Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Simulación en Tiempo Real de la Interacción Agua-Sólido con SPH

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Santiago Daniel Pivetta

Director: Dr. Claudio Delrieux

Buenos Aires, 2017

SIMULACIÓN EN TIEMPO REAL DE LA INTERACCIÓN AGUA-SÓLIDO CON SPH

Es esta tesis presentamos el problema de la simulación de fluidos, en particular agua, y su interacción con cuerpos rígidos en tiempos interactivos. El problema es abordado desde el punto de vista de computación gráfica, por lo que no se busca una solución numéricamente precisa, sino gráficamente convincente. Exploramos Smoothed Particles Hydrodinamycs, un enfoque Lagrangiano que permite aproximar de manera exitosa las ecuaciones de Navie-Stokes. Aprovechando el poder de cómputo de las actuales GPUs, presentamos una implementación basada en OpenCL, que combina distintas técnicas basadas en SPH, y estudiamos el desempeño en tiempo real de un método que permite representar sólidos con menor cantidad de partículas que otras estrategias habituales en SPH. Nuestros resultados muestran que su aplicación en dominios donde se requieren tiempos interactivos, tales como juegos o simulaciones generadas por computadoras, es posible.

In this theses we present the problem of fluid simulation, particularly water, and its interaction in real time with rigid bodies. The problem is addressed from the computer graphics point of view, where we are not interested in precise numeric solutions, but rather visually convincing solutions. We explore a Lagrangian method called Smoothed Particles Hydrodynamics, which allows to successfully aproximate the Navier-Stokes equations. Taking advantage of the computing power of nowadays GPUs, we present an OpenCL based implementation, combining different techniques based on SPH for simulating fluids, focusing on a method that allows to represent rigid bodies using fewer particles than other common strategies for SPH. Our results show that the application of these methods to real time scenarios such as games is perfectly possible.

Palabras claves: Simulación real-time, SPH, interacción cuerpos rígidos.

AGRADECIMIENTOS

Quiero agradecer, en primer lugar, a mi familia, sin cuyo apoyo durante todos estos años, no podría estar hoy donde estoy. A Carla, no sólo por ayudarme con algunos gráficos en este trabajo, sino que estuvo a mi lado siempre, animándome a terminar a pesar del lento avance. Por último, a mis amigos, cuya insistencia me alentó a perseverar hasta el final.

A Memi.

Índice general

1. INTRODUCCIÓN GENERAL

1.1. Introducción a la simulación de fluídos

Los fluídos, en particular los líquidos, son parte de nuestra vida cotidiana. Bebemos agua, vemos un río fluir, observamos la lluvia. El estudio de estos fenómenos ha despertado siempre la curiosidad del hombre. En particular, con el advenimiento de las computadoras durante el siglo XX, y su constante crecimiento en poder de cómputo, uno de los mayores intereses es poder simular dichos fenómenos con diversos fines, ya sean científicos o recreativos.

El presente trabajo se centra en la simulación de fluídos *incompresibles*. Para ello, el primer paso es contar con un modelo matemático que represente la física de los fluídos incompresibles. Durante el siglo XIX los trabajos de *Claude-Louis Navier* y *George Gabriel Stokes* permitieron desarrollar las ecuaciones fundamentales, hoy conocidas como ecuaciones de *Navier-Stokes* [CM90].

Si bien las ecuaciones de *Navier-Stokes* describen con precisión el comportamiento de los fluídos, su formulación (ecuaciones diferenciales parciales) no permiten hallar soluciones analíticas, sino que es necesario utilizar métodos numéricos que permitan obtener, con mayor o menor medida de precisión, soluciones que las satisfagan. Según el objetivo de la simulación, podemos dividir el universo de métodos y técnicas en dos grandes grupos.

Computer Fluids Dynamics (CDF) es el área de investigación que estudia métodos numéricos que permiten obtener soluciones de alta precisión, capaces de obtener resultados suficientemente detallados para permitir el modelado de fluídos y su interacción con objetos tales como aviones, barcos, y construcciones. Estos métodos, sin embargo, suelen tener implementaciones complejas y de un alto costo computacional (ver Fig 1.1).

El área de *Computer Graphics* estudia métodos que permitan obtener simulaciones que sean visualmente convincentes, aunque físicamente no sean del todo realistas. En estos casos, es posible formular las ecuaciones tomando ciertas licencias matemáticas y físicas, de manera tal de poder lograr simular fluídos de manera visualmente convincente [Bri15]. Este tipo de métodos son de particular interés en el cine, videojuegos, o cualquier otro tipo de simulación [MST03] que requiera ser macroscopicamente convincente.

En computación gráfica, las alternativas más difundidas son los métodos *Eulerianos*, basados en grillas, y los *Lagrangianos*, basados en partículas. Los métodos basados en grillas discretizan el espacio en celdas de tamaño fijo, sobre las cuales se mantiene información (parámetros físicos) del fluído que pasa por ellas (ver Fig. 1.2). Los métodos Lagrangianos discretizan la masa del fluído en unidades denominadas partículas [Ree83], cuya posición y velocidad evoluciona con el tiempo. En particular, en este trabajo nos centraremos lograr una simulación en tiempo real, es decir, que la misma pueda ser computada en tiempos interactivos. Uno de los trabajos fundacionales sobre simulación de fluídos en tiempo real fue presentado por Stam [Sta99], utilizando un método basado en grillas.

Uno de los métodos *Langrangianos* más difundidos para la simulación en tiempo real de fluídos, tales como agua, es el de *Smoothed Particles Hydrodynamics* (hidrodinámica de partículas suavizada), o *SPH*. Este método modela el fluído como compuesto por un conjunto de partículas. Cada partícula representa una porción de masa del fluído, y tiene asociados ciertos valores físicos de la simulación, como densidad, presión y velocidad (ver

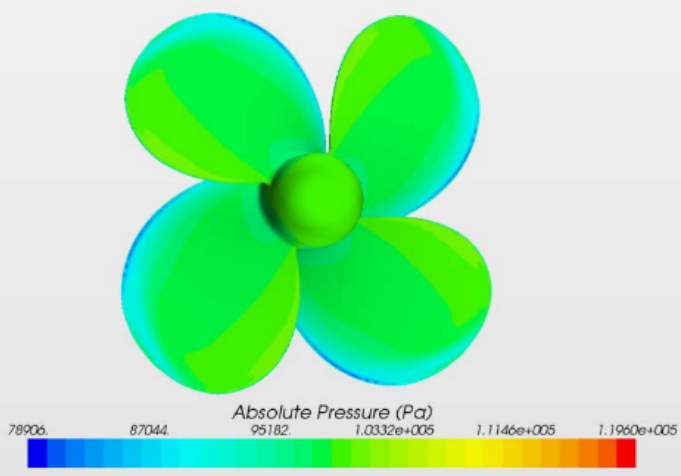

Fig. 1.1: Las técnicas utilizadas en CFD permiten, por ejemplo, simular la presión ejercida por el agua sobre las paletas de una turbina con gran precisión [Mau13].

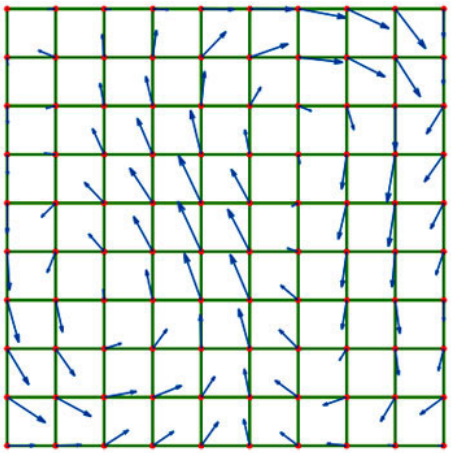

Fig. 1.2: En los métodos Eulerianos, el espacio se divide en celdas. Cada celda posee el valor de densidad, que varía según el campo vectorial de velocidades.

Fig. 1.3). *SPH* surgió en el ámbito de la astrofísica [Mon92], pero fue bien adaptado para la simulación de fluidos [MCG03]. *SPH* presenta ciertas ventajas que lo hacen atractivo para su implementación en sistemas interactivos. En particular, no es necesario hacer un seguimiento de la superficie libre, ya que las mismas partículas representan la masa del fluido, que a su vez garantizan la conservación de masa (lo cual no es trivial en sistemas de grillas).

Si queremos obtener una simulación aún más convincente, tenemos que tener en cuenta la interacción del fluido con el resto de la escena. En particular, nos interesa la interacción con cuerpos rígidos. Podemos considerar dos tipos de interacción: *one-way coupling* cuando el objeto es afectado por el fluído, pero el fluído no es perturbado, y *two-way coupling* (o *weak-coupling*) cuando el fluido también es afectado por el cuerpo rígido [Bri15] (ver Fig. 1.4). Para el caso de SPH, existen varias técnicas para simular estos efectos (ver [Har+07] y [Ihm+10]), aunque no se cuenta con una solución definitiva que sea gráficamente satisfactoria y físicamente correcta. En este trabajo se presentará una implementación

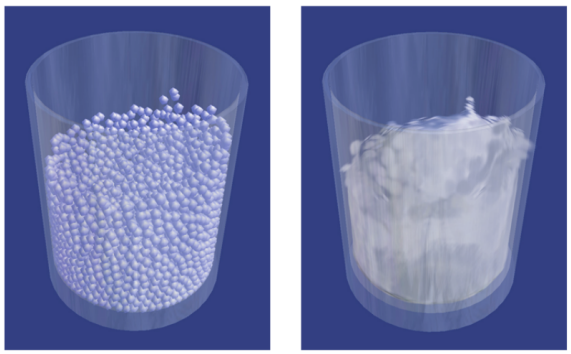

Fig. 1.3: El fluido es representado como partículas [MCG03]. A partir de las mismas, se puede obtener un rendering de la superficie.

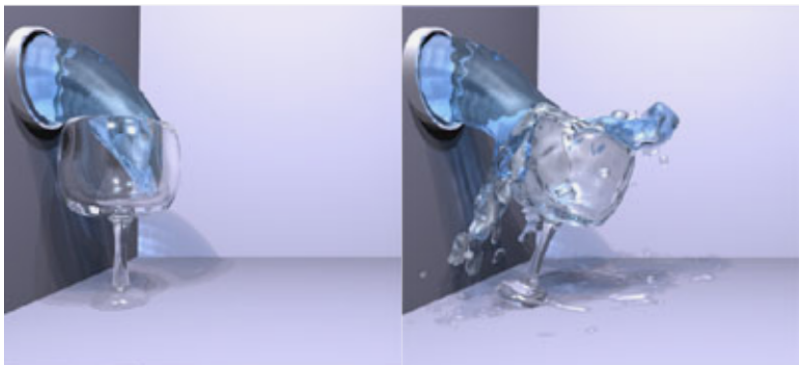

Fig. 1.4: two-way coupling utilizando partículas [Har+07]

basada en simular cuerpos rígidos con partículas [Aki+12].

1.2. Motivación y objetivos

Los objetivos de este trabajo de tesis son:

- Investigar los métodos existentes para la simualción de fluídos basados en SPH, en especial teniendo en cuenta la interacción con sólidos.
- Lograr una implementación eficiente de los mismos en GPU, utilizando OpenCL, la cual permita ejecutar la simulación en tiempo real.

1.3. Desarrollo

El presente trabajo se organiza de la siguiente manera:

- **Capítulo 2**: Se presenta una introducción a las ecuaciones de *Navier-Stokes* para fluidos incompresibles. Luego presentamos la base teórica de *SPH*.
- **Capítulo 3**: En el capítulo 3 veremos cómo se estiman las ecuaciones de *Navier-Stokes* utilizando *SPH*. Veremos una variación del método original, llamada *PCISPH: Predictive-Corrective Incompressible SPH*, que presenta ciertas mejoras respecto a la *incompresibilidad*.

- **Capítulo 4**: Presentamos algunas estrategias utilizadas en SPH para el manejo de colisión con cuerpos rígidos. Finalmente mostramos el método que utilizamos, y como se adapta a la implementación presentada en el capítulo 3.
- **Capítulo 5**: En el capítulo 5 mostramos algunas de las estrategias tradicioanles para la renderización de fluídos, para luego describir la técnica utilizada en este trabajo: *Screen Space Fluid Rendering*.
- **Capítulo 6**: En el capítulo 6 haremos una breve introducción a la programación sobre GPUs, y las dos herramientas más utilizadas: CUDA y OpenCL, para luego introducir algunos de los conceptos básicos de OpenCL.
- **Capítulo 7**: En este capítulo mostraremos los resultados obtenidos con nuestra implementación, asi como también algunos detalles de implementación que hacen al trabajo.
- **Capítulo 8**: Finalmente en el capítulo 8 presentaremos nuestras conclusiones y posibles líneas de trabajo futuro.

2. INTRODUCCIÓN TEÓRICA: MECÁNICA DE LOS FLUIDOS Y SPH

2.1. Ecuaciones de Navier-Stokes

Durante el siglo XIX, el aporte de *Claude-Louis Navier* primero, y luego continuado por *Sir George Gabriel Stokes*, sentó las bases para el estudio de la mecánica de los fluidos: las ecuaciones de *Navier-Stokes*. Dichas ecuaciones han sido utilizadas desde entonces para estudiar fenómenos variados y a diferentes escalas: el comportamiento de océanos, galaxias, o el comportamiento del flujo del aire alrededor del ala de un avión.

En este trabajo nos concentraremos en la simulación de fluidos *incompresibles*. Para este caso, tenemos el siguiente par de ecuaciones diferenciales

$$\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u} \cdot \nabla \boldsymbol{u} + \frac{1}{\rho} \nabla p = \boldsymbol{g} + \nu \nabla \cdot \nabla \boldsymbol{u}\,, \tag{2.1}$$

$$\nabla \cdot \boldsymbol{u} = 0\,, \tag{2.2}$$

donde

- $\boldsymbol{u}$: campo de velocidades del fluido.
- $\boldsymbol{g}$: aceleración gravitatoria.
- ρ: densidad del fluido.
- p: campo de presión.
- ν: viscocidad cinemática.

Daremos una introducción intuitiva al significado de ambas ecuaciones. Se recomienda al lector interesado leer el trabajo de Chorin y Mardsen [CM90], asi como también [Bri15], donde se aborda el tema desde el punto de vista de computación gráfica, y de donde tomamos la derivación de las ecuaciones que se explican a continuación.

2.1.1. Definiciones

A continuación se repasan las definiciones de los elementos del cálculo vectorial básicos que se utilizarán en el desarrollo tanto de las ecuaciones de *Navier-Stokes*, asi como también en *SPH*. Vamos a asumir que siempre estamos en un espacio tridimensional de coordenadas Cartesianas.

Gradiente

Sea $f : \mathbb{R}^3 \to \mathbb{R}$ un campo escalar, se define el gradiente ∇f como

$$\nabla f = \frac{\partial f}{\partial x}\boldsymbol{i} + \frac{\partial f}{\partial y}\boldsymbol{j} + \frac{\partial f}{\partial z}\boldsymbol{k}\,, \tag{2.3}$$

donde $\boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}$ son los vectores unitarios y ortogonales del espacio de coordenadas.

Divergencia

Sea $\boldsymbol{u} : \mathbb{R}^3 \to \mathbb{R}^3$ un campo vectorial, se define el operador diferencial *divergencia* $\nabla \cdot \boldsymbol{u}$ como

$$\nabla \cdot \boldsymbol{u} = \frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} + \frac{\partial u_z}{\partial z} \,. \tag{2.4}$$

Laplaciano

Sea $f : \mathbb{R}^3 \to \mathbb{R}$ un campo escalar, se define el operador *Laplaciano* $\nabla^2 f$ como

$$\nabla^2 f = \nabla \cdot \nabla f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} + \frac{\partial^2 f}{\partial z^2} \,. \tag{2.5}$$

Dicha formulación es válida para un sistema cartesiano de coordenadas.

Derivada material

Sea $\varphi(\boldsymbol{r}, t)$ un campo escalar, donde t representa el tiempo y $\boldsymbol{r} \in \mathbb{R}^3$ una posición en el espacio Euclídeo, se define la derivada material como

$$\frac{Dq}{Dt} = \frac{\partial q}{\partial t} + \nabla q \cdot \frac{d\boldsymbol{r}}{dt} \,. \tag{2.6}$$

2.1.2. Ecuación de *momentum*

La ecuación 2.1 es conocida como ecuación de *momentum*. Indica cómo se mueve el fluido cuando es sometido a distintas fuerzas, internas y externas. De hecho, es posible pensar en dicha ecuación como una reescritura de la segunda ley de Newton $m\boldsymbol{a} = \boldsymbol{F}$. Consideremos una porción de fluido (una partícula) de masa m y volúmen V. Podemos escribir entonces

$$m\frac{d\boldsymbol{u}}{dt} = \boldsymbol{F} \,. \tag{2.7}$$

Dado que $\boldsymbol{u}$ es un campo de velocidades, y que es función de t y $\boldsymbol{r}$ (posición en el espacio), aplicando la regla de la cadena obtenemos

$$m\frac{\partial \boldsymbol{u}}{\partial t} + \nabla \boldsymbol{u} \cdot \boldsymbol{u} = \boldsymbol{F} \,. \tag{2.8}$$

Según la definición de derivada material, podemos reescribir 2.8 como

$$m\frac{D\boldsymbol{u}}{Dt} = \boldsymbol{F} \,. \tag{2.9}$$

El término $\boldsymbol{F}$ se compone de la suma de fuerzas que actúan sobre el fluido, que son

- $m\boldsymbol{g}$: La acción de la gravedad.
- $\boldsymbol{F}_{visc}$: Las fuerzas debidas a la viscosidad del fluido.
- $\boldsymbol{F}_{pres}$: Las fuerzas debidas a la presión interna del fluido.

- $\boldsymbol{F}_{ext}$: Las fuerzas externas que actúan sobre el fluido, por ejemplo colisiones con cuerpos rígidos.

Considerando esto, podemos reescribir la ecuación (2.9), obteniendo

$$m\frac{D\boldsymbol{u}}{Dt} = m\boldsymbol{g} + \boldsymbol{F}_{visc} + \boldsymbol{F}_{press} + \boldsymbol{F}_{ext}\,. \tag{2.10}$$

La fuerza de presión $\boldsymbol{F}_{press}$ se manifiesta cuando existe un desbalance de la presión alrededor de la partícula. Es decir, la fuerza $\boldsymbol{F}_{press}$ se manifiesta en la dirección de menor presión, y de manera proporcional a la diferencia de presión neta sobre la partícula. Una primera aproximación es tomar el gradiente de la presión $-\nabla p$ e integrarlo por el volumen V de la partícula que estamos considerando, con lo cual $\boldsymbol{F}_{press}$ se puede expresar como

$$\boldsymbol{F}_{press} = -V\nabla p\,. \tag{2.11}$$

La fuerza de viscosidad es aquella que se manifiesta como una resistencia del fluido a la deformación. Intuitivamente, es una fuerza que tiende a minimizar las diferencias de velocidades entre partículas. El operador diferencial que permite medir la diferencia de una cantidad respecto al promedio alrededor es el *Laplaciano*. Este operador nos sirve para definir la fuerza de viscosidad, que también se ve afectada por un *coeficiente de viscosidad dinámica* η[1]. Aproximando la integral por V como en el caso anterior, obtenemos

$$\boldsymbol{F}_{visc} = V\eta\nabla\cdot\nabla\boldsymbol{u}\,. \tag{2.12}$$

Reemplazando (2.11) y (2.12) en (2.10), y dividiendo por el volúmen de la partícula V, obtenemos

$$\rho\frac{D\boldsymbol{u}}{Dt} = \rho\boldsymbol{g} + \eta\nabla\cdot\nabla\boldsymbol{u} - \nabla p + \boldsymbol{F}_{ext}\,, \tag{2.13}$$

donde la densidad es $\rho = \frac{m}{V}$. Finalmente, dividiendo por ρ

$$\frac{D\boldsymbol{u}}{Dt} = \boldsymbol{g} + \nu\nabla\cdot\nabla\boldsymbol{u} - \frac{1}{\rho}\nabla p + \frac{1}{\rho}\boldsymbol{F}_{ext}\,, \tag{2.14}$$

con $\nu = \frac{\eta}{\rho}$, denominada *viscosidad cinemática*.

