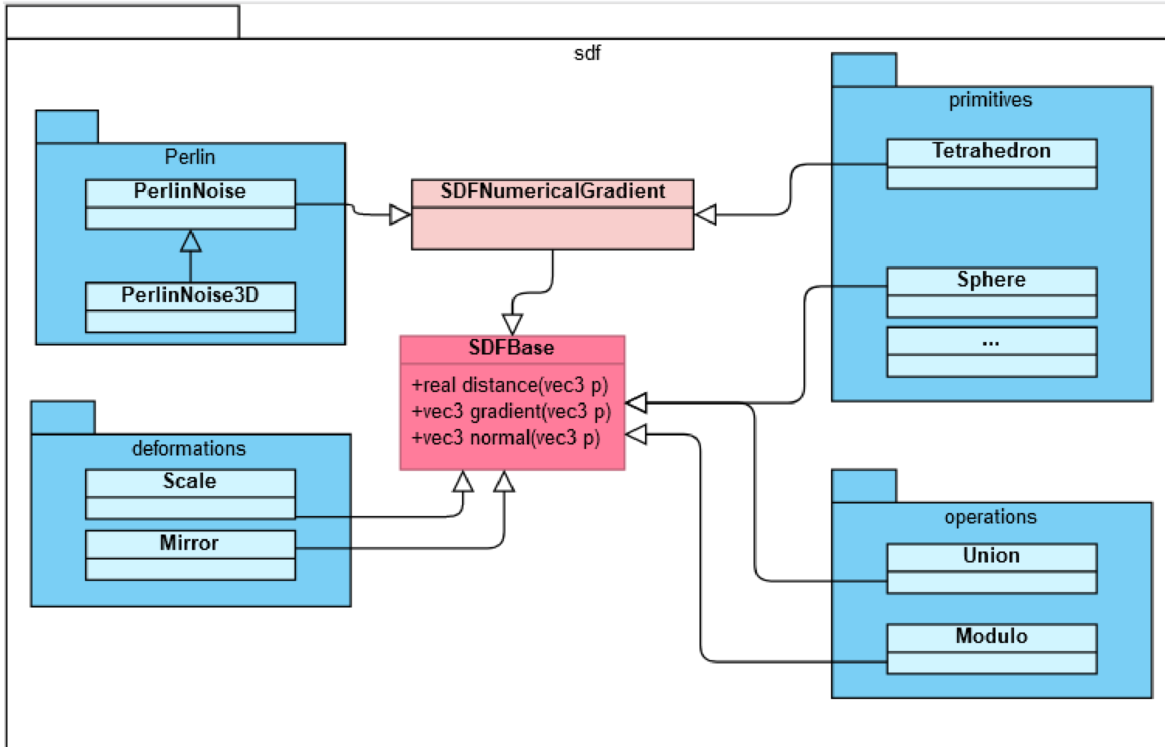


Figura 4.12: Diagrama de clases simplificado del proyecto.

La clase base de todas las figuras definidas por una *SDF* es *SDFBase*. Cuenta con los métodos *distance*, que implementa la *SDF* propiamente dicha; *gradient*, que

devuelve el gradiente (no normalizado) de la función anterior; y *normal*, que devuelve la normal (gradiente normalizado).

Las clases que heredan directamente de *SDFBase*, como *Sphere* o *Mirror* deben implementar el gradiente analíticamente. Idealmente, todas las figuras serían de este tipo, ya que las soluciones analíticas al gradiente son más eficientes. No obstante, en algunos casos es muy complejo, por lo que se proporciona la clase *SDFNumericalGradient* que realiza el cálculo numéricamente (hallando el gradiente en cada coordenada con varias llamadas a la función *distance*). Un ejemplo de este tipo de volúmenes es el ruido de Perlin (clase *PerlinNoise*, sección 4.1.6).

Se han omitido un gran número de clases del diagrama 4.12 por simplicidad, pero todas siguen la estructura mostrada. Además, las clases del proyecto hacen uso del paquete de *utils* 4.13, que cuenta con las clases *vec3*, *Vec4* y *Mat4*, fundamentales para los cálculos geométricos. También contiene *RNG*, con funciones de utilidad de números aleatorios.

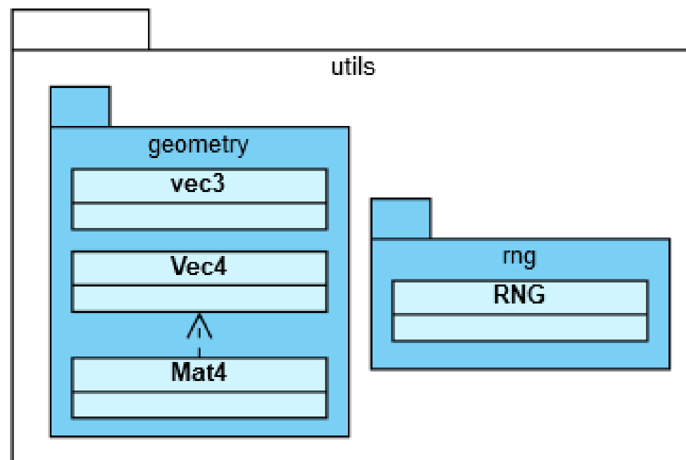


Figura 4.13: Paquete de útiles del proyecto.

4.5.2. PImpl

Por otra parte, a nivel de implementación, se ha aplicado el patrón *PImpl*[2] (*Pointer to Implementation*, o puntero a implementación). El patrón consiste en envolver un puntero oculto a la clase base, en este caso *SDFBase* en una segunda, clase *SDF*. Este patrón tiene distintas ventajas, como permitir el uso de la herencia de *SDFBase* utilizando instancias de *SDF* en lugar de punteros. El diagrama 4.14 ilustra el patrón seguido.

Así, la clase *SDF* contiene un puntero *std::shared_ptr* a una *SDFBase*, y además hereda de la misma, por lo que implementa los métodos explicados. Por una parte, este

[2]PImpl: https://en.cppreference.com/w/cpp/language/pimpl

patrón facilita la manipulación de instancias de cara al programador: permite la copia eficiente (ya que solo se copia un puntero), etc. Por otra parte, esta separación permite el uso de *templates* en la jerarquía de clases, aumentando la eficiencia al permitir realizar optimizaciones en tiempo de compilación.

Por ejemplo, este sistema permite definir el prisma redondeado de 4.5b con una sintaxis tan sencilla como:

```
sdf::SDF box = sdf::Box(); // Prisma
box = sdf::Round(box, 0.25); // Prisma redondeado
```

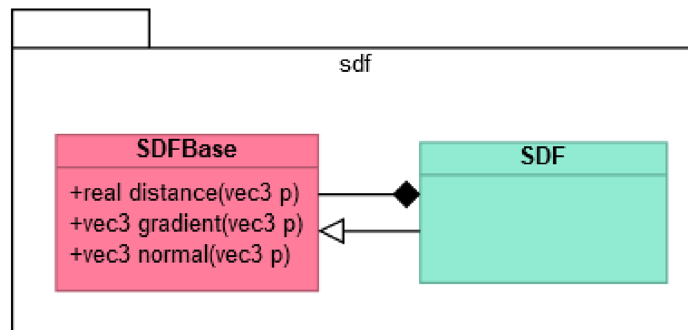


Figura 4.14: Esquema del patrón *PImpl* utilizado.

