



UNIVERSIDAD SIMÓN BOLÍVAR
DECANATO DE ESTUDIOS PROFESIONALES
COORDINACIÓN DE INGENIERÍA DE COMPUTACIÓN

Simulación de un sistema masa-resorte en el GPU

Por:
Juan G. Campa

PROYECTO DE GRADO

Presentado ante la Ilustre Universidad Simón Bolívar
como requisito parcial para optar al título de
Ingeniero de Computación

Sartenejas, Abril de 2010



UNIVERSIDAD SIMÓN BOLÍVAR
DECANATO DE ESTUDIOS PROFESIONALES
COORDINACIÓN DE INGENIERÍA DE COMPUTACIÓN

Simulación de un sistema masa-resorte en el GPU

Por:
Juan G. Campa

Realizado con la asesoría de:
Ivette Carolina Martínez

PROYECTO DE GRADO

Presentado ante la Ilustre Universidad Simón Bolívar
como requisito parcial para optar al título de
Ingeniero de Computación

Sartenejas, Abril de 2010



UNIVERSIDAD SIMÓN BOLÍVAR
DECANATO DE ESTUDIOS PROFESIONALES
COORDINACIÓN DE INGENIERÍA DE COMPUTACIÓN

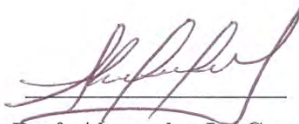
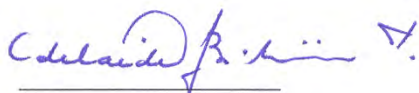
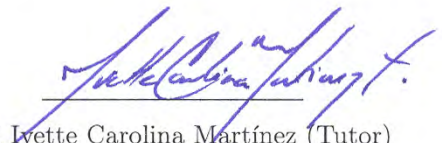
ACTA FINAL PROYECTO DE GRADO

Simulación de un sistema masa-resorte en el GPU

Presentado por:

Juan G. Campa

Este proyecto de grado ha sido aprobado por el siguiente jurado examinador:


Prof. Alexandra La Cruz
Prof. Adelaide Bianchini
Prof. Ivette Carolina Martínez (Tutor)

Sartenejas, Abril de 2010



Universidad Simón Bolívar
Decanato Estudios de Profesionales
Coordinación Ingeniería de la Computación

**MENCION EN PROYECTO DE GRADO,
CARRERA INGENIERÍA DE LA COMPUTACIÓN**

Nosotros, Profesores Jurados y Tutor le otorgamos la **MENCIÓN
EXCEPCIONALMENTE BUENO** al estudiante:

CAMPA MOERBEECK, JUAN GUILLERMO CARNET **04-36790**

En el Proyecto de Grado titulado: **“SIMULACIÓN DE UN SISTEMA
MASA-RESORTE EN EL GPU”**

Por las siguientes razones:

1. El estudiante demostró dominio de los temas requeridos para el desarrollo de trabajo.
2. El estudiante propuso e implementó una modificación al modelo masa-resorte propuesto por Thomas Jacobsen que permite: (1) Ajustar el tiempo que tardan los resortes en regresar a su longitud de descanso. Esto le otorga un mayor nivel de realismo al modelo; permitiendo que pueda ser usado en animaciones. (2) Configurar, usando sólo un parámetro, las propiedades físicas del material del cuerpo para que se ajusten a una rigidez determinada.
3. Consideramos que los aportes y resultados de este trabajo pueden ser publicados en un foro especializado internacional.

Conforme:

Jurado: Prof. Adelaide Bianchini

Jurado: Prof. Alexandra La Cruz

Tutor: Prof. Ivette C. Martínez

Fecha: 12-05/2010
lcm/icmt

RESUMEN

Con la llegada de las tarjetas gráficas programables se marcó el comienzo de la computación de propósito general en GPUs, los cuales son procesadores con una gran capacidad de procesamiento ideales para problemas que presentan un alto nivel de paralelismo de datos. Aplicaciones que han sido satisfactoriamente implementadas en GPU muestran incrementos de velocidad que oscilan desde 2 veces más rápido hasta 1000 veces en algunos casos. La simulación de cuerpos suaves ha sido tradicionalmente un problema de computación de alto rendimiento por la cantidad de datos del problema: muchas partículas independientes constituyen un cuerpo. El modelo para simular cuerpos suaves es muchas veces el del sistema masa-resorte, donde las partículas que componen un cuerpo están unidas por resortes que mantienen la integridad del mismo. La actualización de cada uno de los componentes de un sistema masa-resorte es un problema que presenta un alto nivel de paralelismo de datos: muchos elementos del sistema se actualizan independientemente con una misma rutina.

Para atacar este problema se presenta en este trabajo una implementación de un sistema masa-resorte basada en la publicación de Thomas Jakobsen (2001) pero que se ejecuta completamente en el GPU, hacer esto no solo deja el CPU disponible para otras tareas, sino que presenta importantes mejoras en velocidad. La implementación se creó utilizando el *framework* CUDA de Nvidia con el cual fue necesario considerar la utilización eficiente de memoria y la sincronización entre hilos.

En este trabajo, se agregó al modelo una variable que llamamos S , la cual permite ajustar el tiempo que tardan los resortes en volver a su longitud de descanso, con esto se logra ajustar la suavidad de los cuerpos simulados, lo que añade la posibilidad de configurar las propiedades físicas del material del cuerpo para que se ajusten a una rigidez específica y alcanzar un mayor nivel de realismo.

Para mostrar los resultados, se construyeron cuatro modelos 3D utilizando 3dsMax, los cuales se utilizaron para generar automáticamente instancias de sistemas masa-resorte con diferentes características, se ejecutaron simulaciones con la implementación desarrollada, y se midieron los tiempos de ejecución para compararlos con la implementación de CPU, las mejoras en velocidad de la simulación variaron desde 4 hasta más de 20 veces mejor en algunos casos, en comparación con la versión implementada en CPU.

Palabras Claves: GPGPU, resortes, masa-resorte, CUDA, GPU.

DEDICATORIA

A mi mamá y a mi papá, por apoyarme siempre incondicionalmente.

A mi hermano Mario, por hacer que me interesara por la computación.

A mi hermana Liliana, que me apoya en todo lo que hago.

Y a mi hermana menor Cristina, que será por siempre mi inspiración.



ÍNDICE GENERAL

Índice de cuadros	IX
Índice de figuras	X
Introducción	1
1 Marco teórico	4
1.1 El Sistema masa-resorte	4
1.1.1 Simulación del SMR	5
1.2 Fundamentos Físicos	5
1.2.1 Sistema de partículas	5
1.2.2 Resortes	6
1.3 Métodos de integración	7
1.3.1 Método Euler	7
1.3.2 Método Vertlet	7
1.4 Simulación de resortes	8
1.5 El GPU	10
1.5.1 Contraste entre GPU y CPU	10
1.5.2 Arquitectura del GPU	11
1.5.3 Tecnologías para desarrollar programas GPGPU	14
1.6 Arquitectura de CUDA	15
1.6.1 Jerarquía de memoria en CUDA	16
1.6.2 <i>Kernels</i>	18
1.6.3 Llamadas a <i>kernels</i>	18
1.6.4 Patrón de acceso a memoria	20
2 Diseño de la implementación	21
2.1 Herramientas de trabajo	22
2.2 El paralelismo de datos en el SMR	22
2.2.1 Región crítica de memoria	23
2.2.2 Paralelización del algoritmo de resortes	24
2.3 Implementación de la simulación en CUDA	25
2.3.1 Primer enfoque: Múltiples llamadas	25
2.3.2 Segundo enfoque: Iteraciones dentro del kernel	27
2.4 Algoritmo de actualización de partículas	28
2.5 Transferencia de datos e interoperabilidad con DirectX	28
2.6 Utilización del GPU	29
3 Experimentos y resultados	31
3.1 Sistema utilizado	31
3.2 Modelos de prueba	32

3.3	Medición de tiempo de ejecución	33
3.4	Factor S y dureza del cuerpo.	34
	Conclusiones y recomendaciones	36
	Conclusiones	36
	Recomendaciones	37
	Bibliografía	39
A	Listados de código	42
A.1	Algoritmo de partición de resortes en subconjuntos paralelizables	42
A.2	Algoritmo del tamaño del grid y llamada al kernel. Primer Enfoque.	43
A.3	Algoritmo del tamaño del grid y llamada al kernel. Segundo Enfoque.	43
A.4	Kernel de actualización de resortes. Primer Enfoque.	44
A.5	Kernel de actualización de resortes. Segundo Enfoque.	45
A.6	Código MaxScript para exportar modelos con información física de 3dsMax 2009	45

ÍNDICE DE CUADROS

1.1	Ejemplo de anchos de banda disponibles en distintos componentes del sistema [1] . . .	12
2.1	Ejemplo de planificación de los hilos que ejecutan el kernel de actualización de resortes .	24
2.2	Tamaños de bloque para diferentes cantidades de resortes (Primer enfoque)	26
3.1	Características del sistema utilizado para probar	31
3.2	Características del sistema CUDA utilizado.	32
3.3	Modelos de prueba	33
3.4	Mediciones de tiempos de ejecución de los	33

ÍNDICE DE FIGURAS

1.1	Comportamiento de un resorte para distintos valores de la constante S . Se observa que mientras S se acerca a 0, el resorte es menos rígido (tarda más en regresar a su longitud de descanso).	9
1.2	Diagrama de flujo del acercamiento tradicional al GPGPU utilizando el paradigma de <i>Shaders</i>	13
1.3	Ejemplo de una rejilla de tamaño 3×2 con bloques de tamaño 4×3 . Se puede ver como se organizan los hilos en la misma [21].	17
2.1	Representación gráfica de la instancia genérica de un SMR utilizado para ilustrar ejemplos, la partícula A está conectado por 3 resortes, el resto puede tener más conexiones.	21
2.2	Ejemplo del problema de dependencia entre partículas. A y B , aunque no estén conectadas directamente, son interdependientes.	22
2.3	Conflicto entre resortes que puede ocasionar una condición de carrera, r_1 , r_2 y r_3 no pueden actualizarse en paralelo.	23
2.4	Actuación de los hilos con múltiples llamadas sobre una partición de R de ejemplo. Los cuadros oscuros representan los hilos ejecutándose en el GPU y los cuadros claros representan los subconjuntos de resortes en los que estos actúan.	26
2.5	Actuación de los hilos con iteraciones dentro del <i>kernel</i> sobre una partición de R de ejemplo. En este caso solo se crean hilos una sola vez y estos actúan a través de los distintos subconjuntos de resortes.	27
2.6	Iteraciones que hace nuestro método para una determinada partición ejemplo de R	30
3.1	Modelos 3D utilizados para generar los SMR de prueba. (a) Plano, (b) GeoEsfera, (c) Cochino, (d) Teapot.	34
3.2	GeoEsfera simulada con distintos valores de S . La esfera superior no tiene transformaciones, las de abajo están colisionando con el suelo lo que las hace deformarse.	35

LISTA DE ABREVIATURAS

GPU	Graphics Processing Unit (Unidad procesadora de gráficos)
CPU	Central Processing Unit (Unidad de procesamiento central)
GPGPU	General Purpose GPU (GPU de propósito general)
SMAR	Sistema Masa-Amortiguador-Resorte
RAM	Random Access Memory (Memoria de acceso aleatorio)
VRAM	Video RAM
SIMD	Single Instruction, Multiple Data streams (Una sola instrucción, varios flujos de datos)
OOOE	Out-of-order Execution
MMX	Multimedia Extensions
SSE2	Streaming SIMD Extensions, version 2 (Extension de flujo de SIMD, versión 2)
CUDA	Compute Unified Device Architecture (Arquitectura de dispositivo de computación unificada)
HLSL	High Level Shader Language (Lenguaje de shaders de alto nivel)
GLSL	Graphics Library Shader Language (Lenguaje de shaders de Graphics Library)
OpenCL	Open Compute Library (Librería de cómputo abierta)
SM	Streaming Multiprocessor (Multiprocesadores de flujos)
SP	Scalar Processor (Procesador escalar)
1D	Una Dimensión
2D	Dos Dimensiones
3D	Tres Dimensiones
DSP	Digital Signal Processing (Procesamiento digital de señales)
PCI	Peripheral Component Interconnect (Interconexión entre componentes periféricos)
PCI-E	PCI-Express (PCI-Expreso)
FLOPS	FLoating point OPerations per Second (Operaciones de punto flotante por segundo)

INTRODUCCIÓN

Desde el 2002, con la llegada de las tarjetas gráficas programables [3], se ha visto un aumento en el nivel de realismo de video juegos comerciales. Esto viene impulsado por un aumento equivalente en la capacidad de procesamiento de las computadoras, básicamente el cumplimiento de la Ley de Moore [4] que dice que la cantidad de transistores que se puede poner en un microchip se duplica cada dos años.

Cuando se habla de realismo no solo se habla a la fidelidad gráfica, la animación es también un factor importante, poco a poco las animaciones pre-generadas se han visto desplazadas por animaciones generadas en tiempo real, donde los objetos interactúan con los elementos del ambiente y tienen un comportamiento visualmente aceptable en respuesta a decisiones arbitrarias del usuario. Para poder lograr esto a gran escala, la industria de los video juegos ha tenido que esperar por la industria de la electrónica a que desarrollara procesadores suficientemente rápidos para ejecutar las costosas rutinas de simulación. La simulación física en video juegos abarca desde cuerpos rígidos, cuerpos suaves, fluidos, hasta óptica, entre otras cosas.

