Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

Visualización de fluidos generados por el método HSP

Tesis que presenta

Lic. Andrés Cortés Dávalos

para obtener el Grado de

Maestro en Ciencias de la Computación

Director de la Tesis:

Dr. Luis Gerardo de la Fraga

México, Distrito Federal **Noviembre, 2011**

Resumen

En este trabajo de tesis se ha realizado la visualización de la superficie de un fluido. Se utilizó el método de Hidrodinámica Suavizada con Partículas (HSP) para simular el fluido. Se realizaron dos técnicas de visualización, una mediante la técnica de trazo de rayos y la otra usando aplanados de un fluido moviéndose dentro de un contenedor en forma de cubo. En un contenedor convencional para HSP, sus superficies estarán formadas por partículas; en este trabajo las superficies del contenedor están formadas por superficies poligonales.

La técnica de trazo de rayos permite la mejor visualización, pero es altamente costosa y debe realizarse fuera de línea, por lo que se obtiene un video del fluido ya simulado. El uso de los aplanados permite una visualización interactiva, por lo que se construyó una interfaz gráfica en Qt y OpenGL para la visualización en línea del fluido que está siendo simulado. Con aplanados se logró una visualización a 6.7 cuadros por segundo con 500 partículas.

Además se implementó un sensor, en la forma de un *widget*[1], que permite al usuario observar la variación de ciertas cantidades físicas de interés dentro del fluido como pueden ser: presión, densidad y rapidez[2].

[1]Gadget o artilugio.

[2]En física, se conoce como *rapidez* a la magnitud de la *velocidad*, siendo esta última una cantidad vectorial.

Abstract

In this work we have done the visualization of a fluid surface. We have used the Smoothed Particle Hydrodynamics (SPH) method to simulate the fluid. We have implemented two visualization techniques, one by the raytracing technique and the other using splats of a fluid wich is moving inside a cubic container. In a conventional SPH container, its surfaces are built by particles; in this work, the container surfaces are built of polygonal surfaces.

The raytracing technique allows a better visualization, but it is highly expensive and must be done offline, for which we have to obtain a video of the fluid previously simulated. The use of splats allows for an interactive visualization, for which we have built a graphic interface in Qt and OpenGL for the online visualization of the fluid which is being simulated. With splats we achieved a visualization at 6.7 frames per second with 500 particles.

Also, we have implemented a sensor, in the form of a *widget*, that allows the user to watch the variation of some physical quantities of interest inside the fluid body such as: pressure, density and speed[3].

[3]In physics, *speed* is defined as the magnitude of the *velocity*, which is a vectorial quantity.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

La simulación de fluidos es requerida para la visualización de líquidos y objetos deformables en una computadora. Por ejemplo, la visualización interactiva del corte de una vena y la sangre fluyendo a través del corte. La simulación del corte de un objeto plástico parece que se realiza más fácilmente, computacionalmente hablando, al simular el objeto plástico como si fuese un fluido. En esta tesis se intenta resolver el problema de la visualización de la superficie de un líquido que se está simulando con el método de Hidrodinámica Suavizada con Partículas.

1.1. Motivación

En el Departamento de Computación del Cinvestav se realizó la tesis de maestría *Sólido Deformable Inmerso en un Fluido* [4] de Fernando Arreguín. En ella se implementa uno de los métodos de simulación de fluidos más utilizados en el ámbito de la simulación en tiempo real: Hidrodinámica Suavizada con Partículas (HSP). Sin embargo, en dicho trabajo el fluido simulado se visualizó mediante pequeñas esferas, lo cual no es útil para percibir las cualidades y comportamiento del fluido. Se necesita visualizar el volumen de fluido en toda su extensión. Además, la simulación arroja un conjunto de datos asociados al fluido: presión, velocidad y densidad, los cuales necesitan visualizarse.

Surge entonces la necesidad de buscar e implementar métodos de visualización que sean apropiados para ejecutarse en sistemas de moderado poder de procesamiento con el fin de aprovechar la infraestructura existente. Para la visualización, se han abordado separadamente dos requerimientos difíciles de conciliar: interactividad y nivel de realismo. En este trabajo se analiza el porqué no es posible alcanzar ambos requerimientos a la vez usando hardware convencional.

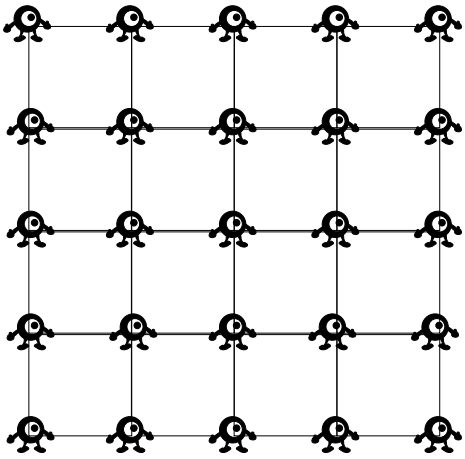


Figura 1.1: Diagrama del espacio de trabajo bidimensional en el enfoque euleriano.

1.2. Simuladores de fluidos

El problema de simular el comportamiento de los fluidos utilizando computadoras corresponde al área de investigación denominada dinámica de fluidos computacional (*Computational Fluid Dynamics, CFD*). Como se afirma en [5], el tema de simulación de fluidos es complicado debido a la interacción que existe entre varios fenómenos tales como convección, difusión, turbulencia o tensión superficial. Existen dos enfoques para la simulación de fluidos, con rejilla *(grid)* y con partículas (*gridless* o *sin rejilla*). A continuación se explican ambos brevemente.

En el primer enfoque se simula el fluido dentro de un espacio de trabajo fijo y usualmente en forma de caja rectangular. Dicho espacio se divide en celdas del mismo tamaño, formando una rejilla (ver figura 1.1). El campo escalar correspondiente a alguna cierta cantidad física de interés, por ejemplo la presión de un fluido, queda descrito por sus valores en las esquinas de las celdas. Estas esquinas actúan como observadores fijos en el espacio que van registrando los valores del campo mientras el fluido va pasando a través de su respectiva posición, por esta razón a este enfoque se le conoce con el nombre de *Euleriano*. Una desventaja de este enfoque es que no es posible realizar la simulación fuera del dominio establecido por la rejilla.

En cambio, en el enfoque por partículas, el campo escalar está definido por un conjunto arbitrario de muestras posiblemente desordenado. Estas muestras corresponden a las partículas que actúan como observadores que llevan consigo los valores del campo en tanto que se van moviendo junto con el fluido (ver figura 1.2). Por este motivo a este enfoque se le conoce como *Lagrangiano*. En este enfoque es posible realizar la simulación en todo el espacio, no se tiene la restricción de un dominio de trabajo como en el caso euleriano.

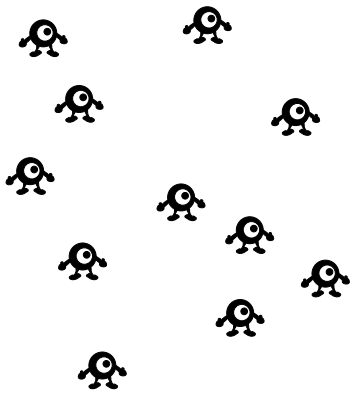


Figura 1.2: Diagrama de partículas para el enfoque lagrangiano en dos dimensiones.

1.3. Estado del arte

Para la realización del presente trabajo de tesis se han consultado diversos artículos relacionados con los temas de simulación y visualización de fluidos. En muchos de los casos, el objetivo de los autores consiste en visualizar una superficie que aproxima el límite exterior del volumen del fluido, lo cual es también uno de nuestros objetivos. En algunos otros, el tema de visualización se menciona de manera secundaria o inclusive se omite totalmente.

A continuación se da un resumen de los diversos trabajos relacionados con nuestro tema y se muestra en la tabla 1.1 el conjunto de técnicas de visualización de fluidos más relevantes que se utilizan en ellos.

En [5], Müller, Charypar y Gross proponen una extensión del método HSP utilizado previamente para animar objetos altamente deformables, además del comportamiento de galaxias, similar al de un fluido. Utilizan las ecuaciones de Navier-Stokes para derivar campos de densidad de fuerza y agregan un término para simular los efectos de la tensión superficial. También sugieren utilizar dos métodos para rastrear y visualizar la superficie libre del fluido: *aplanamiento* y *encajamiento de cubos*, obteniendo despliegues de 15 cps[1] y 5 cps, respectivamente. Según sus autores, el método de *encajamiento de cubos* resulta relativamente lento para una aplicación interactiva, mientras que el de *aplanamiento* da resultados plausibles, pero que pueden ser mejorados. Se sugiere utilizar alguna técnica de sobremuestreo en las partículas, es decir, agregar muestras de partículas de manera artificial para propósitos de despliegue únicamente.

En [10], Müller, Solenthaler, Keiser y Gross proponen una nueva técnica para modelar la interacción entre dos fluidos distintos (o de propiedades distintas) con base

[1] Cuadros por segundo.

Tabla 1.1: Técnicas de visualización de fluidos utilizadas en diversos artículos científicos.

Técnica	**Artículos que la emplean**
Partículas esféricas*	[6], [7], [8]
Aplanamiento*	[5], [9]
Encajamiento de cubos*	[5], [10], [6], [7], [11], [12], [13], [8], [14]
Encajamiento de mosaicos	[15]
Encajamiento de rebanadas	[16]
Sprites	[11]
Elementos de superficie	[9]
Conjuntos de nivel	[17], [18]
Conjuntos de nivel con partículas	[19]
Semi-implícito con partículas móviles	[18]
Trazo de rayos*	[13], [8], [19]
Despliegue volumétrico	[20]
Objetos deformables*	[2]

Se marcan con un $*$ aquellas relevantes para este trabajo.

en el método HSP. Tanto la superficie envolvente de los fluidos como la superficie que forma la interfase[2] entre ellos, se obtienen extrayendo la isosuperficie de un campo escalar con el algoritmo de *encajamiento de cubos*, generando una malla de triángulos que, a su vez, se despliega por medio de trazo de rayos (para esto último ellos utilizaron el programa POVray[3]).

En [6], Müller, Schirm y Teschner proponen un método interactivo para simular sangre en cirugía virtual, como un fluido con superficies libres, basado en HSP. Se presentan dos casos: (a) flujo sanguíneo dentro de una arteria formada por una malla de tetraedros deformables; y (b) sangre que mana de una arteria rota, lo cual implica encontrar la superficie libre que existe en la interfase entre la sangre y el aire. Para el caso (a) la sangre se visualiza como pequeñas esferas cuyo color viene dado por la velocidad de las partículas. En el caso (b) se extrae la superficie libre utilizando el algoritmo de *encajamiento de cubos*. En este artículo el tema principal es obtener el mejor desempeño al simular el comportamiento del fluido de una manera físicamente plausible, puesto que sería utilizado en sistemas de entrenamiento quirúrgico.

En [7], Müller, Schirm, Teschner, Heidelberger y Gross proponen un método para modelar la interacción de fluidos con sólidos deformables. Se basa en el intercambio de

[2]De acuerdo al diccionario de la Real Academia Española, el término *interfase* se refiere a la superficie de separación entre dos fases o partes separables que componen un sistema. Es distinto al término *interfaz* que se refiere a la conexión física y funcional entre dos sistemas independientes, como en el caso de una *interfaz de usuario*.

[3]http://www.povray.org

momento entre las partículas de un simulador de fluidos lagrangiano, es decir, basado en partículas, y objetos sólidos representados mediante mallas de polígonos deformables, utilizando partículas virtuales en la frontera del sólido. Una de las aplicaciones presentadas en este artículo son los simuladores de cirugía, simulando el fluido con superficie libre y visualizándolo ya sea con pequeñas esferas o extrayendo la superficie mediante el algoritmo de *encajamiento de cubos*.

En las notas del curso de simulación de fluidos de SIGGRAPH 2007 [11], Robert Bridson y Mattias Müller dedican solamente cuatro líneas al tema de despliegue del fluido. En el capítulo 9.4, titulado *Rendering*, mencionan que uno de los métodos más simples y rápidos es representar cada partícula con un *sprite*[4], aplicando un filtro de suavizado. Otro método es dibujando una isosuperficie del campo de densidad y triangularla utilizando el algoritmo denominado *encajamiento de cubos*.

En [12], Müller presenta un método para rastrear la superficie libre de un fluido, aplicando dos pasos a una malla de rectángulos. El primer paso es arrastrar los vértices de la malla por efecto del campo de velocidades del fluido. El segundo paso es corregir las partes de la malla que se han intersectado consigo mismas, aplicando las plantillas del método de *encajamiento de cubos*. De manera opcional, es posible aplicar trucos geométricos, aunque no físicos, para prevenir la pérdida de volumen contenido en el interior de la malla. Se afirma que la técnica es eficiente, tanto en cómputo como en consumo de memoria, para producir simulaciones interactivas a 6 cps en una malla de 40 mil triángulos en promedio.

En [16], Rosenberg y Birdwell afirman que en los simuladores de fluidos interactivos, frecuentemente el cuello de botella se encuentra en el paso de extracción de la superficie. Proponen una nueva aproximación al problema de poligonalización de superficies creando una variante del algoritmo de *encajamiento de cubos* denominada *encajamiento de rebanadas* (*marching slices*), en el cual divide el espacio en rebanadas en lugar de cubos, y lo compara con otros algoritmos existentes. También se muestra cómo extender el algoritmo para extraer polígonos en orden de atrás hacia adelante, con el fin de desplegar superficies con transparencia.

En [13] , Lenaerts, Adams y Dutré simulan el paso de un fluido a través de un material poroso y deformable. Se agrega el efecto causado por la ley de Darcy, que gobierna el flujo en un medio poroso, a un simulador HSP. La superficie de los objetos rígidos y deformables se define mediante mallas de triángulos que se animan junto con las partículas del objeto. Las superficies de los fluidos se extraen usando *encajamiento de cubos* y se despliega con trazado de rayos utilizando *POVray*.

[4]El término *sprite* corresponde a la técnica empleada para integrar pequeñas imágenes o animaciones bidimensionales dentro de una escena tridimensional más complicada (http://en.wikipedia.org/wiki/Sprite_(computer_graphics)).

En [8], Clavet, Beaudoin y Poulin presentan un nuevo método basado en partículas para simular fluidos viscosos y elásticos. Como consecuencia de este método surgen efectos de tensión superficial, propiciando la formación de gotas y filamentos. Por otra parte, los efectos de plasticidad elástica y no lineal se obtienen agregando resortes con longitudes de reposo variables, entre partículas. También se simula la interacción entre el fluido y objetos dinámicos. Para efectos de previsualización rápida, las partículas del fluido se despliegan simplemente como esferas hechas de polígonos. Para un despliegue de alta calidad, se utiliza una malla de polígonos, extraída de una isosuperficie de un campo escalar definido por las partículas, mediante el algoritmo de *encajamiento de cubos*. La generación de la imagen se realiza *offline* con un trazador de rayos llamado *Pixie*[5]. Es interesante mencionar, como dato adicional, que los autores afirman que el método es estable para intervalos de tiempo grandes. Se reducen significativamente las inestabilidades numéricas, que abundan en las simulaciones basadas en leyes físicas, y se pueden obtener simulaciones rápidas calculando un solo intervalo de tiempo para cada cuadro de animación. En los resultados se reporta que cada cuadro de animación se genera en 2 segundos para un fluido que consta de 20,000 partículas. Con 1,000 partículas se obtiene una interacción de 10 cps.

En [2], Yongning Zhu y Robert Bridson presentan un método de simulación basada en leyes físicas, para animar el movimiento de la arena. En vez de realizar la simulación de arena compuesta por millares de granos diminutos, se la trata como un contínuo cuyo comportamiento es similar al de un fluido. Básicamente se muestra cómo es posible convertir prácticamente cualquier simulador de fluidos en un simulador de arena, con únicamente unas pocas adiciones para tomar en cuenta la fricción, tanto entre granos como entre grano y frontera. Además, proponen un método alternativo de simulación de fluidos utilizando una nube de partículas que permite simular el arrastre y rastreo de la superficie, pero también se utiliza una rejilla auxiliar para reforzar las condiciones de frontera y la incompresibilidad. Para un despliegue de alta calidad es posible realizar la reconstrucción de la superficie que envuelve las partículas utilizando objetos sin forma (*blobbies*). Sin embargo, con esta técnica es imposible lograr superficies regulares perfectamente planas o esféricas en situaciones donde se espera que un fluido tome estas formas. Para mejorar esta situación, los autores proponen una manera diferente de construir la superficie implícita que envuelve las partículas, aunque se obtienen gotas espurias en regiones cóncavas de tamaño pequeño. Proponen también una manera de detectarlas y eliminarlas, sin alterar el resto de la superficie. Reportan tiempos de 40 a 50 segundos para generar cada cuadro de animación, con 100^3 partículas en una rejilla de 250^3 celdas. Para mejorar los tiempos, sugieren utilizar una aproximación de *mínimos cuadrados móviles* (*MLS, Moving Least Squares*)[21] para obtener tiempos interactivos.

En [14], Wojtan, Thürey, Gross y Turk presentan un método para rastrear de manera precisa el movimiento de la superficie de materiales deformables que efectúe

[5] http://www.renderpixie.com/

cambios topológicos, es decir, que varios objetos se fundan en uno, que uno se separe en varios o que aparezcan agujeros en el fluido. Utilizan un método lagrangiano de rastreo de superficie y se representa a la misma mediante una malla de triángulos. Los cambios de topología se manejan editando dicha malla. Cuando es necesario crear nueva geometría[6], se extrae de la isosuperficie utilizando el algoritmo de *encajamiento de cubos*, el cual se considera en este artículo como el estándar para tales situaciones. Cabe mencionar que en este artículo, la simulación del fluido se realiza con un método euleriano, es decir, no mediante partículas. Es la visualización la que lleva un componente lagrangiano.

En [22], Cleary, Pyo, Prakash y Koo presentan un método basado en partículas para representar fluidos con burbujas y espuma de apariencia realista. No se menciona el método utilizado para extraer la superficie del fluido, pero el despliegue lo efectúa utilizando el software comercial *Maya*, generando los distintos componentes (fluido, espuma y burbujas) en capas distintas y luego los compone usado el software comercial *After Effects*.

En su tesis de maestría [15], Williams Brent propone otra variante del algoritmo de *encajamiento de cubos*, denominada *encajamiento de mosaicos* (*marching tiles*), en la cual utiliza una teselación del volumen con una figura geométrica basada en tetraedros, con el fin de reducir el *aliasing* volumétrico (*voxelización*). Su algoritmo fue utilizado en las producciones cinematográficas *Hellboy II* y *10,000 A.C.* No se ejecuta en tiempo real.

En [9], Keiner, Adams, Gasser, Bazzi, Dutré y Gross presentan un marco de trabajo unificado para la animación, basada en leyes físicas, de sólidos deformables y fluidos. Fusionando las ecuaciones de mecánica de sólidos con las de Navier-Stokes, usando un enfoque lagrangiano basado en partículas, consiguen animar tanto fluidos como sólidos deformables, así como las transiciones de fase entre ellos. Además, proponen un enfoque de generación de superficie híbrido entre una representación implícita y una explícita, que es capaz de soportar cambios topológicos, al mismo tiempo que preserva el nivel de detalle a escala fina. En la parte de la representación implícita de la malla utilizan muestras puntuales orientadas (*surfels* o elementos de superficie), mientras que para la representación explícita mantienen una aproximación en isosuperficies de campos escalares. Con ambas representaciones, se visualiza la superficie tanto de objetos sólidos deformables con detalles finos, como de fluidos en los cuales se pierde el detalle debido a la tensión superficial. En los ejemplos mostrados, la simulación se ejecuta en tiempo interactivo de 0.7 cps, con entre 2,400 y 3,900 partículas. La visualización de la superficie obtenida se realiza, durante una ejecución interactiva, con el método de *aplanamiento* sobre GPUs. Las imágenes en alta resolución y los videos son generados utilizando una versión modificada del trazador

[6]En este contexto, el término *geometría* se refiere a la estructura espacial de los componentes de una malla de polígonos: vértices, aristas y caras.

de rayos POVray, para trazar superficies compuestas de muestras puntuales.

En [23], Yuksel, House y Keyser proponen un nuevo método para simular, en tiempo real, ondas en la superficie de un fluido y su interacción con objetos flotantes. La superficie del fluido consiste en un mapa de altitudes y las ondas en su superficie se simulan con un enfoque orientado a partículas. Utilizan el hardware gráfico para transformar estas *ondas-partículas* en un mapa de altitud, el cual puede desplegarse con la tarjeta gráfica, sin necesidad de extraer la superficie. Dado que únicamente se están simulando las ondas superficiales con mapas de altitud, no se puede modelar el comportamiento del fluido en el volumen contenido en el interior de la superficie.

En [24], Türey, Müller, Schirm y Gross presentan un nuevo método que mejora las simulaciones de fluidos por mapas de altitud, añadiendo de manera automatizada, la geometría de las olas que rompen, imposibles de representar con únicamente los mapas de altitud (también conocidos como *Shallow Water Simulations*). Debido a la naturaleza bidimensional de este tipo de simulaciones, son muy eficientes para simular la superficie de un fluido en tiempo real, pero no son de utilidad cuando se desea simular también lo que ocurre en el interior del fluido.

En [20], Cohen, Tariq y Green describen un sistema interactivo para crear animaciones basadas en simulación de fluidos, que reaccionan al movimiento de otros objetos. Consta de un simulador euleriano diseñado para ejecutarse eficientemente en arquitecturas paralelas y cuyo dominio de simulación se traslada dinámicamente a la posición del objeto controlado por el usuario. El movimiento del fluido se visualiza mediante el efecto ejercido por su campo de velocidades sobre partículas que no están restringidas a permanecer dentro del dominio del simulador. Con ello se elimina el efecto de *fluido confinado dentro de una caja* (*fluid-in-a-box*), inherente a los simuladores eulerianos. Además se aplica una técnica de despliegue volumétrico a las partículas para producir un efecto visual de humo, polvo o niebla.

En [17], Foster y Fedkiw presentan un método general para modelar y animar líquidos, enfocándose en el volumen del fluido simulado. Para ello, mezclan el paradigma de partículas con el de evolución de una superfcie implícita, originando una modificación del método conocido como *conjuntos de nivel* (*level sets*). Se menciona que estos métodos tienen, por separado, fortalezas y debilidades complementarias, pero la manera en que las fusiona permite evitar el efecto de pérdida de volumen inherente a los conjuntos de nivel, mientras se mantiene una superficie suave, imposible de lograr con partículas únicamente. La mayoría de los simuladores de fluidos se limita únicamente a reproducir el efecto de las leyes físicas, sin proporcionar mecanismos de control por parte del usuario (posiblemente por considerarlos como interacciones no físicas o no realistas). Los autores otorgan importancia a la necesidad que se tiene de dichos mecanismos de control en el área de animación. Un ejemplo de la capacidad de control que brinda este enfoque puede apreciarse en la escena de apertura de la

película *Shrek*, cuando el personaje animado interactúa con un fluido viscoso, llena su boca con éste para luego expulsarlo, mediante una curva de control.

En [19], Enright, Marschner y Fedkiw presentan un nuevo método para animación y despliegue de efectos de agua fotorrealistas. Para obtener un comportamiento físicamente plausible de una superficie de agua, proponen un nuevo enfoque de rastreo de superficie *engrosada* (*thickened*), llamado *método de conjuntos de nivel con partículas* (*particle level set method*). Básicamente, se genera una superficie implícita que se va arrastrando por efecto del campo de velocidades. Junto con la superficie se generan partículas, tanto en el interior como en el exterior del volumen de fluido, hasta cierta distancia, que se utilizan como marcadores para corregir errores en la evolución de la superficie. Las partículas internas ayudan a prevenir pérdida de volumen de fluido, mientras las externas simulan efectos de bolsa de aire atrapado por el agua. Para desplegar la superficie obtenida, utilizaron un trazador de rayos de tipo Monte Carlo, con capacidad para generar todos los tipos de iluminación mediante mapas de fotones y cálculo de irradiancia.

En [18], Premoze, Tasdizen, Bigler, Lefohn y Whitaker utilizan un enfoque orientado a partículas para dar al usuario una previsualización rápida, en baja resolución, de la animación que se está generando. Posteriormente, se aumenta la resolución en el número de partículas para producir el movimiento final del fluido. El método utilizado es llamado *semi-implícito con partículas móviles* (*MPS, Moving Particle Semi-Implicit*) y consiste en realizar el paso de animación de manera similar al método HSP, pero eliminando la ligera compresibilidad[7] mediante la solución de una ecuación de Poisson, usando el método de gradiente conjugado. En el estado inicial de las partículas, se construye una superficie implícita que las envuelve y que posteriormente se va evolucionando en cada paso de tiempo sobre una rejilla discreta mediante el método de conjuntos de nivel.

En [25], Stam propone un modelo de simulación de fluidos incondicionalmente estable, que permite tomar pasos de tiempo de mayor longitud, logrando así animaciones más rápidas. El autor resalta que, en aplicaciones de ingeniería, se requiere que la simulación arroje resultados acotados de manera precisa para las cantidades físicas involucradas, siendo de importancia secundaria la forma y apariencia visual del fluido. Por otra parte, en el área de graficación y animación, la forma y comportamiento del fluido son de interés primario, sacrificando en cierta medida la exactitud de la simulación, como es el caso de las aplicaciones de entretenimiento y algunos simuladores. El método propuesto por Stam ofrece estabilidad incondicional, pero el precio se paga en exactitud y no resulta apto para ciertas aplicaciones de ingeniería, ya que sufre de *disipación numérica*. Se ha utilizado con éxito para simular de manera interactiva, diversos efectos gaseosos, pero no es posible simular fluidos con superficie libre, como en el caso del agua. Sólo resta mencionar que este método está basado en

[7] HSP simula fluidos compresibles o ligeramente compresibles.

rejillas, no en partículas.

En el ámbito de las aplicaciones de animación comerciales podemos mencionar el caso de *Maya*. Se trata de una suite de diseño, modelado y animación tridimensional que ha estado en la lista de herramientas preferidas por los animadores profesionales durante años. Otros ejemplos incluyen *Lightwave* y *3D Studio Max*. Como se menciona en el capítulo final del curso de física en tiempo real del SIGGRAPH 2008 [26], Jos Stam desarrolló para la versión 9 de Maya un *simulador unificado* que simula en la misma estructura de datos (basada en simplejos) la interacción entre partículas (0-dimensionales), cabello y cadenas de aristas (1-dimensionales), ropa (2-dimensional), fluidos, cuerpos blandos y sólidos (3-dimensionales). Actualmente este simulador se conoce como **nParticles** y se utiliza el método HSP para resolver la simulación de fluidos. Cabe mencionar que Maya también cuenta con un simulador *euleriano* que realiza la simulación sobre una región acotada del espacio y sin utilizar partículas pero su visualización resulta *voxelizada* para dominios de baja resolución.

También existen herramientas de código abierto que proporcionan prácticamente las mismas prestaciones que el software propietario. La más completa y de creciente aceptación actualmente es *Blender*, la cual cuenta desde su versión 2.37[8] con un simulador que utiliza el algoritmo *LBM* (*Lattice Boltzmann Method*), el cual se encuentra a medio camino entre los métodos con rejilla y los de partículas. Es interesante mencionar que el creador y administrador de este módulo de fluidos en *Blender* es Nils Thürey, autor en [14], [24], [26], [27], entre otros. El módulo está basado en su tesis doctoral y fué implementado en el año 2005.

Ahora bien, desde la versión 2.57, *Blender* incorpora HSP[9] para la simulación de fluidos entre los sistemas de partículas disponibles. Sin embargo, como afirma Thürey en su wiki, *Blender* no cuenta con visualización volumétrica. De manera que el fluido simulado por el método HSP, se visualiza en forma de partículas. En una publicación del blog de desarrolladores de Blender[10], del 18 de Febrero de 2011, se afirma que actualmente se encuentra en desarrollo la implementación de un módulo para la extracción de superficie del fluido. Uno de los objetivos de este trabajo de tesis es implementar un módulo de software con esta misma funcionalidad.

1.4. Planteamiento del problema

En esta tesis se resolverá el problema de visualización de fluidos simulados por el método HSP. Se proponen dos soluciones, como se muestra en la figura 1.3:

[8] http://wiki.blender.org/index.php/Doc:Manual/Physics/Fluids y http://wiki.blender.org/index.php/User:N_t/SummerOfCode2005/Fluid_Animation/Development

[9] http://wiki.blender.org/index.php/Doc:2.5/Manual/Physics/Particles/Physics/Fluid y http://blenderdiplom.com/index.php/en/tutorials/item/30-tutorial-introduction-to-fluid-particles

[10] http://code.blender.org/index.php/2011/02/particle-fluids-refactoring-under-way/

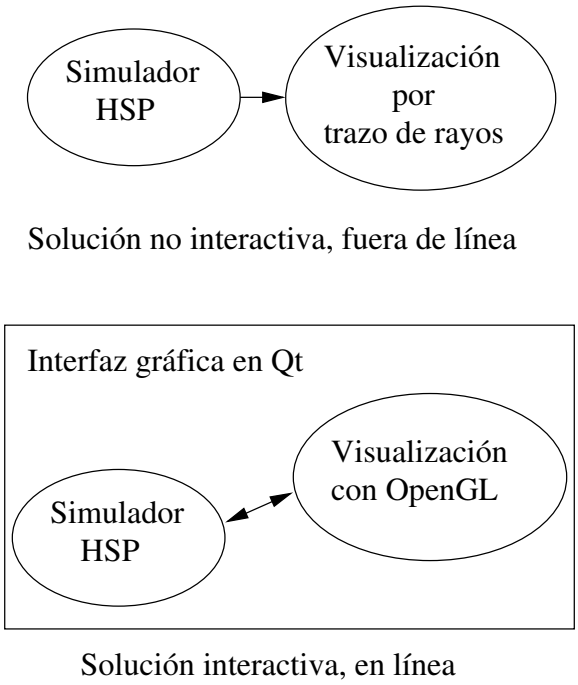


Figura 1.3: Esquema de soluciones.

1. El simulador HSP generará archivos de salida que serán la entrada para un visualizador por trazo de rayos. Se usará el programa POVray[11] para esta última tarea. Esta solución es, por supuesto, fuera de línea y no interactiva: todo el proceso tiene que volver a realizarse si se cambian los parámetros de la simulación y la visualización tiene que volver a realizarse si se observa la escena desde otra posición diferente.

2. Otra solución será construir una interfaz gráfica en *Qt*[12] en la que estará inmerso el simulador HSP y al mismo tiempo de la simulación podrá visualizarse usando *OpenGL*[13]. Se podrá cambiar el punto de vista de la escena usando el ratón y podrá inicializarse otra simulación en cualquier momento.

Dadas las limitaciones en OpenGL para realizar efectos avanzados de visualización (como las cáusticas [28]), se tendrá la mejor visualización con la propuesta no interactiva; sin embargo, una visualización rápida general del comportamiento del fluido simulado podría conseguirse con la segunda solución interactiva.

[11] http://www.povray.org
[12] http://qt.nokia.com/products/
[13] http://www.opengl.org

1.5. Organización de la tesis

En el capítulo 2 se profundiza teóricamente en el funcionamiento del método HSP. También se detallan las técnicas empleadas para la etapa de visualización.

En el capítulo 3 se explica tanto el diseño del contenedor para realizar las simulaciones de esta tesis, como también las modificaciones realizadas al simulador HSP para acelerar su tiempo de ejecución. Además, se muestran los resultados obtenidos con esta realización.

En el capítulo 4 se explican los detalles de implementación de las técnicas de visualización empleadas y se mencionan las ventajas y desventajas de cada una de ellas. También se describe la arquitectura de la solución presentada en este trabajo, la cual consiste en dos módulos: uno para simulación y el otro para visualización. Al final del capítulo se presentan los resultados obtenidos con el módulo de visualización.

En el capítulo 5 se dan las conclusiones y observaciones obtenidas durante la elaboración de la tesis. Finalmente, se mencionan algunas ideas para el trabajo a futuro.

Capítulo 2

Teoría

En este capítulo se presentan brevemente los conceptos teóricos necesarios para la realización de esta tesis. La teoría utilizada para la simulación del fluido se presenta en la sección 2.1. Una descripción de las técnicas empleadas en la visualización se da en la sección 2.2.

2.1. Hidrodinámica suavizada con partículas

La etapa de simulación se realiza con el método de hidrodinámica suavizada con partículas. Se trata de un método de interpolación en el cual, a partir de un conjunto finito de partículas, se interpolan las cantidades físicas en cualquier posición del espacio.

