UNIVERSIDAD DE CARABOBO
Facultad Experimental de Ciencias y Tecnología
Licenciatura en Computación

MODELADO Y ANIMACIÓN REALISTA DEL MOVIMIENTO DE LA SUPERFICIE DE UN CUERPO DE AGUA

JUAN M. GARCÍA A.
Autor

JORGE ERNESTO RODRIGUEZ
Tutor Académico

Bárbula, Octubre 2014

Índice general

Lista de Códigos

Índice de figuras

Introducción

La simulación del agua es un fenómeno complejo que sigue siendo difícil modelarlo interactivamente por simulación en computadora. El presente trabajo se enfoca en el desarrollo y análisis de un algoritmo eficiente y estable para animar ondas de aguas poco profundas en dos dimensiones basado en ecuaciones. Tomando en cuenta el esfuerzo de todas las investigaciones que se han realizado en el campo de la física y dinámica de fluidos, un modelo de fluidos basado en física permite producir movimientos de ondas totalmente realistas.

Estas ecuaciones varían en complejidad, desde el modelo de ecuaciones de Navier-Stokes, que modela la turbulencia o el comportamiento de fluidos con una viscosidad arbitraria en tres dimensiones. Hasta una ecuación sencilla que describe el comportamiento de la propagación de una onda simple en la superficie del agua. Este útimo modelo es el que utilizaremos en este trabajo, donde las ecuaciones están dadas por el estudio implícito de integración Semi-Lagrangiano, que nos permite largos intervalos de tiempo mientras mantiene estabilidad.

En este trabajo presentaremos una problematica a la hora de generar simulaciones de aguas poco profundas, en donde a través de las investigaciones y observando algunos antecedentes sobre estudios similares logramos obtener una propuesta para esta problematica, emplenado una metodología de desarrollo para el proyecto y apoyado en las bases teóricas necesarias para implementar un algoritmo y realizar los experimentos que nos llevaran a obtener los resultados esperados y acertados para resolver esta dicha problematica. Se presentarán los resultados finales obtenidos y las conclusiones finales de la investigacion.

La finalidad de esta investigación es demostrar como este modelo puede ser usado para generar animaciones de ondas sobre la superficie del agua, simulando el efecto que produciría un objeto cayendo al agua.

Capítulo 1

El Problema de Investigación

1.1. Planteamiento del Problema

Una gran cantidad de esfuerzo se ha invertido en la campo de la dinámica de fluidos computacional para crear modelos que describan los movimientos de los fluidos. Estos modelos varían en sus niveles de complejidad y amplitud, y algunos son más adecuados para la animación de un cierto fenómeno que otros. En esta sección, se resume una serie de modelos de dinámica de fluidos que se han propuesto para simular los movimientos de los fluidos incompresibles (es decir, líquidos), se señala el alcance de la aplicabilidad y limitaciones de cada modelo, y se compara su complejidad computacional.
Las ecuaciones de Navier-Stokes, en particular, es el más completo de todos los modelos de fluidos, describe el movimiento de una partícula de fluido en un lugar arbitrario en el campo del líquido en cualquier instante de tiempo, y se derivan de La segunda ley de Newton del movimiento.

En tres dimensiones, las ecuaciones para un fluido incomprensible obtienen la siguiente forma:

$$\begin{aligned} &\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} \\ &= g_x - \frac{1}{\rho}\frac{\partial p}{\partial x} + \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right), \end{aligned} \tag{1.1}$$

$$\begin{aligned} &\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z} \\ &= g_y - \frac{1}{\rho}\frac{\partial p}{\partial y} + \nu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2}\right), \end{aligned} \tag{1.2}$$

$$\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z}$$
$$= g_y - \frac{1}{\rho}\frac{\partial p}{\partial z} + \nu\left(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2}\right), \tag{1.3}$$

donde u, v y w son componentes de la velocidad del fluido en las direcciones $x-$, $y-$ y $z-$; p denota la presión; g denota la constante gravitacional; ρ denota la densidad, que se asume como constante; y ν denota la viscosidad cinempática del fluido. Las ecuaciones de Navier-Stokes son usualmente aplicadas junto con la ecuación de continuidad, que establece la ley de la conservación de la masa:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \tag{1.4}$$

1.2. Objetivos de la Investigación

1.2.1. Objetivo General

Implementar una herramienta de software que permita modelar, visualizar y animar ondas en movimiento en la superficie de un cuerpo de agua con apariencia convincente y con bajo costo computacional.

1.2.2. Objetivos Específicos

1. Hacer una revisió bibliográfica en el área de investigació.
2. Escoger un método que se aproxime lo más posible a los requerimientos establecidos.
3. Diseñar un método para poder llevar a cabo los cálculos de las integrales implícitas.
4. Diseñar la aplicación de modelado de fluidos.
5. Implementar el programa que permita llevar a cabo la simulació a partir de un conjunto de puntos y un estado inicial de la superficie.
6. Hacer mallado y rendering orientado a simular visualmente un cuerpo de líquido que asemeje agua.
7. Identificar y modelar los límites y obstáculos del cuerpo de agua.
8. Validar el funcionamiento de la aplicación.
9. Reportar resultados y conclusiones.

Capítulo 2

Marco Teórico

Para los alcances de esta investigación, el marco teórico referencial se dividirá en tres partes: antecedentes, bases teóricas y definición de términos.

2.1. Antecedentes

Los modelos de simulación basado en las ecuaciones de Navier-Stokes, como el propuesto por Foster y Metaxas (1996) [5], son muy realistas, ya que captura la dinámica, en tres dimensiones, de fluidos con viscosidad arbitraria. Fenómenos interesantes, como un chorro de agua que salpica en un tanque o un gran océano con olas en una superficie de poca profundidad, se puede simular con un alto nivel de realismo. La desventaja de estos modelos es que, puesto que las ecuaciones son en tres dimensiones, el algoritmo toma $\vartheta(N^3)$ tiempo para correr, donde N es el número de sub-intervalos en una red de dimensión R3. Por lo tanto, las tasas de interactividad son difíciles de alcanzar en la actualidad usando estos modelos.

Stam (1999) [6] propuso un modelo estable, basado en las ecuaciones de Navier-Stokes, que modela flujos complejos de líquido. Al resolver las ecuaciones con el método de las características y el tratamiento de los términos de difusión de forma implícita, se pueden utilizar grandes intervalos de tiempo. Sin embargo, los problemas de simulación de fluidos sin fronteras o sin límites (como el agua) y con obstáculos no son abordados.

Chen y sus compañeros de trabajo (Chen y Lobo 1995, Chen y colaboradores, 1997) [7], [8], lograron una simulación interactiva mediante la eliminación de la dependencia vertical y resolviendo las ecuaciones de Navier-Stokes en dos dimensiones. La superficie del líquido es modelada mediante el cálculo de la altura

del campo de presión. Ellos animaron fluidos de diferentes viscosidades, variando el número de Reynolds. Sin embargo, la adopción de un método de tiempo de integración explícita conduce a la inestabilidad potencial si se eligen pequeños intervalos de tiempo, lo que a su vez aumenta el costo computacional.

Además, su enfoque en dos dimensiones no puede modelar fácilmente flujos de líquidos q recipientes con borde curvo, o alrededor de un objeto. También es importante mencionar simuladores comerciales de líquido basados en las ecuaciones de Navier-Stokes: Un ejemplo es *RealWave*, siendo una funcionalidad del Software Realflow realizado por Next Limit S.L., el cual modela la propagación de las ondas gravitacionales sobre una malla, y simula superficies líquidas, como la del mar. Esta propuesta resuelve la simplificación de las ecuaciones de Navier-Stokes, con un factor de viscosidad simulada, utilizando un esquema de integración numérica.

Otro ejemplo es *Digital NatureTools* por Areté Entertainment [12]. En este modelo, el método de espectro de fluidos se utiliza para capturar una discontinuidad de contacto en un flujo multifacético incomprensible (por ejemplo, una interface aire-agua).

Longuet-Higgins y Cokelet (1975, 1978) [9], [10], propone un método numérico para el seguimiento de la evolución temporal de ondas superficiales en tiempo y espacio utilizando el potencial de velocidad de partículas de fluidos en la superficie libre. En cada intervalo de tiempo, una ecuación integral se resuelve para el nuevo componente de la normal de la velocidad. El número de variables independientes en el cálculo es $\vartheta(N^2)$. En esta formulación, tanto la viscosidad como la tensión superficial son ignoradas.

Al asumir la viscosidad cero y teniendo en cuenta sólo movimientos en dos dimensiones, un conjunto aún más simple de ecuaciones de fluidos se pueden derivar: las ecuaciones de aguas poco profundas. Las ecuaciones de aguas poco profundas, a menudo se adoptan en aplicaciones oceanográficas y atmosféricas, para modelados de gran escala global, donde la fuerza de Coriolis, inducida por la rotación de la Tierra, está incluida. Para efectos de animación, sin embargo, la fuerza de Coriolis se considera despreciable y ha sido excluida en las ecuaciones anteriores. Las ecuaciones de aguas poco profundas se limitan a la descripción, en dos dimensiones, de fluidos no viscosos, lo que significa que los líquidos de alta viscosidad no se pueden modelar, y que los líquidos no pueden salpicar. Sin embargo, estas ecuaciones modelan adecuadamente una cantidad suficiente de los movimientos de fluidos.

Para que la simulación sea estable en grandes intervalos de tiempo, un esquema de integración implícita en tiempo debe ser utilizado. Sin embargo, los términos no lineales en la advección de las ecuaciones (el segundo y tercer término) hacen que

la resolución del sistema no sea trivial. Kass y Miller (1990) [5] propusieron una solución a este problema, asumiendo que la velocidad del agua varía lentamente en el espacio, permitiendo resolver en tiempo real las ecuaciones de aguas poco profundas.

Khan (1994) [13] propuso un modelo cinemático de animación creando un barco en movimiento a lo largo de una trayectoria arbitraria. En lugar de utilizar la solución de un problema de dinámica de superficie libre, el modelo superpone ondas circulares que emanan de distintos puntos a lo largo del barco en movimiento para determinar el perfil de la estela. Como uno de los primeros intentos de modelar las olas del agua, Fournier y Reeves (1986) [3] presentan un modelo simple, basado en ecuaciones de ondas en dos dimensiones, para la animación de las olas rompiendo cerca de una playa. Peachey (1986) [2] presenta un enfoque similar. Sin embargo, sus modelos de predicción como el movimiento de las ondas tienen que determinarse a priori, y las perturbaciones introducidas por el usuario pueden no ser manejados fácilmente. Esto significa que puede ser difícil de construir una aplicación interactiva de sus modelos, y movimientos como los que resultan de un objeto cayendo en el agua no pueden ser simulados.
En este trabajo, se presenta un modelo basado en las ecuaciones de aguas poco profundas en dos dimensiones para modelar movimientos de olas, superposiciones de aguas, objetos a la deriva, y obstáculos y paredes de formas varias.

2.2. Bases Teóricas

2.2.1. El método de integración implícita Semi-Lagrangiano

El método semi-lagrangiano puede ser visto como un híbrido entre la aproximación Eureliana y Langragiana. En un esquema de advección Eureliano, el observador se queda en un punto fijo geográficamente a medida que el mundo, o el líquido se desarrolla a su alrededor. Este esquema conserva la regularidad de la malla mientras que el observador permanezca fijo, pero requiere pequeños intervalos de tiempo pequeño con el fin de mantener la estabilidad. Por otra parte, en un esquema Lagrangiano, el observador ve el mundo evolucionar mientras viaja con una partícula de fluido. Esta técnica esta menos restringuida a los requerimientos de estabilidad permitiendo largos intervalos de tiempo. Sin embargo, ya que las partículas de fluido, inicialmente se encuentran regularmente espaciadas se mueven con el tiempo, estas usualmente se vuelven irregularmente espaciadas mientras el sistema evoluciona. El esquema de advección semi-lagrangiano intenta combinar las ventajas de ambos esquemas, la regularidad del esquema Eureliano y la gran estabilidad del esquema Langragiano; integrando a lo largo de la trayectoria de las partículas mientras que se evalúan funciones en los puntos de la malla

en cada intervalo de tiempo.

El meétodo semi-lagrangiano se introdujo por primera vez en una simulación atmosférica por Robert (1981). En la mayoría de las aplicaciones de ingeniería, el método semi-lagrangiano se utiliza en conjunto con la aproximación semi-implicita, donde los términos sin advección son promediados en tiempo con el fin de lograr una precisión de segundo orden. En este trabajo se decidió adoptar un enfoque implícito relativamente simple que no requiere promediar los tiempos, pero solo produce soluciones de primer orden, el cual debe ser lo suficientemente preciso para aplicaciones gráficas.
Cuando se aplica a una ecuación de advección, el método semi-lagrangiano es equivalente a una técnica para la solución de ecuaciones diferenciales parciales. Sin embargo, el método semi-lagrangiano conserva su simplicidad y su práctica utilidad en apliaciones más complejas donde las curvas características pueden derivarse de las trayectorias de las partículas del fluido, ya que la evolución del flujo sigue siendo calculada siguiendo trayectorias de las partículas de fluido. Para mostrar cómo las ecuaciones de aguas poco profundas puede integrarse en el esquema semi-lagrangiano, hay que considerar en primer lugar las ecuaciones de aguas poco profundas en una dimensión:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + g\frac{\partial h}{\partial x} = 0, \tag{2.1}$$

$$\frac{\partial h}{\partial t} + \frac{\partial}{\partial x}[u(h-b)] = 0, \tag{2.2}$$

La cual puede ser reescrita en forma Lagrangiana como:

$$\frac{\mathrm{d}u}{\mathrm{d}t} + g\frac{\partial h}{\partial x} = 0, \tag{2.3}$$

$$\frac{\mathrm{d}h}{\mathrm{d}t} - u\frac{\partial b}{\partial x} + d\frac{\partial u}{\partial x} = 0, \tag{2.4}$$

Donde $d = h - b$ y las derivadas Lagrangianas son definidas como:

$$\frac{\mathrm{d}}{\mathrm{d}t} = \frac{\partial}{\partial t} + u\frac{\partial}{\partial x}, \tag{2.5}$$

Donde

$$\frac{\mathrm{d}x}{\mathrm{d}t} = u(x,t), \tag{2.6}$$

Como se muestra en la figura 2.1, la derivada Lagrangiana es aproximada a lo largo de la trayectoria. Imagina una partícula de fluido que viaja a lo largo de una trayectoria y llega a la posición x_i en el momento t_{n+1}. Entonces, de acuerdo al método semi-lagrangiano, las derivadas se calculan desde su posición en t_{n+1}, que es x_i, y en t_n, el cual es llamado *punto de partida* y se caracteriza como $\tilde{x}_i^n$ en la figura 2.1.Digamos que α_i^n es el desplazamiento de una partícula de fluido en el intervalo de tiempo de t_n hasta t_{n+1}, terminando en el punto bajo x_i. Para una función arbitraria $\psi(x,t)$, $\tilde{\psi}^n$ denota la función correspondiente más alta en el intervalo de tiempo de t_n hasta t_{n+1}: $\alpha_i^n = x_i - \tilde{x}_i^n$. Que es, $\tilde{\psi}^n(x_i) \equiv \psi(x_i - \alpha_i^n, t_n)$. Por lo tanto, las derivadas Lagrangianas son aproximadas como:

$$\frac{du}{dt} = \frac{u^{n+1} - \tilde{u}^n}{\Delta t}, \frac{dh}{dt} = \frac{h^{n+1} - \tilde{h}^n}{\Delta t} \tag{2.7}$$

En la ecuación superior 2.7, u^{n+1} y h^{n+1} se evalúan en los puntos de la red a nivel de tiempo t_{n+1}, mientras que $\tilde{u}^n$ y $\tilde{h}^n$ son evaluados en los puntos de partida en t_n.

Asumimos que la profundidad del agua (d) es aproximadamente constante en el intervalo de tiempo (t_n, t_{n+1}). Mediante la adopción de aproximación 2.7, discretizamos 2.3 y 2.4 en el tiempo implícito para obtener las siguientes ecuaciones:

$$\frac{u^{n+1} - \tilde{u}^n}{\Delta t} + g\frac{\partial h^{n+1}}{\partial x} = 0, \tag{2.8}$$

$$\frac{h^{n+1} - \tilde{h}^n}{\Delta t} - u^{n+1}\frac{\partial b}{\partial x} + d^n\frac{\partial u^{n+1}}{\partial x} = 0, \tag{2.9}$$

La altura del suelo, b, se supone que es independiente del tiempo, lo que lleva a concluir que no está asociada con el tiempo.