Los cuerpos suaves tienden a ser computacionalmente más costosos de simular que los cuerpos rígidos, ya que el cuerpo tiene fuerzas internas que modifican su forma y movimiento. Podemos pensar en un cuerpo suave como un sistema de partículas con masa que interactúan entre sí, atrayéndose y repeliéndose (como si estuvieran conectadas por resortes) para tratar de mantener la estructura del cuerpo, como mostraremos en este trabajo. Es por esto que el estudio del Modelo Masa-Resorte-Amortiguador es esencial para la simulación de cuerpos suaves, y en general para la industria del entretenimiento interactivo. Este modelo implica la simulación de (por lo general) una gran cantidad de partículas unidas por una relativamente gran cantidad de resortes para mantener la estructura del cuerpo. Los cuerpos suaves son un elemento que no ha sido muy explotado por la industria quizá por su alto costo de procesamiento.

Por otro lado las tarjetas gráficas actuales, cuyo componente de procesamiento es el GPU (Graphics Processing Unit, Unidad de procesamiento de gráficos) son diseñadas para procesar simultáneamente una gran cantidad de píxeles en la pantalla y una gran cantidad de vértices de algún modelo geométrico, es por esto que, el GPU es una alternativa muy eficiente para ejecutar programas donde los datos toman formas análogas a una generalización de píxeles o vértices.

Se puede observar que existe un patrón de similitud entre el modelo computacional de una tarjeta gráfica, que intenta ejecutar un programa por cada uno de los millones de píxeles en la pantalla o vértices en un modelo geométrico, y el de un sistema de simulación para cuerpos suaves, que debe ejecutar la misma rutina por cada una de las partículas que componen el sistema y otra rutina por cada conexión que existe entre estas partículas. Ambos esquemas presentan lo que se conoce como paralelismo de datos, es decir, en ambos casos se necesita procesar una gran cantidad de datos con la misma rutina y más importante aún, estos datos no dependen entre sí para poder ser procesados.

El objetivo principal de este proyecto es crear un programa que se ejecute en el GPU, capaz de simular un sistema de masas interconectadas con resortes, el cual pueda ser utilizado principalmente para animar cuerpos suaves. También se investigarán las capacidades no-gráficas del GPU, es decir, su utilización para resolver problemas de propósito general, esto para hacer un aporte a esta rama de la computación que actualmente se denomina de alto rendimiento.

Convenciones

Para facilitar la lectura de este trabajo, los valores vectoriales serán representados con negrita (Ejemplo: $\mathbf{F} = m\mathbf{a}$, F y a son vectores, mientras que m es un valor escalar). También, cuando se hable de partículas, éstas estarán escritas en negrita.

CAPÍTULO 1

MARCO TEÓRICO

En este capítulo se explicarán los fundamentos teóricos en los que se basa nuestro proyecto. Se comenzará proponiendo un modelo matemático para representar el sistema masa-resorte, luego se mostrarán las ecuaciones físicas pertinentes y se hará un profundo estudio de los GPUs (*Graphics Processing Unit* o unidad de procesamiento de gráficos) que es la plataforma donde se desarrolló la implementación.

1.1 El Sistema masa-resorte

Llámesese SMR al modelo matemático definido en este trabajo para definir un Sistema Masa-Resorte. Un SMR está definido por la tupla:

$$(P, R) \tag{1.1}$$

Donde P es un conjunto de partículas y R el conjunto de resortes que las unen, es decir, los elementos de R tienen la forma

$$(\mathbf{p}_1, \mathbf{p}_2, l, k) \tag{1.2}$$

Donde $\mathbf{p}_1, \mathbf{p}_2 \in P$ y l y k representan la longitud de descanso y el coeficiente de elasticidad del resorte respectivamente.

Un SMR se puede ver efectivamente como un grafo no dirigido, donde los nodos son los representados por el conjunto de partículas P y las aristas por el conjunto de resortes R , ya se verá más adelante que utilizar esta definición ayuda a entender algunos conceptos.

Se denomina **tamaño del sistema** a la suma $|P| + |R|$ ya que este valor permite tener una idea de la cantidad de procesamiento necesario para simular un sistema dado.

1.1.1 Simulación del SMR

El problema de simular un SMR, consiste en recrear el comportamiento de un conjunto de partículas P interconectadas arbitraria o implícitamente por resortes R a lo largo de pasos de tiempo discretos. La forma de interconexión implícita es útil para casos muy específicos donde la posición relativa de cada partícula determina su conectividad con otras. La simulación de tela es uno de estos casos ya que cada partícula que la conforma está conectada implícitamente con sus vecinas y éstas están posicionadas en un rejilla establecida. Por ser un elemento relativamente común en varias industrias ya existe una variedad de trabajos sobre la simulación de tela en GPU como el trabajo de C. Zeller (*Cloth Simulation on the GPU*, simulación de tela en el GPU) [5] y el motor de física PhysX de Nvidia [6]. Aunque los métodos desarrollados en estos trabajos son efectivamente capaces de simular tela, éstos no permiten crear sistemas masa-resorte con las masas posicionadas arbitrariamente ni interconectadas arbitrariamente, sino que se posicionan igualmente espaciadas en un arreglo bidimensional y la interconexión viene dada por la posición en el arreglo. El acercamiento presentado en este trabajo toma un enfoque más general, utilizando la forma de interconexión explícita donde cada partícula puede estar conectada con cualquier otra además de permitir que las mismas estén en posiciones arbitrarias y no en una rejilla definida.

1.2 Fundamentos Físicos

En esta sección se explicará brevemente los fundamentos físicos en los cuales se basan los algoritmos que se mostrarán posteriormente.

1.2.1 Sistema de partículas

Teóricamente una partícula se comporta según las leyes newtonianas del movimiento, es decir, las partículas mantienen su estado por inercia si no son afectadas por ninguna fuerza externa, y al ser afectadas experimentan una aceleración que viene dada por la Ec (1.3) [7]:

$$\mathbf{F} = m\mathbf{a} \tag{1.3}$$

Es decir, la fuerza que se le debe aplicar a un objeto de masa m para lograr una aceleración \mathbf{a} debe ser igual a \mathbf{F} . Una vez que tenemos la aceleración (y esta permanece constante) podemos determinar la posición de la partícula \mathbf{x} luego de transcurrido un tiempo \mathbf{t} si la posición y velocidad inicial son

\mathbf{x}_0 y \mathbf{v}_0 respectivamente, con la Ec (1.4) [7]:

$$\mathbf{x} = \mathbf{x}_0 + \mathbf{v}_0 t + \frac{1}{2} \mathbf{a} t^2 \quad (1.4)$$

Este par de ecuaciones establecen el modelo teórico de la simulación de partículas que se propone en este proyecto. El algoritmo de simulación está basado en el de Jakobsen [8] y en líneas generales tiene la forma explicada en el algoritmo 1.1.

Algoritmo 1.1 Estructura general del algoritmo de simulación

```

1: para todo  $p \in P$  hacer
2:   actualizarParticula( $p$ )
3: fin para
4: para todo  $r \in R$  hacer
5:   actualizarResorte( $r$ )
6: fin para
7: DibujarSistema()

```

Primero se actualizan todas las partículas del sistema y luego todos los resortes que las conectan. Como se muestra en la sección 1.4 la llamada *actualizarResorte*(r) afecta también a las partículas conectadas por el resorte r .

1.2.2 Resortes

Un resorte es un mecanismo mecánico que almacena energía al ser deformado [7]. En este trabajo se pretende simular la versión más común conocida como resorte de tensión/extensión que se comprime a lo largo de un eje. Se utilizarán los resortes para conectar dos partículas, y el movimiento relativo de éstas será el que lo deforma. Cada resorte tiene dos propiedades: el coeficiente de elasticidad, que determina la dureza del mismo; y la longitud de descanso, que determina el punto de equilibrio en el cual el resorte no almacena energía. El comportamiento de un resorte de tensión/extensión (que ahora simplemente llamaremos “resorte”) viene dado por el modelo del oscilador armónico simple, cuyo comportamiento lo determina la Ec (1.5) [7].

$$\mathbf{F} = -k\mathbf{x} \quad (1.5)$$

Donde \mathbf{F} es el vector fuerza que actúa sobre ambas partículas \mathbf{p}_1 y \mathbf{p}_2 del resorte, k es el coeficiente de elasticidad y \mathbf{x} es la diferencia entre la longitud del resorte y la longitud de descanso, este valor se conoce como elongación del resorte y podemos ver que la fuerza ejercida por un resorte sobre las partículas que conecta es proporcional a la elongación en un momento dado: mientras más

deformado esté el resorte, más tratará de volver a su longitud de descanso. En la sección 1.4, se mostrará como simular este fenómeno físico en pasos discretos.

1.3 Métodos de integración

Para poder hacer una simulación de un sistema masa-resorte se necesita un método que, dado el estado de la simulación un momento arbitrario t es decir x_t , utilice los principios de la Ec (1.4), para obtener el estado en el tiempo $t + \Delta t$, es decir, calcular la nueva posición así como la nueva velocidad de cada partícula del sistema luego de transcurrido un tiempo Δt (línea 2 del algoritmo 1.1) donde Δt es llamado *tamaño del paso*. Existen ya una variedad de métodos para lograr esto, formalmente conocidos como métodos de integración, aquí damos una breve introducción de los mismos.

1.3.1 Método Euler

El integrador Euler [9] es el acercamiento más simple de todos, en el cual la posición \mathbf{x} y la velocidad \mathbf{v} para el tiempo $t + \Delta t$ se calcula usando la velocidad y la aceleración de la partícula en el tiempo t , es decir:

$$\mathbf{x}_{t+\Delta t} = \mathbf{x}_t + \mathbf{v}_t \Delta t \quad (1.6)$$

y la velocidad se actualiza con:

$$\mathbf{v}_{t+\Delta t} = \mathbf{v}_t + \mathbf{a}_t \Delta t \quad (1.7)$$

El vector aceleración \mathbf{a}_t es el derivado de la Ec (1.3), que corresponde al paso 1 del algoritmo de simulación (algoritmo 1.1) Estas ecuaciones muestran que el método de Euler asume que la velocidad permanece constante mientras transcurre Δt , y la nueva posición se calcula a partir de esta suposición. Este método es considerado, poco eficiente, inestable e impreciso [9], aunque tiene la ventaja de ser muy fácil de implementar como podemos ver en las ecuaciones anteriores.

1.3.2 Método Verlet

Un acercamiento muy usado en algoritmos de dinámica molecular [8] es el del integrador Verlet. En este caso cada partícula es representada por dos valores la posición actual \mathbf{x}_t , y en vez de almacenar la velocidad, se lleva rastro de la posición anterior $\mathbf{x}_{t-\Delta t}$. Para calcular la nueva posición de la partícula, se utiliza:

$$\mathbf{x}_{t+\Delta t} = 2\mathbf{x}_t - \mathbf{x}_{t-\Delta t} + \mathbf{a}(\Delta t)^2 \quad (1.8)$$

Se puede almacenar \mathbf{x} como la “posición anterior” de la partícula para el paso siguiente. Es importante darnos cuenta que este método funciona porque $2\mathbf{x}_t - \mathbf{x}_{t-\Delta t} = \mathbf{x} - (\mathbf{x} - \mathbf{x}_{t-\Delta t})$ y $\mathbf{x} - \mathbf{x}_{t-\Delta t}$ representa una aproximación a la velocidad de la partícula (aproximada con la distancia transcurrida en el paso anterior) [8]. Este método es estable y computacionalmente rápido aunque no siempre es preciso en el sentido de que cierta cantidad de energía puede escapar del sistema [8]. También se debe considerar que es posible agregarle fricción al sistema modificando los coeficientes como en el siguiente ejemplo:

$$\mathbf{x}_{t+\Delta t} = (2 - d)\mathbf{x}_t - (1 - d)\mathbf{x}_{t-\Delta t} + \mathbf{a}(\Delta t)^2 \quad (1.9)$$

Donde d representa la fricción que amortigua el movimiento y para efectos prácticos suele tener un valor cercano a cero. Este método por ser estable y relativamente fácil de calcular, es el escogido para ser implementado en este trabajo. Se debe notar que la escogencia de este método implica que la representación de cada partícula $p \in P$ al momento de implementar este sistema debe tener la forma

$$(\mathbf{p}, \mathbf{p}', m) \quad (1.10)$$

donde \mathbf{p} y \mathbf{p}' son vectores que representan la posición y la posición anterior respectivamente y m representa la masa de la partícula. En general, una implementación del método que integra las ecuaciones del movimiento con el método Verlet tendría la forma mostrada en el algoritmo 1.2.

Algoritmo 1.2 Algoritmo de actualización de partículas Verlet.

```

1: para todo  $\mathbf{x} \in P$  hacer
2:    $\mathbf{aux} \leftarrow \mathbf{x}.\mathbf{p}$ 
3:    $\mathbf{x}.\mathbf{p} \leftarrow (2 - d) * \mathbf{x}.\mathbf{p} - (1 - d) * \mathbf{x}.\mathbf{p}' + \mathbf{a} * dt^2$ 
4:    $\mathbf{x}.\mathbf{p}' \leftarrow \mathbf{aux}$ 
5: fin para
```

Como se ve, es una traducción directa de lo expresado en la Ec (1.9), por lo que resulta sencillo de implementar en cualquier lenguaje.