Como se explica en [29], la interpolación se basa en la teoría de interpolantes integrales usando algún *núcleo*, *ventana* o *kernel* de interpolación, el cual aproxima a una función delta de Dirac. Una cierta cantidad A se interpola en la posición $\mathbf{r}$ mediante la expresión

$$A(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, h) \tag{2.1}$$

donde m_j es la masa de la partícula j, ρ_j es su densidad, $\mathbf{r}_j$ es su posición y W es la ventana de suavizado de radio h. La ventana W es una función par pues cumple $W(\mathbf{r}, h) = W(-\mathbf{r}, h)$ y está normalizado, esto es, $\int W(\mathbf{r}) d\mathbf{r} = 1$.

Los interpolantes son funciones analíticas que se pueden derivar sin la necesidad de utilizar una rejilla. De este modo es posible calcular el gradiente de la cantidad ∇A, así como su laplaciano $\nabla^2 A$, con las siguientes expresiones.

$$\nabla A(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} \nabla W(\mathbf{r} - \mathbf{r}_j, h) \tag{2.2}$$

$$\nabla^2 A(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} \nabla^2 W(\mathbf{r} - \mathbf{r}_j, h) \tag{2.3}$$

Ha sido empleado con éxito en simulación de problemas en astrofísica, como por ejemplo colisiones entre galaxias y dinámica de formación estelar pero es lo suficientemente general para ser usado en cualquier clase de simulación de fluidos [5].

2.1.1. Modelado de fluidos

En este caso el fluido se modela mediante un conjunto finito de partículas que se mueven bajo la influencia de ciertas fuerzas, por ejemplo la gravitacional y las hidrodinámicas. Esto se logra mediante la aplicación de las ecuaciones de Navier-Stokes, que se dan a continuación.

La ecuación de conservación del momento:

$$\rho\left(\frac{\delta \mathbf{v}}{\delta t} + \mathbf{v}\cdot\nabla\mathbf{v}\right) = -\nabla p + \rho\mathbf{g} + \mu\nabla^2\mathbf{v} \tag{2.4}$$

donde ρ es la densidad del fuido, p es la presión, $\mathbf{v}$ es la velocidad, $\mathbf{g}$ es la aceleración gravitacional y μ es un factor de viscosidad.

La ecuación de conservación de la masa:

$$\frac{\delta \rho}{\delta t} + \nabla\cdot(\rho\mathbf{v}) = 0 \tag{2.5}$$

La conservación de la energía se obtiene a través de la ecuación de estado de un gas ideal:

$$\rho = kp \tag{2.6}$$

donde la constante k depende de la temperatura. En [8] se utiliza una variante de la ecuación de estado a la que llama *pseudo-presión* y a la constante k se le llama parámetro de rigidez *(stiffness)*.

Estas ecuaciones se utilizan en todas las simulaciones de fluidos, tanto eulerianas como lagrangianas. Sin embargo, como se explica en [5], debido a que en el enfoque lagrangiano se puede utilizar un número fijo de partículas y cada una de ellas posee una masa constante, podemos omitir completamente la ecuación de conservación de masa (2.5) y así reducir la cantidad de cálculos. Dicho de otra manera, la masa se conserva de manera automática y no es necesario aplicar la ecuación.

También, debido a que las partículas se mueven junto con el fluido, se tiene que el término convectivo $\mathbf{v}\cdot\nabla\mathbf{v}$ de la ecuación de conservación del momento (2.4) es cero, lo cual simplifica la ecuación:

$$\rho\left(\frac{\delta \mathbf{v}}{\delta t}\right) = -\nabla p + \rho\mathbf{g} + \mathbf{f}_v \tag{2.7}$$

En la ecuación (2.7) se puede apreciar que las partículas se mueven bajo el efecto de tres tipos de fuerza volumétrica[1]: la fuerza interna debida al gradiente de presión $-\nabla p$, la fuerza debida a la gravedad $\mathbf{g}$ y la fuerza debida a la viscosidad del fluido $\mathbf{f}_v$, que en la ecuación (2.4) queda descrita por la expresión $\mu\nabla^2\mathbf{v}$.

2.1.2. Viscosidad

En [29], Monaghan calcula la fuerza de viscosidad $\mathbf{f}_v$ ejercida sobre la partícula i mediante una expresión denominada *viscosidad artificial*.

$$\mathbf{f}_v = -\sum_j m_j \rho_j \Pi_{ij} \nabla W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{2.8}$$

donde Π_{ij} es de la forma:

$$\Pi_{ij} = \begin{cases} \frac{-\alpha \bar{c}_{ij}\mu_{ij} + \beta\mu_{ij}^2}{\bar{\rho}_{ij}}; & \text{si} \quad \mathbf{v}_{ij}\cdot\mathbf{r}_{ij} < 0 \\ 0; & \text{si} \quad \mathbf{v}_{ij}\cdot\mathbf{r}_{ij} \geq 0 \end{cases}$$

con

$$\mu_{ij} = \frac{h\mathbf{v}_{ij}\cdot\mathbf{r}_{ij}}{\mathbf{r}_{ij}^2 + \eta^2}$$

donde Monaghan utiliza, para cada cantidad A, la notación $A_{ij} = A_i - A_j$
y $\overline{A}_{ij} = (A_i + A_j)/2$.
Las constantes α y β se eligen de acuerdo a la simulación particular; en su trabajo se utilizan valores de 0.01 y 0.00, respectivamente. La constante c es la velocidad del sonido. Esta es la aproximación implementada en [4].

Sin embargo, para este trabajo, hemos puesto a prueba la aproximación de Müller [5], que resulta más sencilla y es consistente con el uso de las ventanas de suavizado. La fuerza de viscosidad ejercida en la partícula i es:

$$\mathbf{f}_v = \mu\nabla^2\mathbf{v}(\mathbf{r}_i) = \mu\sum_j m_j \frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_j}\nabla^2 W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{2.9}$$

Es posible agregar al modelo otras fuerzas asociadas a fenónemos tales como la tensión superficial[2]. Sin embargo, en el presente trabajo se ha decidido no implementarlas para reducir los cálculos y debido también a que no se requieren para los objetivos de esta tesis, que está orientada a la etapa de visualización.

[1] En la formulación presentada, los términos que aparecen del lado derecho de la ecuación (2.7) corresponden a fuerzas aplicadas por unidad de volumen, por ello les llamamos *volumétricas*. En algunos trabajos se les denomina *densidades de fuerza*. En adelante nos referiremos a ellas simplemente como *fuerzas*, quedando claro en el contexto si se trata de fuerzas totales o por unidad de volumen.

[2] Para un tratamiento de la tensión superficial ver [5].

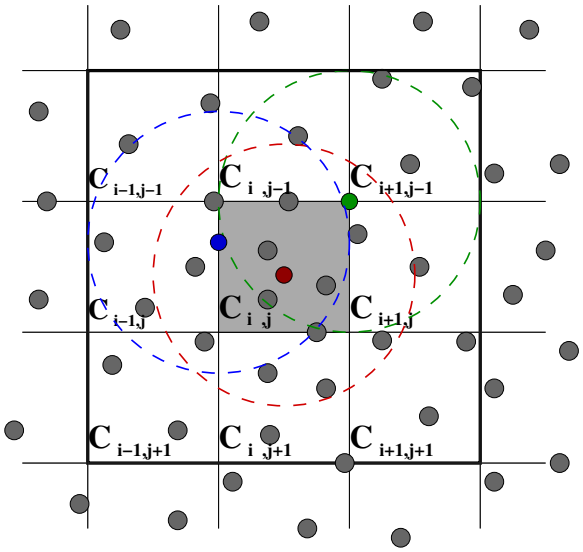


Figura 2.1: Partículas dentro de la celda C_{ij} y sus celdas contiguas en dos dimensiones. Se muestra el radio de soporte h de la ventana W para tres partículas.

Premoze [18] afirma que HSP es un método flexible pero únicamente funciona para fluidos compresibles. Esto representa un problema debido a que algunos de los fluidos que son de interés práctico, por ejemplo el agua, son incompresibles. Sin embargo, se han hecho varias propuestas que permiten utilizar el método para que trabaje con fluidos ligeramente compresibles. En este trabajo no se ha implementado alguna de dichas modificaciones y se le permite al fluido que se comprima en cierto grado, en tanto no resulta notorio en la etapa de visualización.

2.1.3. Búsqueda eficiente de vecinos

El cuello de botella durante la simulación con HSP es la búsqueda de vecinos que contribuyen para cada partícula. La manera ingenua de realizar esta búsqueda es mediante fuerza bruta, verificando cada uno de los $n-1$ posibles vecinos para cada una de las n partículas. Este tipo de búsqueda tiene una complejidad de orden $O(n^2)$. Dado que la ventana de suavizado tiene un soporte[3] finito h, la manera común para hacer más eficiente la búsqueda es dividir el espacio de trabajo en una rejilla de celdas de tamaño h, mediante alguna estructura de datos apropiada. De esta manera, los vecinos que pueden interactuar con una partícula deben estar en su misma celda o en las celdas contiguas, solamente. Este hecho se muestra en la figura 2.1. Según Müller [5], la complejidad de esta búsqueda de vecinos se reduce a $O(mn)$, donde m es el número promedio de partículas dentro de una celda. Esta idea también se utiliza en [4].

[3]El soporte de la ventana es el radio de su área de influencia.

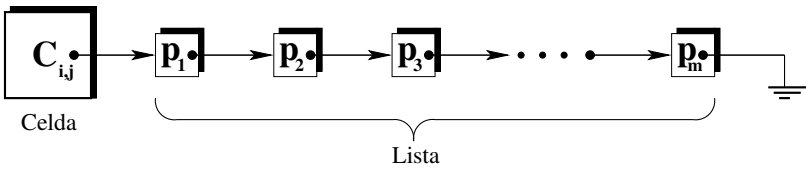


Figura 2.2: Esquema de la estructura de datos implementada: un arreglo de celdas con una lista ligada en cada una.

En este trabajo se usó un arreglo de tamaño fijo para representar las celdas, tanto en dos como en tres dimensiones, debido a que el espacio de trabajo es un cubo unitario que no cambia. Así mismo, para llevar el control de las partículas que se encuentran dentro de cada celda, se implementó en cada una de ellas una lista ligada. El esquema de esta estructura de datos se muestra en la figura 2.2.

Es interesante mencionar lo que se puede lograr utilizando una estructura de datos distinta, por ejemplo, en [8] se simula el fluido en un espacio de trabajo arbitrario, así que los autores utilizan una tabla *hash* para almacenar las celdas que contienen partículas, permitiéndoles realizar la animación en un espacio de trabajo virtualmente infinito. En nuestro caso, el espacio de trabajo es fijo y el número de celdas que utilizamos no es muy elevado, son entre $11^3 = 1331$ y $21^3 = 9261$ celdas, por lo cual resulta más eficiente y simple utilizar un arreglo no dinámico.

2.1.4. Modelado de la superficie del fluido

Cuando se trata de visualizar un fluido constituído por partículas, la primera solución que surge a la mente es utilizar objetos sin forma[4] para modelar una superficie suave que envuelva las partículas. Por otra parte, en [5] Müller propone una manera de modelar la superficie del fluido que va en consonancia con el método HSP, ya que se emplea la misma ventana de suavizado. Se utiliza un campo escalar tal que tiene un valor de uno en la posición de las partículas y tiende a cero al alejarse de ellas. A este campo se le llama *campo de color* (*color field*) en la literatura. El campo de color suavizado se define como:

$$c(\mathbf{r}) = \sum_j m_j \frac{1}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, h) \tag{2.10}$$

De esta manera, la superficie del fluido se define como una isosuperficie[5] del campo de color, es decir, el conjunto de puntos tales que el campo de color tiene cierto un valor de umbral. También es importante mencionar que para cada punto de la superficie, se puede calcular el vector ∇c el cual es perpendicular a la misma y se utiliza en los algoritmos de extracción de superficie de las secciones 4.3 y 4.4.

[4] La técnica de *objetos sin forma* se explica en la sección 2.2.2.

[5] El concepto de isosuperficie de un campo escalar se explica en la sección 2.2.2.

2.2. Métodos de visualización de fluidos

Para visualizar fluidos se han utilizado diversos métodos. Aquellos que hemos identificado como los más utilizados son: *trazo de rayos* (*raytracing*), *objetos sin forma* (*blobby objects*), *aplanamiento* (*splatting*) y *encajamiento de cubo* (*marching cubes*). A continuación se describe brevemente cada uno de ellos.

2.2.1. Trazo de rayos

En el curso del SIGGRAPH de 2007 [30] sobre despliegue de alta calidad se da el algoritmo de trazo de rayos. El algoritmo básico es muy simple y elegante. Es capaz de calcular fenómenos visuales muy difíciles de obtener por cualquier otro método. Por este motivo es la técnica más empleada en animación. Su principal ventaja es que se pueden obtener resultados de la mayor calidad, pero puede ser muy costosa en términos de tiempo y consumo de memoria. El despliegue o *renderizado* consiste en calcular el color de cada punto de una imagen. Esto se consigue al proyectar los puntos desde una escena tridimensional hacia una imagen bidimensional. Básicamente se realizan dos operaciones en cada *pixel*[6]: (a) encontrar el punto de colisión de un rayo con la superficie más cercana al observador y (b) decidir el color que debe llevar el pixel.

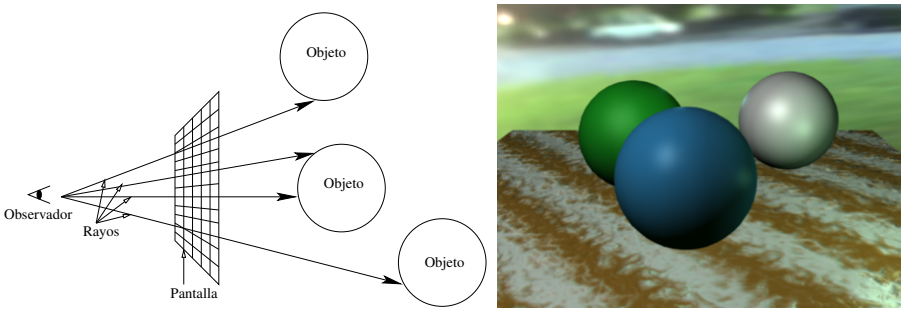


Figura 2.3: Diagrama de rayos lanzados a una escena desde el observador y ejemplo de renderizado en Blender (recursión desactivada).

La técnica de trazo de rayos consiste en simular que se dispara un rayo de luz por cada pixel de la pantalla. Se verifica si la línea del rayo se interseca con algún objeto de la escena que se desea visualizar. De ser así se calcula el ángulo de incidencia y las condiciones de iluminación de dicho punto, así como el material de que está hecho el objeto, y se determina el color que debe lucir el pixel correspondiente. El proceso se ilustra en la figura 2.3 izquierda.

[6]Contracción del término *picture element* (elemento de imagen).

Caso simple

En el caso más sencillo, el algoritmo de trazo de rayos no toma en cuenta reflejos o transparencia y se le conoce como *lanzamiento de rayos* (*raycasting*), dejando el término *trazo de rayos* (*raytracing*) para el caso recursivo, el cual se explicará más adelante. En la figura 2.4 se da el pseudocódigo para lanzamiento de rayos.

```
Procedimiento trazoRayosSimple
inicio
    para cada pixel haz
        calcula el rayo correspondiente al pixel
        para cada objeto en la escena haz
            si el rayo intersecta el objeto y es el más cercano hasta ahora entonces
                almacena la distancia de intersección y el color del objeto
            finsi
        finpara
        si el rayo intersectó algún objeto entonces
            asigna al pixel el color del objeto más cercano
        en otro caso
            asigna al pixel el color del fondo
        finsi
    finpara
fin
```

Figura 2.4: Pseudocódigo del algoritmo simple de lanzamiento de rayos.

Caso recursivo

La técnica recursiva de trazo de rayos es similar a la de lanzamiento de rayos, aquí también se dispara un rayo de luz por cada pixel de la pantalla y se verifica si se

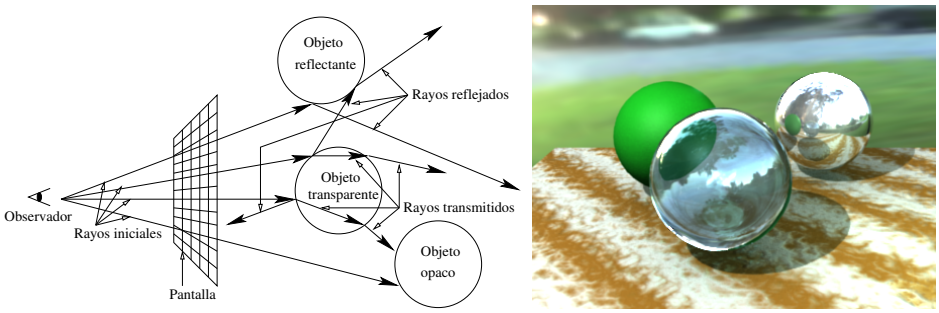


Figura 2.5: Diagrama de trazo de rayos en una escena desde el observador y ejemplo de renderizado en Blender (recursión activada).

intersecta con algún objeto y se determina el color para ese pixel. La diferencia radica en que, según el material de que se trate, se calculan de manera recursiva un nuevo rayo de luz reflejado y uno transmitido. Al seguir estos rayos se generan los efectos de reflejo y transparencia en la escena, obteniendo imágenes fotorrealistas, es decir, similares a las obtenidas con una cámara fotográfica. La cantidad de rayos necesarios para generar cada imagen depende de la escena particular, y se suele requerir una enorme cantidad de cálculos, lo que lleva mucho tiempo de procesamiento y no resulta factible para renderizar las escenas de manera interactiva. En la figura 2.5 a la izquierda se esquematiza este proceso recursivo, y en la figura 2.6 se muestra el pseudocódigo dado en [31] para este algoritmo.

```
Procedimiento trazoRayosRecursivo
inicio
    para cada pixel haz
        calcula el rayo correspondiente al pixel
        color_pixel = trazaRayo(rayo, 1)
    finpara
fin

Función trazaRayo
ENTRADA: un rayo y un valor entero que indica la profundidad de recursión
SALIDA: el color calculado para el pixel
inicio
    para cada objeto en la escena haz
        si el rayo intersecta el objeto y es el más cercano hasta ahora entonces
            almacena la distancia de intersección y el color del objeto
        finsi
    finpara
    si el rayo intersectó algún objeto entonces
        devuelve sombra_rec(objeto, rayo, intersección, normal, profundidad)
    en otro caso
        devuelve el color del fondo
    finsi
fin
```

Figura 2.6: Pseudocódigo del algoritmo recursivo de trazo de rayos. Utiliza la función descrita en la figura 2.7.

Existe una gran variedad de programas, tanto comerciales como gratuitos y de código abierto, que efectúan trazado de rayos de gran calidad. Entre ellos podemos

```
Función sombra_rec
ENTRADA: un objeto, un rayo, el punto de intersección entre ellos,
la normal en ese punto y la profundidad de recursión
SALIDA: el color calculado para el pixel
inicio
    color ← color_ambiente
    para cada luz en la escena haz
        rayo_s ← rayo desde intersección a la luz
        si normal·rayo_s es positivo entonces
            calcular cuánta luz es bloqueada por superficies opacas
            y transparentes, y usarla para escalar las componentes difusas
            y especulares antes de añadirlas al color
        finsi
    finpara
    si profundidad < profundidad máxima entonces
        si el objeto es reflejante entonces
            rayo_r ← rayo en la dirección de reflexión desde intersección
            color_r ← trazaRayo(rayo_r, profundidad + 1)
            escalar color_r por el coeficiente especular y añadir al color
        finsi
        si el objeto es transparente entonces
            rayo_t ← rayo en la dirección de refracción desde intersección
            si no ocurre reflexión total interna entonces
                color_t ← trazaRayo(rayo_t, profundidad + 1)
                escalar color_t por el coeficiente de transmisión y añadir al color
            finsi
        finsi
    finsi
    devuelve color
fin
```

Figura 2.7: Pseudocódigo de la función para el cálculo de sombras, reflejos y transparencias.

mencionar aplicaciones que ofrecen esta funcionalidad como *Blender*[7], *Maya*[8] y *Lightwave*[9], o bien, programas que realizan específicamente el trazo de rayos como *POVray* (*Persistence of Vision Raytracer*)[10] y *YafaRay*[11].

Entre la comunidad científica dedicada a los gráficos por computadora existe mayor familiaridad con el programa POVray, mientras que es mas bien la comunidad de diseñadores y artistas quienes están familiarizados con los otros programas que hemos mencionado. Debido a dicha familiaridad hemos elegido utilizar POVray para la realización de este trabajo. Mediante la técnica de trazo de rayos se obtienen resultados de gran calidad pero se requiere una enorme cantidad de cálculos y esto significa mucho tiempo para el renderizado. Es por ello que existe una extensa y activa investigación para crear implementaciones de trazo de rayos que funcionen en tiempo real. Como ejemplo citamos el trabajo [32] que acelera el trazo de rayos en GPU utilizando aritmética afín y de intervalos. También está el trabajo [33] que efectúa el trazo de rayos de superficies implícitas formadas por funciones C^1-contínuas acotadas. El autor afirma que su método es rápido, robusto y numéricamente estable debido a que utiliza únicamente métodos analíticos. No corresponde al presente trabajo la implementación del trazador de rayos, sino que utilizamos POVray, disponible de manera gratuita y de amplia aceptación.

2.2.2. Objetos sin forma

En el área de graficación por computadora se han desarrollado varias técnicas para modelar objetos deformables. Entre estos objetos podemos mencionar la forma de los músculos de los animales y de las estructuras moleculares, de los fluidos como el agua u objetos que se funden como el acero y la lava. Una de las técnicas más populares para modelar este tipo de objetos se denomina *objetos sin forma*[12], en la cual se define la forma del objeto como una isosuperficie de un campo escalar.

Isosuperficie de un campo escalar

Un campo escalar (en este contexto también se le ha llamado campo de potencial o función de densidad) es una función implícita $\phi : \mathbb{R}^n \rightarrow \mathbb{R}$.

[7] http://www.blender.org
[8] http://usa.autodesk.com/maya/
[9] http://www.newtek.com/lightwave/
[10] http://www.povray.org
[11] http://www.yafaray.org
[12] Los *blobby objects* o *metaballs* son traducidos como objetos sin forma en [34], son llamados *objetos suaves* (*soft objects*) en [35] y *gotitas* (*blobbies*) en [2]. Por otra parte, *blob* es la voz inglesa para gota, que viene del latín *stilla* (del cual procede *destilar*), de manera que podemos utilizar el término *estiliforme*, que indica algo con forma de gota. En este trabajo utilizaremos *blob*, que es como se le llama en POVray.

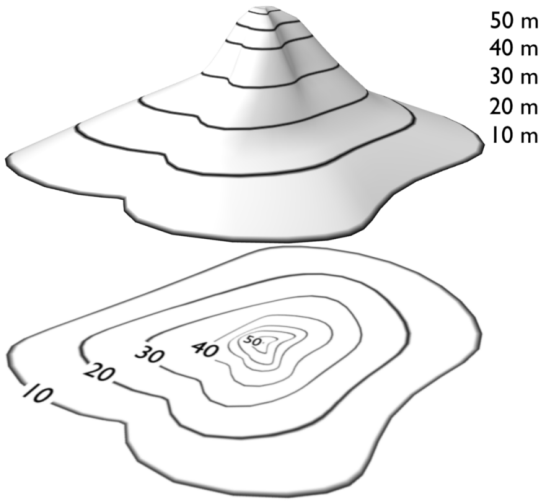


Figura 2.8: Ejemplo de isosuperficies en un campo escalar bidimensional.

Una isosuperficie de un campo escalar es el conjunto de puntos $\mathbf{p}$ del espacio n-dimensional que cumplen $\phi(\mathbf{p}) = k$, donde k es algún valor real dado. Podemos elegir, sin pérdida de generalidad, $k = 0$. En este trabajo nos interesan los casos donde $n \in \{2, 3\}$. En la figura 2.8 se muestra un ejemplo para el caso bidimensional ($n = 2$), en el cual se observa una montaña que tiene una altura asociada a cada coordenada bidimensional. En este caso las isosuperficies constituyen las curvas de nivel que se muestran en la parte inferior.

Campo escalar para superficies moleculares

En el trabajo de Blinn [1] se utilizó por primera vez este método para modelar superficies moleculares. Para esta aplicación particular, resulta ideal la definición de ϕ que Blinn ofrece. Utiliza una combinación de varias funciones gaussianas como sigue:

$$\phi(\mathbf{p}) = \sum_i \beta_i e^{-\alpha_i r_i^2} - T \tag{2.11}$$

con

$$r_i = |\mathbf{p} - \mathbf{p}_i|$$

donde $\mathbf{p}_i$ es la posición de la i-ésima función gaussiana que corresponde a un átomo de la molécula que se desea modelar, los parámetros α_i y β_i son, respectivamente, la desviación estándar de la gaussiana y el peso que se le da. El parámetro T es un umbral (*threshold*) que define la isosuperficie que se elige visualizar. El principio físico que rige la densidad de electrones en cada átomo indica que el factor r_i en la ecuación (2.11) no aparezca elevado al cuadrado, sin embargo, Blinn lo ha puesto de esta manera para ahorrar el cálculo de una raíz cuadrada y así optimizar el tiempo de cómputo, sin degradar visiblemente el resultado final. En la figura 2.9 se muestran

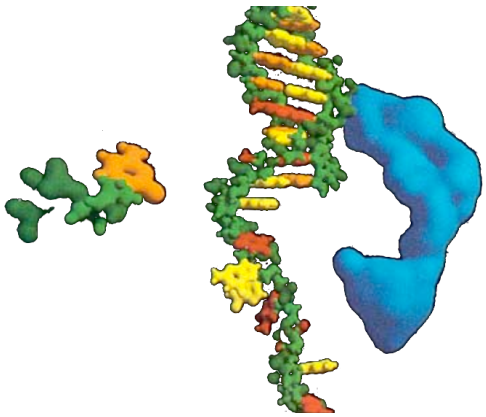


Figura 2.9: Ejemplos de moléculas modeladas con objetos sin forma por Blinn [1].

algunos ejemplos de moléculas modeladas con esta técnica.

Debido a que las superficies obtenidas por esta técnica se asemejan a gotas que se fusionan y se separan, ha sido la elección natural por muchos investigadores que necesitan visualizar fluidos de cualquier tipo. Para el caso de modelar la superficie de un fluido se considera que éste está compuesto de partículas de fluido, en analogía con los átomos que componen las moléculas. De esta manera la superficie obtenida envuelve a las partículas de fluido, como se requiere.

Entre las ventajas de esta técnica destaca su sencillez, que la superficie obtenida es suave y que soporta de manera elegante los cambios de topología necesarios para los objetos fluidos[13], es decir: que un cuerpo se separe en muchos y que muchos se puedan fusionar en uno. Tiene la desventaja de que su eficiencia se reduce considerablemente cuando se incrementa el número de elementos que componen el campo escalar.

Campo escalar para superficies de fluido

Otra desventaja importante, de acuerdo a Zhu y Bridson [2], es que resulta prácticamente imposible modelar superficies de fluido perfectamente planas, cónicas o esféricas a partir de una gran cantidad de partículas de fluido colocadas de manera irregular. Siempre resultan visibles ciertas protuberancias que hacen parecer *granuloso* al fluido, evidenciando la posición de las partículas. Para remediar esto, proponen un campo escalar distinto ϕ con el cual se reconstruye exactamente el campo de distancia con signo:

$$\phi(\mathbf{p}) = |\mathbf{p} - \overline{\mathbf{p}}| - \overline{\mathbf{r}}$$

[13]Para una exposición de los cambios de topología en fluidos y su implementación en mallas deformables, consulte [14].

con

$$\overline{\mathbf{p}} = \sum_i w_i \overline{\mathbf{p}}_i$$

$$\overline{\mathbf{r}} = \sum_i w_i \overline{\mathbf{r}}_i$$

$$w_i = \frac{k(|\mathbf{p} - \mathbf{p}_i| / R)}{\sum_j k(|\mathbf{p} - \mathbf{p}_j| / R)}$$

donde k es una función de ventana apropiada que tiende a cero suavemente y R es el radio de la vecindad alrededor de la posición $\mathbf{p}$. En su trabajo utilizaron la ventana $k(s) = \max(0, (1 - s^2)^3)$ para evitar la necesidad de calcular raíces cuadradas. En la figura 2.10 se muestra la diferencia de resultados entre los dos campos escalares vistos.

Por su naturaleza implícita, el método de visualización de trazo de rayos se puede aplicar de manera directa a los objetos sin forma, sin un paso intermedio como ocurre con otros métodos en los que es necesario extraer primero una superficie formada por polígonos. Aquí es interesante mencionar el trabajo de Gourmel et. al. [35] en el cual realizan una implementación que balancea la carga entre el CPU y el GPU usando CUDA y una estructura de datos de aceleración basada en jerarquías de volumen de acotamiento (*Bounding Volume Hierarchies, BVH*) y árboles k-dimensionales (*kd-trees*), para el trazo de rayos eficiente de una gran cantidad de objetos sin forma.

Hemos utilizado el programa POVray para el trazado de rayos y aprovechamos la implementación de objetos sin forma que éste incluye. Cabe aclarar que no modificamos la implementación del trazador de rayos, ni tampoco implementamos los objetos sin forma. Se utilizó el entorno *blob* de povray aplicado sobre primitivas esféricas de radio h que corresponde al radio de soporte empleado en las ventanas del simuador HSP.

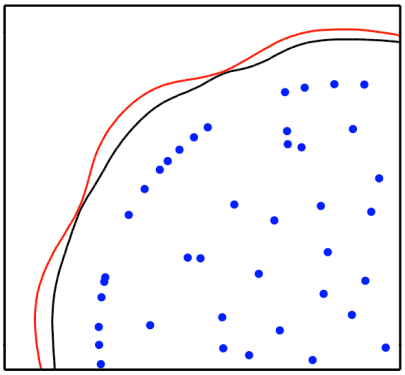


Figura 2.10: Comparación presentada en [2] para el modelado de una forma circular en el caso bidimensional. En azul se muestran las partículas, la curva roja fue obtenida con blobs y la curva negra con el campo escalar de Zhu y Bridson.

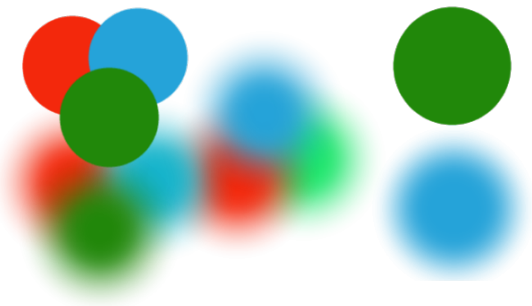


Figura 2.11: Ejemplos de aplanados opacos y difusos.

2.2.3. Aplanamiento

De acuerdo a lo expuesto en la tesis de maestría de María del Rosario Martínez [36], la técnica de *aplanamiento* (*splatting*) linealiza por trozos la superficie del objeto que se desea visualizar, utilizando una primitiva geométrica, denominada aplanado (*splat*), y orientada de acuerdo a la normal que tenga la superficie en el punto correspondiente. El aplanado consiste en una figura poligonal plana. En dicho trabajo se implementó con forma de triángulo, cuadrado y círculo. Cada uno tiene las siguientes características: posición, orientación, área y color. En el caso del aplanado circular su área viene dada por su radio. En cuanto a su color, el aplanado puede ser opaco o difuso. En este último, el color se va degradando del centro hacia afuera, permitiendo que la superficie luzca más suave al mezclarse los colores de los aplanados que se traslapen. En la figura 2.11 se muestran ejemplos de aplanado circular opaco y difuso. La técnica se utiliza frecuentemente para la visualización de grandes cantidades de datos obtenidos mediante el uso de escáneres médicos, como es el caso de los tomógrafos de rayos X. Es una técnica muy rápida y puede ser acelerada por hardware, pero requiere una gran cantidad de puntos para lucir convincente. Por este motivo, para este trabajo se ha tomado una aproximación ligeramente diferente, no orientada a grandes conjuntos de datos obtenidos en rebanadas, sino cantidades menores de datos disconexos y posiblemente desordenados. Nuestra aproximación se explica en detalle en la sección 4.3.

El aplanado resulta similar al concepto de *elemento de superficie* (*surface element* o *surfel* [14]) dado en la literatura de topología discreta y cuya definición se cita en [37] como un objeto $(n-1)$-dimensional orientado en $\mathbb{R}^n$ y, debemos agregar, de tamaño finito. Con $n = 3$ podemos considerarlo como un cuadrado unitario orientado y puede ser asociado con la cara visible de un *vóxel* unitario, suponiendo que el objeto viene aproximado por un conjunto de estos vóxeles orientados.