El método descrito anteriormente es implícito y de primer orden, tanto en tiempo como en espacio. Esto último implica que se pueden tomar grandes intervalos de tiempo sin perder estabilidad. Las ecuaciones 2.8 y 2.9 pueden ser resueltas tomando la derivada parcial de 2.8 con respecto a x, y luego usando la ecuación resultado y 2.8 para eliminar los terminos $\frac{\partial u^{n+1}}{\partial x}$ y u^{n+1} de 2.9 para solo obtener una ecuación de Helmholtx en h^{n+1}:

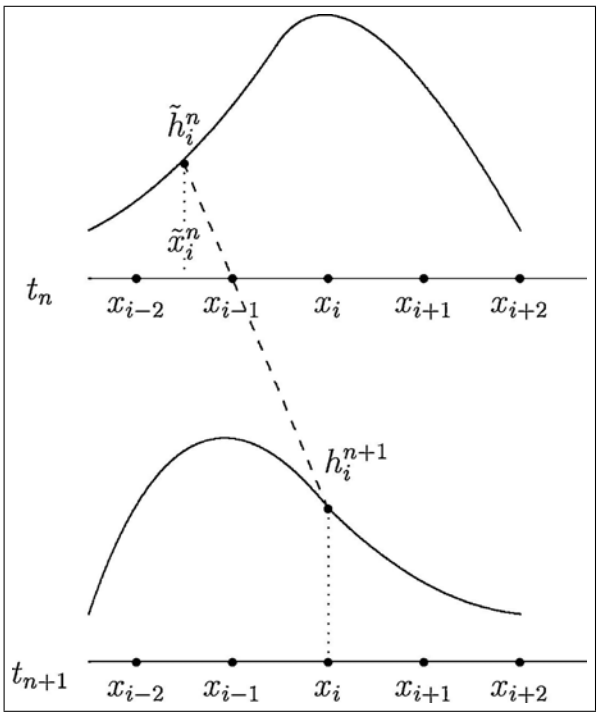


Figura 2.1: Con el esquema Semi-Lagrangiano, la derivada Lagrangiana es aproximada a lo largo de la trayectoria de la partiícula

$$h_i^{n+1} + \Delta t^2 g \frac{\partial b}{\partial x}\frac{\partial h^{n+1}}{\partial x} - \Delta t^2 g d^n \frac{\partial^2 h^{n+1}}{\partial x^2} = \tilde{h}^n + \Delta t \tilde{u}^n \frac{\partial b}{\partial x} - \Delta t d^n \frac{\partial \tilde{u}^n}{\partial x} \tag{2.10}$$

Discretizando espacialmente la ecuación anterior mediante diferencias centrales divididas se obtiene el siguiente sistema tridiagonal, cuya la solución es el nuevo campo de altura:

$$\begin{aligned} h_i^{n+1} + \Delta t^2 g \left(\frac{b_{i+1} - b_{i-1}}{2\Delta x}\right)\left(\frac{h_{i+1}^{n+1} - h_{i-1}^{n+1}}{2\Delta x}\right) \\ -\Delta t^2 g d_i^n \left(\frac{h_{i-1}^{n+1} - 2h_i^{n+1} + h_{i+1}^{n+1}}{\Delta x^2}\right) \\ = \tilde{h}_i^n + \Delta t \tilde{u}_i^n \left(\frac{b_{i+1} - b_{i-1}}{2\Delta x}\right) - \Delta t d_i^n \left(\frac{\tilde{u}_{i+1}^n - \tilde{u}_{i-1}^n}{2\Delta x}\right) \end{aligned} \tag{2.11}$$

2.2.2. El caso en dos dimensiones

En dos dimensiones, discretizando en base al tiempo, las ecuaciones de aguas poco profundas quedarían como sigue:

$$\frac{u^{n+1} - \tilde{u}^n}{\Delta t} + g\frac{\partial h^{n+1}}{\partial x} = 0 \tag{2.12}$$

$$\frac{v^{n+1} - \tilde{v}^n}{\Delta t} + g\frac{\partial h^{n+1}}{\partial y} = 0 \tag{2.13}$$

$$\begin{aligned}\frac{h^{n+1} - \tilde{h}^n}{\Delta t} - \left(u^{n+1}\frac{\partial b}{\partial x} + v^{n+1}\frac{\partial b}{\partial y}\right)\\ +d^n\left(\frac{\partial u^{n+1}}{\partial x} + \frac{\partial v^{n+1}}{\partial y}\right) = 0\end{aligned} \tag{2.14}$$

Como en el caso de una dimensión, tomamos las derivadas espaciales adecuadas de 2.12 y 2.13 y eliminamos los términos de divergencia 2.14. La ecuación resultado de Helmholtz en h es:

$$\begin{aligned}h^{n+1} + \Delta t^2 g\left(\frac{\partial b}{\partial x}\frac{\partial h^{n+1}}{\partial x} + \frac{\partial b}{\partial y}\frac{\partial h^{n+1}}{\partial y}\right)\\ -\Delta t^2 g d^n\left(\frac{\partial^2 h^{n+1}}{\partial x^2} + \frac{\partial^2 h^{n+1}}{\partial y^2}\right)\\ = \tilde{h}^n + \Delta t\left(\tilde{u}^n\frac{\partial b}{\partial x} + \tilde{v}^n\frac{\partial b}{\partial y}\right) - \Delta t d^n\left(\frac{\partial \tilde{u}^n}{\partial x} + \frac{\partial \tilde{v}^n}{\partial y}\right)\end{aligned} \tag{2.15}$$

Esta es la ecuación final después de discretizar.

$$\begin{aligned}h_{i,j}^{n+1} + \Delta t^2 g\left(\frac{b_{i+1,j} - b_{i-1,j}}{2\Delta x}\frac{h_{i+1,j}^{n+1} - h_{i-1,j}^{n+1}}{2\Delta x}\right.\\ \left.+\frac{b_{i,j+1} - b_{i,j-1}}{2\Delta y}\frac{h_{i,j+1}^{n+1} - h_{i,j-1}^{n+1}}{2\Delta y}\right)\\ -\Delta t^2 g d_{i,j}^n\left(\frac{h_{i-1,j}^{n+1} - 2h_{i,j}^{n+1} + h_{i+1,j}^{n+1}}{\Delta x^2}\right.\\ \left.+\frac{h_{i,j-1}^{n+1} - 2h_{i,j}^{n+1} + h_{i,j+1}^{n+1}}{\Delta y^2}\right)\\ = \tilde{h}_{i,j}^n + \Delta t\left(\tilde{u}_{i,j}^n\frac{b_{i+1,j} - b_{i-1,j}}{2\Delta x} + \tilde{v}_{i,j}^n\frac{b_{i,j+1} - b_{i,j-1}}{2\Delta y}\right)\end{aligned}$$

$$-\Delta t d_{i,j}^n \left(\frac{\tilde{u}_{i+1,j}^n - \tilde{u}_{i-1,j}^n}{2\Delta x} + \frac{\tilde{v}_{i,j+1}^n - \tilde{v}_{i,j-1}^n}{2\Delta y} \right) \tag{2.16}$$

La cual se resuelve con el método del gradiente conjugado (Hestenes y Stiefel 1952).La velocidad del agua u^{n+1} y v^{n+1} puede ser actualizada mediante la sustitució de h^{n+1} en 2.12 y 2.13.

2.2.3. Cálculo de una trayectoria

Anteriormente se describió en detalle como se calculan los puntos de partida en la sección 2.2.1. En 2.8 y 2.9, $\tilde{u}^n$ y $\tilde{h}^n$ son evaluados en los *puntos de partida* $(x_i - \alpha_i^n)$ en el momento t_n como se ve en la figura 2.2, donde α_i^n es el desplazamiento de una partícula de fluido en el intervalo de tiempo desde t_n hasta t_{n+1}, finalizando en el punto más bajo x_i. El desplazamiento α_i^n puede ser calculado integrando hacia atrás en el tiempo.
Con una aproximaón de primer orden,

$$\alpha_i^n = \Delta t u^n(x_i) \tag{2.17}$$

Note que los puntos de partida suelen ser puntos fuera de la malla. Esto significa que los valores $\tilde{u}^n$ y $\tilde{h}^n$ pueden no ser conocidos, en cuyo caso se aproximan por la interpolación lineal. Dejemos que x_m denote el m-ésimo punto de la maya: $x_m \equiv m\Delta x$. Supongamos que $x_{m-1} < x_i - \alpha_i^n < x_m$; luego

$$\tilde{u}(x_i) \equiv u^n(x_i - \alpha_i^n) = \frac{(x_i - \alpha_i^n - x_{m-1}u^n(x_m) + (x_m - x_i + \alpha_i^n)u^n(x_{m-1})}{\Delta x} \tag{2.18}$$

Una función similar se puede aplicar para obtener los valores de las alturas en los puntos de partida.

2.2.4. Solución de la estabilidad y presición

Los métodos de integración de tiempo explícitos, por su simplicidad, se utiliza en muchas implementaciones de modelos de fludios (Foster and Metaxas 1996 [5], Chen and Lobo 1995 [7], Chen et al. 1997 [8]. Sin embargo, la principal desventaja de los métodos explícitos de integración es que una rigurosa restricción se impone en el tamaño de los intervalos de tiempo. En otras palabras, a menos que se elija un pequeño intervalo de tiempo, la solución numérica puede diferir de manera exponencial de la verdadera solución. Al elegir un meétodo de integración implícita, hemos asegurado que la estabilidad no está limitada por la magnitud de los términos del gradiente (el segundo y tercer término en 2.13 y 2.14 respectivamente).

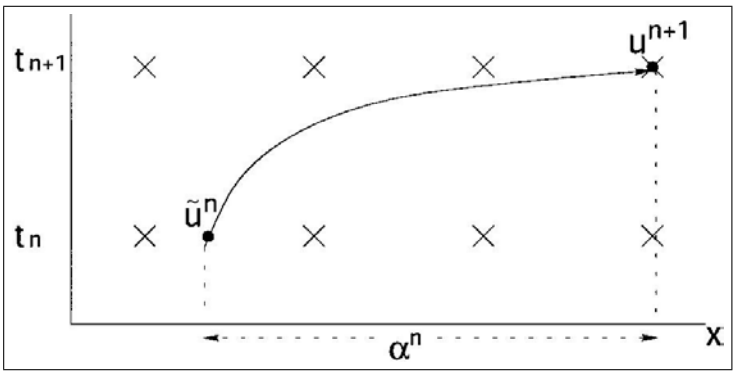


Figura 2.2: Ilustración del método de integración de tiempo Semi-Lagrangiano

Al adoptar el enfoque semi-lagrangiano, también nos aseguramos de que el tamaño de intervalos de tiempo no está limitado por la condición de Courant-Friedrichs Lewy (CFL) [4]. Nuestro algoritmo es estable siempre y cuando la salida de puntos se estimen con suficiente presición. En otras palabras, siempre y cuando los verdaderos puntos de $(xi - \alpha in)$ caen entre 2 puntos de la cuadrícula ($x_m - 1$ y x_m) utilizando la interpolación lineal 2.18 para estimar los valores de la función corriente arriba, la integración sigue siendo estable. Vamos a evitar un largo análisis matemático, pero es suficiente con decir que mientras el producto del paso de tiempo Δt y el viento puro $(\max(|ux|, |uy|, |vx|, |vy|))$ está limitada por alguna constante, la estabilidad numérica está garantizada Pudykiewicz et al. 1985 [1]. La estabilidad del algoritmo se compara experimentalmente a un método explíto.

Una solución numérica se considera inestable cuando, durante un largo período de simulación, el resultado no se parece mucho a la solución que era de esperarse. En todos los casos de prueba, el máximo intervalo de tiempo obtenido por el método implícito semi-lagrangiano es muchas vences mayor que para el método explícito. Esta relajación en la restricción de los intervalos de tiempo permite a un número mucho mejor de intervalos de tiempo que deben adaptarse para la misma longitud de la simulación, reduciendo así el coste total de cómputo tremendamente. Este modelo se adapta completamente a las ecuaciones de aguas poco profundas (sin términos de coriolis) para que las simulaciones puedan ser naturales y realistas. En consecuencia, esta solución debe ser más precisa en lo físico que por ejemplo el método de Kass and Miller 1990 [11], quien basado en su modelo en una versión lineal de las ecuaciones de aguas poco profundas, también sin términos de Coriolis. Nuestra solución numérica es de primer orden en el tiempo y espacio, y, como se señaló anteriormente en esta sección, es relativamente estable, incluso para grandes intervalos de tiempo.

2.2.5. Volumen y conservación de la energía

Se estudiaron las propiedades conservadoras de nuestro algoritmo utilizando los casos antes mencionados. Sin ningún tipo de adición o eliminación del agua, el volumen total, calculado mediante la integración de la profundidad del agua más el dominio del agua, debe permanecer constante a lo largo de la simulación. En todos nuestros experimentos, el cambio en volumen del agua era menos de 5 %, lo que representa una variación es pequeña y apenas perceptible, consideramos que es aceptable.
Por otro lado, nuestro algoritmo, aunque parezca no conservar la energía, la energía total E del agua se calcula como el cuadrado de la magnitud de su velocidad: $E = u^2 + v^2$. Para una simulación lo suficientemente larga, las olas finalmente se amortiguan y la superficie del agua vuelve a su estado calmado y tranquilo. A primera vista, esto parece ser una decepción, ya que no hay término de amortiguamiento en las ecuaciones de aguas poco profundas, lo que implica que una vez iniciada una ola, nunca debe desaparecer.

Sin embargo, también hay que tener en cuenta los efectos de la interpolación espacial y la naturaleza implícita del esquema de integración, por lo que al introducir amortiguación numérica, lo que provoca que la ola pierda su energìa, reduzca la velocidad y posteriormente se calme. Este efecto está de acuerdo con la realidad y esto lo podemos observar en nuestra vida cotidiana, cuando observamos como las olas de un tanque de agua se desaparecen lentamente. Por otra parte, los líquidos reales pierden energía debido a su viscosidad mayor a cero. La pérdida de energía, para nuestras simulaciones, se produce por las mismas razones que contribuyen a la estabilidad del sistema: los valores de la velocidad se obtienen por interpolación dentro de la vecindad de los puntos en la malla y la integración es implícita. Por lo tanto, la pérdida de energía es un compromiso necesario para el logro del método de integración estable.

2.2.6. Eficiencia

En esta sección damos un análisis simple de la complejidad numérica de nuestro algoritmo. La mayor parte del esfuerzo de cálculo entra en la solución de la ecuación de Helmholtz 25. Sea N el número de subintervalos de una dimensión y k el número de iteraciones del método del gradiente conjugado, empleado en la solución 2.16. Al tomar ventaja de las matrices esparcidas, el método del gradiente conjugado tiene complejidad de O (N^2k) Saad and Schultz 1986 [**?**]. En total, en nuestras simulaciones, una o dos iteraciones fueron suficientes para alcanzar el nivel de precisión deseado. Por lo tanto nuestro algoritmo tiene una complejidad general de O (N^2).
Comparamos la eficiencia de nuestro método a la propuesta por Kass and Miller 1990 [11]. En su trabajo, se asumió que la velocidad del agua varía lentamente

en el espacio, lo que permite una versión lineal de las ecuaciones de aguas poco profundas (8) - (10) para ser utilizadas, sin los términos de advección no lineales. La ecuación para h, despues de eliminar u y v del sistema, queda de la forma:

$$\frac{\partial^2 h}{\partial t^2} = gd\left(\frac{\partial^2 h}{\partial x^2} + \frac{\partial^2 h}{\partial y^2}\right), \tag{2.19}$$

2.2.7. Matrices esparcidas

Se llama matriz esparcida a una matriz en la cual la mayoría de sus elementos son cero. Se dice que una matriz es esparcida o dispersa cuando se puede hacer uso de técnicas especiales para sacar ventaja del gran número de elementos cero que posee. como ejemplo la figura 2.3 muestra una matriz esparcida

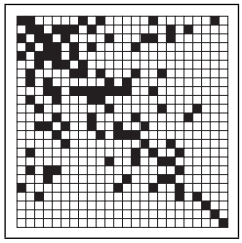


Figura 2.3: Ejemplo de una matriz esparcida

Hay dos tipos de matrices esparcidas:

- Matrices estructuradas: Los elementos no cero forman un patrón regular, por ejemplo, se agrupan a lo largo de un número pequeño de diagonales.
- Matrices no estructuradas: Los elementos no cero se distribuyen de forma irregular.