1.4 Simulación de resortes

La implementación propuesta en [8] pretende simular un caso especial donde el resorte regresa a su posición de descanso en un solo paso. El algoritmo usado para este tipo de resortes es altamente estable y su codificación resulta relativamente fácil con la estructura de datos propuesta por el método Verlet, ya que al mover la posición de una partícula, se está efectivamente afectando su velocidad. Para generalizar este método se ha agregado en este trabajo la constante S que afecta el tiempo que le toma al resorte llegar a su longitud de descanso, modificando la suavidad de los cuerpos a modelar.

Algoritmo 1.3 Algoritmo de actualización de resortes propuesto por [8]. Le agregamos la constante S para poder suavizar los cuerpos.

```

1: para todo  $r \in R$  hacer
2:    $\text{delta} \leftarrow \mathbf{r.p_1} - \mathbf{r.p_2}$ 
3:    $\text{longDelta} \leftarrow \text{sqrt}(\text{delta} \cdot \text{delta})$ 
4:    $\text{diff} \leftarrow (\text{longDelta} - \text{longDescanso})/\text{longDelta}$ 
5:    $\mathbf{r.p_1} \leftarrow \mathbf{r.p_1} + \text{delta} * (0,5 * S) * \text{diff}$ 
6:    $\mathbf{r.p_2} \leftarrow \mathbf{r.p_2} - \text{delta} * (0,5 * S) * \text{diff}$ 
7: fin para

```

$\mathbf{r.p_1}$ y $\mathbf{r.p_2}$ son las partículas en los extremos del resorte r y longDescanso es su longitud de descanso. El valor de diff representa la relación entre el estiramiento del resorte y su longitud de descanso, cuando vale 0 es porque el resorte está descansando, este valor al multiplicarlo por delta nos resulta un vector que representa el movimiento que tiene que hacer la partícula $\mathbf{r.p_1}$ para que el resorte termine en su longitud de descanso, sin embargo, se distribuye este movimiento entre las dos partículas como vemos en las líneas 5 y 6 en donde lo multiplicamos por 0,5. La constante que se agregó en este trabajo S es el coeficiente que define el tiempo que le toma al resorte llegar a su posición de descanso y esta efectivamente modifica la “suavidad” del cuerpo. En la figura 1.1 se generó una gráfica que muestra el efecto que tiene la constante S en un resorte de ejemplo.

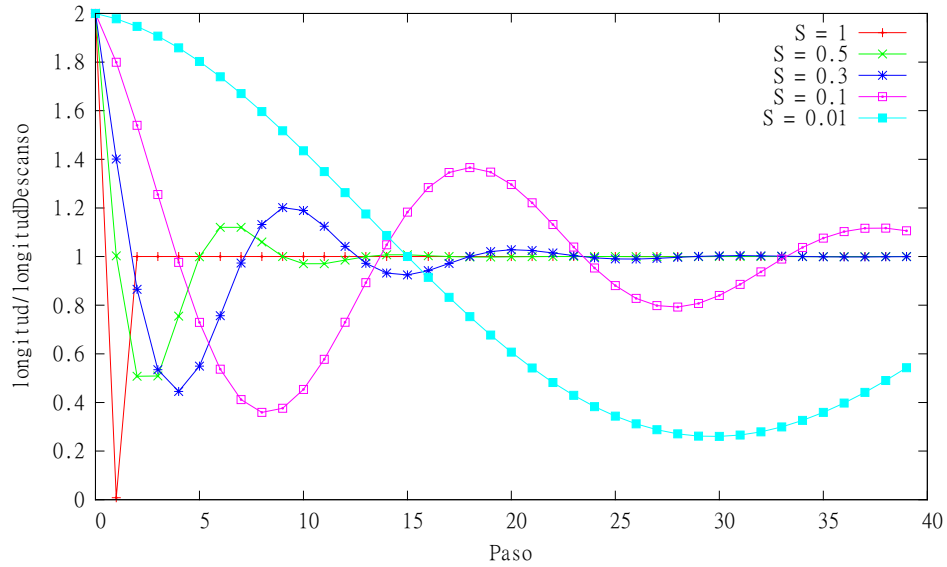


Figura 1.1: Comportamiento de un resorte para distintos valores de la constante S . Se observa que mientras S se acerca a 0, el resorte es menos rígido (tarda más en regresar a su longitud de descanso).

Se puede ver en la figura 1.1 que el caso sin S (cuando S vale 1) el resorte regresa a su longitud de descanso en apenas 2 pasos, y mientras S aumenta se incrementa la suavidad del mismo. En el capítulo 3 se muestra con mayor profundidad los efectos de esta constante que se agregó, sobre una simulación del Sistema masa-resorte.

En un solo ciclo de la simulación este algoritmo debe ejecutarse iterativamente sobre los mismos datos, esto se conoce como método de relajación, o método Gauss-Siedel, y efectivamente converge a la solución que se quiere [8]. Un aspecto importante es que, en caso de que fuera necesario, el programa puede cambiar precisión por rendimiento, es decir, se puede dejar de hacer iteraciones si se detecta que las mismas van a afectar la fluidez de la simulación, sacrificando precisión siempre y cuando sea visualmente aceptable.

1.5 El GPU

Las unidades de procesamiento gráfico (GPU por sus siglas en inglés) son piezas de hardware que tradicionalmente se han utilizado para acelerar la velocidad en la que las computadoras dibujan gráficos. Desde finales del 2002, las dos principales compañías fabricantes de GPUs, Nvidia [10] y ATI [11] (ahora AMD [12]), han venido desarrollando chips programables que en principio son capaces de procesar dos tipos de elementos, pixeles y vértices [3].

Con la introducción de estas tarjetas, se introdujo también la posibilidad de programarlas para usos distintos a la aceleración gráfica, lo que actualmente es conocido como GPGPU (*General Purpose computation in Graphics Processing Units*, o computación de propósito general en unidades de procesamiento gráfico). Se ha visto como implementaciones GPGPU de ciertos problemas han reportado mejoras en tiempos de ejecución de más de mil veces en contraste con versiones anteriores corriendo en CPUs contemporáneos [13] [14], aunque estos son casos aislados es bastante común encontrarse con implementaciones que conducen a mejoras de uno o dos órdenes de magnitud.

1.5.1 Contraste entre GPU y CPU

Los CPU históricamente han sido diseñados para ejecutar programas de propósito general que en algunos casos se ejecutan en paralelo (multi-hilo o multi-proceso en CPUs multinúcleo) pero mayormente se ejecutan en serie, a diferencia de los GPU que procesan gran cantidad de datos en paralelo (tradicionalmente vértices y pixeles) es por esto que gran parte de los transistores contenidos en un CPU son utilizados para control de flujo que permiten que estos programas seriales sean ejecutados más eficientemente usando métodos como la predicción de saltos [15] (*branch prediction*), y la ejecución fuera de orden [15] (OOOE por sus siglas en inglés).

Aunque la computación paralela se ha comenzado a explotar con los CPU multinúcleo¹ y la circuitería dedicada al procesamiento vectorial (MMX [17], SSE2 [18], AltiVec [19], etc), el incremento teórico óptimo en tiempo de ejecución alcanzable es igual al número de núcleos (utilizando el mismo algoritmo), y sabemos que el número de núcleos en los CPUs modernos es por lo general menor a 8, en contraste con los GPUs modernos que presentan hasta 512 núcleos, es decir 64 veces más.

Existen problemas específicos que se pueden resolver ejecutando muchas instancias de un mismo programa con un nivel mínimo o nulo de acoplamiento, es decir, problemas que presentan un alto grado de paralelismo de datos. Por ejemplo, en procesamiento digital de imágenes, se utilizan programas (llamados *kernels*) que actualizan regiones de una imagen independientemente de las demás; o en la suma de dos matrices A y B para obtener un resultado C , donde el elemento C_{ij} solo depende de los elementos A_{ij} y B_{ij} , entonces, si la matriz es de $n \times m$ se pudieran crear $n \times m$ instancias de un programa que solo haga una suma sin incurrir en condiciones de carrera.

Los GPU modernos presentan una arquitectura que se conoce como SIMD [20] (*Single Instruction, Multiple Data streams*, Una sola instrucción, varios flujos de datos), que permite ejecutar en paralelo un mismo programa, llamado *kernel*, con una gran cantidad de datos de entrada diferentes, esta arquitectura resulta ser muy adecuada para resolver problemas que presenten un alto grado de paralelismo de datos. Como veremos en la sección 2.2 el modelo SMR presenta un alto nivel de paralelismo de datos.

1.5.2 Arquitectura del GPU

En los sistemas modernos, típicamente el CPU se comunica con el GPU a través de un bus AGP o PCI Expreso, y es por este mismo bus que el GPU accede a memoria de sistema (RAM) para copiar datos utilizando DMA² [21]. Los GPU tienen acceso a una memoria de video (VRAM) que se encuentra en la misma tarjeta física que el chip, por lo que ofrece la mayor velocidad de acceso. En la tabla 1.1 se muestra un ejemplo de las distintas velocidades de transferencia de datos en diferentes partes relevantes del sistema.

En esta tabla se puede observar que la memoria de video ofrece una velocidad mucho mayor que la memoria de sistema, de hecho el sistema que se usa para este proyecto (Nvidia GeForce GTX 260M)

¹Los CPU Multinúcleo (*dual-core*, *quad-core*, etc) son *chips* con dos o más procesadores en el mismo dado, y son capaces de ejecutar varios hilos al mismo tiempo. Un ejemplo de estos es el Intel Core 2 Duo [16].

²Característica de un sistema que le permite a los dispositivos periféricos acceder a memoria principal (RAM) independientemente del CPU

Tabla 1.1: Ejemplo de anchos de banda disponibles en distintos componentes del sistema [1]

Componente	Ancho de banda
Interfaz de memoria de video	35 GB/seg.
Bus PCI Express (x16)	8 GB/seg.
Interfaz de memoria de sistema (800MHz de FSB)	6.4 GB/seg.

dispone de más de 60 GB/seg. de ancho de banda. Como ya se mostrará más adelante, la implementación que muestra este trabajo hace uso estratégico de este hecho para evitar posibles cuellos de botella en los accesos a memoria.

Los primeros GPU estaban diseñados con lo que se conoce como *fixed pipeline* [22], es decir, tenían funciones predefinidas que aplicaban a los vértices y pixeles, y el usuario simplemente podía modificar algunos parámetros de estas funciones. En líneas generales un GPU funcionaba de esta manera:

1. Se iluminan los vértices.
2. Se proyectan los vértices al plano de la pantalla.
3. Se ensamblan triángulos a partir de los vértices procesados.
4. Se *rasterizan* los triángulos para formar fragmentos.
5. Se procesan los fragmentos (pruebas de profundidad y stencil).
6. Se mezcla con el resultado anterior (*Blending*).

Luego de una importante evolución, los GPUs actuales tienen etapas programables que han permitido la creación de técnicas que antes no eran posibles de ejecutar en tiempo real, Las etapas programables actuales son:

- *Vertex Shader*: permite modificar los vértices, moverlos, iluminarlos, proyectarlos, entre otras cosas. Esta etapa reemplaza los pasos 1 y 2 del *fixed pipeline*.
- *Geometry Shader*: permite modificar los triángulos ya ensamblados, y hasta generar nuevos vértices. Esta etapa es relativamente nueva (A partir de la llegada de las tarjetas con soporte para la especificación DirectX 10 [23]) y ofrece capacidades que eran imposibles con el *fixed pipeline*.

- *Fragment Shader o Pixel Shader*: permite modificar cada pixel que fue generado por los triángulos proyectados en pantalla, esto posibilita la iluminación “por-pixel” así como efectos sobre la imagen final para agregar efectos de brillo, contraste, entre otras muchas cosas. Esta etapa reemplaza el paso 5 del *fixed pipeline*.

Los primeros acercamientos al computación de propósito general en el GPU utilizaban los lenguajes de *Shaders*³ para explotar las capacidades de procesamiento de vértices y pixeles del GPU y con esto resolver distintos tipos de problemas, un acercamiento muy común era representar los datos del problema a una textura para procesarla en el GPU con uno o varios Pixel Shaders como se muestra en la figura 1.2.

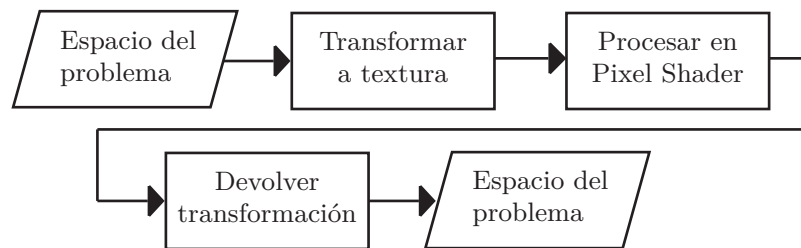


Figura 1.2: Diagrama de flujo del acercamiento tradicional al GPGPU utilizando el paradigma de *Shaders*.

Los datos son transformados del espacio del problema a una o varias texturas, estas texturas se procesan en el *Pixel Shader* para posteriormente leer las texturas resultado (no necesariamente las mismas de entrada) y convertirlas de regreso al espacio del problema.