Considerando la definición de derivada material, vemos que hemos arribado[2] a la formulación que habíamos planteado en la ecuación (2.1)[3]. El término $\nu\nabla\cdot\nabla\boldsymbol{u}$ suele denominarse *término de difusión*, mientras que $\boldsymbol{u}\cdot\nabla\boldsymbol{u}$ suele llamarse *término de advección*.

2.1.3. Condición de incompresibilidad

Los fluidos reales cambian su volumen. De hecho, la propagación de ondas sonoras se explica de esta manera. Sin embargo, asumiremos que el fluido a simular es *incompresible*. Esto se traduce en la ecuación (2.2). No daremos una derivación, pues esto supone un desarrollo más complejo del que presentamos en la sección anterior. Nuevamente, al lector interesado, en el apéndice B de [s]e explica el desarrollo completo. Diremos que, intuitivamente, la ecuación (2.2) impone que el campo de velocidades no tenga divergencia. Es decir, que dada una (pequeña) unidad de volúmen, la cantidad de fluido se conserve.

[1] Consideraremos viscosidades constantes

[2] Sin considerar $\boldsymbol{F}_{ext}$

[3] Nuevamente, esta es una deducción intuitiva. Para ver una deducción formal, consultar [] y en menor medida [Bri15]

2.2. SPH: Smoothed Particles Hydrodynamics

Smoothed Particles Hydrodynamics, o SPH, es un método originalmente desarrollado en el campo de la astrofísica para modelar problemas de flujos compresibles. Fue concebido en el año 1977 por Gringold y Monaghan [GM77], mismo año en el que Lucy [Luc77] llegó a resultadods similares.

SPH es un método de interpolación que permite aproximar valores y derivadas de un campo contínuo, utilizando muestras discretas [Mon05]. Estas muestras, o puntos discretos, son llamados *smoothed particles*, los cuales tienen asignadas ciertas magnitudes como masa o velocidad. A su vez, cada partícula tiene asignada ciertas magnitudes del campo contínuo a estimar (depende del problema), como por ejemplo temperatura, presión, etc.

Dentro del campo de computación gráfica, uno de los primeros trabajos utilizando esta técnica fue presentada en [SF95]. Dentro del ámbito de simulación de fluidos, el trabajo de Roy [Roy95], basado en [Mon94], fue uno de los primeros en adaptar SPH para dicho uso. En cuanto a simulación en tiempo real de fluidos, el trabajo de Müller [MST03] fue uno de los pioneros en utilizar SPH.

2.2.1. Interpolador

En *SPH*, la interpolación de una función $A(\boldsymbol{r})$, que es función de una posición en el espacio, se basa en el *interpolador integral* [Mon05]

$$A(\boldsymbol{r}) = \int_{\Omega} A(\hat{\boldsymbol{r}})\delta(\boldsymbol{r} - \hat{\boldsymbol{r}})\, d\hat{\boldsymbol{r}}\,, \tag{2.15}$$

donde $\boldsymbol{r}$ es un punto cualquiera en Ω, y δ es el delta de Dirac, definido como

$$\delta(\boldsymbol{r}) = \begin{cases} \infty & \text{si } |\boldsymbol{r}| = 0 \\ 0 & \text{si } |\boldsymbol{r}| \neq 0\,. \end{cases} \tag{2.16}$$

En la práctica, A puede ser aproximada por A_I, reemplazando δ por una función W denominada *smoothing kernel* de *ancho* o *soporte* h:

$$A_I(\boldsymbol{r}) = \int_{\Omega} A(\hat{\boldsymbol{r}})W(\boldsymbol{r} - \hat{\boldsymbol{r}}, h)\, d\hat{\boldsymbol{r}}\,. \tag{2.17}$$

A su vez, la integral (2.17) puede ser aproximada de manera discreta utilizando un conjunto finito de puntos de interpolación, reemplazando la integral por una sumatoria, y el diferencial $d\hat{\boldsymbol{r}}$ por el volúmen V_j, el cual puede expresarse como la masa m_j dividida por la densidad ρ. Además, A_j es el valor de la función A en $\boldsymbol{r_j}$:

$$A_S(\boldsymbol{r}) = \sum_j A_j \frac{m_j}{\rho} W(\boldsymbol{r} - \boldsymbol{r_j}, h)\,. \tag{2.18}$$

La ecuación (2.18) es la formulación base para SPH, que permite aproximar cualquier campo contínuo en algún punto del espacio. Luego, asumiendo ciertas propiedades para W, entre las cuales se pide diferenciabilidad de primer y segundo orden, podemos obtener el gradiente y el Laplaciano de A:

$$\nabla A_S(\boldsymbol{r}) = \sum_j A_j \frac{m_j}{\rho} \nabla W(\boldsymbol{r} - \boldsymbol{r_j}, h)\,, \tag{2.19}$$

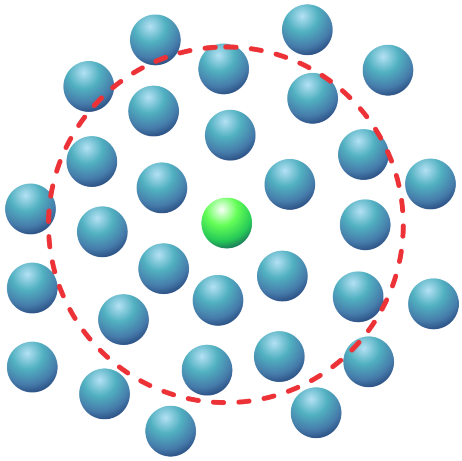

Fig. 2.1: Esquema en 2D para SPH con un kernel W y un vecindario Ω.

$$\nabla^2 A_S(\boldsymbol{r}) = \sum_j A_j \frac{m_j}{\rho} \nabla^2 W(\boldsymbol{r} - \boldsymbol{r_j}, h)\,. \tag{2.20}$$

En [Kel06] puede consultarse un desarrollo riguroso para las ecuaciones (2.19) y (2.20).

2.2.2. Smoothing kernel

Una función W es un kernel apropiado [Mon05] para *SPH* si cumple:

$$\int_\Omega W(\boldsymbol{r}, h)\, d\boldsymbol{r} = 1\,, \tag{2.21}$$

$$\lim_{h\to 0} W(\boldsymbol{r}, h) = \delta(\boldsymbol{r})\,, \tag{2.22}$$

$$W(\boldsymbol{r}, h) >= 0\,, \tag{2.23}$$

con δ el delta de Dirac. Además, si se cumple que

$$W(\boldsymbol{r}, h) = W(-\boldsymbol{r}, h)\,, \tag{2.24}$$

entonces se puede garantizar invarianza respecto a rotaciones del sistema de coordenadas. Si el kernel W cumple (2.21) y (2.25), se puede acotar el error cometido al estimar con (2.18), siendo esta cota $O(h^2)$ [Kel06] (ver Fig. 2.1).

Dependiendo de la aplicación, se debe elegir un kernel apropiado que mejor se ajuste al problema. El kernel Gaussiano fue uno de los primeros usados[GM77], definido como

$$W_{Gaussian}(\boldsymbol{r}, h) = \sigma e^{-q^2}\,, \tag{2.25}$$

donde $q = \frac{||\boldsymbol{r}||}{h}$, y σ se denomina *factor de dimensionalidad*, siendo una constante de ajuste según la dimensión del kernel, para garantizar la condición (2.21). Para el caso de 3 dimensiones, $\sigma = \frac{1}{\pi^{3/2} h^3}$.

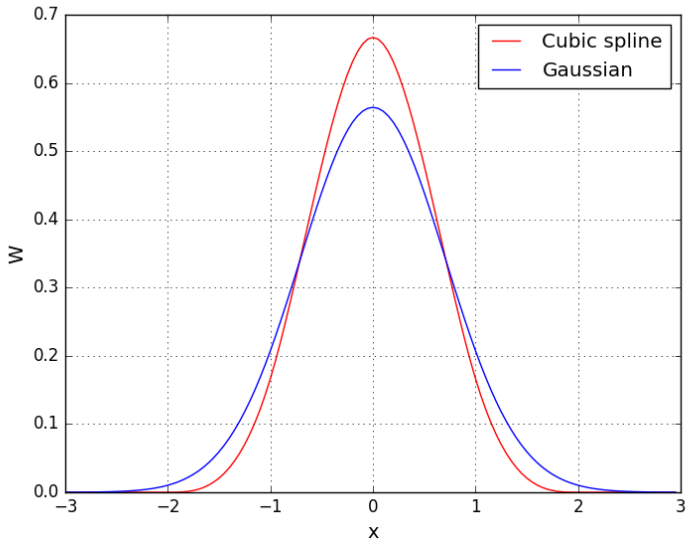

Fig. 2.2: Kernel Gaussiano isotrópico junto con el kernel Spline cúbico, con $h = 1$, ambos en 1D. Se observa que el Spline cúbico es más compacto.

En general, es también deseable que W tenga un radio de soporte limitado o *compacto*: $W(\boldsymbol{r}, h) = 0$ cuando $||\boldsymbol{r}|| > h$. En [Mon05] se propone el uso de M_n splines, de entre los cuales, M_4, llamado *cubic-spline*

$$M_4(\boldsymbol{r}, h) = \sigma \begin{cases} \frac{2}{3} - q^2 - \frac{1}{2}q^3 & \text{si } 0 \leq q < 1 \\ \frac{1}{6}(2-q)^3 & \text{si } 1 \leq q \leq 2 \\ 0 & \text{si } q > 2. \end{cases} \tag{2.26}$$

nuevamente con $q = \frac{||r||}{h}$, y σ el factor de dimensionalidad, siendo, para 3 dimensiones, $\sigma = \frac{3}{2\pi h^3}$

Sin embargo, no hay un consenso sobre cual es el kernel óptimo. Por ejemplo, [BT07] y [SB12] utilizan el kernel *Spline* cúbico (ver Fig. 2.2), mientras que [ZYF10] utilizan un kernel *Spline* de quinto orden. En este trabajo, utilizaremos los kernels propuestos en [MCG03].

3. SPH APLICADO A NAVIER-STOKES

En el capítulo 2 se presentaron las bases de la mecánica de fluidos: las ecuaciones de *Navier-Stokes*. Vimos que el planteo de ecuaciones diferenciales no admite una resolución analítica, por lo que debemos aproximar los resultados con algún método. Para ello presentamos *SPH*, un método basado en partículas que permite estimar el valor de cierta magnitud en un punto del espacio en función del valor de esa magnitud en ciertos puntos discretos conocidos, las partículas.

En este capítulo presentamos la aplicación de *SPH* a las ecuaciones de *Navier-Stokes*, con particular énfasis en soluciones de tiempo real.

3.1. Estimación de densidad

La densidad en la posición de la *i-ésima* partícula será

$$\rho_i = \rho(\boldsymbol{r_i})\,. \tag{3.1}$$

Aplicando lo visto en el capítulo anterior, para aproximar la función ρ en la posición $\boldsymbol{r_i}$, utilizamos la ecuación (2.18)

$$\begin{aligned}
\rho_i &= \rho(\boldsymbol{r_i}) \\
\rho_i &= \sum_j \rho_j \frac{m_j}{\rho_j} W(\boldsymbol{r} - \boldsymbol{r_j}, h) \\
\rho_i &= \sum_j m_j W(\boldsymbol{r} - \boldsymbol{r_j}, h)\,.
\end{aligned} \tag{3.2}$$

3.2. Fuerzas internas

3.2.1. Presión

La estimación de presión es un tema en sí mismo. Una de las maneras más sencillas es derivarla directamente de la ecuación de estado de gases ideales

$$p_i = k\rho_i\,, \tag{3.3}$$

donde k es una constante que depende de la temperatura.

Otra formulación, ampliamente adoptada [Mon94][BT07][LD09][YT13], es la siguiente:

$$p_i = k\left(\left(\frac{\rho_i}{\rho_0}\right)^7 - 1\right)\,, \tag{3.4}$$

donde ρ_0 es la densidad en reposo del fluido. En la práctica, para garantizar la incompresibilidad del fluido, se requiere una constante k grande, y a su vez Δt pequeños para garantizar la estabilidad.

Müller [MST03] utiliza una variante más simple de computar, aunque menos efectiva para garantizar la incompresibilidad

$$p_i = k(\rho_i - \rho_0)\,. \tag{3.5}$$

En general, este tipo de estimación de presión requiere, para garantizar una incompresibilidad aceptable, constantes k grandes, lo cual genera a su vez que no se puedan utilizar Δt muy grandes ya que la simulación se vuelve inestable [Ihm+14]. En nuestra implementación utilizaremos la estimación de presión propuesta en [SP09], que permite garantizar la incompresibilidad de una manera más efectiva, y al mismo tiempo permite Δt mayores, lo cual es mejor si queremos una simulación en tiempo real.

Aceleración

De la ecuación (2.14) tenemos que la aceleración de la *i-esima* partícula debido a la presión es

$$\boldsymbol{a}_i^{press} = -\frac{1}{\rho_i}\nabla p(\boldsymbol{r_i})\,. \tag{3.6}$$

Nuevamente, aplicando la ecuación (2.19), obtenemos

$$\boldsymbol{a}_i^{press} = -\frac{1}{\rho_i}\sum_j p_j \frac{m_j}{\rho_j}\nabla W(\boldsymbol{r_i} - \boldsymbol{r_j}, h)\,. \tag{3.7}$$

Sin embargo, esta fuerza no es simétrica.

Si consideramos la interacción de dos únicas partículas i y j, en general, $p_i \neq p_j$, por lo que

$$m_i \frac{m_j p_j}{\rho_i \rho_j}\nabla W(\boldsymbol{r_i} - \boldsymbol{r_j}, h) \neq m_j \frac{m_i p_i}{\rho_j \rho_i}\nabla W(\boldsymbol{r_j} - \boldsymbol{r_i}, h)\,. \tag{3.8}$$

Esto haría que la ley de Newton *acción-reacción* no se mantenga. Para subsanar esto, [Mon05] muestra que

$$\frac{\nabla p}{\rho} = \nabla\left(\frac{p}{\rho}\right) + \frac{p}{\rho^2}\nabla\rho\,. \tag{3.9}$$

A partir de esa igualdad, propone una reescritura de la ecuación (3.7) que resulta simétrica:

$$\boldsymbol{a}_i^{press} = -\sum_j m_j \left(\frac{p_i}{{\rho_i}^2} + \frac{p_j}{{\rho_j}^2}\right)\nabla W(\boldsymbol{r_i} - \boldsymbol{r_j}, h)\,. \tag{3.10}$$

3.2.2. Viscosidad

De la ecuación (2.14) tenemos que la aceleración de la *i-esima* partícula debido a la viscosidad es

$$\boldsymbol{a}_i^{visc} = \nu\nabla^2\boldsymbol{u}(\boldsymbol{r_i})\,. \tag{3.11}$$

Aplicando la ecuación (2.20), obtenemos

$$\boldsymbol{a}_i^{visc} = \nu\sum_j \boldsymbol{v}_j \frac{m_j}{\rho_j}\nabla^2 W(\boldsymbol{r}_i - \boldsymbol{r}_j, h)\,. \tag{3.12}$$

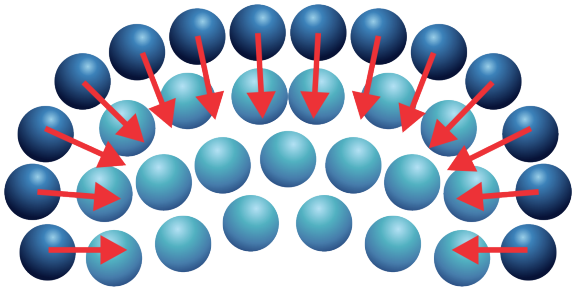

Fig. 3.1: Atracción sobre las moléculas de la superficie libre.

Nuevamente, dado que el campo de velocidad varía de acuerdo a cada partícula, la fuerza resultante no es simétrica. En este caso, existe una manera más fácil de resolver el problema, dado que en realidad, la fuerza debido a la viscosidad sólo depende de la diferencia de velocidades

$$\boldsymbol{a}_i^{visc} = \nu \sum_j (\boldsymbol{v}_j - \boldsymbol{v}_i) \frac{m_j}{\rho_j} \nabla^2 W(\boldsymbol{r}_i - \boldsymbol{r}_j, h)\,. \tag{3.13}$$

3.3. Tensión superficial

La tensión superficial es una fuerza que no hemos tenido en cuenta en la formulación original (2.1), pero podemos incorporarla como una fuerza externa más.

Este fenómeno se debe a las fuerzas cohesivas que existen entre las moléculas del líquido, por ejemplo en el agua. Dentro de la masa del fluido, cada molécula es atraída por sus vecinas en todas las direcciones del espacio, lo que resulta en una fuerza neta nula. Sin embargo, en la superficie libre, la atracción no es igual en todas direcciones, sino que las moléculas son atraídas hacia el interior de la masa del líquido (ver Fig. 3.1).

Existen varias estrategias para simular este fenómeno. Por ejemplo, una de las primeras técnicas presentada en [MST03] se basa en calcular normales para cada partícula y luego definir una magnitud llamada *color field*. La magnitud de la fuerza depende entonces de la curvatura del *color field* en una partícula determinada. Si bien la técnica es suficientemente simple como para aplicarse en un contexto de tiempo real, presenta algunos problemas.

Otras técnicas que intentan mejorar la simulación de tensión superficial son por ejemplo [BT07] y [TM05]. En este trabajo, utilizaremos el método presentado en [AAT13].

3.3.1. Cómputo de tensión superficial

El siguiente modelo de tensión superficial es el propuesto por Akinci en [AAT13]. En la Fig. 3.2 se muestra el efecto del modelo implementado.

Sean dos partículas i y j, se define la fuerza que la partícula j ejerce sobre i debida a la tensión superficial como

$$\boldsymbol{F}_{i\leftarrow j}^{st} = K_{ij} \left(\boldsymbol{F}_{i\leftarrow j}^{cohesion} + \boldsymbol{F}_{i\leftarrow j}^{curvature} \right)\,, \tag{3.14}$$

donde K_{ij} se define como

$$K_{ij} = \frac{2\rho_0}{\rho_i + \rho_j}\,, \tag{3.15}$$

el cual es un factor de corrección para evitar efectos no deseados relacionados con el hecho de que el fenómeno de tensión superficial se da entre moléculas, mientras que una partícula representa una cierta masa de fluido, lo cual puede generar que partículas con pocos vecinos queden aisladas facilmente del resto.