En el primer caso se pueden diseñar métodos basados en la estructura de las matrices, mientras que en el segundo caso sólo se puede hacer uso de la distribución de los elementos no nulos de la matriz.

Algunos de los esquemas de almacenamiento más usados y en los que se trabajará en esta investigación son los siguientes:

- Compressed Row Storage (CRS)
- Compressed Column Storage (CCS)

El formato CRS y el CCS son los más generales, ellos no toman en cuenta ningún supuesto acerca del la estructura de la distribución de los elementos de

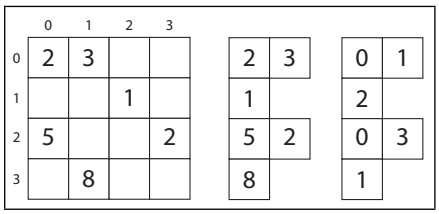


Figura 2.4: Matriz esparcida comprimida por filas

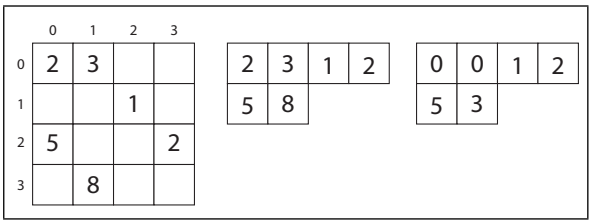


Figura 2.5: Matriz esparcida comprimida por columnas

la matriz y ellos no almacenan ningún elemento innecesario. Las figuras 2.4 y 2.5 ilustran como se comprimen las filas o las columnas para este tipo de formato, estas figuras muestran primero la matriz densa original, luego los valores de la matriz densa comprimidos por su correspondiente formato, y posteriormente muestra los índices ya sea de las filas o columnas de esos valores ya comprimidos.

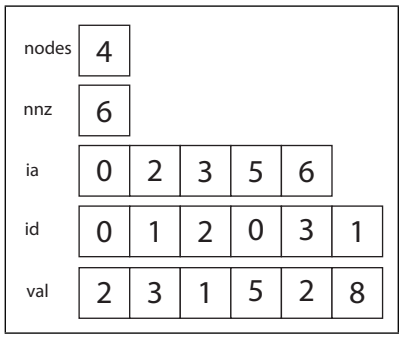


Figura 2.6: Matriz Esparcida almacenada en el formato CSR

El formato CRS coloca los subsecuentes elementos no nulos de la matriz en direcciones contiguas de memoria. Asumiendo que tenemos una matriz no simétrica A, se crean tres vectores: uno con números punto flotante (val), y los otros dos con números enteros (ia, id). El vector val almacena los valores de los diferentes de cero de la matriz A, en el orden que ellos se obtienen cuando se recorre la matriz por fila. El vector ia almacena los índices de las columnas de los elementos en el vector val. Esto es, si $val(k) = a_{i,j}$, entonces $ia(k) = j$. El vector id almacena las posiciones en el vector val que inician una fila; asi, si $val(k) = a_{i,j}$, entonces $id(i) \leq k < id(i+1)$. Por convención, se define $id(n+1) = nnz$, donde nnz es el

número de elementos diferentes de cero en la matriz A, la figura 2.6 muestra una matriz almacenada usando este formato. Los ahorros en almacenamiento por este formato son significativos. En lugar de almacenar n^2 elementos, sólo utilizamos $2nnz + n + 1$ localidades de almacenamiento.

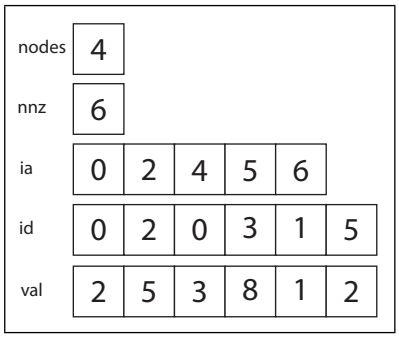


Figura 2.7: Matriz Esparcida almacenada en el formato CCS

Análogo al formato CRS, hay un formato comprimido por columna (CCS). El formato CCS es idéntico al formato CRS, excepto que las columnas de A se almacenan (subsecuentemente) en vez de las filas. En otras palabras, el formato CCS es el formato CRS para AT. La figura 2.7 muestra la misma matriz almacenada en este formato.

2.2.8. Biblioteca UCSparseLib

La biblioteca numérica UCSparseLib fue diseñada (en principio) para resolver grandes sistemas de ecuaciones lineales, tanto en formato denso como disperso. Para tal efecto la biblioteca fue creada con un conjunto de módulos bases, tales como: Matrix (rutinas de lectura y escritura de matrices), Tools (rutinas de utilidades), Factorization (métodos directos), Iterative (métodos iterativos), entre otros, que permiten definir las estructuras básicas para la resolución de sistemas lineales. Esta biblioteca fue desarrollada por el Dr. Germán Larrazábal, e implementada en lenguaje C (estándar ANSI).

2.2.9. OpenGL

OpenGL (Open Graphics Library) es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D. La interfaz consiste en más de 250 funciones diferentes que pueden usarse para dibujar escenas tridimencionales complejas a partir de

primitivas geométricas simples, tales como puntos, líneas y triángulos. Fue desarrollada originalmente por Silicon Graphics Inc. en 1992 y se usa ampliamente en CAD, realidad virtual, representación científica, visualización de información y simulación de vuelo. También se usa en desarrollo de videojuegos, donde compite con Direct3D en plataformas Microsoft Windows.
OpenGL tiene dos propósitos esenciales:

- Ocultar la complejidad de la interfaz con las diferentes tarjetas gráficas, presentando al programador una API única y uniforme.
- Ocultar las diferentes capacidades de las diversas plataformas hardware, requeriendo que todas las implementaciones soporten la funcionalidad completa de OpenGL (utilizando emulación software si fuese necesario).

El funcionamiento básico de OpenGL consiste en aceptar primitivas tales como puntos, líneas y polígonos, y convertirlas en píxeles. Este proceso es realizado po una pipeline gráfica conocida como Máquina de estados de OpenGL. La mayor parte de los comandos de OpenGL bien emiten primitivas a la pipeline gráfica o bien configuran cómo la pipeline procesa dichas primitivas. OpenGL es una API basada en procedimientos de bajo nivel que requiere que el programador dicte los pasa exactos necesarios para renderizar la escena. Esto contrasta con las APIs descriptivas, donde un programador sólo debe describir la escena y puede dejar que la biblioteca controle los detalles para representarla. El diseño de bajo nivel de OpenGL requere que los programadores conozcan en profundidad la pipeline gráfica, a cambio de darles la libertad para implementar algoritmos gráficos novedosos.

OpenGL he influido en el desarrollo de las tarjetas gráficas, promocionando un nivel básico de funcionalidad que actualmente es común en el hardware comercial; algunas de esas contribuciones son:

- Primitivas básicas, líneas y polígonos rasterizados.
- Una pipeline de transformación e iluminación.
- Z-buffering.
- Mapeado de texturas.
- Alpha blending.

Una descripción somera del proceso de la pipeline gráfica podría ser:

- Evaluación, si procede, de las funciones polinomiales que definen ciertas entradas, como las superficies NURBS, aproximando curvas y la geometría de la superficie.

- Operaciones por vértices, transformándolos, iluminándolos según su material y recortando partes no visibles de la escena para reducir el volumen de visión.
- Rasterizaciíon, o conversión de la información previa en píxeles. Los polígonos son representados con el color adeacuado mediante algoritmos de interpolación.
- Operaciones por fragmentos o segmentos, como actualizaciones según valores venideros o ya almacenados de profundidad y de combinaciones de colores, entre otros.
- Por último, los fragmentos son volcados en el Frame Buffer.

Muchas tarjetas gráficas actualmente proporcionan una funcionalidad superior a la básica aquí expuesta, pero las nuevas características generalmente son mejoras de esta pipeline básica más que cambios revolucionarios de ella. Para propositos de este trabajo se uso la versión de OpenGL 2.0.

2.2.10. Especificación de los materiales

Para cada polígono de la escena hay que definir un material de forma que su respusta a la incidencia de luz varíe según sea el caso.

Por tanto tenemos que decirle a OpenGL de que forma tendrá que tratar a cada trozo de geometría.

Se definen 5 caracterśticas fundamentales para un material. Estos componentes son:

- **Reflexión difusa** (diffuse) o color de base que reflejaría el objeto si incidiera sobre él una luz pura blanca.
- **Reflexión especular** (specular), brillo del material.
- **Reflexión ambiental** (ambient), define como un objeto (polígono) determinado refleja la luz que no viene directamente de una fuente de luminosa sinó de la escena en sí.
- **Coeficiente de brillo** o "shininess". Define la cantidad de puntos luminosos y su concentración.
- **Coeficiente de emisión** (emission) o color de la luz que emite el objeto.

Las componentes ambiental y difusa son típicas iguales o muy semejantes. La componente especular suele ser gris o blanca. El brillo nos determina el tamaño del punto de máxima reflexión de luz.
Se pueden especificar diferentes parámetros en cuanto a material para cada polígono.

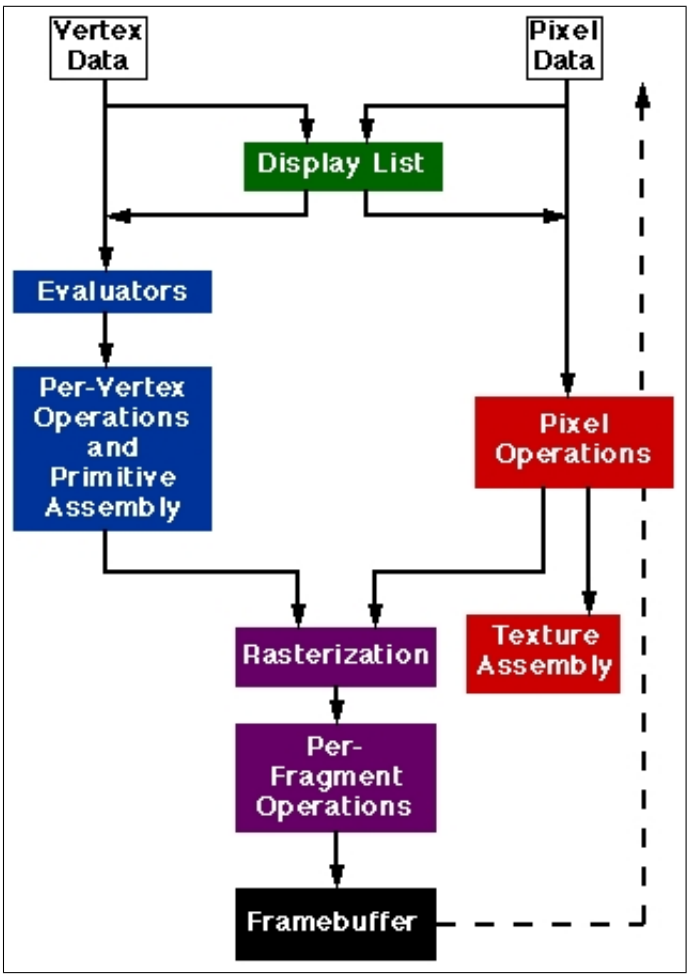


Figura 2.8: OpenGL Rendering Pipeline

Por tanto todo lo que se renderice.ª partir de una llamada heredará esas características. La función es:

```
GLvoid glMaterialfv(GLenum face, GLenum pname,const GLfloat *
   params );
```

Código 2.1: Función para ajustar propiedades de un material en OpenGL

GLenum face	**GLenum pname**	**const GLfloat *params**
GL_FRONT	GL_DIFFUSE	(R, G, B, 1.0)
GL_BACK	GL_AMBIENT	(R, G, B, 1.0)
GL_FRONT_AND_BACK	GL_AMBIENT_AND_DIFFUSE	(R, G, B, 1.0)
	GL_EMISION	(R, G, B, 1.0)
	GL_SPECULAR	(R, G, B, 1.0)
	GL_SHININESS	[0, 128]

Variables y parametros para la configuración de materiales en OpenGL

En la tabla se observan los valores que pueden adoptar los parámetros de la función. En el caso de **face** tenemos tres posibilidades dependiendo si la característica en cuestión debe aplicarse al lado visible (FRONT), al no visible (BACK) o a ambos. En cuanto a **pname** se define aquí cuál es la característica que vamos a definir en concreto. Las posibles son las que hemos comentado para un material. Por último ***params**, donde damos los valores concretos de la característica del color, estan dado en RGB (red, green, blue).
Hay una excepción en el caso de GL_SHININESS. Si usamos esta constante como segundo parámetro, el tercero tendrá que ser un número entre 0 y 128 que controlará la concentración del brillo. Por defecto este valor es 0.

2.2.11. Iluminación en OpenGL

Las escenas en OpenGL tienen varias fuentes de luz que pueden estar apagadas o encendidas independientemente. La luz puede estar iluminando toda la escena (Luz Ambiental) o solamente una dirección determinada. Las fuentes de luz sólo hacen su efecto cuando la escena consta de superficies u objetos que absorben o reflejan luz.

La iluminación se divide en cuatro tipos: emitida, ambiente, difusa y especular.

Para definir las fuentes de luz tenemos que definir primero sus propiedades: color, posición, dirección. Usamos glLight:

```
void glLight(GLenum luz, GLenum nombre, TYPOparam);
```

Código 2.2: Función glLight para iluminación en OpenGL

Se crea la fuente de luz que indica el parámetro luz, hay 8 fuentes de luz en la definición estandar: LIGHT0,...,LIGHT7. El argumento nombre especifica las características de la luz, que son unos parámetros ya definidos en OpenGL:

```
GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_POSITION,
   GL_SPOT_DIRECTION, GL_SPOT_EXPONENT, GL_SPOT_CUTOFF,
   GL_CONSTANT_ATTENUATION, GL_LINEAL_ATTENUATION,
   GL_QUADRATIC_ATTENUATION.
En donde GL_DIFFUSE y GL_SPECULAR solo se usan con la primera
   fuente de luz.
```

Un ejemplo de glLight seria:

```
glLightfv(GL_LIGHT0, GL_AMBIENT, Luz_ambiental);
```

Código 2.3: Ejemplo de Función para glLight

En cuanto al color lo indicamos utilizando sus componentes:

- **Ambiente:** aportación a la luz ambiental de la escena.
- **Difusa:** color de la fuente de la luz.
- **Especular:** color del brillo especular que produce la fuente de luz.

Puede haber dos tipos de fuentes de luz según la posición.

- **Direccionales:** (x,y,z,0). Marcamos el vector dirección y la luz se sitúa en el infinito.
- **Posicionales:** (x,y,z,w). Marcamos la posición en coordenadas homogéneas y la luz se sitúa en la escena.

Las transformaciones que se le hacen a los objetos con la matriz modelo/vista también las sufren las fuentes de luz.

2.2.12. El buffer de color

El buffer de color contiene información de color de los píxeles. Cada píxel puede contener un índice de color o valores rojo/verde/azul/alfa que describen la apariencia de cada píxel. Los píxeles RGBA se vizualizan directamente en pantalla utilizando el color más próximo disponible. La apariencia de los píxeles de color con índice viene determinada por el valor asociado a este índice en una tabla de color RGB.

2.2.13. Doble Buffer

El doble buffer proporciona un buffer de color adicional fuera de la pantalla que se usa a menudo para animación. Con el doble buffer podemos dibujar una escena fuera de la pantalla e intercambiarla rápidamente con la que está en pantalla, eliminando el parpadeo.

El doble buffer sólo afecta el buffer de color y no proporciona un segundo buffer de profundidad, acumulación o estarcido. Si elegimos un formato de píxel con doble buffer, OpenGL selecciona el buffer oculto para dibujar. Podemos cambiar esto usando la función glDrawBuffer para especificar uno de los siguientes valores:

Buffer	**Descripción**
GL_FRONT	Dibujamos sólo en el buffer de color frontal (visible)
GL_BACK	Dibujamos sólo en el buffer de color trasero (oculto)
GL_FRONT_AND_BACK	Dibujamos en ambos buffers de color

2.2.14. Intercambio de buffers

OpenGL soporta doble buffer, pero no hay ninguna función para intercambiar los buffers frontal y oculto. Afortunadamente, cada sistema de ventanas con OpenGL soporta una función para hacerlo. Bajo Windows, esta llamada es:

```
SwapBuffers(hdc);
```

Código 2.4: Intercambio de Buffers

Donde hdc es el contexto de dispositivo de la ventana en la que estamos dibujando. Si estamos usando GLUT, éste ya se encargará de hacerlo por nosotros automáticamente.