Las etapas de *Vertex Shader* y *Pixel Shader* solían tener su propio *hardware* dedicado en el GPU que las ejecutaba a cada una por separado, pero esto creaba un sistema heterogéneo donde el rendimiento del GPU dependía de que el uso de esta circuitería estuviera balanceado. Es por esto que los GPU más recientes son creados con la arquitectura unificada de *Shaders* (*Unified shader architecture*). Los GPUs que presentan esta arquitectura en vez de tener circuitos independientes para procesar vértices y pixeles, tienen los llamados procesadores de flujos [24] (*stream processors*) que son los elementos en el GPU que procesan vértices, geometría y fragmentos de pixeles.

Una vez que las compañías fabricantes de GPUs (Nvidia y ATI) hicieran al componente principal de las tarjetas de video capaz de procesar tipos de datos heterogéneos, solo hacía falta exponer estas capacidades para tener un *framework*⁴ creado para el GPGPU (esto lo hacen CUDA [25],

³Lenguajes utilizados para crear *Vertex*, *Geometry* y *Pixel Shaders*

⁴En este contexto, se refiere a un conjunto de librerías, ejecutables y herramientas.

OpenCL [26] y DirectCompute [23], que se mencionan en la próxima sección) donde los datos de entrada consisten en flujos de elementos del mismo tipo, pero que pueden ser totalmente distintos de los datos tradicionales (vértices, líneas, triángulos y texturas).

1.5.3 Tecnologías para desarrollar programas GPGPU

Durante el desarrollo de este trabajo se evaluaron varias opciones para programar el GPU, las cuales se exponen a continuación.

1.5.3.1 Shaders

Con el surgimiento de los GPU programables, surgieron también lenguajes que se ajustaban al paradigma propuesto por esta tecnología, los programas que se crean con estos lenguajes son llamados *Shaders* y se usan para crear efectos de *rendering* con una gran flexibilidad, a diferencia con el *fixed pipeline*. Las desventajas de utilizar *Shaders* son:

- Los lenguajes para crearlos (HLSL, GLSL, Cg) no exponen capacidades del *hardware* que muy probablemente iban a ser necesarias para crear nuestra implementación como lecturas y escrituras arbitrarias de memoria o memoria compartida.
- Los lenguajes están orientados a programación gráfica, por lo que los datos de entrada se manejan como texturas o vértices lo que puede ser incómodo de manejar.
- La salida está limitada a 4 *floats* de 32 bits en lugares predefinidos.

Aun así, la principal ventaja de utilizar *Shaders* es que la mayoría de las tarjetas gráficas actuales tienen la capacidad de ejecutarlos.

1.5.3.2 CUDA

CUDA (*Compute Unified Device Architecture*, cuya traducción más ajustada sería Arquitectura de dispositivo de computación unificada) es una tecnología relativamente nueva (la primera versión de prueba fue lanzada en febrero del 2007) creada por Nvidia [25]. Las principales ventajas de utilizar CUDA son:

- Se utiliza “C for CUDA” que es una extensión de C99 para programar el GPU.
- Permite hacer lecturas y escrituras de direcciones arbitrarias en memoria de video.
- Expone una memoria compartida entre hilos, que puede ser utilizada como *cache* (manejado manualmente).
- Es una tecnología especialmente desarrollada para crear programas GPGPU.

CUDA tiene la limitante de estar solo soportado en tarjetas Nvidia, pero como el hardware disponible para la implementación del proyecto efectivamente posee una tarjeta de esta marca, y Nvidia posee alrededor del 60 % del mercado de GPUs [27], no se consideró como un factor decisivo.

1.5.3.3 OpenCL y DirectCompute

Actualmente existen dos alternativas a CUDA, una es OpenCL, propuesta por Apple Computer [28] y la otra es DirectCompute, propuesta por Microsoft Corporation [29]. Haciendo una revisión rápida, nos dimos cuenta de que no habían suficientes recursos que nos permitieran entender profundamente el funcionamiento de estas tecnologías, así como una comunidad de desarrolladores, lo que iba a limitar la capacidad de implementación y más adelante de optimización.

Dadas estas condiciones, se tomó la decisión de utilizar CUDA como herramienta para programar el GPU ya que ofrecía las capacidades que se necesitaban además de suficientes fuentes de consulta como “NVIDIA CUDA Programming Guide 2.3- [21] ”NVIDIA CUDA C Programming Best Practices Guide” [30], el manual de referencia de CUDA [31] y una establecida comunidad de desarrolladores que estaban (y continúan estando) activamente desarrollando con esta librería, especialmente en la comunidad científica.

1.6 Arquitectura de CUDA

En esta sección se hablará de las generalidades de la arquitectura de CUDA y de los elementos que se deben conocer para poder crear programas utilizando esta librería.

La arquitectura de los GPUs que soportan CUDA, es la del G80 de Nvidia y sus sucesores [21]. Estos procesadores están compuestos de un arreglo de *Streaming Multiprocessors*

(Multiprocesadores de flujos) o SM, los cuales a su vez contienen 8 *Scalar Processors* (Procesadores de Escalares) o SP y son estos los que ejecutan cada hilo de ejecución de un kernel. Es importante destacar que las aplicaciones GPGPU tienden a ser escalables ya que automáticamente un GPU con más multiprocesadores ejecutará un programa mas rápido que un GPU con menos [21].

Los programas en CUDA deben organizarse en bloques de hilos (*thread blocks*), cada bloque se ejecuta en un solo multiprocesador, el cual distribuye los hilos en grupos de 32, llamados *warps*, y utiliza sus 8 procesadores escalares para procesar cada instrucción de un warp en 4 ciclos de reloj. Cuando un *warp* ejecuta una lectura de memoria, el multiprocesador planifica la ejecución de otro mientras el primero termina, efectivamente aumentando la utilización del GPU.

Aunque cada bloque se ejecuta en un solo multiprocesador, estos pueden tener varios bloques activos al mismo tiempo, el número de bloques activos por multiprocesador depende de la ocupación de los registros por cada kernel (es decir, la cantidad de registros necesarios para ejecutar el kernel) y la cantidad de memoria compartida que utilice cada bloque (ver sección 1.6.1).

Para comodidad del programador el identificador o índice de cada hilo es un vector de 3 enteros, y como el tamaño del bloque es definido por el programador, éste puede ajustar el tamaño de los bloques para que se ajusten a la naturaleza de su problema. Por ejemplo, si se quiere atacar un problema donde los datos están naturalmente organizados en una estructura bidimensional, como una matriz 2D, típicamente el programador escogerá bloques de dos dimensiones, por el contrario, si el problema es unidimensional, como por ejemplo para procesar digitalmente alguna señal (DSP), los bloques serán de una dimensión. Los bloques a su vez se organizan en una rejilla *grid*) bidimensional. Los bloques y la rejilla son dos herramientas que sirven para determinar el número de hilos a crear, y al mismo tiempo organizar esos hilos de tal forma de que tengan una “posición” en los datos del problema.

A modo de ilustración, se muestra en la figura 1.3 un ejemplo de como se organizan los hilos en una rejilla de tamaño 3×2 con bloques de tamaño 4×3 .

1.6.1 Jerarquía de memoria en CUDA

Cada multiprocesador tiene acceso a los siguientes tipos de memoria:

- Registros: Un conjunto de registros de 32 bits en cada multiprocesador.

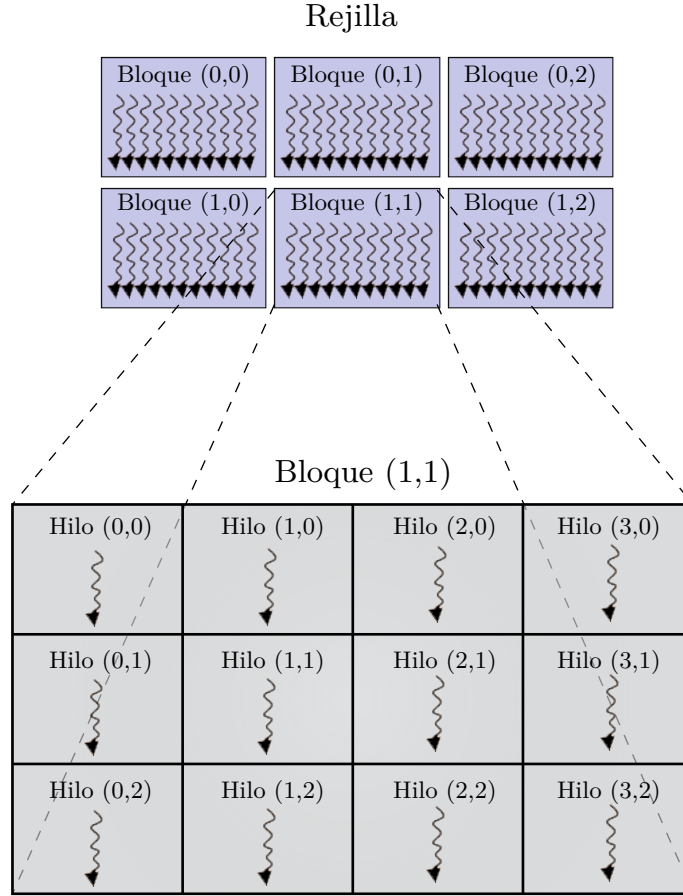


Figura 1.3: Ejemplo de una rejilla de tamaño 3×2 con bloques de tamaño 4×3 . Se puede ver como se organizan los hilos en la misma [21].

- Memoria compartida: Un *cache* de muy baja latencia, manejado por software y compartido por todos los SP de un multiprocesador, y por ende, por todos los hilos de un bloque.
- Memoria de constantes: Un *cache* que acelera las lecturas del espacio de constantes que es una región especial en memoria de video.
- Memoria de texturas: Un *cache* que acelera las lecturas del espacio de texturas, que también es una región especial en memoria de video.
- Memoria de video (o de dispositivo): Memoria sin *cache* de lectura y escritura pero de relativamente alta latencia (una lectura o escritura puede demorar alrededor de 500 ciclos de reloj).

1.6.2 *Kernels*

Los programas que ejecuta el GPU son denominados *kernels* y se dice que son funciones de dispositivo (*device function*), por que se ejecutan en el dispositivo de video y son llamadas desde un programa común y corriente (ejecutándose en el CPU) llamado anfitrión (*host program*) [21]. Los kernels son escritos como funciones en una extensión de C llamada ***C for CUDA***. Para que una función puede ser compilada como un kernel para ser ejecutado en el GPU, tiene que estar escrita con uno de los siguientes calificadores [31]:

- `__global__` Estas son las funciones que se ejecutan en el GPU y que pueden ser llamadas desde el programa anfitrión que se ejecuta en el CPU. Podemos pensar en ellas como las función principal o main de un programa de GPU.
- `__device__` Solo puede ser llamada desde una función que se ejecuta en el dispositivo (una función `__global__` o alguna otra función `__device__`).

También una función puede ser etiquetada con el calificador `__host__` y el calificador `__device__` al mismo tiempo para que el compilador genere dos versiones, una para GPU y otra para CPU.

El propósito de un *kernel* es típicamente ejecutarse sobre muchos datos para aplicarle la misma transformación a cada uno, es por esto que el GPU normalmente se utiliza para ejecutar muchas instancias del *kernel* en hilos independientes que se ejecutan en paralelo (organizados en bloques, que a su vez se organizan en una rejilla de bloques, ver sec. 1.6).

1.6.3 Llamadas a *kernels*

Para ejecutar un *kernel* en el GPU utilizando CUDA, se tiene que especificar las dimensiones de cada bloque de hilos y la de la rejilla, así como el tamaño de la sección de memoria compartida que se le debe asignar a cada bloque, para hacer esto CUDA provee un tipo nuevo llamado `dim3` que representa una medida entera en 3 dimensiones (un struct de tres `int`: `x`, `y`, `z`) y la llamada se debe hacer utilizando la sintaxis especial introducida por ***C for CUDA*** la cual tiene la siguiente forma [31]:

```
función<<<GridDim, BlockDim, SharedDim>>>(...);
```

Donde `GridDim` es de tipo `dim3` y representa el tamaño bidimensional de la rejilla de bloques (se ignora el valor de `GridDim.z`), `blockDim` es también de tipo `dim3` y representa la dimensión tridimensional de cada bloque, y `SharedDim` es un `size_t` que representa el tamaño en bytes de la región de memoria compartida que tendrá a disposición cada bloque. El resto de los elementos forman parte de la sintaxis normal de llamada de funciones en C (nombre y parámetros).

Para permitir que se aproveche el tiempo de CPU mientras se ejecuta algún *kernel* las llamadas son asíncronas, es decir, retornan inmediatamente, para hacer que la aplicación se bloquee hasta que termine el último *kernel* ejecutado se utiliza la función `cudaThreadSynchronize()`.

