Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

COMPUTACIÓN PARALELA EN UNIDADES DE PROCESAMIENTO GRÁFICO (GPU)

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

JOON YOUNG KIM

PROFESORA GUÍA:
SRA. MARIA CECILIA RIVARA ZÚÑIGA

PROFESOR CO-GUÍA:
LUIS EMILIO ANTONIO MATEU

PROFESOR INTEGRANTE:
MAURICIO EDUARDO PALMA

SANTIAGO DE CHILE

2007

Resumen

El objetivo de esta memoria es el estudio y desarrollo de aplicaciones de computación general en tarjetas gráficas. Los avances tecnológicos han permitido que hardware especializado para la visualización de imágenes sea lo suficientemente poderoso como para implementar en sus procesadores programas que son habitualmente asociados a las CPU. Esta memoria explora y evalúa el uso de estos recursos para aplicaciones numéricas y de generación de mallas.

Para este objetivo se ha desarrollado una aplicación que simula la dinámica de fluidos y se exploró la posibilidad de aplicar algoritmos de refinado de mallas. Este tipo de algoritmos son intensivos en cómputo, ya que necesitan solucionar ecuaciones diferenciales usando métodos numéricos. Aplicando los conceptos que se requieren para programar este tipo de algoritmos a una GPU se busca optimizar su rendimiento y lograr una funcionalidad completa.

A través de la memoria se explican los conceptos matemáticos detrás de la mecánica de fluidos, y se describe la forma en la que se pueden descomponer para su posterior implementación en un procesador gráfico, que es altamente paralelo, y tiene diferencias sustanciales con la arquitectura de un procesador general. No se pudo aplicar un algoritmo en la GPU de refinamiento de mallas debido a limitantes físicas de su arquitectura, pero el estudio es útil para futuras investigaciones.

En conclusión, el programa creado muestra que es posible la adaptación de tales algoritmos, en hardware que a pesar de no estar diseñado para ellos entrega los mismos resultados que si fuesen programados de forma habitual. Esto además libera recursos que pueden ser utilizados para otros fines, o el uso de ambos procesadores, el CPU y la GPU, para la creación de programas que se ejecuten de forma más rápida y eficiente debido a la mayor cantidad de recursos disponibles.

Agradecimientos

A mi familia, por su apoyo incondicional

A la profesora M. Cecilia Rivara, por su ayuda durante mi carrera

Memoria financiada parcialmente por proyecto Fondecyt 1040713

I. Introducción

Desde los inicios de la Computación, todas las tareas computacionales eran manejadas a través de una central de procesamiento, llamada CPU, Unidad Central de Proceso (Central Processing Unit). A medida que se avanza en la tecnología, se han creado unidades de procesamiento destinadas a distintas tareas tales como controlar el flujo de datos, gestionar la memoria disponible y procesar tareas anexas a los datos principales, entre otros.

Donde más se ha avanzado en los últimos años es en el desarrollo de poderosas GPU, Unidad de Proceso Gráfico (Graphic Processing Unit). En las tarjetas gráficas actuales, las GPU son usadas para tareas tan complejas como procesar algoritmos de z-buffer, culling, rendering, que entre otros, son esenciales para mostrar escenas 3D en la pantalla.

Debido a que generalmente, las CPU eran más rápidas y menos costosas que las GPU, la mayoría de las tareas comunes en un proceso se hacen en la primera. Sin embargo, los últimos avances han demostrado que las GPU son tan o más poderosas que las CPU en cuanto a poder de procesamiento, y en ese punto se centra este tema, que es poder usar los recursos potenciales y las ventajas que esto acarrearía.

Un punto importante a considerar es la diferencia esencial de arquitectura entre un CPU y un GPU. El primero está optimizado para tareas secuenciales, tiene un registro completo, acepta todos los tipos estándar de datos, y tiene un caché grande de memoria disponible. El segundo está optimizado para tareas en

paralelo, debido a la naturaleza de los píxeles en la pantalla. Tiene menos memoria pero puede hacer tareas pequeñas con gran eficiencia.

Por ende, se debe estudiar qué procesos o tareas son más eficientes en paralelo que en forma secuencial, para aprovechar mejor las fortalezas de una GPU. Para esto, se debe analizar qué algoritmos se podrían adaptar para su uso en paralelo, o estudiar versiones existentes de éstos. Además, se debe tomar en cuenta que las GPU son mejor usadas con computaciones pequeñas pero en gran cantidad, a diferencia de los CPU que aprovechan su mayor memoria y flujo de datos para computaciones grandes.

En esta memoria, para el estudio del uso de la GPU en procesos generales, se estudió la implementación de algoritmos de refinamiento de triangulaciones y la simulación de fluido dinámicos. Para los algoritmos de refinamiento de triangulaciones no se obtuvo una solución satisfactoria, debido a limitaciones del hardware actual en las GPU. La simulación de fluidos dinámicos, en cambio, tiene propiedades que teóricamente pueden ser mejor aprovechadas por la arquitectura de las GPU.

I.1 Motivación

Debido a que las tarjetas gráficas de última generación han sido mejoradas para manejar complejas escenas 3D, cuando no se están creando tales escenas no se usa todo el potencial de cómputo de la GPU. El objetivo de esta memoria es poder aprovechar el poder de procesamiento de éste en simulaciones matemáticas, o la resolución de algoritmos que serían menos eficientes en la CPU.

Existen diversas áreas de la Computación en la que la naturaleza paralela de las GPU sería de provecho. Una de éstas es la edición de imágenes, donde las GPU pueden usar su programación de píxeles para la creación de filtros y efectos. Otra es la solución de ecuaciones lineales, que dada la eficiencia del álgebra de matrices de las GPU ayudarían a su resolución rápida. Otro uso más frecuente actualmente es el uso de las GPU para crear efectos en las escenas 3D en tiempo real, tales como niebla, piel, agua, y nubes, mediante transformaciones y cálculos en los píxeles a medida que éstos son procesados en la escena.

En general, se puede usar el poder de las GPU en tareas tanto gráficas como generales. El tema principal de esta memoria es poder usar tal capacidad en tareas que usualmente se asocian a las CPU, pero que adaptándolas a la arquitectura de una GPU se podrían hacer de igual o mejor manera, o en el peor caso, liberar recursos de la CPU para otras tareas.

En esta memoria se ha decidido por implementar algoritmos de simulación de fluidos dinámicos y de refinamiento de mallas geométricas, donde se aplicarán algunos algoritmos adaptados a la naturaleza de las GPU. Es de notar que este tipo de algoritmos son intensivos en cómputo, pero se realizan

sobre muchos puntos, lo que podría ser aprovechado por la arquitectura de la GPU, orientada a píxeles. Es interesante ya que no existe mucho trabajo previo acerca de realizar este tipo de algoritmos en otro medio aparte de una CPU. Las GPU están optimizadas para trabajar en celdas, lo que ayuda al cálculo de propiedades del fluido en cada punto del espacio. Esta memoria estudiará el beneficio que traería convertir estos algoritmos para que funcionen de forma paralela en la GPU.

En cuanto al refinamiento de triangulaciones, no se obtuvo un resultado esperado, que era obtener un algoritmo que se ejecutara en la GPU. Sin embargo, la exploración de este tema nos arrojó conclusiones que serían útiles para futuras investigaciones, lo que es un resultado positivo en sí.

I.2 Objetivos

El objetivo general de esta memoria es el estudio e implementación de algoritmos computacionales en una GPU, para obtener un mejor rendimiento o uso de recursos. Su análisis indicará las ventajas prácticas o desventajas de utilizarlos. Para lograr tal objetivo la memoria se centra en la implementación de algoritmos utilizados para la visualización y modelación de fluidos dinámicos y el refinamiento de mallas geométricas.

Específicos

Esta memoria considera los siguientes puntos específicos de estudio:

- Desarrollar un algoritmo de refinamiento de triangulaciones en la GPU.
- Desarrollar una modelación de dinámica de fluidos en la GPU.
- Estudiar un lenguaje de programación para las GPU, en este caso CG.
- Adaptar los algoritmos generales asociados a dinámica de fluidos al paradigma de programación en GPU.
- Generar un programa visual que demuestre la correcta aplicación de los algoritmos de dinámica de fluidos.
- Analizar beneficios de ocupar una GPU para procesamiento general.

Se obtuvo resultados satisfactorios con respecto a la simulación de fluidos dinámicos, pero no se pudo concretar un algoritmo de refinamiento de triangulaciones. Esto se debe a limitaciones en la arquitectura de la GPU, que no tiene incluida la opción de modificar la geometría de objetos en su memoria. Esto se explicará en detalle durante la memoria.

II. Arquitectura del Sistema

En esta memoria se utiliza una tarjeta de video de la familia Geforce 6x, la NVIDIA Geforce 6600gt. La siguiente tabla indica las especificaciones técnicas del equipo y GPU utilizadas.

Para esta memoria se utiliza el siguiente computador:

- AMD Athlon 64bit 3.2+ GHz
- 1 GB memoria DDR RAM
- Disco duro SATA 200 GB
- Placa madre Nforce 3 AGP
- Tarjeta de video NVIDIA Geforce 6600GT 128 MB RAM

Las especificaciones técnicas de la tarjeta de video son:

- procesador 500 GHz
- 128 MB memoria GDDR3 RAM 900 MHz
- Bus de memoria 14.4 GB/seg
- 8 Píxel pipelines (tubo de datos)
- Interfaz de memoria 128 bits
- 8 procesadores de fragmentos (píxel shader)
- 8 procesadores de vértices (vertex shader)
- 375 millones de vértices por segundo

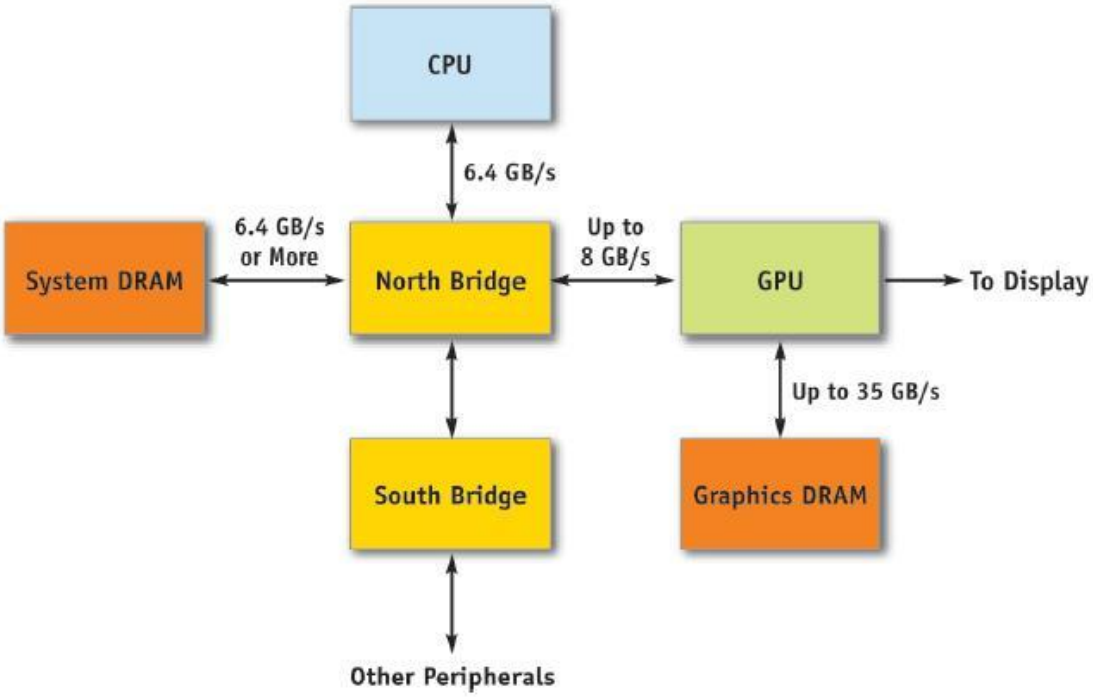


Figura 1

- En la Figura 1 [1] se ve el flujo de datos. Se puede apreciar que la velocidad de transmisión de datos entre la GPU y su memoria es considerablemente superior a la que existe entre la CPU y la memoria RAM. Sin embargo, la velocidad es la misma aproximadamente entre la memoria RAM, la CPU y la GPU.