2.2.15. Transparencias

Antes de empezar a hablar de cómo implementar transparencias en OpenGL, hablaremos un poco de la función de blending (mexclar). la mezcla de OpenGL proporciona un control a nivel de píxel del almacenamiento de valores RGBA en el buffer de color. Las operacione de mezcla no pueden emplearse en el modo indexado de color y están desactivadas en las ventanas de color indexado. Para activar la mezcla en ventanas RGBA, primero debemos invocar a glEnable(GL_BLEND). Tras ésto, llamamos a glBlendFunc con dos argumentos: las funciones origen y destino para la mezcla de color, que se especifican en las siguientes tablas.

Por defecto, estos argumentos son GL_ONE y GL_ZERO, respectivamente, lo que equivale a glDisable(GL_BLEND).

Función Origen	**Descripción**
GL_ZERO	Color fuente = 0, 0, 0, 0
GL_ONE	Uso $\langle ? \rangle ColorFuente$
GL_DST_COLOR	El color de origen se multiplica por el color de píxel de destino
GL_ONE_MINUS_DST_COLOR	El color de origen se multiplica por (1, 1, 1, 1; color de destino)
GL_SRC_ALPHA	El color de origen se multiplica por el valor alfa de origen
GL_ONE_MINUS_SRC_ALPHA	El color de origen se multiplica por (1, valor alfa de origen)
GL_DST_ALPHA	El color de origen se multiplica por el valor alfa de destino. Microsoft OpenGL no lo soporta
GL_ONE_MINUS_DST_ALPHA	El color de origen se multiplica por (1, valor alfa de destino). Microsoft OpenGL no lo soporta
GL_SRC_ALPHA_SATURATE	El color de origen se multiplica por el mínimo de los valores alfa de origen y (1, valor de destino). Microsoft OpenGL no lo soporta

La transparencia es quizás el uso más típico de las mezclas, empleada a menudo en ventanas, botellas y otros objetos 3D a través de los cuales podemos ver. Estas son las funciones de mezcla para estas aplicaciones:

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Código 2.5: Funciones para configurar transparencias

Esta combinación toma el color de origen y lo escala basándose en la componente alfa, para sumarle luego el color de destino escalado en 1 menos el valos alfa. Dicho de otra manera, esta función de mezcla toma una fracción de color de dibujo actual y cubre con ella el píxel que hay en pantalla. La componente alfa del color puede valer de 0 (totalmente transparente) a 1 (totalmente opaco), como sigue:

```
Rd = Rs * As + Rd * (1 - As)
Gd = Gs * As + Gd * (1 - As)
```

```
Bd = Bs * As + Bd * (1 - As)
```

Código 2.6: Valores de transparencia

Dado que sólo se emplea la componente alfa de color de origen, no necesitamos una tarjeta gráfica que soporte planos de color alfa en el buffer de color. Algo que tenemos que recordar con la transparencia de mezcla alfa es que la verificación normal de profundidad puede interferir con el efecto que tratamos de conseguir. Para asegurarnos de que las líneas y polígonos transparentes se dibujen apropiadamente, debemos dibujarlos siempre de atrás hacia adelante. Aquí tenemos un ejemplo del uso de transparencias que dibuja una tetera opaca que se ve a través de otra transparente.

Figura 2.9: Ejemplo de transparencia en OpenGL

2.3. Términos y Definiciones

Método de las caracteristicas En matemáticas, el método de las características es una técnica para la solución de ecuaciones diferenciales parciales

Advección En la física, la ingeniería, y ciencias de la tierra, la advección es un mecanismo de transporte de una sustancia o de los bienes conservados por un fluido debido al movimiento a granel de líquidos. Un ejemplo de advección es el transporte de contaminantes o sedimentos de un río por el flujo de agua en grandes cantidades de aguas abajo.

API Interfaz de programación de aplicaciones, es el conjunto de funciones y procedimientos que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

CAD el diseño asistido por computadora, más conocido por sus siglas inglesas CAD (computer-aided- design), es el uso de un amplio rango de herramientas computacionales que asisten a ingenieros, arquitectos y diseñadores.

Direct3D es parte de DirectX, propiedad de Microsoft. Consiste en una API para la programación de gráficos 3D.

Pipeline La segmentación (en inglés pipelining, literalmente tubería) es un método por el cual se consigue aumentar el rendimiento de algunos sistemas electrónicos digitales.

Rasterización La rasterización es el proceso por el cual una imagen descrita en un formato gráfico vectorial se convierte en un conjunto de píxeles o puntos para ser desplegados en un medio de salida digital, como en una pantalla de computadora, una impresora electrónica o una imagen de mapa de bits (bitmap).

Z-buffer En los gráficos por computadora, el z-buffering es la parte de la memoria de un adaptador de video encargada de gestionar las coordenadas de profundidad de las ímagenes en los gráficos en tres dimensiones (3-D), normanmente calculados por hardware y en algunas veces por software.

NURBS B-splines racionales no uniformes o NURBS, es un modelo matemático muy utilizado en la computación gráfica para generar y representar curvas y superficies.

FrameBuffer Se le llama framebuffer a una categoría de dispositivos gráficos, que representan cada uno de los píxeles de la pantalla como ubicaciones de la memoria de acceso aleatorio.

Capítulo 3

Marco Metodológico

–Área de Investigación: Computacion Gráfica

3.1. Metodología de Desarrollo

El desarrollo de esta investigación está basado en una metodología típica de proyectos de investigación en el área de ciencias aplicadas. Los pasos a seguir vienen dados por las etapas propias del método científico:

3.1.1. Observación

La primera fase está definida por la detección de un problema y una revisión bibliográfica en función de establecer el estado del arte en el área de la investigación del problema detectado, así como la identificación de las estrategias más adecuadas para abordar el problema. En este caso, el problema observado es la necesidad de simular ondas de movimiento de agua con apariencia realista a un bajo costo computacional.

3.1.2. Planteamiento de la hipótesis

Seguidamente se planteará una solución hipotética del problema. En este caso nuestra hipótesis seria, ¿Será posible construir una aplicación de software que permita modelar y visualizar animaciones realistas del movimiento del agua en una superficie poco profunda con tiempo de ejecución interactivos?.

3.1.3. Experimentación

Posteriormente, la solución hipotética planteada será implementada con el objeto de validar su eficacia en la solución del problema observado. En este caso, la fase de experimentación viene definida por el diseño e implementación de una aplicación de software usando Opengl y C como lenguajes de programación para

la ejecución de una animación basada en el modelo matemático escogido. El desarrollo de este experimento es detallado en los capítulos "Descripción de la solución propuestaz Resultados".

3.1.4. Validación y Análisis

Una fase de validación y reporte de resultados, junto a un análisis y comparación de los resultados obtenidos con el desarrollo de la aplicación y los resultados reportados de otros estudios culminará el proyecto.

Finalmente, dado que la fase de experimentación supone la construcción de una aplicación de software, se hace necesario echar mano de alguna estrategia metodológica para el desarrollo de esta aplicación. En nuestro caso, usaremos una metodología basada en el ciclo de vida y el uso de prototipos incrementales. En tal sentido, el desarrollo de la aplicación supondrá la realización de las siguientes etapas:

3.1.5. Análisis y definición de requerimientos

Consiste en la identificación de los servicios, restricciones y metas del sistema se definen a partir de la comprensión del dominio de información del software, así como la función requerida, comportamiento, rendimiento e interconexión.

3.1.6. Diseño de sistema y del software

Consiste en el proceso de diseño del sistema tanto a nivel de hardware como de software. Establece la arquitectura completa del sistema. El diseño del software identifíca y describe las abstracciones fundamentales del sistema software y sus relaciones.

3.1.7. Implementación y pruebas de unidades

Durante esta etapa, el diseño del software se concreta como un conjunto de unidades de programas. La prueba de unidades implica verificar que cada una cumpla su especificación.

3.1.8. Integración y prueba de sistema

Los programas o las unidades individuales de programas se de integran y prueban como un sistema completo para asegurar que se cumplan los requerimientos del software.

3.1.9. Funcionamiento y mantenimiento

Por lo general, esta es la fase más larga del ciclo de vida. El sistema se instala y se pone en funcionamiento práctico. El mantenimiento implica corregir errores no descubiertos en las etapas anteriores del ciclo de vida.

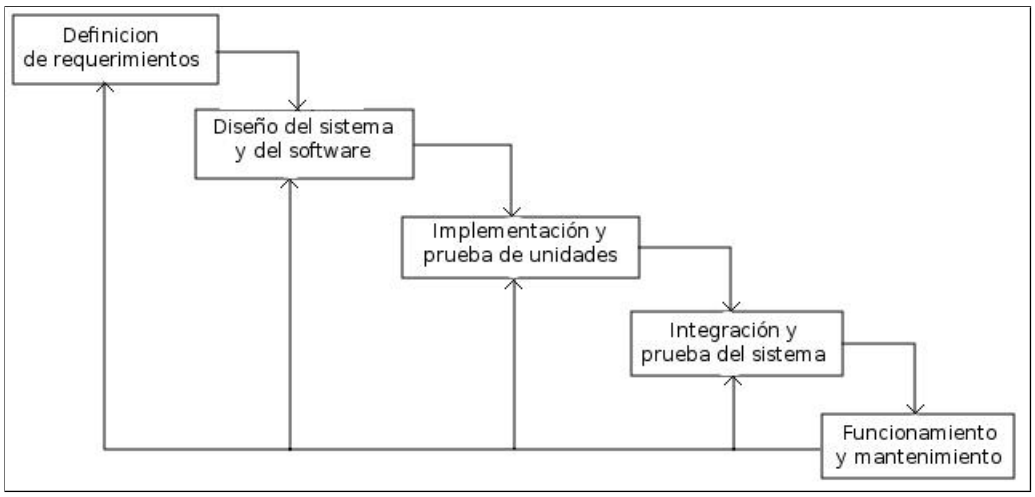


Figura 3.1: Modelo de cascada o ciclo de vida básico

Capítulo 4

Propuesta de Solución

4.1. Planteamiento de la Solución

Se propone como primera instancia, desarrollar un algoritmo que simule el movimiento de una superficie de agua, en donde se pueda observar la propagación de las ondas al ejercer fuerza sobre la superficie. El principal objetivo es poder generar la animación en tiempo real y que al mismo tiempo tenga un aspecto realista, de manera que los calculos computacional sean lo más bajo posible. Para poder obtener unos resultados de manera rápida debemos conseguir un método de resolución de matrices esparcidas y sistemas de ecuaciones que nos permitan obtener los resultados de forma inmediata para poder procesarlos y renderizarlos.

Por otro lado el desarrollo de una interfaz lo suficientemente creible para poder representar los movimientos del agua y la configuración optima de los parametros para que la simulación sea estable y mantenga su uniformidad. Otro dato importante es la memoria, se necesita identificar un método que permita almacenar complejas matrices y que al mismo tiempo se puedan procesar rápidamente, para esto usaremos matrices esparcidas además de la memoria dinámica, de modo que siempre este en memoria los datos que se estan procesando, esto es devido a la cantidad de matrices que se generar para obtener la simulación además de la dimensión de la malla que incrementa significativamente los tiempos de cálculo.

Para obtener el máximo rendimiento de la memoria procesamos los datos de forma dinámica de la siguiente manera:

- Una estructura de dato tipo Punto que contiene los datos de cada punto en (x,y,z).
- Una estructura de dato tipo Vector que contiene los datos de la normal de cada punto, este lo usaremos solo para los calculos de render.

- Una matriz que almacena todas ecuaciones del sistema dependiendo de la posición de cada punto, esta matriz contiene la mayor cantidad de datos y depende mucho de la dimensión del sistema.

- Una matriz que almacena el punto de partida de la altura de cada punto, una matriz que contiene la altura del sub-suelo de cada punto, dos matriz que contiene los datos de fuerza de dirección en u y v de cada punto, dos matrices para los puntos de partida de u y v respectivamente y un vector de terminos que representan las incognitas del sistema de ecuaciones.

Debido a las diferentes técnicas y métodos existentes debemos conseguir el más estable y que al mismo tiempo consuma la menor cantidad de calculos posible, es por esto que se propone usar el método de integración Semi-Lagrangiano 2.2.1, de modo que al despejar h de la ecuación 2.16, obtenemos la ecuación resultado:

Simplificamos diciendo que los siguientes terminos Bi y Bj son de la siguiente manera:

$$Bi = \frac{b_{i+1,j} - b_{i-1,j}}{2} \tag{4.1}$$

$$Bj = \frac{b_{i,j+1} - b_{i,j-1}}{2} \tag{4.2}$$

Usando Bi y Bj respectivamente obtenemos h para cada intervalo de tiempo:

$$= \tilde{h}^n_{ij} + \frac{\Delta t}{2\Delta xy}\left(\tilde{u}_{ij}B_i + \tilde{v}_{ij}B_j\right) - \frac{\Delta t d^n_{ij}}{2\Delta xy}\left(\tilde{u}^n_{i+1,j} - \tilde{u}^n_{i-1,j} + \tilde{v}^n_{i,j+1} - \tilde{v}^n_{i,j-1}\right) \tag{4.3}$$

Una vez obtenido la ecuación para las alturas, generamos el sistema de ecuaciones para la malla y los puntos que la conforman. La representación seria como se muestra:

$$\begin{bmatrix} h_{0,0} & h_{0,1} & \cdots & h_{0,n} \\ h_{1,0} & h_{1,1} & \cdots & h_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ h_{m,0} & h_{m,1} & \cdots & h_{m,n} \end{bmatrix} = \begin{bmatrix} x_{0,0} \\ x_{0,1} \\ \vdots \\ x_{n,m} \end{bmatrix}$$

4.2. Almacenamiento de datos

4.2.1. Estructura de datos

Para optimizar el uso de los datos y como son almacenados en memoria, se crearon estructuras de datos y se hacia la reserva de memoria previa a la necesaria para calcular una simulación.

Entre las estructuras de datos que podemos comentar esta la usada para representar un punto en el espacio o la malla. Así mismo también se hizo uso de una estructura similar para almacenar el vector normal al que apunta dicho punto para poder generar reflacciones de la superficie en el render.

```
typedef struct{
    GLfloat x;
    GLfloat y;
        GLfloat z;
}point;

point *Pts=(point*)NULL;

typedef struct{
    GLfloat x;
    GLfloat y;
        GLfloat z;
}vector;
```

Código 4.1: Declaración de estructura de datos

El código 4.1 muestra como se hizo uso de las estructura de datos para cada punto.

4.2.2. Almacenamiento y reservación de espacio de memoria para el sistemas de ecuaciones

Una de las principales caracterque debe tener la aplicación a desarrollar, es la capacidad de consumir la menor cantidad posible de espacio en la memoria del computador para así, poder construir superficies con grandes cantidades de puntos. Es por ello que se procedío a estudiar diversos esquemas de almacenamiento existentes, para lograr almacenar el sistema de ecuaciones de forma rápida y precisa, enfocándonos en aquellos esquemas de almacenamiento que, en un intento por tomar ventaja de la gran cantidad de elementos nulos que poseen las matrices esparcidas, tienen por objetivo principal representar únicamente los elementos no nulos del sistema y aun así, ser capaces de realizar las operaciones comunes de una matriz.