Para ilustrar lo explicado anteriormente, se muestra un programa en C sencillo pero completo donde se hace una llamada a un kernel CUDA vacío, con un tamaño de bloque y rejilla de ejemplo. Este código ejecuta 14.720 ($8 \times 8 \times 23 \times 10$) instancias del kernel (hilos), el listado del código se muestra a continuación:

```
1  #include <cuda.h>
2  #include <stdio.h>
3
4  // Declaración del kernel.
5  __global__ void helloWorld();
6
7  int main(int argc, char** argv)
8  {
9      // Establecemos el tamaño de grid y de bloque
10     dim3 dimGrid(8, 8, 0);
11     dim3 dimBlock(23, 10, 0);
12
13     // Llamamos al kernel
14     helloWorld<<< dimGrid, dimBlock >>>(d_str);
15
16     return 0;
17 }
18
19 // Función que se ejecuta en el GPU.
20 __global__ void helloWorld()
21 {
22     // Este valor representa un identificador del hilo y lo
23     // utilizamos como índice para acceder los datos.
24     int idx = blockIdx.x * blockDim.x + threadIdx.x;
25 }
```

1.6.4 Patrón de acceso a memoria

Como se mostró en la sección 1.6.1 los accesos a memoria de video son prohibitivamente costosos, afortunadamente CUDA implementa un sistema llamado **acceso unificado** (*coalesced access*) donde los accesos de todos los hilos de un *half-warp* (la primera o la segunda mitad de un *warp*, es decir, 16 hilos) se hacen en una o dos transacciones de memoria de 32, 64 o 128 bytes. Para que el acceso sea unificado en una sola transacción se tienen que cumplir las siguientes condiciones:

1. Los hilos del *half-warp* deben acceder a palabras de:
 - 4 bytes: resultando en una transacción de 64 bytes.
 - 8 bytes: resultando en una transacción de 128 bytes.
 - 16 bytes: resultando en dos transacciones de 128 bytes.
2. Las palabras accedidas deben estar en el mismo segmento de memoria de 64, 128 o 256 bytes.
3. Los hilos deben acceder a memoria en orden: el *i*ésimo hilo accederá la *i*ésima palabra del segmento.

Si un *half-warp* no cumple con alguna de las condiciones anteriores, el GPU terminará haciendo una transacción de memoria por cada hilo del *warp*, lo que afecta enormemente el rendimiento del *kernel* [21], es por esto que es importante diseñar las estructuras de datos de tal forma que los hilos terminen haciendo accesos unificados a memoria.

También es necesario que cada lectura esté alineada a su tamaño, es decir, alineada a 4 para las lecturas de 4 bytes, a 8 para las lecturas de 8 bytes y a 16 para las lecturas de 16 bytes, para hacer esto se puede utilizar el especificador de alineación `__align__()` en la declaración de una estructura, por ejemplo, para una estructura de 8 bytes se utiliza `__align__(8)`, para estructuras de 16 bytes o más, hay que alinearlas con `__align__(16)` [21].

CAPÍTULO 2

DISEÑO DE LA IMPLEMENTACIÓN

Con los conocimientos obtenidos durante la investigación teórica expuestos en el capítulo anterior, se procederá a crear un diseño de la implementación que se ajustara al paradigma de programación GPGPU expuesto por la librería CUDA, con el objetivo fundamental de ejecutar en paralelo las subrutinas de los métodos propuestos en los algoritmos 1.2 y 1.3 para mejorar sus tiempos de ejecución. Para tener un marco de referencia con el cual medir el rendimiento de la implementación propuesta, se creará también una implementación que se ejecute en serie en el CPU y se utilizarán los tiempos de ejecución para comparar resultados.

Durante este capítulo se utilizará varias veces una instancia de un SMR genérico para ilustrar los ejemplos dados, este modelo lo presentamos en la figura 2.1.

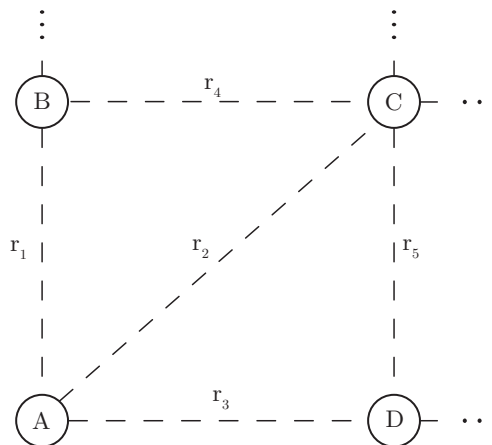


Figura 2.1: Representación gráfica de la instancia genérica de un SMR utilizado para ilustrar ejemplos, la partícula **A** está conectado por 3 resortes, el resto puede tener más conexiones.

2.1 Herramientas de trabajo

El proyecto se desarrolló sobre el sistema operativo *Windows 7* utilizando *Visual C++ 2008 Express Edition* [32] como ambiente de desarrollo, se utilizó la herramienta *CUDA Visual Profiler* [25] para encontrar cuellos de botella y mejorar el rendimiento.

Existen versiones de CUDA para otras plataformas como CUDA.NET o jCUDA, pero para el inicio del proyecto estas no tenían soporte para interoperabilidad con DirectX por lo que decidimos trabajar con la versión oficial para C/C++.

2.2 El paralelismo de datos en el SMR

Viendo al SMR como un grafo (ver sección 1.1), se puede decir que el estado de un nodo (una partícula) depende del estado de los nodos alcanzables desde el mismo (a través de los resortes del SMR), como ilustración de este hecho se muestra la figura 2.2, en donde se puede ver que si la partícula **A** sufre alguna transformación, es decir, se mueve a causa de alguna fuerza externa, naturalmente podemos imaginarnos que la partícula **B**, aunque no esté directamente conectada con **A**, se verá de alguna forma afectada porque los resortes propagarán los efectos del movimiento originado en **A**.

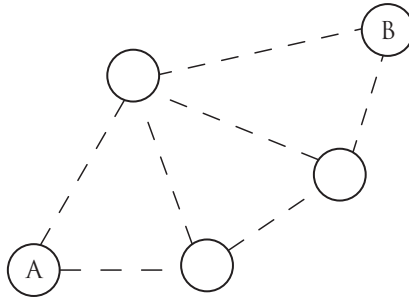


Figura 2.2: Ejemplo del problema de dependencia entre partículas. **A** y **B**, aunque no estén conectadas directamente, son interdependientes.

Este alto nivel de dependencia entre todas las partículas del sistema parece indicar que no hay mucho paralelismo de datos posible en el problema, sin embargo, tanto el algoritmo de actualización

de partículas, como el de actualización de resortes mostrado en los algoritmos 1.2 y 1.3 respectivamente, utilizan un acercamiento que actualiza cada elemento independientemente de los demás, lo que ofrece la oportunidad de distribuir su ejecución en diferentes hilos del GPU, siempre y cuando no se tenga el riesgo de corrupción de datos por alguna condición de carrera como explicamos en la sección 2.2.1.

2.2.1 Región crítica de memoria

El algoritmo de actualización de resortes utiliza la posición de sus dos partículas tanto como datos de entrada como de salida (ver algoritmo 1.3), esto puede crear una condición de carrera si dos o más resortes están conectados a la misma partícula como ocurre con la partícula **A** en la figura 2.3, en donde, los resortes r_1 , r_2 y r_3 podrían competir entre ellos si tratasen de ajustar la posición de **A** en paralelo.

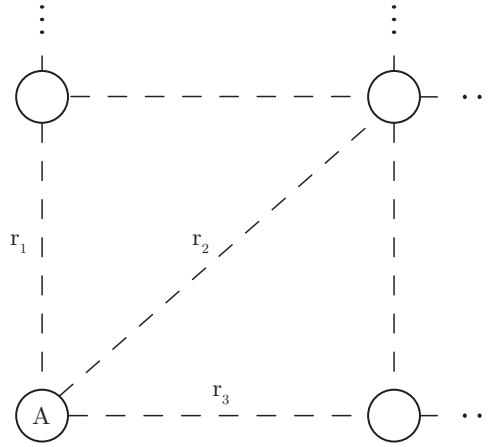


Figura 2.3: Conflicto entre resortes que puede ocasionar una condición de carrera, r_1 , r_2 y r_3 no pueden actualizarse en paralelo.

Se puede decir que, si existen dos o más resortes conectados a la misma partícula, lo cual es bastante común en muchos modelos, y estos se actualizan en paralelo, existirá una condición de carrera lo que ocasionará la ocurrencia de los eventos que se describen a continuación:

- Al procesar los resortes, los hilos leerán la posición de la partícula al mismo tiempo, y el resultado del algoritmo resultará incorrecto. El método iterativo propuesto en el algoritmo 1.3 actualiza cada resorte en serie, utilizando la salida de la actualización del primer resorte como entrada para el segundo, la del segundo como entrada del tercero, y así sucesivamente.

- Varios resortes actualizarán la posición de la partícula al mismo tiempo, es decir, varios hilos tratarán de escribir en la misma dirección de memoria al mismo tiempo, y según lo expuesto en la guía de programación de CUDA [21], solo una de las escrituras sería satisfactoria, lo que desactivaría los efectos del resto de los resortes.

La solución a este problema se expone en la sección siguiente.

2.2.2 Paralelización del algoritmo de resortes

La solución que se propone para que no ocurra ninguno de los eventos descritos en la sección 2.2.1 es ejecutar las actualizaciones de resortes en distintas iteraciones ejecutadas en serie, donde ningún resorte en una iteración afecte a la misma partícula que otro, por lo que se pueden actualizar en paralelo. Para hacer esto debemos crear una partición de R en subconjuntos R_i lo cuales deben cumplir con la propiedad de la expresión 2.1 para efectivamente ser paralelizables.

$$\{\forall p, q, i \mid R_i \subset R \wedge p, q \in R_i : p_1 \neq q_1 \wedge p_1 \neq q_2 \wedge p_2 \neq q_1 \wedge p_2 \neq q_2\} \quad (2.1)$$

Por ejemplo, utilizando el modelo genérico de la figura 2.1 se propone una planificación de los kernels de resorte de la forma que se muestra en la tabla 2.1.

Tabla 2.1: Ejemplo de planificación de los hilos que ejecutan el kernel de actualización de resortes

Iteración	1		2		3
Resortes en paralelo	r_1	r_2	r_3	r_4	r_2
Partículas afectadas	A, B	C, D	A, D	B, C	A, C

Se puede observar que en ninguna iteración existen dos resortes conectados a la misma partícula. Partir R en estos subconjuntos tiene como consecuencia la creación de una relación de entre la conectividad del grafo y la cantidad de paralelización de datos disponible en el modelo: mientras aumenta el primero, el segundo disminuye. Por ejemplo, si un nodo tiene grado N , se tendrán que actualizar las N aristas (resortes) incidentes en N iteraciones distintas. En la tabla 2.1 se puede ver un ejemplo, donde los resortes que afectan a la partícula **A** (r_1 , r_2 y r_3), son actualizados en 3 iteraciones distintas. Por lo explicado anteriormente, se puede decir que el grado máximo del grafo (la mayor cantidad de conexiones en algún nodo) representa una cota inferior para el número de iteraciones necesarias para actualizar todos los resortes.

Los subconjuntos de resortes que se actualizan en paralelo pueden ser creados utilizando el algoritmo 2.1 que se propone en este trabajo y cuya implementación en C++ anexamos en el listado A.1.

Algoritmo 2.1 Algoritmo de partición de resortes en subconjuntos paralelizables. Crea una partición cuyos subconjuntos pueden ser ejecutados en paralelo.

```

1:  $particion \leftarrow \{\}$ 
2: mientras  $|R| > 0$  hacer
3:    $subconjunto \leftarrow \{\}$ 
4:    $particulasEnSubconjunto \leftarrow \{\}$ 
5:   para todo  $r \in R$  hacer
6:     si  $r_1 \in particulasEnSubconjunto \vee r_2 \in particulasEnSubconjunto$  entonces
7:       continuar
8:     fin si
9:      $insertar(r_1, particulasEnSubconjunto)$ 
10:     $insertar(r_2, particulasEnSubconjunto)$ 
11:     $insertar(r, subconjunto)$ 
12:     $eliminar(r, R)$ 
13:   fin para
14:    $insertar(subconjunto, particion)$ 
15: fin mientras
16: devolver  $particion$ 

```

La implementación de la simulación del SMR propuesta en este trabajo toma cada uno de los subconjuntos devueltos por el algoritmo 2.1 y los ejecuta uno por uno: cada resorte de un subconjunto en paralelo, y un subconjunto a la vez. Esto resuelve el problema explicado en la sección 2.2.1 y expone el paralelismo de datos del problema.

2.3 Implementación de la simulación en CUDA

Dada la cualidad experimental de este trabajo, se vió la necesidad de determinar a través de ensayo y error, pero basado en una amplia investigación, la mejor forma de implementar los algoritmos propuestos utilizando la librería CUDA. En principio se trataron dos alternativas una de las cuales (la segunda) resultó ser mucho más eficiente como se verá en las próximas dos secciones.

2.3.1 Primer enfoque: Múltiples llamadas

En nuestro primer enfoque se ejecuta cada iteración de resortes en llamadas independientes a *kernels*, es decir, por cada subconjunto paralelizable se hace una llamada a CUDA para que ejecute el *kernel* sobre los resortes de ese subconjunto, esto se ejemplifica en la figura 2.4.

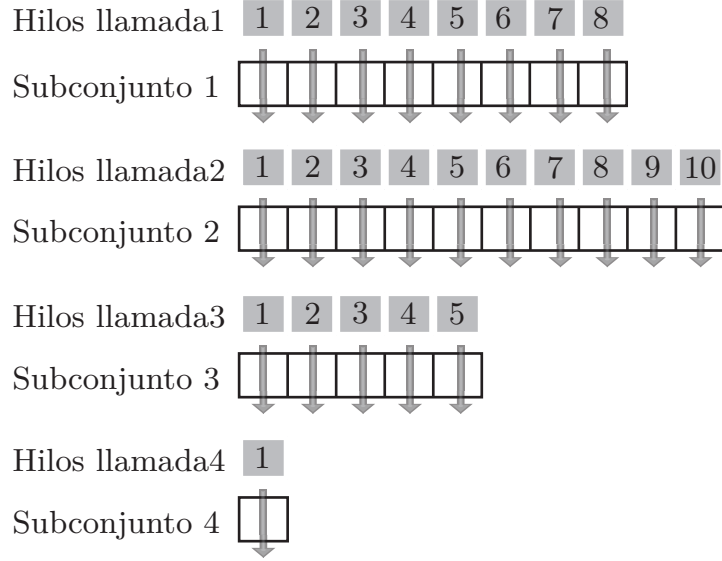


Figura 2.4: Actuación de los hilos con múltiples llamadas sobre una partición de R de ejemplo. Los cuadros oscuros representan los hilos ejecutándose en el GPU y los cuadros claros representan los subconjuntos de resortes en los que estos actúan.