Término de cohesión

La fuerza de cohesión que la partícula j ejerce sobre i se define como

$$\boldsymbol{F}_{i\leftarrow j}^{cohesion} = -\gamma m_i m_j C(||\boldsymbol{r}_i - \boldsymbol{r}_j||)\frac{\boldsymbol{r}_i - \boldsymbol{r}_j}{||\boldsymbol{r}_i - \boldsymbol{r}_j||}\,, \tag{3.16}$$

donde γ es el coeficiente de tensión superficial, y C es un *Spline* definido como

$$C(r,h) = \frac{32}{\pi h^9}\begin{cases}(h-r)^3 r^3 & \text{si } 2r > h \wedge r \leq h \\ 2(h-r)^3 r^3 - \frac{h^6}{64} & \text{si } r > 0 \wedge 2r \leq h \\ 0 & \text{caso contrario}\,.\end{cases} \tag{3.17}$$

Término de curvatura

La tensión superficial puede pensarse en términos de energía. Una molécula en contacto con otra tiene un estado de energía menor a una que no esta en contacto. Las moléculas en el interior del fluído estan en contacto en toda su vecindad, mientras que las moléculas de la superficie libre no, por lo cual poseen un estado mayor de energía. La tensión superficial tiende a minimizar el área de superficie libre [Whi72]. Para garantizar esta minimización es que se introduce el término de curvatura.

La fuerza que la partícula j ejerce sobre la partícula i debido al término de *curvatura* se define como

$$\boldsymbol{F}_{i\leftarrow j}^{curvature} = -\gamma m_i(\boldsymbol{n}_i - \boldsymbol{n}_j)\,, \tag{3.18}$$

donde $\boldsymbol{n_i}$ es la normal de la superficie libre en la *i-ésima* partícula, que se puede aproximar utilizando *SPH* como

$$\boldsymbol{n}_i = s\sum_j \frac{m_j}{\rho_j}\nabla W(\boldsymbol{r}_i - \boldsymbol{r}_j, h)\,, \tag{3.19}$$

con s un factor de corrección para independizar de la escala a las normales calculadas.

Puede verse con cierta facilidad que en las zonas planas, el término de curvatura es $\mathbf{0}$, ya que $\boldsymbol{n}_i - \boldsymbol{n_j} = \mathbf{0}$. Lo mismo para el interior del fluido, donde $\boldsymbol{n}_i \approx \mathbf{0}$ y $\boldsymbol{n}_j \approx \mathbf{0}$. En la Fig. 3.2 se puede observar el efecto que la tensión superficial ejerce sobre un volumen libre.

3.4. Kernels para tiempo real

En la sección 2.2.2 introdujimos el concepto de *smoothing kernel*, y cuáles son las condiciones que debe cumplir. Vimos que existen diversos kernels, según la aplicación. En el presente trabajo utilizaremos los kernels introducidos en [MST03], que, por su sencillez para computarlos en comparación a otros kernels, hace que se adapten bien para ser empleados en tiempo real.

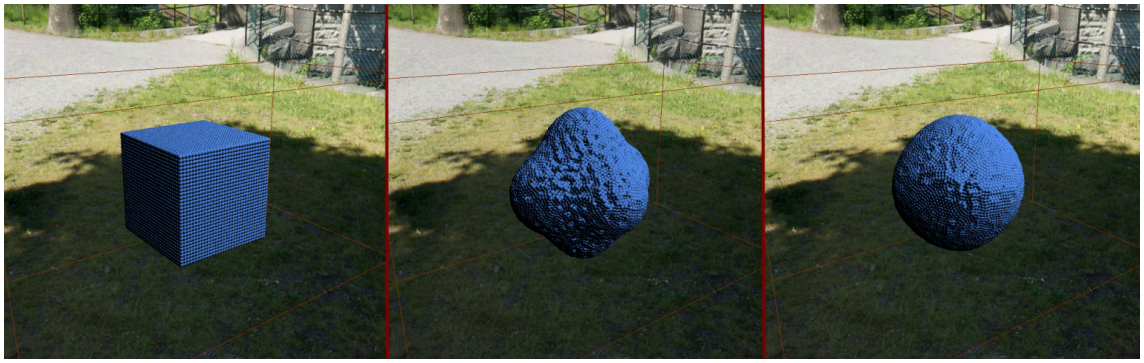

Fig. 3.2: Implementación del modelo en el presente trabajo. El cubo, por efecto de la tensión superficial, se transforma en una esfera. En la escena no hay gravedad.

3.4.1. Poly6 kernel

El siguiente kernel será utilizado para todos los cálculos, a menos que se indique lo contrario. La formulación es la siguiente:

$$W_{poly6}(\boldsymbol{r}, h) = \frac{315}{64\pi h^9} \begin{cases} \left(h^2 - ||\boldsymbol{r}||^2\right)^3 & \text{si } ||\boldsymbol{r}|| \leq h \\ 0 & \text{caso contrario}\,. \end{cases} \tag{3.20}$$

Su gradiente y Laplaciano son

$$\nabla W_{poly6}(\boldsymbol{r}, h) = -\frac{945}{32\pi h^9}\boldsymbol{r} \begin{cases} \left(h^2 - ||\boldsymbol{r}||^2\right)^2 & \text{si } ||\boldsymbol{r}|| \leq h \\ 0 & \text{caso contrario}\,, \end{cases} \tag{3.21}$$

$$\nabla^2 W_{poly6}(\boldsymbol{r}, h) = -\frac{945}{32\pi h^9} \begin{cases} \left(h^2 - ||\boldsymbol{r}||^2\right)\left(3h^2 - 7||\boldsymbol{r}||^2\right) & \text{si } ||\boldsymbol{r}|| \leq h \\ 0 & \text{caso contrario}\,. \end{cases} \tag{3.22}$$

3.4.2. Spiky kernel

Para el cálculo de presión, no podemos utilizar el kernel *poly6* dado que se verifica que

$$\lim_{||\boldsymbol{r}|| \to 0} \nabla W_{poly6}(\boldsymbol{r}, h) = \boldsymbol{0}\,. \tag{3.23}$$

Esto generaría que, a medida que las partículas se acercan, la fuerza de repulsión entre ellas debido a la presión decrezca, generando *clústers* de partículas con facilidad. Para evitar este fenómeno, para el cálculo de presión utilizaremos el *Spiky* kernel

$$W_{spiky}(\boldsymbol{r}, h) = \frac{15}{\pi h^6} \begin{cases} \left(h - ||\boldsymbol{r}||\right)^3 & \text{si } ||\boldsymbol{r}|| \leq h \\ 0 & \text{caso contrario}\,. \end{cases} \tag{3.24}$$

Su gradiente y Laplaciano son

$$\nabla W_{spiky}(\boldsymbol{r}, h) = -\frac{45}{\pi h^6}\frac{\boldsymbol{r}}{||\boldsymbol{r}||} \begin{cases} \left(h - ||\boldsymbol{r}||\right)^2 & \text{si } ||\boldsymbol{r}|| \leq h \\ 0 & \text{caso contrario}\,, \end{cases} \tag{3.25}$$

$$\nabla^2 W_{spiky}(\boldsymbol{r}, h) = -\frac{90}{\pi h^6}\frac{1}{||\boldsymbol{r}||}\begin{cases}(h - ||\boldsymbol{r}||)\,(h - 2||\boldsymbol{r}||) & \text{si } ||\boldsymbol{r}|| \leq h \\ 0 & \text{caso contrario}\,.\end{cases} \tag{3.26}$$

Vemos que ya no tenemos el problema que teníamos con el *poly6* kernel

$$\lim_{||\boldsymbol{r}||\to 0^-} \nabla W_{spiky}(\boldsymbol{r}, h) = \frac{45}{\pi h^6}\,, \tag{3.27}$$

$$\lim_{||\boldsymbol{r}||\to 0^+} \nabla W_{spiky}(\boldsymbol{r}, h) = -\frac{45}{\pi h^6}\,. \tag{3.28}$$

3.4.3. Viscosity kernel

La viscosidad es un fenómeno que ocurre por fricción entre moléculas, y que por lo tanto tiende a disipar la energía cinética del fluido en calor. Es decir, dadas dos partículas que estan suficientemente cerca para interactuar, el *Laplaciano* del *smoothing kernel* no puede ser negativo, porque ello implicaría la aparición de fuerzas que produzcan un aumento en la velocidad de ambas partículas. Esto puede causar problemas de estabilidad en aplicaciones de tiempo real, donde la cantidad de partículas tiende a ser baja (en comparación con otras aplicaciones) y por lo tanto el campo de velocidad no esta bien muestreado. Para evitar estos problemas, para el cálculo de la viscosidad se utilizará el siguiente kernel

$$W_{viscosity}(\boldsymbol{r}, h) = \frac{15}{2\pi h^3}\begin{cases}-\frac{||\boldsymbol{r}||^3}{2h^3} + \frac{||\boldsymbol{r}||^2}{h^2} + \frac{h}{2||\boldsymbol{r}||} - 1 & \text{si } ||\boldsymbol{r}|| \leq h \\ 0 & \text{caso contrario}\,.\end{cases} \tag{3.29}$$

Con gradiente y Laplaciano dados por

$$\nabla W_{viscosity}(\boldsymbol{r}, h) = \frac{15}{2\pi h^3}\boldsymbol{r}\begin{cases}-\frac{3||\boldsymbol{r}||}{2h^3} + \frac{2}{h^2} - \frac{h}{2||\boldsymbol{r}||^3} & \text{si } ||\boldsymbol{r}|| \leq h \\ 0 & \text{caso contrario}\,,\end{cases} \tag{3.30}$$

$$\nabla^2 W_{viscosity}(\boldsymbol{r}, h) = \frac{45}{\pi h^6}\begin{cases}(h - ||\boldsymbol{r}||) & \text{si } ||\boldsymbol{r}|| \leq h \\ 0 & \text{caso contrario}\,.\end{cases} \tag{3.31}$$

3.5. Algoritmo (parte I)

Con lo que hemos presentado hasta aquí, podemos presentar una primera versión del algoritmo para simular el fluido

Algorithm 1 SPH (WCSPH)

for all $particula_i$ **do**
  computar vecindario $N_i(t)$
end for
for all $particula_i$ **do**
  computar $\rho_i(t)$ con ec. (3.2)
  computar $p_i(t)$ con ec. (3.4) o (3.5)
end for
for all $particula_i$ **do**
  computar $\boldsymbol{F}_i^{press}(t)$ con ec. (3.10) y (3.25)
  computar $\boldsymbol{F}_i^{visc}(t)$ con ec. (3.13) y (3.31)
  computar $\boldsymbol{F}_i^{st}(t)$ con ec. (3.14)
  $\boldsymbol{F}_i(t) = m_i\boldsymbol{g} + \boldsymbol{F}_i^{st}(t) + \boldsymbol{F}_i^{press}(t) + \boldsymbol{F}_i^{visc}(t)$
end for
for all $particula_i$ **do**
  $\boldsymbol{v}_i(t + \Delta t) = \boldsymbol{v}_i(t) + \frac{1}{m_i}\Delta t\boldsymbol{F}_i(t)$
  $\boldsymbol{x}_i(t + \Delta t) = \boldsymbol{x}_i(t) + \Delta t\boldsymbol{v}_i(t + \Delta t)$
end for

Dada una partícula i, el vecindario N_i corresponde a las partículas vecinas tales que su distancia a i es menor o igual h, el radio de soporte del kernel W.

Este primer algoritmo presentado es, en esencia, el presentado originalmente en [MST03]. Si para el cómputo de presión utilizamos la ecuación (3.5), entonces el algoritmo es el presentado en [BT07]. Esta *familia* de algoritmos *SPH* suele clasificarse como *EoS Solver*[1], ya que la estimación de la presión deriva de la ecuación de estado de gases ideales [Ihm+14].

3.6. PCISPH

Como vimos en el algoritmo 1, la presión se calcula en base a la ecuación de estado de gases ideales. Esto impone una cota superior sobre el Δt que podemos elegir.

En [SP09], se propone una variante denominada *Predictive-Corrective Incompressible SPH*. En *PCISPH* se propone cambiar la manera de estimar la presión de manera iterativa. Se estima la posición y velocidad futura para $t_n + \Delta t$, se estima la densidad, y luego se actualiza la presión, que a su vez se utiliza para reestimar la fuerza debida a la presión. Una vez calculada dicha fuerza, se realiza el proceso nuevamente, pero ahora con la nueva presión. Cuando la variación de densidad alcanza el umbral deseado, se actualizan las posiciones y velocidades definitivas, y comienza un nuevo ciclo del algoritmo.

3.6.1. Algoritmo (parte II)

A continuación presentamos el algoritmo *PCISPH*, adaptando la formulación original para incorporar el modelo de tensión superficial presentado en la sección 3.3. Además, utilizamos los kernels presentados en la sección 3.4

[1] EoS: Equation of State

Algorithm 2 PCISPH

for all $particula_i$ **do**
  computar vecindario $N_i(t)$
end for
for all $particula_i$ **do**
  computar $\rho_i(t)$ inicial con ec. (3.2)
  $p_i(t) = 0$
  $\boldsymbol{F}_i^{press}(t) = \mathbf{0}$
  computar $\boldsymbol{F}_i^{visc}(t)$ con ec. (3.13) y (3.31)
  computar $\boldsymbol{F}_i^{st}(t)$ con ec. (3.14)
  $\boldsymbol{F}_i^0(t) = m_i\boldsymbol{g} + \boldsymbol{F}_i^{st} + \boldsymbol{F}_i^{visc}$
end for
while $\rho_{err}^*(t+\Delta t) > \eta$ **or** $(iter < minIterations)$ **do**
  for all $particula_i$ **do**
    $\boldsymbol{v}_i^*(t+\Delta t) = \boldsymbol{v}_i(t) + \frac{\Delta t}{m_i}\left(\boldsymbol{F}_i^0(t) + \boldsymbol{F}_i^{press}(t)\right)$
    $\boldsymbol{x}_i^*(t+\Delta t) = \boldsymbol{x}_i(t) + \boldsymbol{v}_i^*(t+\Delta t)\Delta t$
  end for
  for all $particula_i$ **do**
    computar $\rho_i^*(t+\Delta t)$ con ec. (3.2)
    computar $\rho_{err_i}^*(t+\Delta t)$
    computar $p_i(t) = p_i(t) + \delta\rho_{err_i}^*(t+\Delta t)$
  end for
  for all $particula_i$ **do**
    computar $\boldsymbol{F}_i^{press}(t)$ con ec. (3.10), utilizando $p_i(t)$
  end for
end while
for all $particula_i$ **do**
  $\boldsymbol{v}_i(t+\Delta t) = \boldsymbol{v}_i(t) + \frac{\Delta t}{m_i}\left(\boldsymbol{F}_i^0(t) + \boldsymbol{F}_i^{press}(t)\right)$
  $\boldsymbol{x}_i(t+\Delta t) = \boldsymbol{x}_i(t) + \boldsymbol{v}_i(t+\Delta t)\Delta t$
end for

El valor de ρ_{err}^* debe ser interpretado como $max(\rho_{err_i}^*)$, es decir, el máximo de los errores cometidos al estimar la densidad de las partículas: $\rho_{err_i}^* = \rho_i^* - \rho_0$.

La cantidad de iteraciones del algoritmo esta definida por dos condiciones:

- η: un umbral del error aceptado en la estimación de la densidad (qué grado de compresibilidad es aceptado).
- *minIterations*: la cantidad mínima de iteraciones a realizar, sin considerar cuán bien se está estimando la densidad.

PCISPH puede clasificarse en la categoría *Iterative EOS solver with splitting* [Ihm+14].

3.6.2. Derivación de la presión

El cambio introducido en *PCISPH* es la manera de calcular la presión de manera iterativa. El objetivo es encontrar una presión p que al ser aplicada cambie la posición de la partícula de manera tal que la densidad predecida sea la densidad en reposo ρ_0.

De la ecuación (3.2) tenemos que

$$\begin{aligned}\rho_i(t+\Delta t) &= \sum_j m_j W(\boldsymbol{r_i}(t+\Delta t) - \boldsymbol{r_j}(t+\Delta t), h) \\ &= \sum_j m_j W(\boldsymbol{r_i}(t) + \Delta\boldsymbol{r_i}(t) - \boldsymbol{r_j}(t) - \Delta\boldsymbol{r_j}(t), h) \\ &= \sum_j m_j W(\boldsymbol{d}_{ij}(t) + \Delta\boldsymbol{d}_{ij}(t), h)\,,\end{aligned}$$

donde $\boldsymbol{d}_{ij}(t) = \boldsymbol{r_i}(t) - \boldsymbol{r_j}(t)$ y $\Delta\boldsymbol{d}_{ij}(t) = \Delta\boldsymbol{r_i}(t) - \Delta\boldsymbol{r_j}(t)$.

Asumiendo que $\Delta\boldsymbol{d}_{ij}(t)$ es relativamente *pequeño*, se puede aplicar la aproximación de Taylor de primer orden a $W(\boldsymbol{d}_{ij}(t) + \Delta\boldsymbol{d}_{ij}(t), h)$, por lo que obtenemos el siguiente desarrollo

$$\begin{aligned}\rho_i(t+\Delta t) &= \sum_j m_j W(\boldsymbol{d}_{ij}(t), h) + \nabla W(\boldsymbol{d}_{ij}(t), h) \cdot \Delta\boldsymbol{d}_{ij}(t) \\ &= \sum_j m_j W(\boldsymbol{r_i}(t) - \boldsymbol{r_j}(t), h) + \sum_j m_j \nabla W(\boldsymbol{r_i}(t) - \boldsymbol{r_j}(t), h) \cdot (\Delta\boldsymbol{r_i}(t) - \Delta\boldsymbol{r_j}(t)) \\ &= \rho_i(t) + \Delta\rho_i(t)\,,\end{aligned}$$

donde el término $\Delta\rho_i(t)$ es desconocido. Sin embargo, renombrando $W_{ij} = W(\boldsymbol{r_i}(t) - \boldsymbol{r_j}(t), h)$, y considerando que la masa de todas las partículas son iguales y constantes, podemos escribir

$$\begin{aligned}\Delta\rho_i(t) &= m \sum_j \nabla W_{ij} \cdot (\Delta\boldsymbol{r_i}(t) - \Delta\boldsymbol{r_j}(t)) \\ &= m \left(\sum_j \nabla W_{ij} \Delta\boldsymbol{r}_i(t) - \sum_j \nabla W_{ij} \Delta\boldsymbol{r}_j(t) \right) \\ &= m \left(\Delta\boldsymbol{r}_i(t) \sum_j \nabla W_{ij} - \sum_j \nabla W_{ij} \Delta\boldsymbol{r}_j(t) \right)\,. \end{aligned} \tag{3.32}$$

El término $\Delta\boldsymbol{r}_i(t)$ podemos deducirlo del método de integración utilizado en el algoritmo 2, *Semi-implicit Euler*. Para ello, ignoraremos todas las fuerzas actuando sobre la *i-ésima* partícula, salvo la presión

$$\Delta\boldsymbol{r}_i(t) = \Delta t^2 \frac{\boldsymbol{F}_i^{press}}{m}\,. \tag{3.33}$$

Si además consideramos que las partículas vecinas comparten la misma presión $\tilde{p}_i$, y que la densidad corresponde a ρ_0, la densidad si el fluido fuera incompresible, entonces, utilizando la ecuación (3.10) podemos escribir lo siguiente

$$\boldsymbol{F}_i^{press} = -m^2 \sum_j \left(\frac{\tilde{p}_i}{\rho_0^2} + \frac{\tilde{p}_i}{\rho_0^2} \right) \nabla W_{ij}^{spiky} = -m^2 \frac{2\tilde{p}_i}{\rho_0^2} \sum_j \nabla W_{ij}^{spiky}\,. \tag{3.34}$$

Recordamos que para la presión usamos el kernel definido en la ecuación (3.24). Reemplazando en la ecuación (3.33) obtenemos

$$\Delta \boldsymbol{r}_i(t) = -\Delta t^2 m \frac{2\tilde{p}_i}{\rho_0^2} \sum_j \nabla W_{ij}^{spiky}. \tag{3.35}$$

Dado que las fuerzas de presión son simétricas, la *j-ésima* partícula en el vecindario de la *i-ésima* partícula recibe la siguiente fuerza

$$\boldsymbol{F}_{j|i}^{press} = -m^2 \left(\frac{\tilde{p}_i}{\rho_0^2} + \frac{\tilde{p}_i}{\rho_0^2} \right) \nabla W_{ij}^{spiky} = -m^2 \frac{2\tilde{p}_i}{\rho_0^2} \nabla W_{ij}^{spiky} \tag{3.36}$$

Entonces, el cambio de posición $\Delta \boldsymbol{r}_j$ debido a la acción de la fuerza $\boldsymbol{F}_{j|i}^{press}$ es

$$\Delta \boldsymbol{r}_{j|i}(t) = \Delta t^2 m \frac{2\tilde{p}_i}{\rho_0^2} \nabla W_{ij}^{spiky}. \tag{3.37}$$

Como sólo estamos considerando la acción de la *i-ésima* partícula sobre sus vecinas, podemos asumir que $\Delta \boldsymbol{r}_j(t) = \Delta \boldsymbol{r}_{j|i}(t)$. Entonces, reemplazando (3.37) y (3.35) en la ecuación (3.32), obtenemos

$$\begin{aligned}
\Delta \rho_i(t) &= m \left(-\Delta t^2 m \frac{2\tilde{p}_i}{\rho_0^2} \sum_j \nabla W_{ij}^{spiky} \cdot \sum_j \nabla W_{ij} - \sum_j \nabla W_{ij} \cdot \Delta t^2 m \frac{2\tilde{p}_i}{\rho_0^2} \nabla W_{ij}^{spiky} \right) \\
&= \Delta t^2 m^2 \frac{2\tilde{p}_i}{\rho_0^2} \left(-\sum_j \nabla W_{ij}^{spiky} \cdot \sum_j \nabla W_{ij} - \sum_j \nabla W_{ij} \cdot \nabla W_{ij}^{spiky} \right).
\end{aligned}$$

Si despejamos, llegamos a la expresión

$$\tilde{p}_i = \frac{\Delta \rho_i(t)}{\beta \left(-\sum_j \nabla W_{ij}^{spiky} \cdot \sum_j \nabla W_{ij} - \sum_j \nabla W_{ij} \cdot \nabla W_{ij}^{spiky} \right)}, \tag{3.38}$$

donde $\beta = \Delta t^2 m^2 \frac{2}{\rho_0^2}$.