Entre los distintos esquemas de almacenamiento de matrices esparcidas, los más comunes son: el llamado formato de coordenadas (Coordinate Format), el formato CSR por sus siglas en ingles (Compressed Sparse Row), el formato CSC igualmente por sus siglas en ingles (Compressed Sparse Column) y el formato Modified Sparse Row (MSR). Sin embargo, para el caso particular de esta investigación, será utilizado el formato CSR. Debido a su forma de almacenar datos se ajusta mejor a nuestro modelo de simulación.

```
//Matriz de resolucion del sistema de ecuaciones
double **Matriz=(double**)NULL;
//Matriz de las alturas y de Departure Pnts
double **h=(double**)NULL, **dph=(double**)NULL;
//Matriz Altura del Sub-suelo de cada punto
double **b=(double**)NULL;
//Matriz de los U's y departure Pnts U's
double **u=(double**)NULL, **dpu=(double**)NULL;
//Matriz de los V's y departure Pnts V's
double **v=(double**)NULL, **dpv=(double**)NULL;
//Vector de terminos del SE
double *terminos=(double*)NULL;

//Declaración de TDMatrix para UCSparseLib
TDMatrix SM;
```

Código 4.2: Declaración de matrices dinámicas

El código 4.2 muestra como se declararon las matrices, cuantes y como se llaman cada una de las usadas en el algoritmo.
Una vez declaradas todas las variables para las matrices procedemos a reservar los espacios de memoria para cada matriz, incluyendo la matriz de punto que almacena la posición en el espacio de cada punto de la malla. Siendo n la dimensión de la malla.

```
Pts = (point*)malloc((dim)*sizeof(point));

//Iniciamos las demas matrices para realizar los calculos
//Creamos la Matriz de H's
h = (double**)malloc((n)*sizeof(double*));
//Creamos la Matriz de Departure Pnts H's
dph = (double**)malloc((n)*sizeof(double*));
//Creamos la Matriz de B's
b = (double**)malloc((n)*sizeof(double*));
//Creamos la Matriz de U's
u = (double**)malloc((n)*sizeof(double*));
//Creamos la Matriz de Departure Pnts U's
dpu = (double**)malloc((n)*sizeof(double*));
//Creamos la Matriz de V's
v = (double**)malloc((n)*sizeof(double*));
//Creamos la Matriz de Departure Pnts V's
dpv = (double**)malloc((n)*sizeof(double*));

//Creamos el vector de Terminos independientes
terminos = (double*)malloc((dim)*sizeof(double));
for(i=0; i<dim; i++)
                terminos[i] = 0;

//Reservamos el espacion completo para las matrices
for(i=0; i<n; i++){
```

```
        h[i] = (double*)malloc((n)*sizeof(double));
        dph[i] = (double*)malloc((n)*sizeof(double));
        b[i] = (double*)malloc((n)*sizeof(double));
        u[i] = (double*)malloc((n)*sizeof(double));
        dpu[i] = (double*)malloc((n)*sizeof(double));
        v[i] = (double*)malloc((n)*sizeof(double));
        dpv[i] = (double*)malloc((n)*sizeof(double));
        dpv[i] = (double*)malloc((n)*sizeof(double));
}
```

Código 4.3: Reservación del espacio de memoria para las matrices

El código 4.3 se hace evidencia de la reservación de espacio en memoria dada la dimensión de la malla.

Para liberar toda la memória utilizada por el programa creamos un procedimiento el cual se encarga de esta función.

```
void LiberarMemoria(){
        int i;
        for(i=0; i<dim; i++){
                free(Matriz[i]);
        }
        free(Matriz);
        for(i=0; i<n; i++){
                free(h[i]);
                free(b[i]);
                free(u[i]);
                free(v[i]);
        }
        free(h);
        free(b);
        free(u);
        free(v);
        free(terminos);
}
```

Código 4.4: Procedimiento para liberar los datos almacenados en memória

4.3. Desarrollo del Sistema de Ecuaciones

4.3.1. Esquema CSR (Compressed Sparse Row)

El formato de almacenamiento CSR es probablemente el formato más popular para el almacenamiento de matrices esparcidas debido a que, por su estructura propia, facilita la realización de cálculos típicos.

Ahora bien, siendo nz el número de elementos no nulos de la matriz A, la estructura de datos propia del formato CSR se describe a continuación:

Sea,

$$\text{A}= \begin{bmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 0 & 12 \end{bmatrix}$$

- Un arreglo AA, que almacena fila por fila, los valores distintos de cero. Su longitud es de nz.
- Un arreglo de enteros JA, también de dimensión nz, que almacena los índices de las columnas de los elementos diferentes de cero almacenados en el arreglo AA.
- Un arreglo de entreos IA que contiene los puntos de inicio de cada fila en el arreglo AA y JA. Así, el contenido de IA_i es la posición en el arreglo AA y JA donde inicia la i-ésima fila. La longitud de IA es $n+1$, donde IA_{n+1} contiene el índice de AA y JA en el cual se iniciaría una fila ficticia $n+1$.

Así la matriz A en el formato CSR se representa de la siguiente manera:

$$\text{AA}= \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{bmatrix}$$

$$\text{JA}= \begin{bmatrix} 0 & 3 & 0 & 1 & 3 & 0 & 2 & 3 & 4 & 2 & 3 & 4 \end{bmatrix}$$

$$\text{IA}= \begin{bmatrix} 0 & 2 & 5 & 9 & 11 & 12 \end{bmatrix}$$

Para fines de esta investigación, se seleccionó este metodo para representar el sistema de ecuaciones y de esta manera resolverlo empleando la libreria UCSparseLib.

4.3.2. Solución del sistema de ecuaciones

Para solucionar el sistema de ecuaciones debemos tener en cuenta que una vez obtenidos los puntos de partida generamos el sistema de ecuaciones para cada intervalo de tiempo, al mismo tiempo que calculamos el vector de terminos independientes mencionados anterioremente. Para la solución del sistema se hizo uso de la libreria UCSparseLib, dicha libria contiene numerables metodos y formas de realizar operaciones con matrices y al mismo tiempo nos permite resolver sistemas de ecuaciones mediante el uso de distintos métodos.
Para los propositos de este trabajo se utilizo un método iterativo llamado BiCGstab. Sus argumentos son:

- SM matrix del sistema lineal.

- X vector solución.
- terminos vector del lado derecho.
- PM precondicionador.
- iparam parametro entero.
- dparam parametro doble.
- plot Historial de convergencia ($||r||/||rhs||$).
- totiter número total de iteraciones.
- ierr código de error de retorno.

Este método resuelve sistemas de tipo $[AA][M]^{-1}[M]xx = rhs$ y $[AA]xx = rhs$. El método de BiCGstab está basado en el método del gradiente conjugado.

```
//Variables
TDprecond PM;
int totiter, ierr, itcg;
int iparam[5];
double dparam[3];
double *X, *plot, *Y;
int iteraciones = 150;

//Inicializacion de variables
iparam[0]=1;
iparam[1]=iteraciones;  //Numero maximo de iteraciones
iparam[2]=0;                    //Tipo de precondicionador
iparam[3]=0;
iparam[4]=0;                    // 1 = Si se desea registro de la traza;
   0 = caso contrario

dparam[0] = 1.0E-12;    //Epsilon (maximo grado de error permitido)
dparam[1] = 1.0E-12;
dparam[2] = 0.0;
X                       = (double*)malloc(nbrows*sizeof(double));

//Aplicación del metodo

dSet(nbrows, X, 0.0);
if(iparam[4])
        plot = ALLOC( iparam[1]+1, double, "plot");
else
        plot = (double *)NULL;

BiCGstab(SM, X, terminos, &PM, iparam, dparam, plot, &totiter, &
   ierr);
```

Código 4.5: Utilización de la libreria UCSparseLib para la solución del sistema de ecuaciones

En el código 4.5 se puede observar como se emplea la librería UCSparseLib y el método utilizado para obtener la solución del sistema de ecuaciones, en donde SM representa el sistema de ecuaciones, *terminos* representa el vector de terminos independientes y X es donde se almacena la solución para cada varible del sistema de escuaciones.

4.4. Desarrollo de la Simulación

4.4.1. Esquema de la Simulación

Para poder generar la simulacíon o la animación de la superficie de agua se necesitan una serie de parametros y variables a considerar. En este trabajo demostraremos como se pueden obtener diferentes resultados dependiendo de las variables iniciales que se tomen y la forma en que se aplican las fuerzas en la superficie. Uno de los parametros que tenemos que tomar en consideración es g, que no es mas que la fuerza de atracción entre las particulas del cuerpo de agua (mientras mayor sea el valor de g mayor inestabilidad tendra la simulación). Como lo que buscamos es mantener estabilidad y rapidez a la hora de obtener la secuencia de frames se debe considerar un valor lo más optimo posible para obtener una simulación estable.
Otro dato importante a la hora de realizar la simulación es la densidad de las matrices, es decir, el tamaño de la malla que también podriamos interpretar como la resolución de la superficie, mientras más resolución contenga la superficie, mejor será el aspecto de la superficie pero a su vez mayor serán los tiempos de calculo, ademaás de que se necesitará más memoria RAM para almacenar los datos de las matrices. En este trabajo estamos considerando que las dimensiones de la malla o superficie son cuadradas, por lo que m y n serán tomados con un mismo valor. Aparte de la dimensión de la malla también podemos nombrar los valores de Δx y Δy, que no es mas que la distancia entre los puntos de la superficie. Estos valores no influyen en la resolución de la malla pero si en el tamaño de la superficie.

El tiempo es un factor importante para mantener la estabilidad, nuestro método semi-Lagrangiano nos permite ajustar los intervalos de tiempo de la simulación dado el valor de Δt. Podemos interpretar este valor como la cantidad de cuadros por segundo de la animación (fps, frames per second) o la cantidad de Substeps de la animación. Mientras menor sea el número de Δt mejor estabilidad tendrá la simulación y más lento se observará el movimiento de la simulación, debido a que mayor precisión se tendrá para calcular cada moviemiento de una particula en un instante de tiempo (quiere decir, cantidad de calculos por intervalo de tiempo),

mientras que si Δt es un numero grande la simulación se calculará más rápido pero perdiendo estabilidad con el tiempo.

Como se mencionó anteriormente el algoritmo usa matrices con memoria dinámica para poder realizar los calculos rápidamente. El esquema de la simulación esta hecho de la siguiente manera:

- Inicializar h^0, v^0, u^0 y b.
- Para cada intervalo de tiempo t_{n+1}, hacemos (Ciclo de la simulación):
  - Calculamos los puntos de partida usando la ecuación 2.17.
  - Calcular h, u y v en los puntos de partida usando la ecuación 2.18.
  - Resolvemos el sistema de ecuaciones para h^{n+1} 2.16
  - Actualizamos los valores de u^{n+1} y v^{n+1} con las ecuaciones 2.12 y 2.13.
- Luego Renderizamos los valores de h calculados.

Una vez demostrado esto, explicaremos más detalladamente como funciona el algoritmo.

```
void Simulation(){
        int nbrows,i;
        nbrows = dim;    //numero de elementos

        //Calculamos los Puntos de Partida para esta Iteración
        DeparturePoints();
        //Actualizamos el Sistema de Ecuaciones
        ActualizarSistemEcu();
        //Resolvemos el Sistema de Ecuaciones para las alturas (h)
        //------- SOLUCION DEL SISTEMA DE ECUACIONES
        TDprecond PM;
        int totiter, ierr, itcg;
        int iparam[5];
        double dparam[3];
        double *X, *plot, *Y;
        int iteraciones = 150;

        iparam[0]=1;
        iparam[1]=iteraciones;
        iparam[2]=0;
        iparam[3]=0;
        iparam[4]=0;

        dparam[0] = 1.0E-12;
        dparam[1] = 1.0E-12;
```

```
        dparam[2] = 0.0;
        X = (double*)malloc(nbrows*sizeof(double));

        dSet(nbrows, X, 0.0);
        if(iparam[4])
                plot = ALLOC( iparam[1]+1, double, "plot");
        else
                plot = (double *)NULL;

        BiCGstab(SM, X, terminos, &PM, iparam, dparam, plot, &
           totiter, &ierr);

        //Actualizamos el Vector de las Alturas
        ActualizarAlturas(X);
        //Actualizamos las Matrices para U y V en la siguiente iteración
        ActualizarMatricesUV(X);
}
```

Código 4.6: Ciclo de la simulación para cada intervalo de tiempo

4.4.2. Inicializar valores

Para inicializar los valores primero debemos saber la dimensión de la matriz. Debemos obtener los valores de n y m respectivamente para apartar el espacio de memoria que usaremos para la simulación. Es importante conocer para que es utilizado cada vector y cada matriz. Como explicamos anteriormente denotaremos a las matrices y vectores de la siguiente forma:

- h, matriz que almacena la altura de cada punto.
- b, matriz que almacena la altura del subsuelo de cada punto.
- u y v, matrices que almacenan la velocidad en x y y respectivamente.
- dph, dpu y dpv, matrices que almacenan los puntos de partida respectivo de h,u y v.

Una vez reservado el espacio de memoria para dichas matrices inicializamos todas en 0 (cero).

Para conseguir una estable simulación se tuvo que experimentar con distintos valores para g, despues de hacer distintas pruebas obtuvimos un valor bastante optimo para la simulación (0,0006006456). En cuanto a Δt es un valor mas fácil de ajustar, ya que solo depende de cuantos por segundo queremos que se cálcule la simulación. Para $\Delta t = segundo/cuadrosporsegundo$, para esto utilizamos 2 valores 25 y 30 fps, siendo de estar manera $\Delta t = 0{,}041666661$ y $\Delta t = 0{,}033333333$ respectivamente. Para definir los valores de Δx y Δy, solo dependemos de que

tan grande queremos que sea el cuerpo de agua, a mayor distancia mayor tamaño tendrá la superficie, además también pudimos observar que esto influye en la resolución del cuerpo de agua, para obtener una buena resolución debemos usar valores pequeños para Δx y Δy y valores moderados para n (dimensión de la matriz de puntos), si queremos que el cuerpo sea mas grande podemos aumentar la distancia entre los puntos pero perderá resolución y para compensarlo debemos aunmentar la cantidad de puntos y como explicamos anteriormente esto acarrea más cálculos y tiempo de simulación.
Las variables globales usadas en la simulación estan mostradas en el siguiente código.

```
#define g 0.0006006456
#define Pi 3.141592654
#define deltaT 0.033333333      //30 fps
float deltaX = 0.1;
float deltaY = 0.1;
```

Código 4.7: Inicialización de variables globales

4.4.3. Puntos de partida

Para calcular los puntos de partida debemos saber de donde viene la particula a la cual le vamos a asignar la nueva altura, como se puede ver en 2.1. Para esto debemos calcular cual es la velocidad que tienen los vecinos de cada punto con respecto a la particula a considerar. Denotaremos estas aproximaciones de velocidad y altura a UM, Um, VM, Vm, $\tilde{x}$, $\tilde{y}$ y a su vez dpu y dpv. Decimos que para calcular el punto de partida para dpu, debemos primero obtener los valores α y β para determinar el punto de donde proviene la particula. El cual obtenemos de la siguiente forma:

$$\alpha = \Delta t * u_{i,j} \tag{4.4}$$

$$\beta = \Delta t * v_{i,j} \tag{4.5}$$

Y calculamos los puntos de partida para cada punto en x y y de la siguiente forma:

$$dpX = X_i - \alpha \tag{4.6}$$

$$dpY = Y_j - \beta \tag{4.7}$$

Luego obtenemos la parte entera de la división entre dpX y Δx e igualmente con respecto a y para obtener xm y ym. Y para xM decimos que es $xm + 1$, igualmente con y. Usando el mismo concepto podemos calcular los puntos de partida para h, u y v.
Como se puede ver en 2.18, para calcular los puntos de partida necesitamos tener

previamente los valores de xm y ym, para obtener el valor del punto de partida debemos calcular Um al igual que uM, lo hacemos de la siguiente manera:

$$Um = \frac{((u_{xM,ym} * ((yM * \Delta y) - dpy)) + (u_{xM,yM} * (dpy - (ym * \Delta y))}{\Delta x} \tag{4.8}$$

$$UM = \frac{((u_{xm,ym} * ((yM * \Delta y) - dpy)) + (u_{xm,yM} * (dpy - (ym * \Delta y))}{\Delta x} \tag{4.9}$$

$$dpu_{i,j} = \frac{((UM * (dpx - (xm * \Delta x))) + (um * ((xm * \Delta x) - dpx)))}{\Delta x} \tag{4.10}$$

De igual manera calculamos dpv y dph respectivamente para el intervalo de tiempo para luego introducir lo valores y crear el sistema de ecuaciones. Una vez obtenido los puntos de partida solucionamos el sistema de ecuaciones para obtener todos las alturas, h, para luego reemplazarlas y hacer el render por cada iteración de tiempo.

```
void DeparturePoints(){

double dpX=0,dpY=0,Xi,Yj,integer,fractional;
double UM, Um, VM, Vm, Hm, HM, tempX=0, tempY=0;
double alfa=0, beta=0;
int i, j, aux=0, Xm=0, XM=0, Ym=0, YM=0;
//Calculamos puntos de Partida
for(i=1; i<n-1; i++){
        for(j=1; j<m-1; j++){

                Xi = deltaX*i;
                Yj = deltaY*j;
                //Calculamos los alfa y beta para determinar los puntos de
                    donde proviene la particula
                alfa = deltaT*u[i][j];
                beta = deltaT*v[i][j];
                dpX = Xi - alfa;         //Departure Point en X
                dpY = Yj - beta;         //Departure Point en Y

                //Calculamos los Xm y Ym
                //Extraemos la parte entera de la división
                fractional = modf(dpX / deltaX, &integer);
                Xm = integer;
                XM = Xm + 1;
                //Extraemos la parte entera de la división
                fractional = modf(dpY / deltaY, &integer);
                Ym = integer ;
                YM = Ym + 1;

                /*Calculamos H's, U's y V's para los puntos de
                    partida
                Puntos de Partida para U's y V's
```

```
                    Para calcular los dpU necesitamos tener UM y Um
                    */

                    UM = (( u[XM][Ym]*((YM*deltaY) - dpY) ) + ( u[XM
                        ][YM]*(dpY - (Ym*deltaY)) )) / deltaX;
                    Um = (( u[Xm][Ym]*((YM*deltaY) - dpY) ) + ( u[Xm
                        ][YM]*(dpY - (Ym*deltaY)) )) / deltaX;
                    dpu[i][j] = ( ( UM*(dpX - (Xm*deltaX)) ) + ( Um
                        *((XM*deltaX) - dpX) ) ) / deltaX;

                    //Ahora calculamos dpV, pero primero necesitamos VM y Vm
                    VM = (( v[XM][YM]*(dpX - (Xm*deltaX)) ) + ( v[Xm
                        ][YM]*((XM*deltaX) - dpX) ))/ deltaY;
                    Vm = (( v[XM][Ym]*(dpX - (Xm*deltaX)) ) + ( v[Xm
                        ][Ym]*((XM*deltaX) - dpX) ))/ deltaY;
                    dpv[i][j] = ( ( Vm*((YM*deltaY) - dpY) ) + ( VM*(
                        dpY - (Ym*deltaY)) ) ) / deltaY;

                    /*Puntos de Partida para H's
                    Para calcular los dpH necesitamos igualmente HM y
                         Hm;*/
                    HM = (( h[XM][Ym]*((YM*deltaY) - dpY) ) + ( h[XM
                        ][YM]*(dpY - (Ym*deltaY)) ))/ deltaY;
                    Hm = (( h[Xm][Ym]*((YM*deltaY) - dpY) ) + ( h[Xm
                        ][YM]*(dpY - (Ym*deltaY)) ))/ deltaY;
                    dph[i][j] = (( HM*( dpX - (Xm*deltaX) ) ) + ( Hm
                        *( (XM*deltaX) - dpX ) )) / deltaX;

                    }
            }
}
```

Código 4.8: Puntos de Partida para la simulación

En el código 4.8 se muestra como se calculan los puntos de partida para cada matriz.