4.5.3. Integración con *Mitsuba*

En cuanto a la integración del código con *Mitsuba*, se ha elaborado un *wrapper*, *SDFWrapper*, que contiene una instancia de la clase *SDF* y un *Marcher* para su intersección. Hereda de la clase *Shape* de *Mitsuba*, que representa una forma geométrica cualquiera en el renderer, e implementa los métodos para su instanciación (a partir de un fichero *xml*) y su intersección mediante *ray marching*. En la figura 4.15 vemos un resumen de la estructura software.

La interpretación del fichero *xml* permite definir *SDFs* relativamente complejas desde el mismo fichero, mediante la concatenación de modificaciones (de los tipos presentados en las secciones anteriores) sobre la figura original. Por ejemplo, la escena de la esfera repetida de 4.6 se puede representar (omitiendo el material) como:

```
<shape type="sdfwrapper">
  <!-- SDF inicial, una esfera unitaria centrada en el origen -->
  <string name="basesdf" value="sphere"/>
  <!-- Operaciones: -->
  <string name="order" value="modulo scale"/> <!-- orden -->
  <string name="modulo" value="10,10,10"/> <!-- repeticion -->
  <string name="scale" value="25"/> <!-- escala -->
</shape>
```

En la sección de aplicaciones 5.2.1, detallaremos un nuevo material implementado en *Mitsuba*, *SDF_BSDF*, que representa la mezcla de dos materiales delimitados espacialmente por una *SDF*. Se ha denominado así porque en *Mitsuba* se refieren a los materiales como *BSDFs*, o *Bidirectional Scattering Distribution Functions*.

En la figura 4.15 vemos que la implmentación del nuevo material *SDF_BSDF* sigue un esquema similar al de la clase *SDFWrapper*. La nueva clase hereda de la base correspondiente del renderer, *BSDF*, implementando los métodos correspondientes y permitiendo su carga desde el fichero *xml* de la escena de forma similar al *SDFWrapper*.

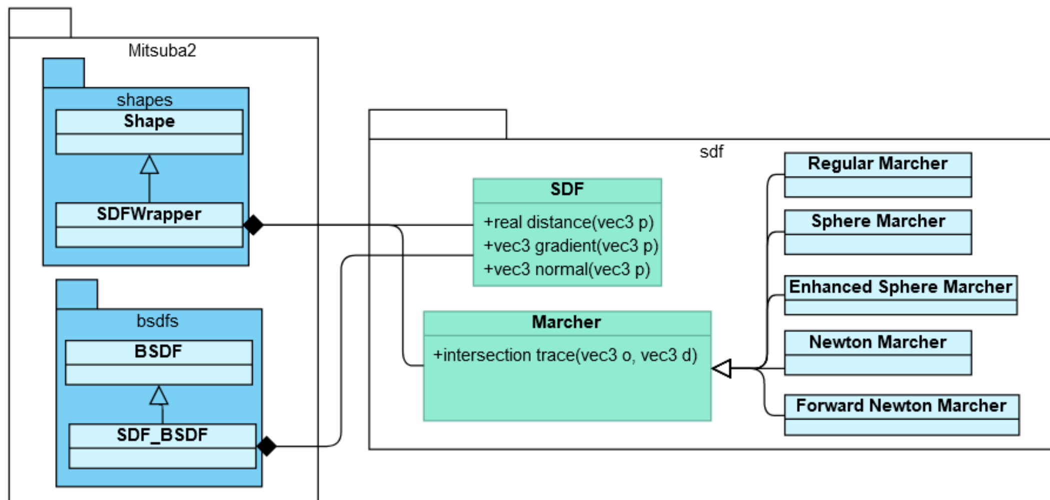


Figura 4.15: Esquema de la integración con *Mitsuba*. El *SDFWrapper* contiene tanto la *SDF* como un *Marcher* para resolver su intersección.

Capítulo 5

Resultados

5.1. Análisis de algoritmos

Para comparar los distintos algoritmos implementados se han elaborado una serie de escenas simples de prueba, y se ha implementado un programa que realiza una proyección ortogonal de la escena y recoge distintos datos de cada rayo, con cada algoritmo. Para cada rayo, se almacena la profundidad de la intersección; la normal en el punto; el error (en la profundidad, cuadrático) con respecto al primer algoritmo, que se considera la referencia; y el número de llamadas a la función *SDF* de la escena, como medida del coste del algoritmo. Esto permite visualizar cada uno de estos datos como un píxel en la imagen final.

A continuación, se analizarán algunas de estas escenas.

5.1.1. Experimento: Prisma redondeado

En la figura 5.1 se han comparado los algoritmos (en cada columna *Regular Marching*, *Sphere Marching*, *Enhanced Sphere Marching*, *Newton Marching* y *Forward Newton Marching*, respectivamente) para un volumen de un prisma redondeado como el mostrado en la sección de *modificaciones* (4.2) en la figura 4.5b. Las filas corresponden a la profundidad de la intersección; el error en la profundidad en cada intersección; la normal en ese punto; y finalmente el número de evaluaciones de la función de distancia f_{SDF}.

Esta figura muestra como todos los algoritmos han conseguido un error nulo en todos sus píxeles excepto el *Newton Marcher*, que presenta unos artefactos en las zonas de discontinuidades de la *SDF* (donde se da un cambio de gradiente en la superficie del objeto).

En la tabla 5.1 se proporciona el promedio de los valores en esta figura, además de el tiempo de intersección medido, en segundos.

En cuanto al número de pasos, el más costoso ha resultado ser el *Regular Marcher*,