II.1 Arquitectura de la Tarjeta de Video

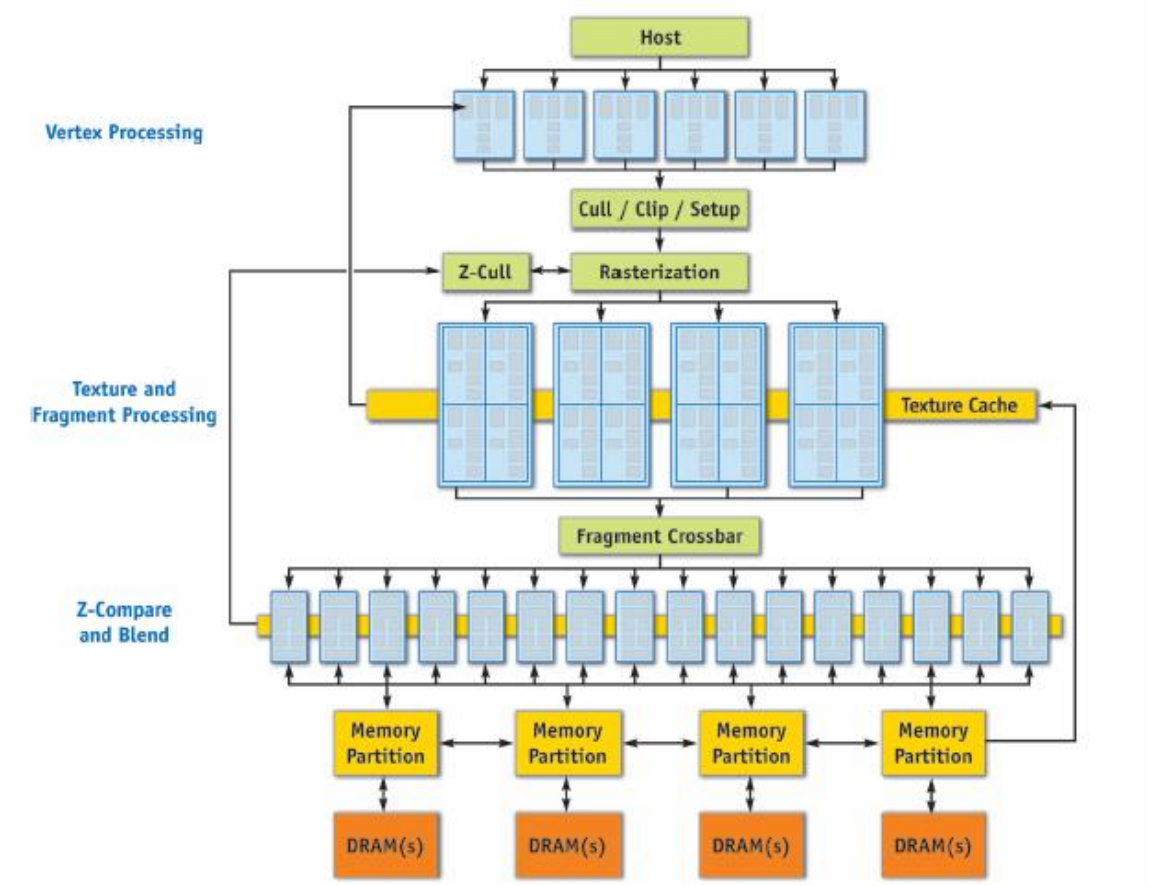


Figura 2

En la Figura 2 [1] se pude observar la arquitectura simplificada de la Geforce 6600GT. El host (en este caso, la CPU) envía datos y comandos a la GPU a través de la memoria RAM. La CPU envía un *stream* de comandos que inicializan y modifican el estado de vértices y texturas. Los procesadores de vértices (llamados vertex shaders) son capaces de analizar los datos, y al ser programables, cambiarlos según lo pedido por la CPU y enviar hacia los siguientes pasos la información necesaria para mostrarlos en pantalla o escribir

en memoria. La capacidad de los vertex shaders de ser programables son lo que hace posible crear programas no gráficos en las GPU.

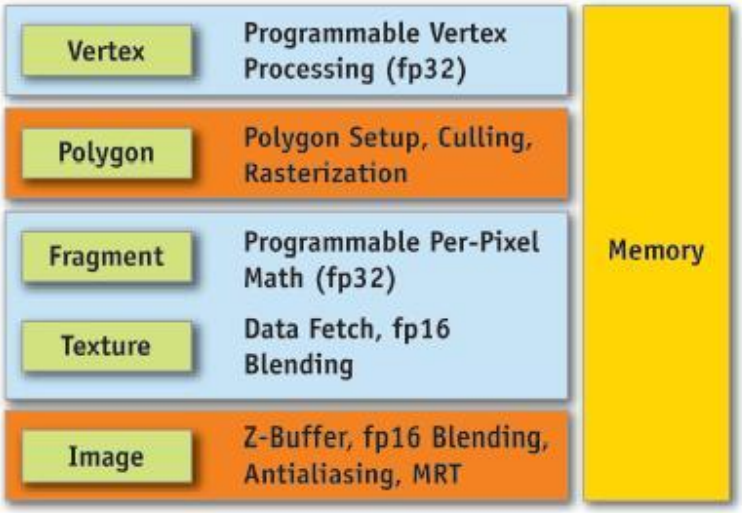


Figura 3

En la Figura 3 [1], se muestra la vista de la arquitectura como un *pipeline* gráfico. Se muestre el procesador programable de los vértices, el motor de transformación y *culling* de polígonos, el procesador programable de fragmentos (o píxeles) y el motor de texturas y filtros de imágenes. Es importante notar que tanto el motor de vértices y el de fragmentos tienen una precisión de punto flotante de 32 bits, lo que ayuda a la exactitud de los datos a programar. Los datos están guardados según el estándar IEEE 754 de precisión simple. Sin embargo, las operaciones no siguen la norma IEEE 754, sino una propia que se adapta al hardware, pero que entrega resultados similares.

Para el uso de aplicaciones no gráficas de la GPU, se puede ver una vista alternativa en la Figura 4 [1]. Se aprecia que existen dos bloques programables: el procesador de vértices y el de fragmentos, que soportan punto flotante de 32 bits, tanto para operaciones como para datos intermedios. Ambos usan el motor de texturas como una unidad de acceso de datos, a una velocidad superior a los 14.4 GB/segundo.

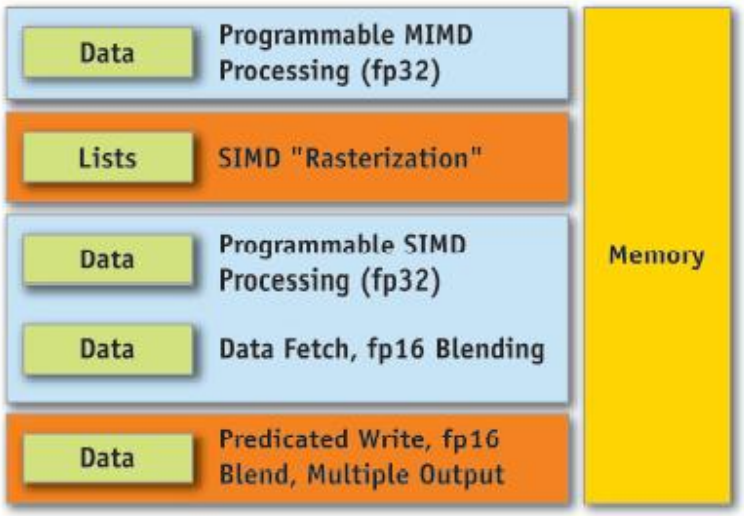


Figura 4

A través de la arquitectura podemos apreciar que la programación de aplicaciones no gráficas en tarjetas de video se centra en la capacidad de programar los procesadores de vértices y fragmentos, por lo que es importante saber su potencial de procesamiento.

Procesador de Vértices (Vertex Shader)

- Posee 512 instrucciones estáticas y 65536 instrucciones dinámicas. Las instrucciones estáticas son el número de instrucciones de un programa al momento de compilar. Las instrucciones dinámicas son el número de instrucciones al momento de ejecutar el programa.
- Posee 32 registros globales que se pueden usar en un programa de vértices.
- Instanciamiento. Significa que se pueden mandar múltiples copias de un objeto con una sola llamada.
- Control de Flujo dinámico. La GPU soporta la ramificación y los ciclos de forma nativa.

- 6 operaciones MAD (multiplicar-sumar) vectoriales y una operación escalar por ciclo de reloj

Procesador de Fragmentos (Fragment Shader)

- Posee 65536 instrucciones estáticas y 65536 instrucciones dinámicas, pero existen limitaciones en cuánto tiempo el sistema puede esperar para que el programa termine de ejecutar.
- Múltiples Objetivos de Renderizado (MRT). Esto significa que el procesador de fragmentos puede enviar 4 canales de colores, más un valor de profundidad. Esto es útil en datos escalares, ya que se pueden enviar hasta 16 valores en una sola pasada.
- Control de Flujo Dinámico.
- Atributos Indexados. Permite que ciclos puedan ejecutar la misma operación en distintas salidas.
- 16 operaciones MAD vectoriales, más 16 operaciones de multiplicación por ciclo.

Dado que la Geforce 6600GT contiene 8 procesadores de vértices y 8 procesadores de fragmentos, se puede estimar que existen unas 310 operaciones posibles en un solo ciclo de reloj, y dada la velocidad de 500 GHz de la GPU, se pueden ejecutar billones de instrucciones por segundo, lo que valida el potencial de las GPU para ejecutar programas no gráficos.

II.1 Lenguaje de Programación de GPU

Para este trabajo se ha elegido el lenguaje de programación CG, creado por NVIDIA [11]. CG es un lenguaje basado en C, pero con modificaciones creadas para compilar programas que optimicen el uso de la GPU. El código CG es casi el mismo que C, con la misma sintaxis para declaraciones, llamadas de funciones y la mayoría de los tipos de datos.

A diferencia de las CPU, que tienen solo un procesador programable, las tarjetas de video actuales tienen al menos dos, el procesador de vértices y el procesador de fragmentos (pixeles). CG es usado para programar ambos, y se puede compilar como código nativo de GPU en la ejecución, o antes. Este código es usado en conjunto con OpenGL o DirectX para mostrar los resultados en pantalla.

Las CPU en general tienen las mismas capacidades, por lo que un programa en C puede compilarse para la mayoría de ellas. En cambio, las GPU son muy distintas entre ellas, y esto se refleja en que CG necesita saber qué tipo de capacidades tiene la GPU a usar. CG usa el concepto de *profiles* para identificar los conjuntos de funciones posibles de utilizar en una GPU, por ejemplo, si usa DirectX 8.0 o DirectX 9.0.

El código usado en C en general se basa en la función main() que es donde empieza la ejecución. En cambio, los programas en CG pueden tener cualquier nombre y tienen la siguiente sintaxis:

[tipo de dato de retorno] <nombre de programa>(<parámetros>) [:<nombre semántico>] { /*.....*/ }

Las GPU trabajan con *streams* (flujos) de datos. El procesador de fragmentos trabaja con streams de píxeles y el procesador de vértices en streams de vértices.

Los programas en CPU en general se diseñan para ser ejecutados una sola vez. Los programas en GPU, en cambio, son ejecutados repetidamente, cada vez por un elemento del stream. Para esto, CG usa un modelo basado en programación de streams, que tiene capacidades diferentes a lo esperado en C.

Tipos de Datos en CG

CG tiene 7 tipos de datos básicos.

- Float: número de punto flotante de 32 bits.
- Half: número de punto flotante de 16 bits.
- Int: número entero de 32 bits.
- Fixed: número de punto fijo de 12 bits.
- Bool: dato booleano usado en comparaciones tipo if.
- Sampler: referencia a un objeto de textura.
- String: usado para contener datos dentro de CG

También existe soporte para vectores, usando variaciones de los datos básicos. Por ejemplo, float3 es un vector con 3 números tipo float. Se puede utilizar matrices con el mismo concepto, por lo float3x3 sería una matriz de 3x3 números tipo float.
En general, CG es un lenguaje similar a C, enfocado a las capacidades de las GPU, con una librería de funciones dedicadas a tareas gráficas, tales como anti-aliasing, bump mapping, y varios efectos especiales. Para los objetivos de este trabajo, CG permite el acceso a los procesadores de vértices y fragmentos, para programarlos de tal manera que permita su uso para tareas no gráficas, tales como el refinamiento de mallas geométricas.

Modo de Uso

Para crear un programa en CG, se puede compilar independientemente y agregarlo a un programa principal mediante llamadas, o se puede compilar mediante el uso de librerías del programa principal. Por ejemplo, se usa C o C++ como base, que se preocupa de las interacciones con el sistema, crear ventanas, esperar comandos de teclado, entre otros. Para incluir CG se necesita llamar al código de éste antes de un llamado OpenGl que dibuje una figura geométrica o asigne una textura. Esto hará que la GPU organice sus shaders de tal forma que aplique los comandos de CG a la figura o textura siguiente.