La ecuación (3.38) significa que para alcanzar una variación de densidad $\Delta \rho_i$ tenemos que aplicar una presión $\tilde{p}_i$. En el algoritmo 2 conocemos el error $\rho_{err_i}^* = \rho_i^* - \rho_0$, por lo que necesitamos aplicar una variación de densidad $-\rho_{err_i}^*$ para que $\rho_i^* = \rho_0$, que es lo que buscamos:

$$\tilde{p}_i = \frac{-\rho_{err_i}^*}{\beta \left(-\sum_j \nabla W_{ij}^{spiky} \cdot \sum_j \nabla W_{ij} - \sum_j \nabla W_{ij} \cdot \nabla W_{ij}^{spiky} \right)}. \tag{3.39}$$

Dado que para partículas con vecindarios *pobres* esta fórmula no estima del todo bien, en realidad se utiliza un coeficiente δ, precomputado considerando una partícula prototipo con un vecindario completo

$$\delta = \frac{-1}{\beta \left(-\sum_j \nabla W_{ij}^{spiky} \cdot \sum_j \nabla W_{ij} - \sum_j \nabla W_{ij} \cdot \nabla W_{ij}^{spiky} \right)}. \tag{3.40}$$

La expresión que utilizamos para estimar la presión en el algoritmo 2 es

$$\tilde{p}_i = \delta \rho^*_{err_i}. \tag{3.41}$$

Vale aclarar que en la formulación original de PCISPH, se utiliza el mismo kernel W en todo el desarrollo de la ecuación (3.40). Nosotros hemos introducido los kernels presentados en [MST03]. Los resultados son equivalentes.

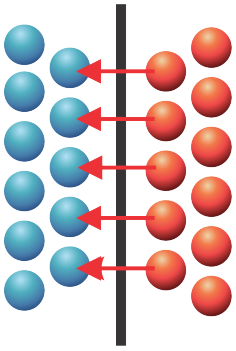

Fig. 4.1: El sólido es representado con partículas (rojas) que ejercen una fuerza de repulsión sobre las partículas del fluido.

4. INTERACCIÓN CON CUERPOS RÍGIDOS

Hasta ahora, no hemos tenido en cuenta la interacción del fluido con los posibles obstáculos con los que se pueda encontrar. Si recordamos el algoritmo 2, vemos que en ningún paso tenemos en cuenta ninguna fuerza que provenga de la interacción del fluido con otra fuerza. Si bien la interacción con cualquier cuerpo rígido debe seguir las leyes fundamentales de la mecánica, podemos tomar ciertas conseciones y clasificar dicha interacción según cómo el cuerpo rígido es afectado por la acción del fluido. Tenemos entonces dos tipos de interacción [Bri15]:

- *One-way coupling*: El cuerpo rígido se mueve por la acción del fluido, pero la reacción que éste ejerce sobre el fluido es despreciable, por lo que puede ser omitida. Es el caso de sólidos que son mucho más ligeros que el fluido.
- *Two-way coupling*: El cuerpo rígido es suficientemente ligero como para que la acción del fluido produzca cambios en su movimiento, pero a su vez, el fluido es afectado por el cuerpo rígido. Esto es también conocido como *weak-coupling*.

Podemos considerar además un tercer escenario en el que el cuerpo rígido es tan masivo en comparación con el fluido, que el mismo es prácticamente imperturbable ante la acción del fluido, que sí se ve afectado por la interacción con el sólido. En general, en las simulaciones, los *boundaries* del *contenedor* del fluido son tratados de esta manera.

4.1. Manejo de boundary en SPH

Una de las maneras más utilizadas para representar la interacción con *boundaries* de sólidos es utilizando partículas, conocidas como *boundary particles*. Entre las distintas técnicas existentes que utilizan *boundary particles*, una suele ser aplicando una fuerza de penalidad basada en la distancia entre la partícula del sólido y la del fluido [Mül+04] [Har+07] [MK09] (ver Fig. 4.1). Sin embargo estos métodos requieren fuerzas bastante grandes que a su vez limitan el Δt. Además, dado que el vecindario de las partículas de fluido tiende a ser pobre, las partículas tienden a juntarse.

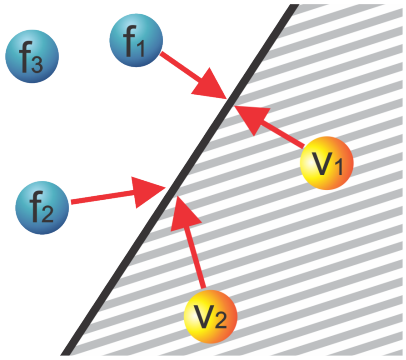

Fig. 4.2: Partículas virtuales creadas en la cercanía de la interfaz con el sólido [HA06].

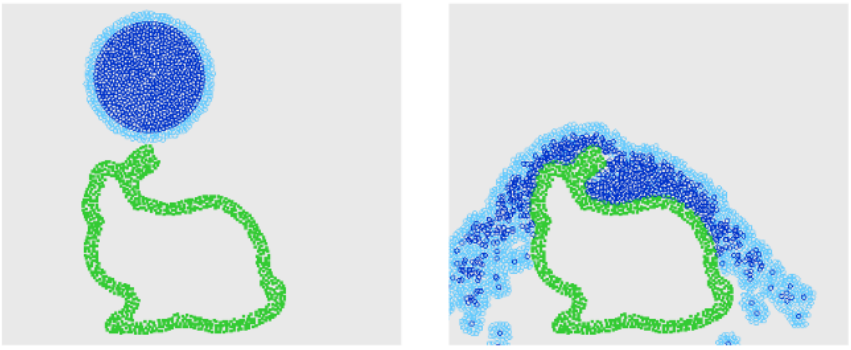

Fig. 4.3: Partículas de fluido(azul oscuro); partículas de aire (celeste); partículas fantasma (verde) [SB12].

Otra variante suele ser el uso de *ghost particles* (partículas fantasma). Por ejemplo, en [HA06], se crean dinámicamente partículas con las mismas propiedades que las partículas del fluido, pero con su velocidad espejada respecto a la superficie del sólido (ver Fig. 4.2). En [SB12], se utilizan dos variantes de partículas fantasma. Por un lado se generan dinámicamente partículas de aire alrededor de la superficie libre del fluido, que ayudan a compensar las deficiencias de SPH cuando el vecindario es pobre, y por otro lado se muestrea la superficie de los sólidos con partículas fantasma que toman dinámicamente las propiedades de las partículas de fluido cercanas (ver Fig. 4.3).

Alternativamente, si la superficie del sólido es suficientemente simple, puede representarse analíticamente, tal como se propone en [Kel06]. Respecto a *two-way interaction*, algunos enfoques tratan a las partículas de fluidos como una colección de cuerpos rígidos que colisionan con el sólido [CBP05]. Uno de los métodos más recientes, y que funciona muy bien para tiempo real (aunque no lo abordamos en este trabajo) es Position Based Fluids[MM13], donde también, la colisión con sólidos se resuelve como una colisión de cuerpos rígidos.

Por otro lado, en otros enfoques, la fuerza que ejercen las partículas de fluido sobre las de sólidos son parcialmente compuestas por las fuerzas hidrodinámicas, como en [OKR09] donde solo se considera la presión, o como en [Har+07] donde también se considera la viscosidad.

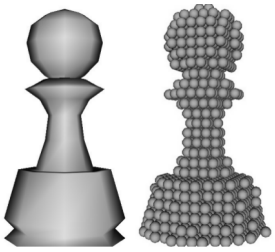

Fig. 4.4: Un peón de ajedrez, representado con partículas [Har+07].

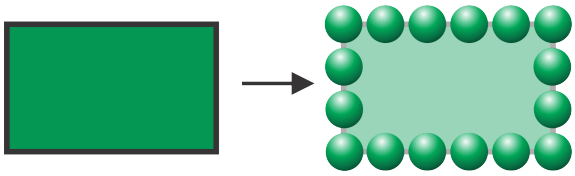

Fig. 4.5: Únicamente la superficie del sólido es muestreada con partículas.

4.2. Corrección de densidad

Para nuestro trabajo hemos adaptado el enfoque propuesto en [Aki+12], ya que presenta ciertas características que lo hacen interesante para ejecutar en tiempo real. A diferencia de muchos enfoques como en [Har+07] (ver Fig. 4.4), sólo la superficie del sólido es muestreada por partículas (ver Fig. 4.5). Esto reduce significativamente la cantidad de partículas interactuando en la escena, y por lo tanto, el uso de memoria. Otra característica interesante por la que decidimos probar este método es su simplicidad, asi como también el alto grado de independencia respecto al método utilizado para resolver la física de cuerpos rígidos. Para el presente trabajo, decidimos probar una versión simplificada del trabajo de Akinci. Una de las simplificaciones que introducimos es tener en cuenta únicamente la fuerza de presión, descartando el modelo de viscosidad que proponen para simular fricción con cuerpos rígidos. Tomamos esta decisión en pos de obtener una mayor eficiencia en escenarios interactivos.

La base del método esta tratar de mejorar la estimación de densidad cerca de los *boundaries*. Como se mencionó antes, la fórmula (3.2) funciona bien cuando la partícula esta rodeada por un vencindario completo. Pero al estar cerca de un sólido, el vecindario comienza a tener deficiencias. La solución adoptada es contemplar las partículas del *boundary* para la estimación de la densidad.

Sea b_i una partícula del *boundary* del sólido, la misma representa un volúmen V_{b_i} de la superficie del sólido

$$V_{b_i} = \frac{m_{b_i}}{\rho_{b_i}} = \frac{m_{b_i}}{\sum_k m_{b_k} W_{ik}}, \tag{4.1}$$

donde m_{b_i} representa la masa de la partícula de sólido, y k, las partículas *sólidas* del vecindario.

Por otro lado, la densidad ρ_{f_i} de una partícula de fluido cercana a un *boundary* puede estimarse de manera más precisa[SP08] como

$$\rho_{f_i} = m_{f_i} \sum_j W_{ij} + m_{f_i} \sum_k W_{ik} \,, \tag{4.2}$$

donde j denota el vecindario de fluido y k el vecindario de partículas *boundary*. Entonces podemos escribir

$$V_{b_i} = \frac{m_{b_i}}{m_{b_k} \sum_k W_{ik}} = \frac{1}{\delta_{b_i}} \,. \tag{4.3}$$

Pero existe un problema. La ecuación (4.2) funciona bien para distribuciones de partículas homogéneas. Sin embargo, en nuestro caso, no esta garantizado que el muestreo de la superficie del sólido sea homogénea. Esto haría que la densidad en zonas con mayor muestreo de partículas quede sobreestimada, generando problemas de estabilidad.

Para mitigar el problema, se propone en [Aki+12] la siguiente manera de estimar ρ_{f_i}, teniendo en cuenta el volúmen V_{b_i}

$$\rho_{f_i} = m_{f_i} \sum_j W_{ij} + \sum_k \Psi_{b_i}(\rho_{0_i}) W_{ik} \,, \tag{4.4}$$

donde ρ_{0_i} es la densidad en reposo, que en nuestro caso será la misma para todas las partículas, y $\Psi_{b_i}(\rho_0) = \rho_0 V_{b_i}$.

Otra simplificación que introducimos al modelo es calcular los valores V_{b_i} una única vez al principio de la simulación, en lugar de hacerlo en cada *frame*. Esto nos permite nuevamente ahorrar tiempo de cómputo, mientras que los efectos de hacerlo una única vez son casi despreciables en escenas interactivas.

4.3. Presión

Recordando la ecuación (3.10), podemos escribir la fuerza que una partícula ejerce sobre otra debido a la presión como

$$F^p_{i \leftarrow j} = -m_i m_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2} \right) \nabla W_{ij} \,. \tag{4.5}$$

Asumiendo que $p_i \approx p_j$ y $\rho_i \approx \rho_j$, al igual que en el desarrollo de PCISPH [SP09], entonces tenemos que

$$F^p_{i \leftarrow j} = -2 m_i m_j \left(\frac{p_i}{\rho_i^2} \right) \nabla W_{ij} \,. \tag{4.6}$$

Finalmente, aplicando la misma idea que en la derivación de la densidad, ecuación (4.4), podemos escribir la fuerza que una partícula del *boundary* ejerce sobre una de fluido como

$$F^p_{f_i \leftarrow b_j} = -2 m_i \Psi(\rho_{0_i}) \left(\frac{p_i}{\rho_i^2} \right) \nabla W_{ij} \,. \tag{4.7}$$

Y por supuesto, por la conservación de momento

$$F^p_{b_j \leftarrow f_i} = -F^p_{f_i \leftarrow b_j} . \tag{4.8}$$

4.4. Algoritmo (parte III)

El siguiente algoritmo es el que hemos utilizado en este trabajo. Agregamos el manejo de cuerpos rígidos presentado aquí a la adaptación de PCISPH que habíamos presentado en el Algoritmo 2.

Algorithm 3 PCISPH with boundary handling

for all $cuerpo\_rigido_i$ **do**
  sincronizar partículas *boundary* con la posición de $cuerpo\_rigido_i$
end for
for all $particula\_fluido_i$ **do**
  computar vecindario $N_i(t)$ incluyendo particulas de fluido y sólido
end for
for all $particula\_fluido_i$ **do**
  computar $\rho_i(t)$ inicial con ec. (4.4)
  $p_i(t) = 0$
  $\boldsymbol{F}_i^{press}(t) = \mathbf{0}$
  computar $\boldsymbol{F}_i^{visc}(t)$ con ec. (3.13) y (3.31)
  computar $\boldsymbol{F}_i^{st}(t)$ con ec. (3.14)
  $\boldsymbol{F}_i^0(t) = m_i\boldsymbol{g} + \boldsymbol{F}_i^{st} + \boldsymbol{F}_i^{visc}$
end for
while $\rho_{err}^*(t + \Delta t) > \eta$ **or** $(iter < minIterations)$ **do**
  for all $particula_i$ **do**
    $\boldsymbol{v}_i^*(t + \Delta t) = \boldsymbol{v}_i(t) + \frac{\Delta t}{m_i}\left(\boldsymbol{F}_i^0(t) + \boldsymbol{F}_i^{press}(t)\right)$
    $\boldsymbol{x}_i^*(t + \Delta t) = \boldsymbol{x}_i(t) + \boldsymbol{v}_i^*(t + \Delta t)\Delta t$
  end for
  for all $particula_i$ **do**
    computar $\rho_i^*(t + \Delta t)$ con ec. (4.4)
    computar $\rho_{err}^*(t + \Delta t)$
    computar $p_i(t) += \delta\rho_{err}^*(t + \Delta t)$
  end for
  for all $particula_i$ **do**
    computar $\boldsymbol{F}_i^{press}(t)$ con ec. (3.10) y (4.7), utilizando $p_i(t)$
  end for
end while
for all $particula\_rigida_i$ **do**
  computar $\boldsymbol{F}_{b_i}^{press}(t)$ con ec. (4.8)
end for
for all $cuerpo\_rigido_i$ **do**
  computar el total de fuerza ejercida por el fluido F_{rigid_i}
  computar el total de torque ejercido por el fluido τ_{rigid_i}
end for
for all $particula_i$ **do**
  $\boldsymbol{v}_i(t + \Delta t) = \boldsymbol{v}_i(t) + \frac{\Delta t}{m_i}\left(\boldsymbol{F}_i^0(t) + \boldsymbol{F}_i^{press}(t)\right)$
  $\boldsymbol{x}_i(t + \Delta t) = \boldsymbol{x}_i(t) + \boldsymbol{v}_i(t + \Delta t)\Delta t$
end for
for all $cuerpo\_rigido_i$ **do**
  actualizar posición y velocidad de $cuerpo\_rigido_i$ (utilizando algun solver de física de cuerpos rígidos, por ejemplo Bullets Physics)
end for

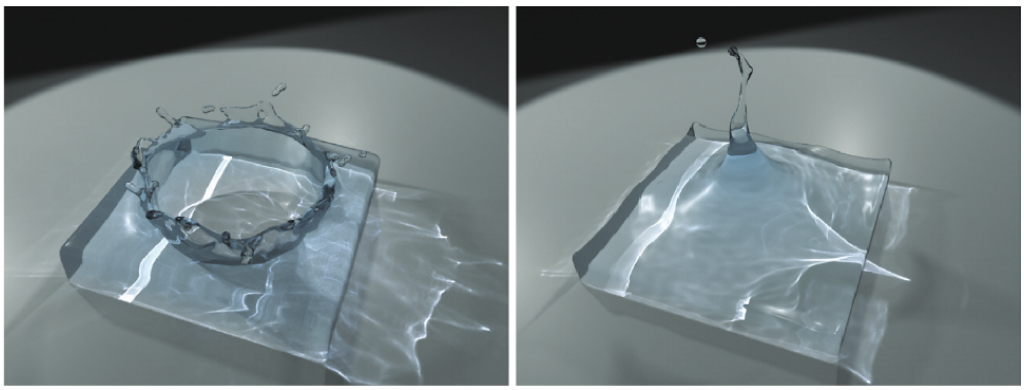

Fig. 5.1: Render generado con la técnica de *kernels anisotrópicos* y *ray-casting* [YT13].