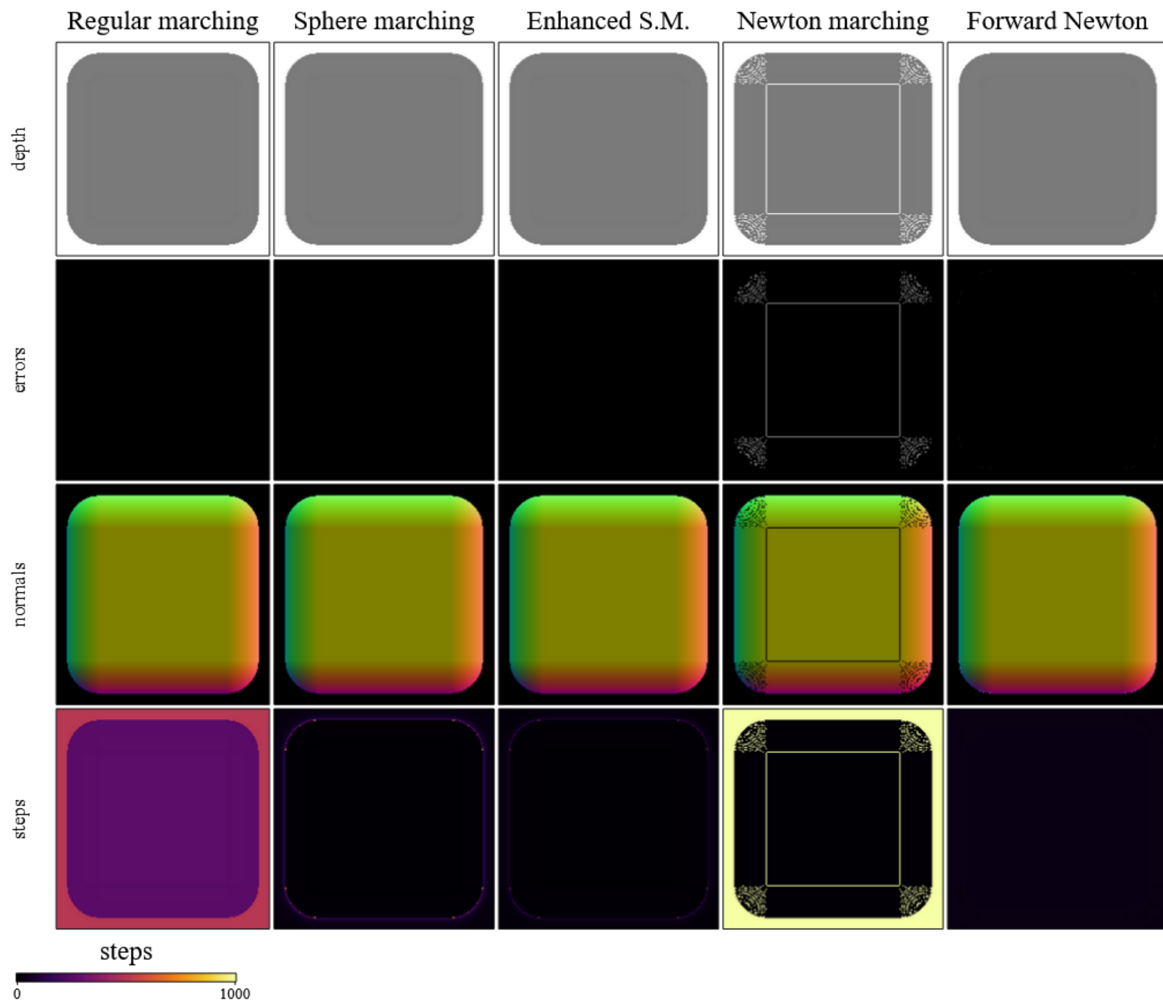


Figura 5.1: Comparación de los algoritmos para un prisma redondeado.

	Regular	Sphere	Enhanced S.	Newton	Forward N.
Pasos	304.861	13.772	9.38034	238.188	27.4862
Tiempo (s)	1.08	0.12	0.12	3.51	0.62
Error	0	4.43E-06	5.12E-06	16.742	0.0928301

Tabla 5.1: Promedio de cada métrica para cada algoritmo para el prisma redondeado.

con un promedio de 305 llamadas por intersección.

El *Sphere Marcher* se encuentra en segundo lugar, con un promedio de 14 evaluaciones de la *SDF*. Aun así, en la visualización de las llamadas en la figura 5.1 se comprueban los problemas con las superficies paralelas explicados anteriormente en la sección 2.2.3: los rayos que pasan cerca de una figura de forma tangente, sin intersectarla, realizan un gran número de pasos.

Este problema se ve reducido con el *Enhanced Sphere Marching*, que consigue reducir el promedio de llamadas a 9 mediante el algoritmo explicado en 2.2.4 (aunque el tiempo se mantiene idéntico).

El *Newton Marcher* ha realizado 238 en promedio, la mayoría en los casos en los que no intersecta, ya que el algoritmo se queda estancado en un mínimo local hasta que alcanza el máximo número de pasos permitido, como se ha explicado en la sección 3.1. Por otra parte, los casos en los que sí intersecta vemos que los alcanza en menos llamadas que el resto de métodos. Aun así, su ineficiencia y su elevado error son evidentes.

Finalmente, se observa que el *Forward Newton Marcher* soluciona el problema de los mínimos locales: ya no se alcanza el máximo número de pasos, ya que el algoritmo tiende a avanzar hacia la distancia máxima. En la figura 5.1 observamos que el número de pasos es completamente homogéneo para todos los rayos. El promedio de llamadas, 27, es algo superior a los mejores algoritmos, pero del mismo orden y con un error superior pero comparable.

5.1.2. Experimento: túnel

Se ha elaborado una escena formada por la unión (4.3.1) de cuatro planos equidistantes a la cámara que se unen a una cierta distancia frente a esta (similar al interior de una pirámide visto desde su base). Este debería ser un caso favorable a los algoritmos propuestos basados en el método de Newton por dos motivos: la derivada de los planos es analítica, por lo que es más eficiente; y la derivada es constante en el recorrido de los rayos hasta la intersección.

En 5.2 observamos como todos los algoritmos han obtenido un error nulo excepto el *Sphere Marching*. En este caso, se debe a que ha alcanzado el máximo número de pasos sin encontrar la raíz. El *Enhanced Sphere Marching*, por otra parte, sí que ha llegado a intersectar la escena, aunque prácticamente alcanzando el máximo número de pasos.

Como era previsible, los algoritmos de Newton han encontrado las raíces muy rápidamente (como se ha explicado anteriormente en la figura 3.2b). En la tabla 5.2, vemos como los tres han alcanzado un error prácticamente nulo en unos pocos pasos.