Como se puede ver, solo se crean los hilos que son necesarios para procesar los resortes de cada subconjunto (redondeado al próximo múltiplo de 256, como veremos ahora), a diferencia del enfoque de la sección siguiente.

Se utiliza un tamaño de bloque de $(256, 1, 1)$, es decir, 256 hilos por bloque, y se crean tantos bloque como sean necesarios en la rejilla para que la cantidad de hilos sea la mínima con un valor mayor o igual a la cantidad de resortes del subconjunto, lo cual se ejemplifica en la tabla 2.2. Anexo está el código que calcula el tamaño de la rejilla y hace llamada, así como el del *kernel* (listados A.2 y A.4 respectivamente).

Tabla 2.2: Tamaños de bloque para diferentes cantidades de resortes (Primer enfoque)

Resortes en subconjunto	Tamaño de rejilla	Número de hilos	Utilización
256	(1,1,1)	256	100 %
257	(2,1,1)	512	50 %
1024	(4,1,1)	1024	100 %
10241 ($256 \times 40 + 1$)	(11, 1, 1)	10496	97 %

Como se puede observar en la tabla 2.2, si el numero de resortes no es múltiplo de 256 (el tamaño de bloque), el número de hilos tendrá que redondearse al próximo múltiplo de 256 (porque el tamaño de bloque es constante a lo largo de la rejilla), pero se puede ver que mientras más aumenta

la cantidad de resortes en el subconjunto, menos importa si el número de resortes es múltiplo de 256 porque la utilización tenderá a 100 %.

El principal problema de este enfoque es que el *overhead* que tiene cada ejecución de un *kernel* (tiempo de ejecución de CUDA y *driver* de video) es muy alto y se pierde mucho tiempo que pudiera utilizarse para la simulación [21].

2.3.2 Segundo enfoque: Iteraciones dentro del kernel

Dado el problema del primer enfoque, se vio la necesidad de probar otro acercamiento pero esta vez haciendo solo una llamada al *kernel*, y que este hiciera las iteraciones sobre cada subconjunto de resortes internamente. Como se explica en la sección 1.6, cada hilo tiene un índice que le dictará cuál resorte en cada subconjunto debe actualizar por lo que en la llamada se pasan todos los subconjuntos en un arreglo de arreglos de resortes. Este enfoque ejemplifica en la figura 2.5.

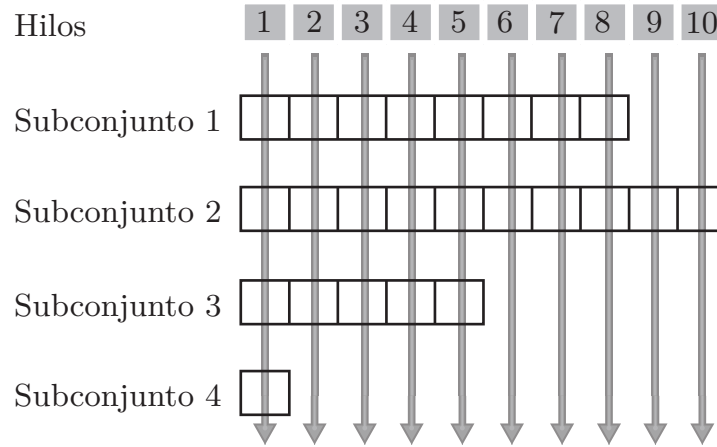


Figura 2.5: Actuación de los hilos con iteraciones dentro del *kernel* sobre una partición de R de ejemplo. En este caso solo se crean hilos una sola vez y estos actúan a través de los distintos subconjuntos de resortes.

Como se ve en la figura 2.5, se crean suficientes hilos para actualizar al **mayor** subconjunto de resortes y un mismo hilo ejecuta la rutina de actualización para todos los resortes que tengan un mismo índice en distintos subconjuntos, esto trae como desventaja que cuando la diferencia entre el mayor subconjunto y algún otro es muy grande, existirá una gran cantidad de hilos ociosos. Aunque intuitivamente esta desventaja pareciera tener un alto impacto en el desempeño, esta implementación demostró ser mucho más eficiente que la primera porque elimina el *overhead* de hacer llamadas a *kernels*, lo cual se puede llegar a tomar muchísimo tiempo.

El código del *kernel* y los parámetros de su llamada se encuentran en los listados A.3 y A.5 anexas, el cual es básicamente una implementación en C de 1.3 para los tipos de datos que propone este trabajo.

2.4 Algoritmo de actualización de partículas

Para ejecutar el método Verlet en el GPU no hace falta construir estructuras de datos especiales como se hizo con los resortes, sino que este método es directamente paralelizable puesto que en cada ciclo se actualiza una única partícula.

Para ejecutarlo en el GPU simplemente se crea un hilo por cada partícula del sistema y que éste actualice el estado de la misma. En este caso el *kernel* estará aprovechando los accesos a memoria unificado (sección 1.6.4), cada hilo estará leyendo 16 bytes (4 floats de 4 bytes cada uno) en orden: el *i*ésimo hilo lee la *i*ésima partícula, y por supuesto se aseguró de que nuestro *buffer* de partículas esté alineado a 16 bytes porque se utiliza el tipo predefinido por CUDA `float4`.

2.5 Transferencia de datos e interoperabilidad con DirectX

En muchas aplicaciones GPGPU los resultados se necesitan en la aplicación anfitrión, por lo que se utiliza el GPU para procesar datos de la siguiente forma:

1. Se copian los datos del RAM de sistema a VRAM.
2. Se ejecutan kernels en GPU para procesar los datos en VRAM.
3. Se copian los resultados de VRAM al RAM de sistema.

En este caso, se necesitan los datos recién procesados para dibujarlos en pantalla, tarea que se hace en el mismo GPU, por lo que se transfieren los datos una sola vez al GPU y se intercalan actualizaciones de partículas y resortes con llamadas a dibujar de DirectX [23]¹. Esto es posible ya que CUDA ofrece un modo de interoperabilidad con las librerías gráficas DirectX y OpenGL [33],

¹DirectX es una colección de APIs (*Application Programming Interface*) creadas para facilitar las complejas tareas relacionadas con multimedia, especialmente programación de juegos y vídeo.

donde un mismo *buffer* en VRAM puede ser accesado por CUDA y por la librería gráfica sin necesidad de copiar los datos.

La implementación propuesta realiza la siguiente secuencia de operaciones:

1. Se copian los datos del RAM de sistema a VRAM
2. Se dibujan lo vértices
3. Se actualizan las partículas como se describe en la sección 2.4.
4. Se actualizan los resortes como se describe en la sección 2.3.2.
5. Se repite 2, 3, y 4, hasta que termine la simulación.

Es importante destacar que para aprovechar la interoperatividad CUDA-DirectX, las estructuras de datos deben ser compatibles. En el Vertex Shader que se utiliza para dibujar se considera que la estructura de datos de entrada es una partícula (y no un vértice común y corriente), por suerte la estructura es muy parecida: un vértice tradicionalmente tiene valores para X , Y , y Z , nuestras partículas también tiene X , Y , y Z además de la masa M , que el *Vertex Shader* simplemente ignora.

2.6 Utilización del GPU

Supóngase que se aplica el algoritmo de partición de resortes (algoritmo 2.1) a un modelo hipotético constituido por 24 resortes y se determinó que los resortes debían ir distribuidos en 4 subconjuntos paralelizables como muestra la figura 2.6. Supongamos también que la simulación correrá en un GPU con 2 multiprocesadores (SM),

En este ejemplo, se puede notar que la utilización del GPU varía según la cardinalidad de cada subconjunto paralelizable. En la primera se utiliza el 50 % de la capacidad del GPU, en la segunda el 62.5 %, en la tercera el 31 %, y la última solo utiliza un 5 %, lo interesante es que, en todos los casos, el tiempo de ejecución de cada una será estadísticamente parecido ya que cada resorte se actualizará en paralelo (sin tomar en cuenta los accesos a memoria) y de hecho se pudieran agregar resortes en alguna iteración hasta llegar a un nivel de 100 % de utilización y el rendimiento permanecería casi igual.

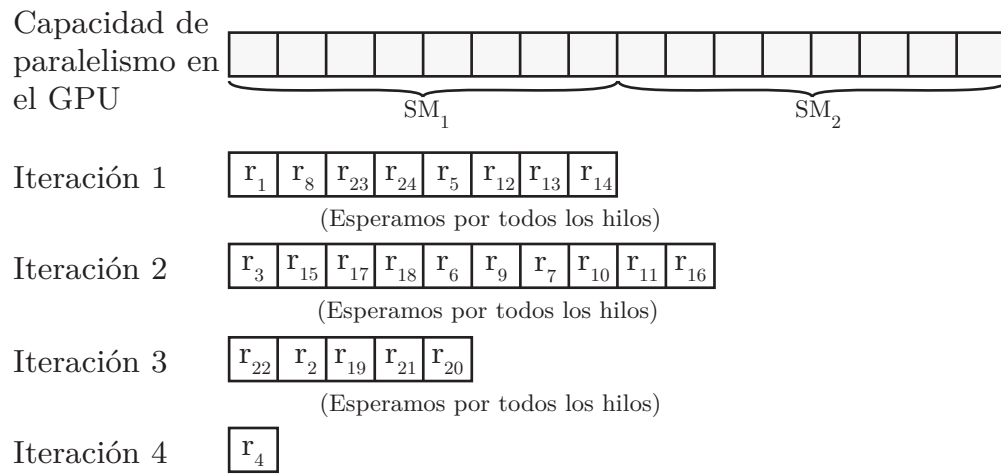


Figura 2.6: Iteraciones que hace nuestro método para una determinada partición ejemplo de R .

CAPÍTULO 3

EXPERIMENTOS Y RESULTADOS

Luego de implementar nuestro diseño, se mostrará en este capítulo los resultados obtenidos luego de varios experimentos, así como las herramientas utilizadas y las pruebas que llevamos a cabo con el sistema. Se probó la versión de GPU (segundo enfoque) y una versión de control que se ejecuta en el CPU y que actualiza tanto las partículas como los resortes en serie.

3.1 Sistema utilizado

Para realizarle pruebas a nuestra implementación, se utilizó un sistema con las características mostradas en la tabla 3.1.

Tabla 3.1: Características del sistema utilizado para probar

CPU	Intel Core i7 Q720
Frecuencia CPU	1.60 GHz
RAM	4 GB (1333 MHz FSB)
GPU	Nvidia GeForce GTX 260M
GPU RAM (VRAM)	1 GB
Frecuencia GPU	1.25 GHz

La información específica sobre las capacidades CUDA del GPU se muestran en la tabla 3.2.

Tabla 3.2: Características del sistema CUDA utilizado.

Versión <i>driver</i> CUDA	3.0
Versión <i>runtime</i> CUDA	2.3
Cantidad de multiprocesadores	14
Cantidad de núcleos (<i>stream processors</i>)	112
Registros por bloque	8192
Memoria compartida por bloque	16384 bytes
Número máximo de hilos por bloque	512
Ejecución y copia de datos concurrente	Si

3.2 Modelos de prueba

Para realizar pruebas a la implementación del sistema de simulación del SMR se crearon varios modelos 3D utilizando 3ds Max y se creó el *script* anexo en el listado A.6 para generar las partículas y los resortes de las instancias de prueba. Los modelos 3D exportados no contienen información sobre sus resortes (excepto por la dureza) sino que ésta se genera proceduralmente al cargarlos utilizando un procedimiento sencillo que depende de la distancia entre las partículas y el factor k , es decir, dados los valores de *minDistPart* y *maxDistPart* para un modelo, existirá un resorte entre las partículas **A** y **B** si se cumple el predicado de la Ec (3.1).

$$dist(A, B) \leq maxDistPart * k \wedge dist(A, B) \geq minDistPart \quad (3.1)$$

Es decir, existirá un resorte si las partículas están dentro de una distancia predefinida. El *script* que se implementó en 3ds Max exporta un valor de k para cada partícula según la tonalidad (de negro) que tenga en 3ds Max, así que una vez establecido los valores de mínimo y máximo, se puede controlar desde 3ds Max la conectividad que tendrá cada partícula simplemente pintando sobre el modelo: mientras más oscuro se pinte en una zona, menos conectada estará con las partículas cercanas y por ende será más suave en la simulación. Este método funciona bien y facilita enormemente la agregación de información de propiedades físicas a un modelo 3D. El k mostrado en la figura 3.1, que pertenece al resorte, se calcula como el promedio de los k en cada partícula.

La tabla 3.3 muestra los modelos de prueba que se construyeron y posteriormente imágenes de los mismos de la figura 3.1 a la 3.4.