5. VISUALIZACIÓN

En el siguiente capítulo presentaremos la técnica utilizada para la visualización del fluido. El objetivo es lograr una representación lo más cercana, visualmente, al comportamiento real del fluido y su interacción con la luz: *refracción* y *reflexión*. Al mismo tiempo, deseamos que la técnica sea lo suficientemente sencilla para permitir ser aplicada en un contexto de tiempo real. Teniendo en cuenta esta limitación, se decidió utilizar una técnica basada en *screen space rendering* (espacio de pantalla).

5.1. Introducción

La mayoría de las técnicas de visualización de fluido siempre se basaron en generar un *mesh* de la superficie libre, generándola directamente [Sto+99], o utilizando una función que genera una superficie implícita, que permite luego generar un *mesh* utilizando una técnica muy difundida como *Marching Cubes* [LC87] o *Marching Tiles* [Wil08]. Recientemente, una técnica que genera resultados visualmente muy buenos se basa en el suavizado de la función implícita utilizando kernels anisotrópicos [YT13]. Luego de obtener la superficie, para el renderizado y simulación de efectos tales como reflexión y refracción suelen utilizarse técnicas tales como *ray-casting*, en cuyo caso no se llega a generar el mesh, sino que se utiliza la función que define la superficie implícita (ver Fig. 5.1).

La técnica de renderizado que decidimos aplicar para este trabajo está basada en el trabajo de Green [Gre10]. Dicha técnica se basa en proyectar las partícualas sobre una textura, y construir en ese espacio bidimensional el renderizado de la superficie. Dado que la técnica trabaja exclusivamente en la proyección bidimensional del *viewport*, es por ello que recibe la denominación de *Screen Space*. Dado que en ningún momento se genera ningún mesh para la superficie, esta técnica es mucho más fácil de computar que las mencionadas anteriormente, lo que la hace ideal para entornos en tiempo real (ver Fig. 5.2).

5.2. Screen Space Fluid Rendering

Una vista a alto nivel del algoritmo de renderización es la siguiente:

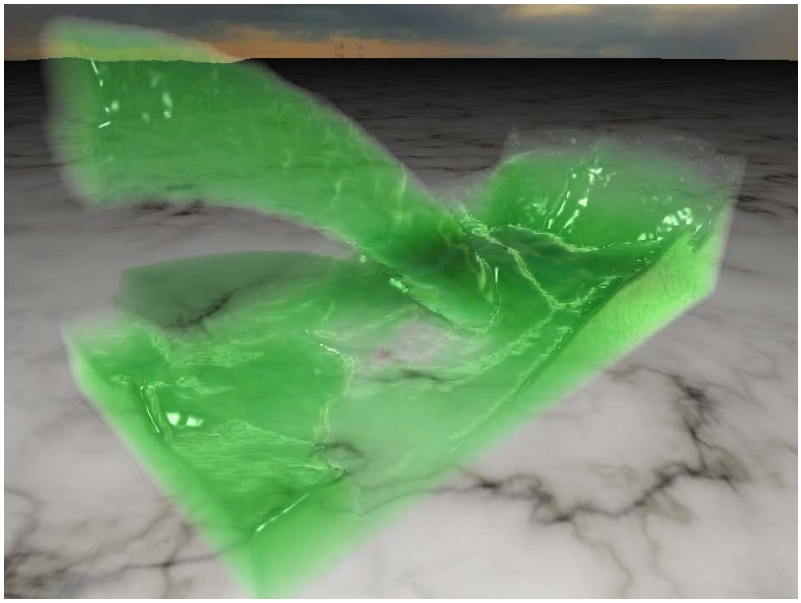

Fig. 5.2: Render generado con la técnica de *Screen Space Fluid Rendering* [Gre10].

1. Renderizar cada partícula como *point-sprite*. Guardar el valor de profundidad, *depth*, en una textura. En dicha textura se guarda la profundidad mínima para cada pixel.

2. Suavizar la textura de profundidades.

3. Computar las normales de la superficie del fluido definida por la textura de profundidad.

4. *Shade sufrace*: dibujar hacia el buffer de la escena la superficie del fluido a partir de la textura de profundidad y la de normales.

En la propuesta original del método, se incluye un paso previo al cómputo de la imagen final, que es el cálculo de una textura denominada *thickness*, en la cual se acumulan los valores de profundidad. Luego, esta información se utiliza para generar el efecto de cambio de color debido a la absorción y *scattering* de la luz a medida que atraviesa la masa de fluido. Decidimos no incluirlo ya que estamos renderizando pequeñas cantidades de fluido, y a dicha escala, este efecto no es observable al ojo humano (Ver Fig. 5.3).

5.2.1. Depth stage

Para obtener una representación de la superficie del fluido, primero procedemos a renderizar las partículas como *point sprites*, es decir, *quads* orientados en la dirección del observador. El *fragment shader* de esta etapa se encarga de guardar en una textura, a la cual llamamos *depth texture*, el valor de profundidad para cada *sprite*. En esta etapa, aún no renderizamos nada a la escena. En la Fig. 5.5 se puede observar la renderización de las partículas como **sprites**. En la Fig. 5.6 se observa la renderización de la profundidad únicamente.

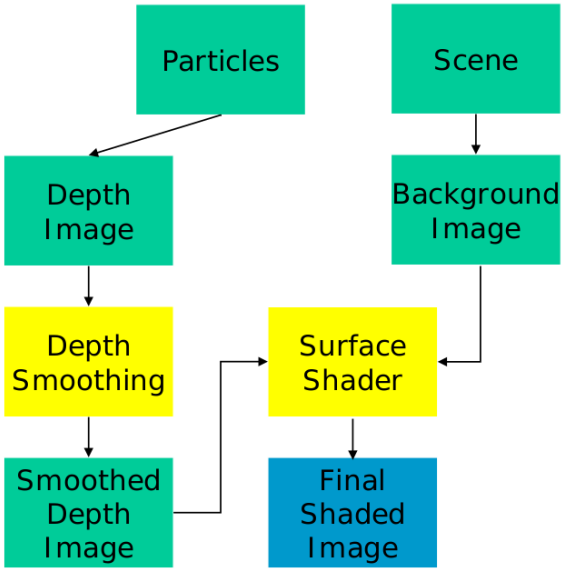

Fig. 5.3: Esquema general del SSFR, sin tener en cuenta la etapa de *thickness* [Gre10].

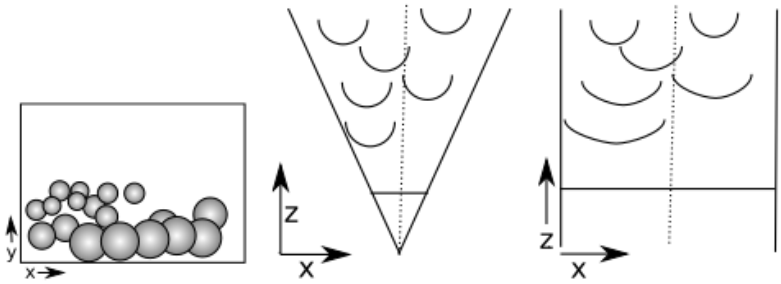

Fig. 5.4: Proyección de las partículas como sprites [VGS09].

5.2.2. Smoothing stage

Si utilizamos la textura de profundidades tal como queda luego de la primera etapa, obtendríamos una superficie poco suave, llena de relieves indiseados producto de las partículas renderizadas como esferas. Para suavizar la superficie aplicamos un filtro. En este caso, aplicaremos el filtro propuesto en [Gre10]. que nos permite suavizar la superficie, al mismo tiempo que preservar bordes. El filtro esta descripto en el algoritmo 4. La Fig. 5.7 muestra el resultado de esta etapa.

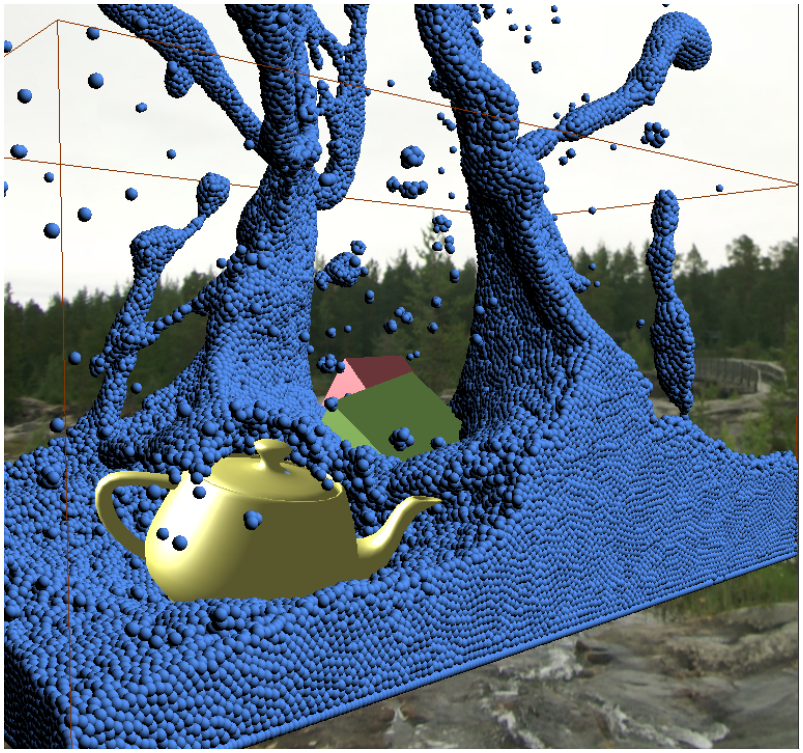

Fig. 5.5: Partículas renderizadas como *sprites*. Escena tomada del presente trabajo.

Algorithm 4 Bilateral filter

```
for all (x, y) do
  depth = read_pixel(x, y)
  sum = 0.0
  wsum = 0.0
  for x_w in range(x − WINDOW, x + WINDOW) do
    for y_w in range(y − WINDOW, y + WINDOW) do
      sample = read_pixel(x + x_w, y + y_w)
      if abs(sample-depth) ≤ threshold then
        r = length((x_w, y_w)) * blur_scale
        w = exp(-r*r)
        sum += sample * w
        wsum += w
      end if
    end for
  end for
  write_pixel((x, y), sum/wsum)
end for
```

5.2.3. Normals stage

Una vez que contamos con la textura de profundidades suavizada, necesitamos computar las normales que corresponderían a la superficie del fluido. Pero no contamos con la

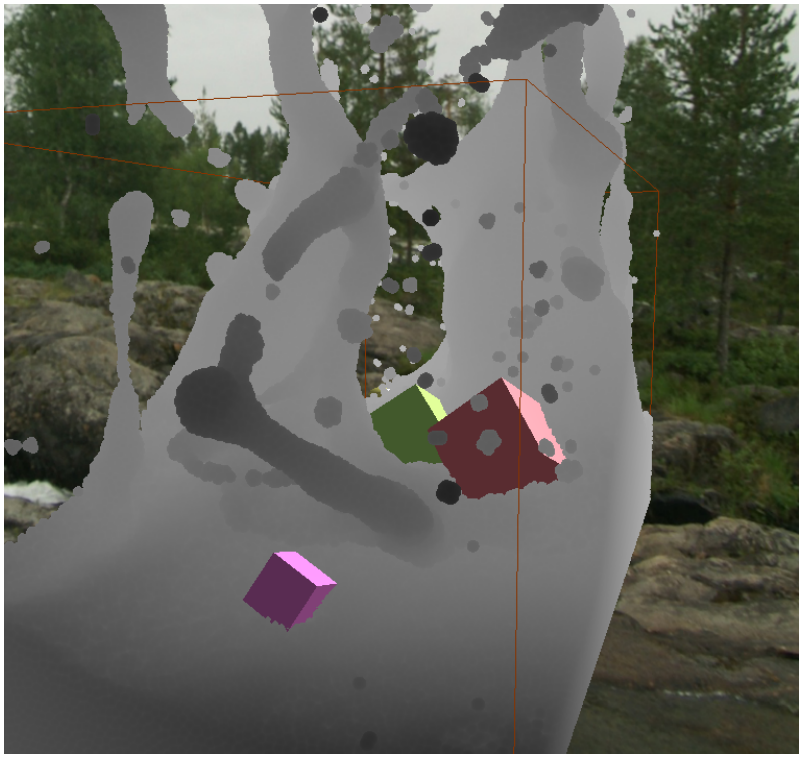

Fig. 5.6: Visualización de la textura de profundidades luego de renderizar la profundidad hacia la *depth texture*.

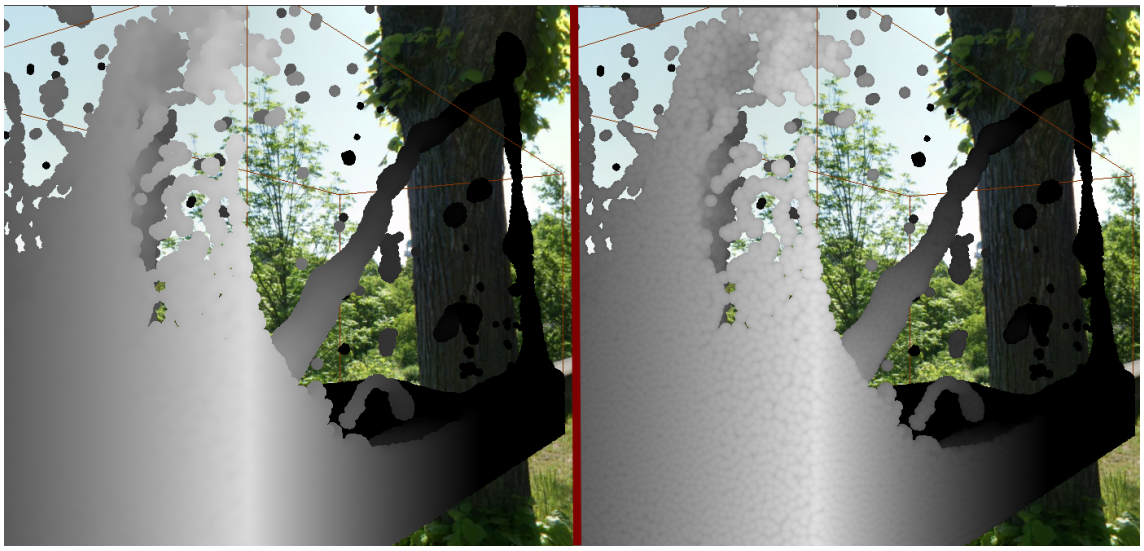

Fig. 5.7: A la izquierda, la textura ha sido suavizada. A la derecha, la textura sin suavizar.

superficie tridimensional, sino con su proyección. Para calcular las normales sobre la superficie 3D, necesitamos encontrar las posiciones de la superficie 3D. Esto es posible, ya que contamos con la proyección 2D sobre la textura, y además contamos con el valor de profundidad. Aplicando la transformación inversa del espacio de pantalla al espacio de la cámara, obtenemos las posiciones en el espacio 3D, y luego utilizando derivadas parcia-

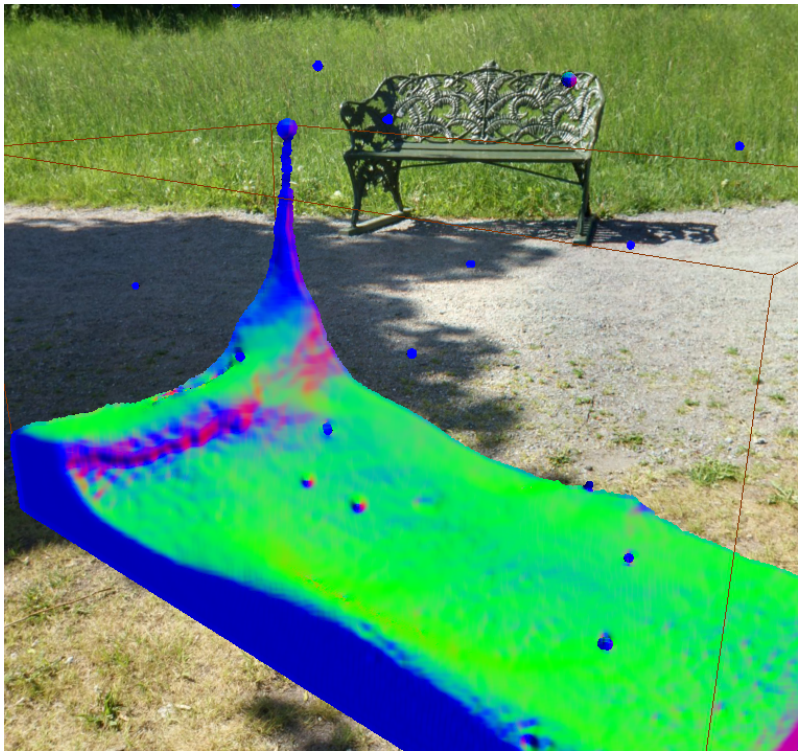

Fig. 5.8: La textura de normales renderizada. Los colores corresponden a las 3 componentes de cada vector.

les, es posible obtener las normales. El algoritmo se describe en Alg. 5. Finalmente las normales son guardadas en una nueva textura. El resultado puede verse en la Fig. 5.8.