[14] Por analogía con el término *pixel* se construyen los términos *surfel* y *vóxel* que denotan elemento de superficie y elemento de volumen, respectivamente.

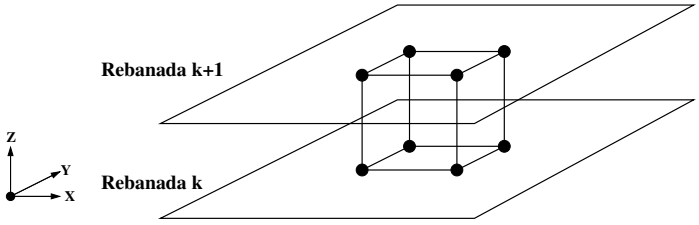


Figura 2.12: Un cubo definido entre dos rebanadas.

2.2.4. Encajamiento de cubos

El algoritmo de *encajamiento de cubos* (*marching cubes*) fué introducido por Lorensen y Cline [38] para visualizar superficies tridimensionales a partir de los datos volumétricos obtenidos por escáneres médicos tales como tomógrafos de rayos X y de resonancia magnética.

La idea básica es ir descubriendo la manera en que la superficie buscada intersecta un cubo que se va desplazando a lo largo y ancho del volumen comprendido entre dos rebanadas adyacentes de datos. Recordemos que los datos médicos suelen venir dados en rebanadas y por ello resulta adecuado utilizar la idea de un cubo que se va desplazando entre rebanadas, como se ilustra en la figura 2.12, y a su paso va construyendo la superficie.

Hemos mencionado en la sección 1.3, páginas 5 y 7, un par de variantes de este algoritmo, las cuales utilizan otras primitivas capaces de teselar el volumen de datos, que se van desplazando para construir la superficie. También es posible utilizar tetraedros[15], además de rebanadas [16] y mosaicos poligonales [15]. Para un estudio acerca de esta técnica, sus variantes y estrategias de optimización, consulte [3].

Método estándar

Volvamos ahora a la descripción del método estándar dado por Lorensen y Cline [38]. Ya se mencionó que introdujeron la idea de ir desplazando un cubo a través del volumen de datos para determinar la superficie correspondiente a una cierta intensidad del valor escalar. Para cada posición del cubo, se verifica el valor escalar medido en sus ocho esquinas. Según su valor, la esquina se halla fuera, dentro o sobre la superficie buscada. Existen 256 posibles configuraciones para las ocho esquinas. Sin embargo, algunos casos son reflejos, rotaciones o inversiones de otros y, descontando las repeticiones, se reducen los casos a sólo 15.Cabe mencionar que con estos casos surgen problemas de ambigüedad que son resueltos en otros trabajos utilizando más de 15 casos.

[15] http://paulbourke.net/geometry/polygonise/

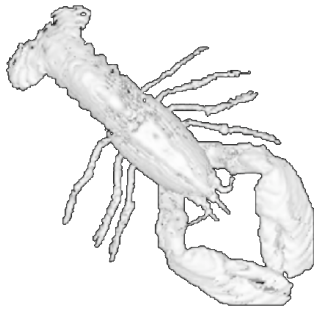


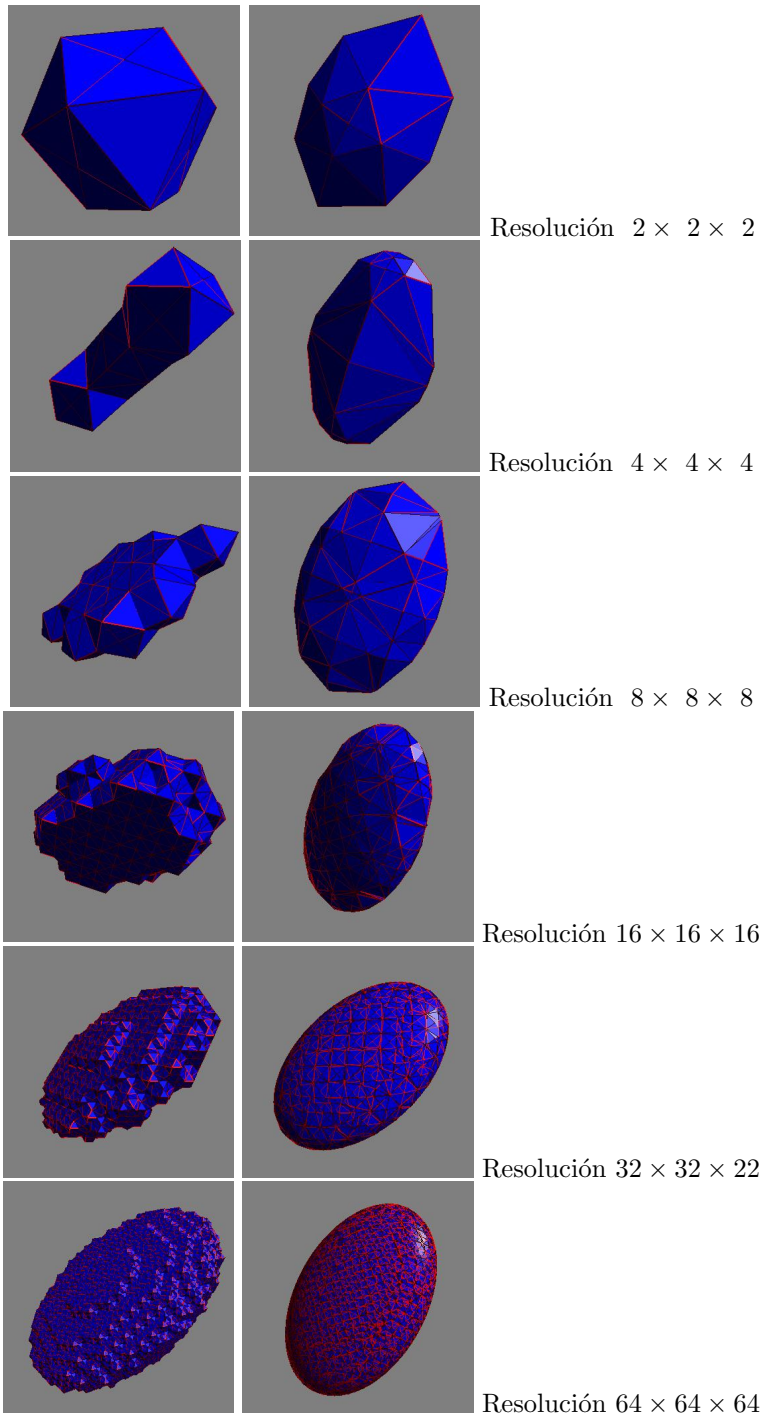
Figura 2.13: Superficie extraída a partir los datos de una langosta con el algoritmo de encajamiento de cubos de Lorensen y Cline. Imagen presentada en [3].

Con ayuda de una tabla de consulta *(look-up table)* es posible determinar de manera eficiente cuál es la configuración para alguna posición del cubo. Con cada iteración, se agrega a la superficie un conjunto de polígonos, por ejemplo triángulos, a partir de una plantilla predefinida y al terminar se tiene una reconstrucción poligonal aproximada de dicha superficie. En la figura 2.13 se muestra un ejemplo de una superficie extraída mediante esta técnica.

En general, el algoritmo es eficiente y es el más popular para recostrucción de superficies en tiempo real. Por ejemplo, ha sido empleado para extraer la isosuperficie de un campo escalar de manera más rápida que al utilizar otras técnicas como el trazo de rayos. Una desventaja es que su eficiencia depende de la cantidad y resolución de los datos, es decir, para cubos muy pequeños y espacios de trabajo muy grandes, el algoritmo toma un tiempo de procesamiento mayor. Sin embargo, es posible paralelizar la evaluación de las esquinas y realizar el cálculo de muchas posiciones a la vez. Existe gran cantidad de trabajos dedicados a optimizar esta técnica para cantidades de datos enormes, como es el caso de [39] que utiliza el conjunto de instrucciones *SIMD* (*Single Instruction Multiple Data*, una instrucción con múltiples datos) de los equipos x86 recientes.

Otra desventaja es la apariencia *voxelizada* que es característica de esta técnica, que resulta evidente en superficies creadas con datos de baja resolución, como en los ejemplos mostrados del lado izquierdo de la figura 2.14. Este efecto es causado por el hecho de que al aplicar las plantillas de creación de triángulos, éstos son fijos con sus esquinas ubicadas siempre a la mitad de una arista del cubo. Para resolver esto[16], es posible utilizar la intersección de la superficie con la arista en lugar del punto medio. Los resultados de esta aproximación se muestran del lado derecho de la figura 2.14.

[16] http://www.quantum-physics.polytechnique.fr/physix/wiki/index.php/Marching_Square_and_Marching_Cube_algorithms



Imágenes de
http://www.quantum-physics.polytechnique.fr/physix/wiki/index.php/Marching_Square_and_Marching_Cube_algorithms

Figura 2.14: A la izquierda se muestran resultados obtenidos para extraer la superficie de un hiperboloide con el algoritmo estándar. A la derecha se muestran los resultados de usar la intersección de la superficie con las aristas del cubo en lugar de los puntos medios.

Capítulo 3

Modificaciones al simulador

Para la implementación del módulo de simulación de nuestra aplicación, se tomó como punto de partida un simulador HSP dado en el trabajo [4]. Este módulo realiza la simulación independientemente de la manera en que se van a visualizar los resultados; simplemente va generando los estados del fluido como se ilustra en la figura 3.1. Se modificó el modelo de colisión entre las partículas de fluido y las superficies del contenedor. Además, se estructuró el código en funciones que permitan modificarlo de manera sencilla para satisfacer los nuevos requerimientos de este trabajo. Los requerimientos que se establecieron para la realización de este proyecto son:

- Utilizar un contenedor con forma de cubo unitario y de esquinas redondeadas.
- Que la colisión del fluido contra el contenedor sea usando su superficie, y no mediante partículas de frontera.
- Que se inicialice el fluido en una forma esférica al centro del contenedor y luego se deje caer bajo el efecto de la gravedad.
- Que el contenedor pueda girar sobre un eje y el contenido simulado fluya por efecto de la gravedad.

En las secciones siguientes se explica cómo se realizaron las modificaciones pertinentes. Fue posible disminuir de forma notable el tiempo de simulación gracias principalmente a dos mecanismos: uno fue acelerar la búsqueda de vecinos mediante la estructura de datos descrita en la sección 2.1.3 (en la pág. 16), el otro fue cambiar la ventana de

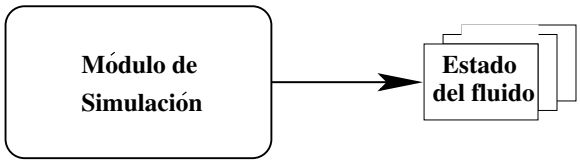


Figura 3.1: Esquema del módulo de simulación.

suavizado por otros más eficientes, lo cual se detalla en la sección 3.1 (en la pág. 32). El diseño del contenedor se detalla en la sección 3.2 (en la pág. 37). El cambio que se realizó en el modelo de la colisión de las partículas de fluido con el contenedor se describe en la sección 3.3 (en la pág. 52). Finalmente, los últimos dos requerimientos fueron resueltos mediante la animación de fuerzas externas al fluido. El caso de la formación de una bola de fluido se encuentra en la sección 3.4 (en la pág. 56) y la rotación de la caja en la sección 3.5 (en la pág. 59).

Adicionalmente, para obtener una simulación interactiva, se agregó un mecanismo de limitación de la velocidad de las partículas con el fin de incrementar la estabilidad de la simulación cuando se utilizan pasos de tiempo mayores. A este mecanismo lo llamamos *velocidad terminal artificial* y se explica en la sección 3.6 (en la pág. 60).

3.1. Ventanas

Como hemos mencionado en la sección 2.1 sobre hidrodinámica suavizada con partículas, se utiliza una ventana de suavizado W, para la interpolación, que cumple las condiciones de ser par y normalizada. De acuerdo a Monaghan [29], estas dos propiedades bastan para que el método funcione, y en general cada uno es libre de elegir el tipo de ventana que mejor satisfaga las necesidades particulares de la aplicación. Para realizar simulaciones en astrofísica, generalmente no se requiere de interacción por parte del usuario y es más importante la exactitud. Para estos casos, en la literatura usualmente se utiliza el *B-Spline* propuesto por Monaghan y Lattanzio, empleado y definido en [4] como:

$$W(\mathbf{r},h) = C \begin{cases} 1-\frac{3}{2}q^2+\frac{3}{4}q^3; & \text{si} \quad 0 \le q < 1 \\ \frac{1}{4}(2-q)^3; & \text{si} \quad 1 \le q < 2 \\ 0; & \text{de lo contrario} \end{cases} \tag{3.1}$$

donde $q = \frac{|\mathbf{r}|}{h}$, $|\mathbf{r}|$ es la distancia al centro de la partícula suavizada y h es el radio de soporte de la ventana. El valor de la constante C depende de la dimensión en la que se efectúa la simulación. En una dimensión $C = \frac{2}{3h}$, en dos dimensiones $C = \frac{10}{7h^2}$ y para tres dimensiones $C = \frac{1}{\pi h^3}$. En la figura 3.2 se muestra la gráfica de la ventana $W : \mathbb{R}^2 \to \mathbb{R}$, para un corte transversal de tal ventana.

Sin embargo, Müller [5] afirma que la elección de la ventana afecta la exactitud, velocidad y estabilidad de la implementación. Agrega también la propiedad de que, tanto la ventana W como su gradiente ∇W, tiendan a cero cuando nos alejamos a una distancia h, pues esto favorece la estabilidad del algortimo. A Müller le interesan las aplicaciones de animación interactiva de fluidos, a expensas de algún nivel de exactitud en la simulación. Lo importante es que los fluidos simulados resulten físicamente plausibles, sin necesidad de lograr predicciones exactas acerca de su comportamiento.

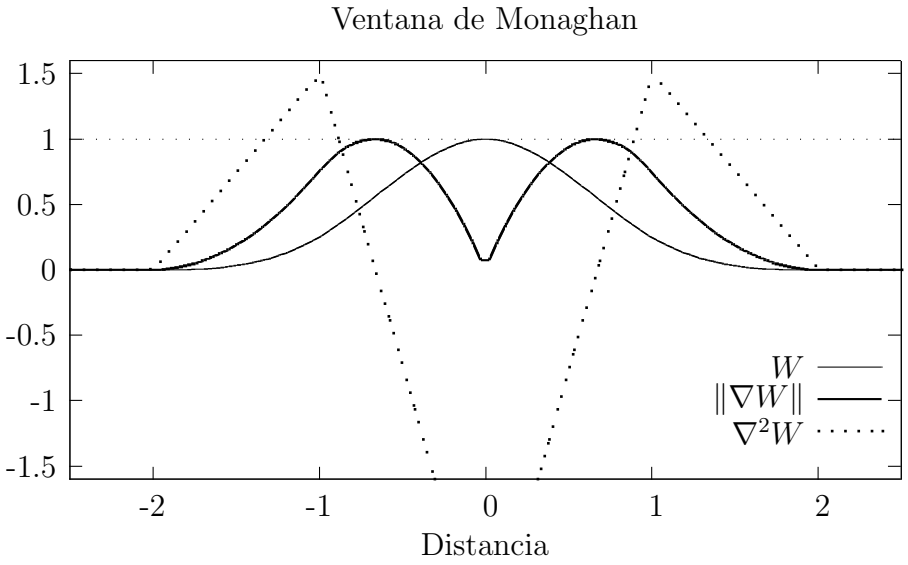


Figura 3.2: Gráfica de la ventana B-Spline de Monaghan en línea delgada, la magnitud del gradiente en línea gruesa y el Laplaciano en línea punteada.

Para ello propone tres ventanas diseñadas con el fin de acelerar el algoritmo, definidas como sigue:

$$W_{\text{poly6}}(\mathbf{r},h) = C_{\text{poly6}} \begin{cases} (h^2-|\mathbf{r}|^2)^3; & \text{si} \quad 0 \le |\mathbf{r}| < h \\ 0; & \text{de lo contrario} \end{cases} \tag{3.2}$$

$$W_{\text{spiky}}(\mathbf{r},h) = C_{\text{spiky}} \begin{cases} (h-|\mathbf{r}|)^3; & \text{si} \quad 0 \le |\mathbf{r}| < h \\ 0; & \text{de lo contrario} \end{cases} \tag{3.3}$$

$$W_{\text{viscosity}}(\mathbf{r},h) = C_{\text{viscosity}} \begin{cases} -\frac{|\mathbf{r}|^3}{2h^3} + \frac{|\mathbf{r}|^2}{h^2} + \frac{h}{2|\mathbf{r}|} - 1; & \text{si} \quad 0 \le |\mathbf{r}| < h \\ 0; & \text{de lo contrario} \end{cases} \tag{3.4}$$

El valor de las constantes C_{poly6}, C_{spiky} y $C_{\text{viscosity}}$, depende de la dimensión en la que se efectúa la simulación. Los valores se dan en la tabla 3.1. En las figuras 3.3, 3.4 y 3.5 se muestran las gráficas de las tres ventanas de Müller: W_{poly6}, W_{spiky} y $W_{\text{viscosity}}$, respectivamente.

La primera diferencia, entre las ventanas de Müller y la de Monaghan, es que las primeras tienden a cero a medida que nos alejamos a una distancia h, mientras que la segunda lo hace a una distancia $2h$. Esto se ilustra en la figura 3.6, donde se indica, para la partícula roja, el área de influencia de radio h en verde y el área de influencia de radio $2h$ en azul. Podemos observar en este diagrama que, dada una cierta configuración de partículas ubicadas en las inmediaciones de la partícula roja, se encuentran menos partículas en la zona verde de radio h que en la zona azul de radio $2h$. Por lo tanto, se reduce el número de partículas en la búsqueda de vecinos.

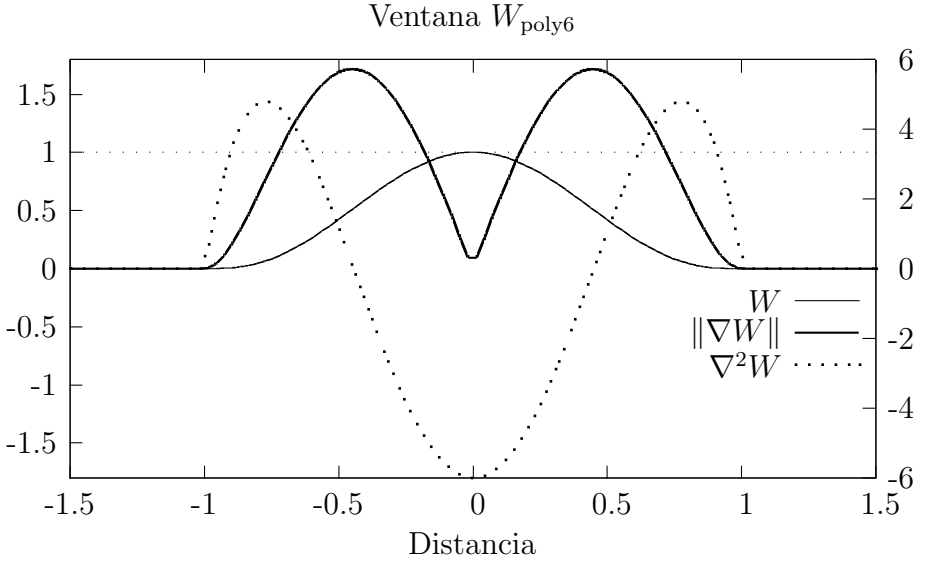


Figura 3.3: Gráficas de la ventana *poly6* de Müller en el caso tridimensonal, a través de un eje que pasa por el centro. Se muestra la ventana en línea delgada, la magnitud del gradiente en línea gruesa y el Laplaciano en línea punteada.

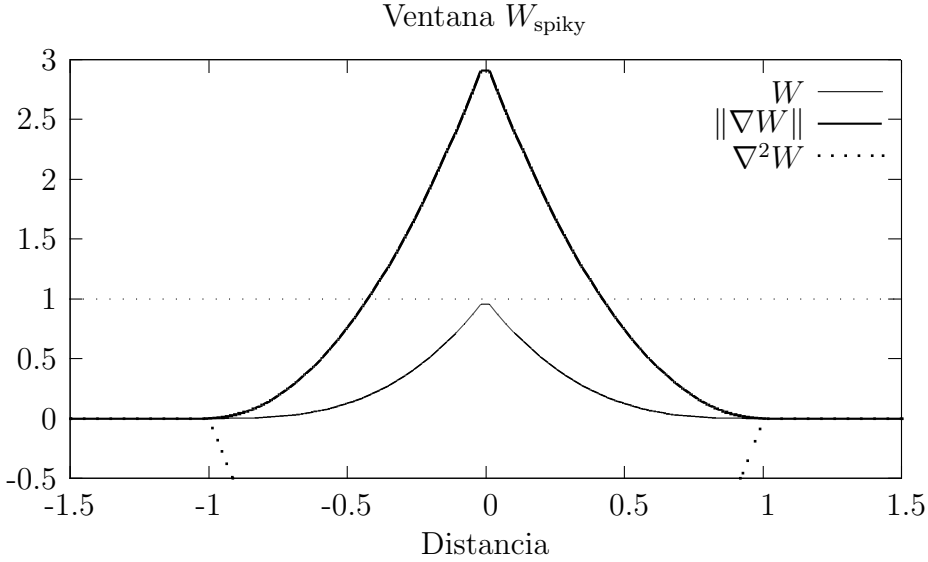


Figura 3.4: Gráficas de la ventana *spiky* de Müller en el caso tridimensonal, a través de un eje que pasa por el centro. Se muestra la ventana en línea delgada, la magnitud del gradiente en línea gruesa y el Laplaciano en línea punteada.

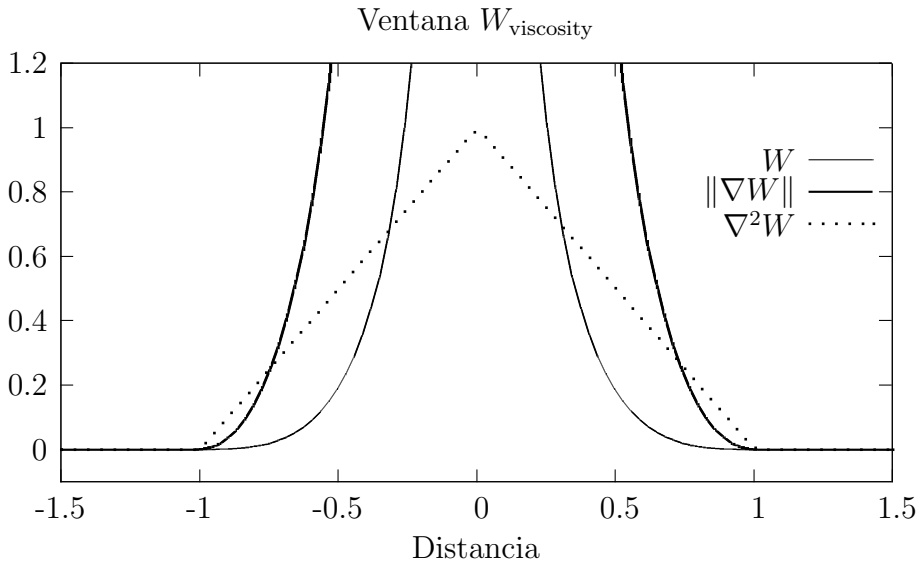


Figura 3.5: Gráficas de la ventana *viscosity* de Müller en el caso tridimensonal, a través de un eje que pasa por el centro. Se muestra la ventana en línea delgada, la magnitud del gradiente en línea gruesa y el Laplaciano en línea punteada.

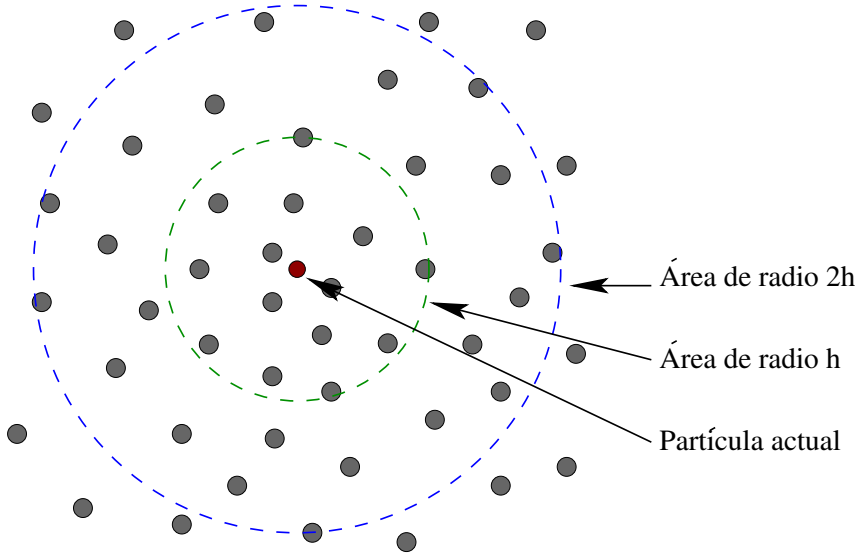


Figura 3.6: Diagrama de alcance desde la partícula roja, para un radio de soporte h en comparación con un radio $2h$.

Tabla 3.1: Valores para las constantes C_{poly6}, C_{spiky} y $C_{\text{viscosity}}$ en dos y tres dimensiones.

Espacio	$\mathbf{C}_{\text{poly6}}$	$\mathbf{C}_{\text{spiky}}$	$\mathbf{C}_{\text{viscosity}}$
2D	$\frac{4}{\pi h^8}$	$\frac{10}{\pi h^5}$	$\frac{10}{3\pi h^2}$
3D	$\frac{315}{64\pi h^9}$	$\frac{15}{\pi h^6}$	$\frac{15}{2\pi h^3}$

La ventana W_{poly6}, definida en la ecuación (3.2), es más eficiente porque es muy simple y la distancia $|\mathbf{r}|$ sólo aparece elevada al cuadrado, esto significa que puede evaluarse sin necesidad de calcular la raíz cuadrada, la cual es una operación costosa. Esta ventana es utilizado en todos los cálculos excepto en dos casos, para los cuales se diseñaron las otras dos ventanas.

Si se utiliza la ventana W_{poly6} para el cálculo de las fuerzas de presión, de acuerdo a Müller, las partículas forman *clusters* cuando se encuentran a muy alta presión. Esto es debido a que, cuando las partículas se encuentran muy juntas, la fuerza de repulsión entre ellas se desvanece pues el gradiente de la ventana tiende a cero cerca de la partícula, como se aprecia en la figura 3.3. Müller resuelve el problema utilizando para el cálculo de presión, la ventana W_{spiky} (ecuación (3.3)) propuesto por Desbrun, cuyo gradiente no se desvanece cerca del centro y genera la fuerza de repulsión necesaria para separar las partículas que llegan a juntarse demasiado.

El otro caso en que no se pueden utilizar las ventanas estándar es el cálculo de la viscosidad. Como se menciona en la sección 2.1.2, ecuación (2.9), la fuerza de viscosidad es calculada por Müller utilizando el Laplaciano de la ventana. El problema radica en que cuando las partículas llegan a acercarse demasiado unas a otras, el Laplaciano se vuelve negativo, ocasionando que las partículas se aceleren en lugar de frenarse y se produce un efecto opuesto al viscoso. Es posible que éste sea el motivo por el cual Monaghan introduce la viscosidad artificial (ecuación (2.8)). Este efecto se observa en aplicaciones interactivas, donde el número de partículas es relativamente bajo y resulta en problemas de estabilidad. Para resolver este problema se diseñó la ventana $W_{\text{viscosity}}$ (ecuación (3.4)) con el cual se mejora considerablemente la estabilidad de la simulación. Nótese en la figura 3.5 que el Laplaciano de $W_{\text{viscosity}}$ es positivo y aumenta al centro.

En la aplicación creada en el presente trabajo de tesis se puede realizar la simulación de las dos maneras descritas:

- utilizando la ventana B-Spline y la viscosidad de Monaghan, o bien,
- utilizando las tres ventanas de Müller y calculando la viscosidad mediante el Laplaciano de la ventana $W_{\text{viscosity}}$.

Sin embargo, se recomienda utilizar la segunda opción si se desea una simulación interactiva.

3.2. Diseño del contenedor

En la simulación es necesario un contenedor que confine las partículas de fluido dentro de un espacio de trabajo dado. En este trabajo se ha realizado una abstracción del contenedor en la forma de una función que detecta si una partícula determinada se sale del mismo, verificando si hace colisión con alguna de las superficies que lo conforman.

A la función se le proporciona como entrada un segmento de recta $\overline{\mathbf{ab}}$, definido por los puntos inicial y final del desplazamiento de una partícula durante un intervalo de tiempo. También se le proporciona información sobre la forma del contenedor deseado. Con ello, el algoritmo determina si el segmento interseca alguna de las superficies que conforman al contenedor. Si ocurre la intersección, se devuelve el punto de contacto $\mathbf{p}'$ y un vector $\mathbf{n}$ normal a la superficie en dicho punto. Esta información es suficiente para manejar la reacción a la colisión. En este trabajo se ha modelado la reacción a la colisión de la manera expuesta en la sección 3.3.

Función *dentroContenedor*
ENTRADA: Un segmento de recta $\overline{\mathbf{ab}}$, la forma del contenedor definida bajo ciertos parámetros.
SALIDA: Un valor booleano que indica si hubo contacto entre el segmento y la forma, y si lo hubo, regresa el punto de contacto $\mathbf{p}' = (x', y')$ y el vector normal $\mathbf{n}$ a la superficie en dicho punto.

Las implementaciones de esta función deben cumplir las siguientes condiciones:

- Que el punto inicial $\mathbf{a}$ se encuentre siempre dentro o sobre la superficie del contenedor.
- Que el punto $\mathbf{b}$ pueda estar en cualquier parte, incluyendo la posición de $\mathbf{a}$, esto ocurre cuando la partícula no se mueve.
- Que el punto de contacto obtenido $\mathbf{p}'$ siempre esté sobre la superficie del contenedor.
- Que el vector normal $\mathbf{n}$ se encuentre definido en todos los puntos de la superficie del contenedor.

Se implementó el contenedor con forma de caja unitaria con esquinas redondeadas[1] para cumplir con esta última condición, y se hizo para dos y tres dimensiones. A continuación se detallan los algoritmos diseñados para estos dos casos.

[1] Nos referiremos a ella de manera corta como *caja redondeada*.

3.2.1. Diseño de un contenedor bidimensional

Para la simulación de fluidos por medio de partículas se necesita diseñar un contenedor en el que se encuentre confinado el fluido que se desea simular. Dicho contenedor tiene la forma de una caja con las esquinas redondeadas como se muestra en la figura 3.7 para el caso bidimensional:

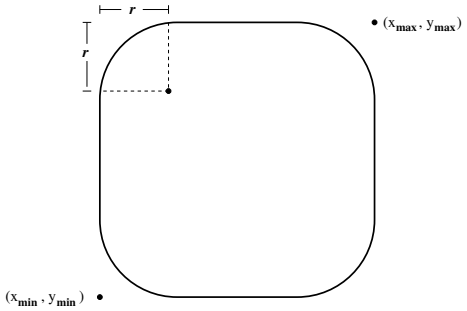


Figura 3.7: Caja redondeada en dos dimensiones.

En este caso, el problema reside en poder detectar cuando una partícula cruza el contenedor (para de esta forma devolverla dentro de él). Una partícula estará dentro de la caja en un tiempo t y hay que detectar si la misma partícula cruza el contenedor en el tiempo $t+1$. La posición de la partícula será $\mathbf{p}_0$ en el tiempo t y $\mathbf{p}_1$ en el tiempo $t+1$, por lo que el problema es detectar cuando el punto $\mathbf{p}_1$ está fuera de la caja, y si está fuera, calcular el punto de intersección del borde del contenedor con la línea $\overline{\mathbf{p}_0\mathbf{p}_1}$.

Se considerará una prueba básica: un punto $\mathbf{p} = (x, y)$ está dentro de una caja, como la mostrada en la figura 3.8, si:

$$x_{\min} \leq x \leq x_{\max} \quad \& \quad y_{\min} \leq y \leq y_{\max}$$

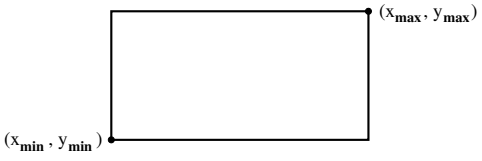


Figura 3.8: Una caja definida por dos puntos.