4.4.4. Actualización del Sistema de Ecuaciones

El sistema de ecuaciones de debe actualizar con cada iteración, es decir para cada iteración de tiempo. Debido a que se esta usando la libreria de UCSparseLib actualizamos el sistema en el mismo formato ya comprimido de la matriz esparcida. De esta manera en el código 4.9 se puede observar como se copian los nuevos datos a la matriz esparcida para una nueva iteración. Además se puede observar también como se actualiza el vector de terminos independientes para cada ecuación del sistema.

```
//Modificar los valores en TDMatrix
int ii,jj,i,j,dim, cont;
```

```
TDSparseVec  row;
double B1=1, B2=1, deltaTcuad, Dij, dtXcuad, dtYcuad;
double Bi=0, Bj=0, deltaG, deltaXY;
dim = n*n;
deltaTcuad = deltaT * deltaT;
dtXcuad = deltaX * deltaX;
dtYcuad = deltaY * deltaY;
deltaXY = ( deltaX + deltaY ) / 2;
deltaG = ((deltaT/deltaXY)*(deltaT/deltaXY)*g);

for (ii= n; ii< dim-n; ii++){
        For_TDMatrix_Row( SM, ii, row, ACCESS_WRITE )
        {
                j = ii%n+1;
                i = (int)(ii/n);
                Dij = h[i][j]-b[i][j];
                Bi = ( b[i+1][j] - b[i-1][j] ) / 2;
                Bj = ( b[i][j+1] - b[i][j-1] ) / 2;
                for (jj= 1; jj< row.nz; jj++)
                {
                        if(jj == 2)
                        row.val[jj] = 1 + ( 4 * deltaG * Dij );

                        if(jj == 1)
                        row.val[jj] = -( deltaG * ( Bj + Dij ) );

                        if(jj == 3)
                        row.val[jj] = +( deltaG * ( Bj - Dij ) );

                        if(jj == 0)
                        row.val[jj] = +( deltaG * ( Bi - Dij ) );

                        if(jj == 4)
                        row.val[jj] = -( deltaG * ( Bi + Dij ) );
                }
        }
}
//CALCULO DEL VECTOR DE TERMINOS INDEPENDIENTES
cont=n+1;
for(i=1; i<n-1; i++){
        for(j=1; j<m-1; j++){
                Dij = h[i][j]-b[i][j];
                Bi = ( b[i+1][j] - b[i-1][j] ) / 2;
                Bj = ( b[i][j+1] - b[i][j-1] ) / 2;
                terminos[cont] = dph[i][j] + ((deltaT/(2*deltaXY)
                    )*((dpu[i][j]*Bi)+(dpv[i][j]*Bj)))
                - (((deltaT*Dij)/2*deltaXY) * (dpu[i+1][j] - dpu[
                    i-1][j] + dpv[i][j+1] - dpv[i][j-1]));
                cont++;
        }
        cont+=2;
}
```

Código 4.9: Actualización de los datos del sistema de ecuaciones y del vector de terminos independientes

4.4.5. Actualización en las matrices de la velocidad de las particulas

4.5. Interfaz y Render

4.5.1. Superficie

Existen distintos métodos para generar la superficie del cuerpo de agua, sin embargo, para los propositos de este trabajo solo uno de los métodos funcionaba correctamente sin presentar problemas. Mediante la creación de un polígono formado por triangulos podemos represar la superficie. Es indispensable para la creación de este método conocer bien la posición de cada punto el momento de generar el cuerpo de agua para que el mismo tenga consistencia. Una de las cosas más importante para la implementación de este método es el calculo de la normal de cada cara del polígono para cada intervalo de tiempo, de esta manera podremos renderizar brillos, especular, refracciones, entre otras. En este trabajo nos basamos en el cálculo de los polígonos usando la función de OpenGL GL_TRIANGLE_STRIP, que es un polígono formado a base de triangulos. En el código 4.10 se puede observar como se genera la superficie, así como también como es asignado la normal para cada triángulo.

```
for(j=1; j<m-2; j++){
        glBegin(GL_TRIANGLE_STRIP);

        A = PuntoP((i*n)+j);
        B = PuntoP((i*n)+(j+1));

   glVertex3f( Pts[(i*n)+j].x, Pts[(i*n)+j].y, Pts[(i*n)+j].z );
   glVertex3f( Pts[(i*n)+(j+1)].x, Pts[(i*n)+(j+1)].y, Pts[(i*n)
      +(j+1)].z );

        for(k=1; k<m-2; k++){

                C = PuntoP(((i+k)*n)+j);

                vecNormalized = Normal(A,B,C);
                glNormal3f(vecNormalized.x, vecNormalized.y,
                   vecNormalized.z);
                glVertex3f( Pts[((i+k)*n)+j].x, Pts[((i+k)*n)+j].
                   y, Pts[((i+k)*n)+j].z );
```

```
                A = B;
                B = C;
                C = PuntoP(((i+k)*n)+(j+1));

                vecNormalized = Normal(A,B,C);
                glNormal3f(vecNormalized.x, vecNormalized.y,
                    vecNormalized.z);
                glVertex3f( Pts[((i+k)*n)+(j+1)].x, Pts[((i+k)*n)
                    +(j+1)].y, Pts[((i+k)*n)+(j+1)].z );
        }
        glEnd();
 }
```

Código 4.10: Creación de la superficie de agua

4.5.2. Cálculo de la normal

Para obtener las distintas reflecciones, brillos y reflacciones de la superficie debemos conocer el valor de la normal de cada triangulo de la superficie para cada intervalo de tiempo. Para esto se necesita la información los 3 puntos que conforman el triangulo y así poder brindarle la información a OpenGL y el mismo pueda procesar la iluminación correctamente con los distintos angulos de cámara.

```
vector Normal(point A, point B, point C){

        vector V1,V2,V3;
        double normalized;
        V1.x = B.x - A.x;
        V1.y = B.y - A.x;
        V1.z = B.z - A.z;

        V2.x = C.x - A.x;
        V2.y = C.y - A.y;
        V2.z = C.z - A.z;

        V3.x = (A.y * B.z) - (A.z * B.y);
        V3.y = (A.z * B.x) - (A.x * B.z);
        V3.z = (A.x * B.y) - (A.y * B.x);

        //Normalizamos
        normalized = sqrt((V3.x * V3.x) + (V3.y * V3.y) + (V3.z *
            V3.z));
        V3.x /= normalized;
        V3.y /= normalized;
        V3.z /= normalized;

        return V3;
}
```

Código 4.11: Cálculo de la normal de una cara de la superficie

4.5.3. Visualización de la malla de la superficie

Para visualizar la malla de la supericie es necesario la posición de cada punto. Luego se recorren todos los puntos de la malla y se creán las líenas que representan la malla de la superficie.

```
glPointSize(1.0f);
glColor4f(1.0, 1.0, 1.0, 1.0);
for(i=1; i<n-2; i++)
for (j=1; j<m-2; j++){
        glBegin(GL_LINES);
                glVertex3f( Pts[(i*n)+j].x, Pts[(i*n)+j].y, Pts[(
                   i*n)+j].z );
                glVertex3f( Pts[(i*n)+(j+1)].x, Pts[(i*n)+(j+1)].
                   y, Pts[(i*n)+(j+1)].z );
        glEnd();
        glBegin(GL_LINES);
                glVertex3f( Pts[(i*n)+(j+1)].x, Pts[(i*n)+(j+1)].
                   y, Pts[(i*n)+(j+1)].z );
                glVertex3f( Pts[((i+1)*n)+(j+1)].x, Pts[((i+1)*n)
                   +(j+1)].y, Pts[((i+1)*n)+(j+1)].z );
        glEnd();
        glBegin(GL_LINES);
                glVertex3f( Pts[((i+1)*n)+(j+1)].x, Pts[((i+1)*n)
                   +(j+1)].y, Pts[((i+1)*n)+(j+1)].z );
                glVertex3f( Pts[((i+1)*n)+j].x, Pts[((i+1)*n)+j].
                   y, Pts[((i+1)*n)+j].z );
        glEnd();
        glBegin(GL_LINES);
                glVertex3f( Pts[(i*n)+j].x, Pts[(i*n)+j].y, Pts[(
                   i*n)+j].z );
                glVertex3f( Pts[((i+1)*n)+j].x, Pts[((i+1)*n)+j].
                   y, Pts[((i+1)*n)+j].z );
        glEnd();
}
```

Código 4.12: Creación de la malla de la superficie

En el código 4.12 se muestra como se construye la malla de la superficie a travéz de los puntos almancenados.

4.5.4. Iluminación y materiales

Para la iluminar de la escena se utilizó una iluminación de 2 puntos, que consta de dos luces en puntos estrategicos en la escena. Cada luz contiene cierta cantidad de emisión y color, esto se hace para una mejor visualización de las sombras y brillos que se puedan generar en la escena. Además de estas 2 luces se agrego una luz ambiental para darle tonalidad a la escena completa.
Para la creación de materiales o shaders, se utilizaron las propiedades de los ma-

teriales de OpenGL como lo son, luz ambiental, especular, difusión de la luz y brillo. Cada parametro fue ajustados a las necesidades de la iluminación de la escena para una mejor representación de la simulación.
En el siguiente código 4.13, se pueden observar los parametros utilizados para la configuración del material de agua.

```
glShadeModel(GL_SMOOTH);
glEnable(GL_COLOR_MATERIAL);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glColor4f(0.196078, 0.6, 0.8, 0.8);

//propiedades del material
GLfloat lmodel_ambient[] = {0.01, 0.01, 0.1, 1.0};
GLfloat mat_ambient[] = {0.5, 0.5, 0.8, 1.0};
GLfloat mat_diffuse[] = { 0.1, 0.5, 0.8, 1.0 };
GLfloat mat_specular[] = {0.7, 0.9, 0.9, 0.0};
GLfloat mat_shininess[] = {5.0};
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient);
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, mat_shininess);
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, lmodel_ambient);
```

Código 4.13: Configuración del material del cuerpo de agua

Capítulo 5

Resultados y Conclusiones

5.1. Parámetros de Ejecución

Todas las pruebas se llevaron a cabo en un computador de 64 bits corriendo Linux Mint 12 Lisa usando Gnome 2.32.1, núcleo Linux 3.0.0-21-generic, Ext4 como sistema de ficheros y para la versión de OpenGL utilizada es 2.1.2 NVIDIA 304.88. Los códigos fueron compilados usando el compilador gcc versión 4.6.1 (Ubuntu/Linaro 4.6.1-9ubuntu3) y la versión de shading del compilador de OpenGL es 1.20. Las librerias usadas para la compilación del código son gl lGL lglut lGLU GLUT.

El computador posee las siguientes especificaciones:

- Procesador Pentium(R) Dual-Core CPU E5300 @2.60GHz 2.60GHz. L2 Caché de 2048Kb por núcleo.
- 4GB de RAM DDR2-800 (PC2-6100), latencia 5-5-5-18.
- Disco duro Western Digital WD Blue, SATA 3.0Gb/s, 500GB, 7200 RPM, caché 16MB, tiempo promedio de busqueda de 8.9ms.

La tarjeta de video posee las siguientes especificaciones:

- Modelo Nvidia GeForce 9600 GT
- 64 Núcleos de procesamiento
- NVIDIA driver versión 340.52
- Reloj del procesador 1625MHz
- Config. de memoria estándar 512MB
- Interfaz de memoria 256-bit

- Ancho de banda de memoria 57.6GB/s
- Tasa de relleno (miles de mollones de píxeles): 20.8
- Vértices por segundo 338 millones

5.2. Estados iniciales de prueba

Para la realización de las pruebas del algoritmo se partieron de 4 principales estados iniciales que demostrarán el funcionamiento de dicho algoritmo. Estas fueron selecionadas de un conjunto de estados iniciales concretos y precisos para poder observar los resultados esperados. Entre los estados iniciales de prueba se encuentran los siguientes:

5.2.1. Ondas en sentido del eje X

Consta en representar una serie de ondas (olas) repetidas en sentido al eje x, con una dirección de fuerza en el mismo sentido del eje x.

```
int i,j;
double radio;
for(i=1; i<n-1; i++)
{
        for(j=1; j<m-1; j++){
                radio =  i*deltaX ;
                h[i][j] = 0.7 * (sin( radio * (1*Pi) ));
                u[i][j] = ((i*deltaX)/radio)/2;
                v[i][j] = ((j*deltaY)/radio)/2;
                //Puntos para Graficar.
                Pts[(i*n)+j].y = h[i][j];
        }
}
```

Código 5.1: Puntos de partida para ondas en sentido del eje X

En el código 5.1 se puede mostrar como se genera el estado inicial para que las ondas sean replicadas a lo largo del eje X. En la figura **??** se puede mostrar el estado inicial explicado anteriormente.

5.2.2. Ondas en sentido del eje Y

Consta en representar una serie de ondas (olas) repetidas en sentido al eje y, con una dirección de fuerza en el mismo sentido del eje y.

```
int i,j;
double radio;
for(i=1; i<n-1; i++)
{
        for(j=1; j<m-1; j++){
                radio =  j*deltaX ;
                h[i][j] = 0.7 * ( sin( radio * (1*Pi) ));
                u[i][j] = ((i*deltaX)/radio)/2;
                v[i][j] = ((j*deltaY)/radio)/2;
                Pts[(i*n)+j].y = h[i][j];
        }
}
```

Código 5.2: Puntos de partida para ondas en sentido del eje Y

En el código 5.2 se puede mostrar como se genera el estado inicial para que las ondas sean replicadas a lo largo del eje Y.

5.2.3. Ondas partiendo de una esquina de la superficie

Este estado inicial representa las mismas ondas en forma circular y partiendo desde una de las esquinas de la "piscina.º estanque donde se encuentra el cuerpo de agua. Para ello usamos una serie de parámetros que nos permiten ajustar la altura de las olas, la cantidad de olas. Además de esto, la velocidad inicial utilizada es igual a una fuerza que movera las ondas hacia la esquina opuesta a donde se inician.

```
int i,j;
double radio;
for(i=1; i<n-1; i++)
{
        for(j=1; j<m-1; j++){
                radio = sqrt((( i*deltaX )*( i*deltaX )) + (( j*
                   deltaY )*( j*deltaY ))) ;
                h[i][j] = 0.6*(sin( radio * (1*Pi) ));
                Pts[(i*n)+j].y = h[i][j];
                u[i][j] = ((i*deltaX)/radio)/4;
                v[i][j] = ((j*deltaY)/radio)/4;
                if(radio > 10){
                        h[i][j] = 0.0;
                }
        }
}
```

Código 5.3: Puntos de partida para ondas desde una esquina

En el código 5.3 se puede mostrar como se genera el estado inicial para que las ondas sean replicadas a lo largo del cuerpo de agua partiendo desde una de las

esquinas del tanque de la piscina.