Algorithm 5 Normals computation

```
position = get_eye_pos(depth_texture, tex_coord) {La función get_eye_pos calcula la posición en 3D a partir de la textura de profundidades y la coordenada uv actual}
dx1_position = get_eye_pos(depth_texture, tex_coord + (1, 0)) - position
dx2_position = position - get_eye_pos(depth_texture, tex_coord - (1, 0))
if abs(dx1.z) > abs(dx2.z) then
  dx1 = dx2
end if
dy1_position = get_eye_pos(depth_texture, tex_coord + (0, 1)) - position
dy2_position = position - get_eye_pos(depth_texture, tex_coord - (0, 1))
if abs(dy1.z) > abs(dy2.z) then
  dy1 = dy2
end if
normal = normalize(cross(dx1, dx2))
```

5.2.4. Etapa final: shading

El *shading* final es una combinación de varios efectos para lograr la apariencia de agua. La ecuación es la siguiente:

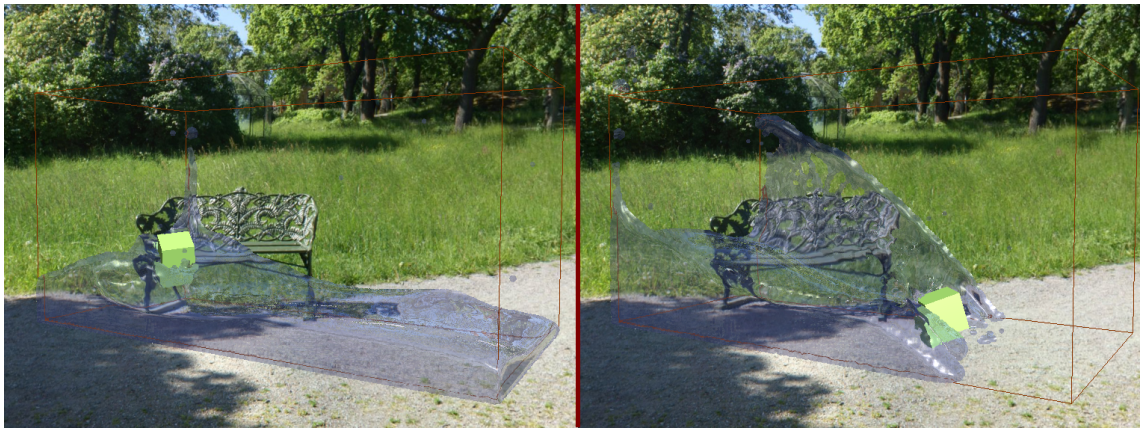

Fig. 5.9: Shading final.

$$C_{out} = \frac{1}{2}\left(C_{global} + C_{refr}(1 - F(\boldsymbol{n}\cdot\boldsymbol{v}))\right) + C_{refl}F(\boldsymbol{n}\cdot\boldsymbol{v}) + C_{specular}(\boldsymbol{n}\cdot\boldsymbol{h})^{\alpha}\,, \qquad (5.1)$$

donde

- C_{global} es un color global para el fluido.
- C_{refr} es el color correspondiente a la refracción. La refracción en el espacio de pantalla la calculamos tal como se propone en el capítulo 19 de [PF05].
- $F(\boldsymbol{n}\cdot\boldsymbol{v})$: este término corresponde a la reflexión de Fresnel. Es computado con la aproximación de *Schilik* [Gre10].
- C_{refl}: es el color debido a la reflexión. Para obtenerlo simplemente utilizamos la normal de la superficie, y leemos la posición correspondiente en la textura *cubemap* que utilizamos para el entorno. Un defecto del modelo utilizado es que únicamente utilizamos dicha textura para el efecto de reflexión, por lo que por ejemplo no se reflejan elementos que estén en la superficie del fluido.
- $C_{specular}$: componente de color tomada del modelo *Blinn-Phong*

El resultado final puede verse en la Fig. 5.9

6. PROGRAMACIÓN DE PROPÓSITO GENERAL EN GPU

6.1. Introducción

La programación de propósito general en GPU, o GPGPU por sus siglas en Inglés, *General-Purpose computing on Graphics Processing Units*, es una *disciplina* relativamente reciente, surgida en los años 2000 con el surgimiento de placas de video con un *pipeline* programable. La empresa Nvidia[1] fue la primera en introducir, en Mayo de 2001, la primera placa de video programable, la *GeForce3* (NV20). La misma presentaba soporte para OpenGL 1.2, con la posibilidad de programar *vertex shaders* y *fragment shaders*. En el año 2002, la compañía ATI[2] introdujo la placa de video *Radeon 9700* (R300), la cual presentaba soporte para operaciones de punto flotante y *looping* en los shaders, lo cual dió una mayor flexibilidad a las implementaciones de *shaders*.

Dada la naturaleza de paralelismo masivo de las GPU, programadores de todas las áreas han aprovechado esta arquitectura desde entonces, *portando* distintos algoritmos para aprovechar el procesamiento en paralelo de los datos. En los comienzos, valiéndose de las APIs gráficas, tales como OpenGL o DirectX. Por ejemplo, para aplicar un filtro a una imagen, el *kernel* se programa como un *pixel shader* y la imagen original se guarda en una textura 2D. Para ejecutar el filtro, se debe renderizar un rectángulo, de tamaño apropiado, sobre el cual se aplica la textura. El *vertex shader* entonces aplica el filtro en paralelo para cada pixel de la textura. Esta técnica por ejemplo fue utilizada en el algorítmo de visualización en este trabajo, para calcular las normales a partir de la *depth-texture*.

Sin embargo, a medida que las arquitecturas de las GPU se fueron haciendo más complejas, exponiendo más funcionalidades, utilizar APIs gráficas para resolver problemas de naturalezas distintas no era el mejor camino. Es por ello que para exponer las nuevas características y dar una mejor flexibilidad al programador, surgieron distintas APIs o *frameworks* especializados en explotar a la placa de video como un procesador SIMD masivo. Las dos alternativas más difundidas hoy en día son *CUDA*, de Nvidia, y *OpenCL*, un standard abierto mantenido por el grupo Khronos[3].

6.2. CUDA y OpenCL

CUDA es un SDK (software development kit) desarrollado por Nvidia, que permite utilizar la GPU para la programación de propósito general. La primera versión de esta herramienta surgió junto con la aparición de la *GeForce 8800 GTX*, la primera GPU de Nvidia construída con una nueva arquitectura para dar soporte a CUDA.

El lenguaje CUDA esta basado en C99, y hoy en día cuenta con extensiones para ser utilizado desde diversos lenguajes como C++, Python, o incluso Fortran[4].

La principal ventaja que presenta CUDA respecto a otras opciones es la cantidad de bibliografía disponible, herramientas y algoritmos optimizados y soporte. La desventaja en utilizar esta solución para GPGPU es que únicamente es posible ejecutar sobre hardware

[1] http://www.nvidia.com
[2] ATI Technologies fue adquirida y absorbida, en 2005, por AMD
[3] https://www.khronos.org/
[4] https://developer.nvidia.com/cuda-fortran

Nvidia. Al momento de escribir este trabajo, Nvidia cuenta con casi el 70 % del mercado de GPUs[5]. Esta posición dominante le permite ir quitando silenciosamente el soporte para otras alternativas, en particular *OpenCL*, con el aparente fin de promover la adopción de CUDA. Esto ha llevado a que la comunidad CUDA crezca, en detrimento de otras comunidades, lo cual se puede apreciar en la abundancia de bibliografía y herramientas para CUDA que no abundan, por ejemplo, para *OpenCL*.

OpenCL es un standard surgido en el año 2009. Originalmente propuesto por parte de Apple, actualmente es mantenido por el grupo *khronos*[6]. *OpenCL* define una API, basada en C99 y C++14[7], para la ejecución sobre sistemas heterogéneos: CPUs, GPUs, FPGAs, etc. En cuanto a la programación orientada a GPUs, comparte, en parte, la misma arquitectura y conceptos que CUDA. A diferencia de CUDA, al ser un standard abierto, permite ejecutarse en cualquier dispositivo que tenga soporte. La portabilidad es una de las ventajas frentre a CUDA. Además, como hemos mencionado, permite balancear la carga de trabajo entre distintos dispositivos, por ejemplo CPU y GPU, o incluso entre GPUs de distintos fabricantes (aunque esto es algo limitado). Sin embargo, en cuanto a eficiencia, depende del hardware. Es dificil desarrollar *kernels* que funcionen bien tanto en CPUs como en GPUs. Si bien la API es transparente, no puede obviarse la arquitectura sobre la cual se esta ejecutando. Finalmente, en comparación con CUDA, según pruebas [FVS11], CUDA puede ser hasta un 30 % más eficiente. Estas diferencias pueden ser atribuídas a optimizaciones del compilador de CUDA específicas para el hardware Nvidia, que no están disponibles para *OpenCL*.

6.3. Modelo de cómputo en OpenCL

A continuación describimos brevemente el modelo de programación que plantea *OpenCL*, con un enfoque en GPU.

6.3.1. Arquitectura

Como hemos mencionado con anterioridad, *OpenCL* define un framework para el cómputo heterogéneo, por lo que la arquitectura de dicho framework es a alto nivel. *OpenCL* define ciertos conceptos que están disponibles mediante su API, pero la implementación física dependerá del hardware que estemos utilizando.

La arquitectura de cualquier aplicación *OpenCL* se divide entre *host* y *device*. El hardware que actúa de *host*, típicamente una CPU, controla uno o más *compute devices*. Los *devices* son las piezas de hardware que efectúan el cómputo, por ejemplo, una o más GPUs. Cada *device* esta compuesto de *Compute units*, que a su vez están compuestos por *Processing elements*, que son los encargados finalmente de la ejecución de los *kernels* (ver Fig. 6.1). La definición exacta de *Compute unit* depende del hardware. Por ejemplo, en CPU, podemos pensar en *cores*. Si hablamos de GPUs, depende del fabricante. En hardware AMD, reciben el nombre de *Stream Cores* o *SIMD Engines*. En hardware Nvidia, existen los *Stream Multiprocessors*, o *CUDA cores*.

[5] https://jonpeddie.com/press-releases/details/add-in-board-market-decreased-in-q117-from-last-quarter-with-nvidia-gaining

[6] Del cual Nvidia forma parte, entre otros.

[7] a partir de OpenCL 2.0

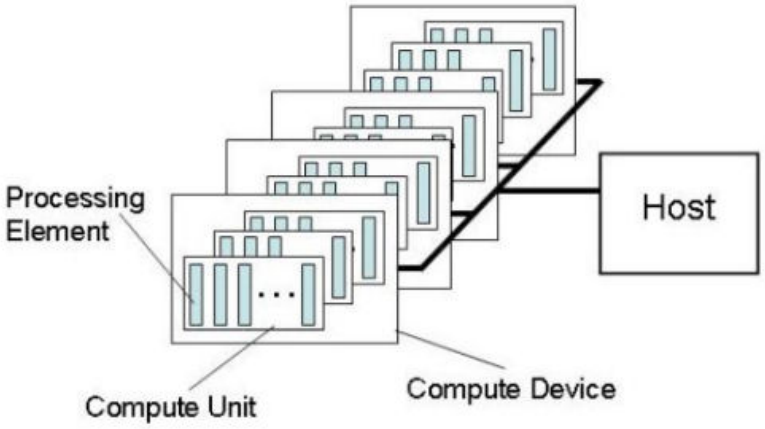

Fig. 6.1: Arquitectura básica de una aplicación OpenCL.

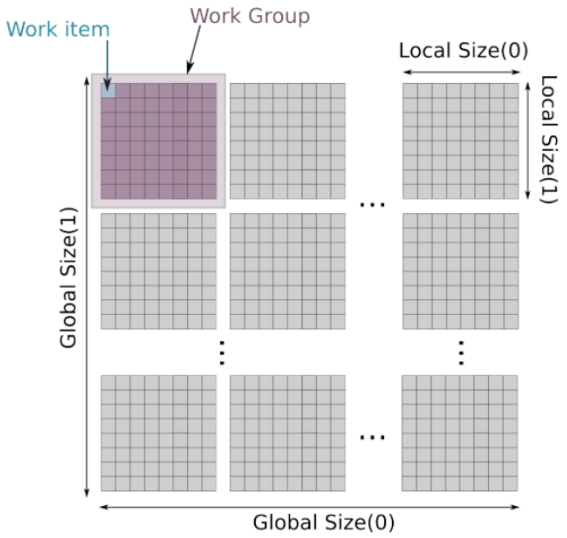

Fig. 6.2: Organización de *work-items* en una ejecución 2D.

6.3.2. Modelo de ejecución

A grandes rasgos, del lado del *host*, la aplicación *OpenCL* consiste en consultar por los *compute devices* disponibles, enviar trabajo para ejecutarse, balanceando la carga, y finalmente recoger los resultados. Del lado de los *devices*, el código que ejecutan son los llamados *kernels*. Dicho código se escribe en *OpenCL C*, un lenguaje basado en C99. Cada *processing element* ejecuta una instancia del *kernel* en paralelo. En *OpenCL*, cada elemento de ejecución se llama *work-item* (en CUDA recibe el nombre de *thread*). Cada *work-item* se agrupa en unidades deniminadas *work-group* [Mun+12] (*thread-block* en CUDA). Esta agrupación de unidades de ejecución se pueden organizar en un dominio de 1, 2 o 3 dimensiones (ver Fig. 6.2).

En general, entonces, un programa *OpenCL* consisten en

1. El *host* define un dominio N-dimensional sobre una región de memoria. Cada work-item se asocia a un índice de dicho espacio de memoria. Cada *work-item* ejecutará el mismo *kernel*.

2. El *host* organiza los *work-items* en *work-groups*. Cada item dentro grupo se ejecutará de manera concurrente, compartiendo cierto espacio de memoria.

3. Se copia de memoria DRAM a la memoria de la GPU los datos sobre los cuales trabajar, y se comienza la ejecución de cada *work-group*.

4. Una vez finalizada la ejecución de todos los *work-groups*, se descarga de la memoria de la GPU el resultado del cómputo.

Cabe aclarar que, sobre GPUs, cada *processing element* ejecuta código puramente secuencial. No existe la predicción de branches, por lo que todos los threads que se ejecutan concurrentemente, ejecutan la misma instrucción. Por ejemplo, el hardware Nvidia agrupa la ejecución de threads en *warps*, que constan de 32 *threads* [LR11]. Si parte de esos 32 threads ejecuta un branch, y parte otro branch, el hardware forzará a ejecutar primero los threads de un branch, y luego el otro. Es por ello que la mayor eficiencia se alcanza cuando los 32 threads comparten el mismo camino de ejecución.

6.3.3. Modelo de memoria

La jerarquía de memoria definida por *OpenCL* define distintos niveles de memoria disponibles. Si bien las distintas implementaciones, tanto en Nvidia como AMD difieren, a grandes rasgos, es una buena abstracción de cómo funciona la memoria de las GPUs modernas. En el nivel más alto, la memoria del programa se separa entre la memoria del *host* y la del *device*. Ambos son espacios de memoria completamente diferentes, y para enviar información de un espacio al otro, es necesario realizarlo de manera explícita. A nivel *device*, la memoria se organiza de manera jerarquica, en distintos niveles (ver Fig. 6.3):

- **Global**: el espacio de memoria global es un espacio de memoria disponible para todos los *work-items*. Leer o escribir en este espacio es lento, en comparación con los otros niveles de la jerarquía. A este nivel, no existe ningún mecanismo de sincronización entre *work-items* ni *work-groups*. *OpenCL* no garantiza un orden en el que esta memoria se lee o se escribe, por lo que hay que tener especial cuidado. Un último detalle a tener en cuenta es la manera en que se accede. Cuanado en un kernel se lee un valor de este espacio de memoria, como vimos en la sección anterior, todos los *work-items* ejecutan dicha instrucción, pero cada *work-item* leerá una zona distinta de memoria. Esto genera distintos patrones de acceso a la memoria. En general, debe favorecerse el acceso *coalescente* [Nvi09]

- **Local**: La memoria local es un espacio de memoria compartido por el *work-group*. Este nivel es de un tamaño mucho más limitado que el disponible a nivel global, mientras que las velocidades de lectura y escritura son mucho más altas. A este nivel, es posible sincronizar los *work-items* de un mismo grupo. *OpenCL* dispone de ciertos mecanismos de sincronización como *barriers*. sin embargo, no es posible contar con que los items del mismo grupo se ejecuten *todos* concurrentemente, o en algún orden particular. Como hemos visto, cada implementación ejecuta de acuerdo a su arquitectura (por ejemplo, los *warps* de Nividia).

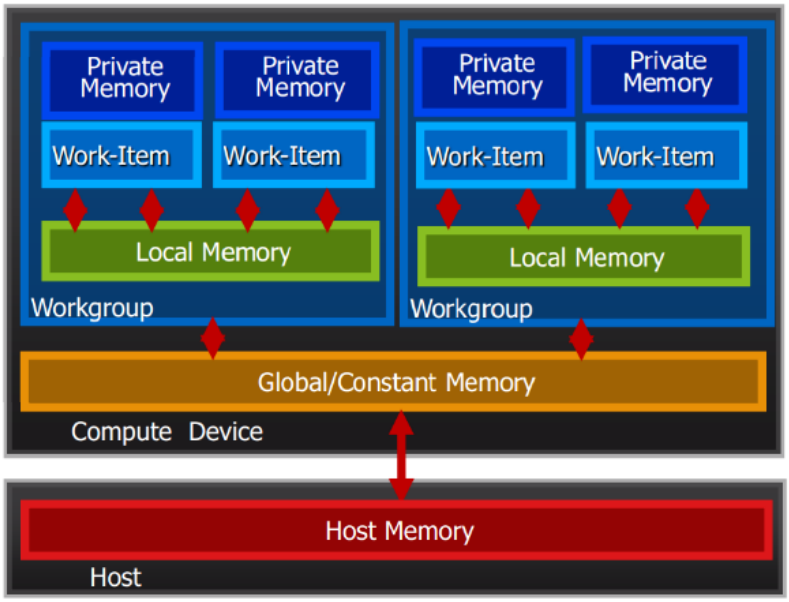

Fig. 6.3: Modelo de memoria en *OpenCL*.

- **Private**: este nivel es propio de cada *work-item*. Es la memoria de mayor velocidad en cuanto a lectura escritura, pero la más escasa. Cada *work-item* utiliza este nivel de memoria para guardar lo que se conoce como *registers*, o registros, las variables internas propias de cada implementación de un *kernel* (ver Fig. 6.4). Dado que este espacio es bastante limitado, si se utilizan muchas variables al mismo tiempo, a veces ocurre un fenómeno conocido como *register spilling* [Mic11], en donde el hardware es obligado, por falta de espacio, a volcar parte de la memoria privada del *work-item* al espacio de memoria global de manera temporal. Esto penaliza la eficiencia de ejecución.

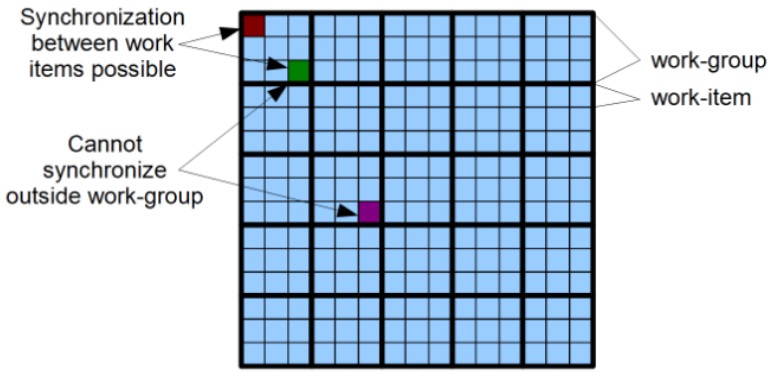

Fig. 6.4: Sincronización entre *work-items*

7. IMPLEMENTACIÓN Y RESULTADOS

7.1. Detalles de implementación

A continuación detallamos las herramientas utilizadas para desarrollar el presente trabajo, así como algunos detalles de implementación.

7.1.1. Aplicación

La aplicación desarrollada para el presente trabajo fue realizada utilizando herramientas que permitan la portabilidad entre distintos sistemas, con el objetivo de que pueda ser probada en diferentes ambientes. El lenguaje elegido para desarrollar la aplicación fue C++14. Tradicionalmente es el lenguaje por excelencia utilizado por aplicaciones gráficas, tales como juegos, donde se requiere eficiencia.

El software consiste en una aplicación de escritorio basada en el framework *Qt* 5.7[1]. Dicho framework permite, entre muchas otras cosas, el manejo de ventanas de manera independiente del gestor de ventanas utilizado y sistema operativo.

La aplicación fue diseñada especialmente para facilitar la parametrización de la simulación de manera sencilla. La interfaz de usuario permite cambiar ciertos parámetros y explorar de manera interactiva cómo cambia el comportamiento de la simulación frente a nuevos valores, como se muestra en la Fig. 7.1.

7.1.2. Física de cuerpos rígidos

Nuestra implementación para la simulación del fluido y su interacción con cuerpos rígidos no tiene en cuenta la simulación de cuerpos rígidos. Es decir, se computa la fuerza ejercida por el fluido sobre el cuerpo rígido, pero no hemos implementado el cómputo de la dinámica del mismo debida a esta fuerza, y las demás fuerzas producto de la interacción con otros cuerpos rígidos. Ver algoritmo 3. Sin embargo, debido a cómo es el algoritmo, es fácil desacoplar esta etapa y delegarla. En nuestro caso, hemos elegido utilizar *BulletPhysics*[2], un *motor* de física *open-source*.