Entonces, el problema se puede resolver por medio de un algoritmo con tres pasos.

Paso 1 Si $\mathbf{p}_1 = (x_1, y_1)$ está dentro de la caja de la figura 3.9 izquierda, esto es, dentro de la caja definida por los puntos $(x_{\min}+r, y_{\min})$ y $(x_{\max}-r, y_{\max})$, el algoritmo termina, regresando el valor de *FALSO*, indicando que no hubo intersección con el contenedor.

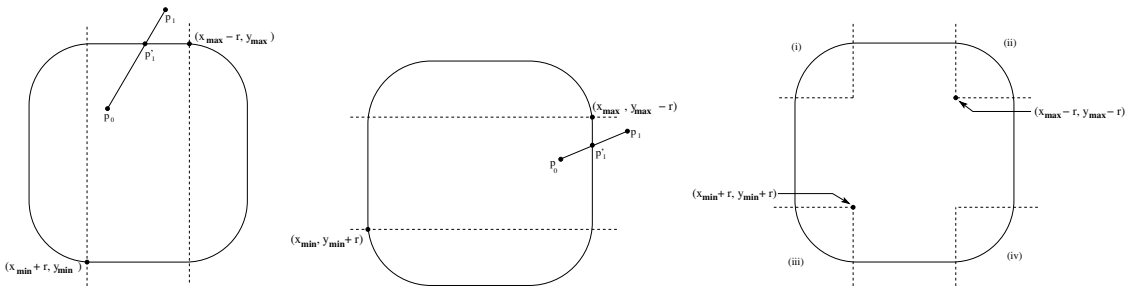


Figura 3.9: Diagramas para el paso 1, 2 y 3.

Si $(y_1 > y_{\max})$ se recorta con línea $y = y_{\max}$, mediante $x' = x_0 + (x_1 - x_0)(y_{\max} - y_0)/(y_1 - y_0)$, $y' = y_{\max}$.

Si $(x_{\min} + r \leq x' \leq x_{\max} - r)$ el algoritmo termina, regresando el valor de *VERDADERO*, indicando que hubo intersección con el contenedor. El vector $\mathbf{n}$ unitario normal a la caja en el punto (x', y') será $\mathbf{n} = (0, 1)$.

Si $(y_1 < y_{\min})$ se recorta con línea $y = y_{\min}$, mediante $x' = x_0 + (x_1 - x_0)(y_{\min} - y_0)/(y_1 - y_0)$, $y' = y_{\min}$.

Si $(x_{\min} + r \leq x' \leq x_{\max} - r)$ el algoritmo termina, regresando el valor de *VERDADERO*, indicando que hubo intersección con el contenedor. El vector $\mathbf{n}$ unitario normal a la caja en el punto (x', y') será $\mathbf{n} = (0, -1)$.

Paso 2 Si el punto $\mathbf{p}_1 = (x_1, y_1)$[2] está dentro de la caja de la figura 3.9 central, definida por los puntos $(x_{\min}, y_{\min} + r)$ y $(x_{\max}, y_{\max} - r)$, el algoritmo termina, regresando el valor de *FALSO*.

Si $(x_1 > x_{\max})$ se recorta con línea $x = x_{\max}$, mediante $y' = y_0 + (y_1 - y_0)(x_{\max} - x_0)/(x_1 - x_0)$, $x' = x_{\max}$.

Si $(y_{\min} + r \leq y' \leq y_{\max} - r)$ el algoritmo termina, regresando el valor de *VERDADERO*. El vector $\mathbf{n}$ unitario normal a la caja en el punto (x', y') será $\mathbf{n} = (1, 0)$.

Si $(x_1 < x_{\min})$ se recorta con línea $x = x_{\min}$, mediante $y' = y_0 + (y_1 - y_0)(x_{\min} - x_0)/(x_1 - x_0)$, $x' = x_{\min}$.

Si $(y_{\min} + r \leq y' \leq y_{\max} - r)$ el algoritmo termina, regresando el valor de *VERDADERO*. El vector $\mathbf{n}$ unitario normal a la caja en el punto (x', y') será $\mathbf{n} = (-1, 0)$.

Paso 3 En este punto, sólo falta checar la posición del punto $\mathbf{p}'_1 = (x', y')$ recortado, lo que nos resulta en los cuatro casos mostrados en la figura 3.9 derecha.

Caso (i) El punto $\mathbf{p}'_1 = (x', y')$ está en la esquina superior izquierda, esto es cumple:

$$x' < x_{\min} + r \quad \& \quad y' > y_{\max} - r$$

y se debe recortar la línea $\overline{\mathbf{p}_0\mathbf{p}'_1}$ respecto al cuarto de círculo superior izquierdo.

Caso (ii) El punto $\mathbf{p}'_1 = (x', y')$ se halla en la esquina superior derecha, esto es, cumple

$$x' > x_{\max} - r \quad \& \quad y' > y_{\max} - r$$

[2]que puede ya estar recortado por la aplicación del paso 1

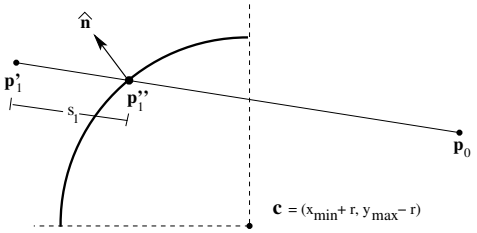


Figura 3.10: Diagrama para el caso (i).

y se debe recortar la línea $\overline{\mathbf{p}_0\mathbf{p}_1'}$ respecto al cuarto del círculo superior derecho.

Caso (iii) El punto $\mathbf{p}_1' = (x', y')$ cumple

$$x' < x_{\min} + r \quad \& \quad y' < y_{\min} + r$$

y se debe recortar la línea $\overline{\mathbf{p}_0\mathbf{p}_1'}$ respecto al cuarto del círculo inferior izquierdo.

Caso (iv) El punto $\mathbf{p}_1' = (x', y')$ cumple

$$x' > x_{\max} - r \quad \& \quad y' < y_{\min} + r$$

y hay que recortar la línea $\overline{\mathbf{p}_0\mathbf{p}_1'}$ respecto al cuarto del círculo inferior derecho.

Para las intersecciones $\overline{\mathbf{p}_0\mathbf{p}_1'}$ con el cuarto de círculo, el punto $\mathbf{p}_0$ está dentro de la caja redondeada, pero no necesariamente dentro de la zona del cuarto de círculo.

El punto de intersección $\mathbf{p}_1''$ será $\mathbf{p}_1'' = \mathbf{p}_1' + s\hat{u}$, donde s es una de las soluciones s_1, s_2 de la ecuación $||\mathbf{p}_1' + s\mathbf{u} - \mathbf{c}|| = r$ con $\mathbf{u} = \frac{\mathbf{p}_0 - \mathbf{p}_1'}{||\mathbf{p}_0 - \mathbf{p}_1'||}$ y $\mathbf{c} = (x_{\min} + r, y_{\max} - r)$.

Esta ecuación la podemos escribir (ver [40], pp. 177-178) en la forma:

$$s^2(\mathbf{u} \cdot \mathbf{u}) + 2s(\mathbf{m} \cdot \mathbf{u}) + \mathbf{m} \cdot \mathbf{m} - r^2 = 0$$

donde $\mathbf{m} = \mathbf{p}_1' - \mathbf{c}$.

Se calcula:

$$d = (\mathbf{m} \cdot \mathbf{u})^2 - \mathbf{m} \cdot \mathbf{m} - r^2$$

Si d es negativo, se concluye que no existen soluciones reales para la ecuación, algo que no sucede dado que $\mathbf{p}_1'$ está en alguna de las esquinas. Si d es positivo, las soluciones serán:

$$s_1 = -\mathbf{m} \cdot \mathbf{u} - \sqrt{d} \quad \& \quad s_2 = -\mathbf{m} \cdot \mathbf{u} + \sqrt{d}$$

Si una de las soluciones es negativa, $\mathbf{p}_1'$ está dentro de la caja redondeada, entonces el algoritmo termina regresando el valor de *FALSO*. Si las dos soluciones son positivas, el punto $\mathbf{p}_1'$ está fuera del cuarto de círculo de la esquina redondeada, y se escogerá la solución más cercana al punto $\mathbf{p}_1'$, esto es, la menor de ellas[3], como se muestra para el caso (i) en la figura 3.10. El algoritmo regresará *VERDADERO* y se debe calcular el vector unitario $\mathbf{n} = \frac{\mathbf{p}_1'' - \mathbf{c}}{||\mathbf{p}_1'' - \mathbf{c}||}$.

El algoritmo en pseudocódigo quedará como en la figura 3.11.

[3]Observemos que cuando $d > 0$, se tiene $s_1 \leq s_2$, por lo que $s = s_1$.

Algoritmo *dentroCajaRedondeada2D*
ENTRADA: Los puntos $\mathbf{p}_0 = (x_0, y_0)$ & $\mathbf{p}_1 = (x_1, y_1)$ del segmento $\overline{\mathbf{p}_0\mathbf{p}_1}$,
la esquina inferior izquierda $\mathbf{A} = (x_{\min}, y_{\min})$, la esquina superior derecha $\mathbf{B} = (x_{\max}, y_{\max})$
de la caja redondeada y el radio r de la esquina.
SALIDA: Un valor booleano que indica si hubo contacto, y si lo hubo,
regresa el punto de contacto $\mathbf{p}' = (x', y')$ del segmento $\overline{\mathbf{p}_0\mathbf{p}_1}$ con la caja redondeada
y el vector normal $\mathbf{n}$ de la caja en el punto $\mathbf{p}'$.
begin
codigo ← *codigo2D* ($\mathbf{p}_1$, $(x_{\min} + r, y_{\min})$, $(x_{\max} - r, y_{\max})$) // PASO 1
if (!codigo) **return** FALSO
else if (codigo & ARRIBA)
$x' \leftarrow x_0 + (x_1 - x_0)\frac{y_{\max} - y_0}{y_1 - y_0}$, $y' \leftarrow y_{\max}$, $\mathbf{p}_1 \leftarrow \mathbf{p}'$
if $(x_{\min} + r \leq x' \leq x_{\max} - r)$
$\mathbf{n} \leftarrow (0, 1)$
return ($\mathbf{p}'$, $\mathbf{n}$, VERDADERO)
endif
else if (codigo & ABAJO)
$x' \leftarrow x_0 + (x_1 - x_0)\frac{y_{\min} - y_0}{y_1 - y_0}$, $y' \leftarrow y_{\min}$, $\mathbf{p}_1 \leftarrow \mathbf{p}'$
if $(x_{\min} + r \leq x' \leq x_{\max} - r)$
$\mathbf{n} \leftarrow (0, -1)$
return ($\mathbf{p}'$, $\mathbf{n}$, VERDADERO)
endif
endif
codigo ← *codigo2D* ($\mathbf{p}_1$, $(x_{\min}, y_{\min} + r)$, $(x_{\max}, y_{\max} - r)$) // PASO 2
if (!codigo) **return** FALSO
else if (codigo & DERECHA)
$y' \leftarrow y_0 + (y_1 - y_0)\frac{x_{\max} - x_0}{x_1 - x_0}$, $x' \leftarrow x_{\max}$, $\mathbf{p}_1 \leftarrow \mathbf{p}'$
if $(y_{\min} + r \leq y' \leq y_{\max} - r)$
$\mathbf{n} \leftarrow (1, 0)$
return ($\mathbf{p}'$, $\mathbf{n}$, VERDADERO)
endif
else if (codigo & IZQUIERDA)
$y' \leftarrow y_0 + (y_1 - y_0)\frac{x_{\min} - x_0}{x_1 - x_0}$, $x' \leftarrow x_{\min}$, $\mathbf{p}_1 \leftarrow \mathbf{p}'$
if $(y_{\min} + r \leq y' \leq y_{\max} - r)$
$\mathbf{n} \leftarrow (-1, 0)$
return ($\mathbf{p}'$, $\mathbf{n}$, VERDADERO)
endif
endif
codigo ← *codigo2D* ($\mathbf{p}_1$, $(x_{\min} + r, y_{\min} + r)$, $(x_{\max} - r, y_{\max} - r)$) // PASO 3
$c_x \leftarrow x_{\min} + r$, $c_y \leftarrow y_{\min} + r$ // $\mathbf{c} = (c_x, c_y)$ esquina inferior derecha
if (codigo & ARRIBA)
$c_y \leftarrow c_y + y_{\max} - y_{\min} - 2r$ // mover a esquina superior
endif
if (codigo & DERECHA)
$c_x \leftarrow c_x + x_{\max} - x_{\min} - 2r$ // mover a esquina derecha
endif
$\mathbf{u} \leftarrow \frac{\mathbf{p}_0 - \mathbf{p}'}{||\mathbf{p}_0 - \mathbf{p}'||}$
return *intersecaRayoCirculo* ($\mathbf{p}_1$, $\mathbf{p}'$, r, $\mathbf{c}$, $\mathbf{u}$, $\mathbf{n}$)
end

Figura 3.11: Algoritmo *dentroCajaRedondeada2D*.

El algoritmo anterior utiliza las siguientes funciones:

Función *intersecaRayoCirculo*
ENTRADA: Un rayo definido por el punto $\mathbf{p}$ y el vector unitario $\mathbf{u}$, el círculo definido por el punto central $\mathbf{p}_c$ y el radio r.
SALIDA: Un valor booleano que indica si hubo contacto, y si lo hubo, devuelve el primer punto de contacto $\mathbf{p}'$ del rayo con el círculo y el vector normal $\mathbf{n}$ en dicho punto.

```
begin
    m ← p − p_c
    b ← m · u
    c ← m · m − r^2
    d ← b^2 − c
    if (d < 0)
        print "ERROR: Esto no puede ser!"
        exit
    else
        s ← −b − √d
        if (s < 0)
            return FALSO
        endif
        p' ← p + su
        n ← (p' − p_c) / ||p' − p_c||
    endif
    return (p', n, VERDADERO)
end
```

Función *codigo2D*
ENTRADA: El punto $\mathbf{p} = (x, y)$, la esquina inferior izquierda $\mathbf{A} = (x_{\min}, y_{\min})$, la esquina superior derecha $\mathbf{B} = (x_{\max}, y_{\max})$ de una caja.
SALIDA: El código de region de cuatro bits.

```
    ARRIBA = 0x01, ABAJO = 0x02,
    DERECHA = 0x04, IZQUIERDA = 0x08
begin
    codigo ← 0
    if (y > y_max)
        codigo |= ARRIBA
    else if (y > y_min)
        codigo |= ABAJO
    endif
    if (x > x_max)
        codigo |= DERECHA
    else if (x > x_min)
        codigo |= IZQUIERDA
    endif
    return codigo
end
```

3.2.2. Diseño de un contenedor tridimensional

En esta sección trataremos el caso del diseño del contenedor tridimensional, el cual tiene la forma de una caja con las esquinas redondeadas como se muestra en la figura 3.12:

Figura 3.12: Caja redondeada en tres dimensiones.

El problema reside en poder detectar cuando una partícula cruza el contenedor (para de esta forma devolverla dentro de él). Una partícula estará dentro de la caja en un tiempo t y hay que detectar si la misma partícula cruza el contenedor en el tiempo $t+1$. La posición de la partícula será $\mathbf{p}_0$ en el tiempo t y $\mathbf{p}_1$ en el tiempo $t+1$, por lo que el problema es detectar cuando el punto $\mathbf{p}_1$ está fuera de la caja, y si está fuera, calcular el punto de intersección del borde del contenedor con la línea $\overline{\mathbf{p}_0\mathbf{p}_1}$.

La caja de la figura 3.13 está representada por los puntos $(x_{\min}, y_{\min}, z_{\min})$ & $(x_{\max}, y_{\max}, z_{\max})$. Para verificar si un punto $\mathbf{p} = (x, y, z)$ está dentro de esta caja, se debe cumplir que:

$$x_{\min} \leq x \leq x_{\max} \quad , \quad y_{\min} \leq y \leq y_{\max} \quad \& \quad z_{\min} \leq z \leq z_{\max}$$

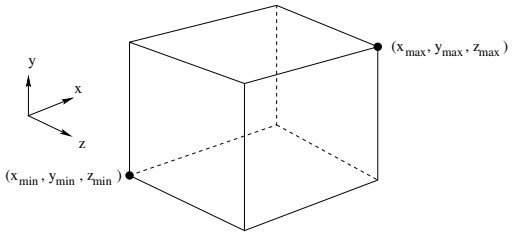


Figura 3.13: Una caja definida por dos puntos.

Entonces, nuestro problema consiste en verificar si un punto $\mathbf{p}_1$ está dentro de un contenedor redondeado, y si está fuera, encontrar el punto de intersección entre la línea $\overline{\mathbf{p}_0\mathbf{p}_1}$ y la envoltura del contenedor redondeado.

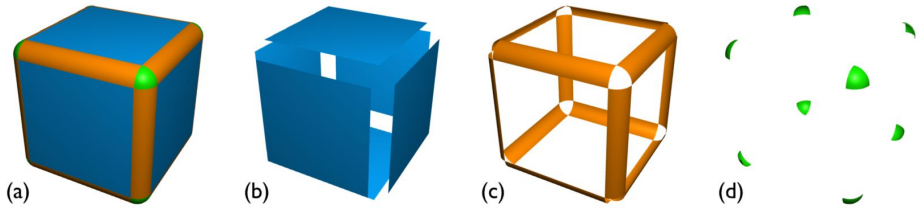


Figura 3.14: División de la caja redondeada para la detección de colisiones. (a) Muestra las tres zonas, (b) solo los planos en las caras, (c) las secciones cilíndricas y (d) la zona esférica.

Se puede resolver el problema en tres pasos (figura 3.14): en el primer paso se verifica el punto recortado respecto a las caras (b), en el segundo paso se verifica el punto recortado respecto a las secciones de cilindro (c) que constituyen las aristas redondeadas de la caja; finalmente se verifica el punto recortado respecto a las secciones de esfera (d) en las esquinas redondeadas.

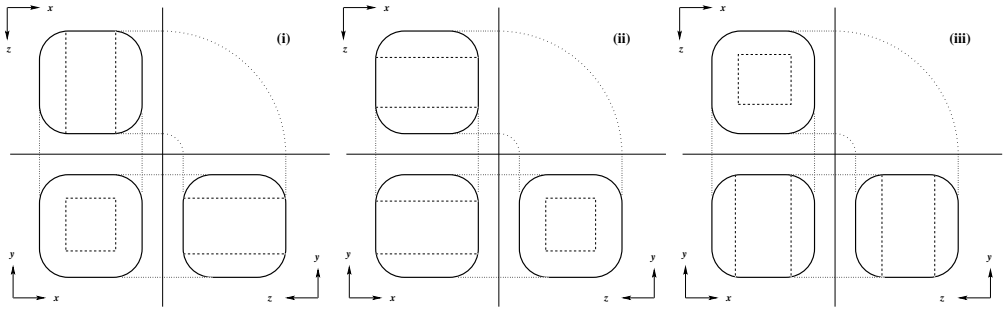


Figura 3.15: Diagrama de los tres casos del paso 1.

Paso 1. La caja correspondiente a la figura 3.15 inciso (i), está definida por los puntos $(x_{\min}+r, y_{\min}+r, z_{\min})$ y $(x_{\max}-r, y_{\max}-r, z_{\max})$. Si $\mathbf{p}_1 = (x_1, y_1)$ está dentro de esta caja el algoritmo termina, devolviendo el valor de *FALSO*, lo cual indica que no hubo intersección con el contenedor.

Si $z < z_{\min}$ ó $z > z_{\max}$, se encuentra el punto de intersección (ver [40], pp. 175-176) de la línea $\overline{\mathbf{p}_0\mathbf{p}_1}$ con el plano $z = z_{\min}$ ó $z = z_{\max}$, respectivamente. Llamemos a este punto $\mathbf{p}' = (x', y', z')$.

Si $x_{\min} + r \leq x' \leq x_{\max} - r$ & $y_{\min} + r \leq y' \leq y_{\max} - r$ el algoritmo termina, devuelve el valor *VERDADERO*, junto con el punto $\mathbf{p}'$ y el vector unitario $\mathbf{n} = (0, 0, 1)$ ó $\mathbf{n} = (0, 0, -1)$, respectivamente.

La caja correspondiente a la figura 3.15 inciso (ii), está definida por los puntos $(x_{\min}, y_{\min}+r, z_{\min}+r)$ y $(x_{\max}, y_{\max}-r, z_{\max}-r)$. Si $\mathbf{p}_1 = (x_1, y_1)$ se encuentra dentro de esta caja el algoritmo termina y devuelve *FALSO*.

Si $x < x_{\min}$ ó $x > x_{\max}$, se encuentra el punto de intersección de la línea $\overline{\mathbf{p}_0\mathbf{p}_1}$ con el plano $x = x_{\min}$ ó $x = x_{\max}$, respectivamente. Si el punto calculado $\mathbf{p}' = (x', y', z')$ cumple $y_{\min} + r \leq y' \leq y_{\max} - r$ & $z_{\min} + r \leq z' \leq z_{\max} - r$, el algoritmo termina,

devuelve *VERDADERO* con el punto $\mathbf{p}'$ y el vector unitario $\mathbf{n} = (1, 0, 0)$ ó $\mathbf{n} = (-1, 0, 0)$, respectivamente.

La caja correspondiente a la figura 3.15 inciso (iii), está definida por los puntos $(x_{\min}+r, y_{\min}, z_{\min}+r)$ y $(x_{\max}-r, y_{\max}, z_{\max}-r)$. Si $\mathbf{p}_1 = (x_1, y_1)$ se encuentra dentro de esta caja el algoritmo termina regresando *FALSO*.

Si $y < y_{\min}$ ó $y > y_{\max}$, se calcula el punto de intersección de la línea $\overline{\mathbf{p}_0\mathbf{p}_1}$ con el plano $y = y_{\min}$ ó $y = y_{\max}$, según corresponda. Si el punto calculado es $\mathbf{p}' = (x', y', z')$, verificamos si $x_{\min}+r \leq x' \leq x_{\max}-r$ & $z_{\min}+r \leq z' \leq z_{\max}-r$, en tal caso el algoritmo termina, devuelve *VERDADERO* con el punto $\mathbf{p}'$ y el vector unitario $\mathbf{n} = (0, 1, 0)$ ó $\mathbf{n} = (0, -1, 0)$, respectivamente.

Si tuviésemos una caja unitaria, su volumen sería igual a 1 (100 %). El volumen verificado en el paso 1(i) es $v_{11} = l(l-2r)^2$. En el paso 1(ii) y 1(iii) el volumen verificado es: $v_{12} = 2r(l-2r)^2$. Si l=1.0 y r=0.05, v_{11}=0.81 y v_{12}=0.081.

El paso 1(i) verifica el 81 % del volumen, y los pasos 1(ii) y 1(iii) verifican cada uno el 8.1 %; en total, se verifica el 97.2 %. En este estado del algoritmo se tiene el punto recortado $\mathbf{p} = (x, y, z)$ contra la caja que envuelve al contenedor. En el segundo paso vamos a verificar si dicho punto se encuentra en alguna de las regiones sombreadas de la figura 3.16.

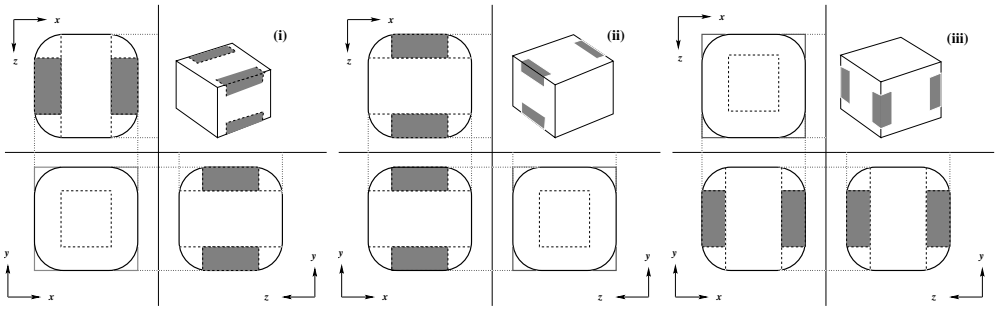


Figura 3.16: Diagrama de los tres casos del paso 2.

Paso 2. Consideremos el caso de la figura 3.16 inciso (i). Un punto $\mathbf{p} = (x, y, z)$ se encuentra dentro de la región sombreada si $z_{\min} + r \leq z \leq z_{\max} - r$ y además cumple alguno de los siguientes casos:

Caso (a): Si $x < x_{\min} + r$ & $y < y_{\min} + r$.
Caso (b): Si $x < x_{\min} + r$ & $y > y_{\max} - r$.
Caso (c): Si $x > x_{\max} - r$ & $y < y_{\min} + r$.
Caso (d): Si $x > x_{\max} - r$ & $y > y_{\max} - r$.

En cada uno de estos casos vamos a recortar la línea $\overline{\mathbf{p}_0\mathbf{p}_1}$ con respecto a un cilindro con eje paralelo al eje Z. Por ejemplo, para el caso (d) ilustrado en la figura 3.17, recortamos respecto al cilindro cuyo eje es la recta $x = x_{\max} - r$ & $y = y_{\max} - r$. Para calcular el punto de recorte podemos utilizar la función *intersecaRayoCirculo()* para encontrar la intersección, sobre el plano XY, de la proyección en dos dimensiones de la línea $\overline{\mathbf{p}_0\mathbf{p}_1}$ y el círculo de radio r y centro en $(x_{\max}-r, y_{\max}-r)$. Con ello obtenemos las

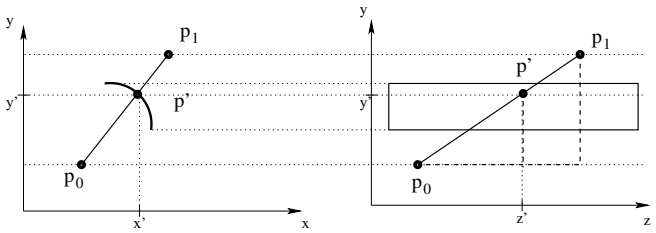


Figura 3.17: Recorte respecto a un cilindro en el paso 2(i), caso (d).

coordenadas (x', y') del punto de intersección $\mathbf{p}'$ y sólo queda calcular la coordenada z' a partir de la semejanza de los triángulos mostrados en línea punteada en el diagrama derecho de la figura 3.17, mediante la siguiente fórmula:

$$z' = p_{0z} + (y' - p_{0y})\frac{p_{1z} - p_{0z}}{p_{1y} - p_{0y}}$$

Finalmente verificamos si el punto recortado $\mathbf{p}'$ se encuentra sobre la caja redondeada, es decir, si $z_{\min} + r \leq z' \leq z_{\max} - r$, el algoritmo termina devolviendo *VERADERO* junto con el punto $\mathbf{p}'$ y el vector unitario normal a la superficie del cilindro $\mathbf{n} = (n_x, n_y, n_z)$ donde (n_x, n_y) se calcula en la función *intersecaRayoCirculo()* y $n_z = 0$. Para los casos (a), (b) y (c), el cálculo del punto de intersección es similar.

Ahora bien, para la situación de la figura 3.16 inciso (ii), recortaremos la línea $\overline{\mathbf{p}_0\mathbf{p}_1}$ contra un cilindro con eje paralelo al eje X, si $x_{\min} + r \leq x_1 \leq x_{\max} - r$ y además cumple con:

Caso (a): Si $y_1 < y_{\min} + r$ & $z_1 < z_{\min} + r$.
Caso (b): Si $y_1 < y_{\min} + r$ & $z_1 > z_{\max} - r$.
Caso (c): Si $y_1 > y_{\max} - r$ & $z_1 < z_{\min} + r$.
Caso (d): Si $y_1 > y_{\max} - r$ & $z_1 > z_{\max} - r$.

Para resolver estos casos, adoptamos la misma estrategia descrita anteriormente, utilizando la función *intersecaRayoCirculo()*, pero utilizando la proyección de la línea $\overline{\mathbf{p}_0\mathbf{p}_1}$ sobre el plano YZ.

Análogamente, para la figura 3.16 inciso (iii), la línea $\overline{\mathbf{p}_0\mathbf{p}_1}$ será recortada respecto a un cilindro con eje paralelo al eje Y, si $y_{\min} + r \leq y_1 \leq y_{\max} - r$ y tendremos casos similares:

Caso (a): Si $x_1 < x_{\min} + r$ & $z_1 < z_{\min} + r$.
Caso (b): Si $x_1 < x_{\min} + r$ & $z_1 > z_{\max} - r$.
Caso (c): Si $x_1 > x_{\max} - r$ & $z_1 < z_{\min} + r$.
Caso (d): Si $x_1 > x_{\max} - r$ & $z_1 > z_{\max} - r$.

Que se resolverán proyectando ahora sobre el plano XZ.

Paso 3. En este punto del algoritmo sólo falta checar la posición del punto recortado $\mathbf{p}'$ para averiguar en cuál de las ocho esquinas, que no se han verificado aún, se encuentra su posible intersección.

Los ocho casos posibles son:

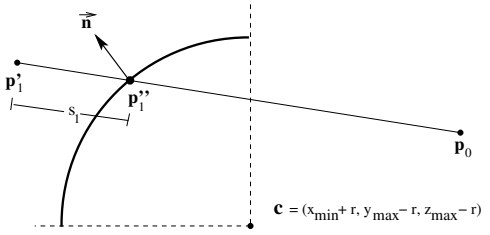


Figura 3.18: Diagrama del caso (d). Se muestra la vista de la sección de esfera y los segmentos de línea tridimensionales, proyectados sobre el plano definido por los puntos $\mathbf{p}_0$, $\mathbf{p}'_1$ y $\mathbf{c}$.

Caso (a): Si $x' < x_{\min} + r$, $y' < y_{\min} + r$ & $z' < z_{\min} + r$.
Caso (b): Si $x' < x_{\min} + r$, $y' < y_{\min} + r$ & $z' > z_{\max} - r$.
Caso (c): Si $x' < x_{\min} + r$, $y' < y_{\max} - r$ & $z' < z_{\min} + r$.
Caso (d): Si $x' < x_{\min} + r$, $y' < y_{\max} - r$ & $z' > z_{\max} - r$.
Caso (e): Si $x' < x_{\max} - r$, $y' < y_{\min} + r$ & $z' < z_{\min} + r$.
Caso (f): Si $x' < x_{\max} - r$, $y' < y_{\min} + r$ & $z' > z_{\max} - r$.
Caso (g): Si $x' < x_{\max} - r$, $y' < y_{\max} - r$ & $z' < z_{\min} + r$.
Caso (h): Si $x' < x_{\max} - r$, $y' < y_{\max} - r$ & $z' > z_{\max} - r$.

Para calcular la intersección de la línea $\overline{\mathbf{p}_0\mathbf{p}'_1}$ con el octavo de esfera correspondiente, debemos recordar que el punto $\mathbf{p}_0$ está dentro de la caja redondeada, pero no necesariamente dentro de la zona del octavo de esfera.

El punto de intersección $\mathbf{p}''_1$ será $\mathbf{p}''_1 = \mathbf{p}'_1 + s\hat{u}$, donde s es una de las soluciones s_1, s_2 de la ecuación $||\mathbf{p}'_1 + s\mathbf{u} - \mathbf{c}|| = r$ con $\mathbf{u} = \frac{\mathbf{p}_0 - \mathbf{p}'_1}{||\mathbf{p}_0 - \mathbf{p}'_1||}$ y $\mathbf{c} = (x_{\min} + r, y_{\max} - r, z_{\min} + r)$.

Esta ecuación la podemos escribir (ver [40], pp. 177-178) en la forma:
$s^2(\mathbf{u} \cdot \mathbf{u}) + 2s(\mathbf{m} \cdot \mathbf{u}) + \mathbf{m} \cdot \mathbf{m} - r^2 = 0$, donde $\mathbf{m} = \mathbf{p}'_1 - \mathbf{c}$. Se calcula: $d = (\mathbf{m} \cdot \mathbf{u})^2 - \mathbf{m} \cdot \mathbf{m} - r^2$. Si d es negativo, se concluye que no existen soluciones reales para la ecuación, algo que no sucede dado que $\mathbf{p}'_1$ está en alguna de las esquinas. Si d es positivo, las soluciones serán:

$$s_1 = -\mathbf{m} \cdot \mathbf{u} - \sqrt{d} \quad \& \quad s_2 = -\mathbf{m} \cdot \mathbf{u} + \sqrt{d}$$

Si una de las soluciones es negativa, $\mathbf{p}'_1$ está dentro de la caja redondeada, entonces el algoritmo termina regresando el valor de *FALSO*. Si las dos soluciones son positivas, el punto $\mathbf{p}'_1$ está fuera del octavo de esfera de la esquina redondeada, y se escogerá la solución más cercana al punto $\mathbf{p}'_1$, esto es, la menor de ellas[4], como se muestra para el caso (d) en la figura 3.18. El algoritmo regresará *VERDADERO* y se debe calcular el vector unitario $\mathbf{n} = \frac{\mathbf{p}''_1 - \mathbf{c}}{||\mathbf{p}''_1 - \mathbf{c}||}$. El algoritmo en pseudocódigo quedará como en las figuras 3.19 y 3.20. Utiliza las funciones dadas en las figuras 3.21 y 3.22.