Lo siguiente sería una función CG:

```
float4 cambiacolor( float4 color : COLOR )
{
        float4 OUT;
        OUT = color;
        OUT[0] = 0;
        return OUT;
}
```

Esta función simple toma el color asignado a OpenGl en ese momento (llamado COLOR), y le quita el elemento rojo de éste, retornando las partes verde y azul de éste intactos. En un programa C, esto se incluye antes de la llamada a dibujar, como:

```
void dibujar()
{
        glColor4f(1.0,0.0,0.0,1.0);
        cgGLBindProgram(cambiacolor);
        glBegin(GL_TRIANGLES);
        glVertex2f(-0.8, 0.8);
        glVertex2f(0.8, 0.8);
        glVertex2f(0.0, -0.8);
        glEnd();
}
```

Si obviamos la parte CG, esta función asigna un color rojo a OpenGl, y dibuja un triángulo con ese color. Ahora, al insertar la llamada a CG, se puede transformar los valores de OpenGl para obtener, en este caso, un triángulo de color negro. A pesar de la simpleza de este ejemplo, se puede ver que podemos tomar un elemento de la GPU, y transformar la salida según nuestra función en CG.

Dado que el objetivo de esta memoria no es profundizar en la herramienta usada para su programación, se puede encontrar más información en "*The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics", Randima Fernando, Mark J. Kilgrad,* Addison-Wesley Professional; Pap/Cdr edition (February 26, 2003). También ver en los anexos el código incluido.

III. Aplicación de la GPU para algoritmos de refinamiento de triangulaciones

III.1 Refinamiento de Mallas Geométricas

Las GPU son hardware especializado, con un número menor de operaciones que las CPU, pero algunas de éstas son considerablemente más rápidas que su contraparte. Un ejemplo de esto serían las operaciones vectoriales como la multiplicación de matrices, que la GPU hace más rápido debido a que está optimizado para ello. Aprovechar estas ventajas del hardware especializado de la GPU nos debería dar un incremento en el rendimiento de aplicaciones que las explotan.

Una de las ramas importantes de la computación gráfica es el estudio y aplicación de técnicas de refinamiento de mallas geométricas. Generalmente, se usa una malla geométrica discreta para definir un espacio donde ciertas computaciones son realizadas, tales como cálculo de pesos, dinámicas de fluidos o simulación de estructuras, entre otros. Para obtener cálculos más precisos o definir áreas de una mejor forma, se requiere de refinar las mallas geométricas de alguna forma, y para este fin existen muchas técnicas de refinamiento.

En pocas palabras, un refinamiento de mallas consiste en crear un mayor nivel de detalle en una malla geométrica computacional mediante la inserción de primitivas (objetos geométricos simples que la GPU puede dibujar de forma nativa), siendo el más usado el triángulo. Esto se hace mediante algoritmos de inserción dinámicos (que se adaptan a la malla en tiempo de ejecución) o mediante el reemplazo de una malla por otra más o menos refinada que la anterior.

El grupo de refinamientos más usados en el último tiempo son los refinamientos adaptables de mallas (adaptive mesh refinements). Este grupo se define por refinar, o subdividir, mallas según la necesidad de la visualización. En otras palabras, se adaptan a lo que se requiere observar en un espacio o tiempo dados. Un ejemplo de refinamiento adaptable de mallas sería el dibujar un círculo como un triángulo si está muy lejos, como un cuadrado si está un poco más cerca, y agregar lados mientras más cerca se encuentre, hasta que sea un círculo definido por muchas aristas. Estos refinamientos se caracterizan por ser guiados por una función que indique el nivel de subdivisión según algún criterio, en vez de reemplazar una malla por otra.

Anteriormente, se usaban varias mallas de distinto nivel de detalle para modelar algún objeto o simulación. Usando las nuevas técnicas, se puede obtener distintos niveles de refinamiento sin la necesidad de varias mallas, lo que se traduce en un ahorro de almacenaje en disco y completo control del nivel de detalle en los lugares que se necesiten.

Los algoritmos de refinamiento de mallas basados en subdivisiones, tales como Loop (Biermann et al) y Catmull-Clark, requieren del uso de información dinámica adyacente, que es obtenida a través de estructuras de datos dinámicas. Esto es posible de implementar en CPU, pero en las GPU tales estructuras de datos no son factibles. Esto se debe a la naturaleza paralela de los cómputos generados, y que las operaciones realizables en cada ciclo son limitadas en número.

III.2 Algoritmo de refinamiento LEPP Bisección

Existen varios algoritmos de refinamiento de mallas geométricas, que entregan resultados según su enfoque, que puede ser rapidez, exactitud, o uso de recursos. Este trabajo se basa en el algoritmo LEPP Bisección. Este algoritmo garantiza mallas refinadas localmente que contienen triángulos de buena calidad, o sea, que no tiene ángulos menores a 30°, en la mayoría de los casos. En la práctica, entrega refinamientos que tienen un óptimo número de puntos y las mejores mallas con el número necesario de triángulos.

Este algoritmo se basa en la definición del LEPP (Longest Edge Propagation Path), o camino de propagación por la arista más larga de un triángulo La definición formal es la siguiente:

Definición LEPP: Para cualquier triángulo t0 de una triangulación T, el camino de propagación por la arista más larga será una lista ordenada de todos los triángulos t0, t1, t2, ...t(n-1), tn, tal que ti es el triángulo vecino de t(i-1) por el lado más largo de t(i-1), para i = 1,2,....n. Adicionalmente, será llamado LEPP(t0).

Definición triángulo terminal: dos triángulos adyacentes (t,t*) serán llamados un par de triángulos terminales si comparten sus respectivas aristas mas largas comunes. Adicionalmente t será un triángulo terminal si su arista más larga se encuentra con un borde de la malla.

Usando estas dos definiciones, se puede mostrar el algoritmo LEPP bisección como sigue:

```
Algoritmo LEPP bisección (t, T)
    Mientras t se encuentre sin modificar hacer
        Encontrar el LEPP de t
        Bisectar el último triángulo de LEPP(t).
```

Existen varios algoritmos que usan el LEPP, tales como LEPP Delaunay, Backwards Delaunay, entre otros. Sin embargo, todos tienen la misma estructura algorítmica, y difieren generalmente en la forma de refinar el triángulo terminal. La variante de bisección fue escogida ya que produce resultados de una calidad semejante a los otros, con menos pasos y menos computación, ya que no requiere calcular la triangulación Delaunay, que es caro en términos de instrucciones. Esto es útil ya que en las GPU están basadas en instrucciones atómicas, por lo que el uso de computaciones más simples es una ventaja si se modifica este algoritmo para funcionar en paralelo.

III.3 Refinamiento de mallas en la GPU

El algoritmo LEPP Bisección señalado anteriormente funciona correctamente si es diseñado con una CPU en mente, y es posible programarlo exitosamente con varias técnicas, tales como listas enlazadas, punteros, estructuras o arreglos, entre otros. Luego la información es enviada a una GPU, cuyo único trabajo es mostrar la malla en la pantalla.

Este trabajo se enfoca en los pasos necesarios para que el trabajo de la GPU sea mayor, liberando recursos de la CPU para otras tareas. Sin embargo, la transición no es trivial, ya que ambos procesadores, la CPU y la GPU, tienen arquitecturas distintas y funcionalidades que no comparten.

Primero, en las GPU no se pueden usar punteros, estructuras, listas enlazadas, o cualquier abstracción de datos que requiere de información o datos siendo usados al mismo tiempo. Las GPU funcionan con datos que son del tipo

vectorial, que identifican información tales como píxeles y vértices. Los vectores pueden representar colores, posiciones, transparencias, etc. Toda esta información es procesada en paralelo; por lo tanto, mientras la GPU procesa los píxeles de la pantalla, no puede acceder a la información del píxel vecino ya que está siendo procesado al mismo tiempo. Esto significa que obtener información dinámica de datos siendo procesados en paralelo es difícil en la práctica. En suma, cada vértice, píxel o vector de datos es independientemente procesado.

Otro problema es cómo se guardan los datos en la GPU. En la CPU se pueden guardar, modificar y eliminar variables independientes, además de crear arreglos y varias estructuras de datos según la necesidad del momento. En las GPU, sin embargo, es necesario tener todos los datos en un arreglo: tal arreglo puede ser 1D, 2D o 3D, pero la noción de variables independientes no existe. Por lo tanto, todos los datos necesarios deben ser creados como arreglos, ordenados según se necesiten. Si se quiere guardar datos, éstos también serán escritos como un arreglo de algún tipo.

Es interesante indicar que la memoria que accede un GPU es bastante inferior en cantidad que la de un CPU. Los CPU además cuentan con más memoria caché, por que el nivel de almacenamiento en memoria de las GPU no es comparable con las CPU, y hay que tener en cuenta eso durante la adaptación de un algoritmo. Además, las GPU tienen un nivel de computación mucho mayor que la latencia de compartir información; esto significa que es mejor tener computaciones intensas sobre todos los datos que compartir datos entre computaciones. Esto se debe a que el ancho de banda de comunicación entre la GPU y la memoria principal es de 2 a 4 Gygabites por segundo, y el poder de procesamiento de una GPU es de las docenas y hasta cientos de GFLOPS (miles de millones de operaciones de punto flotante por segundo).

Otro punto menor es la poca factibilidad de condicionales del tipo if – else en las GPU. Como todos los datos son procesados en paralelo, no se puede

deducir los problemas que esto trae en las GPU, pero sí existen bajas de rendimiento en las GPU capaces de condicionales. Actualmente, se pueden incluir condicionales en las GPU, pero es caro en términos de instrucciones, y en ciertos casos, una rama mal definida puede parar la ejecución de otras partes del código.

Las diferencias y problemas enunciados hasta ahora consideran sólo pequeños cambios en el diseño de un algoritmo adaptado de una CPU a una GPU. Sin embargo, el mayor problema en la adaptación de un algoritmo de refinamiento de mallas consiste en la imposibilidad en el hardware actual subdividir superficies en la GPU, o hacer “tessellation”. Las GPU son programables mediante sus procesadores de píxeles y vértices; esto significa que cada píxel o vértice es modificable, pero no se pueden agregar nuevos vértices mediante la programación. Esto es, sin duda, una de las razones que no existe mucho estudio de algoritmos de refinamiento de mallas en la GPU, siendo ésta un área tan cercana a la computación gráfica.

Por lo tanto, la mayor dificultad será resolver cómo agregar puntos a una malla (la base de un refinamiento) cuando no existe soporte nativo para esto en las GPU.

III.4 Revisión del estado del arte en refinamiento de triangulaciones en la GPU

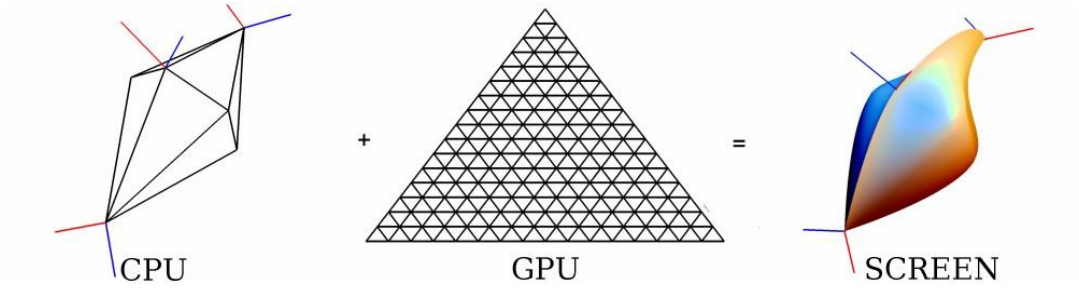
Aunque existan pocos trabajos acerca del problema de la inserción de puntos mediante la GPU, sí existen algunos que lo resuelven mediante distintos métodos.

Una propuesta interesante es de “GPU-based trimming and tessellation of NURBS and T-spline surfaces, Michael Guthe, Ákos Balázs, Reinhard Klein”. Como no existe soporte en hardware de las GPU para NURBS ajustados (trimmed NURBS), desarrollan un algoritmo que entre otras cosas envía un

arreglo de puntos de control, tales como curvas de bézier, y rellenan las curvas con triángulos desde la GPU. Esto genera un gran ahorro de ancho de banda entre la tarjeta gráfica y la CPU, por lo que se pueden mostrar NURBS de forma más detallada y con un rendimiento superior que si se usara sólo la CPU. Este paper demuestra la posibilidad de refinar mallas, en este caso NURBS, a pesar de no existir soporte directo.