5.2.4. Ondas partiendo desde el centro de la superficie

Para poder ver la propagación de onda hacia las esquinas es necesario hacer esta prueba. Consta de generar un estado inicial aplicando una fuerza en el centro del cuerpo de agua y replicando ondas en forma circulas hacia los bordes de la piscina. Para ello usamos una serie de parámetros que nos permiten ajustar la altura de las olas, la cantidad de olas. Además de esto, la velocidad inicial utilizada es igual a una fuerza que movera las ondas hacia afuera.

```
int i,j;
double radio;
for(i=1; i<n-1; i++){
        for(j=1; j<m-1; j++){

                radio = sqrt((( (i - (n/2))*deltaX )*( (i - (n/2)
                   )*deltaX )) + (( (j-(m/2))*deltaY )*( (j-(m/2)
                   )*deltaY )));
                h[i][j] = ( 0.9 * sin( radio * (2*Pi) ) );
                Pts[(i*n)+j].y = h[i][j];
                if(radio == 0){
                        u[i][j] = 0;
                        v[i][j] = 0;
                }else{
                        u[i][j] = (((i-(n/2))*deltaX)/radio)/2;
                        v[i][j] = (((j-(m/2))*deltaY)/radio)/2;
                }
                if(radio > 1.5){
                        //u[i][j] = 0;
                        //v[i][j] = 0;
                        h[i][j] = 0.0;
                }
        }
}
```

Código 5.4: Puntos de partida para ondas desde una esquina

En el código 5.4 se puede mostrar como se genera el estado inicial para que las ondas sean replicadas a lo largo del cuerpo de agua partiendo desde el centro de la piscina.

5.3. Resultados

5.3.1. Tiempos de Ejecución

En cuanto a los tiempos de ejecución varian dependiendo de la cantidad de puntos usados en el sistema o malla. Se generaron varias pruebas empleando distintos tamaños para la malla de la superficie del agua. A pesar de que no se puedo evidenciar muchos cambios se logró registrar los tiempos como se pueden ver en la siguiente tabla. Los tiempos registrados son todamos desde el momento que inicia la simulación hasta el momento en que se estabiliza el cuerpo de agua.

Todas las pruebas de tiempo de ejecución se realizaron con los siguientes parametros:

- $\Delta x = 0.1$ y $\Delta y = 0.1$
- $g = 0.0006006456$
- $\Delta t = 0.033333333$ (30 fps)
- $\Pi = 3.141592654$

5.3.2. Tiempos de ejecución para malla de 50 puntos

$(\Delta t)fps$	Tiempos	Estabilidad
30 fps	45 segs	100 %
24 fps	30 segs	99 %
15 fps	25 segs	98 %
5 fps	5 segs	85 %

Tabla de tiempos de ejecución para onda en sentido X y Y

$(\Delta t)fps$	Tiempo	Estabilidad
30 fps	50 segs	100 %
24 fps	35 segs	99 %
15 fps	25 segs	99 %
5 fps	10 segs	95 %

Tabla de tiempos de ejecución para onda desde una esquina

$(\Delta t)fps$	Tiempos	Estabilidad
30 fps	25 segs	100 %
24 fps	15 segs	99 %
15 fps	10 segs	98 %
5 fps	3 segs	65 %

Tabla de tiempos de ejecución para onda desde el centro

5.3.3. Tiempos de ejecución para malla de 100 puntos

$(\Delta t)fps$	Tiempos	Estabilidad
30 fps	80 segs	100 %
24 fps	55 segs	99 %
15 fps	15 segs	98 %
5 fps	10 segs	80 %

Tabla de tiempos de ejecución para onda en sentido X y Y

$(\Delta t)fps$	Tiempos	Estabilidad
30 fps	53 segs	100 %
24 fps	37 segs	99 %
15 fps	28 segs	98 %
5 fps	15 segs	95 %

Tabla de tiempos de ejecución para onda desde una esquina

$(\Delta t)fps$	Tiempos	Estabilidad
30 fps	20 segs	100 %
24 fps	18 segs	99 %
15 fps	10 segs	98 %
5 fps	4 segs	45 %

Tabla de tiempos de ejecución para onda desde el centro

5.3.4. Tiempos de ejecución para malla de 125 puntos

$(\Delta t)fps$	Tiempos	Estabilidad
30 fps	140 segs	100 %
24 fps	90 segs	99 %
15 fps	32 segs	98 %
5 fps	21 segs	75 %

Tabla de tiempos de ejecución para onda en sentido X y Y

$(\Delta t)fps$	Tiempos	Estabilidad
30 fps	62 segs	100 %
24 fps	48 segs	99 %
15 fps	36 segs	98 %
5 fps	24 segs	95 %

Tabla de tiempos de ejecución para onda desde una esquina

$(\Delta t)fps$	Tiempos	Estabilidad
30 fps	36 segs	100 %
24 fps	28 segs	99 %
15 fps	15 segs	98 %
5 fps	5 segs	40 %

Tabla de tiempos de ejecución para onda desde el centro

5.3.5. Almacenamiento en memoria

En la siguiente tabla se puede observar los tiempos para reservar el espacio de memoria y generar el sistema inicial de ecuaciones, además se podrá comparar las dimensiones de las matrices para cada tamaño de la malla.

Cantidad de Puntos	Tamaño del SE	Tiempo
50	2500 incognitas	0.5 segs
100	10000 incognitas	1.5 segs
125	15625 incognitas	2.5 segs
135	18225 incognitas	3.5 segs
140	19600 incognitas	5 segs

Tabla de tiempos de ejecución para onda desde el centro

5.3.6. Fuerza de atracción entre particulas

Para calcular la simulación se utiliza un parámetro llamado gravedad g, el cual es el valor de atracción o repulsión entre los puntos de la malla. Se probaron distintos valores para g con el fin de obtener un valor que genere la mayor estabilidad y precisión. La siguiente tabla muestra las diferencias de los valores.

g	Estabilidad	Presición
0.0006006456	100 %	100 %
0.00098	99 %	98 %
0.00983	98 %	95 %
0.098	0 %	1 %

Tabla de tiempos de ejecución para onda desde el centro

Un dato importante de acotar es la resolución que no es más que, que tan densa puede llegar a ser la malla o superficie para tener un mejor acabado, esto se puede interpretar como: que a menor Δx y Δy mejor presición, pero requiere mayor cantidad de puntos para tener una buena visualización de la simulación, por lo tanto, genera mayor cantidad de incognitas en el sistema de ecuaciones y el computador necesitará mayor capacidad de memoria y procesamiento para funcionar fluidamente.

5.3.7. Gráficas de los resultados

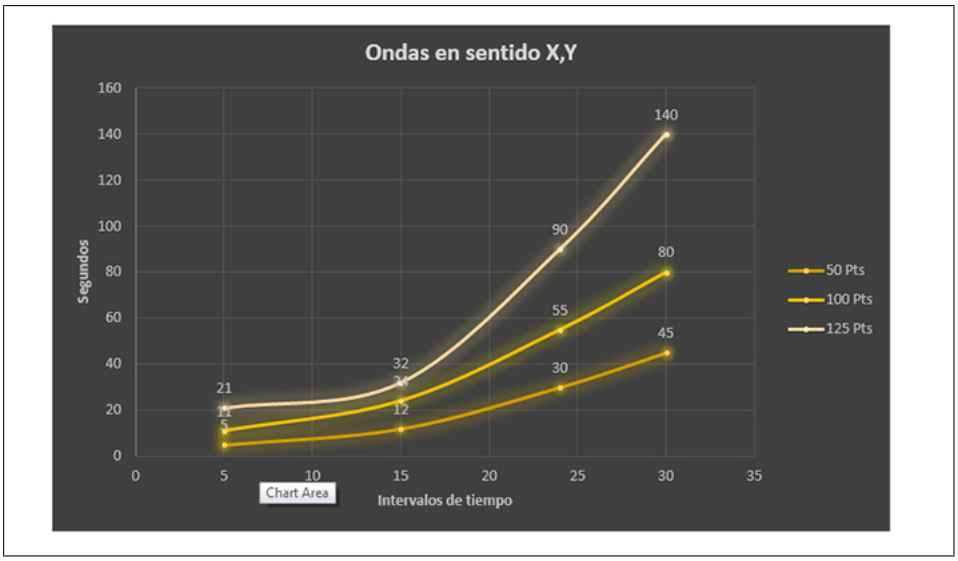


Figura 5.1: Gráfica de los tiempos de ejecución de las prueba de las ondas generadas en sentido de las coordenadas x y y

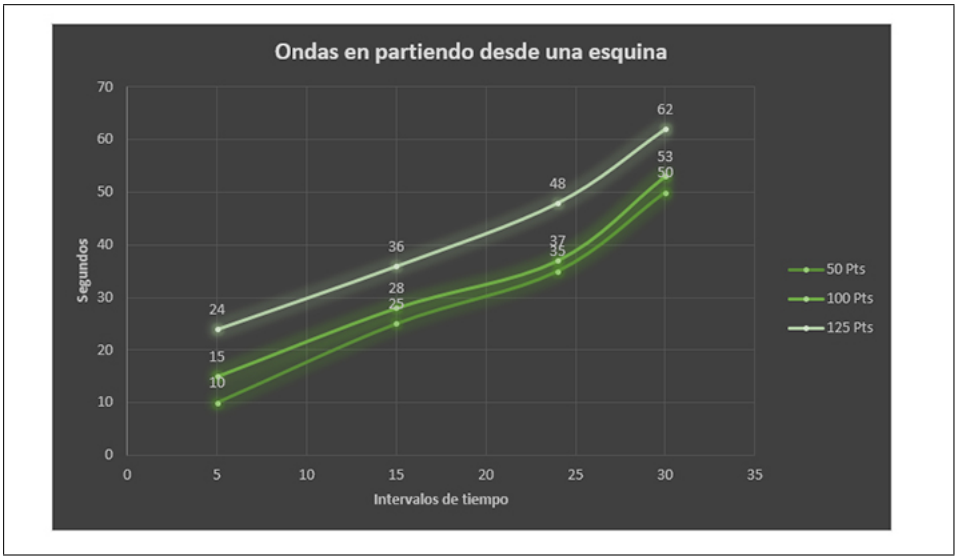


Figura 5.2: Gráfica de los tiempos de ejecución de las prueba de las ondas generadas desde una esquina

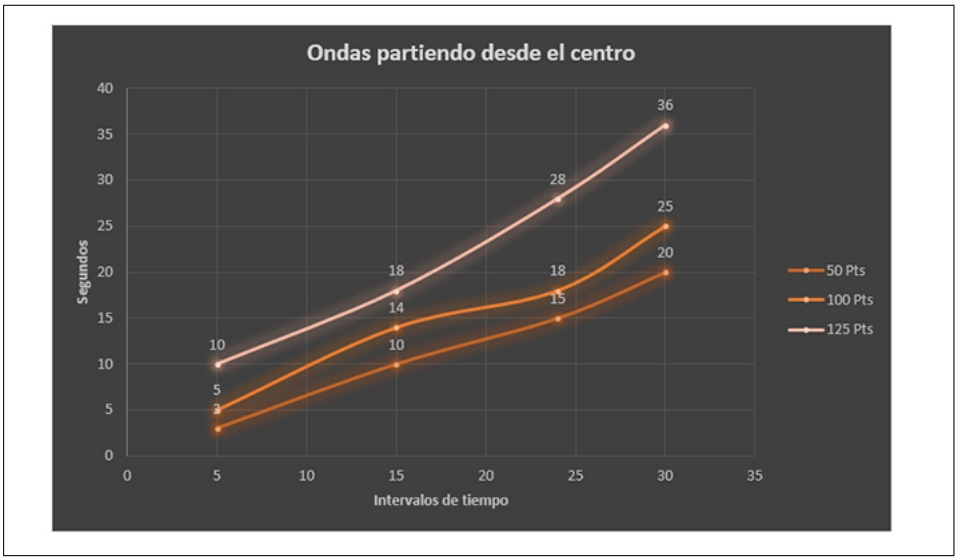


Figura 5.3: Gráfica de los tiempos de ejecución de las prueba de las ondas generadas desde el centro

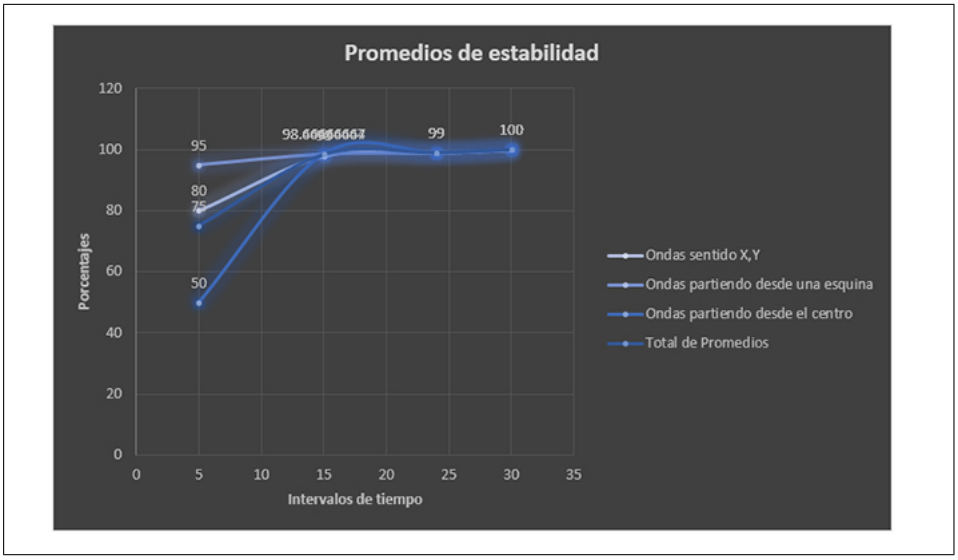


Figura 5.4: Gráfica de la estabilidad en las distintas pruebas para cada Δt probado

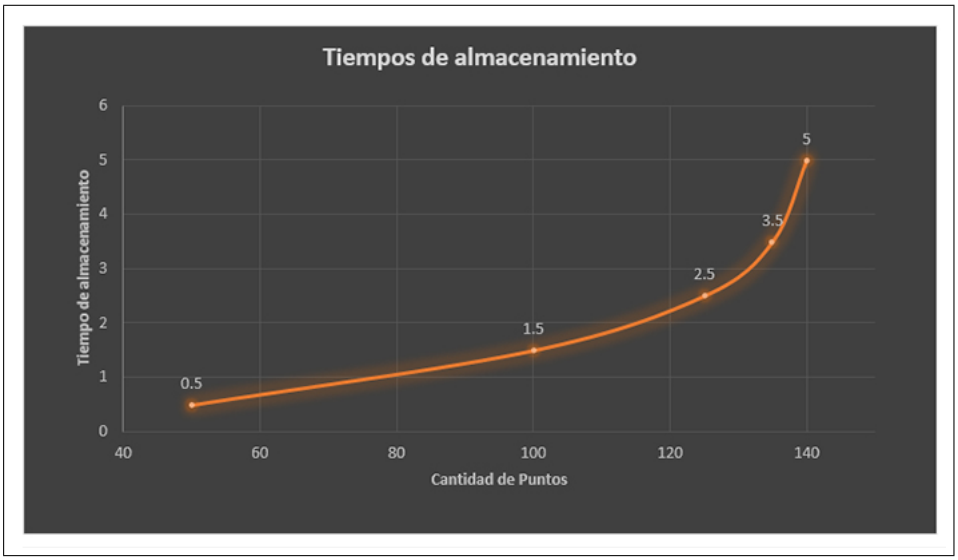
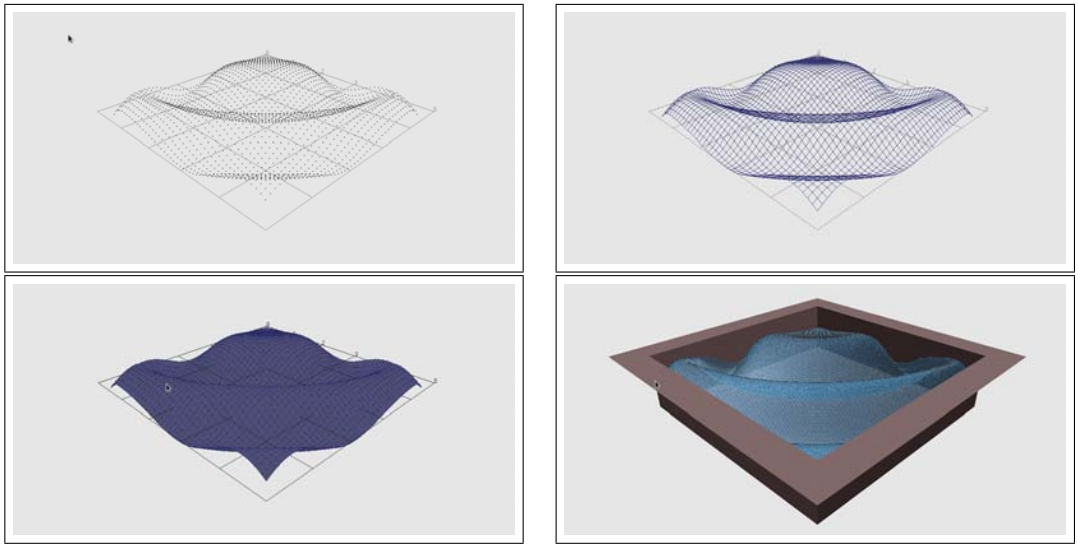


Figura 5.5: Gráfica de tiempos de reservación de memoria

5.3.8. Evolución y apariencia de las ondas

El resultado obtenido para obtener el movimiento de las ondas es bastante acertado, para el siguiente trabajo se realizaron distintas representaciones de la malla para poder ver la simulación de las ondas. Entre los distintos tipos de muestra que generamos se encuentra el de puntos y el mallado de la superficie del cuerpo de agua, como se puede observar el las siguientes imagenes.