[4]Observemos que cuando $d > 0$, se tiene $s_1 \leq s_2$, por lo que $s = s_1$.

Algoritmo *dentroCajaRedondeada3D*
ENTRADA: Los puntos $\mathbf{p}_0 = (x_0, y_0, z_0)$ & $\mathbf{p}_1 = (x_1, y_1, z_1)$ del segmento $\overline{\mathbf{p}_0\mathbf{p}_1}$,
la esquina inferior izquierda $\mathbf{A} = (x_{\min}, y_{\min}, z_{\min})$, la esquina superior derecha $\mathbf{B} = (x_{\max}, y_{\max}, z_{\max})$
de la caja redondeada y el radio r de la esquina.
SALIDA: Un valor booleano que indica si hubo contacto, y si lo hubo,
regresa el punto de contacto $\mathbf{p}' = (x', y', z')$ del segmento $\overline{\mathbf{p}_0\mathbf{p}_1}$ con la caja redondeada
y el vector normal $\mathbf{n}$ de la caja en el punto $\mathbf{p}'$.
begin
codigo ← *codigo3D* ($\mathbf{p}_1$, $(x_{\min} + r, y_{\min} + r, z_{\min})$, $(x_{\max} - r, y_{\max} - r, z_{\max})$) // PASO 1(i)
if (!codigo) **return** FALSO
else if (codigo & FRENTE)
$\mathbf{p}'$ ← intersecaPlano($\mathbf{p}_0$,$\mathbf{p}_1$,$(0, 0, z_{\max})$, $(0, 0, 1)$), $\mathbf{p}_1 \leftarrow \mathbf{p}'$
if ($x_{\min} + r \leq x' \leq x_{\max} - r$ & $y_{\min} + r \leq y' \leq y_{\max} - r$)
$\mathbf{n} \leftarrow (0, 0, 1)$
return ($\mathbf{p}'$, $\mathbf{n}$, VERDADERO)
endif
else if (codigo & ATRAS)
$\mathbf{p}'$ ← intersecaPlano($\mathbf{p}_0$,$\mathbf{p}_1$,$(0, 0, z_{\min})$, $(0, 0, -1)$), $\mathbf{p}_1 \leftarrow \mathbf{p}'$
if ($x_{\min} + r \leq x' \leq x_{\max} - r$ & $y_{\min} + r \leq y' \leq y_{\max} - r$)
$\mathbf{n} \leftarrow (0, 0, -1)$
return ($\mathbf{p}'$, $\mathbf{n}$, VERDADERO)
endif
endif
codigo ← *codigo3D* ($\mathbf{p}_1$, $(x_{\min}, y_{\min} + r, z_{\min} + r)$, $(x_{\max}, y_{\max} - r, z_{\max} + r)$) // PASO 1(ii)
if (!codigo) **return** FALSO
else if (codigo & DERECHA)
$\mathbf{p}'$ ← intersecaPlano($\mathbf{p}_0$,$\mathbf{p}_1$,$(x_{\max}, 0, 0)$, $(1, 0, 0)$), $\mathbf{p}_1 \leftarrow \mathbf{p}'$
if ($y_{\min} + r \leq y' \leq y_{\max} - r$ & $z_{\min} + r \leq z' \leq z_{\max} - r$)
$\mathbf{n} \leftarrow (1, 0, 0)$
return ($\mathbf{p}'$, $\mathbf{n}$, VERDADERO)
endif
else if (codigo & IZQUIERDA)
$\mathbf{p}'$ ← intersecaPlano($\mathbf{p}_0$,$\mathbf{p}_1$,$(x_{\min}, 0, 0)$, $(-1, 0, 0)$), $\mathbf{p}_1 \leftarrow \mathbf{p}'$
if ($y_{\min} + r \leq y' \leq y_{\max} - r$ & $z_{\min} + r \leq z' \leq z_{\max} - r$)
$\mathbf{n} \leftarrow (-1, 0, 0)$
return ($\mathbf{p}'$, $\mathbf{n}$, VERDADERO)
endif
endif
codigo ← *codigo3D* ($\mathbf{p}_1$, $(x_{\min} + r, y_{\min}, z_{\min} + r)$, $(x_{\max} - r, y_{\max}, z_{\max} + r)$) // PASO 1(iii)
if (!codigo) **return** FALSO
else if (codigo & TOP)
$\mathbf{p}'$ ← intersecaPlano($\mathbf{p}_0$,$\mathbf{p}_1$,$(0, y_{\max}, 0)$, $(0, 1, 0)$), $\mathbf{p}_1 \leftarrow \mathbf{p}'$
if ($x_{\min} + r \leq x' \leq x_{\max} - r$ & $z_{\min} + r \leq z' \leq z_{\max} - r$)
$\mathbf{n} \leftarrow (0, 1, 0)$, **return** ($\mathbf{p}'$, $\mathbf{n}$, VERDADERO)
endif
else if (codigo & BOTTOM)
$\mathbf{p}'$ ← intersecaPlano($\mathbf{p}_0$,$\mathbf{p}_1$,$(0, y_{\min}, 0)$, $(0, -1, 0)$), $\mathbf{p}_1 \leftarrow \mathbf{p}'$
if ($x_{\min} + r \leq x' \leq x_{\max} - r$ & $z_{\min} + r \leq z' \leq z_{\max} - r$)
$\mathbf{n} \leftarrow (0, -1, 0)$, **return** ($\mathbf{p}'$, $\mathbf{n}$, VERDADERO)
endif
...

Figura 3.19: Algoritmo *dentroCajaRedondeada3D* (paso 1).

Algoritmo *dentroCajaRedondeada3D* (continúa)
...
  endif
  codigo ← *codigo3D* ($\mathbf{p}_1$, $(x_{\min}+r, y_{\min}+r, z_{\min}+r)$, $(x_{\max}-r, y_{\max}-r, z_{\max}-r)$)
  $c_x \leftarrow x_{\min}+r$, $c_y \leftarrow y_{\min}+r$, $c_z \leftarrow z_{\min}+r$ // esquina inferior trasera derecha
  if (codigo & ARRIBA)
    $c_y \leftarrow c_y + y_{\max} - y_{\min} - 2r$ // mover a esquina superior
  endif
  if (codigo & DERECHA)
    $c_x \leftarrow c_x + x_{\max} - x_{\min} - 2r$ // mover a esquina derecha
  endif
  if (codigo & FRENTE)
    $c_z \leftarrow c_z + z_{\max} - z_{\min} - 2r$ // mover a esquina frontal
  endif
  if (!(codigo & FRENTE) & !(codigo & ATRAS)) // PASO 2 (i)
    $\mathbf{u} \leftarrow \frac{(p_{0x},p_{0y})-(p_{1x},p_{1y})}{||(p_{0x},p_{0y})-(p_{1x},p_{1y})||}$
    resultado = *intersecaRayoCirculo* ((p_{1x}, p_{1y}), (x', y'), r, (c_x, c_y), $\mathbf{u}$, (n_x, n_y))
    if (!resultado) **return** FALSE
    $z' = p_{0z} + (y' - p_{0y})\frac{p_{1z}-p_{0z}}{p_{1y}-p_{0y}}$
    if $(z_{\min}+r \leq z' \leq z_{\max}-r)$
      $n_z \leftarrow 0$, **return** ($\mathbf{p}'$, $\mathbf{n}$, VERADERO)
    endif
  endif
  if (!(codigo & IZQUIERDA) & !(codigo & DERECHA)) // PASO 2 (ii)
    $\mathbf{u} \leftarrow \frac{(p_{0y},p_{0z})-(p_{1y},p_{1z})}{||(p_{0y},p_{0z})-(p_{1y},p_{1z})||}$
    resultado = *intersecaRayoCirculo* ((p_{1y}, p_{1z}), (y', z'), r, (c_y, c_z), $\mathbf{u}$, (n_y, n_z))
    if (!resultado) **return** FALSE
    $x' = p_{0x} + (y' - p_{0y})\frac{p_{1x}-p_{0x}}{p_{1y}-p_{0y}}$
    if $(x_{\min}+r \leq x' \leq x_{\max}-r)$
      $n_x \leftarrow 0$, **return** ($\mathbf{p}'$, $\mathbf{n}$, VERADERO)
    endif
  endif
  if (!(codigo & ARRIBA) & !(codigo & ABAJO)) // PASO 2 (iii)
    $\mathbf{u} \leftarrow \frac{(p_{0z},p_{0x})-(p_{1z},p_{1x})}{||(p_{0z},p_{0x})-(p_{1z},p_{1x})||}$
    resultado = *intersecaRayoCirculo* ((p_{1z}, p_{1x}), (z', x'), r, (c_z, c_x), $\mathbf{u}$, (n_z, n_x))
    if (!resultado) **return** FALSE
    $y' = p_{0y} + (x' - p_{0x})\frac{p_{1y}-p_{0y}}{p_{1x}-p_{0x}}$
    if $(y_{\min}+r \leq y' \leq y_{\max}-r)$
      $n_y \leftarrow 0$, **return** ($\mathbf{p}'$, $\mathbf{n}$, VERADERO)
    endif
  endif
  // PASO 3
  $\mathbf{u} \leftarrow \frac{(p_{0x},p_{0y},p_{0z})-(p_{1x},p_{1y},p_{1z})}{||(p_{0x},p_{0y},p_{0z})-(p_{1x},p_{1y},p_{1z})||}$
  return *intersecaRayoEsfera* ((p_{1x}, p_{1y}, p_{1z}), (x', y', z'), r, $(c_x, c_y.c_z)$, $\mathbf{u}$, (n_x, n_y, n_z))
end

Figura 3.20: Algoritmo *dentroCajaRedondeada3D* (pasos 2 y 3).

Función *intersecaRayoEsfera*
ENTRADA: Un rayo definido por el punto $\mathbf{p}$ y el vector unitario $\mathbf{u}$, el círculo definido por el punto central $\mathbf{p}_c$ y el radio r.
SALIDA: Un valor booleano que indica si hubo contacto, y si lo hubo, devuelve el primer punto de contacto $\mathbf{p}'$ del rayo con el círculo y el vector normal $\mathbf{n}$ en dicho punto.

```
begin
    m ← p − p_c
    b ← m · u
    c ← m · m − r^2
    d ← b^2 − c
    if (d < 0)
        print "ERROR: Esto no puede ser!"
        exit
    else
        s ← −b − √d
        if (s < 0)
            return FALSO
        endif
        p' ← p + su
        n ← (p' − p_c) / ||p' − p_c||
    endif
    return (p', n, VERDADERO)
end
```

Función *codigo3D*
ENTRADA: El punto $\mathbf{p} = (x, y, z)$, la esquina inferior izquierda $\mathbf{A} = (x_{\min}, y_{\min}, z_{\min})$, la esquina superior derecha $\mathbf{B} = (x_{\max}, y_{\max}, z_{\max})$ de una caja.
SALIDA: El código de region de seis bits.

```
    ARRIBA = 0x01, ABAJO = 0x02,
    DERECHA = 0x04, IZQUIERDA = 0x08,
    FRENTE = 0x10, ATRAS = 0x20
begin
    codigo ← 0
    if (y > y_max)
        codigo |= ARRIBA
    else if (y > y_min)
        codigo |= ABAJO
    endif
    if (x > x_max)
        codigo |= DERECHA
    else if (x > x_min)
        codigo |= IZQUIERDA
    endif
    if (z > z_max)
        codigo |= FRENTE
    else if (z > z_min)
        codigo |= ATRAS
    endif
    return codigo
end
```

Figura 3.21: Funciones *intersecaRayoEsfera* y *codigo3D*.

Función *intersecaRayoCirculo*
ENTRADA: Un rayo definido por el punto $\mathbf{p}$
y el vector unitario $\mathbf{u}$, el círculo definido por
el punto central $\mathbf{p}_c$ y el radio r.
SALIDA: Un valor booleano que indica si hubo
contacto, y si lo hubo, devuelve el primer punto
de contacto $\mathbf{p}'$ del rayo con el círculo
y el vector normal $\mathbf{n}$ en dicho punto.

```
begin
    m ← p − p_c
    b ← m · u
    c ← m · m − r^2
    d ← b^2 − c
    if (d < 0)
        print "ERROR: Esto no puede ser!"
        exit
    else
        s ← −b − √d
        if (s < 0)
            return FALSO
        endif
        p' ← p + su
        n ← (p' − p_c) / ||p' − p_c||
    endif
    return (p', n, VERDADERO)
end
```

Función *intersecaPlano*
ENTRADA: Un segmento definido por los puntos
$\mathbf{p}_0$, $\mathbf{p}_1$, y un plano definido por
un punto $\mathbf{p}$ y un vector unitario $\mathbf{n}$.
SALIDA: Un valor booleano que indica si hubo
contacto, y si lo hubo, devuelve el punto
de contacto $\mathbf{p}'$ del segmento con el plano.

```
begin
    u ← p_1 − p_0
    t ← (p · n − p_0 · n)/(u · n)
    if (0 ≤ t ≤ 1)
        p' ← p_0 + tu
        return (p', VERDADERO)
    endif
    return FALSO
end
```

Figura 3.22: Funciones *intersecaRayoCirculo* e *intersecaPlano*.

3.3. Modelo de la colisión

Una vez que se ha detectado que ocurrió una colisión entre la trayectoria $\overline{\mathbf{ab}}$ de la partícula y el contenedor, y se ha calculado el punto de contacto $\mathbf{p}'$ junto con la normal $\mathbf{n}$ de la superficie, es necesario manejar la reacción que tendrá la partícula ante la colisión. Usualmente la reacción es hacer que la partícula *rebote* contra la superficie, como si fuese una pelota que choca contra una pared. Para que ocurra el rebote, uno u otro de los objetos involucrados debe ser elástico, es decir, comportarse como si fuera un resorte. Es conveniente considerar que es la superficie la que se comporta como un resorte con un elevado coeficiente de restitución k y que cuenta también con una constante de amortiguamiento[5] b. De esta manera, lo indicado es utilizar el modelo matemático del resorte amortiguado, el cual se describe a continuación.

Resorte amortiguado

Se tiene una masa m sujeta a un resorte ideal sin masa que posee una constante de restitución k y sujeto a un efecto de amortiguamiento cuantificado por la constante b. Por la ley de Hooke, cuando la masa se desplaza una distancia x, el resorte ejerce sobre ésta una fuerza de restitución de magnitud $F_{\text{restitucion}} = kx$ y de dirección opuesta al desplazamiento. Así mismo, se ha comprobado experimentalmente que la fuerza de amortiguamiento es de magnitud proporcional a la velocidad[6] de la masa $v = \dot{x}$ y en dirección opuesta al movimiento: $F_{\text{amortiguamiento}} = b\dot{x}$. Finalmente, el movimiento de la masa viene dado por la ley de Newton: $F_{\text{total}} = ma = m\ddot{x}$. Para nuestro resorte amortiguado, la fuerza total es: $F_{\text{total}} = -F_{\text{restitucion}} - F_{\text{amortiguamiento}}$. Esto nos da la siguiente ecuación diferencial:

$$m\ddot{x} + b\dot{x} + kx = 0 \tag{3.5}$$

donde x es el desplazamiento de la masa m, b es la constante de viscosidad dada en kg/s y k es la constante de restitución dada en kg/s^2.

Resolviendo la ecuación (3.5) se tienen tres casos, según el valor del determinante $b^2 - 4mk$:

1. Si $b^2 - 4mk > 0$: se trata de un movimiento sobreamortiguado.
   La solución es:
   $$x(t) = c_1 e^{r_1 t} + c_2 e^{r_2 t} \tag{3.6}$$
   donde
   $$\begin{aligned} r_1 &= \frac{-b - \sqrt{b^2 - 4mk}}{2m} \\ r_2 &= \frac{-b + \sqrt{b^2 - 4mk}}{2m} \end{aligned}$$

[5] En [4], cap. 3.2, se utiliza esta misma aproximación pero a la constante de amortiguamiento se le llama disipación viscosa de la superficie. En el presente trabajo se prefiere la primera denominación para mantenernos en el contexto del modelo del resorte amortiguado.

[6] Utilizamos la notación: $\dot{x} = \frac{d}{dt}x$ y $\ddot{x} = \frac{d^2}{dt^2}x$.

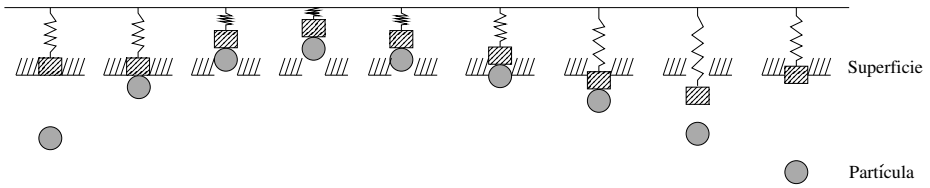


Figura 3.23: Secuencia de diagramas que ilustra el modelo de colisión de una partícula contra una superficie elástica, la cual actúa como un resorte amortiguado.

2. Si $b^2 - 4mk = 0$: se trata de un movimiento con amortiguamiento crítico. La solución es:
$$x(t) = e^{-rt}(c_1 + c_2 t) \tag{3.7}$$
donde
$$r = \frac{b}{2m}$$

3. Si $b^2 - 4mk < 0$: se trata de un movimiento subamortiguado. La solución es:
$$x(t) = e^{-rt}(c_1 \cos(\omega t) + c_2 \operatorname{sen}(\omega t)) \tag{3.8}$$
donde
$$\begin{aligned} r &= \frac{b}{2m} \\ \omega &= \frac{\sqrt{4mk - b^2}}{2m} \end{aligned}$$

en todos los casos c_1 y c_2 son constantes que dependen de las condiciones iniciales del movimiento.

Como se menciona en [4], en los casos sobreamortiguado y críticamente amortiguado la partícula eventualmente quedará adherida a la superficie, lo cual no es lo que se quiere simular. Estamos buscando un movimiento subamortiguado, por lo tanto es necesario que la constante b cumpla la expresión $b^2 - 4mk < 0$, esto nos lleva a elegir $b < 2\sqrt{mk}$. Recordemos que b se define positivo, entonces, lo anterior escrito de otra manera queda:
$$b = 2\xi\sqrt{mk} \tag{3.9}$$
con $0 \leq \xi < 1$. Cuando $\xi = 0$ se tiene una colisión perfectamente elástica, mientras que cuando tiende a 1 el choque resulta cada vez más amortiguado.

Colisión entre partícula y superficie

Ahora nos encontramos en condiciones de resolver el problema de la partícula que choca contra la superficie elástica con velocidad $v_0 = y'(t_0) \neq 0$ en el momento $t_0 = 0$

y ubicada en la posición $x_0 = x(t_0) = 0$ (ver figura 3.23). Hasta aquí el modelo coincide con el de [4]. La diferencia radica en la manera de resolver la ecuación diferencial para este problema. En la versión previa del simulador se adopta una aproximación numérica para calcular la fuerza que la superficie ejerce sobre la partícula. Entonces el rebote se resuelve en el siguiente paso de la simulación. Para pasos de integración mayores, como los utilizados en nuestro trabajo, se acumulan los errores numéricos y la partícula gana velocidad en lugar de perderla, ocasionando problemas de inestabilidad.

Para nuestra aplicación, utilizamos la solución exacta de la ecuación diferencial para garantizar que la partícula no gane velocidad. Se calcula exactamente la velocidad con la que la partícula abandona la superficie, aunque no la hacemos rebotar en el momento exacto. De hecho, de acuerdo al desarrollo dado más adelante, si el rebote comienza en el tiempo $t_0 = 0$, el momento exacto en que la partícula abandona la superficie es $t_f = \pi/\omega = 2\pi m/\sqrt{4mk - b^2}$, sin embargo, la devolvemos antes, en el momento t_0. Pero esto no genera inestabilidades numéricas porque la partícula nunca gana velocidad y, además, para constantes k muy grandes, la diferencia de tiempo $\Delta t = t_f - t_0$ es muy pequeña.

Procedamos ahora a encontrar la solución exacta a nuestro problema. Aplicando la condición inicial $x_0 = 0$ a la ecuación (3.8), se tiene:

$$\begin{aligned} x_0 &= e^{-rt_0}(c_1 \cos(\omega t_0) + c_2 \operatorname{sen}(\omega t_0)) \\ &= e^0(c_1 \cdot 1 + c_2 \cdot 0) = c_1 = 0 \end{aligned}$$

Entonces,

$$x(t) = c_2 e^{-rt} \operatorname{sen}(\omega t)$$

Luego, la velocidad es:

$$v(t) = x'(t) = c_2 e^{-rt}(\omega \cos(\omega t) - r \operatorname{sen}(\omega t))$$

Ahora bien, como condición inicial tenemos un valor de $v_0 \neq 0$:

$$\begin{aligned} v_0 &= c_2 e^{-rt_0}(\omega \cos(\omega t_0) - r \operatorname{sen}(\omega t_0)) \\ &= c_2 e^0(\omega \cdot 1 - r \cdot 0) = c_2 \omega \\ \Rightarrow c_2 &= \frac{v_0}{\omega} \end{aligned}$$

Por lo tanto, las ecuaciones de movimiento de la partícula son:

$$x(t) = \frac{v_0}{\omega} e^{-rt} \operatorname{sen}(\omega t) \tag{3.10}$$

$$v(t) = \frac{v_0}{\omega} e^{-rt}(\omega \cos(\omega t) - r \operatorname{sen}(\omega t)) \tag{3.11}$$

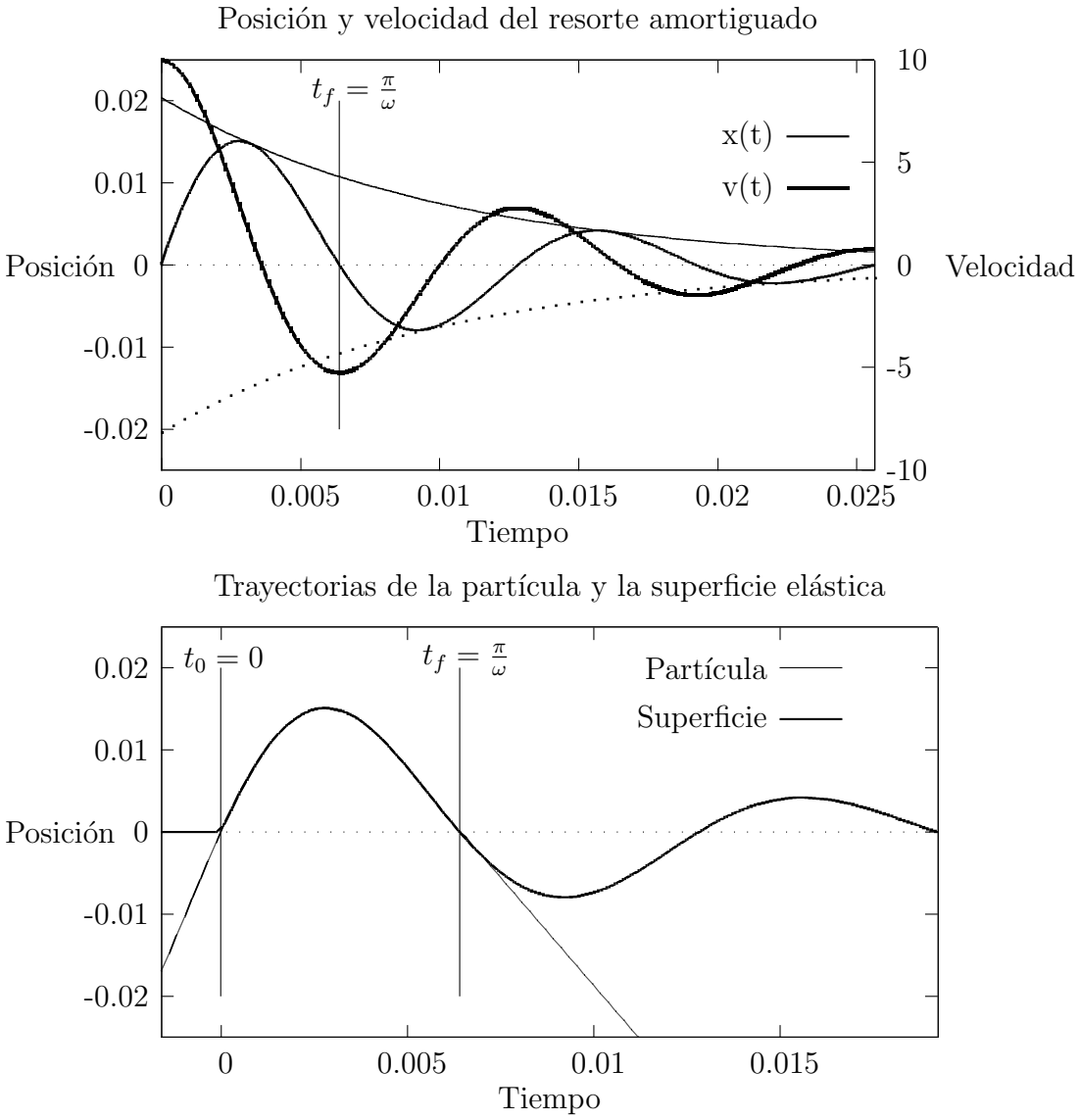


Figura 3.24: Gráfica de $x(t)$ y v(t) de la superficie elástica (superior) y trayectoria de la partícula (inferior). Se indica el tiempo t_f en que termina la colisión.

En la figura 3.24, parte superior, se muestran las gráficas que corresponden a la posición $x(t)$ y la velocidad $v(t) = x'(t)$ de la superficie elástica actuando como un resorte amortiguado. En la parte inferior se muestra la posición de la superficie elástica en línea gruesa y la trayectoria de la partícula en línea delgada. También se indica el primer momento t_f en que el resorte regresa a la posición $x = 0$, que corresponde al momento en que termina la colisión y la partícula abandona la superficie para continuar con su trayectoria rectilínea. El tiempo t_f es tal que $\omega t_f = \pi$. Luego el valor buscado es $t_f = \pi/\omega$.

Entonces la velocidad con la que la partícula abandona la superficie es:

$$\begin{aligned} v(t_f) &= \frac{v_0}{\omega} e^{-rt_f}(\omega\cos(\omega t_f) - r\,\mathrm{sen}(\omega t_f)) \\ &= \frac{v_0}{\omega} e^{-\frac{r\pi}{\omega}}(\omega\cos(\pi) - r\,\mathrm{sen}(\pi)) \\ &= \frac{v_0}{\omega} e^{-\frac{r\pi}{\omega}}(\omega(-1) - r\cdot 0) = -\omega\frac{v_0}{\omega} e^{-\frac{r\pi}{\omega}} \\ &= -v_0 e^{-\frac{r\pi}{\omega}} \end{aligned}$$

Es decir, la velocidad de la partícula v_0 al comenzar la colisión, ve reducida su magnitud en un factor $e^{-\frac{r\pi}{\omega}}$ y su dirección es opuesta al movimiento inicial, esto es, rebota con una velocidad menor o igual a la que llevaba inicialmente.

El factor de reducción de la velocidad depende de la constante de amortiguamiento de la superficie b. Recordemos que $r = \frac{b}{2m}$, entonces $e^{-\frac{r\pi}{\omega}} = e^{-\frac{b\pi}{2m\omega}}$. Si no existe amortiguamiento, $b = 0$, entonces $v(t_f) = -v_0 e^{-\frac{b\pi}{2m\omega}} = -v_0 e^0 = -v_0$. Esto significa que la partícula rebota con la misma magnitud de velocidad con que llegó a la superficie.

El análisis anterior ocurre en una dimensión, de hecho en la dirección $\mathbf{n}$ normal a la superficie. Para resolver la colisión en dos y tres dimensiones, descomponemos la velocidad $\mathbf{v}_0$ de la partícula en los componentes normal $\mathbf{v}_\perp$ y tangencial $\mathbf{v}_\parallel$ a la superficie en el punto de contacto $\mathbf{p}'$. Es decir $\mathbf{v}_0 = \mathbf{v}_\parallel + \mathbf{v}_\perp$, tal que $\mathbf{v}_\parallel \cdot \mathbf{v}_\perp = 0$ y $\mathbf{v}_\parallel \cdot \mathbf{n} = 0$. De esta manera, la velocidad de la partícula al terminar la colisión es:

$$\mathbf{v}_f = \mathbf{v}_\parallel - e^{-\frac{r\pi}{\omega}}\mathbf{v}_\perp \tag{3.12}$$

donde el factor $e^{-\frac{r\pi}{\omega}}$ es el mismo para todas las partículas de masa m y para todos los puntos de la superficie del contenedor. Su valor se precalcula al inicio de la simulación una sola vez, permitiendo que el manejo de la colisión sea eficiente.

3.4. Creación de la bola de fluido

Para agrupar las N partículas en una bola de fluido hemos adoptado una aproximación dinámica en la cual utilizamos el propio simulador y una técnica sencilla de

control mediante la aplicación de un campo de fuerzas. La idea básica es agregar un campo de fuerzas de atracción central que atraiga las partículas hacia un punto en el centro del contenedor. Como las partículas son desplazadas por el simulador HSP, éstas tienden a agruparse alrededor del punto central, pero sin traslaparse debido a la repulsión de unas sobre otras. Para entender mejor esta idea, podemos imaginarla con un comportamiento semejante a la formación de un planeta en un punto de atracción gravitatoria, en el cual se va acumulando la materia en una forma esferoide.

La formación de la bola de fluido se logró exitosamente mediante la aplicación del siguiente campo de fuerzas central. Definimos el campo vectorial $\mathbf{f}_{\text{central}} : \mathbb{R}^n \longrightarrow \mathbb{R}^n$, con $n \in \{2, 3\}$, de la siguiente manera:

$$\mathbf{f}_{\text{central}}(\mathbf{p}) = \begin{cases} 0; & \text{si} \quad 0 < \|\mathbf{r}\| \leq \epsilon \\ F\frac{\mathbf{r}}{\|\mathbf{r}\|}; & \text{si} \quad \epsilon < \|\mathbf{r}\| \leq R \\ 0; & \text{si} \quad R < \|\mathbf{r}\| \end{cases}$$

donde $\mathbf{r} = \mathbf{p}_{\text{centro}} - \mathbf{p}$ es la dirección desde la partícula ubicada en $\mathbf{p}$ hacia el centro de atracción ubicado en $\mathbf{p}_{\text{centro}}$, ϵ es un valor muy pequeño pero suficiente para evitar una división por cero en el cálculo de la dirección de la fuerza $\frac{\mathbf{r}}{\|\mathbf{r}\|}$, F es la magnitud de la fuerza aplicada y R es el radio máximo de influencia del campo de fuerzas. El campo se muestra en la figura 3.25. Como queremos que todo el fluido sea atraído hacia el centro, hacemos R lo suficientemente grande para que abarque toda la caja. Si quisiéramos atraer sólo una fracción de las partículas, simplemente debemos reducir el radio de influencia R.

No se utilizó la fuerza de atracción de Newton $f = G\frac{m_1 m_2}{r^2}$, donde G es la constante de gravitación universal y r es la posición de la masa m_1 respecto a la masa m_2, o viceversa. Esto fue con el propósito de evitar la singularidad que se forma en el centro ($r = 0$), donde se encontraría la hipotética masa puntual m_2, causante de la atracción. Dicho de otra manera, la partícula que se acerque demasiado al centro de atracción sufriría una aceleración enorme. Eventualmente, se produciría una división por cero al llegar al centro, o bien, saldría disparada por el lado opuesto golpeando las demás partículas. La singularidad se muestra en la figura 3.26. Se grafica la magnitud de la fuerza gravitacional que ejercen dos masas puntuales m_1 y m_2 que se acercan entre sí.