Una solución al refinamiento de mallas en GPU que provee una ayuda importante a este trabajo es mostrado en “Generic Mesh Refinement On GPU, Tamy Boubekeur and Christophe Schlick”. En éste, demuestran la posibilidad de refinar en la GPU mediante una técnica de patrones.

En resumen, su algoritmo consiste en tener una malla gruesa creada en la CPU y un “patrón de refinamiento” en la GPU. Este patrón consiste en una malla fina de triángulos, por ejemplo de 5x5. Entonces, cargan la malla gruesa en la GPU, para obtener los vértices de ésta. Luego, en vez de mostrar cada triángulo de la malla gruesa, hacen un renderizado del patrón de refinamiento donde éste se encuentra. Esto crea efectivamente un refinamiento visual.



Usando la idea del patrón de refinamiento, podemos diseñar el algoritmo LEPP Bisección en la GPU, cambiando la idea global del patrón a una local.

III.5 LEPP Bisección en la GPU

El algoritmo LEPP se basa en la generación de una lista de triángulos que corresponden al camino de propagación por la arista más larga de un triángulo inicial. Dada la arquitectura de la GPU, tal lista no se puede calcular ya que requiere de información de vecinos y de estructuras de datos que no se pueden definir mediante el lenguaje de programación de las GPU. En este caso, la lista requiere de punteros o referencias a otros triángulos de la lista, y en CG no se puede administrar punteros o llamarlos dentro de sus funciones.

Por lo tanto, para el algoritmo LEPP Bisección en una GPU se hará el cálculo de la lista LEPP(t) en la CPU. Cuando se encuentre el triángulo terminal, se enviará la información de cuál triangulo es a la GPU para ser bisectado ahí.

Un problema con esto sería que no genera beneficio si la bisección se hace en la CPU o en la GPU, ya que no necesita de mucho cálculo. Si embargo, si acumulamos varios LEPP a calcular, entonces el ahorro generado por no enviar la malla cada vez a la GPU sería de consideración.

Los pasos de este algoritmo serían:

1. Dentro de una malla de triángulos T, definir los triángulos iniciales t0,t1,t2,....,tn.
2. encontrar el LEPP de cada ti.
3. Enviar el triángulo terminal de cada LEPP a la GPU.
4. La GPU bisecta los triángulos recibidos mediante un patrón de refinamiento.
5. Seguir hasta que no existen triángulos en las listas LEPP.

Para que este algoritmo sea eficiente, se necesita tener una malla en la memoria del CPU y otra igual en la GPU. Cada triángulo debe definirse mediante un índice. Al encontrar los triángulos terminales, se envía un arreglo de los índices de éstos a la GPU. La GPU luego adapta un patrón de refinamiento, que en este caso sería simplemente dos triángulos unidos por una arista, a cada índice. Esto necesita el uso de procesadores de vértices, para adaptar los vértices de la malla con los del patrón, y el lado que será bisectado.

Con un número suficientemente grande de listas de triángulos LEPP, el ahorro por no enviar la malla a la GPU sería suficiente como para obtener un incremento en el rendimiento del algoritmo.

III.6 Problemas de LEPP Bisección en GPU

Uno de los objetivos de modificar este algoritmo a una GPU es obtener beneficios de rendimiento. Sin embargo, debido a que la mayoría de los pasos en el algoritmo propuesto son ejecutados en la CPU, no existe una densidad de cómputo necesaria para obtener tales beneficios. Calcular el LEPP de un triángulo es costoso en recursos, aún más considerando que no se puede adaptar tal proceso en la GPU, debido a limitaciones de arquitectura. Si lo único que se genera en la GPU es la bisección, que es la menor parte de los cálculos, entonces todo el peso de la ejecución sería en la CPU. Sin importar qué tan rápido la GPU haga las bisecciones, tendrá que esperar a que la CPU termine de generar los LEPPs, y que los envié a la GPU.

Por lo tanto, podemos conjeturar que esta implementación no tendrá beneficios con respecto a la implementación en la CPU, a menos que el número de bisecciones a realizar sea en el orden de los millones, para aprovechar la mayor rapidez en el cómputo de vértices en la GPU. Aún así, gran parte de la

carga sigue estando en la CPU por lo que el cuello de botella seguirá siendo éste.

III.7 Refinamiento por Procesador de Vértices

Otra implementación posible es utilizar el concepto de patrones y aplicarlo de una forma global, en vez de localizada como LEPP Bisección. Usando una heurística de los principios de LEPP Bisección, se pueden tomar las aristas más largas de una malla de triángulos y aplicar bisección en aquellos triángulos que comparten esa arista. Esto se basa en que por lo menos la mayor de las aristas pertenece siempre a un triángulo terminal, por lo que si se aplica bisección a un subconjunto de éstas se incrementará el refinamiento global de una forma eficiente.

Los pasos del algoritmo en la CPU serían:

1. Dentro de una malla de triángulos T, ordenar las aristas de mayor longitud a menor, y escoger un subconjunto L de las mayores.
2. Tomar la arista más larga de L, obtener los triángulos que la comparten.
3. Aplicar bisección en esos triángulos.
4. Dado que la arista ha sido dividida, volver a 2. y seguir con la próxima.

Si tomamos en cuenta que en la GPU, el Procesador de Vértices trabaja en paralelo, se puede modificar el algoritmo en:

1. Dentro de una malla de triángulos T, ordenar las aristas de mayor longitud a menor, y escoger un subconjunto L de las mayores.
2. Mandar el subconjunto L al Procesador de Vértices, siendo cada elemento un par de vértices.

3. el Procesador de Vértices encuentra los triángulos t que comparten las aristas, y aplica el patrón de bisección a éstas.

Las ventajas de esta solución incluyen una menor utilización de recursos de la CPU, y una mayor eficiencia en la bisección, ya que se conoce el subconjunto de vértices al que se le aplica, por lo que se puede mandar aquella información a la GPU y ser procesada en paralelo. Nuevamente, en el último paso la bisección se realiza mediante el patrón guardado en la GPU, que lo visualiza sobre la malla general simulando la bisección.

III.8 Problemas de Refinamiento por Procesador de Vértices

El refinamiento por Procesador de Vértices consigue una ventaja importante sobre la de LEPP Bisección en GPU: una menor densidad de cómputo en la CPU, y una mayor rapidez de la bisección en el último paso, ya que se puede hacer en paralelo. Esto puede generar la rapidez necesaria para que exista un mayor rendimiento mediante el uso de la GPU en vez de la CPU.

Sin embargo, estas dos soluciones tienen un problema mayor. La aplicación de patrones en la GPU, para sobrellevar el problema de no poder agregar vértices dentro de la GPU, es una solución visual. Al momento de visualizar una malla refinada mediante estos métodos, se observará que el refinamiento se produce, incluso con ventajas de rendimiento sobre los métodos basados en CPU. Lamentablemente, esto no se traduce cuando se requiere obtener los datos de la nueva malla refinada. Dado que los patrones son sólo pequeñas mallas independientes sobre la malla original, sus valores no son añadidos a ésta. Si se requiere de obtener los datos de la nueva malla refinada, se debe procesar todos los nuevos patrones e interpolar los datos de éstos con la malla original.

El proceso de adquirir los datos de la malla refinada puede ser rápido si el refinamiento es pequeño; pero costoso si los cambios son grandes. Debido a que los beneficios en rendimiento de una solución basada en GPU están ligados a la densidad de cálculo, mientras más refinamientos se hagan, peor será el proceso de mezclar los datos de patrones y malla original. Por supuesto, esto no es un problema en las soluciones en CPU, ya que éstos modifican la malla original.

El uso de patrones como solución al problema de la falta de hardware en la GPU para añadir vértices a una lista es original y muy útil en aplicaciones puramente visuales, pero las falencias mostradas le restan utilidad en otro tipo de aplicaciones que necesitan de los datos obtenidos a través del refinamiento. Por ejemplo, si después de un refinamiento se necesita calcular la fuerza de gravedad en los nuevos puntos, con estas soluciones se requeriría de más cálculos para tomar en cuenta los nuevos puntos en la malla. Debido a que existe un mayor número de aplicaciones con mallas geométricas refinadas dedicadas a más acciones que la pura visualización, esta memoria intenta encontrar soluciones que sean más completas.

III.9 Refinamiento basado en GPGPU

Las soluciones mostradas intentan atacar el problema del refinamiento de mallas mediante el uso directo de vértices y conceptos propios de la Computación Gráfica, usando como medio la posibilidad de programar los vértices en la GPU. A pesar de que existen muchos problemas que se pueden solucionar de esta manera, todos ellos se relacionan con la posibilidad de alterar propiedades de los vértices en sí, tales como su posición, y no el añadir o quitar vértices de una lista enviada a la GPU.

Como no se puede hacer “tessellation” mediante el hardware actual, se puede simular el uso de listas o arreglos mediante un concepto llamada GPGPU, General Purpose Computation on GPU, o Computación General en GPU. Dada

la posibilidad de programar los Procesadores de Vértices y Píxeles, se pueden hacer cómputos que no sean sólo asociados a la Computación Gráfica, sino a programas generales asociados a la CPU.

Para poder realizar cómputos en la GPU, se necesita enviarle datos que sean distintos a vértices o píxeles. Como no pueden usar arreglos u otros tipos de datos, las aplicaciones GPGPU aprovechan las texturas como arreglos. Las texturas en general son usadas para asociar vectores de colores, normales y luminosidad a una figura geométrica, y la GPGPU usa tales vectores como datos para sus aplicaciones no gráficas. Normalmente, si uno asocia una textura a un triangulo o alguna cara, la GPU ejecutará los procesos del Procesador de Píxeles, para modificar los colores o luminosidad de la textura, por ejemplo, para darle un suavizado de Gauss. Lo que hace la GPGPU es modificar los procesos para que en vez de actuar con colores actúe con otro tipo de datos y otro tipo de algoritmos. Los resultados se guardan en la textura misma que luego puede ser interpretada.

Existen muchas aplicaciones GPGPU que obtienen beneficios frente a su contrapartida en CPU, dado la rapidez y arquitectura paralela de los Procesadores de Píxeles en las GPU para ciertas operaciones. Algunos ejemplos serían el uso de GPGPU en algoritmos de ordenamiento, procesamiento de datos, simulación de fluidos, entre otros.

Para esta memoria, el uso de texturas como arreglo de datos nos da la posibilidad de trabajar de forma símil a la CPU, usando las texturas como una lista de vértices de una malla geométrica. Gracias a esto, podemos añadir o quitar vértices, y por lo tanto, realizar un refinamiento de mallas que mantiene los datos en un solo lugar.

Otra ventaja consiste en que existen más Procesadores de Píxeles que de Vértices en una GPU, lo que otorga una mayor capacidad de cómputo que

las soluciones anteriores. Junto con la posibilidad de modificar el número de vértices, una solución al refinamiento de mallas basado en GPGPU puede traer ventajas apreciables.

Lamentablemente, la aplicación de estas técnicas es un problema que requiere más tiempo del que se dispone para esta memoria. Sin embargo, los puntos discutidos aquí arrojan más información acerca del tema, y se exploraron las ventajas y desventajas que éstas tienen. Se espera que esto sea de ayuda para futuras investigaciones acerca de la aplicación de técnicas de refinamiento de mallas geométricas en la GPU.

IV. Mecánica de Fluidos

Para entender los conceptos utilizados en esta memoria se necesita una breve introducción a la mecánica de fluidos. Esta es la base de donde se adaptan los algoritmos y ecuaciones necesarias para desarrollar una simulación de fluidos.

La mecánica de fluidos es una rama de la física que se especializa en el movimiento de fluidos, tanto gaseosos como líquidos. Los fluidos se caracterizan por su incapacidad de resistir esfuerzos cortantes, o sea, el esfuerzo interno resultante de las tensiones paralelas a la sección transversal de un sólido deformable. La hipótesis en la que se basa la mecánica de fluidos es la hipótesis del medio continuo.

En la hipótesis del medio continuo se considera a que los fluidos son continuos en el espacio que ocupan, lo que significa que se ignora su estructura molecular y las discontinuidades asociadas a ésta. Por lo tanto, esta hipótesis supone que las propiedades de un fluido son funciones continuas.

Un concepto importante en la mecánica de fluidos es el de partícula fluida. Una partícula fluida es una masa elemental del fluido que en un momento determinado se encuentra en un punto del espacio. Esta masa elemental debe contener un gran número de moléculas, y además no debe contener en su interior una variación en las propiedades macroscópicas del fluido. De este modo, se puede asignar a cada partícula fluida un valor por propiedad del fluido.