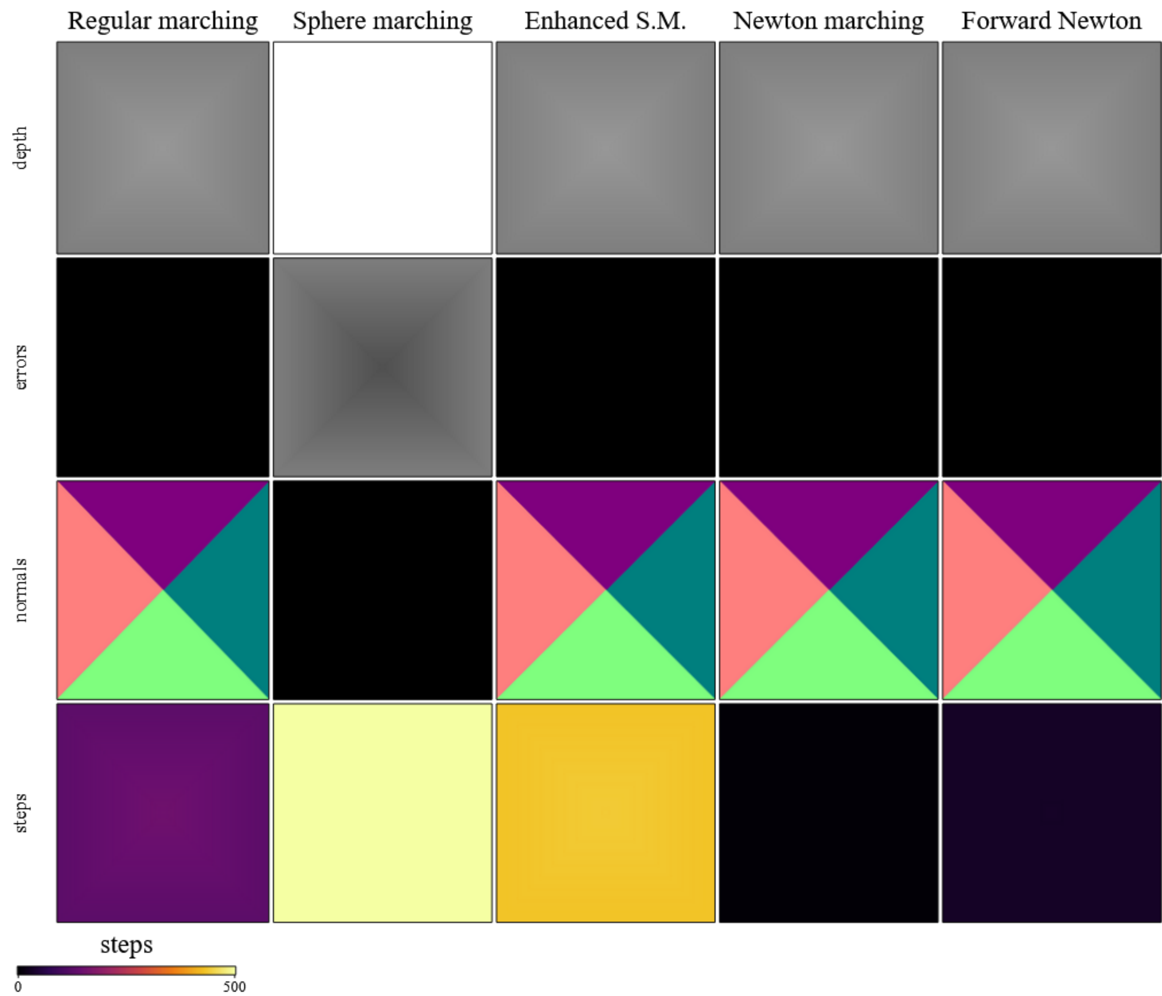


Figura 5.2: Comparación de los algoritmos para la unión de cuatro planos.

	Regular	Sphere	Enhanced S.	Newton	Forward N	Forward N2
Pasos	136.104	500	433.704	2	31.0072	6
Tiempo (s)	0.54	2.77	3.44	0.06	0.52	0.12
Error	0	218650	0.008832	2.53E-05	2.52E-05	2.53E-05

Tabla 5.2: Promedio de cada métrica para cada algoritmo para la unión de cuatro planos.

5.1.3. Experimento: Cambios en la derivada

Como hemos visto, los algoritmos basados en el método de *Newton* funcionan especialmente bien cuando la derivada es constante. Por otra parte, funciones con un gran número de cambios en la derivada, como la generada por las repeticiones (figura 4.6), serán previsiblemente problemáticas. Un ejemplo más sencillo es la unión de volúmenes a distintas distancias.

Se ha elaborado una *SDF* mediante la unión de cinco esferas, de izquierda a derecha, cada una algo más lejos de la cámara que la anterior. Así, los rayos que intersecten con las más lejanas, pasarán antes cerca de otras esferas más cercanas. Los métodos de *Newton* podrían encontrar problemas por dos motivos: los mínimos locales al utilizar la derivada de la esfera más cercana en lugar de la que se va a intersectar; y los puntos no derivables (picos generados por la función de mínimo de la unión).

En la figura 5.3 vemos los resultados de este experimento. Aunque en la mayor parte de los puntos los algoritmos de *Newton* han encontrado las raíces buscadas, en muchos casos han obtenido la raíz equivocada (intersección al otro lado de las esferas). Esto se puede observar claramente con el color de las normales en las esferas más lejanas (a la derecha).

En este caso, además, como se puede ver en la tabla 5.3 el *Forward Newton* ha obtenido un error mayor que el algoritmo básico.

	Regular	Sphere	Enhanced S.	Newton	Forward N
Pasos	472.917	12.4588	9.90928	877.792	181.529
Tiempo (s)	2.62	0.12	0.13	21.48	5.03
Error	0	4.26E-07	0.030327	0.06056	2.04885

Tabla 5.3: Promedio de cada métrica para cada algoritmo para la unión de varias esferas a distintas distancias.

5.1.4. Análisis: conclusiones

Como hemos visto, el algoritmo de *Forward Newton* presenta un error relativamente bajo en la mayoría de las escenas (exceptuando casos concretos como 5.1.3), y en algunas (con superficies planas) funciona especialmente bien.

No obstante, en muchas presenta un rendimiento inferior a los algoritmos basados en *Sphere Tracing*. En la tabla 5.4 se muestra la mediana de cada métrica en las 17 escenas de prueba elaboradas (distintas *SDFs* similares a las de los dos apartados anteriores). Como se puede observar, el *Forward Newton Marcher* realiza en general del orden de 10 veces más pasos que el *Sphere Tracing*.

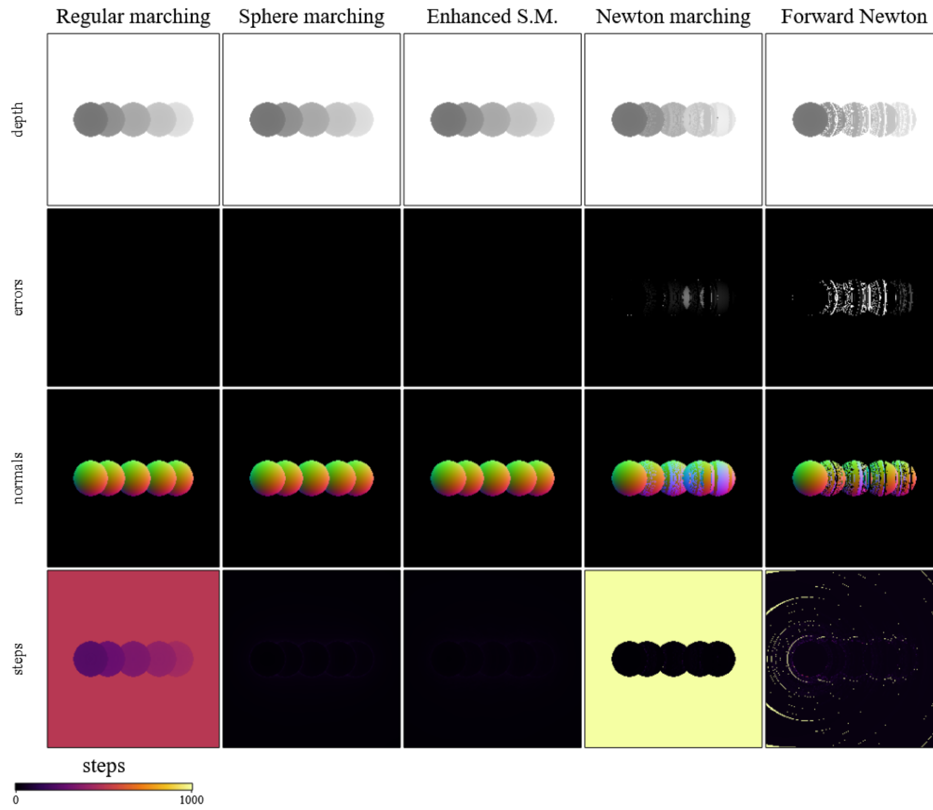


Figura 5.3: Comparación de los algoritmos para la unión de esferas a distintas distancias.