Este método presenta un inconveniente cuando la viscosidad del fluido es muy baja: las partículas continúan orbitando alrededor del centro y chocando entre sí (como si se tratase de estrellas en una galaxia) durante un largo tiempo antes de perder ímpetu y finalmente detenerse. Cuando la viscosidad es mayor no ocurre el problema, las partículas van perdiendo velocidad por sí mismas en un tiempo razonable. Para resover este problema con viscosidades bajas, implementamos un mecanismo de reducción de la velocidad por fricción artificial. Simplemente multiplicamos, en cada iteración, la velocidad de las partículas que se encuentran dentro del radio de influencia R por un factor μ_f menor que 1. En nuestro caso, un valor de 0.001 da buenos

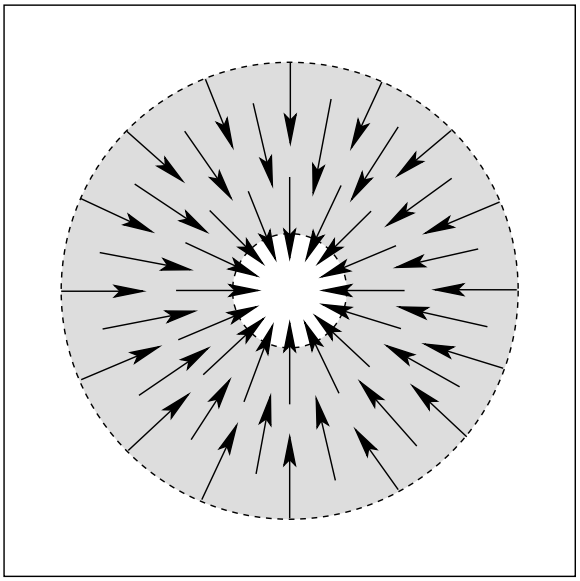


Figura 3.25: Diagrama del campo de fuerza utilizado. El área sombreada tiene asignados vectores de magnitud F.

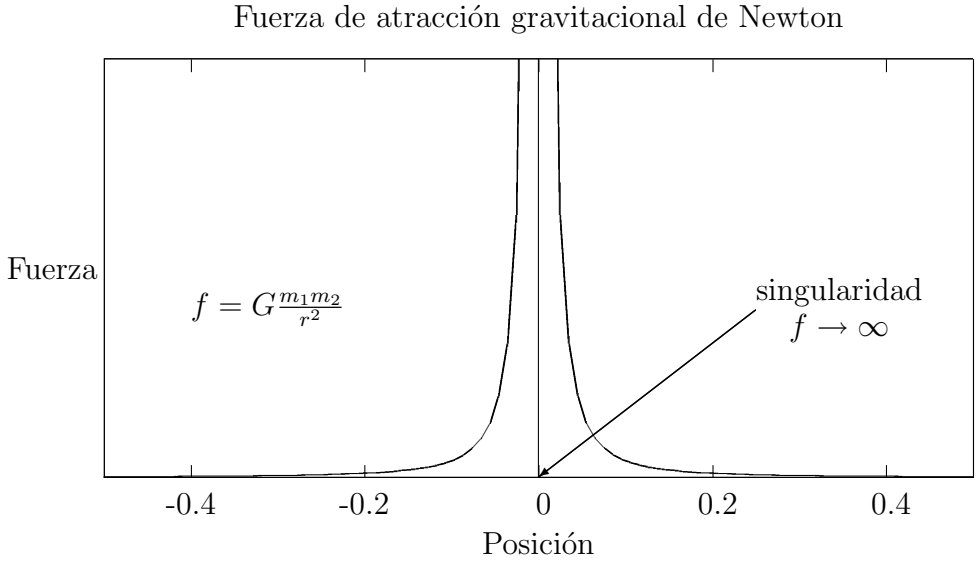


Figura 3.26: Gráfica de la singularidad que ocurre entre dos masas puntuales muy cercanas.

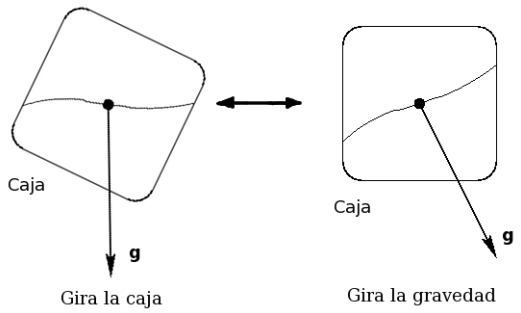


Figura 3.27: La caja gira respecto a la aceleración gravitacional y viceversa.

resultados. Un factor μ_f mayor provoca que no se llegue a formar la bola de fluido y un valor menor aumenta el tiempo en que pierden velocidad las partículas.

Durante la simulación, damos a F un valor constante de 10 Newtons durante los primeros 10 segundos, en los cuales la fuerza de gravedad está desactivada. Después, cuando queremos dejar caer la bola de fluido, activamos la gravedad (ver sección 3.5) y desactivamos la fuerza central, así como la fricción artificial.

3.5. Animación de la gravedad

Se requiere que el contenedor pueda girar sobre un eje alrededor de su centro y que su contenido fluya en su interior en reacción a dicho movimiento. Sin embargo, al girar el contenedor sus superficies, se van a desplazar al tiempo que van girando y van a impulsar las partículas de fluido de una manera distinta al caso en que el contenedor está inmóvil. Hemos descrito en la sección 3.3 el modelo de colisión desarrollado en este proyecto y no está diseñado para manejar el caso en que la superficie del contenedor se mueve y gira.

Con el fin de utilizar nuestro modelo de colisión, hemos resuelto el giro del contenedor desde un punto de vista relativo, como se ilustra en la figura 3.27. Al girar el contenedor en una dirección respecto a un cierto eje, el fluido recibe la aceleración gravitacional girada el mismo ángulo pero en sentido opuesto respecto al mismo eje de rotación. Podemos reproducir la situación, de manera aproximada, considerando que es la aceleración gravitacional la que gira en el sentido opuesto, respecto al contenedor que permanece inmóvil. De esta manera, las partículas fluyen siguiendo la aceleración gravitacional que va girando. Finalmente, al momento de visualizar la escena con el contenedor y las partículas de fluido, la escena completa se gira de manera que la aceleración gravitacional quede apuntando en la dirección correcta. Para el usuario, la aceleración gravitacional parece inmóvil y es el contenedor el que está girando.

Con esta estrategia, se han obtenido resultados físicamente plausibles, al tiempo que se ahorran los cálculos correspondientes al impulso impartido al fluido por las superficies del contenedor al girar.

3.6. Velocidad terminal artificial

Se hicieron varios intentos para efectuar la simulación de manera interactiva en equipos de moderado poder de cómputo. Dentro del contexto del simulador, con el término interacción nos referimos concretamente a que el usuario pueda manipular el giro del contenedor. Adicionalmente, queremos dar al usuario la habilidad de activar y desactivar la aceleración gravitatoria y la fuerza central, de manera que pueda manipular en tiempo real el movimiento del fluido simulado. Esto es independiente de la interactividad en la visualización. Para lograr este nivel de interacción, es necesario que los pasos de integración sean relativamente grandes. Con ello se logra que el simulador reduzca el número de operaciones necesarias para animar un intervalo de tiempo dado.

Sin embargo, al aumentar el tamaño del paso de tiempo, se incrementan los errores numéricos y la simulación se vuelve inestable. Observando los distintos resultados obtenidos, notamos que en los casos estables las partículas no se aceleran demasiado, mientras que en las inestables sí lo hacen. Esto sugiere una posible solución al problema, que se explica a continuación.

Es bien sabido en dinámica de fluidos que un cuerpo desplazándose dentro de un fluido bajo la acción de una aceleración constante no alcanza velocidades arbitrariamente altas debido, principalmente, a la fuerza de fricción o rozamiento del cuerpo con el fluido, la cual se incrementa en proporción a la velocidad del cuerpo. Existe un límite de velocidad que los cuerpos pueden alcanzar dentro de un fluido dado. A esta velocidad se le conoce como *velocidad límite* o *velocidad terminal*[7].

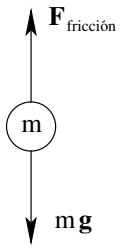


Figura 3.28: La fuerza de fricción iguala la gravitacional.

[7] http://es.wikipedia.org/Velocidad_límite

Tabla 3.2: Información del equipo utilizado.

Nombre del modelo:	MacBook Pro
Identificador del modelo:	MacBookPro5,5
Nombre del procesador:	Intel Core 2 Duo
Velocidad del procesador:	2.53 GHz
Número de procesadores:	1
Número total de núcleos:	2
Caché de nivel 2:	3 MB
Memoria:	4 GB
Velocidad del bus:	1.07 GHz

Algo similar ocurre cuando la simulación se realiza con pasos de tiempo suficientemente pequeños, las partículas no se aceleran demasiado a causa de su interacción con las demás, lo cual limita su velocidad de manera correcta. Sin embargo, cuando se acumulan los errores de integración con pasos de tiempo mayores, las partículas se aceleran cuando no deberían hacerlo. Al agregar artificialmente un límite de velocidad para las partículas, fue posible reducir significativamente las inestabilidades causadas y se logró acelerar la simulación lo suficiente para brindar interactividad al usuario. El valor de este límite artificial de velocidad se estableció empíricamente en 10 m/s, mediante pruebas directas con el simulador. Adicionalmente, se tiene que al establecer un límite de velocidad fijo, no se tienen que realizar cálculos que disminuyan la velocidad de ejecución del simulador.

Dado un vector $\mathbf{v}$, éste se limita a una $\mathbf{v}_{\max}$ dada usando la siguiente expresión:

$$\mathbf{v}' = \mathbf{v}_{\max} \frac{\mathbf{v}}{\|\mathbf{v}\|}$$

3.7. Resultados de la simulación

Se ha puesto a prueba el simulador, tanto en dos como en tres dimensiones, en un equipo comercial portátil con las características listadas en la tabla 3.2. Se implementó el movimiento del fluido descrito al inicio de este capítulo, primero se forma una bola de fluido que posteriormente se deja caer por efecto de la gravedad. Luego se gira la caja sobre el eje Z a un ángulo de 45° y se regresa a su orientación inicial. Se repite este giro pero ahora a un ángulo de 90° y finalmente a 180°. La bola de fluido se forma mediante la aplicación del campo de fuerzas de la sección 3.4, en donde la magnitud de la fuerza F se aplica durante los primeros 10 segundos, lo suficiente para formar la bola, y luego desaparece. Esta fuerza se define en función del tiempo de simulación:

$$F(t) = \begin{cases} 5 \text{ Nw}; & \text{si} \quad t < 10\ s \\ 0 \text{ Nw}; & \text{si} \quad t \geq 10\ s \end{cases}$$

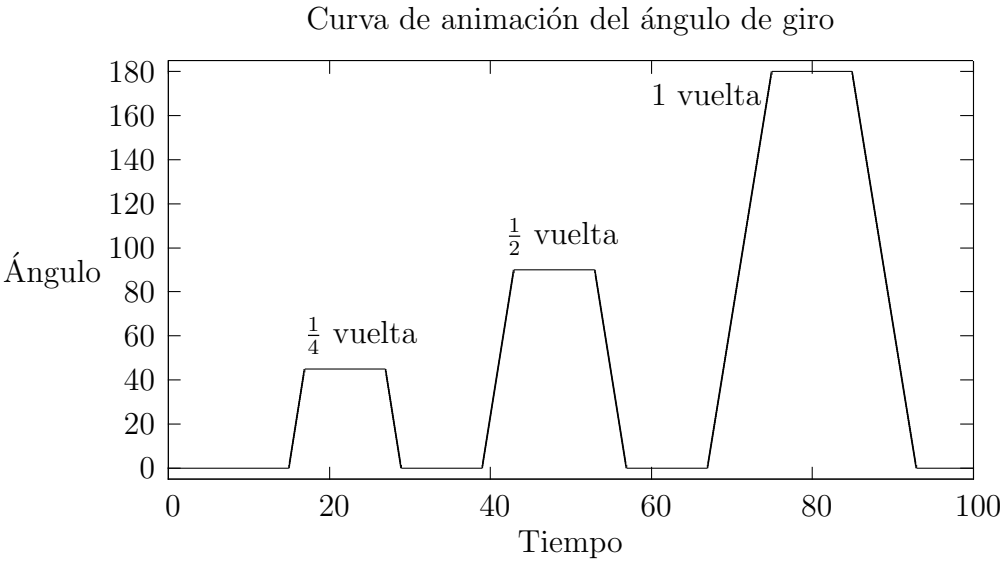


Figura 3.29: Curva de animación del giro de la gravedad.

Al principio la gravedad está desactivada y se activa a los 10 segundos. La magnitud de la gravedad g se define como sigue:

$$g(t) = \begin{cases} 0 \ \frac{m}{s^2}; & \text{si} \quad t < 10 \ s \\ 9.81 \ \frac{m}{s^2}; & \text{si} \quad t \geq 10 \ s \end{cases}$$

La animación del giro se implementó mediante interpolación lineal y se muestra su gráfica en la figura 3.29. Los parámetros de simulación que son comunes a los casos 2D y 3D, se muestran en la tabla 3.3.

Tabla 3.3: Parámetros de simulación comunes a los casos 2D y 3D.

Parámetro	**Variable**	**Valor**
Magnitud de la gravedad	G	9.81
Lado de la caja	L	1
Constante de restitución	K	500000
Masa de cada partícula	m	2
Cantidad de amortiguamiento	ξ	0.2
Constante de amortiguamiento	b	$2\xi\sqrt{mK}$
Coeficiente de viscosidad	$\mu_{\text{viscosidad}}$	0.001
Velocidad terminal artificial	V_{max}	10

Tabla 3.4: Parámetros de simulación para 2D.

Parámetro	**Variable**	**Valor**
Radio de soporte de la ventana	h	0.05
Número de celdas por lado	k_{cel}	20
Número máximo de partículas	N_p	300
Incremento de tiempo	δt	0.0005

Tabla 3.5: Tabla de mediciones de tiempo en el simulador 2D.

Número de Partículas	**Tiempo (s)**	**Número de Partículas**	**Tiempo (s)**
300	172	290	164
280	158	270	152
260	146	250	139
240	133	230	126
220	119	210	112
200	106	190	101
180	94	170	90
160	83	150	77
140	70	130	64
120	57	110	52
100	46	90	41
80	36	70	30
60	26	50	19
40	14	30	9
20	6	10	3

Caso bidimensional

Los parámetros de simulación para el caso bidimensional, se muestran en la tabla 3.4. Con estos parámetros de realizaron 30 simulaciones de 90 segundos cada una, para diferentes cantidades de partículas de fluido y se midió su tiempo de ejecución en segundos. Los tiempos obtenidos se muestran en la tabla 3.5 y en la figura 3.30 (pág. 64) se grafican estos tiempos junto con el ajuste calculado mediante el programa GNUplot.

Observaciones

Los datos se aproximan a una recta. Se ha ajustado a una parábola $T = \alpha N^2 + \beta N + \gamma$, donde T es el tiempo y N es el número de partículas. Los valores α, β y γ, son arrojados por GNUplot en la forma $x \pm \Delta x$, donde x es la aproximación final y Δx es

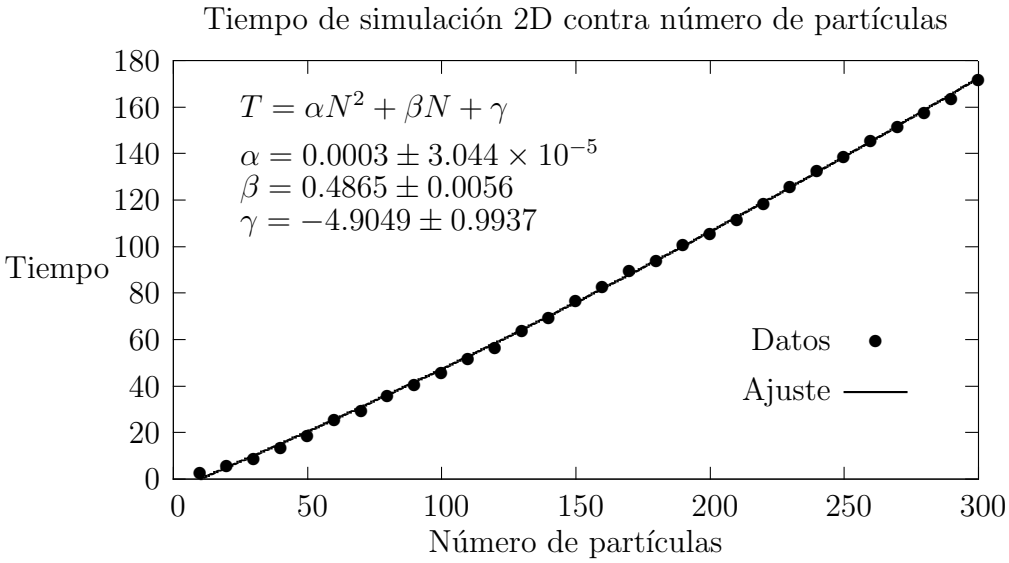


Figura 3.30: Gráfica de tiempos del simulador en 2D.

el error asintótico estándar. Se observa que el coeficiente α del término cuadrático es muy pequeño respecto a los demás, de manera que el ajuste se aproxima a una recta en el intervalo en que se efectuaron las mediciones. Según Müller en [5], la simulación es de orden $O(mn)$ donde n es el número de celdas (que en nuestro caso es $20^2 = 400$ y es constante) y m es el promedio de partículas dentro de una celda. Este número m depende del número de partículas N dentro del contenedor, en la proporción mostrada en la gráfica. Dado que se abarcan 90 segundos de simulación, notemos que con 170 partículas se ejecuta la simulación en tiempo real. Con 300 partículas la simulación se efectúa a la mitad de la velocidad real, como en *cámara lenta*.

Caso tridimensional

Los parámetros de simulación para el caso tridimensional, se muestran en la tabla 3.6 (pág. 65). Para este caso se realizaron 50 simulaciones de 90 segundos cada una, con distinto número de partículas de fluido, midiendo su tiempo de ejecución en segundos. Los tiempos obtenidos se muestran en la tabla 3.7 (pág. 65) y en la figura 3.31 (pág. 66) se da la gráfica y el ajuste calculado con GNUplot.

Observaciones

También se ha ajustado a una parábola $T = \alpha N^2 + \beta N + \gamma$, donde T es el tiempo y N es el número de partículas. Los valores α, β y γ, son arrojados por GNUplot en la forma $x \pm \Delta x$, donde x es la aproximación final y Δx es el error asintótico estándar. Ocurre también, que el coeficiente α del término cuadrático es muy pequeño respecto a los demás, como en el caso bidimensional, pero en esta ocasión los datos exhiben

Tabla 3.6: Parámetros de simulación para 3D.

Parámetro	**Variable**	**Valor**
Radio de soporte de la ventana	h	0.1
Número de celdas por lado	k_{cel}	10
Número máximo de partículas	N_p	500
Incremento de tiempo	δt	0.001

Tabla 3.7: Tabla de mediciones de tiempo en el simulador 3D.

Número de Partículas	**Tiempo (s)**	**Número de Partículas**	**Tiempo (s)**
500	358	490	348
480	340	470	328
460	313	450	301
440	298	430	291
420	281	410	268
400	260	390	253
380	244	370	231
360	221	350	214
340	201	330	196
320	184	310	179
300	165	290	160
280	151	270	144
260	137	250	128
240	120	230	111
220	105	210	97
200	89	190	83
180	77	170	69
160	63	150	58
140	52	130	46
120	41	110	37
100	33	90	29
80	25	70	21
60	18	50	14
40	12	30	9
20	6	10	5

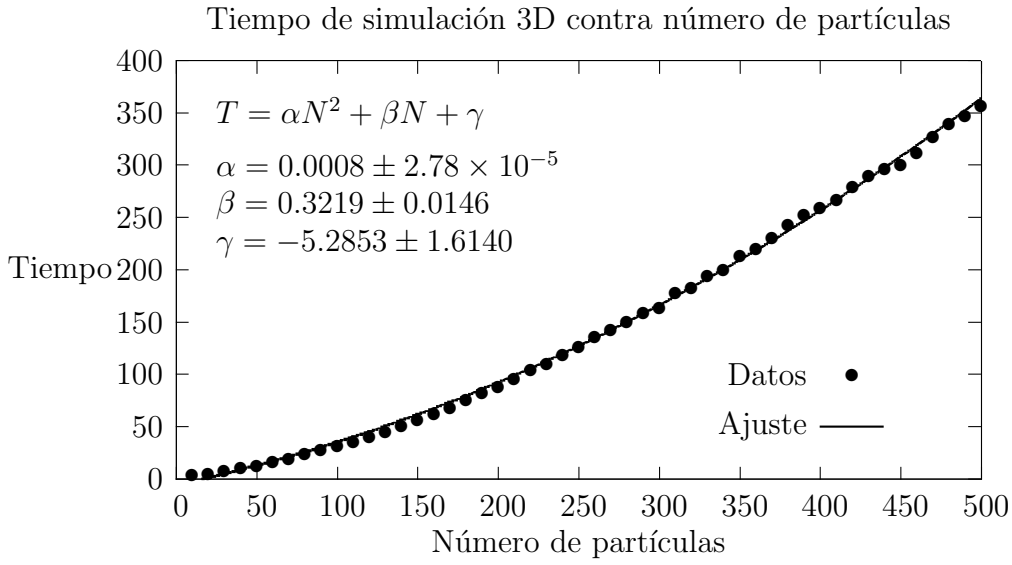


Figura 3.31: Gráfica de tiempos del simulador en 3D.

un comportamiento marcadamente más parabólico. Con 200 partículas se simulan los 90 segundos en tiempo real y con 310 partículas se tiene un efecto de *cámara lenta*, a la mitad de la velocidad real.

Capítulo 4

Implementación del módulo de visualización

Como hemos mencionado anteriormente, el módulo de visualización de nuestra aplicación nos permite visualizar un conjunto de datos, ya sea previamente simulados o en tiempo real durante el proceso de simulación. Estos datos los podemos desplegar interactivamente y en tiempo real de varias maneras, como se indica en las secciones 4.3 y 4.4. O bien, podemos generar una animación de alta calidad de la superficie del fluido, lo cual no es interactivo y se describe a continuación en la sección 4.2. En este punto aclaramos que, en el contexto del visualizador, con el término interacción nos estamos refiriendo a la habilidad del usuario de elegir la manera en que se visualiza el fluido, así como rotar el punto de vista, mientras se despliega la simulación. Esto es independiente de la interactividad en la simulación. En la figura 4.1 se muestra un diagrama que describe la etapa de visualización de la aplicación.

4.1. Arquitectura de la solución

En este trabajo de tesis se ha implementado una aplicación de software para simular y visualizar un fluido, utilizando una computadora personal de características comunes. La aplicación fue creada utilizando las bibliotecas Qt y OpenGL permitiéndole ser compilada en varias plataformas. Dado que no se utilizó algún tipo de paralelismo, puede ser ejecutada en cualquier computadora comercial. Básicamente

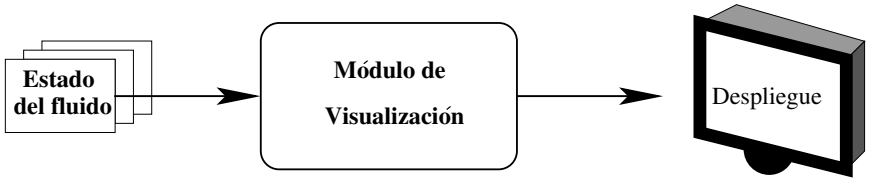


Figura 4.1: Diagrama del módulo de visualización.

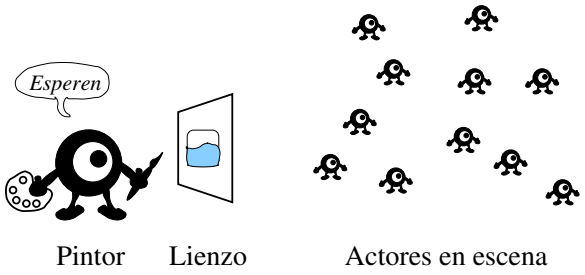


Figura 4.2: Metáfora del modelo de sincronización.

la aplicación se compone de dos módulos o etapas principales: simulación y visualización. En un principio se implementaron dos programas separados, que realizaban una de estas tareas cada uno. Sin embargo, posteriormente fue posible efectuar la simulación en tiempos interactivos y se optó por unir ambas implementaciones, logrando así visualizar el fluido mientras se está simulando. La unión de ambos módulos se realizó de manera directa. Básicamente se requiere ejecutar dos tareas y naturalmente son realizadas en dos hilos distintos. En la actualidad, la mayoría de los equipos tienen dos o más núcleos en el procesador, en esas condiciones no existe algún retraso entre los hilos.

Modelo de sincronización

El modelo de sincronización de los hilos se basa en exclusión mutua (*mutual exclussion, mutex*) utilizando las clases *QThread* y *QMutex* de la biblioteca Qt. Por tratarse de únicamente dos hilos, es muy sencillo sincronizar el uso del único recurso compartido, esto es, el estado de las partículas. Es similar a una situación en la que varios actores se encuentran en un escenario actuando alguna obra y deben ser retratados en ciertos momentos por un pintor. Los actores deben seguir el guión hasta que el pintor les solicita que se queden quietos para pintarlos, el pintor tardará un tiempo determinado (y deseablemente breve) en completar el retrato y les informa que ha terminado, entonces los actores continúan la obra. Ahora supongamos que los actores están realizando un desplazamiento en el momento que hay que pintarlos, entonces el pintor debe esperar un poco a que todos lleguen a la posición designada, antes de comenzar el retrato correspondiente. Este comportamiento, ilustrado metafóricamente en la figura 4.2, se implementó con el uso de un solo candado de tipo *mutex*. En la figura 4.3 se muestra un diagrama de la arquitectura de la solución, mostrando los dos módulos y el flujo de datos entre ellos y hacia el despliegue que recibe el usuario.

Velocidad *vs.* precisión

Es oportuno mencionar que, tanto en la etapa de simulación como en la de visualización, debemos establecer un compromiso entre la velocidad de ejecución y la

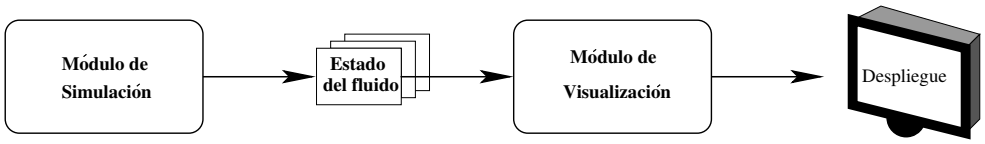


Figura 4.3: Diagrama de la arquitectura de la solución propuesta.

precisión de la aproximación obtenida. Debemos estar conscientes del hecho de que en todas las simulaciones numéricas, el resultado es una aproximación a la realidad física. Mientras mayor sea la precisión de la simulación obtenida, se requiere de un mayor tiempo de procesamiento. Asimismo, en la etapa de visualización, la superficie que se desea visualizar no se puede dibujar en la pantalla de la computadora con una precisión infinita. Se debe generar una aproximación finita y mientras mayor sea el nivel de detalle buscado, mayor es el tiempo de procesamiento requerido. Con esto en mente, se diseñó cada módulo de manera que es posible para el usuario elegir entre una ejecución interactiva del módulo u obtener resultados no interactivos pero más precisos. Los módulos son independientes en su capacidad de interacción, uno respecto al otro. La aplicación soporta las cuatro modalidades mostradas en la tabla de configuraciones 4.1.

Tabla 4.1: Posibles configuraciones de la aplicación.

	Simulador interactivo	**Simulador no interactivo**
Visualizador interactivo	Soportado	Soportado
Visualizador no interactivo	Soportado	Soportado

En línea *vs.* fuera de línea

En el módulo de simulación, la precisión de la aproximación viene dada en proporción inversa al tamaño del paso de tiempo del integrador (ver apéndice A). Además, con ayuda de un valor booleano, se puede elegir si el usuario puede o no interactuar con el fluido y la manera en que se van a generar los datos. En el caso no interactivo, los datos se guardan en archivos de texto de manera que pueden ser utilizados en otro momento para una reproducción contínua y posiblemente interactiva. Para el caso interactivo, el hilo de visualización obtiene los datos directamente de la estructura de datos que contiene las partículas (ver sección 2.1.3).

De manera similar, en el módulo de visualización se puede elegir el origen de datos, ya sea desde archivos de texto o directo del simulador. Los datos obtenidos se pueden visualizar de manera interactiva utilizando pequeñas esferas o el método de extracción

de superficie que se describirá en la sección 4.4. También se implementó un sencillo mecanismo, en la forma de un widget, para mostrar de manera gráfica e interactiva los valores de las cantidades escalares que arroja el simulador. Este mecanismo se explica en la sección 4.4.1. Si además se quiere obtener una visualización de mayor calidad y de manera no interactiva, se genera un conjunto de archivos de descripción de escenas para el trazador de rayos POVray. En un paso posterior, se utiliza este trazador de rayos para generar la animación no interactiva y de alta calidad. La descripción de las escenas de POVray se encuentra en la sección 4.2.

4.2. Visualización no interactiva

Primero presentamos el caso más sencillo, el caso no interactivo. Para generar una animación de alta calidad de la superficie del fluido utilizamos la técnica de trazo de rayos (sección 2.2.1, en la pág. 18) para renderizar un conjunto de objetos sin forma (sección 2.2.2, en la pág. 22). El trazo de rayos es la mejor técnica para generar imágenes de alta calidad y, de hecho, es la única que permite generar reflejos y transparencias físicamente realistas.

Básicamente, el visualizador carga los datos de la simulación desde la fuente de datos elegida, es decir, desde archivos de texto previamente creados o directamente de la estructura de datos del simulador. Se genera el conjunto de datos de manera que se conoce el estado de las partículas en un momento dado, para 24 instantes en cada segundo, es decir, se genera la simulación con una frecuencia de 24 estados por segundo. Para cada estado de la simulación, se genera un archivo de descripción de escenas para el trazador de rayos POVray.

Estructura de la escena

La escena contiene los siguientes objetos para ser desplegados por el trazador de rayos: las partículas de fluido representadas como pequeñas esferas de radio $0.2h$ de color verde, la superficie del fluido definida mediante objetos deformables creados a partir de formas esféricas de radio h con color ligeramente azul y un material que asemeja agua, el contenedor con forma de caja redondeada con un material semejante al vidrio y un plano de fondo con textura que permita observar los efectos de transparencia y reflejos en los objetos principales. Si no se coloca algún objeto de fondo, tanto el contenedor como el fluido difícilmente se pueden ver. Además de estos objetos, que son los que de hecho se verán en la escena, es necesario colocar una cámara que define el punto de vista que tendrá el proceso de renderizado y una lámpara que ilumine la escena.

Descripción del contenedor

Para describir en POVray la superficie del contenedor utilizamos la diferencia booleana de dos primitivas con forma de caja redondeada, uno ligeramente más pequeño

que el otro. Esto es con el fin de darle a la caja un cierto grosor, para que el efecto de refracción de los rayos de luz sea realista al llegar al fluido en su interior. POVray cuenta con una primitiva para modelar una caja con esquinas redondeadas. Viene definida como una isosuperficie de una función escalar llamada *f_rounded_box*, a la cual se le asigna un tamaño en cada una de las tres direcciones y el radio de la esquina. Se utiliza el objeto *isosurface* para definir la isosuperficie de la función escalar. En la figura 4.4 se muestra el contenedor renderizado con POVray. No se muestra con fluido en su interior. Nótese los efectos de reflejo y transparencia obtenidos. Para mayor claridad, presentamos a continuación la descripción de una caja redondeada:

```
isosurface {
    function {
        f_rounded_box ( x, y, z, radio, semi_lado_x, semi_lado_y, semi_lado_z )
    }
    max_gradient 1 contained_by {
        sphere { 0, RAD }
    }
}
```

Figura 4.4: Contenedor renderizado con POVray.