Existen dos formas de describir el movimiento de un fluido. La descripción Lagrangeana modela el comportamiento siguiendo a cada partícula fluida en su movimiento, por lo que se busca funciones que indiquen la posición y descripción de sus propiedades como fluido en cada instante. La descripción

Euleriana se basa en asignar a cada punto del espacio un valor para las propiedades del fluido, por lo que no está ligado a la partícula fluida que se encuentre en ese punto y momento. Generalmente, la descripción Euleriana es la más usada, ya que en la mayoría de los casos entrega mayor utilidad.

La mecánica de fluidos se rige por tres ecuaciones fundamentales: La ecuación de continuidad, la ecuación de cantidad de movimiento, y la ecuación de conservación de energía. Estas ecuaciones son derivadas de los principios de conservación de la mecánica y termodinámica, aplicada a fluidos. Este conjunto de ecuaciones, en su forma diferencial, se les llama ecuaciones de Navier-Stokes.

Debido a la complejidad de estas ecuaciones, no existe una solución general para todo tipo de fluidos. Para solucionarlos, se requiere estudiar cada caso para encontrar simplificaciones en el modelo que faciliten su resolución. En muchos casos es difícil encontrar soluciones analíticas, por lo que una solución numérica con el apoyo de computadores es un área creciente en este tema.

IV.1 Revisión del estado del arte en simulación de fluidos

La mecánica de fluidos y su simulación está íntimamente ligada a la Computación. Esto se debe a que la complejidad de las ecuaciones que la modelan, en la mayoría de los casos, sólo se pueden resolver mediante métodos numéricos y no analíticos. Debido a esto, los cálculos necesarios para simular fluidos de distinta naturaleza ha sido trabajo de algoritmos usados en un computador.

Antes de que existieran simulaciones gráficas, se utilizaba el método Lagrangiano, basado en las partículas fluidas, para modelar diversos fenómenos. Uno de lo más antiguos es de Lucy [17] , para resolver problemas astrofísicos usando Hidrodinámicas de partículas suavizadas (SPH, por sus siglas en inglés). Estas partículas guardan la información de densidad de masa y velocidad, y son seguidas durante la simulación. Los valores en cada punto del espacio se derivan de estas partículas. Reeves [16] fue uno de los precursores del uso de sistemas de partículas para la simulación de fluidos, y ha sido una técnica que ha gozado de alta popularidad en estos tiempos. Premoze et al [18] idearon un método Lagrangeano para simular fluidos realistas, basado en la solución de Navier-Stokes a través de Moving-Particle Semi-Implicit Method (MPS por sus siglas en inglés). MPS asegura mayores niveles de incompresibilidad que SPH.

Los primeros trabajos en computación gráfica se basaban en la simulación Euleriana, o sea, basado en cálculos en puntos del espacio. Foster y Metazas [19] fueron los primeros en proponer la solución de las ecuaciones de Navier-Stokes en tres dimensiones, para poder recrear la visualización de un fluido dinámico. Stam [11] ideó los pasos para solucionar las ecuaciones de Navier-Stokes para fluidos incompresibles que se utilizan como base en esta memoria, pero su implementación era en la CPU. Además, creó un esquema

semi-Lagrangeano para simular gases dinámicos de gran estabilidad a expensas de viscosidad artificial. Más recientemente, Carlson et al [20] desarrollaron una solución Euleriana de fluidos altamente viscosos que pueden simular objetos derritiéndose. Goktekin et al [21] simulan fluidos que además de ser viscosos tienen propiedades plásticas y elásticas, aumentando el nivel de complejidad de las simulaciones.

Actualmente, existen varios trabajos que implementan la simulación de dinámica de fluidos en la GPU. En esta memoria se usa el método Euleriano, que provee una separación homogénea del espacio al cual aplicar computaciones. Esto es una ventaja al programar para GPU, debido al paralelismo de los algoritmos que se pueden aplicar sobre una malla de datos. Mientras más homogénea la distribución de lugares donde se ejecutan los algoritmos, se logrará una mayor ventaja en el rendimiento de la simulación. De los trabajos vistos, la mayoría de las implementaciones en GPU usan el modelo Euleriano, y lo mismo pasa con aquellas implementadas en CPU con el modelo Lagrangeano.

IV.2 Desarrollo de las ecuaciones de Navier-Stokes

Para poder desarrollar una simulación de dinámica de fluidos es necesario discutir los algoritmos asociados a éste. Con este fin discutiremos las ecuaciones que rigen el movimiento de un fluido. Estas ecuaciones se llaman ecuaciones de Navier-Stokes, y se necesita descomponerlas para adquirir ecuaciones más simples, que sean posibles de implementar en un algoritmo computacional.

Las ecuaciones de Navier-Stokes reciben su nombre de Claude-Louis Navier y George Gabriel Stokes. Son un conjunto de ecuaciones que describen el movimiento de un fluido. Describen de forma correcta fenómenos tales como el movimiento del aire, la atmósfera terrestre, corrientes oceánicas y todo fenómeno ligado a fluidos.

Las ecuaciones diferenciales que conforman a Navier-Stokes describen el movimiento de los fluidos. Es de interés notar que estas ecuaciones no establecen relaciones entre las variables, sino que indican relaciones entre el índice de cambio entre éstas. Esto significa que lo importante es qué tan rápido o lento una variable cambia con respecto a otra.

Las siguientes ecuaciones son una forma particular de las ecuaciones de Navier-Stokes, que toman en consideración que el fluido a modelar es incompresible y la densidad es constante. El fluido es modelado por su velocidad vectorial U(x,t) y su campo de presión escalar p(x,t) donde x = (x,y) y el tiempo es t. Si la velocidad y presión son conocidos para t = 0, entonces las ecuaciones de Navier-Stokes para fluidos incompresibles serían las mostradas en las ecuaciones 1. y 2.

$$\frac{\delta U}{\delta t} = -(U \cdot \nabla)U - \frac{1}{\rho}\nabla p + v\nabla^2 U + F$$

Ecuación 1.

$$\nabla u = 0$$

Ecuación 2.

ρ Representa la densidad del fluido constante, *v* la viscosidad y $F = (fx, fy)$ representa las fuerzas externas que actúan en el líquido. La primera ecuación es realmente dos ecuaciones, ya que U es un vector:

$$\frac{\delta u}{\delta t} = -(U \cdot \nabla)u - \frac{1}{\rho}\nabla p + v\nabla^2 u + f_x$$

Ecuación 3.

$$\frac{\delta v}{\delta t} = -(U \cdot \nabla)v - \frac{1}{\rho}\nabla p + v\nabla^2 v + f_y$$

Ecuación 4.

Con las ecuaciones 2, 3 y 4 tenemos tres ecuaciones y tres incógnitas.

Para entender mejor estas ecuaciones, las siguientes definiciones son de ayuda.

Advección ($-(U \cdot \nabla)U$) : La velocidad de un fluido causa que transporte objetos y otros en su flujo. El primer término de la ecuación 1 representa la advección del fluido y su capacidad de transportarse a sí mismo.

Presión ($-\frac{1}{\rho}\nabla p$): Debido a que las moléculas de un fluido se mueven alrededor de cada una, crean distorsiones en la propagación de las fuerzas. Esto hace que las partículas no se muevan de forma homogénea, sino que se congregan y aceleran según el segundo término de la ecuación 1.

Difusión ($v\nabla^2 u$): Los fluidos son más o menos densos dependiendo de sus propiedades, y esto hace que fluyan de forma rápida o lenta. A esto se llama viscosidad, o la resistencia de un fluido al flujo. La resistencia causa difusión del momento, y es el tercer término de la ecuación 1.

Fuerzas Externas (F): Indica la aceleración causada por fuerzas externas al fluido. Éstas pueden ser locales o de cuerpo. Las fuerzas locales son aplicadas a una parte del fluido; las fuerzas de cuerpo son aplicadas a todo el fluido, como ocurre con la gravedad.

IV.3 Descomposición de las ecuaciones de Navier-Stokes

Para poder encontrar una solución numérica a las ecuaciones de Navier-Stokes, se necesita dividir las ecuaciones en otras más simples. El algoritmo mostrado aquí se basa en una desarrollada por J. Stam [11].

Primero, se necesita conocer la descomposición de Helmhotz-Hodge. El cálculo de vectores indica que es posible descomponer un vector en sus componentes básicos, de la forma $v = x\hat{i} + y\hat{j}$. Lo mismo se puede hacer con un campo de vectores, según lo indica el siguiente teorema.

Teorema de Descomposición de Helmholtz-Hodge

Un campo de vectores w en D se puede descomponer de la forma:

$$w = u + \nabla p$$

Ecuación 5.

Donde u tiene divergencia cero y es paralela a δD , o sea, $u \cdot n = 0$ en δD .

Este teorema es usado para indicar en las ecuaciones de Navier-Stokes que el campo de velocidad, w, tiene divergencia distinta de cero, pero necesitamos que sea cero para cumplir la ecuación de continuidad. Por lo tanto, usando Helmholtz-Hodge, se puede encontrar la nueva velocidad sustrayendo el gradiente del campo de presión.

$$u = w - \nabla p$$

Ecuación 6.

Por lo tanto, según la ecuación 6, podemos computar las tres partes que afectan la velocidad: advección, difusión y fuerzas externas para obtener w, y necesitaríamos calcular el gradiente de la presión para obtener la velocidad final.

Para esto, se utiliza de nuevo Helmholtz-Hodge. Si aplicamos el operador de divergencia a ambos lados de la ecuación 5, nos da

$$\nabla w = \nabla \cdot (u + \nabla p) = \nabla \cdot u + \nabla^2 p$$

Ecuación 7.

Como sabemos por la ecuación 2 que $\nabla u = 0$, nos queda que

$$\nabla^2 p = \nabla \cdot w$$

Ecuación 8.

La ecuación 8 es llamada ecuación de Poisson para la presión de un fluido. Esto significa que podemos resolver la ecuación 8. después de computar w, y usar *p* y w para computar u en la ecuación 6.

Para poder obtener w, usamos la ecuación 6. y le aplicamos un operador P, que proyecta un campo de vectores w en su componente no divergente, u. El

operador P se basa en la descomposición de Helmholtz-Hodge. Si aplicamos P a la ecuación 6. queda

$$\mathrm{P}w = \mathrm{P}u + \mathrm{P}(\nabla p)$$

Ecuación 9.

Por definición de P, Pw = Pu = u. Por lo tanto, $\mathrm{P}(\nabla p) = 0$. Esto nos ayuda a simplificar las ecuaciones de Navier-Stokes. Aplicando P en ambos lados de la ecuación 1,

$$\mathrm{P}(\frac{\delta U}{\delta t}) = \mathrm{P}(-(U \cdot \nabla)U - \frac{1}{\rho}\nabla p + v\nabla^2 U + F)$$

Ecuación 10.

Si u es no divergente, entonces $\frac{\delta U}{\delta t}$ también lo es, por lo que $\mathrm{P}(\frac{\delta U}{\delta t}) = \frac{\delta U}{\delta t}$. Por la ecuación 9. tenemos que $\mathrm{P}(\nabla p) = 0$, por lo que nos queda

$$\frac{\delta U}{\delta t} = \mathrm{P}(-(U \cdot \nabla)U + v\nabla^2 U + F)$$

Ecuación 11.

Con la ecuación 11., tenemos una expresión que se puede definir en un algoritmo para solucionar las ecuaciones de Navier-Stokes. Primero, se debe obtener la advección, difusión y las fuerzas externas. Resolviendo éstos nos da el campo de velocidad divergente w, al cual se aplica el operador de proyección P para obtener el campo no divergente u. Para lograr esto encontramos *p* de la ecuación 8., y restamos el gradiente de *p* de w, en la ecuación 6.

Por lo tanto, los pasos para resolver las ecuaciones de Navier-Stokes son los siguientes:

1. Usar la ecuación 11. como base del algoritmo.
2. Encontrar la advección del fluido.
3. Encontrar la difusión del fluido.
4. Encontrar las fuerzas externas aplicadas al fluido.
5. Aplicar el operador de proyección sobre los resultados de los pasos 2, 3 y 4.
6. Se obtiene el campo de velocidad divergente w.
7. Se usa w para encontrar la presión *p* de la ecuación 8.
8. Se usa w y *p* para encontrar u de la ecuación 6.
9. Se tienen todos los datos necesarios para modelar el movimiento de un fluido según Navier-Stokes.

V. Implementación de la simulación de fluidos

La implementación de un modelo de fluidos se basa, en esta memoria, en una simulación Euleriana. Esto significa que se divide el espacio en varias celdas, y se calcula las propiedades del fluido en cada una de ellas. Debido a la naturaleza paralela de esta simulación, se ha decidido implementar la solución en una GPU.