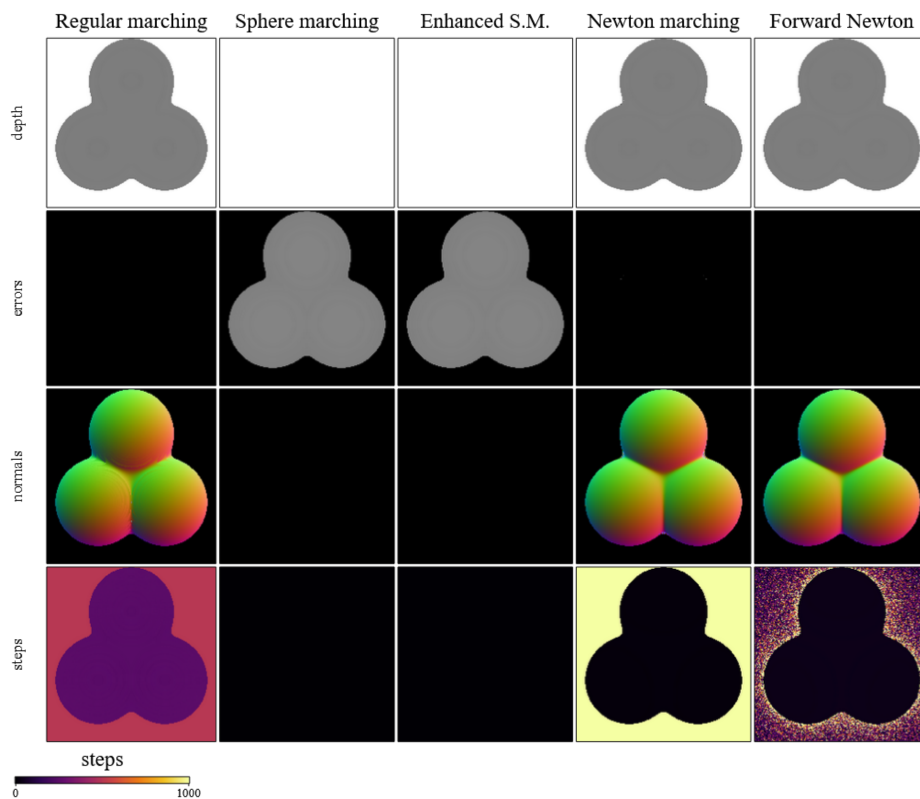


Figura 5.4: Definición de la unión mediante el producto.

Comprobamos también que el *Enhanced Sphere Tracing* es consistentemente mejor que su predecesor en todos los aspectos. El error, aunque ligeramente superior, es despreciable.

	Regular	Sphere	Enhanced S.	Newton	Forward N
Pasos	440.579	11.3928	8.50484	714.562	142.442
Tiempo (s)	2.62416	8.5711	0.127701	21.4848	12.3632
Error	0	1.2533E-06	5.12E-06	0.01184	0.00179

Tabla 5.4: Mediana de cada métrica para cada algoritmo en las 17 escenas de prueba elaboradas.

Por otra parte, los algoritmos propuestos presentan ciertas ventajas frente a los tradicionales. Por ejemplo, permiten la intersección de funciones de distancia no exacta, como la definición de uniones (exclusivas, la intersección de los volúmenes será hueca en el volumen final) mediante el producto. Sean f_a y f_b dos *SDFs*, su unión exclusiva se puede definir como:

$$f_{producto}(\mathbf{p}) = f_a(\mathbf{p}) \cdot f_b(\mathbf{p}) \tag{5.1}$$

El resultado ya no será una *SDF* euclídea, ya que los valores devueltos no corresponden a distancias, pero los puntos pertenecientes a la superficie se mantienen (si $f_a = 0$ o $f_b = 0$, $f_a \cdot f_b = 0$). Definiendo la unión de esta forma, obtenemos la figura 5.4.

Como se puede observar, los algoritmos de *Sphere Tracing* no pueden hallar las intersecciones, pero el resto sí.

Así, aunque en general los primeros algoritmos son más robustos, el *Forward Newton Marching* presenta cualidades únicas y podría permitir nuevas formas de definir la geometría, mediante funciones no necesariamente de distancia pero derivables, con distintos operadores y cualidades de los presentados por las *SDFs*.

Sería necesaria una investigación más exhaustiva para desarrollar y estudiar este tipo de geometrías.

5.2. Aplicaciones

Ya se ha visto en las anteriores secciones la utilidad de las *SDFs* para definir geometría, pero estas funciones pueden utilizarse con muchos otros fines. En este trabajo, presentamos dos muestras de estas posibles aplicaciones.

5.2.1. Delimitación de materiales

Como se ha mencionado brevemente en la sección 4.5, se ha elaborado un *plugin* en *mitsuba*, *sdfbsdf*, que permite delimitar espacialmente distintos materiales en una misma figura. Por ejemplo, en la esfera de la figura 5.5 se ha realizado una mezcla de un material difuso rojo y uno conductor (especular). Se han delimitado por una *SDF* de ruido de Perlin en tres dimensiones. En cada punto, se obtiene el valor de la *SDF* y se devuelve la mezcla correspondiente de los dos materiales.

Se define *dist*, un parámetro de distancia del material que codifica la suavidad de la mezcla entre los dos materiales. Es decir, si *dist* es cero, el cambio es directo. El material devuelto depende de la siguiente función:

$$material(p) = \begin{cases} mat1, & \text{si } sdf(p) \geq dist \\ mat2, & \text{si } sdf(p) \leq -dist \\ lerp(mat1, mat2, sdf(p)/dist) & \text{si no} \end{cases}$$

Así, si la *SDF* devuelve un valor mayor que dicho parámetro de distancia, se devuelve el primer material. Si devuelve un valor menor que -*dist*, el segundo, y si la distancia es intermedia, la interpolación lineal de ambos.