7.1.3. OpenCL kernels

Como mencionamos en el capítulo 6, la simulación fue implementada utilizando OpenCL. La versión utilizada fue 1.2[3], dado que el trabajo fue desarrollado y probado utilizando hardware *Nvidia*. El algoritmo 3 fue implementado en diversos *kernels*, organizados como se muestra en la Fig. 7.2

Grid

Los *kernels* que implementan las funciones relacionadas con la grilla para la búsqueda de vecinos son

[1] https://www.qt.io/
[2] http://bulletphysics.org/wordpress/
[3] Al momento de escribir este informe, la versión más reciente es 2.2

Fig. 7.1: El panel de la derecha permite cambiar valores de la simulación. La sección inferior permite visualizar la cantidad de partículas en escena, y los FPS.

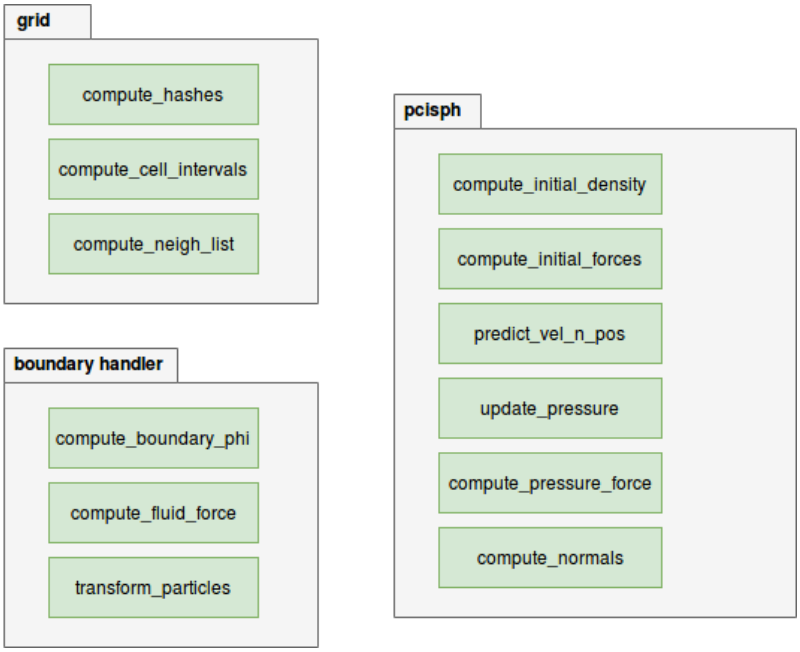

Fig. 7.2: Los *kernels* de la aplicación están agrupados en 3 unidades lógicas.

- *compute_hashes*: para cada partícula, calcula el hash que le corresponde de a cuerdo a la celda de la grilla en la que se encuentra, utilizando la curva z

- *compute_cell_intervals*: dado el buffer de posiciones ordenadas según el hash, calcula, para cada celda de la grilla, el par de índices i, j correspondiente a las partículas que pertenecen a la celda.

- *compute_neigh_list*: para cada partícula, arma la lista de índices de posiciones que corresponden a los vecinos que se encuentran dentro del radio de soporte.

Boundary handler

Kernels para el manejo de las partículas de sólido

- *compute_boundary_phi*: para cada partícula de sólido de un cuerpo rígido, computa $\rho_0 V_{b_i}$.
- *compute_fluid_force*: para cada partícula de sólido, computa la fuerza ejercida por las partículas de fluido vecinas.
- *transform_particles*: dado un vector de posición y rotación del cuerpo rígido, actualiza la posición de las partículas de dicho cuerpo rígido en el espacio.

PCISPH

Este módulo implementa los kernels principales

- *compute_initial_density*: computa la estimación inicial de densidad para cada partícula de fluido.
- *compute_initial_forces*: computa las fuerzas que actúan sobre cada partícula de fluido, sin condiderar la presión.
- *predict_vel_n_pos*: actualiza la posición y velocidad de cada partícula de fluido, útilizando el método de Euler *semi-implicito*.
- *update_pressure*: para cada particula de fluido, actualiza la presión según el esquema de PCISPH.
- *compute_pressure_force*: para cada partícula de fluido, calcula la fuerza debida a la presión estimada.
- *compute_normals*: para cada partícula de fluido, calcula la normal correspondiente.

7.1.4. Búsqueda de vecinos

Una de las operaciones más utilizadas a lo largo del algoritmo 3 consiste encontrar, para una partícula, las partículas vecinas, dado un radio de búsqueda fijo. La implementación de dicha operación fue realizada adaptando el trabajo Green [Gre10] para OpenCL.

Para cada partícula del fluido, es necesario encontrar sus vecinos, dado el *smoothing radius* propuesto por el método SPH. Es por ello que, para realizar las búsquedas utilizamos una grilla tridimensional sobre el espacio. Cada celda de dicha grilla tendrá de lado el valor del *smoothing radius*. De esta manera, dada una partícula dentro de una celda de la grilla, quedará garantizado que para encontrar las partículas vecinas dentro del radio de búsqueda, sólo basta con buscar en las 8 celdas adyacentes, y en la propia celda donde se encuentra la partícula (ver Fig. 7.3).

En nuestra implementación mantenemos el estado de las partículas en *buffers* separados, uno para cada propiedad que afecta a cada partícula. En particular, los buffers con los que comienza cada frame son los que corresponden a la posición y velocidad. Dado que

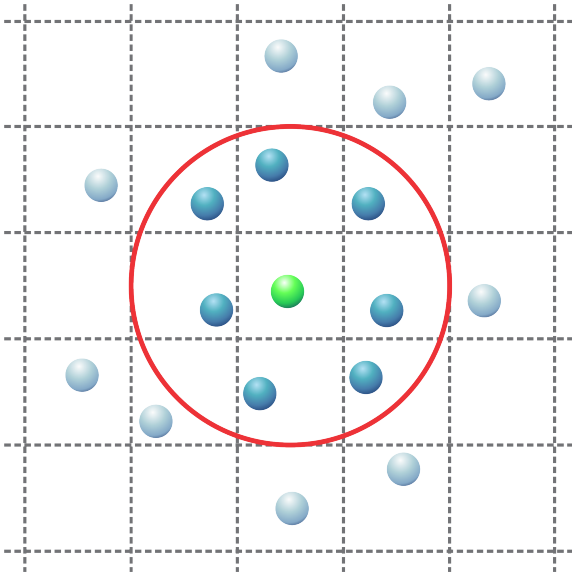

Fig. 7.3: Los vecinos de la partícula verde estan en las celdas adyancentes.

permanentemente se acceden a posiciones de memoria de partículas vecinas, es deseable que estos accesos sean lo más eficientes posibles y evitar *cache-misses*. Es decir, es deseable que las partículas cercanas en el espacio, lo estén en el buffer. El algoritmo implementado para calcular los vecindarios es el siguiente:

1. Para cada partícula, se calcula un hash asociado, basado en la posición de la celda en la grilla. Dicho hash se computa utilizando la curva Z, *Z-order curve* [Mor66] (ver Fig. 7.4).

2. Se ordenan los buffers de posiciones y velocidades de las partículas de acuerdo al hash calculado. El sorting se realiza utilizando una versión paralela de *radix-sort*. La implementación elegida es provista por el paquete *CLOGS v1.5.0*, el cual ofrece una buena performance respecto a otras implementaciones [Mer15] (ver Fig. 7.5).

3. Para cada celda de la grilla, se calcula el rango de índices dentro del buffer de posiciones (y velocidades). Dado que el hash computado en 1 es igual para todas las partículas de una misma celda, entonces todas las partículas de esa celda estarán de manera contigua en el *buffer*. Estos rangos se guardan en un nuevo *buffer*. Cada posición de este nuevo buffer corresponde a una celda de la grilla.

Una vez que para cada celda sabemos el rango $[i, j]$ de partículas que están dentro de la misma, entonces el vecindario puede ser consultado recorriendo los rangos para cada una de las 26 celdas adyacentes, más la propia celda de la partícula.

Los subsiguientes *kernels* parten de los buffers de posición y velocidad ya ordenados, por lo que no es necesario ordenar los buffers correspondientes a la densidad, normal, presión y fuerza, pues las escrituras se realizan en el orden deseado.

7.1.5. Optimizaciones adicionales

- *Local memory*: Incluimos el uso de la memoria local (*on-chip*). Para cada kernel, el i-ésimo thread lee de los buffers de posición, velocidad, etc, según corresponda, los

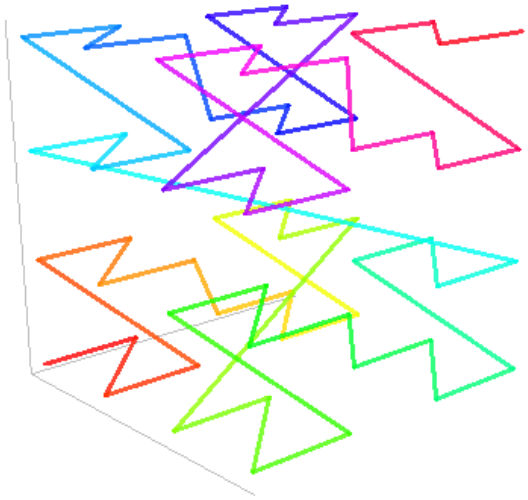

Fig. 7.4: Curva Z para llenado del espacio.

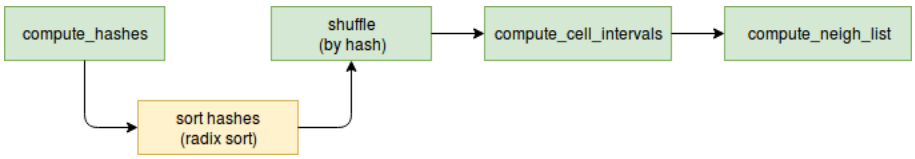

Fig. 7.5: Kernels involucrados en el cómputo de vecindarios para cada partícula.

valores que necesita, guardándolos en un *cache* de memoria local, compartido por todos los *threads*. El hecho de tener ordenados los buffers, hace que la probabilidad de que en dicho *cache* se encuentren los vecinos de i es alta. El uso de esta técnica nos reportó mejoras del 20 % al 30 % en el rendimiento.

- Utilizamos `float4` en lugar de `float3` para favorecer la alineación en memoria.
- Todas las constantes son, o .bien inyectadas al momento de compilación (como `defines`) o bien declaradas en la sección de memoria constante.
- Dado que la lectura de vecinos suele generar patrones de acceso a memoria no coalescentes, utilizamos `image1d_buffer_t` para el acceso a los buffers, aprovechando la *cache* especial para texturas que muchas GPUs implementan. Esto nos reportó mejoras del orden de entre 5 a 10 *fps* adicionales.
- Compilamos nuestros *kernels* utilizando las opciones `-cl-mad-enable` y `-cl-fast-relaxed-math`

7.2. Resultados

A continuación presentamos tiempos de ejecución promedio, para distintas escenas, y para distinta cantidad de partículas en escena, así como los resultados visuales obtenidos.

7.2.1. Hardware

El hardware utilizado para el desarrollo y mediciones fue el siguiente

- CPU: Intel Core i7 930 @ 2.80GHz
- RAM: 18GB
- Video: Nvidia GeForce GTX 970, 4GB VRAM
- OS: Linux Mint 18.1, kernel 4.4.0-87-generic (x86-64)

La versión del driver Nvidia es 352.63, con CUDA SDK 7.5.

7.2.2. Parámetros

Para todas las pruebas, los valores de los parámetros de simulación fueron los siguientes:

Parámetro	Valor
ρ_0	1000.0
Δt	0.005
$\vec{g}$	9.8
γ	1.0
ν	1.7
PCISPH máx iter	7
PCISPH min iter	3
PCISPH $\Delta\rho$ threshold	1 %

El *support radius* siempre será $4r$, donde r es el radio de la partícula.

7.2.3. Mediciones

Se probaron distintas escenas, y para cada escena, se midió el promedio de cuadros por segundo de la escena (*fps*), tomando, en todos los casos 2000 cuadros de duración. Los resultados son los siguientes:

Escena *double dam break*

Partículas en escena	Radio de partícula	Renderizado partículas	Renderizado Screen Space
177408	0.008	131.47 fps	117.17 fps
87500	0.01	229.72 fps	196.16 fps
52920	0.012	323.76 fps	254.74 fps

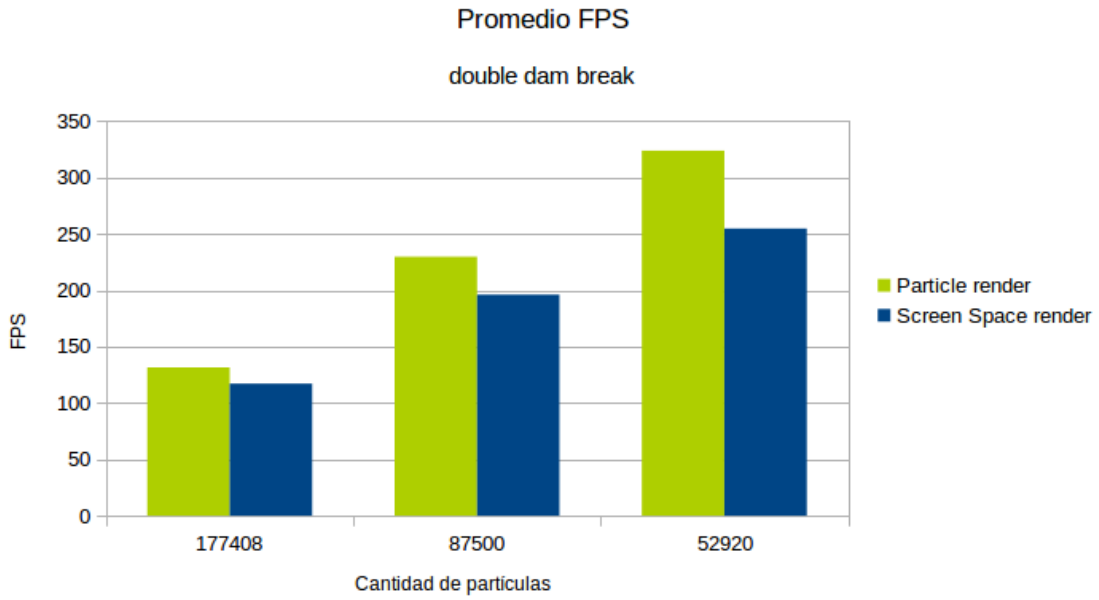

Fig. 7.6: Promedio de cuadros por segundo.

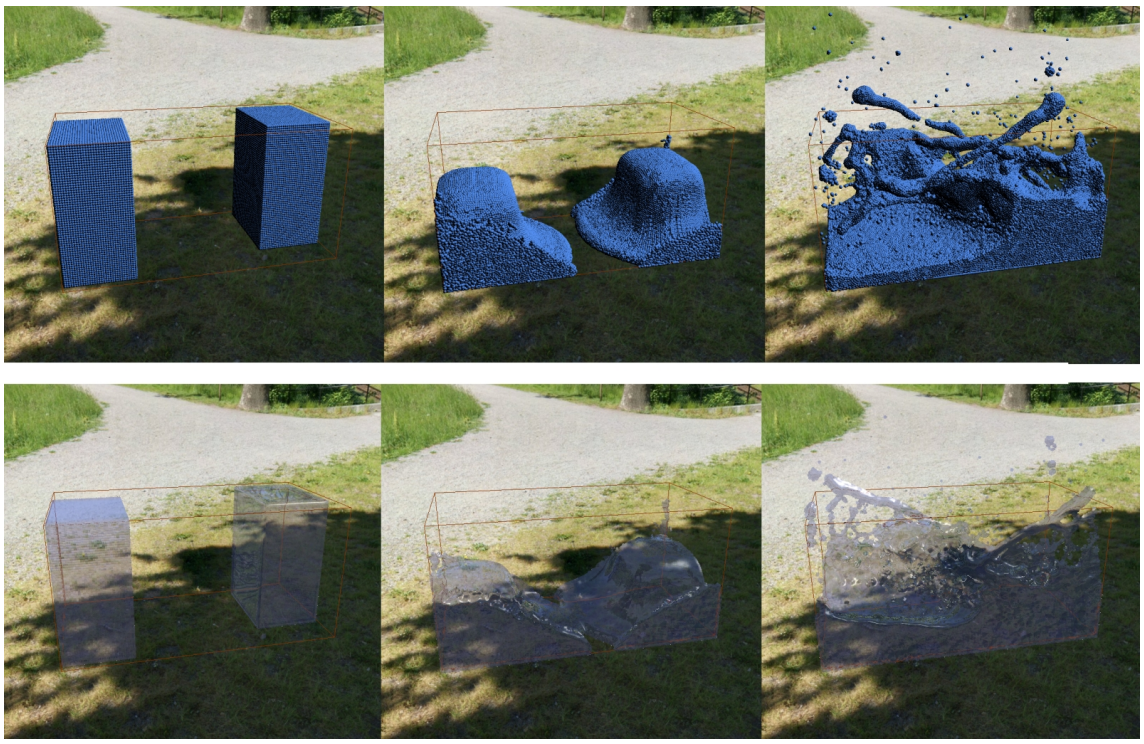

Fig. 7.7: Escena *double dam break*.

Escena *bunny*

Partículas en escena	Radio de partícula	Renderizado partículas	Renderizado Screen Space
104720	0.008	142.39 fps	131.12 fps
55438	0.01	196.45 fps	175.68 fps
33724	0.012	227.34 fps	193.46 fps

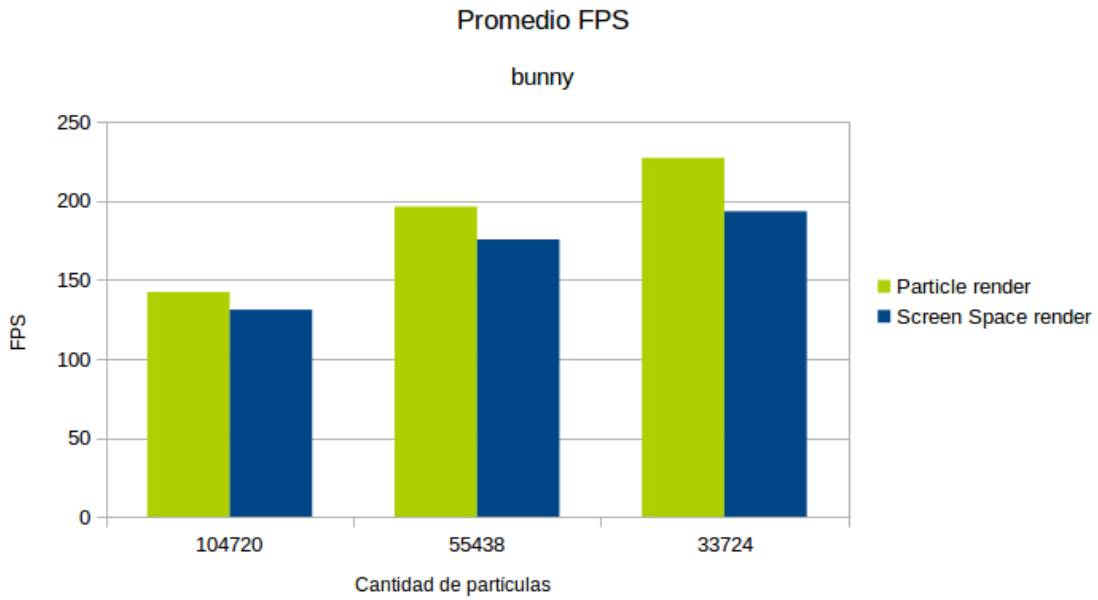

Fig. 7.8: Promedio de cuadros por segundo.