La mayoría de las técnicas de simulación de fluidos se aplica para su desarrollo en la CPU, incluido los pasos mostrados para solucionar las ecuaciones de Navier-Stokes. Para poder implementar la solución en GPU, los pasos del algoritmo son similares, pero el paso final, la programación y secuencia de ejecución, son distintos.

Una diferencia esencial es el manejo de datos. En la CPU se pueden usar arreglos, valores temporales, y varias estructuras de datos para poder guardar datos y los vínculos entre ellos. En la GPU, sólo existe una estructura de datos manejable: las texturas. Las texturas son análogas a los arreglos, pero tienen limitaciones que las hacen menos aptas a tareas más complejas. Las texturas tienen sólo tres o cuatro canales de colores en los cuales guardar información de una celda dada. Afortunadamente, esta cantidad es adecuada para realizar las operaciones que una simulación de fluidos requiere.

Para el flujo de datos, la CPU realiza los pasos del algoritmo mediante iteraciones. Itera sobre todas las celdas, y luego hace otra iteración sobre cada una de ellas. Las GPU no pueden hacer iteraciones de esa forma, debido a la naturaleza de flujo paralelo que tienen. En cambio, la arquitectura de las GPU consideran a cada celda y le aplican las computaciones en forma paralela; por lo tanto, no necesita iterar sobre todas las celdas sino que se aplican las

instrucciones directamente sobre ellas. Al desarrollar una implementación, es símil a pensar que todas las celdas ejecutan la computación enviada simultáneamente.

Para los programadores, usualmente no es necesario preocuparse en dónde se va a escribir los datos, ya que cuando se actualizan lo hacen de forma transparente escribiendo en los arreglos o variables utilizados. En la GPU, en cambio, cada cambio se tiene que escribir en una textura, debido a que la memoria que usa siempre escribe a un búfer llamado *frame buffer*, que no puede ser accesado directamente. El frame buffer es donde se escribe usualmente la información de la pantalla para ser visualizada en la próxima actualización de ésta.

Tomando en cuenta estas diferencias, más las inherentes en los cambios de paradigma de programación, se puede implementar un programa que resuelva las ecuaciones mostradas anteriormente en la GPU.

V.2 Diseño general de la implementación

A continuación se detalla los rasgos generales del diseño de la implementación realizada. Debido a la diferencia en la arquitectura entre una implementación en una CPU normal y una GPU, una descripción de la forma en que trabaja la simulación es de ayuda para una mejor comprensión de su implementación.

El programa se divide en dos partes: el ciclo principal y los bloques ejecutados por el procesador de fragmentos. El ciclo principal está programado para ser ejecutado en la CPU, y los bloques en la GPU.

El ciclo principal es donde se inicializan las variables necesarias para ejecutar una ventana que dibuja en la pantalla. Esto se realiza usando las librerías de OpenGL, que también permiten acceso a los procesadores de fragmentos. En esta parte asignamos las texturas y figuras geométricas que computan los bloques de código en la GPU.

En los bloques de código asignados a los procesadores de fragmentos de la GPU se encuentran los algoritmos que resuelven las ecuaciones de Navier-Stokes descompuestas. Se tienen bloques distintos para computar la advección, la difusión viscosa, las fuerzas y la proyección.

Para la simulación de un fluido, en el ciclo principal creamos un cuadrado que abarca la ventana de visualización. La forma en la que los procesadores de fragmentos funcionan es que al momento de asignar una textura a un objeto, la GPU procesará el bloque de código asociado a esa textura. Así, cuando le

asignamos una textura al cuadrado inicial, se ejecutará el bloque de código asociado. Usando varias copias de texturas, podemos computar las ecuaciones necesarias sobre el cuadrado.

La forma en la que funciona la simulación es que cada píxel del cuadrado será considerado como una celda de una matriz bidimensional. La simulación de fluidos, según el modelo Euleriano, asigna a cada punto del espacio un valor. En este caso, cada punto está modelado por un píxel. Al aplicar una textura sobre los puntos, el procesador de fragmentos aplicará los algoritmos necesarios para encontrar el campo de vectores de velocidad asociados a cada punto. Así, durante cada paso de tiempo tenemos en cada celda un vector de velocidad asignado.

Una vez que se tiene el campo de vectores, se pueden usar colores para simular la densidad y movimiento de sustancias en el fluido.

Por lo tanto, el orden de la implementación es el siguiente:

- Crear un cuadrado que abarque la ventana de visualización.
- Asignarle una textura para que llame al procesador de fragmentos.
- Con cada asignación, se ejecutan los bloques asociados a las ecuaciones de advección, viscosidad, entre otros.
- Se suman las ecuaciones según el algoritmo dado por la descomposición de Navier-Stokes para obtener el campo de vectores de velocidad.
- Incluir variables de color y densidad para obtener una visualización de la dinámica del fluido.

A continuación se muestra la forma en que se adaptan las ecuaciones obtenidas de descomponer Navier-Stokes a una implementación en GPU. Cada ecuación está asociada a un bloque de código que se ejecutará cada vez que se asigne una textura al cuadrado que se crea en el ciclo principal.

V.2 Algoritmo de Advección

Como se definió anteriormente, la advección es la velocidad en la que el fluido transporte objetos y a sí mismo. Esto se puede representar como si cada partícula fluida dentro del fluido se mueve cierta distancia según la magnitud y dirección de la velocidad. Para saber la distancia en la que se mueve se pueden usar dos métodos para resolver ecuaciones diferenciales, implícita o explícita. La explícita es el método de Euler,

$$p(t+\delta t) = p(t) + u(t)\delta t$$

Ecuación 12.

donde p es la posición, t es tiempo y u es la advección. Un problema de la integración explícita es que no es estable para pasos de tiempo grandes, y causan problemas de borde cuando $u(t)\delta t$ es mayor que el tamaño de una celda. Sin embargo, el mayor problema es un choque con el paradigma de programación en la GPU. Los procesadores de fragmentos no pueden cambiar la celda en la cual se ejecutan; esto imposibilita el mover la celda a la cual se está ejecutando a otro lugar, siguiendo la trayectoria de la partícula. Por lo tanto, no se podría implementar en una GPU.

La integración implícita soluciona el problema anterior. En vez de computar hacia donde se dirigen las partículas según el siguiente paso de tiempo, se lee la trayectoria de la partícula fluida en su posición anterior, y se copia cuánto de la partícula fluida se encontraría en la posición actual. Esto se puede definir como

$$c(p,t+\delta t) = c(p-u(p,t)\delta t,t)$$

Ecuación 13.

donde c es la cantidad de partícula fluida, p es la posición, u la advección y t el tiempo. En definitiva, se tiene que la cantidad de partícula fluida en cada celda se define como la cantidad que se movió del instante de tiempo anterior. La ecuación 13. es la base para la implementación en la GPU de la advección.

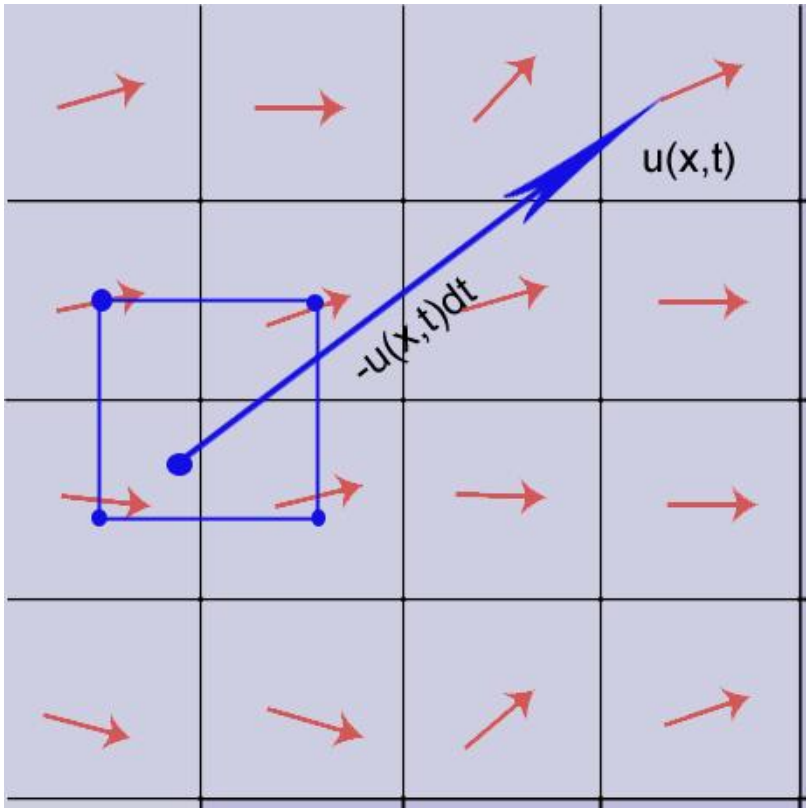


Figura 5

En la Figura 5 se muestra una representación de un campo de vectores. Para obtener la cantidad de partícula fluida en una celda, se usan las cantidades que se obtienen del instante de tiempo anterior. Se usa interpolación bilineal de las cuatro celdas más cercanas a la posición anterior para una mayor precisión en los resultados.

Entonces, usando interpolación implícita, el bloque de código asociado a la advección se programa en el procesador de fragmentos de la siguiente forma, en seudo código:

```
Advección ( coordenadas c , tiempo t, velocidad u, cantidad q )
{
        Posición = c – t * VelocidadCelda (u ,c )
        Nueva Posición = Interpolación ( q, posición)
}
```

Dado que la velocidad u y la cantidad q tienen que ser pasadas al procesador de fragmentos mediante texturas, se necesita de funciones que indiquen en qué celda o coordenada se calcula la posición anterior, y se usa VelocidadCelda. Luego para calcular la nueva posición se usa interpolación bilineal entre la cantidad q y la posición. Es de notar que la cantidad q es también una textura con todas las cantidades asociadas a cada píxel, por lo que se necesita la posición anterior para poder interpolar con las celdas más cercanas.

V.3 Algoritmo de difusión viscosa

La ecuación diferencial parcial asociada a la difusión viscosa es

$$\frac{\delta u}{\delta t} = v\nabla^2 u$$

Ecuación 14.

Dado que anteriormente definimos que usar integraciones explícitas conllevan a problemas de estabilidad e implementación, usamos la integración implícita.

$$(\mathrm{I} - v\delta t\nabla^2)u(x, t+\delta t) = u(x, t)$$

Ecuación 15.

En la Ecuación 15, tenemos a I que es matriz de identidad, y ∇^2 es el operador Laplaciano. Esto representa una ecuación de Poisson para la velocidad. Para resolver las ecuaciones de Poisson existen varios métodos, en este caso se usará la iteración de Jacobi por su facilidad de implementación. Las iteraciones de Jacobi convergen a una solución más lentamente que otros métodos, pero la arquitectura de las GPU permiten realizar varias iteraciones con poco costo computacional.

El seudo código que muestra la función que realiza una iteración de Jacobi es la siguiente.

Jacobi (coordenadas c , alfa, beta, vector x, vector b)

{

izquierda = Vector(x, c – (1,0))

derecha = Vector(x, c + (1,0))

abajo = Vector(x, c – (0,1))

arriba = Vector(x, c + (0,1))

consb = Vector (b, c)

*resultado = (derecha + izquierda + abajo + arriba + alfa * consb) * beta*

}

Según la iteración de Jacobi, en este caso alfa sería $\frac{(\delta x)^2}{v\delta t}$, y beta sería $\frac{1}{(4+\frac{(\delta x)^2}{v\delta t})}$. Los vectores b y x representan las texturas donde se guardan las velocidades, y la función Vector(a,b) entrega el vector de la textura a en la posición b. La función de Jacobi se puede iterar varias veces, según el nivel de error que se requiera.

V.4 Algoritmo de aplicación de fuerza

Para calcular el uso de fuerzas externas se requiere sólo de una ecuación que nos diga dónde se realiza la fuerza y la magnitud de ésta. Si asumimos que una aplicación de fuerza entrega cierta cantidad de un elemento dentro del fluido, por ejemplo color, se puede definir esta cantidad como

$$C = F\delta t \exp\left[\frac{(x - x_p)^2 + (y - y_p)^2}{r}\right]$$

Ecuación 16.

La Ecuación 16. tenemos a F como la fuerza realizada, que puede ser definida según el mecanismo que realiza esta fuerza. Por ejemplo, el tiempo en que se tiene presionada una tecla. (x, y) es la posición de la celda donde se calcula la cantidad, y (x_p, y_p) es la posición en la que se realizó la fuerza.