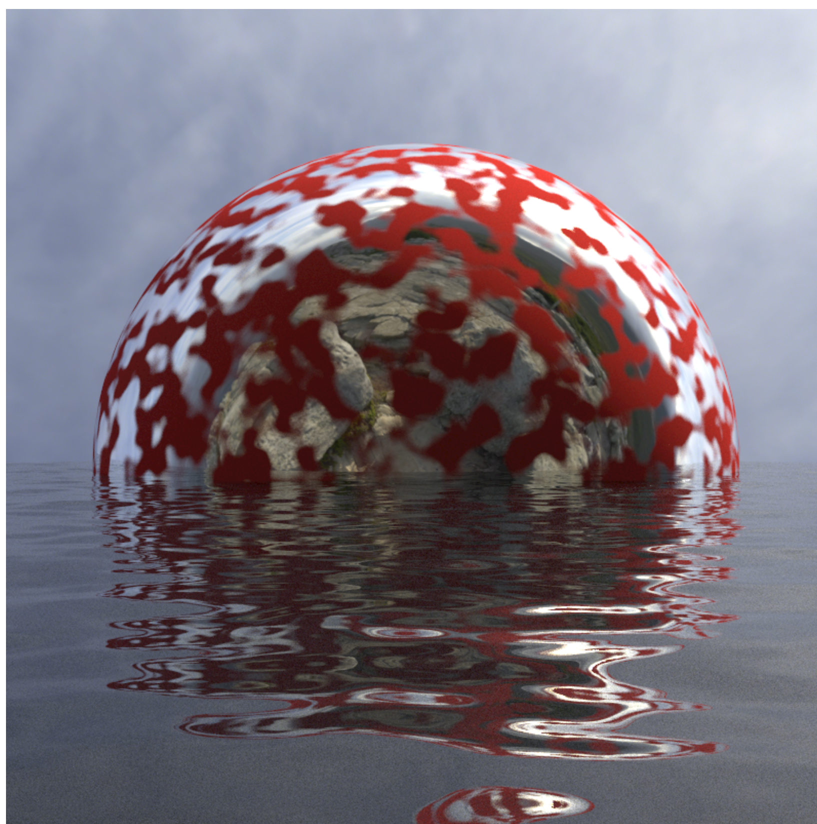


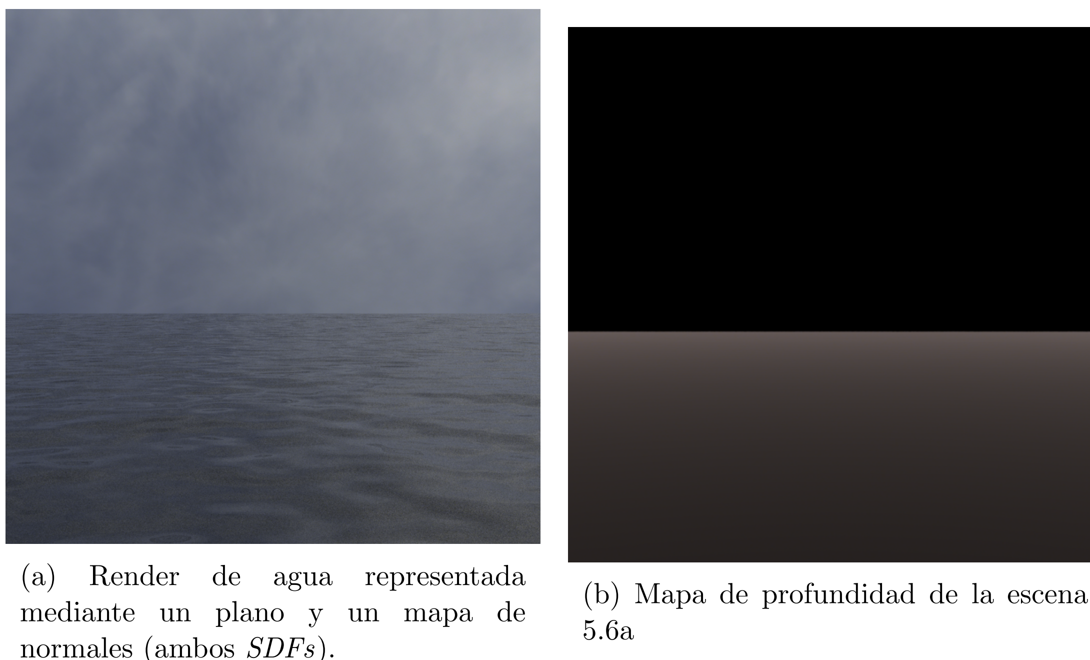
Figura 5.5: Mezcla de dos materiales limitados por ruido en 3D.

5.2.2. Mapa de normales

Una opción para ganar eficiencia es simplificar la geometría de la escena y relegar el detalle a las normales, que utilizará el algoritmo de iluminación. Este concepto se utiliza comúnmente en los métodos tradicionales mediante los mapas de normales, que modifican la normal a partir del valor almacenado en memoria en una textura [11].

Con *SDFs* la implementación es más directa: se representa la geometría mediante dos *SDFs* distintas. La primera, más simple y rápida, se evaluará en el proceso de *Ray Marching*. La segunda, similar pero con más detalles de alta frecuencia, se utilizará para obtener la normal en el punto.

En 5.6 vemos una aplicación típica de esta técnica. La geometría de la escena está compuesta por un único plano horizontal sin alterar, pero las normales utilizadas para la iluminación se determinan mediante un ruido de *Perlin* bidimensional como el descrito en la sección 4.1.6. En 5.6a se observa la apariencia de agua resultante, mientras que en 5.6b se observa el mapa de profundidad de la escena (cada punto se colorea en función de su distancia: más cerca, más oscuro, más lejos, más blanco), donde vemos que la geometría es efectivamente plana.



(a) Render de agua representada mediante un plano y un mapa de normales (ambos *SDFs*).

(b) Mapa de profundidad de la escena 5.6a

Figura 5.6: Escena de agua representada mediante un plano y un mapa de normales (ambos *SDFs*). Render y mapa de profundidad, respectivamente.

En la figura 5.7 vemos otro caso de uso algo más avanzado de este concepto. La *SDF* de la geometría es un plano desplazado mediante cinco octavas de ruido de Perlin con distintas amplitudes. Por otra parte, para las normales se utilizan doce octavas, dando a la superficie una apariencia más detallada sin aumentar prácticamente el

coste. El agua es similar, como en la figura 5.6, la *SDF* de la geometría es un plano, y el desplazamiento se ha aplicado solo a las normales.

En esta figura también se puede observar otra aplicación de la mezcla de materiales implementada: se ha delimitado la roca de la nieve mediante un plano horizontal.

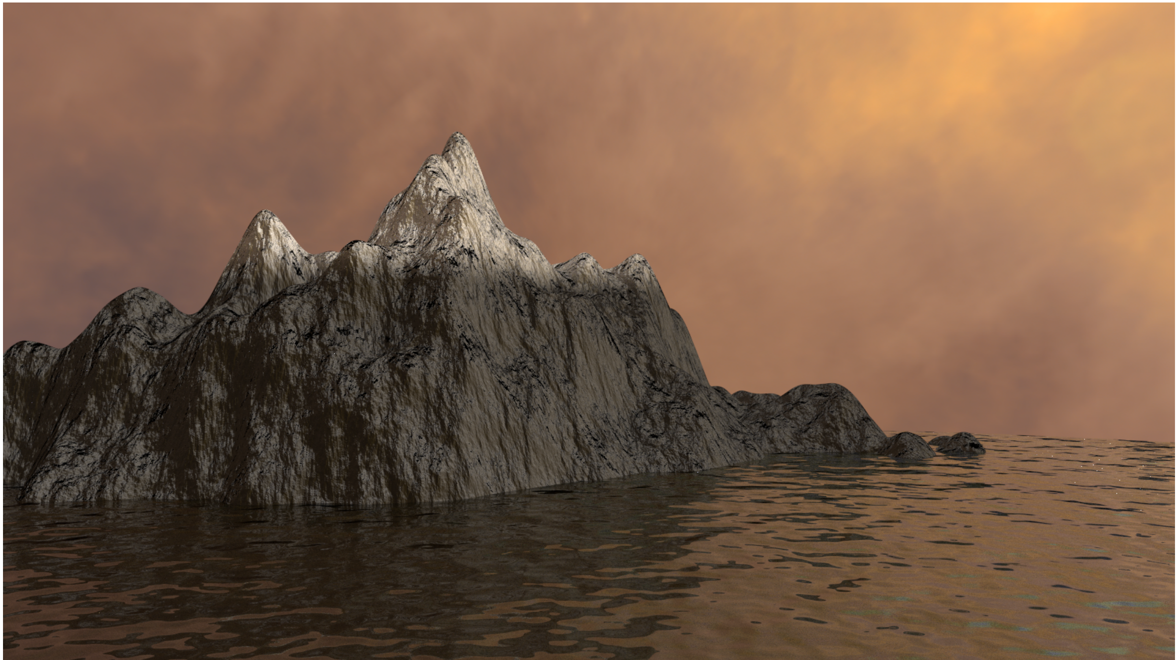


Figura 5.7: Montañas generadas con ruido de Perlin (renderizado en 2,4 h).

5.2.3. Rascacielos

En la figura 5.8 se puede observar otro ejemplo del uso de simetrías y repeticiones de dominio para multiplicar la complejidad de la geometría de forma eficiente. Las simetrías se utilizan en los ejes horizontales (x y z) para multiplicar las columnas principales. La repetición de dominio (módulo limitado, no infinito), se aplica en el eje vertical para simular los distintos pisos multiplicando el segundo prisma, dispuesto en horizontal. Finalmente, un solo prisma en el centro del edificio modela las ventanas, con un material dieléctrico. El segundo edificio es la replicación mediante una última simetría de todo lo anterior.