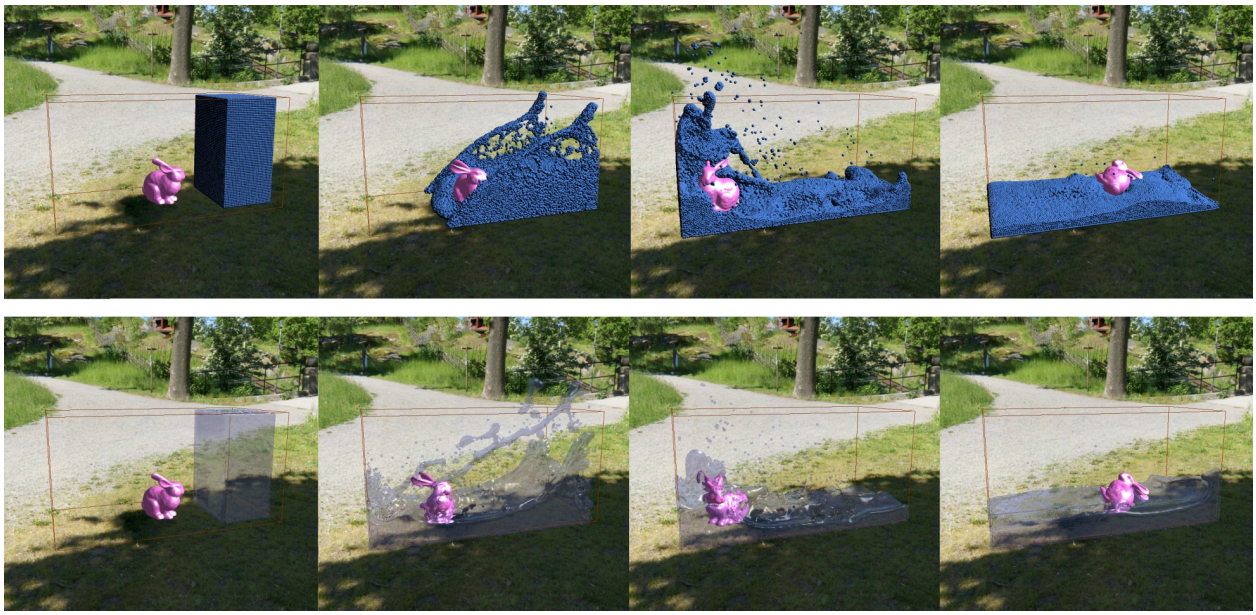

Fig. 7.9: Escena *bunny*.

Escena *many objects*

Partículas en escena	Radio de partícula	Renderizado partículas	Renderizado Screen Space
199764	0.008	84.01 fps	78.84 fps
102876	0.01	115.58 fps	107.76 fps
64234	0.012	138.93 fps	128.11 fps

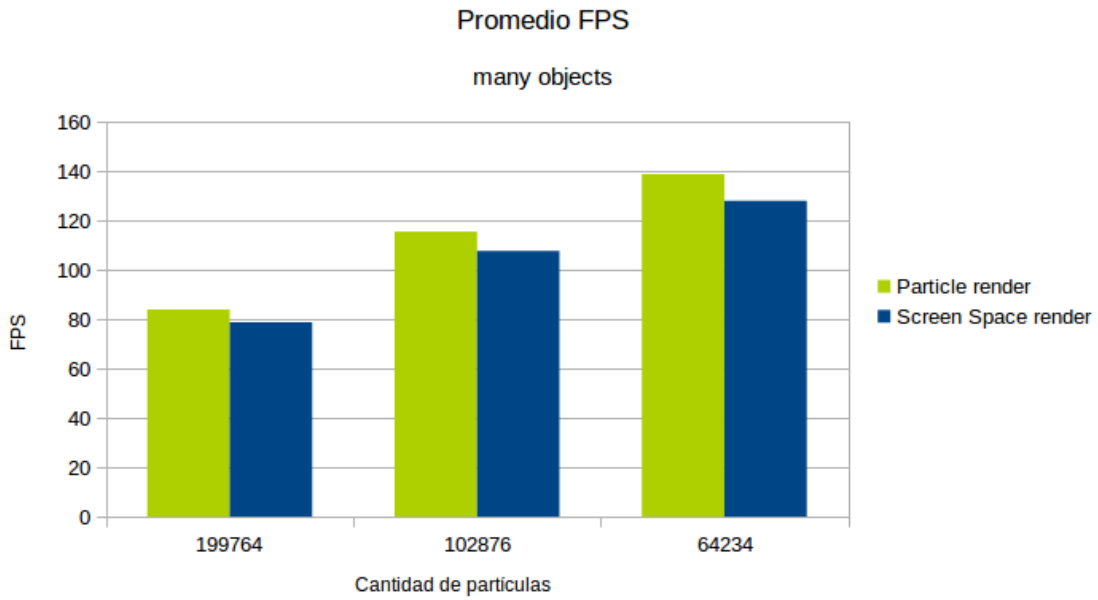

Fig. 7.10: Promedio de cuadros por segundo.

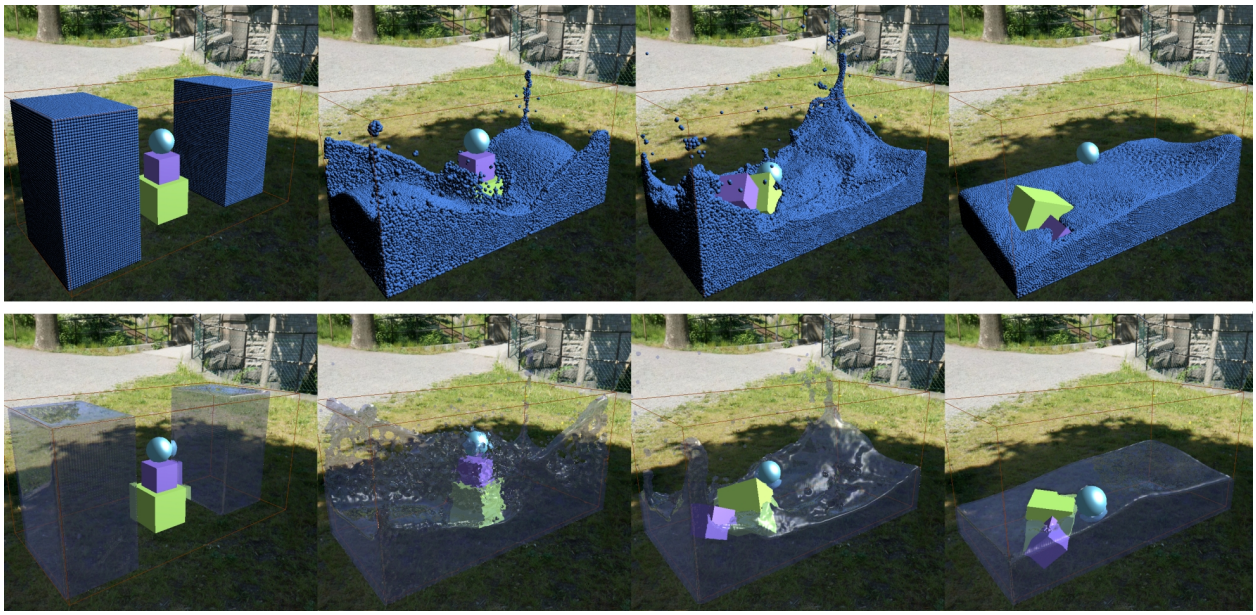

Fig. 7.11: Escena *many objects*.

8. CONCLUSIONES Y TRABAJO FUTURO

La simulación gráfica de fluidos basada en partículas es, sin dudas, un método muy conveniente para las aplicaciones que requieren tiempos interactivos. Su naturaleza los hace especialmente atractivos para ser implementados de manera paralela, en particular sobre hardware GPU. En el presente trabajo hemos presentado una introducción a SPH, tanto en su versión estandar [MCG03] como su variante PCISPH [SP09]. Hemos visto, además, cómo representar a los sólidos con partículas [Aki+12] e integrarlas al algoritmo SPH para producir interacción entre el fluido y los mismos. Finalmente introducimos una técnica de rendering *Gre10* eficiente, y que logra resultados visualmente aceptables.

El aporte de nuestro trabajo es presentar una implementación paralela sobre GPU de los métodos anteriormente descriptos, integrados para lograr simulaciones de fluidos con interacción con sólidos en tiempos interactivos. Creemos que, si bien existen muchos aspectos para optimizar, esta implementación podría ser incluída en juegos u otras aplicaciones que requieran de simulación de fluidos en tiempos interactivos. Entendemos que el campo de la simulación gráfica y del hardware ha madurado lo suficiente para que este tipo de métodos puedan ser utlizados en software de uso comercial sin problemas. Además, hemos probado que OpenCL es una plataforma viable para desarrollar este tipo de técnicas. Creemos que, sin dudas, el futuro de la simulación de fluidos en tiempo real estará fuertemente ligada a los métodos de partículas.

8.1. Trabajo futuro

Si bien PCISPH presenta una notable mejora respecto a la versión original de SPH, nuestros ensayos mostraron que para $\Delta t > 0{,}005$ la simulación comienza a volverse inestable, sobre todo para grandes cantidades de partículas. En los últimos años varias técnicas han surgido que mejoran la estabilidad de la simulación y que, si bien poseen un costo computacional más elevado que las técnicas aquí presentadas, permiten manejar Δt más grandes. Esto implica lograr simulaciones en tiempo real más estables. Entre estos trabajos se encuentra *Position Based Fluids* [MM13]. También existe Implicit Incompressible SPH[Ihm+13], que luce prometedor.

Respecto al rendering, una posible mejora es explorar el método propuesto por Yu [YT13], en donde en lugar de utilizar los sprites circulares de las partículas para realizar el *screen space* rendering, se intenta encontrar y aplicar una transformación de acuerdo a la posición de la partícula respecto del resto. De esta forma se obtienen resultados visualmente superiores.

En cuanto a OpenCL, si bien continúa teniendo soporte y constantes actualizaciones, recientemente *Khronos* a lanzado un nuevo estandard para el desarrollo de aplicaciones gráficas: *Vulkan*. La promesa de quitar complejidad en la implementación de los drivers de las placas gráficas (con el costo de una mayor responsabilidad y complejidad para el usuario) promete mejorar notablemente el rendimiento. Además cuenta, al menos por el momento, con amplio apoyo de los principales fabricantes de placas gráficas. Esto hace que sea interesante analizar si una implementación utilizando *Vulkan* aportaría mejoras significativas respecto de OpenCL o CUDA.

Una de las secciones que más tiempo consumen del algoritmo presentado es la búsqueda

y generación de listas de vecinos para cada partícula. Nuestra implementación utiliza *radix-sort*, pero el uso de la variante de *counting-sort* presentada por Hoetzlein [Hoe14] podría aportar mejoras significativas.

Finalmente, explorar el uso de múltiples GPU, o múltiples dispositivos, tal como lo permite OpenCL, podría significar una mejora considerable en la performance.

BIBLIOGRAFíA

[CM90] Alexandre J. Chorin y Jerrold E. Mardsen. *A Mathematical Introduction to Fluid Dynamics*. 3.ª ed. Springer, 1990.

[Mau13] Robert G. Maughan. «Towed Water Turbine Computational Fluid Dynamics Analysis». Utah State University, 2013.

[Bri15] Robert Bridson. *Fluid simulation for computer graphics*. 2.ª ed. A K Peters, 2015.

[MST03] Matthias Müller, Simon Schirm y Matthias Teschner. «Interactive Blood Simulation for Virtual Surgery Based on Smoothed Particle Hydrodynamics». En: (2003).

[Ree83] William T. Reeves. «Particle systems — a technique for modeling a class of fuzzy objects». En: *ACM Transactions on Graphics* 2 (1983), págs. 91-108.

[Sta99] Jos Stam. «Stable fluids». En: *SIGGRAPH 99 Conference Proceedings* (1999), págs. 121-128.

[Mon92] J. J. Monaghan. «Smoothed particles hydrodynamics». En: *Annual review of astronomy and astrophysics* 30 (1992), págs. 543-574.

[MCG03] Matthias Müller, David Charypar y Markus Gross. «Particle-Based Fluid Simulation for Interactive Applications». En: *Eurographics/SIGGRAPH Symposium on Computer Animation* (2003).

[Har+07] Takahiro Harada y col. «Real-time Coupling of Fluids and Rigid Bodies». En: *APCOM* (2007), págs. 1-13.

[Ihm+10] Markus Ihmsen y col. «Boundary handling and adaptive time-stepping for PCISPH». En: *Workshop on Virtual Reality Interaction and Physical Simulation VRIPHYS* (2010), págs. 79-88.

[Aki+12] Nadir Akinci y col. «Versatile Rigid-Fluid Coupling for Incompressible SPH». En: *ACM Transactions on Graphics* (2012).

[GM77] R. A. Gingold y J. J. Monaghan. «Smoothed particle hydrodynamics: theory and application to non-spherical stars». En: *Royal Astronomical Society, Monthly Notice* (1977), págs. 375-389.

[Luc77] L. B. Lucy. «A numerical approach to the testing of the fission hypothesis». En: *Astronomical Journal* 82 (1977), págs. 1013-1024.

[Mon05] J. J. Monaghan. «Smoothed particles hydrodynamics». En: *Reports on Progress in Physics* (2005).

[SF95] J. Stam y E. Fuime. «Depicting Fire and other Gaseous Phenomena using Difussion Processes». En: *Computer Graphics, 29th Annual Conference Series* (1995), págs. 129-136.

[Roy95] Trina M. Roy. «Depicting Fire and other Gaseous Phenomena using Difussion Processes». Master. University of Illinois at Chicago, 1995.

[Mon94] J. J. Monaghan. «Simulating free surface flows with SPH». En: *Journal of Computational Physics* 110(2) (1994), págs. 399-406.

[Kel06] Micky Kelager. *Lagrangian Fluid Dynamics Using Smoothed Particle Hydrodynamics*. 2006.

[BT07] M. Becker y M. Teschner. «Weakly compressible SPH for free surface flow». En: *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2007), págs. 209-217.

[SB12] H. Schechter y R. Brindson. «Ghost SPH for animating water». En: *ACM Transactions on Graphics (Proceedings SIGGRAPH)* (2012), 61:1-61:8.

[ZYF10] B. Zhu, X. Yang e Y. Fan. «Creating and preserving vortical details in SPH fluid». En: *Computer Graphics Forum 29* (2010), págs. 2207-2214.

[LD09] T. Lenares y P. Dutré. «Mixing fluids and granular materials». En: *Computer Graphics forum 28* (2009), págs. 213-228.

[YT13] J. Yu y G. Turk. «Reconstructing surfaces of particle-based fluids using anisotropic kernels». En: *ACM Trans. Graph. 32* (2013), 5:1-5:12.

[Ihm+14] M. Ihmsen y col. «SPH Fluids in Computer Graphics». En: *EUROGRAPHICS 2014* (2014).

[SP09] B. Solenthaler y R. Pajarola. «Predictive-corrective incompressible SPH». En: *ACM Transactions on Graphics (Proceedings SIGGRAPH) 28* (2009), 40:1-40:6.

[TM05] A. Tartakovsky y P. Meakin. «Modeling of surface tension and contact angles with smoothed particle hydrodynamics». En: *Physical Review E 72* (2005).

[AAT13] Nadir Akinci, Gizem Akinci y Matthias Teschner. «Versatile Surface Tension and Adhesion for SPH Fluids». En: *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH Asia 2013* 32 (2013).

[Whi72] Harvey E. White. *Modern College Physics*. 6.ª ed. Van Nostrand Reinhold Co., 1972.

[Mül+04] M. Müller y col. «Interaction of fluids with deformable solids». En: *Computer Animation and Virtual Worlds 15* (2004), págs. 159-171.

[MK09] J. Monaghan y J. Kajtar. «SPH particle boundary forces for arbitrary boundaries». En: *Computer Physics Communications 180* (2009), págs. 1811-1820.

[HA06] X. Hu y N. Adams. «A multi-phase SPH method for macroscopic and mesoscopic flows». En: *Journal of Computational Physics 213* (2006), págs. 844-861.

[CBP05] S. Clavet, P. Beadouin y P. Poulin. «Particle-based viscoelastic fluid simulation». En: *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2005), págs. 219-228.

[MM13] Miles Macklin y Matthias Müller. «Position Based Fluids». En: *ACM Transactions on Graphics (TOG) - SIGGRAPH 2013 Conference Proceedings* (2013), págs. 219-228.

[OKR09] S. Oh, Y. Kim y B. Roh. «Impulse-based rigid body interaction in SPH». En: *Computer Animation and Virtual Worlds 20* (2009), págs. 215-224.

[SP08] B. Solenthaler y R. Pajarola. «Density Contrast SPH Interfaces». En: *Proc. of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2008), págs. 211-218.

[Sto+99] D. Stora y col. «Animating lava flows». En: *Graphics Interface* (1999), págs. 203-210.

[LC87] W. E. Lorense y H. E Cline. «Marching cubes: A high resolution 3d surface construction algorithm». En: *SIGGRAPH Computer Graphics 21, 4* (1987), págs. 163-169.

[Wil08] B. W. William. «Fluid Surface Reconstruction from Particles». Master. The University Of British Columbia, 2008.

[Gre10] Simon Green. *Screen Space Fluid Rendering for Games*. 2010.

[VGS09] W. Van der Laan, S. Green y M. Sainz. «Screen Space Fluid Rendering with Curvature Flow». En: *Proceedings of the 2009 symposium on Interactive 3D graphics and games* (2009).

[PF05] Matt Pharr y Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005.

[FVS11] Jianbin Fang, Ana Lucia Varbanescu y Henk Sips. «A Comprehensive Performance Comparison of CUDA and OpenCL». En: *Proc. Int'l Conf. on Parallel Processing* (2011).

[Mun+12] A. Munshi y col. *OpenCL Programming Guide*. Addison-Wesley, 2012.

[LR11] J. Luitjens y Steven Rennich. *CUDA Warps and Occupancy*. 2011. URL: `http://developer.download.nvidia.com/CUDA/training/cuda_webinars_WarpsAndOccupancy.pdf`.

[Nvi09] Nvidia. *Advanced CUDA Webinar: Memory optimizations*. 2009. URL: `http://developer.download.nvidia.com/CUDA/training/NVIDIA_GPU_Computing_Webinars_CUDA_Memory_Optimization.pdf`.

[Mic11] Paulius Micikevicius. *Local Memory and Register Spilling*. 2011. URL: `http://developer.download.nvidia.com/CUDA/training/register_spilling.pdf`.

[Mor66] G. M. Morton. «A Computer Oriented Geodetic Data Base; and a New Technique in File Sequencing». En: (1966).

[Mer15] Bruce Merry. «A performance comparison of sort and scan libraries for GPUs». En: *Parallel Processing Letters, 25(4)* (2015).

[Ihm+13] M Ihmsen y col. «Implicit Incompressible SPH». En: *IEEE Transactions on Visualization and Computer Graphics* (2013), págs. 426-435.

[Hoe14] Rama Hoetzlein. *Fast Fixed-Radius Nearest Neighbors: Interactive Million-Particle Fluid*. 2014. URL: `http://on-demand.gputechconf.com/gtc/2014/presentations/S4117-fast-fixed-radius-nearest-neighbor-gpu.pdf`.