V.5 Algoritmo de Proyección

Como fue mencionado durante el desarrollo de descomposición de las ecuaciones de Navier-Stokes, la proyección consiste en resolver la ecuación de Poisson para la presión y restar el gradiente de la presión al campo de vectores de velocidad. Este es el algoritmo de GPU que toma más pasos, por lo que consiste en varios bloques que requieren de varias texturas para ser computadas.

Primero, se ejecuta un bloque que calcula la divergencia del campo de vectores w de la Ecuación 6. La divergencia se calcula simplemente como

$$\nabla \cdot W = \frac{a_{i+1,j} - a_{i-1,j}}{2\delta x} + \frac{b_{i+1,j} - b_{i-1,j}}{2\delta y}$$

Ecuación 17.

usando diferencias finitas. Esto nos da, cuando se aplica a cada celda, un campo de vectores que es usado como parámetro para calcular la iteración de Jacobi para la ecuación de Poisson de la presión.

Según la Ecuación 8., $\nabla \cdot W$ sería el parámetro b de la ecuación Ax = b, donde Ax sería $\nabla^2 p$. Por lo tanto, para poder ejecutar la iteración de Jacobi guardamos el resultado de la Ecuación 17. en una textura auxiliar para poder ser

utilizada por la función de Jacobi. A diferencia de la función asociada a la difusión viscosa, en la presión el parámetro alfa es $-(\delta x)^2$ y beta es $\frac{1}{4}$.

Al computar iteraciones de Jacobi a la ecuación de Poisson de la presión, obtenemos un valor para la presión con un error que depende de cuántas veces iteramos. Entonces, se crea otro bloque que calcula el gradiente de la presión, usando los valores que se han calculado como parámetro. El gradiente de la presión es simple de implementar ya que sigue la fórmula :

$$\nabla p = \left(\frac{a_{i+1,j} - a_{i-1,j}}{2\delta x}, \frac{b_{i+1,j} - b_{i-1,j}}{2\delta y}\right)$$

Ecuación 18.

Con esto se ha obtenido el campo divergente w y el gradiente de presión p. Según Helmholtz-Hodge en la ecuación 6, estos dos parámetros nos dan el campo de vectores de velocidad no divergente u, que es lo que necesitamos para simular la dinámica de fluidos.

VI. Integración y Resultados

Utilizando los conceptos y desarrollos presentados a través de esta memoria, ha sido posible implementar una visualización de fluidos basado en la ejecución de algoritmos en la GPU. Como un fluido no se puede visualizar sin otros objetos o elementos interactuando con éste, se ha incluido un parámetro de color que se va distribuyendo según las fuerzas externas y las propiedades del fluido. Debido a la naturaleza dinámica de los fluidos, no se puede apreciar mediante imágenes la gama de movimientos que pueden efectuar.

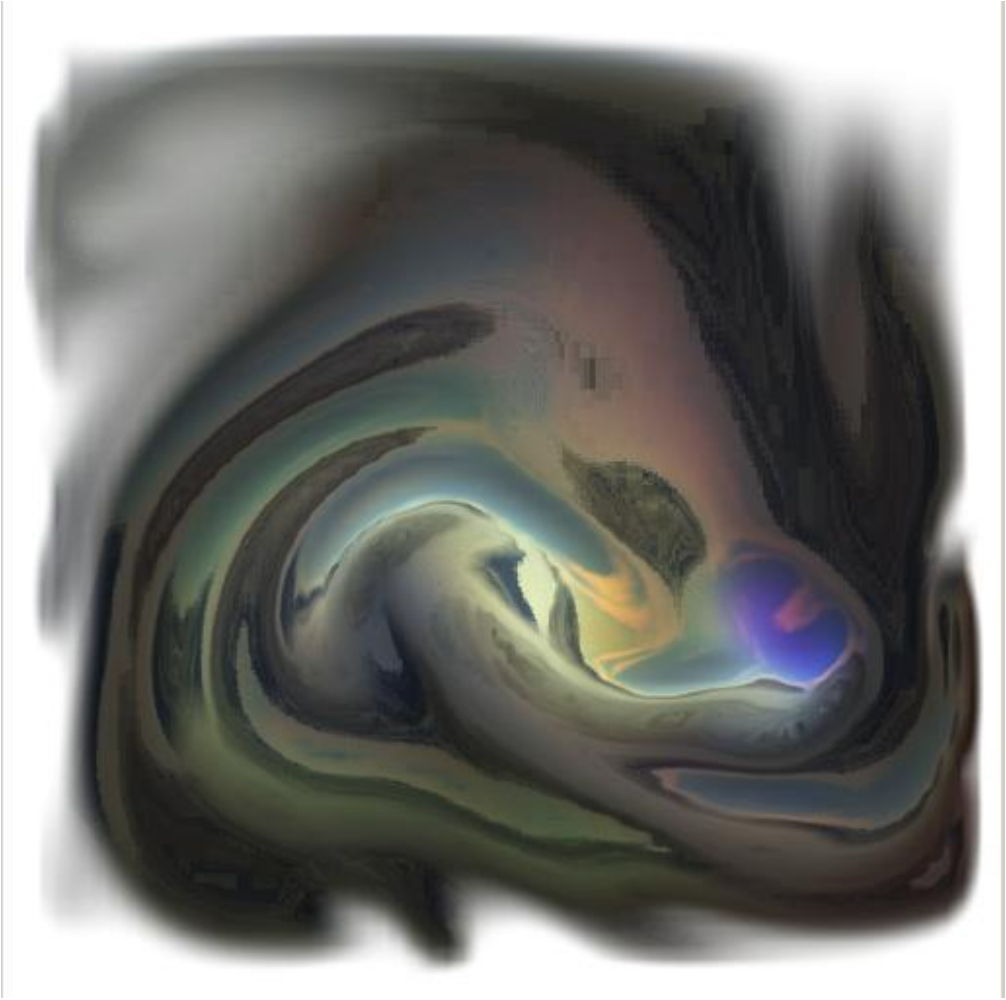


Figura 6.

En la Figura 6 se muestra la visualización de un fluido. Usando fuerzas externas, que en este caso fue el movimiento del mouse, se insertan varios colores que se mueven alrededor del fluido según las ecuaciones que lo rigen. Cambiando la viscosidad, se obtienen imágenes como la Figura 7.

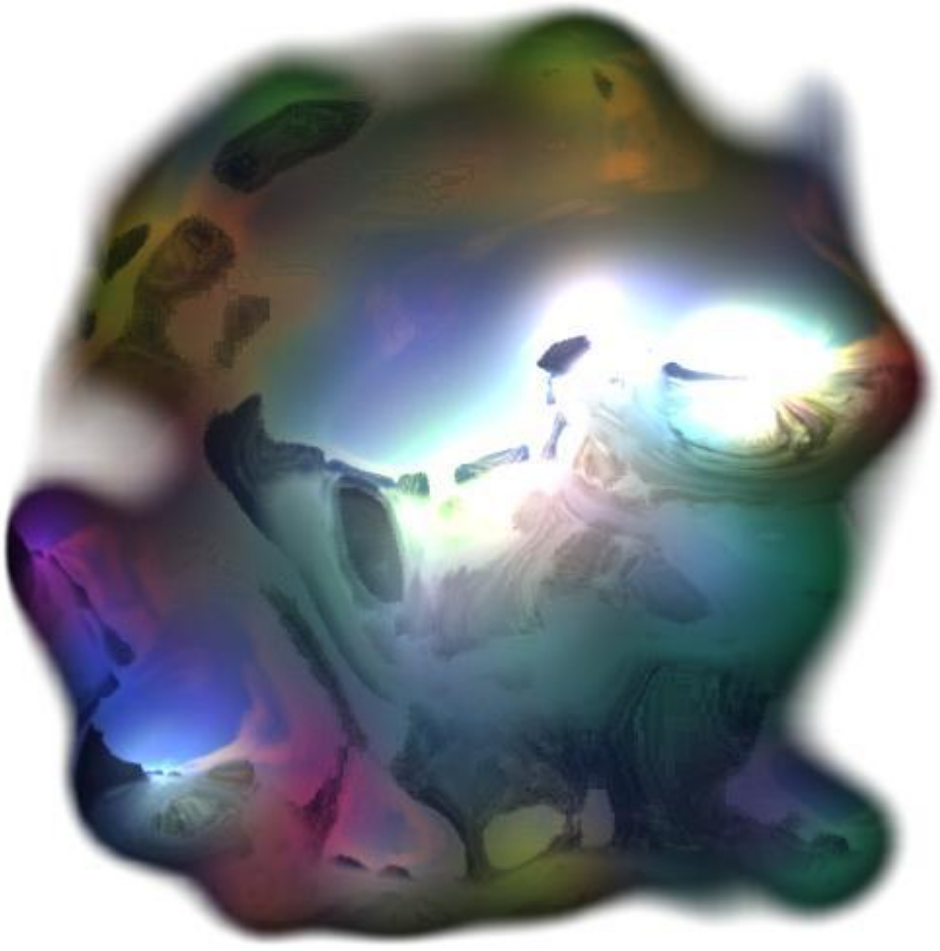


Figura 7.

Se puede apreciar que se distingue la poca movilidad de los colores alrededor del fluido comparado con la Figura 6, donde los colores atraviesan el fluido y se mezclan con otros más lejanos.

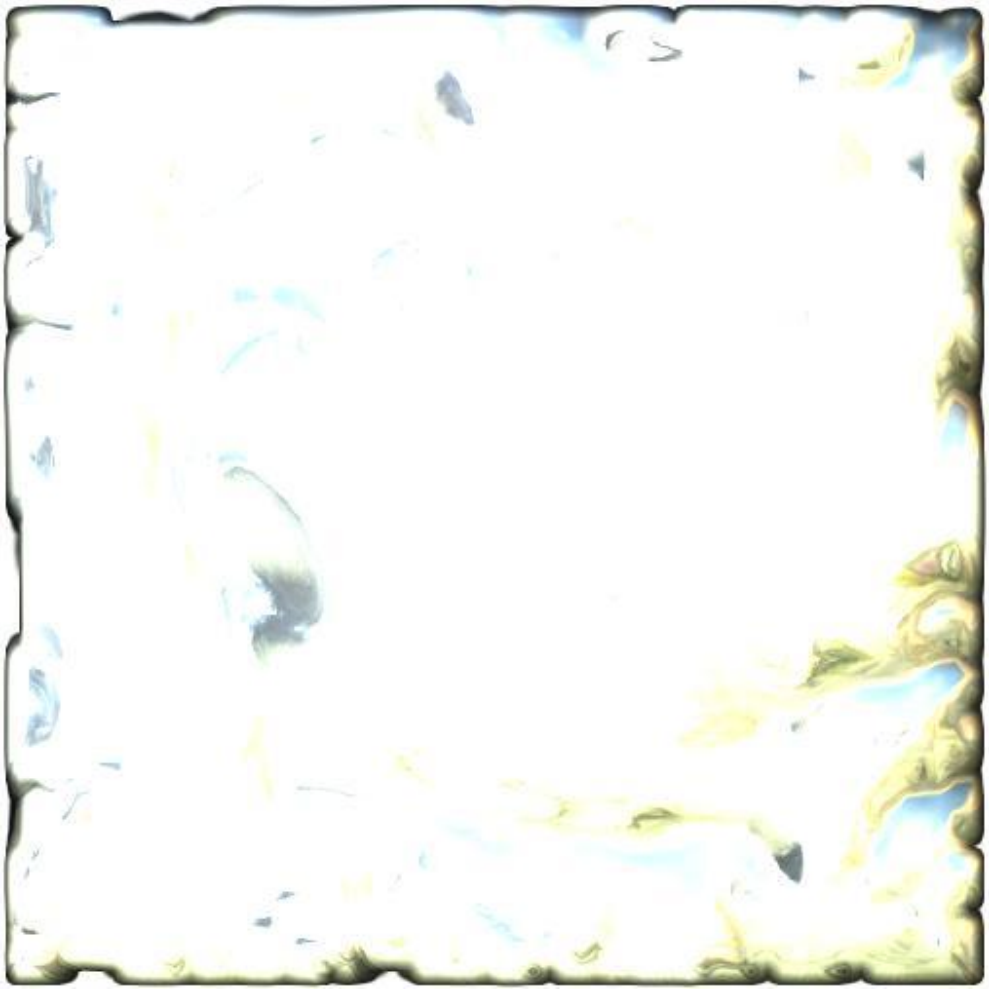


Figura 8.

En la Figura 8 se muestra lo que pasa cuando se inserta una gran cantidad de color a un fluido, usando una fuerza que los empuja hacia los bordes. Para resolver cualquier problema relaciona con ecuaciones diferenciales definidos en un dominio finito, se necesitan condiciones de inicio y de borde para que se comporten bien. En esta implementación las condiciones de inicio son cero velocidad y presión. Las condiciones de borde son necesarias para resolver las ecuaciones de velocidad y presión; como la simulación está limitada por los

bordes de la ventana de visualización se definió que en éstas la velocidad sería cero.