Descripción de la superficie del fluido

Para describir la superficie del fluido, se utiliza el objeto *blob* de POVray, en el cual se especifican las primitivas que definen el campo escalar (en este caso son N esferas) y el valor de umbral que define la isosuperficie, mediante la palabra reservada *threshold*, como se muestra en la figura 4.5 (izquierda). Sin embargo, como el simulador detecta la colisión de las partículas exactamente sobre la superficie del contenedor, el objeto sin forma (*blob*) generado se sale del mismo. Un ejemplo de esto se muestra en la figura 4.5 (derecha), se grafica en azul la isosuperficie del campo escalar generado por las partículas contenidas dentro de la caja. Para mantener la superficie del fluido dentro del contenedor, definimos al fluido como la intersección de la forma interior de la caja redondeada con el objeto deformable, de la siguiente manera:

```
blob {
    threshold 0.9
    sphere { < x1, y1, z1 >, h 1 }
    sphere { < x2, y2, z2 >, h 1 }
    ...
    sphere { < xN, yN, zN >, h 1 }
}
```

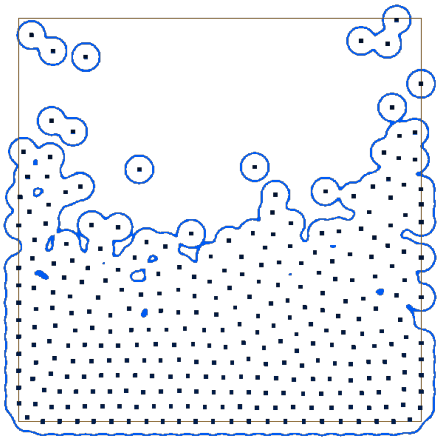


Figura 4.5: El objeto *blob* (definido a la izquierda) se sale de la caja redondeada (esquema bidimensional a la derecha).

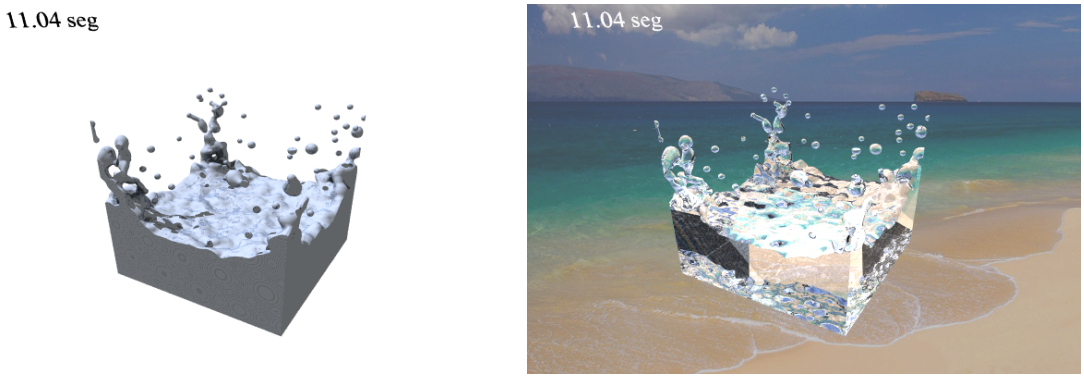


Figura 4.6: Superficie del fluido renderizada con *blob* de POVray. A la izquierda con material sólido y a la derecha con material transparente.

```
intersection {
    object { CAJA_REDONEADA_INTERIOR }
    blob {
        threshold 0.9
        sphere { < x1, y1, z1 >, h 1 }
        sphere { < x2, y2, z2 >, h 1 }
        ...
        sphere { < xN, yN, zN >, h 1 }
    }
    scale .99
    pigment { color COLOR_AGUA }
    interior { ior 1.3 }
}
```

Notemos que el objeto que hemos modelado como superficie del fluido simulado, se escala ligeramente por un factor de 0.99. Esto se hace con el fin de evitar un error

de ambigüedad que ocurre cuando se intenta trazar rayos que pasan a través de dos o más superficies coincidentes, es decir, que ocupan el mismo lugar. La solución de escalar imperceptiblemente uno de los objetos coincidentes se sugiere en el manual de POVray[1]. La superficie de fluido obtenida con este método se muestra en la figura 4.6. No se muestra la caja redondeada para apreciar mejor su forma. En la figura 4.7 aparecen juntos la caja redondeada y el fluido. Observe los efectos de transparencia y reflejos simulados por el trazado de rayos.

Otro problema que es necesario mencionar tiene que ver con la profundidad de recursión en el algoritmo de trazo de rayos. En POVray, por defecto se tiene una profundidad de recursión de 6. Este valor es suficiente para la mayoría de las aplicaciones usuales del programa. Sin embargo, cuando colocamos dos o más objetos transparentes anidados unos dentro de otros, los objetos interiores se renderizan completamente negros, como se puede apreciar en la figura 4.7 del lado izquierdo, debido a que el nivel de recursión no es suficiente para decidir su color. Esto se puede resolver fácilmente aumentando el nivel de recursión de la siguiente manera:

```
global_settings { max_trace_level 15 }
```

Debemos considerar que aumentar el nivel de recursión inevitablemente incrementará el tiempo de renderizado. Se obtiene la imagen del lado derecho de la figura 4.7.

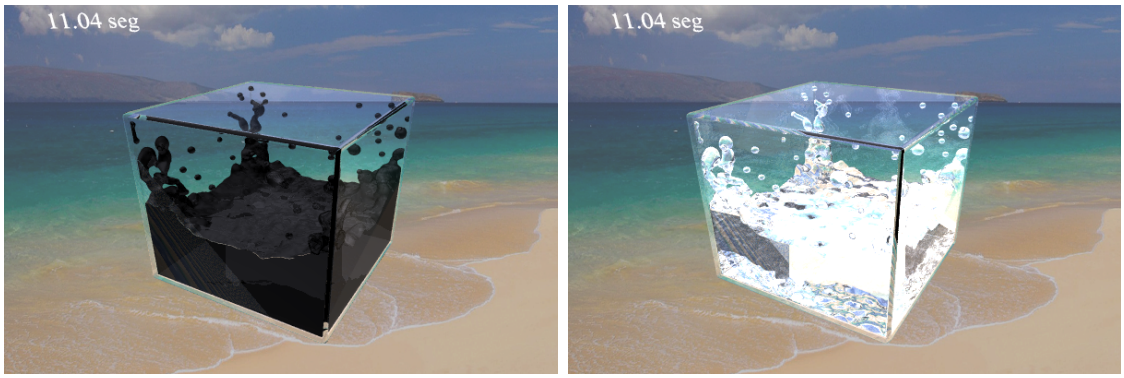


Figura 4.7: Caja redondeada y fluido juntos. A la izquierda se tiene el nivel de recursión por defecto. A la derecha se ha incrementado a un valor de 15.

Renderizado

Una vez que se han generado los archivos de descripción de escena, se procede a renderizar las imágenes correspondientes, utilizando POVray. Esta etapa se automatiza mediante un programa creado en lenguaje C. La velocidad de generación de las

[1] http://www.povray.org/documentation/view/3.6.0/146/

imágenes depende completamente de la implementación de POVray y el equipo que se esté utilizando. Además, el tiempo que se tarda el renderizado de cada una, aumenta con el número de partículas y el nivel de recursión elegido. Finalmente, se genera un video con las imágenes creadas utilizando el programa *mencoder*. Dicho video se crea para reproducirse a 24 fps, de manera que la animación resultante se reproduce en el tiempo correcto de la simulación. Un ejemplo de uso del programa *menconder* es:

```
mencoder "mf://fluido????.png" -mf type=png:fps=24 -o fluido.avi \
                -ovc lavc -lavcopts vcodec=wmv2
```

En el primer argumento se indica la secuencia de imágenes en formato PNG que fueron creadas con POVray, las cuales tienen por nombre *fluido*, seguido de un número de secuencia de 4 dígitos, seguido de la extensión. También se indica el formato de las imágenes PNG y la frecuencia de reproducción del video a 24 cuadros por segundo. Luego se indica el nombre del archivo de salida y otras opciones del video que será generado utilizando el codec *wmv2*, que corresponde al formato de video *wmv*.

Existen trabajos en la literatura que de hecho modifican la implementación de POVray (como es el caso de [9] en donde se modifica para trazar superficies compuestas por muestras puntuales), mientras que otros abordan el problema de implementar un trazador de rayos propio. Esto queda fuera del propósito del presente trabajo, en vez de ello nos hemos enfocado al tema de visualización interactiva.

4.3. Visualización interactiva bidimensional

Como hemos mencionado en la sección 2.2.3, para la visualización interactiva utilizamos el método de aplanamiento. Se construye una aproximación a la isosuperficie *n-dimensional* del campo de color definido en 2.1.4, linealizándola mediante elementos de superficie *(n-1)-dimensionales*.

En el caso concreto de visualización de la superficie en dos dimensiones, $n = 2$, la isosuperficie del campo escalar se conoce como *curva de nivel*, pues es de dimensión uno. A esta curva la aproximamos mediante un conjunto de segmentos de línea orientados de tal manera que van delineando el contorno de la curva. En nuestro algoritmo, utilizamos la cantidad N_{res} como una medida de la resolución, la cual indica que se divide el espacio de trabajo en $N_{\text{res}} \times N_{\text{res}}$ celdas. Para construir la curva, se evalúa el campo de color $c(\mathbf{p})$ en las $(N_{\text{res}} + 1)^2$ esquinas. Si el valor obtenido para una cierta esquina $\mathbf{p}$ se encuentra dentro del intervalo $[\text{MIN}_{\text{color}}, \text{MAX}_{\text{color}}]$, se dibuja en esa esquina un segmento de línea orientado de manera perpendicular al gradiente del campo de color $\nabla c(\mathbf{p})$ en ese punto. La longitud del segmento es el doble de la longitud del lado de una celda, con el fin de que los segmentos se cierren y la curva no quede dibujada como con línea punteada. En la figura 4.8 se muestra un ejemplo de la curva delineada mediante segmentos de línea. También se muestra la ubicación de las partículas mediante pequeñas cajas. En la aplicación también es posible visualizar

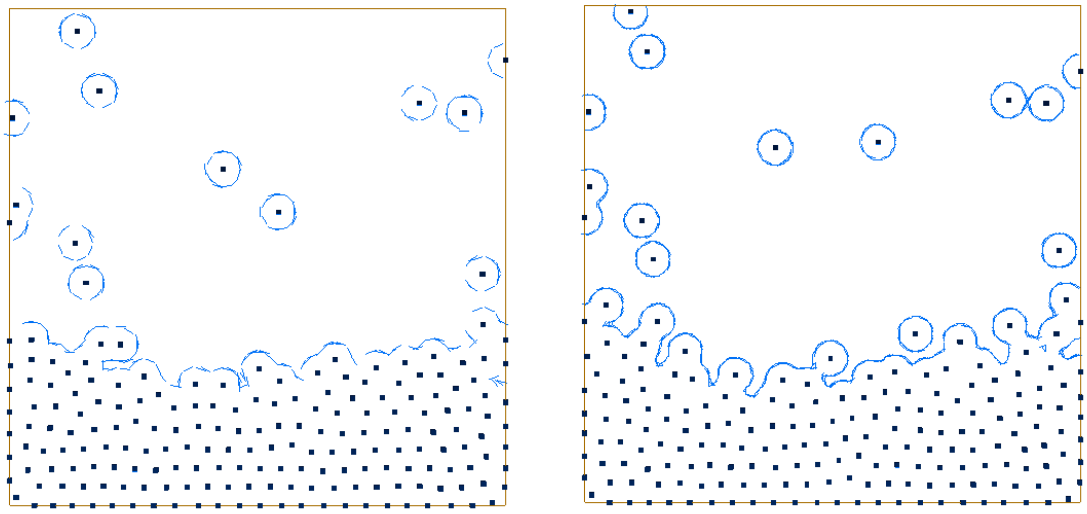


Figura 4.8: Ejemplos de curva de nivel, aproximada mediante segmentos de línea orientados en resolución baja (116^2) y alta (200^2).

las partículas como esferas de radio h. El algoritmo se muestra a continuación:

Procedimiento **CurvaNivel**
Entrada: Número de celdas de la caja, definido por el número de pixeles *PIX* que mide ésta por lado y un incremento *INC* que restringe la resolución con que se efectúan las evaluaciones.
Resultado: Se dibuja con líneas el contorno de la curva de nivel formada por los puntos cuyo valor en el campo de color se encuentre en el intervalo $[\text{MIN}_{\text{color}}, \text{MAX}_{\text{color}}]$.

```
begin
    /* Para cada pixel a una resolución de N_res = PIX / INC */
    para y de 0 a PIX con incremento INC
        para x de 0 a PIX con incremento INC
            /* Calcula el punto p correspondiente dentro de la caja */
            p ← ( x/PIX, y/PIX )
            /* Evalúa el campo de color en p */
            color ← c(p)
            si MIN_color ≤ color ≤ MAX_color
                Aplica al pixel (x, y) la transformación que corresponda para ubicarlo en pantalla
                Dibuja una línea de largo 2·INC y orientada perpendicularmente a ∇c(p)
            finsi
        finpara
    finpara
end
```

Este algoritmo es de orden $O(N_{\text{res}}^2)$ y para valores muy grandes de N_{res} el algoritmo resulta ineficiente. Sin embargo, para el caso bidimensional utilizamos rejillas con resoluciones desde $100^2 = 10000$ celdas y hasta $400^2 = 160000$ celdas, obteniendo resultados interactivos. Pero cuando se implementó este algoritmo para el caso tridi-

mensional, que es de orden $O(N_{\text{res}}^3)$, ya no fue posible obtener interacción por parte del usuario. Estamos hablando de rejillas de mínimo 100^3, esto es, un millón de celdas. Para tres dimensiones el algoritmo resulta demasiado lento y por ello fue necesario diseñar un nuevo algoritmo, el cual se explica en la sección 4.4, a continuación.

4.4. Visualización interactiva tridimensional

Para visualizar de manera interactiva el fluido en el caso tridimensional, creamos un algoritmo de reconstrucción de superficie, al que hemos denominado *encajamiento de esferas* (*marching spheres*). La construcción de este algoritmo surge de la combinación de ideas presentes en dos de los algoritmos de reconstrucción de superficie más populares: aplanamiento (sección 2.2.3) y encajamiento de cubos (sección 2.2.4). De hecho, está fuertemente basado en el primero, implementado en la manera que hemos descrito en la sección 4.3. Tiene en común con éste, el que la superficie *n-dimensional*, ahora con $n = 3$, se reconstruye mediante elementos de superficie *(n-1)-dimensionales* orientados de manera perpendicular al gradiente del campo escalar. Igualmente, se tiene que no es necesaria mayor información acerca de la conectividad entre dichos elementos y tampoco necesitan estar ordenados.

Difiere del método de aplanamiento en la manera en que se buscan los puntos del dominio suficientemente próximos a la isosuperficie buscada, es decir, los puntos superficiales. Hemos visto, en el caso bidimensional, que el método de aplanamiento busca los puntos candidatos a ser superficiales sobre una rejilla de N_{res}^3 celdas, lo cual para el caso tridimensional es ineficiente. Para reducir la cantidad de cálculos, hemos restringido la búsqueda de candidatos sólo a puntos ubicados en la vecindad de las partículas de fluido, pues evidentemente es en estas regiones donde realmente se pueden encontrar puntos superficiales. Esto puede verificarse a partir de la definición del campo escalar empleado, es decir, el campo de color que se define en la sección 2.1.4 de la siguiente manera:

$$c(\mathbf{p}) = \sum_j m_j \frac{1}{\rho_j} W(\mathbf{p} - \mathbf{r}_j, h)$$

Y cuyo gradiante es:

$$\nabla c(\mathbf{p}) = \sum_j m_j \frac{1}{\rho_j} \nabla W(\mathbf{p} - \mathbf{r}_j, h)$$

De acuerdo a Müller en [5], un punto $\mathbf{p}$ que es superficial cumple con la condición de que la magnitud del gradiente en ese punto es mayor que un cierto umbral l pequeño y positivo, esto es:

$$\|\nabla c(\mathbf{p})\| > l \tag{4.1}$$

También se tiene que el vector unitario normal a la superficie es

$$\mathbf{n}(\mathbf{p}) = -\frac{\nabla c(\mathbf{p})}{\|\nabla c(\mathbf{p})\|}$$

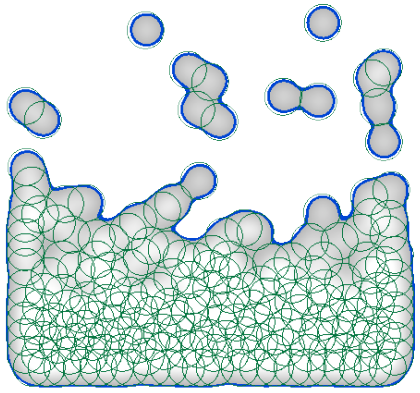


Figura 4.9: Ejemplo del campo de color bidimensional con curva de nivel en azul y zonas de influencia en verde.

debido a que el gradiente del campo escalar apunta en la dirección en la que el valor del campo aumenta, esto es, apunta hacia el interior de la isosuperficie.

Veamos porqué los puntos superficiales no pueden estar alejados de las partículas. Sea un punto arbitrario $\mathbf{q}$, tal que se encuentra fuera del radio de influencia de todas las partículas. El gradiente en ese punto es $\nabla c(\mathbf{q}) = \mathbf{0}$, entonces por la condición (4.1), $\mathbf{q}$ no es superficial. Por lo tanto, todos los puntos superficiales se encuentran dentro del radio de influencia h de alguna partícula.

Ahora bien, ¿qué tan cerca se encuentran los puntos superficiales respecto al centro de las partículas? En la figura 4.9 se muestra la gráfica del campo de color bidimensional para $N = 300$ partículas, así como las zonas de influencia de las partículas, con radio h. Se observa que los puntos de la curva de nivel, correspondiente a un valor de color cercano a cero, se encuentran a una distancia aproximada ligeramente menor que el radio de influencia h. Esto nos indica que los puntos superficiales del campo de color no se encuentran demasiado cerca del centro de las partículas, sino mas bien cerca del borde de sus zonas de influencia.

Con esta observación en mente, se ha diseñado el algoritmo cuya idea básica es semejante a la que que yace detrás del método de encajamiento de cubos, en el cual se va desplazando una plantilla cúbica a través del espacio de trabajo, evaluando el campo en sus esquinas y, de acuerdo al resultado, se va construyendo la isosuperficie agregando polígonos. En nuestro algoritmo se va desplazando una plantilla esférica, formada por k puntos equidistantes del centro (ver figura 4.10), a distancia $0.85h$. En cada uno de ellos se evalúa el campo escalar y se verifica si el resultado se encuentra dentro de cierto rango, en cuyo caso se agrega un elemento de superficie orientado de manera perpendicular al gradiente del campo en ese punto (ver figura 4.12).

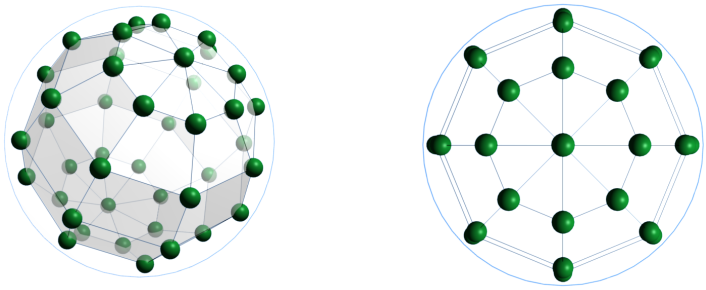


Figura 4.10: Diagrama que muestra los puntos que forman la plantilla esférica desde dos perspectivas distintas. El círculo azul indica el radio de influencia de la partícula.

Opcionalmente, se puede refinar el resultado acercando el punto a la isosuperficie mediante algún método de minimización, antes de calcular su orientación. En nuestra implementación se puede elegir la aplicación de tres iteraciones del *método de descenso por gradiente* (*MDG*)[2], para acercar aún más el punto superficial a la isosuperficie de color cero, agregando una pequeña cantidad de cálculos a la visualización. En la figura 4.13 se compara la superficie extraída con y sin el uso del MDG. Nótese como los elementos de superficie se acercan a la superficie de las esferas que indican el límite de la zona de influencia de las partículas. El algoritmo se muestra en la figura 4.11.

La complejidad de este algoritmo es $O(kN)$, donde N corresponde al número de partículas y k al número de puntos que forman cada plantilla esférica. En nuestra implementación hemos usado $k = 96$ puntos en cada esfera y hasta 500 partículas, lo que nos da 48000 evaluaciones del campo de color, sin usar MDG. Esto contrasta con el millón de evaluaciones que sería necesario efectuar con el método de aplanamiento de la sección 4.3.

4.4.1. Diseño del widget sensor

Uno de los objetivos de la tesis es diseñar un mecanismo que muestre al usuario, de manera gráfica, las cantidades físicas que se están simulando al interior del fluido. Para el caso tridimensional proponemos un *widget* gráfico que muestre las cantidades escalares dentro del fluido, mostrando una rebanada a la vez. Lo hacemos así debido a que si intentamos mostrar todo un volumen de datos, no es posible ver aquellos datos que se encuentren obstruídos por otros más cercanos al observador, es decir, sólo se verían los datos en la superficie del volumen y quedarían ocultos los que estén ubicados detrás de éstos.

El sensor se puede mover de manera interactiva, permitiendo al usuario observar las cantidades escalares en distintas regiones de interés, dentro del espacio de trabajo.

[2]Presentamos una breve descripción del MDG en el apéndice B

Procedimiento **MarchingSpheres**
Entrada: El conjunto de N partículas P y
el conjunto de k puntos E que forman la plantilla esférica.
Resultado: Se dibuja en la escena tridimensional un conjunto de elementos de superficie que delinean la isosuperficie del campo de color definida en un intervalo cercano a cero.

```
begin
  para cada p ∈ P
    para cada e ∈ E
      /* Calcula el punto p' correspondiente dentro de la caja */
      p' ← p + e
      si p' está dentro de la caja
        /* Evalúa el campo de color en p' */
        color ← c(p')
        si ε ≤ color ≤ MAX_color
          si se va a usar MDG
            repite 3 veces
              p' ← p' − α · ∇c(p')
            finrepite
          finsi
          /* Calcular gradiente en el nuevo punto p' */
          n = ∇c(p')
          Dibuja un elemento de superficie ubicado en el punto p'
                y orientado perpendicularmente a n
        finsi
      finsi
    finpara
  finpara
end
```

El valor de ϵ es muy pequeño pero positivo y suficiente para garantizar que se pueda calcular el gradiente en p'. El valor de $\text{MAX}_{\text{color}}$ es de 0.1. El valor de α es de 0.01.

Figura 4.11: Pseudocódigo del algoritmo de encajamiento de esferas.

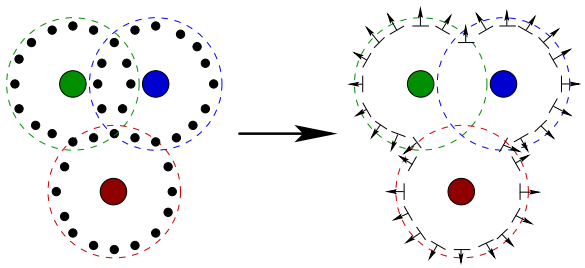


Figura 4.12: Esquema bidimensional que ilustra el resultado del algoritmo para tres partículas. Se muestran sus radios de soporte, los puntos candidatos a superficiales en negro (izquierda) y los elementos de superficie obtenidos (derecha).

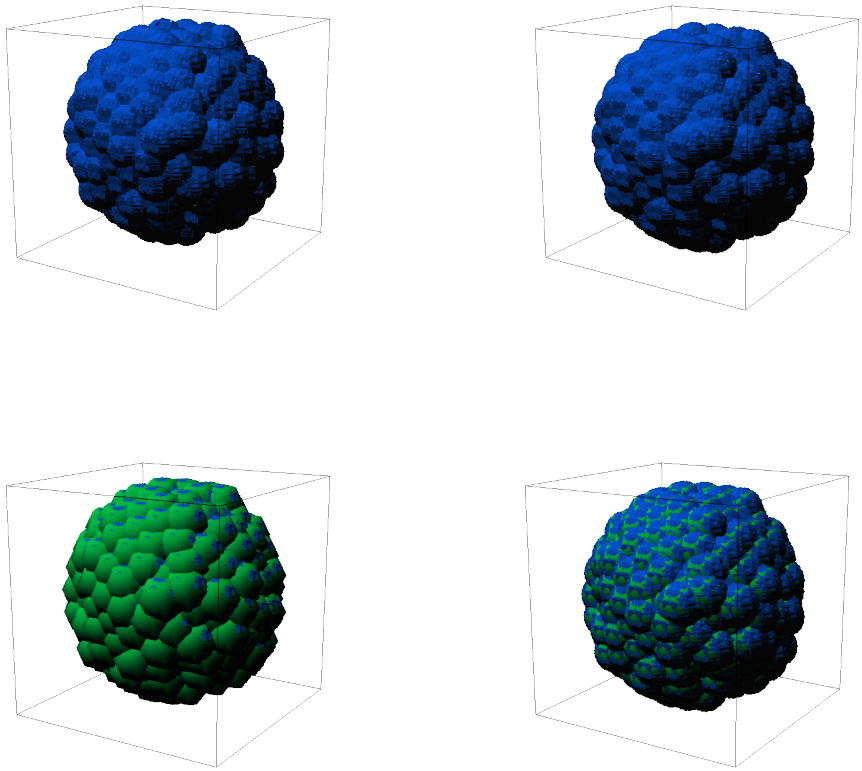


Figura 4.13: Comparación del resultado sin MDG (columna izquierda) y con MDG (columna derecha). En el renglón inferior se destaca la diferencia indicando la zona de influencia de las partículas mediante esferas verdes de radio h.

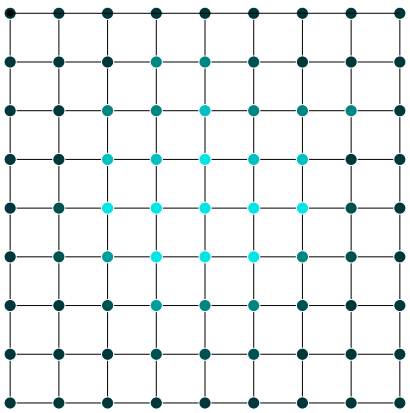


Figura 4.14: Esquema del sensor formado por $k^2 = 64$ celdas. Se muestran las esquinas de las celdas con su color asociado.

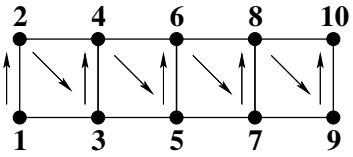


Figura 4.15: Esquema de una tira de cuadriláteros de OpenGL.

El usuario puede visualizar, en rebanadas, la cantidad en todo el volumen de fluido. En este prototipo del sensor, se pueden visualizar cantidades escalares, tales como la presión y la densidad, mostrando su valor en una escala de colores. No se puede visualizar una cantidad vectorial, como la velocidad o la aceleración del fluido, pero se puede visualizar el campo de magnitudes de estas cantidades.

En nuestra aplicación se tiene implementado el sensor para valores de presión, pero es muy sencillo modificarlo para mostrar densidad, rapidez o magnitud de la aceleración. Para realizar el cálculo de la presión p en un punto arbitrario del espacio $\mathbf{q}$ aplicaremos la interpolación utilizada en el simulador HSP, capítulo 2.1:

$$p(\mathbf{q}) = \sum_j m_j \frac{p_j}{\rho_j} W(\mathbf{q} - \mathbf{q}_j, h)$$

donde j itera sobre las partículas, W es la misma ventana empleada en el simulador, m_j es la masa de la partícula j, $\mathbf{q}_j$ es su posición, ρ_j su densidad y p_j es su presión. Hemos elegido mostrar los valores en escala de color cian, para promover el contraste y visibilidad respecto a los otros colores presentes en la escena: partículas en color verde y superficie en color azul. También es posible modificar la escala de colores o el rango de la cantidad que corresponde a cada color.

Concretamente, el sensor es un cuadrado unitario, dividido en k celdas por lado, es decir, es una malla poligonal formada por k^2 pequeños cuadrados. Se muestra el sensor esquemáticamente en la figura 4.14. A cada una de sus $(k+1)^2$ esquinas se le asigna un color, en función del valor que tome la cantidad en ese punto. Si el valor de k es suficientemente grande, se obtiene una imagen a color que representa los valores de la cantidad escalar sobre el cuadrado. Dado que la visualización, tanto en 2D como en 3D, la realiza la aplicación utilizando una escena tridimensional creada con la biblioteca OpenGL, hemos construido el sensor cuadrado mediante una serie de *tiras de cuadriláteros*, llamadas *GL_QUAD_STRIP* en OpenGL.

Una tira de cuadriláteros de OpenGL, como la mostrada en la figura 4.15, se construye agregando los vértices de los cuadros en el orden dado, de la siguiente manera:

```
glBegin (GL_QUAD_STRIP);
    glMaterialfv (GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, color_1);
    glVertex3f (v_1x, v_1y, v_1z);
    glMaterialfv (GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, color_2);
    glVertex3f (v_2x, v_2y, v_2z);
    ...
    glMaterialfv (GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, color_m);
    glVertex3f (v_mx, v_my, v_mz);
glEnd ();
```

donde la tira está formada por m vértices coplanares $\mathbf{v}_i$, cada uno del color que corresponda. En la aplicación se dibujan k de estas tiras con k cuadrados cada una, calculando para cada esquina el color que le corresponde. Podemos utilizar una resolución del sensor k relativamente menor que si se dibujara la imagen pixel por pixel. Esto es debido a que si cada esquina de un cuadrilátero en OpenGL tiene un color distinto, se interpola linealmente el color de los puntos interiores (ver figura 4.16). Debido a esta interpolación, la imagen se genera con una resolución mayor a la dada por el número de vértices del sensor.

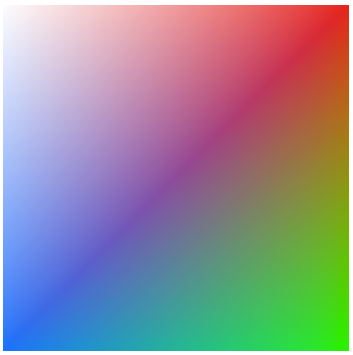


Figura 4.16: Ejemplo de interpolación de color en un cuadrilátero con vértices de distinto color.

Este algoritmo es de orden k^2, donde k es la resolución del cuadro sensor. Obtenemos resultados interactivos para resoluciones de $100^2 = 10000$ celdas y en la figura 4.17 se muestra un ejemplo de la ejecución del widget mostrando valores de presión.

El pseudocódigo del algoritmo se muestra a continuación:

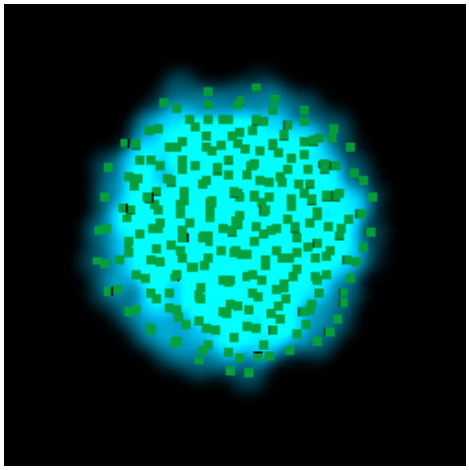


Figura 4.17: Ejemplo del sensor de presión con 100^2 celdas. Se muestran las partículas en verde.

Procedimiento **dibujaSensorPresión**
Entrada: La coordenada z del sensor cuadrado perpendicular a dicho eje.
Resultado: Se dibuja el cuadrado poligonal coloreado en función de
los valores de la cantidad escalar en su posición.
begin
  para cada renglón del sensor
    Comienza la tira de cuadriláteros
    para cada columna del sensor más uno
      Calcula el color de la esquina inferior izquierda
      Agrega a la tira un vértice en la posición de la esquina inferior izquierda
        con el color calculado
      Calcula el color de la esquina superior izquierda
      Agrega a la tira un vértice en la posición de la esquina superior izquierda
        con el color calculado
    finpara
    Termina la tira de cuadriláteros
  finpara
end

4.5. Resultados de visualización

Las pruebas del módulo de visualización, para dos y tres dimensiones, se realizaron en el mismo equipo cuyas caracterísstas se listaron en la tabla 3.2 (pág. 61).

Caso bidimensional

Para realizar las mediciones de tiempo de visualización en dos dimensiones, se ejecutó el algoritmo de la sección 4.3 (pág 74) con los datos de 100 partículas para diferentes resoluciones, que van de 50 a 500 celdas por lado.

Tabla 4.2: Tabla de mediciones del tiempo de visualización en 2D.