Las paredes, como las montañas de la sección anterior, cuentan con un desplazamiento de las normales para simular la textura del cemento.

5.2.4. Tetraedro de Sierpinski

El tetraedro de Sierpinski es uno de los fractales tridimensionales clásicos. Es una generalización del triángulo de Sierpinski a tres dimensiones. En cada iteración, el tetraedro original se reduce a la mitad de tamaño respecto a uno de sus vértices

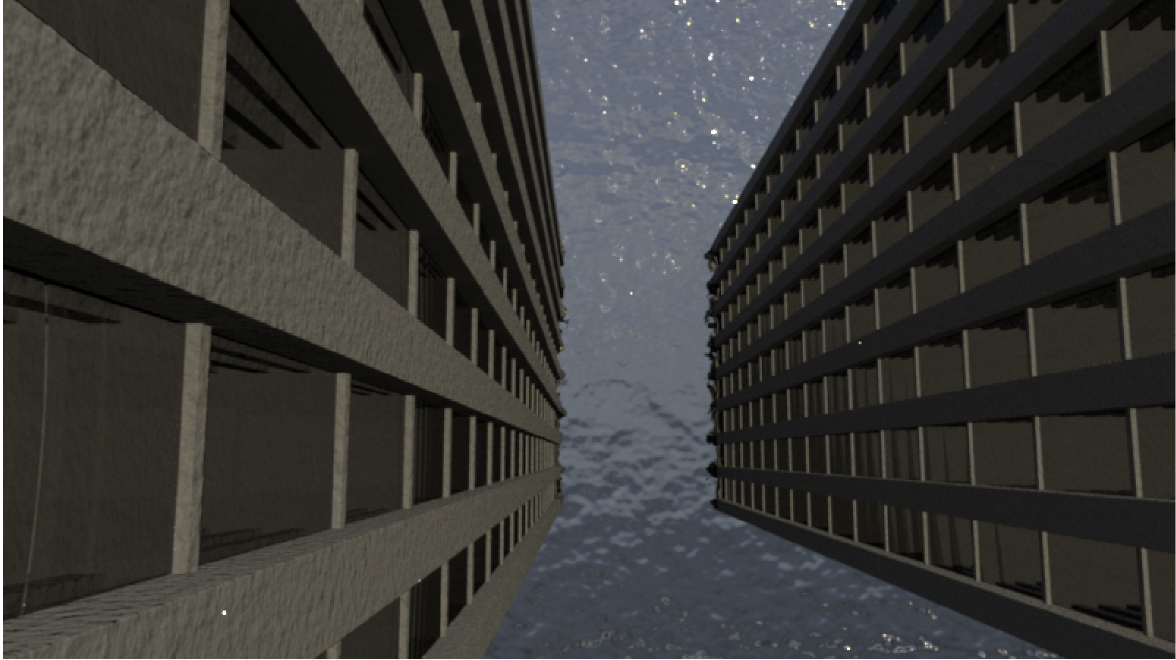


Figura 5.8: Rascacielos generados mediante tres prismas replicados un gran número de veces (renderizado en una hora).

(operación de *escala*) y se duplica en la dirección de los otros tres (operación de simetría). Así, se consigue una nueva geometría con forma externa de tetraedro, con la que se repite el proceso recursivamente. En la figura 5.9 se observa un tetraedro de sierpinski con seis iteraciones en una *cornell box*.

Como en el caso de los árboles expuestos en la sección 4.4, al utilizarse la simetría el coste aumenta linealmente con el número de iteraciones, en lugar de exponencialmente si se usaran triángulos o instanciaciones de tetraedros.

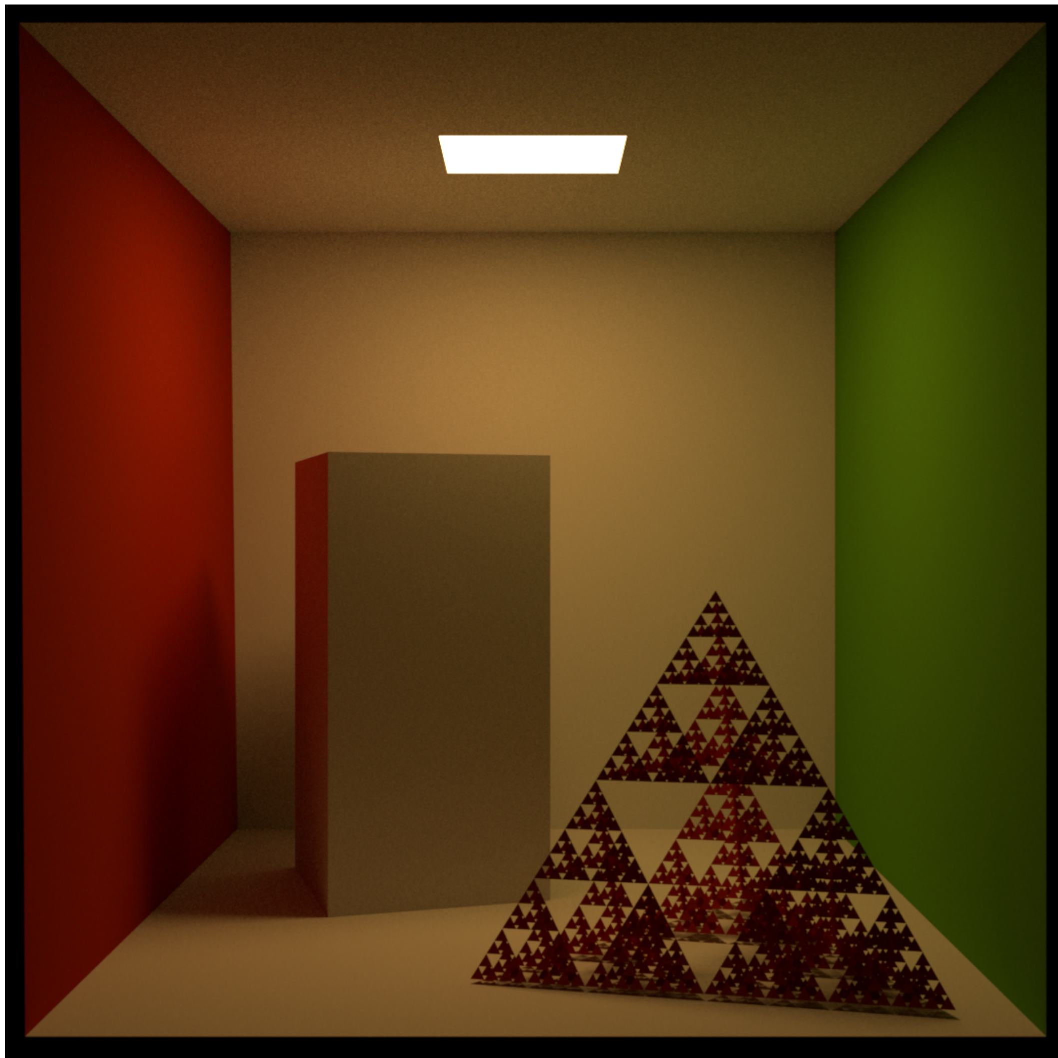


Figura 5.9: Tetraedro de Sierpinski con seis iteraciones, *SDF* de coste lineal con el número de iteraciones gracias a las simetrías.