VII. Conclusiones y Extensiones

Se ha presentado en esta memoria el estudio y desarrollo de un visualizador de dinámica de fluidos. Gracias al avance realizado por distintos investigadores en el área existen varias implementaciones, fuentes y mejoras relacionadas al tema. Usando los pasos de algoritmos diseñados para simplificar la tarea de convertir la matemática usada en la mecánica de fluidos a una implementación computacional, se ha podido satisfacer los objetivos de esta memoria.

A través del trabajo realizado se ha comprobado que es posible ejecutar tareas pensadas para la computación general en hardware específico, que en este caso son las tarjetas gráficas. Éstas han sido diseñadas para computar píxeles y objetos tridimensionales en una pantalla, pero el avance de la tecnología ha abierto las puertas para que se utilicen para otros fines. Siendo la arquitectura de las GPU altamente paralela, se ha podido implementar algoritmos que aprovechan esta ventaja y crean programas que funcionan de forma eficiente y en muchas ocasiones con un mayor rendimiento.

La dinámica de fluidos es un tema que es interesente para varias áreas científicas, por lo que la implementación de programas que faciliten su simulación es de gran ayuda. Además, el desarrollo de algoritmos rápidos en la GPU estimulan el crecimiento del realismo en aplicaciones que son tanto científicas como de recreación, al simular efectos especiales en juegos.

Uno de los problemas de esta simulación es que no es rigurosa en cuanto a la resolución de las ecuaciones de Navier-Stokes. Usando iteraciones de Jacobi es simple y da resultados relativamente buenos, pero introduce errores que dependen del número de iteraciones. Se puede implementar otros métodos

de resolución de matrices de ecuaciones, pero son más complicadas y requieren de mayor poder de cómputo.

Dado que la gran parte de los algoritmos son ejecutados en la GPU, se libran recursos de la CPU para otras tareas. Esto es un gran incentivo para la creación de más aplicaciones de computación general en las GPU, ya que mientras más avance el desarrollo de las tarjetas gráficas se obtiene un nicho de recursos poco explotado por los computadores. En general, la única forma en la que una tarjeta gráfica utiliza gran parte de sus capacidades es cuando ejecuta programas de visualización tridimensional. Como una gran parte de los usuarios no usan tales programas o juegos, se puede aprovechar el poder de cómputo de las tarjetas gráficas en otro tipo de aplicaciones.

El trabajo presentado es sólo una base en la que múltiples tipos de simulaciones se pueden crear. Se puede extender para simular fluidos más específicos, tales como el agua y las nubes. Lo que se necesita es variables de inicio y de borde, como también de color y densidad, que se ajusten al fenómeno o efecto deseado. Para fluidos que tienen un bajo nivel de viscosidad, tales como el humo, se necesita implementar la vorticidad. La vorticidad es un factor que realiza flujos rotacionales que agrega una mayor gama de movimientos a un fluido.

Otra extensión es realizar la simulación en tres dimensiones. La base matemática presentada aquí es suficiente para tal propósito. La dificultad se concentra en la visualización. En tres dimensiones, se requiere de otra forma de representar elementos en el flujo. Una forma de hacerlo sería con texturas tridimensionales, que tiene la desventaja de no ser compatible con varias tarjetas gráficas. Otra solución sería el uso de sólidos, con algoritmos tales como el de cubos marchantes para simular un fluido espeso.

El primer objetivo de esta memoria era el desarrollo de algoritmos para el refinamiento de mallas geométricas. Se estudió las posibilidades y se exploraron posibles soluciones, pero no se encontró una que justificara los costos de una implementación híbrida entre una CPU y una GPU. Sólo existen implementaciones híbridas debido a una gran limitación de las GPU, que no tienen en su arquitectura la posibilidad de alterar objetos con inserciones de puntos o removerlos. Las futuras generaciones de tarjetas gráficas, en especial la serie Geforce 8 de NVIDIA, tendrán esta funcionalidad. Se espera que cuando estas tarjetas sean adquiribles, se pueda estudiar más a fondo la posibilidad de realizar refinamientos en la GPU, con las ventajas que esto conlleva.

VIII. Referencias

A continuación se presentan referencias a material tanto usado para este informe, como para su uso durante la memoria.

[01] Randima Fernando, "*GPU Gems 2, Programming techniques for High-Performance Graphics and General-Purpose Computation*", NVIDIA, Addison-Wesley

[02] Ian Buck, Stanford University, "*Data Parallel Computing on General Hardware*", http://graphics.stanford.edu/projects/brookgpu/GH03-Brook.ppt

[03] Randima Fernando, *"GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics* ", NVIDIA, Addison-Wesley

[04] Naga K. Govindaraju, Nikunj Raghuvanshi, Michael Henson, Dinesh Manocha, " *Fast and Approximate Stream Mining of Quantiles and Frequencies Using Graphics Processors*", http://gamma.cs.unc.edu/STREAMING/

[05] Naga K. Govindaraju, Nikunj Raghuvanshi, Dinesh Manocha, Wei Wang, Ming Lin, "*Fast Database Operations using Graphics Processors*", http://gamma.cs.unc.edu/DB/

[06] NVIDIA, "*GPU Programming Guide* ", http://developer.nvidia.com/object/gpu_programming_guide.html

[07] M.C Rivara, "*New Longest-Edge Algorithm for the Refinement and/or Improvement of Unstructured Triangulations*", International Journal for Numerical Methods in Engineering, Vol. 40, 1997

[09] "*CG Toolkit*", lenguaje de programación para GPU http://developer.nvidia.com/object/cg_toolkit.html

[09] *"Bitonic Sort"* http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm

[10] Martin Rumpf y Robert Strzodka, "*Graphics Processor Units: New Prospects for Parallel Computing*", University of Bonn, Institute for Numerical Simulation, Wegelerstr. 6, 53115 Bonn, Germany

[11] STAM J., "*Stable Fuids*", *SIGGRAPH* (1999), pág. 121-128.

[12] Michael Guthe Akos Balazs , "*GPU-based trimming and tessellation of NURBS and T-Spline surfaces*", Universitat Bonn, ACM Transactions on Graphics, Volume 24 , Issue 3 (July 2005), Proceedings of ACM SIGGRAPH 2005

[13] Guido Ranzuglia, Paolo Cignoni, Fabio Ganovelli, Roberto Scopigno, "*Implementing mesh-based approaches for deformable objects on GPU*", http://vcg.isti.cnr.it/Publications/2006/RCGS06/gpgpu.pdf

[14] Simon Clavet, Philippe Beaudoin, and Pierre Poulin, *"Particle-based Viscoelastic Fluid Simulation*", Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation, pág. 219 - 228 , año 2005

[15] Martin Rumpf1 and Robert Strzodka, "*Graphics Processor Units: New Prospects for Parallel Computing*", Numerical Solution of Partial Differential Equations on Parallel Computers, volume 51 of Lecture Notes in Computational Science and Engineering. Springer-Verlag, 2005.

[16] Reeves W. T., "*Particle systems a technique for modelling a class of fuzzy objects*". In *SIGGRAPH* (1983), pp. 359-376.

[17] Lucy L., "*A numerical approach to the testing of the fission hypothesis*", *Astronomical Journal 82* (1977), 1013. Referenciado a partir de [14].

[18] Premoze S., Tasdizen T., Bigler J., Lefohn A., Whitaker R. T., "*Particle-based simulation of Fluids*". *Computer Graphics Forum 22*, 3 (2003), pp. 401-410.

[19] Foster N., Metaxas D., "*Realistic animation of liquids*". *Graphical Models and Image Processing 58*, 5 (1996), pp. 471-483.

[20] Carlson M., Mucha P. J., Horn R. B. V., Turk G., "*Melting and Flowing*", In *SIGGRAPH/Eurographics Symposium on Computer Animation* (2002), pp. 167-174.

[21] Goktekin T. G., Bargteil A. W., O'brienJ. F.: "*A method for animating viscoelastic Fluids"*. In *SIGGRAPH* (2004), pp. 463-468.

IX. Apéndices

A continuación se muestran partes del código relacionados con CG, que procesa las funciones necesarias para calcular las ecuaciones de Navier-Stokes.

```
// La función de advección

void adveccion(float2 coords : WPOS,out float4 OUT : COLOR, uniform
float t,uniform float disipacion, uniform float rdx, uniform
samplerRECT u,uniform samplerRECT x)

{

float2 pos = coords - t * rdx * f2texRECT(u, coords);

OUT = disipacion * bilinear(x, pos);
}

//la función de interpolación bilinear en una textura

float4 bilinear(samplerRECT tex, float2 s)
{
  float4 st;
  st.xy = floor(s - 0.5) + 0.5;
  st.zw = st.xy + 1;

  float2 t = s - st.xy; //interpolating factors

  float4 tex11 = f4texRECT(tex, st.xy);
  float4 tex21 = f4texRECT(tex, st.zy);
  float4 tex12 = f4texRECT(tex, st.xw);
  float4 tex22 = f4texRECT(tex, st.zw);

  // interpolando
  return lerp(lerp(tex11, tex21, t.x), lerp(tex12, tex22, t.x), t.y);
}

// función de divergencia

void divergencia(half2 coords  : WPOS,out half4  OUT : COLOR,
uniform half halfrdx,uniform samplerRECT w)
{
  half4 vL, vR, vB, vT;
  vecinos(w, coords, vL, vR, vB, vT);

  OUT = halfrdx * (vR.x - vL.x + vT.y - vB.y);
}
```

```
//la función usada en divergencia para encontrar vecinos

void vecinos(samplerRECT tex, half2 s, out half4 izq,out half4 der,out
half4 abajo,out half4 arriba)
{
  izq   = h4texRECT(tex, s - half2(1, 0));
  der  = h4texRECT(tex, s + half2(1, 0));
  abajo = h4texRECT(tex, s - half2(0, 1));
  arriba = h4texRECT(tex, s + half2(0, 1));
}

// Esta función realiza un paso de jacobi en una ecuación de poisson

void jacobi(half2 coords : WPOS,out half4 OUT  : COLOR,uniform half
alfa, uniform half rBeta, uniform samplerRECT x, uniform samplerRECT b)
{

  half xL, xR, xB, xT;
  jvecinos(x, coords, xL, xR, xB, xT);

  half bC = h1texRECT(b, coords);

  OUT.x = (xL + xR + xB + xT + alfa * bC) * rBeta;
}

// función usada en jacobi para dar vecinos

void jvecinos(samplerRECT tex, half2 s,out half izq,out half der,out
half abajo,out half arriba)
{
  izq  = h1texRECT(tex, s - half2(1, 0));
  der  = h1texRECT(tex, s + half2(1, 0));
  abajo = h1texRECT(tex, s - half2(0, 1));
  arriba    = h1texRECT(tex, s + half2(0, 1));
}

// función de gradiente

void gradiente(half2 coords  : WPOS,out half4 OUT    : COLOR,
uniform half halfrdx, uniform samplerRECT p,uniform samplerRECT w)
{
  half pL, pR, pB, pT;

  jvecinos(p, coords, pL, pR, pB, pT);

  half2 grad = half2(pR - pL, pT - pB) * halfrdx;

  OUT = h4texRECT(w, coords);
  OUT.xy -= grad;
}
```

```
// función de verticidad (interpolación)

void vorticidad(half2 coords : WPOS,out half  OUT : COLOR,uniform half
halfrdx, uniform samplerRECT u)
{
  half4 uL, uR, uB, uT;
  vecinos(u, coords, uL, uR, uB, uT);

  OUT = halfrdx * ((uR.y - uL.y) - (uT.x - uB.x));
}


// función de la fuerza de la verticidad

void vortfuerza(half2       coords : WPOS,out half2  OUT :
COLOR,uniform half halfrdx,  uniform half2  dxscale,  uniform half
timestep,uniform samplerRECT vort,     uniform samplerRECT u)
{
  half vL, vR, vB, vT, vC;
  jvecinos(vort, coords, vL, vR, vB, vT);

  vC = h1texRECT(vort, coords);

  half2 fuerza = halfrdx * half2(abs(vT) - abs(vB), abs(vR) - abs(vL));

  static const half EPSILON = 2.4414e-4;
  // esto es 2^-12, para     normalizar
  half magSqr = max(EPSILON, dot(fuerza, fuerza));
  fuerza = fuerza * rsqrt(magSqr);

  fuerza *= dxscale * vC * half2(1, -1);

  OUT = h2texRECT(u, coords);

  OUT += timestep * force;
}
```