Como columna adicional, en la tabla 3.3 está la relación entre número de vértices y número de

Tabla 3.3: Modelos de prueba

Nombre	Cantidad de Vértices	Cantidad de resortes	Relación
GeoEsfera	812	11.950	0.067
Teapot	1.598	58.946	0.027
Plano	2.601	157.858	0.016
Cochino	5.104	450.381	0.011

resortes, esto sirve como indicador para determinar la cantidad de paralelismo de datos disponible, si este valor aumenta es porque el modelo tiene más resortes distribuidos en menos partículas, por lo que el método de partición del conjunto de resortes tenderá a crear más subconjuntos (ver sección 2.2.1). Podemos ver en la tabla que mientras el tamaño del modelo aumenta, la relación vértices-resortes disminuye por lo que la cantidad de paralelismo de datos disponible disminuye también.

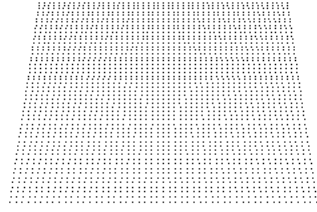
3.3 Medición de tiempo de ejecución

Se utilizó el reloj de alta resolución del sistema operativo para medir los tiempos de ejecución de cada versión. Se ejecutaron mil iteraciones de cada versión con cada uno de los cuatro modelos, y se promedió el tiempo de ejecución, también se calculó la desviación estándar σ para tener una idea de la variabilidad del mismo, se midió el tiempo de ejecución conjunta de la actualización de vértices y de resortes. Los resultados se muestran en la tabla 3.4.

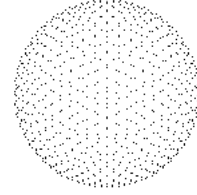
Tabla 3.4: Mediciones de tiempos de ejecución de los

Nombre	Tiempo CPU (ms)	Tiempo en GPU (ms)	Mejora
GeoEsfera	6.691 ($\sigma=0.80$)	1.430 ($\sigma=0.98$)	4.6X
Teapot	33.760 ($\sigma=1.75$)	3.764 ($\sigma=2.11$)	8.9X
Plano	76.350 ($\sigma=7.70$)	3.930 ($\sigma=1.88$)	19.4X
Cochino	249.770 ($\sigma=12.93$)	11.700 ($\sigma=1.33$)	21.3X

Se puede ver en la tabla 3.4 que mientras el tamaño del sistema aumenta, la mejora contra la versión de CPU se hace más importante, aún cuando en la sección anterior se mostró que la cantidad de paralelismo de datos disminuye, esto se explica quizá porque el *overhead* de la llamada se hace



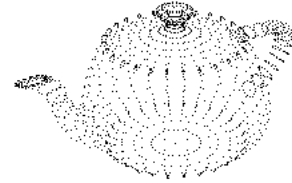
(a)



(b)



(c)



(d)

Figura 3.1: Modelos 3D utilizados para generar los SMR de prueba. (a) Plano, (b) GeoEsfera, (c) Cochino, (d) Teapot.

menos significativo mientras aumenta el tamaño del sistema, situación que se advierte en [21].

3.4 Factor S y dureza del cuerpo.

Se pudieron realizar experimentos con el factor S que se le agregó al modelo (ver sección 1.4) y se observó como este factor modifica la suavidad de los cuerpos para ajustar el sistema al material que se desea simular. La figura 3.2 muestra como ejemplo la GeoEsfera simulada con tres distintos valores de S y se pueden observar los cambios evidentes de suavidad en el cuerpo: mientras S disminuye, también lo hace la dureza del mismo.

Este nuevo elemento que se agregó durante el desarrollo del proyecto, ofrece un mayor nivel de

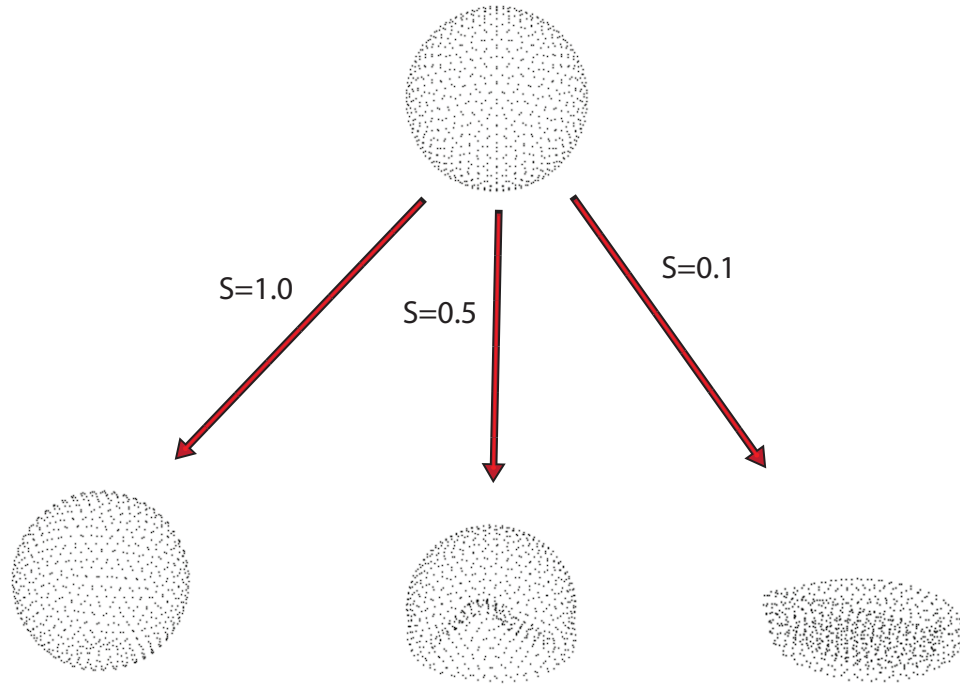


Figura 3.2: GeoEsfera simulada con distintos valores de S . La esfera superior no tiene transformaciones, las de abajo están colisionando con el suelo lo que las hace deformarse.

flexibilidad en cuanto a los materiales que se pueden simular. No solo permite modificar la suavidad de un cuerpo, sino que crea la oportunidad, por ejemplo, de crear cuerpos que modifiquen sus propiedades dinámicamente, es decir, se puede ajustar el factor S durante la simulación y hacer el cuerpo más o menos suave sin tener que cambiar la configuración de los resortes (agregar o eliminar resortes).

Es importante destacar que también, como se mencionó anteriormente, se puede modificar la suavidad del cuerpo modificando la cantidad de resortes en el sistema, si se agregan más resortes, el cuerpo tiende a ser más duro y viceversa. Para hacer el cuerpo lo más rígido posible, el sistema debe tener todas las partículas conectadas con todas, y el factor S con valor 1.

CONCLUSIONES Y RECOMENDACIONES

Este capítulo presenta las conclusiones obtenidas en el proceso de investigación, diseño e implementación de este trabajo, así como recomendaciones para trabajos futuros basados en él.

Conclusiones

Se logró desarrollar una implementación del sistema masa-resorte que funciona por completo en un GPU Nvidia, la cual mostró mejoras en algunos casos de más de 2000 % del rendimiento del código equivalente ejecutándose en un CPU de última generación. Se resolvió el problema de la paralelización del sistema masa-resorte con interconexión explícita entre partículas y se agregó un elemento que le aporta realismo a la simulación (la constante S) con un costo relativamente bajo (una multiplicación extra). Pudimos ver que este sistema se puede utilizar satisfactoriamente para simular cuerpos suaves mientras se realizan otras tareas en el CPU.

Luego del diseño de las estructuras de datos, la implementación fue relativamente fácil de crear, programar en C, un lenguaje de propósito general, hace el proceso mucho más ágil. También la posibilidad de manipular tipos de datos arbitrarios y no tener que transformar el espacio del problema a un espacio específico de computación gráfica provee el marco de trabajo necesario para crear una aplicación GPGPU en un tiempo relativamente corto.

Existen varios aspectos que se deben considerar antes de tratar de resolver un problema con una solución en GPU, no solo la cantidad de paralelismo de datos disponible es importante, sino la posibilidad de crear estructuras que exploten las capacidades de acceso a memoria del GPU, principalmente las lecturas unificadas, ya que estas aportan una mejora importante al rendimiento.

Aunque nuestra implementación fue hecha específicamente para los GPUs Nvidia, existen tecnologías (OpenCL, DirectCompute) con las cuales se pueden crear las mismas aplicaciones para otras marcas y modelos.

Durante el desarrollo de este trabajo, entre el 2009 y 2010, fueron lanzados al mercados GPUs

mucho más rápidos que el utilizado en este trabajo, así como productos específicamente desarrollados para aplicaciones de propósito general en el GPU (como las tarjetas Tesla de Nvidia, y la microarquitectura Larrabee de Intel [34]). Los modelos más eficientes de GPUs son capaces de procesar más de un billón de operaciones de punto flotante por segundo, mientras que la capacidad de los CPU se encuentra más por los cientos de millones de FLOPS, esto es una muestra de que la utilización de procesadores de muchos núcleos está siendo ampliamente considerada por la industria.

Sin duda la programación GPGPU es una nueva rama de la computación de alto rendimiento muy atractiva por su bajo costo y alto nivel de esparcimiento, existen GPUs relativamente potentes en millones de computadoras personales hoy en día, y las tarjetas más potentes cuestan generalmente menos de 1000 dólares estadounidenses.

Ya existen aplicaciones importantes que toman ventaja de el poder masivo de paralelización del GPU, estas van desde ramas de inteligencia artificial hasta la criptografía y el análisis climático, entre muchas otras [35]; y entre estas se han visto incrementos de velocidad similares o en algunos casos mejores que los obtenidos en este trabajo.

Recomendaciones

El sistema desarrollado en este trabajo establece una base que servirá para posibles trabajos futuros en alguna de las ramas de la computación tocadas por lo que aquí explicamos algunas recomendaciones pertinentes.

En cuanto al desarrollo de aplicaciones en el GPU, el diseño de la estructura de datos es sin duda el elemento más importante, los accesos a memoria suelen ser el cuello de botella y estos se reducen significativamente diseñando una estructura de datos que se ajuste a los requerimientos del GPU. Posiblemente las estructuras utilizadas en este trabajo pueden ser mejoradas para incrementar el rendimiento en cuanto accesos a memoria, principalmente la utilizada para representar resortes.

Por otro lado, el algoritmo de particionamiento en subconjuntos podría ser mejorado para minimizar el número de subconjuntos generados, y en consecuencia aumentar el paralelismo disponible. Quizá el coeficiente Gini, que mide el nivel de desigualdad existente en un conjunto, se podría utilizar para disminuir la desigualdad en los tamaños de los subconjuntos o simplemente para saber donde se pueden agregar resortes sin afectar significativamente el rendimiento.

Por último, en la implementación creada, se toma como datos de entrada un modelo de partículas y resortes que es utilizado para la simulación física, y en este caso utilizamos el mismo modelo para su representación visual. Existirán ocasiones donde se necesite un modelo visualmente complejo, pero una simulación física sencilla (por lo costoso de la misma). Una mejora importante podría ser la separación del modelo físico del modelo visual para así poder tener modelo visualmente complejos pero físicamente sencillos, es decir, que no exista necesariamente una relación 1:1 entre los vértices del modelo 3D y las masas del modelo físico.

BIBLIOGRAFÍA

- [1] K. Emmett and F. Randima, "The geforce 6 series gpu architecture," in *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, (Nueva York, NY, EEUU), p. 29, ACM, 2005.
- [2] A. "Jitterbug", "Pig." Disponible en internet: <http://www.turbosquid.com>. Consultado el 29 de Enero de 2010.
- [3] GPGPU.org, "Gpgpu.org." Disponible en internet: <http://gpgpu.org/about>. Consultado el 29 de Febrero de 2010.
- [4] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, April 1965.
- [5] Zeller and Cyril, "Cloth simulation on the gpu," in *SIGGRAPH '05: ACM SIGGRAPH 2005 Sketches*, (Nueva York, NY, EEUU), p. 39, ACM, 2005.
- [6] N. Corporation, "Physx." Disponible en internet: http://www.nvidia.com/object/physx_new.html. Consultado el 16 de Abril de 2010.
- [7] Sears, Zemansky, Young, and Freedman, *Física Universitaria*. Naucalpan de Juárez, México: Pearson Educación, 2004.
- [8] T. Jakobsen, "Advanced character physics." Disponible en internet: <http://www.teknikus.dk/tj/gdc2001.htm>, 2001. Consultado el 9 de Junio de 2009.
- [9] A. Witkin and D. Baraff, "Physically based modeling: Principles and practice." Disponible en internet: <http://www.cs.cmu.edu/~baraff/sigcourse/notesb.pdf>, 1997. Consultado el 12 de Enero de 2010.
- [10] N. Corporation, "Nvidia." Disponible en internet: <http://www.nvidia.com/>. Consultado el 16 de Abril de 2010.
- [11] A. M. Devices, "Amd." Disponible en internet: <http://www.ati.com/>. Consultado el 16 de Abril de 2010.