Evolución de la elaboración del render final

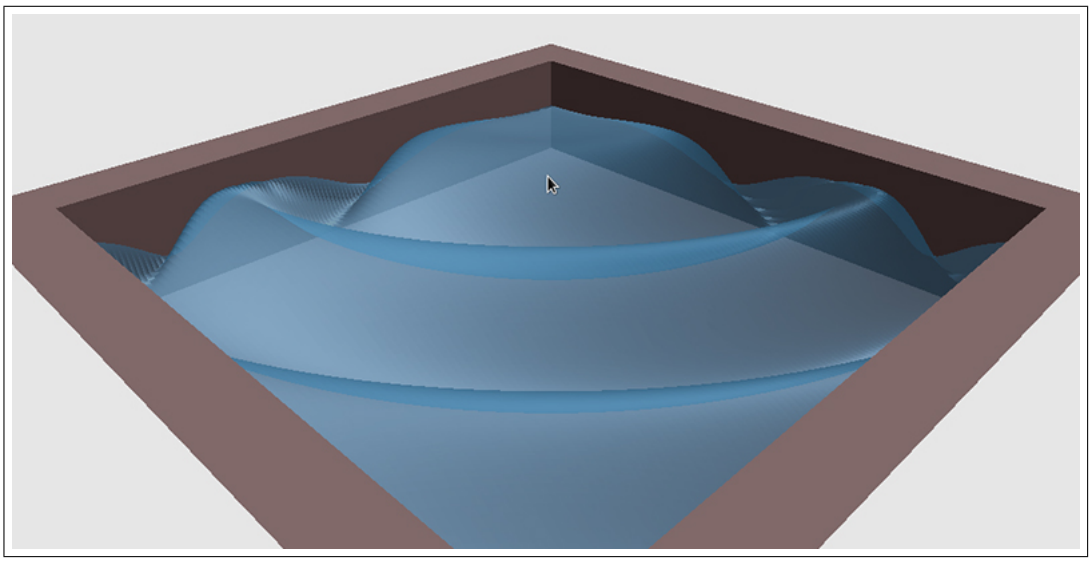
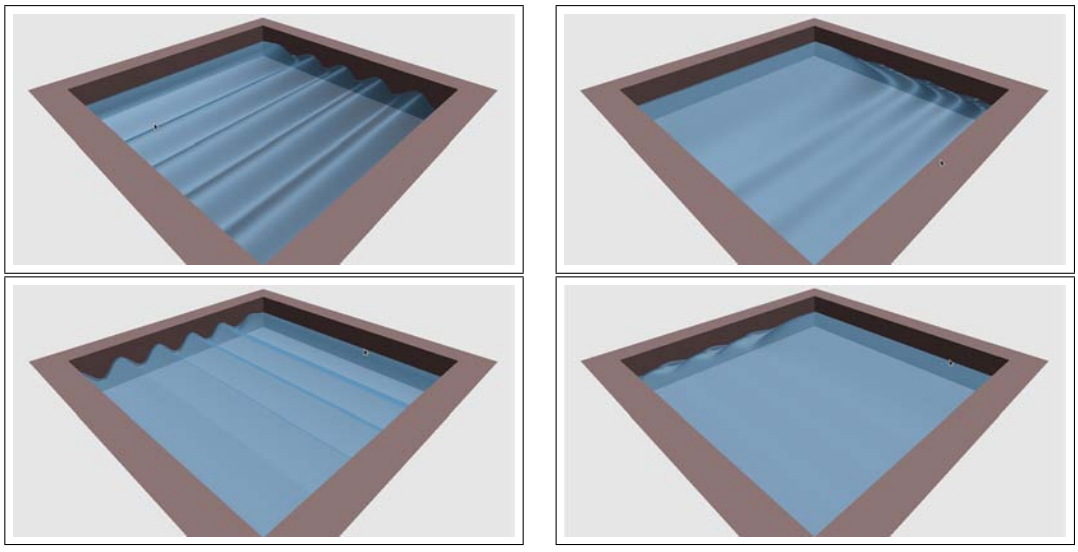


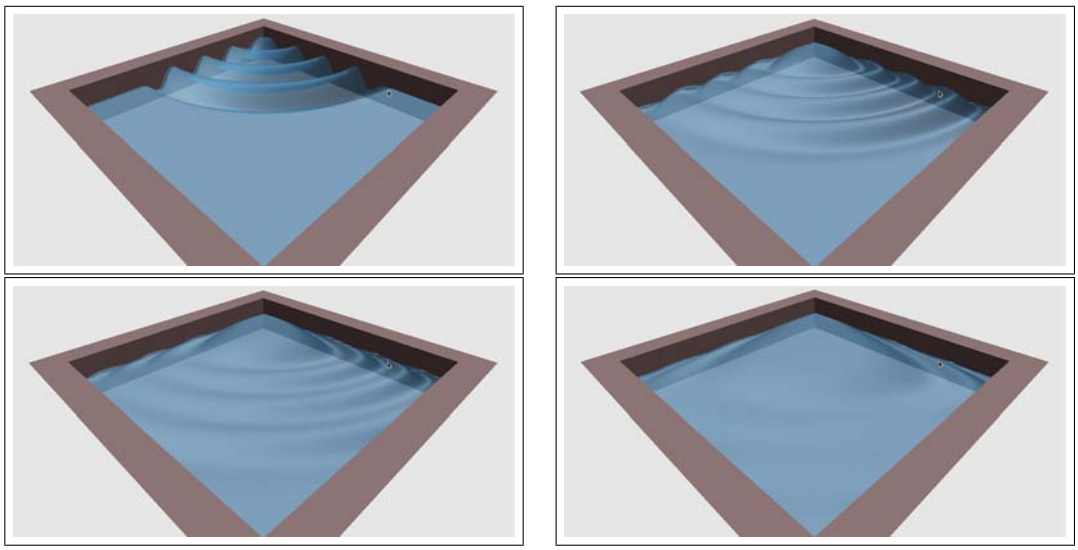
Figura 5.6: Último paso para obetener el render final

5.3.9. Render

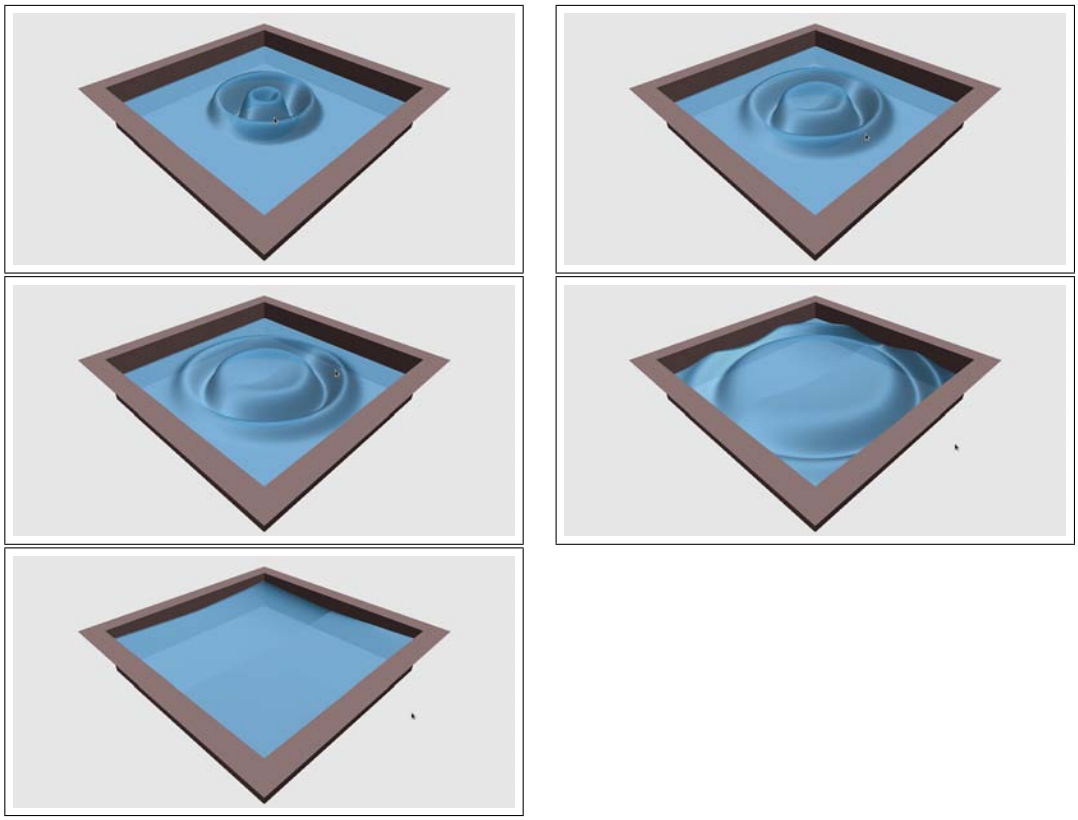
Para generar un acabado realista se usaron ciertas técnicas de iluminación y configuración de shaders. Se utilizó las librerias de OpenGL para crear el acabado final. Se utilizaron luces de 3 puntos. La configuración de los shaders y luz ambiental fueron tomadas en referencia a los valores reales del las superficies de agua. En las siguientes imagenes se puede observar los renders para las distintas pruebas.



Imágenes de la simulación render Final



Simulación con ondas generadas desde una esquina del estanque



Simulación con ondas generadas desde el centro del estanque

5.4. Conclusiones

Se demostró, que por medio de el método de integración implícita en 2 dimensiones se puede representar un cuerpo de agua y obtener los cálculos de forma inmediata para renderizar y mostrar el movimiento del agua, con tan solo tener en cuenta los valores necesarios y ajustar de forma correcta los parametros podemos obtener una simulación estable, con gran precisión y eficiencia. Además de esto podemos decir que no hubiera sido posible representar estas simulaciónes sin la utilización de un método de almacenamiento de datos a gran escala, debido a que a mayor cantidad de puntos mejor aspecto se logra obtener en la simulación, el método de almacenamiento de datos de CRS nos permitió sacarle el rendimiento máximo a nuestra capacidad de memoria, de modo que solo se almacenaron los datos relevantes para la simulación.

Se obtuvieron resultados positivos aplicando fuerzas básicas como ondas en un mismo sentido, como generar una secuencia de ondas provenientes de una de las esquenas del estanque de agua, y adicional a este, replicas de ondas generadas desde el centro de la superficie de agua. Dichos resultados evidencian la importancia que tiene la forma de calcular o simular ondas en un cuerpo de agua, debido a que existen muchos métodos y algoritmos. El de este trabajo pudó demostrar que obtiene resultados inmediatos y que al mismo tiempo esto permite ver la simulación practicamente en tiempo real.

Otro resultado positivo que se obtuvo fue el uso de la librería de OpenGL, que nos permitió representar la animación con un acabado bastante parecido a lo que se puede observar en el mundo real. Además del uso de DobleBuffer, algunas técnicas de render e iluminación nos asegura el renderizado de forma rápida y acertada del movimiento de agua, de modo que se pueda observar la animación mientras se cálcula cada intervalo de tiempo, sin ver ningun parpadeo o extraños destellos de brillo intermitentes.

5.5. Trabajos futuros

Se recomienda integrar una función que permita el rebote de las ondas al chocar con los bordes o paredes que contenga el cuerpo de agua, de esta forma poder visualizar el comportamiento completo del cuerpo de agua hasta lograr su estabilidad. Adicionalmente a esto se poder integrar cuerpos adicionales u objetos en el medio del tanque de agua para así observar el rebote de las ondas en dichos objetos.

Por otro lado, se puede integrar a futuro el uso de partículas en la simulación, tomando en cuenta que el cuerpo de agua se genera correctamente, de este se

pudiesen generar pequeñas partículas en las ondas y generar salpicaduras dependiendo de la fuerza que contenga las particulas vecinas en ese momento, de forma que se calculen las nuevas particulas cuadro a cuadro una vez se tenga el estado completo del cuerpo de agua en ese instante de tiempo.

Se puede también integrar un formato de creación de estados iniciales, que el mismo dependa del lugar de impacto. Ya que se a demostrado que la propagación de las ondas funciona con el modelo planteado. Por otra parte se puede implementar una función que permita guardar cada frame de la simulación como imagen de modo que se simule una sola vez y luego se pueda con otro programa cargar la secuencia de las imagenes y de este modo generar de nuevo la animación ya calculada y renderizada.

Por otro lado se puede implementar movimientos de cámara mas ajustados a al modelo que permitan observar el comportamiento desde cualquier ángulo o posición de cámara. Además de esto incorporar refracciones y causticas para terminar de darle un aspecto realista a la simulación siempre y cuando se cuente con una tarjeta gráfica que permita el cálculo rápidamente.

Bibliografía

[1] Preliminary result from a partial lrtap model used on an existing meteorological forecast model. *Atmos-Ocean 32*, pages 267–303, 1985.

[2] Modeling waves and surf. In *Computer Graphics*, pages 65–74, 1986.

[3] A simple model of ocean waves. In *Computer Graphics*, pages 75–84, 1986.

[4] Numerical method for wave equations for geophysical fluid mechanics. Springer, Berlin Heidelberg New York, 1998.

[5] N. Foster and D. Metaxas. Realistic animation of liquids. In *Graphical Models and Image Processing*, pages 471–483, 1996.

[6] S. J., editor. *Stable fluids*. SIGGRAPH '99 Proceedings, Computer Graphics, 1999.

[7] C. JX and L. N. da V. Toward interactive-rate simulation of fluids with moving obstacles using navier-stokes equations. In *Graph Model Im Proc*, pages 107–116, 1995.

[8] C. JX, L. N. da V, H. CE, and M. JM. Real-time fluid simulation in a dynamic virtual environment. In *IEEE Comput Graph*, pages 52–61, 1997.

[9] M. S. Longuet-Higgins and E. D. Cokelet. The deformation of steep surface waves on water i. a numerical method of computation. *Proceedings of the Royal Society of London*, pages 1–26, 1975.

[10] M. S. Longuet-Higgins and E. D. Cokelet. The deformation of steep surface waves on water ii. growth of normal-mode instabilities. *Proceedings of the Royal Society of London*, pages 1–28, 1978.

[11] K. M. and M. G. Rapid, stable fluid dynamics for computer graphics. In *Comput Graph*, pages 49–55, 1990.

[12] K. M., F. R.P., and L. X-D. A boundary condition capturing method for multiphase incompressible flow. In *UCLA Comput Appl Math*, pages 99–27, 2000.

[13] K. R. A simple model of ship wakes. Master's thesis, University Brit Columbia, 1994.