Celdas por lado	**cps**	**Tiempo (s)**	**Celdas por lado**	**cps**	**Tiempo (s)**
500	15.0602	0.0664	490	15.6740	0.0638
480	16.3399	0.0612	470	17.0648	0.0586
460	17.7305	0.0564	450	18.5185	0.0540
440	19.3798	0.0516	430	20.2429	0.0494
420	21.2766	0.0470	410	22.3214	0.0448
400	23.4742	0.0426	390	24.7525	0.0404
380	25.9067	0.0386	370	27.3224	0.0366
360	28.9017	0.0346	350	30.4878	0.0328
340	32.2581	0.031	330	34.2466	0.0292
320	36.2319	0.0276	310	39.0625	0.0256
300	41.3223	0.0242	290	44.2478	0.0226
280	47.6190	0.021	270	51.0204	0.0196
260	54.9451	0.0182	250	59.5238	0.0168
240	64.1026	0.0156	230	69.4444	0.0144
220	76.9231	0.013	210	83.3333	0.012
200	90.9091	0.011	190	102.0408	0.0098
180	113.6364	0.0088	170	125	0.008
160	142.8571	0.007	150	161.2903	0.0062
140	185.1852	0.0054	130	217.3913	0.0046
120	250	0.004	110	294.1176	0.0034
100	357.1429	0.0028	90	454.5455	0.0022
80	555.5556	0.0018	70	714.2857	0.0014
60	1000	0.001	50	1250	0.0008

Se mide el tiempo necesario para dibujar la curva de nivel, el cual se reporta (en segundos) en la tercera y sexta columnas de la tabla 4.2. El número de veces que se puede dibujar la curva de nivel en un segundo es el recíproco de este tiempo, y se reporta en la segunda y quinta columnas de la misma tabla (en cuadros por segundo, cps). En las figuras 4.18 y 4.19 (pág. 85) se muestran, respectivamente, las gráficas de tiempo y frecuencia de visualización, en función de la resolución utilizada.

Observaciones

Como hemos mencionado en la la sección 4.3 (pág 74), se espera un comportamiento cuadrático en la medición de tiempos de visualización en función de la cantidad de celdas por lado, lo cual se confirma experimentalmente en el ajuste realizado con GNUplot para la gráfica de la figura 4.18 (pág. 85). Se ajustó a una parábola de la forma $T = \alpha N_c^2 + \beta N_c + \gamma$, donde N_c es el número de celdas por lado y T es el tiempo en segundos. Los valores α, β y γ, son arrojados por GNUplot en la forma $x \pm \Delta x$,

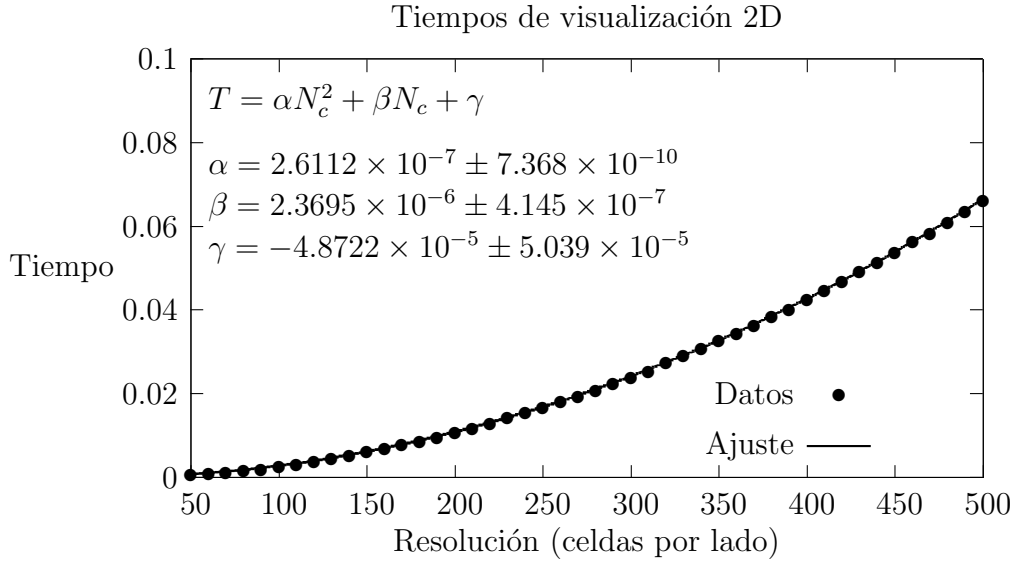


Figura 4.18: Gráfica de tiempos de visualización en 2D.

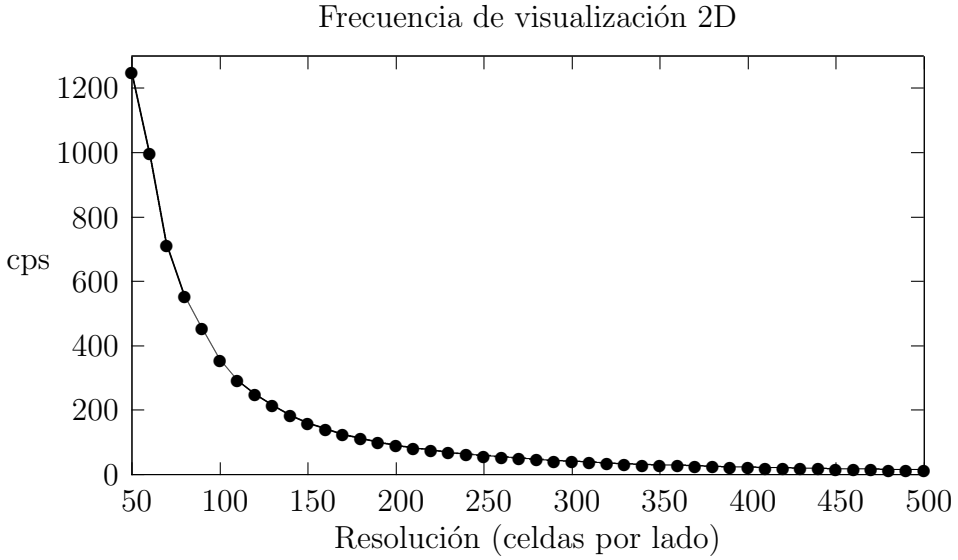


Figura 4.19: Gráfica de frecuencia de visualización en cuadros por segundo (cps) para dos dimensiones.

donde x es la aproximación final y Δx es el error asintótico estándar.

De la gráfica de la figura 4.19 (pág. 85) y la tabla 4.2 (pág. 84) obsérvese que para todas las resoluciones utilizadas se obtuvo una visualización interactiva de al menos 15 cuadros por segundo[3]. Ahora bien, dado que la longitud del paso de tiempo δt del simulador se eligió para realizar 24 pasos de simulación en un segundo, obtendremos la visualización en tiempo real, esto es a 24 cps, con una resolución de alrededor de 390 celdas por lado.

Caso tridimensional

En el caso de medición de tiempos de visualización en tres dimensiones, se puso a prueba el algoritmo de la sección 4.4 (pág 76), utilizando una plantilla esférica compuesta de 96 puntos y variando el número de partículas en un rango que va desde 40 hasta 800 partículas.

Se mide el tiempo necesario para construir la superficie envolvente del fluido, el cual se reporta (en segundos) en la tercera y sexta columnas de la tabla 4.3 (pág. 87). El número de veces que se puede dibujar la superficie en un segundo es el recíproco de este tiempo, y se reporta en la segunda y quinta columnas de la misma tabla (en cuadros por segundo, cps). En las figuras 4.20 (pág. 87) y 4.21 (pág. 88) se muestran, respectivamente, las gráficas de tiempo y frecuencia de visualización, en función del número de partículas.

Observaciones

Se comprueba experimentalmente que el algoritmo tiene una complejidad temporal lineal, en función del número de partículas. Se ajustó a una línea recta de la forma $T = \alpha N_p + \beta$, donde N_p es el número de partículas y T es el tiempo en segundos. Los valores α y β, son arrojados por GNUplot en la forma $x \pm \Delta x$, donde x es la aproximación final y Δx es el error asintótico estándar.

Como se observa en la tabla 4.2 (pág 84), para visualizar la simulación a 15 cps o más, es necesario tener 250 partículas o menos. También, dado que la longitud del paso de tiempo δt del simulador se eligió para realizar 24 pasos de simulación en un segundo, si queremos una frecuencia de 24 cps, debemos usar alrededor de 110 partículas.

Tener frecuencias de al menos 15 cps es una recomendación frecuente para realizar animaciones. Sin embargo, para visualización de fluidos en 3D, los autores suelen reportar frecuencias menores a 15 cps. Por ejemplo, en [5] y [12], Müller reporta

[3]Recordemos que para percibir el movimiento de una animación, se debe mostrar al usuario con una frecuencia de al menos 10 cps.

Tabla 4.3: Tabla de mediciones del tiempo de visualización en 3D.

Número de partículas	**cps**	**Tiempo (s)**	**Número de partículas**	**cps**	**Tiempo (s)**
800	4.0377	0.2477	700	4.6325	0.2159
600	5.4085	0.1849	500	6.6820	0.1497
480	6.8235	0.1466	460	7.1782	0.1393
440	7.6139	0.1313	420	8	0.125
400	8.3609	0.1196	380	9.2176	0.1085
360	9.4478	0.1058	340	9.6649	0.1035
320	10.9296	0.0915	300	11.6065	0.0862
280	11.7943	0.0848	260	14.096	0.0709
240	15.2545	0.0656	220	15.5102	0.0645
200	15.5882	0.0642	180	18.1944	0.0550
160	20.6393	0.0485	140	22.26	0.0449
120	23.075	0.0433	100	25.3125	0.0395
80	35.5217	0.0282	60	37.6667	0.0265
40	48.7273	0.0205			

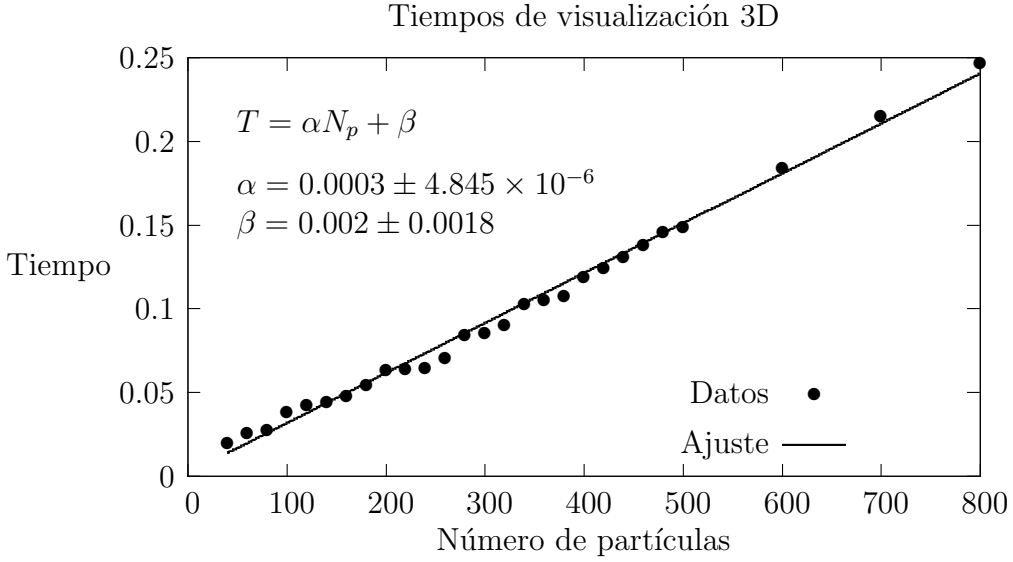


Figura 4.20: Gráfica de tiempos de visualización en 3D.

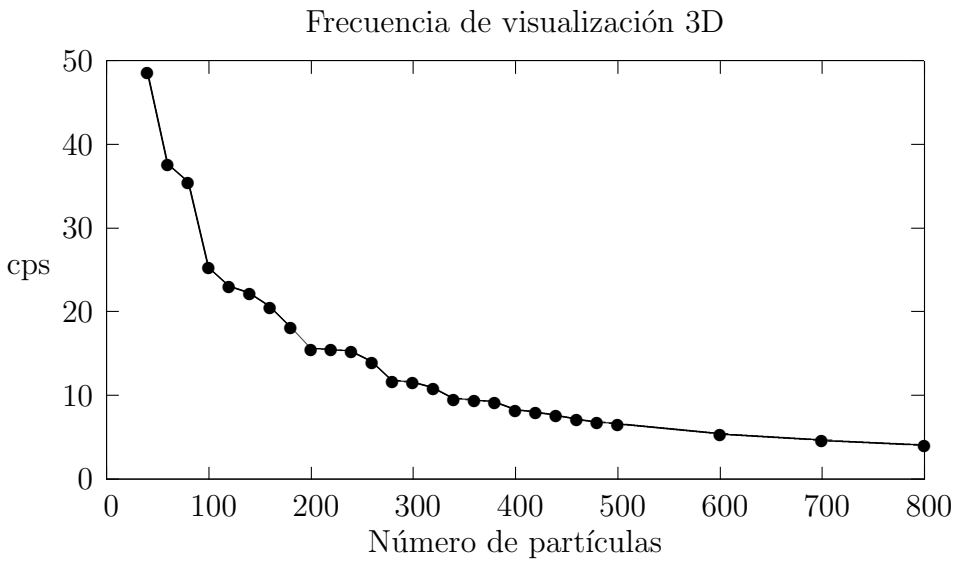


Figura 4.21: Gráfica de frecuencia de visualización en cuadros por segundo (cps) para tres dimensiones.

Tabla 4.4: Tabla de mediciones del tiempo de visualización en 3D con más partículas.

Número de partículas	**cps**	**Tiempo (s)**
1000	2.32	0.43
2500	0.95	1.05
5000	0.47	2.13

frecuencias de 5 cps con aplanados, y de 6 cps con encajamiento de cubos, respectivamente. Inclusive en [9] se reporta una frecuencia de 0.7 cps con 2400 a 3900 partículas, usando aplanamiento.

Entre nuestros resultados resaltamos, para 90 segundos de simulación: 6.7 cps para 500 partículas y 4 cps para 800 partículas. Con el fin de comparar con los trabajos mencionados, se midió el desempeño del algoritmo para un mayor número de partículas y, por limitaciones de tiempo, para 10 segundos de simulación, obteniéndose los resultados de la tabla 4.4. Con todo esto, consideramos que nuestros resultados son competitivos, respecto a los trabajos mencionados en el estado del arte.

Capítulo 5

Conclusiones

El objetivo principal de la tesis fue implementar el módulo de visualización, en línea y fuera de línea, para un simulador HSP. En dos dimensiones se visualiza la curva envolvente del fluido en tiempo real, es decir, conforme se va simulando el fluido, utilizando aplanados lineales.

En tres dimensiones se produjo una contribución original, se trata de la propuesta del método de encajamiento de esferas, que se explica en la sección 4.4, pág. 76. Hasta donde sabemos, este método no existe en la literatura de visualización de fluidos. No se había planeado en un principio desarrollar esta contribución, pero la idea surgió al momento de pasar de un algoritmo de orden $O(n^2)$ en 2D a uno de orden $O(n^3)$ para 3D, como se explica al final de la sección 4.3, pág. 74. Este método está diseñado para visualizar una superficie envolvente a partir de un conjunto no muy numeroso de partículas, por lo cual es eficiente y de complejidad lineal. Sin embargo, no es así en otras aplicaciones como, por ejemplo, la visualización de datos médicos, en la cual se tienen enormes cantidades de datos por visualizar y los métodos de encajamiento de cubos y de aplanamiento son los indicados.

Se realizó también la visualización de alta calidad y fuera de línea, utilizando trazo de rayos y generando un video que muestra la animación del fluido con 5000 partículas.

En el planteamiento del problema dado en la sección 1.4 (pág. 10) se requiere que la solución implementada sea capaz de simular el fluido y que se pueda generar la visualización, ambos en línea o fuera de línea. En un principio se planteó el problema de esta manera, pero al unificar los módulos de simulación y visualización en la misma aplicación (sección 4.1, pág. 67) se abrieron las otras dos posibilidades: simulación fuera de línea con visualización en línea y simulación en línea con visualización fuera de línea.

Se modificó con éxito el simulador HSP para cubrir todos los nuevos requerimientos enunciados en el capítulo 3, pág. 31, y en el cual se detalla la manera en que esto se hizo. En este apartado tenemos uno de los mejores aciertos de la tesis, se trata

del modelo de colisión descrito en la sección 3.3, pág. 52. La solución exacta de la ecuación diferencial (3.5), de la pág. 52, resultó sencilla, útil, elegante y eficiente, para el manejo de la colisión.

También se logró acelerar en cierta medida el proceso de simulación, aunque no era esto un requerimiento para la tesis, mediante el cambio de las ventanas de suavizado (sección 3.1, pág. 32) y el uso de una estructura de datos para la búsqueda eficiente de vecinos (sección 2.1.3, pág. 16). La velocidad de simulación es aceptable para un equipo con un procesador y dos núcleos, es decir, sin usar paralelismo o GPUs.

Esto se afirma en base a las pruebas efectuadas, cuyos resultados se muestran en la sección 3.7. De las tablas de tiempos 3.5 (pág. 63) y 3.7 (pág. 65), observamos que se logra una simulación en tiempo real en el equipo utilizado, con 200 partículas. Si se simula con 300 partículas, la simulación se percibe como en *cámara lenta*, a la mitad de la velocidad real de las partículas, lo cual ya no resulta cómodo para interactuar con la simulación, por ser lenta.

De hecho, durante las pruebas con la aplicación, se notó que se pudo interactuar cómodamente hasta un punto intermedio entre la velocidad real y la mitad de ella, esto es, hasta una velocidad de simulación de 0.65 veces la velocidad real. Esto ocurre con alrededor de 250 partículas, con lo cual los 90 segundos de simulación se efectúan en 139 segundos de cómputo. En estas circunstancias, el efecto de *cámara lenta* no es tan pronunciado como en el caso de 300 partículas y por ello la interacción es relativamente menos incómoda.

5.1. Trabajo a futuro

Como sugerencias para desarrollar en el futuro tenemos las siguientes ideas:

- Optimización del simulador utilizando GPUs.
- Diseño de contenedores con formas diversas. Lo ideal sería modelar cualquier forma, ya sea como malla de polígonos o con operaciones booleanas, y usarla como contenedor.
- Implementación eficiente del método de encajamiento de esferas (sección 4.4, pág. 76), utilizando GPUs.
- Diseño de un sensor que visualice gráficamente un campo vectorial, ya que en este trabajo el sensor visualiza únicamente campos escalares (sección 4.4.1, pág. 78).

Durante el desarrollo de la tesis, identificamos algunas características presentes en otros trabajos, que sería útil agregar a la aplicación:

- Simular la interacción entre dos o más fluidos distintos, como en [10].
- Simular objetos deformables como si fuesen fluidos, como en [8].
- Simular cambios de fase, es decir, fluido que se solidifica o sólido que se derrite, como en [9].
- Implementar un trazador de rayos eficiente en GPUs, como en [41] o modificar POVray para trazar superficies definidas por el campo escalar de Zhu y Bridson [2].
- Utilizar dicho campo escalar en las implementaciones de esta tesis, las cuales se describen en las secciones 4.3 (pág. 74) y 4.4, (pág. 76).

Apéndice A

Integración numérica: Leap Frog

En los simuladores usualmente se necesita resolver ecuaciones diferenciales ordinarias de segundo orden. Un ejemplo de ellas lo constituyen las ecuaciones de movimiento de una partícula $\mathbf{p}$ de masa m, dentro de un campo de fuerzas general $\mathbf{f}$, de la forma:

$$m\frac{d^2\mathbf{p}}{dt^2} = \mathbf{f}(\frac{d\mathbf{p}}{dt}, \mathbf{p}, t)$$

Un método sencillo y ampliamente utilizado para integrar este tipo de ecuaciones diferenciales, especialmente para sistemas dinámicos, es el denominado *salto de rana* (*leap frog*). Consiste en aproximar la trayectoria de la partícula $\mathbf{p}(t)$ en el espacio, mediante pequeños segmentos de recta, siendo cada segmento el avance de la partícula de un punto $\mathbf{p}_i = \mathbf{p}(t_i)$ al siguiente $\mathbf{p}_{i+1} = \mathbf{p}(t_{i+1})$, a una cierta velocidad constante durante un intervalo pequeño de tiempo $\delta t = t_{i+1} - t_i$.

La diferencia con otros métodos similares es que el valor de velocidad utilizado no corresponde a la velocidad de la partícula en el tiempo t_i, es decir $\mathbf{v}_i = \mathbf{v}(t_i)$, sino que se utiliza la velocidad en el tiempo $t_{i+\frac{1}{2}}$, esto es $\mathbf{v}_{i+\frac{1}{2}} = \mathbf{v}(t_{i+\frac{1}{2}})$.

Las ecuaciones del método son[1]:

$$\begin{aligned} \mathbf{p}_{i+1} &= \mathbf{p}_i + \mathbf{v}_{i+\frac{1}{2}}\delta t \\ \mathbf{v}_{i+\frac{1}{2}} &= \mathbf{v}_{i-\frac{1}{2}} + \mathbf{a}_i\delta t \end{aligned}$$

donde se requieren como condiciones iniciales los valores $\mathbf{p}_0$ y $\mathbf{v}_{-\frac{1}{2}}$.

Notemos que normalmente los problemas de movimiento vienen con condiciones iniciales $\mathbf{p}_0$ y $\mathbf{v}_0$, por lo cual es necesario calcular la velocidad inicial $\mathbf{v}_{-\frac{1}{2}}$ mediante algún otro esquema de integración, como por ejemplo el de Euler. Las sucesivas posiciones $\mathbf{p}_i$ y velocidades $\mathbf{v}_{i+\frac{1}{2}}$ se van calculando a intervalos o saltos de longitud δt, pero las velocidades están desfasadas un tiempo $\frac{\delta t}{2}$ respecto a las posiciones.

[1] http://en.wikipedia.org/wiki/Leapfrog_integration

Al respecto del método de integración *salto de rana*, es interesante mencionar un comentario del libro [42] en la página 192:

> *Parece obvio que un buen programa de simulación requiere un buen algoritmo para integrar las ecuaciones en las cuales se fundamenta éste. A primera vista uno puede pensar que la velocidad del algoritmo de integración numérica es de la mayor importancia, pero usualmente no es el caso, ya que el tiempo empleado en integrar las ecuaciones de movimiento es pequeño en comparación con el tiempo necesario para buscar las partículas interactuantes.*

También se menciona que el método tiene las siguientes propiedades:

- Reversibilidad temporal
- Conservación de la energía
- Precisión para pasos de integración grandes

El método es reversible en el tiempo, esto significa que uno puede avanzar la partícula k pasos hacia adelante, usando un cierto valor de δt, y luego es posible aplicar otros k pasos hacia atrás, usando ahora $-\delta t$ y la partícula regresará a su posición inicial.

Con conservación de la energía nos estamos refiriendo a que la partícula no gana o pierde energía mientras se mueve en un campo de fuerzas conservativo, como ocurre con otros métodos de integración (por ejemplo el de Euler), que conservan la energía durante periodos de tiempo cortos, pero no lo hacen a largo plazo.

La precisión[2] para pasos de tiempo grandes es importante en la reducción del cálculo necesario para cada unidad de tiempo de simulación.

[2]Se ha traducido el término *accuracy* (exactitud) como precisión, utilizando el hecho de que en el Diccionario de la Real Academia Española se los puede considerar como sinónimos.

Apéndice B

Método de descenso por gradiente

El *método de descenso por gradiente* (*MDG*) es un algoritmo de optimización de primer orden[1], utilizado para encontrar un mínimo local de una función escalar $f : \mathbb{R}^n \Rightarrow \mathbb{R}$, para alguna dimensión n.

Se basa en la observación de que si la función $f(\mathbf{x})$ está definida y es derivable en una vecindad del punto $\mathbf{p}$, entonces el valor de $f(\mathbf{x})$ se reduce más rápido si uno se desplaza a partir de $\mathbf{p}$ en la dirección del gradiente negativo de f en $\mathbf{p}$, es decir, $\nabla f(\mathbf{p})$. Entonces, si $\mathbf{q} = \mathbf{p} - \alpha\nabla f(\mathbf{p})$, para algún α suficientemente pequeño, se tiene que $f(\mathbf{q}) \leq f(\mathbf{p})$.

El algoritmo consiste en tomar una aproximación inicial $\mathbf{p}_0$, un candidato a ser mínimo local de f, y se calcula la sucesión:

$$\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \cdots, \mathbf{p}_k$$

mediante la relación de recurrencia:

$$\mathbf{p}_{i+1} = \mathbf{p}_i - \alpha\nabla f(\mathbf{p}_i)$$

con lo cual tendremos:

$$f(\mathbf{p}_1) \geq f(\mathbf{p}_2) \geq f(\mathbf{p}_3) \geq \cdots \geq f(\mathbf{p}_k)$$

La relación de recurrencia se aplica un número finito de veces k o hasta que se cumple una condición de paro, como por ejemplo, que la diferencia entre elementos consecutivos de la sucesión, $\mathbf{p}_j$ y $\mathbf{p}_{j+1}$, sea menor que algún valor $\epsilon > 0$ y suficientemente pequeño. Si se quiere encontrar un máximo local, basta con avanzar en la dirección positiva del gradiente, en cuyo caso se le conoce como *método de ascenso por gradiente*.

[1] http://en.wikipedia.org/wiki/Gradient_descent

Bibliografía

[1] James F. Blinn. A generalization of algebraic surface drawing. *SIGGRAPH Comput. Graph.*, 16:273–, July 1982.

[2] Yongning Zhu and Robert Bridson. Animating sand as a fluid. *ACM Trans. Graph.*, 24(3):965–972, 2005.

[3] Timothy S. Newman and Hong Yi. A survey of the marching cubes algorithm. *Computers & Graphics*, 30(5):854 – 879, 2006.

[4] Fernando García Arreguín. Objeto deformable inmerso en un fluido. Master's thesis, CINVESTAV, 2007.

[5] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Symposium on Computer Animation*, pages 154–159, 2003.

[6] Matthias Müller, Simon Schirm, and Matthias Teschner. Interactive blood simulation for virtual surgery based on smoothed particle hydrodynamics. *Technology and Health Care*, 12:25–31, April 2004.

[7] Matthias Müller, Simon Schirm, Matthias Teschner, Bruno Heidelberger, and Markus H. Gross. Interaction of fluids with deformable solids. *Journal of Visualization and Computer Animation*, 15(3-4):159–171, 2004.

[8] Simon Clavet, Philippe Beaudoin, and Pierre Poulin. Particle-based viscoleastic fluid simulation. In *Symposium on Computer Animation*, pages 219–228, 2005.

[9] Richard Keiser, Bart Adams, Dominique Gasser, Paolo Bazzi, Philip Dutré, and Markus Gross. A unified lagrangian approach to solid-fluid animation. In *Symposium on Point-Based Graphics*, pages 125–148, 2005.

[10] Matthias Müller, Barbara Solenthaler, Richard Keiser, and Markus Gross. Particle-based fluid-fluid interaction. In *Symposium on Computer Animation*, pages 237–243, 2005.

[11] Matthias Müller-Fischer and Robert Bridson. Fluid simulation course notes. Symposium on Computer Animation/SIGGRAPH, 2007. http://www.cs.ubc.ca/ rbridson/fluidsimulation/.

[12] Matthias Müller. Fast and robust tracking of fluid surfaces. In Dieter W. Fellner and Stephen Spencer, editors, *Symposium on Computer Animation*, pages 237–245. ACM, 2009.

[13] Toon Lenaerts, Bart Adams, and Philip Dutré. Porous flow in particle-based fluid simulations. *ACM Trans. Graph.*, 27(3), 2008.

[14] Christopher Wojtan, Nils Thürey, Markus H. Gross, and Greg Turk. Deforming meshes that split and merge. *ACM Trans. Graph.*, 28(3), 2009.

[15] Brent Warren Williams. Fluid surface reconstruction from particles. Master's thesis, The University of British Columbia, 2008.

[16] Ilya D. Rosenberg and Ken Birdwell. Real-time particle isosurface extraction. In Eric Haines and Morgan McGuire, editors, *SI3D*, pages 35–43. ACM, 2008.

[17] Nick Foster and Ronald Fedkiw. Practical animation of liquids. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 23–30, New York, NY, USA, 2001. ACM.

[18] Simon Premoze, Tolga Tasdizen, James Bigler, Aaron Lefohn, and Ross T. Whitaker. Particle-based simulation of fluids. *Computer Graphics Forum*, 22:401–410, September 2003.

[19] Douglas Enright, Stephen Marschner, and Ronald Fedkiw. Animation and rendering of complex water surfaces. *ACM Trans. Graph.*, 21:736–744, July 2002.

[20] Jonathan M. Cohen, Sarah Tariq, and Simon Green. Interactive fluid-particle simulation using translating eulerian grids. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, I3D '10, pages 15–22, New York, NY, USA, 2010. ACM.

[21] P. Lancaster and K. Salkauskas. Surfaces generated by moving least squares methods. *American Mathematical Society: Math. of Comp.*, 37(155), 1981.

[22] Paul W. Cleary, Soon Hyoung Pyo, Mahesh Prakash, and Bon Ki Koo. Bubbling and frothing liquids. *ACM Trans. Graph.*, 26, July 2007.

[23] Cem Yuksel, Donald H. House, and John Keyser. Wave particles. *ACM Trans. Graph.*, 26(3):99, 2007.

[24] Nils Thürey, Matthias Müller-Fischer, Simon Schirm, and Markus H. Gross. Real-time breakingwaves for shallow water simulations. In Marc Alexa, Steven J. Gortler, and Tao Ju, editors, *Pacific Conference on Computer Graphics and Applications*, pages 39–46. IEEE Computer Society, 2007.

[25] Jos Stam. Stable fluids. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '99, pages 121–128, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[26] Matthias Müller, Jos Stam, Doug James, and Nils Thürey. Real time physics class notes. Symposium on Computer Animation/SIGGRAPH, 2008. http://www.matthiasmueller.info/realtimephysics/index.html.

[27] Roland Angst, Nils Thürey, Mario Botsch, and Markus H. Gross. Robust and efficient wave simulations on deforming meshes. *Comput. Graph. Forum*, 27(7):1895–1900, 2008.

[28] Cem Yuksel and John Keyser. Fast real-time caustics from height fields. *The Visual Computer*, 25(5-7):559–564, 2009.

[29] J. J. Monaghan. Simulating free surface flows with sph. *Journal of Computational Physics*, 110(2):399 – 406, 1994.

[30] Henrik Wann Jensen and Per Christensen. High quality rendering using ray tracing and photon mapping. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[31] James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes, and Richard F. Philips. *Introducción a la graficación por computador*. Addison-Wesley Iberoamericana, 1996. ISBN: 0-201-62599-7.

[32] Aaron Knoll, Younis Hijazi, and Hans Hagen. Fast ray tracing of arbitrary implicit surfaces with interval and affine arithmetic.

[33] Andrei Sherstyuk. Fast ray tracing of implicit surfaces. *Computer Graphics Forum*, 18:139–147, June 1999.

[34] Donald Hearn and M. Pauline Baker. *Gráficos por computadora con OpenGL*. Prentice Hall, 3a. edition, Noviembre 2006. ISBN: 8420539805 ISBN-13: 9788420539805.

[35] Olivier Gourmel, Anthony Pajot, Mathias Paulin, Loïc Barthe, and Pierre Poulin. Fitted bvh for fast raytracing of metaballs. *Computer Graphics Forum*, 29:281–288, May 2010.

[36] María del Rosario Martínez Gómez. Sistema de visualización para un tomógrafo de rayos x. Master's thesis, CINVESTAV, 2006.

[37] Hanspeter Pfister, Mattias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives, 2000.

[38] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21:163–169, August 1987.

[39] Timothy S. Newman, J. Brad Byrd, Pavan Emani, Amit Narayanan, and Abouzar Dastmalchi. High performance simd marching cubes isosurface extraction on commodity computers. *Computers & Graphics*, 28(2):213 – 233, 2004.

[40] Christer Ericson. *Real-Time Collision Detection.* Morgan Kaufmann, 2005. ISBN: 1-55860-732-3.

[41] Olivier Gourmel, Anthony Pajot, Loïc Barthe, Mathias Paulin, and Pierre Poulin. Bvh for efficient raytracing of dynamic metaballs on gpu. In *SIGGRAPH Talks.* ACM, 2009.

[42] Martin O. Steinhauser. *Computational Multiscale Modeling of Fluids and Solids.* Springer-Verlag, 2008. ISBN-13: 978-3-540-75116-8.

Los abajo firmantes, integrantes del jurado para el examen de grado que sustentará el **Lic. Andrés Cortés Dávalos**, declaramos que hemos revisado la tesis titulada:

VISUALIZACIÓN DE FLUIDOS GENERADOS POR EL MÉTODO HSP

Y consideramos que cumple con los requisitos para obtener el Grado de Maestro en Ciencias de la Computación.

Atentamente,

Dr. Carlos Artemio Coello Coello

Dr. Debrup Chakraborty

Dr. Luis Gerardo de la Fraga