Capítulo 6

Conclusiones

En este trabajo se han mostrado muchas de las diversas ventajas de las *SDFs* como forma de representar la geometría. Además de las distintas formas de manipularse y combinarse, se han detallado los fundamentos matemáticos subyacentes a estas operaciones. La sencillez de estas operaciones proporciona un control intuitivo de estas geometrías al programador.

Se ha implementado una librería con características modernas de *C++* para la definición y manipulación de estos volúmenes. La estructura de clases organizada ha permitido mostrar una interfaz simple de utilizar y una mayor abstracción que la que se suele alcanzar en entornos como *Shadertoy* [4], donde las *SDFs* se implementan directamente como funciones.

Además, se han explicado, implementado y comparado los algoritmos más populares para la intersección de estas geometrías: *Sphere Tracing* y su versión mejorada, *Enhanced Sphere Tracing*. Se han propuesto dos nuevos algoritmos basados en el método de *Newton*.

Se han analizado todos estos algoritmos con una variedad de escenas, y se han discutido tanto las limitaciones de los nuevos métodos como sus ventajas. Los métodos de Newton son menos eficientes en algunos casos, y presentan problemas con las variaciones en las derivadas y las funciones no derivables. Por otra parte, cuentan con una gran eficiencia en superficies planas y permiten intersectar funciones que no devuelven distancias exactas.

Esta última propiedad podría inspirar trabajo futuro en el que se desarrolle una colección de primitivas y operaciones similar a la de las *SDFs* pero con funciones derivables en lugar de euclídeas.

Finalmente, se ha integrado la librería elaborada con el renderer *Mitsuba*, permitiendo renderizar escenas con modelos de materiales y emisores avanzados. Además, se han mostrado una serie de aplicaciones prácticas de las *SDFs* al margen de la propia definición de geometría, incluyendo la delimitación de materiales y la

representación de mapas de normales para ofrecer detalle visual sin demasiado coste computacional.

Capítulo 7

Bibliografía

[1] Eric Galin, Eric Guérin, Axel Paris, and Adrien Peytavie. Segment Tracing Using Local Lipschitz Bounds. *Computer Graphics Forum*, 2020. https://hal.archives-ouvertes.fr/hal-02507361.

[2] John Hart. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12, 06 1995.

[3] Benjamin Keinert, Henry Schäfer, Johann Korndörfer, Urs Ganse, and Marc Stamminger. Enhanced Sphere Tracing. In Andrea Giachetti, editor, *Smart Tools and Apps for Graphics - Eurographics Italian Chapter Conference*. The Eurographics Association, 2014.

[4] Inigo Quilez and Pol Jeremias. https://www.shadertoy.com/.

[5] Csaba Bálint and Gábor Valasek. Accelerating sphere tracing. 11 2018.

[6] Wikipedia. Método de newton — wikipedia, la enciclopedia libre, 2021. https://es.wikipedia.org/w/index.php?title=M%C3%A9todo_de_Newton\&oldid=133881648.

[7] Iñigo Quilez. 3d sdfs. https://iquilezles.org/www/articles/distfunctions/distfunctions.htm.

[8] Reputeless. Perlinnoise. https://github.com/Reputeless/PerlinNoise.

[9] Ken Perlin. Improving noise. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 681–682, 2002.

[10] Paul Bourke. Lsystem user notes, Jul 1991. http://paulbourke.net/fractals/lsys/.

[11] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically based rendering from theory to implementation*. 2017.

[12] Jan Novák and Carsten Dachsbacher. Rasterized bounding volume hierarchies. *Computer Graphics Forum*, 31(2pt2):403–412, 2012. `https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2012.03019.x`.

Lista de Figuras

2.1. Rasterización y *Ray Tracing*. 5
2.2. Esquema de un volumen definido por una función f *SDF*. En azul, el exterior del volumen (puntos $\mathbf{o}$). En violeta (puntos $\mathbf{i}$), el interior. En negro ($\mathbf{s}$), la superficie. 7
2.3. Intersecciones con *Ray Marching* de paso fijo y *Regular Marching*. El volumen a intersectar, definido por una *SDF*, es el rojo. Se muestra un rayo de la cámara en azul, y los distintos puntos correspondientes a cada iteración de los algoritmos. 9
2.4. Casos del algoritmo de *Sphere Tracing*. 11
2.5. Comparación de *Sphere Tracing* y *Enhanced Sphere Tracing* para un mismo caso. 13
2.6. Detección de un posible error en *Enhanced Sphere Tracing*, $w = 1{,}5$. . . 13

3.1. Método de Newton. 16
3.2. Dos ejemplos de *Newton Marching* en dos dimensiones. 17
3.3. Los dos problemas principales del *Newton Marching*. 18

4.1. Dos primitivas definidas por *SDFs* renderizadas en Mitsuba en una *Cornell Box*. 22
4.2. Plano horizontal definido por una *SDF* con un material dorado, rodeado de un mapa de entorno. 23
4.3. Línea y tetraedro, definidos por *SDFs*, renderizados en Mitsuba en una *Cornell Box*. 24
4.4. Tres secciones de una *SDF* definida por ruido de Perlin en tres dimensiones. 25
4.5. Dos modificaciones de *SDFs* renderizadas en *Mitsuba*. 27
4.6. Render de una *SDF* de una esfera replicada infinitamente mediante el módulo. 28
4.7. Dos combinaciones distintas de dos esferas definidas por *SDFs* (intersección y diferencia), renderizadas en *Mitsuba*. 29

4.8. Esferas (*SDFs*) desplazadas mediante una función seno vertical renderizadas en *Mitsuba*. Comparación del parámetro s de seguridad. . 30
4.9. Árboles generados mediante LSystems. Con 4, 5 y 7 iteraciones, respectivamente. 31
4.10. Tiempo de render de un árbol (en segundos) con cada método en función del número de iteraciones. 32
4.11. Árboles generados mediante Simetrías. Con 1, 4 y 8 planos, respectivamente. 33
4.12. Diagrama de clases simplificado del proyecto. 33
4.13. Paquete de útiles del proyecto. 34
4.14. Esquema del patrón *PImpl* utilizado. 35
4.15. Esquema de la integración con *Mitsuba*. El *SDFWrapper* contiene tanto la *SDF* como un *Marcher* para resolver su intersección. 36

5.1. Comparación de los algoritmos para un prisma redondeado. 38
5.2. Comparación de los algoritmos para la unión de cuatro planos. 40
5.3. Comparación de los algoritmos para la unión de esferas a distintas distancias. 42
5.4. Definición de la unión mediante el producto. 42
5.5. Mezcla de dos materiales limitados por ruido en 3D. 44
5.6. Escena de agua representada mediante un plano y un mapa de normales (ambos *SDFs*). Render y mapa de profundidad, respectivamente. 45
5.7. Montañas generadas con ruido de Perlin (renderizado en 2,4 h). 46
5.8. Rascacielos generados mediante tres prismas replicados un gran número de veces (renderizado en una hora). 47
5.9. Tetraedro de Sierpinski con seis iteraciones, *SDF* de coste lineal con el número de iteraciones gracias a las simetrías. 48

Lista de Tablas

5.1. Promedio de cada métrica para cada algoritmo para el prisma redondeado. 38
5.2. Promedio de cada métrica para cada algoritmo para la unión de cuatro planos. 40
5.3. Promedio de cada métrica para cada algoritmo para la unión de varias esferas a distintas distancias. 41
5.4. Mediana de cada métrica para cada algoritmo en las 17 escenas de prueba elaboradas. 43