- [12] A. M. Devices, “Amd.” Disponible en internet: <http://www.amd.com/>. Consultado el 16 de Abril de 2010.
- [13] P. Pospichal and J. Jaros, “Gpu-based acceleration of the genetic algorithm.” Disponible en internet: www.gpgpgpu.com/gecco2009/7.pdf, 2009. Consultado el 9 de Junio de 2009.
- [14] T. Pock, Unger, M., Cremers, D., and H. Bischof, “Fast and exact solution of total variation models on the gpu,” in *CVGPU08*, 2008.
- [15] T. Grust, “Branch prediction, chapter 3.” Disponible en internet: <http://www-db.informatik.uni-tuebingen.de/files/teaching/ss09/dbcpu/dbms-cpu-3.pdf>. Consultado el 17 de Abril de 2010.
- [16] I. Corp., “Intel core 2 duo desktop processor.” Disponible en internet: http://download.intel.com/products/processor/core2duo/desktop_prod_brief.pdf. Consultado el 18 de Abril de 2010.
- [17] U. W. ”M. Mittal”, .A. Peleg”, “MmxTMtechnology architecture overview,” *Intel Technology Journal*, 3rd Quarter, 1997.
- [18] I. Corp., “Definición de sse2 y sse3.” Disponible en internet: <http://www.intel.com/support/sp/processors/sb/cs-030123.htm>. Consultado el 17 de Abril de 2010.
- [19] P. Seebach, “Unrolling altivec.” Disponible en internet: <http://www.ibm.com/developerworks/power/library/pa-unrollav1/>. Consultado el 18 de Abril de 2010.
- [20] M. Flynn, “Some computer organizations and their effectiveness,” *IEEE Trans. Comput.*, vol. C-21, pp. 948+, 1972.
- [21] NVIDIA, “Nvidia cuda programming guide 2.3.” Disponible en internet: http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide2009.pdf. Consultado el 29 de Enero de 2010.
- [22] K. Huizing and H.-W. Shen, “The graphics rendering pipeline.” Disponible en internet: <http://www.win.tue.nl/~keesh/ow/2IV40/pipeline2.pdf>. Consultado el 19 de Abril de 2010.
- [23] M. Corporation, “Microsoft directx.” Disponible en internet: <http://www.microsoft.com/games/en-US/aboutGFW/pages/directx.aspx>. Consultado el 19 de Abril de 2010.
- [24] J. Owens, “Streaming architectures and technology trends,” in *Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation* (F. Randima, ed.), pp. 457–470, Pearson Higher Education, 2005.

- [25] NVIDIA, “Nvidia gpu computer developer home page.” Disponible en internet:
<http://developer.nvidia.com/object/gpucomputing.html>. Consultado el 19 de Abril de 2010.
- [26] “Opencl.” Disponible en internet: <http://www.khronos.org/opencl/>. Consultado el 19 de Abril de 2010.
- [27] B. Hardwidge, “Nvidia has double ati’s desktop gpu market share.” Disponible en internet:
<http://www.bit-tech.net/news/hardware/2009/04/30/nvidia-increases-market-share/1>, 30 de Abril 2009. Consultado el 19 de Abril de 2010.
- [28] “Apple.” Disponible en internet: <http://apple.com>. Consultado el 19 de Abril de 2010.
- [29] “Microsoft corporation.” Diponible en internet: <http://microsoft.com>. Consultado el 19 de Abril de 2010.
- [30] NVIDIA, “Nvidia cuda c programming best practices guide.” Disponible en internet:
http://developer.download.nvidia.com/compute/cuda/3.0/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide
2009. Consultado el 29 de Enero de 2010.
- [31] NVIDIA, “Nvidia cuda reference manual.” Disponible en internet:
<http://developer.download.nvidia.com/compute/cuda/3.0/toolkit/docs/CudaReferenceManual.pdf>,
2009. Consultado el 29 de Enero de 2010.
- [32] “Microsoft dreamspark.” Disponible en internet:
<https://www.dreamspark.com/Products/product.aspx?productid=9>. Consultado el 19 de Abril de 2010.
- [33] “Opengl - the industry standard for high performance graphics.” Disponible en internet:
<http://opengl.org>. Consultado el 19 de Abril de 2010.
- [34] “Intel corporation.” Disponible en internet: <http://intel.com>. Consultado el 19 de Abril de 2010.
- [35] NVIDIA, “Cuda showcase.” Disponible en internet:
http://www.nvidia.com/object/cuda_showcase_stage.html. Consultado el 19 de Abril de 2010.

APÉNDICE A

LISTADOS DE CÓDIGO

En esta sección están añadidos algunos listados de código referenciados en el trabajo.

A.1 Algoritmo de partición de resortes en subconjuntos paralelizables

```
1      conjuntosResortes = vector<vector<Resorte>>>();
2      particlesAlreadyInSet = bool[particleCount];
3      springsAlreadyInSomeSet = bool[springCount];
4      working = true;
5      while (working)
6      {
7          working = false;
8          memset(particlesAlreadyInSet, 0, particleCount * sizeof(
9              bool));
10         set vector<Resorte>();
11
12         for (int i = 0; i < springCount; i++)
13         {
14             // Si el resorte ya esta en algun set, lo
15             // ignoramos.
16             if (springsAlreadyInSomeSet[i]) continue;
17
18             spring = springs[i];
19             p1 = springs[i].p1;      // Indices de las
20             // particulas a
21             p2 = springs[i].p2;      // cada lado del resorte
22
23             // Si ya hay un resorte que afecte alguna de estas
24             // particulas, continuamos.
25             if (particlesAlreadyInSet[particleIndex1] ||
26                 particlesAlreadyInSet[particleIndex2])
27             {
28                 continue;
29             }
30             particlesAlreadyInSet[p1] = true;
31             particlesAlreadyInSet[p2] = true;
```

```

28         working = true;
29         springsAlreadyInSomeSet[i] = true;
30         // Metemos el resorte en el set actual.
31         set.push(spring);
32     }
33
34     // Guardamos este conjunto de resortes.
35     springSets.push_back(set);
36 }

```

A.2 Algoritmo del tamaño del grid y llamada al kernel. Primer Enfoque.

```

1  void DistanceConstraint::Execute(void *pos, void *springs, unsigned int
   springCount, int iterations, float dt)
2  {
3      dim3 block(clampValue(springCount, 0, 256), 1, 1);
4      int gridX = springCount / block.x;
5      if ((springCount % block.x) > 0)
6          gridX++;
7      dim3 grid(gridX, 1, 1);
8
9      ExecuteDistanceConstraint(block, grid, (float4 *)pos, (float4 *)
   springs, springCount, iterations, dt);
10 }
11
12 extern "C"
13 void ExecuteDistanceConstraint(dim3 block, dim3 grid, float4* pos, float4*
   springs, unsigned int springCount, int iterations, float dt)
14 {
15     DistanceConstraint<<<grid, block>>>(pos, springs, springCount, dt);
16
17     // Revisamos si hubo un error.
18     cudaError_t error = cudaGetLastError();
19     if (error != cudaSuccess) {
20         printf("DistanceConstraint() failed to launch error = %d\n", error
   );
21     }
22 }

```

A.3 Algoritmo del tamaño del grid y llamada al kernel. Segundo Enfoque.

```

1  void DistanceConstraint::ExecuteBatched(void *pos, void **springArrays,
   void *springArraySizes, unsigned int springArrayCount, int
   largestArraySize, int iterations, float dt)
2  {
3      dim3 block(clampValue(largestArraySize, 0, 256), 1, 1);
4      int gridX = largestArraySize / block.x;

```

```

5
6     if ((largestArraySize % block.x) > 0)
7         gridX++;
8     dim3 grid(gridX, 1, 1);
9
10    ExecuteBatchedDistanceConstraint(block, grid, (float4 *)pos, (float4
11        **)springArrays, (int *)springArraySizes, springArrayCount,
12        iterations, dt);
13
14    }
15
16    extern "C"
17    void ExecuteBatchedDistanceConstraint(dim3 block, dim3 grid, float4 *pos,
18        float4 **springArrays, int *springArraySizes, unsigned int
19        springArrayCount, int iterations, float dt)
20    {
21        BatchedDistanceConstraint<<<grid, block>>>(pos, springArrays,
22            springArraySizes, springArrayCount, dt);
23
24        // Check for errors
25        cudaError_t error = cudaGetLastError();
26        if (error != cudaSuccess) {
27            printf("DistanceConstraint() failed to launch error = %d\n", error
28                );
29        }
30    }
31
32    }
33

```

A.4 Kernel de actualización de resortes. Primer Enfoque.

```

1    extern "C" __global__ void DistanceConstraint(float4 *posArray, float4 *
2        springArray, const int springCount, const float dt)
3    {
4        unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;
5
6        if (i < springCount)
7        {
8            float4 spring = springArray[i];
9            int index1 = __float_as_int(spring.x);
10            int index2 = __float_as_int(spring.y);
11            float restLength = spring.z;
12            float k = spring.w;
13
14            float4 x1 = posArray[index1];
15            float4 x2 = posArray[index2];
16            float4 delta = x2 - x1;
17            float deltaLength = sqrt(delta.x * delta.x + delta.y * delta.y +
18                delta.z * delta.z);
19            float diff = (deltaLength - restLength)/ deltaLength;
20            x1 = x1 + delta * 0.5f * diff;
21            x2 = x2 - delta * 0.5f * diff;
22
23            posArray[index1] = x1;
24            posArray[index2] = x2;
25        }
26    }
27

```

```

23     }
24 }

```

A.5 Kernel de actualización de resortes. Segundo Enfoque.

```

1  extern "C" __global__ void BatchedDistanceConstraint(
2      float4 *posArray,
3      float4 **springArrays,
4      int *springArraySizes,
5
6      const int springArrayCount,
7      const float dt)
8  {
9      unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;
10
11     // Idea: the springArraySizes could be loaded into shared memory first
12     // or maybe each element could be loaded by the first thread only.
13     for (int j = 0; j < springArrayCount; j++)
14     {
15         __syncthreads();
16         // Cantidad de resortes en este set.
17         int springCount = springArraySizes[j];
18         if (i < springCount)
19         {
20             float4 spring = springArrays[j][i];
21             int index1 = __float_as_int(spring.x);
22             int index2 = __float_as_int(spring.y);
23             float restLength = spring.z;
24             float k = spring.w;
25
26             float4 x1 = posArray[index1];
27             float4 x2 = posArray[index2];
28             float4 delta = x2 - x1;
29             float deltaLength = sqrt(delta.x * delta.x + delta.y * delta.y
30                                     + delta.z * delta.z);
31             float diff = (deltaLength - restLength) / deltaLength;
32             x1 = x1 + delta * 0.5f * k * diff;
33             x2 = x2 - delta * 0.5f * k * diff;
34
35             posArray[index1] = x1;
36             posArray[index2] = x2;
37         }
38     }

```

A.6 Código MaxScript para exportar modelos con información física de 3dsMax 2009


```

1  -- Author: Juan Campa (2009)
2
3  global outputDir = @"C:\Users\Juan\Documents\Tesis\src\Visualizator2\
   Content\"
4  global GetVertexColor
5
6  -- Settings
7  global defaultK = 1
8
9  fn Error msg = ( MessageBox msg );
10
11 fn ExportMesh o exportData: #(#verts, #faces, #color) =
12 (
13     if (o == undefined) then ( Error "Undefined mesh passed"; return
   () )
14     if (o.Name == undefined) then ( Error "Undefined mesh passed";
   return() )
15
16     tmesh = snapshotAsMesh o
17     out_name = outputDir + o.Name;
18     out_file = createfile out_name
19     num_verts = tmesh.numverts
20     num_faces = tmesh.numfaces
21
22     if (findItem exportData #verts != 0) then
23         format "%\n" num_verts to:out_file
24     else
25         format "%\n" 0 to:out_file
26     if (findItem exportData #faces != 0) then
27         format "%\n" num_faces to:out_file
28     else
29         format "%\n" 0 to:out_file
30
31     if (findItem exportData #verts != 0) then
32         for v = 1 to num_verts do
33             (
34                 local vertexCol
35                 --try
36                 (
37                     vertexCol = GetVertexColor tmesh v
38                 )
39                 --catch
40                 --(
41                 --     print "Using default K."
42                 --     vertexCol = (color defaultK 0 0)
43                 --)
44                 vert = getVert tmesh v
45                 format "% % %\n" vert.x vert.z -vert.y (
   vertexCol.r/255.0) to:out_file
46             )
47     if (findItem exportData #faces != 0) then
48         for f = 1 to num_faces do
49             (
50                 face = getFace tmesh f
51                 format "% % %\n" (face.x as integer) (face.y as
   integer) (face.z as integer) to:out_file

```

```

52         )
53         close out_file
54         delete tmesh
55
56         edit out_name
57     )
58
59     fn GetVertexColor o index =
60     (
61         if (classof o == TriMesh) then
62             tmesh = o
63         else
64             tmesh = snapshotAsMesh o
65
66         local faceCount = getNumFaces tmesh
67         local faceIndex
68         local position -- x, y or z
69         if (getNumCPVVerts tmesh == 0) then
70             throw "No color associated with this vertex"
71         for i = 1 to faceCount do
72             (
73                 faceIndex = i
74                 face = getFace tmesh i
75                 if (face.x == index) then
76                     (
77                         position = 1
78                         exit
79                     )
80                 else if (face.y == index) then
81                     (
82                         position = 2
83                         exit
84                     )
85                 else if (face.z == index) then
86                     (
87                         position = 3
88                         exit
89                     )
90             )
91         )
92
93         colorFace = getVCFace tmesh faceIndex
94         if (position == 1) then
95             return getVertColor tmesh colorFace.x
96         if (position == 2) then
97             return getVertColor tmesh colorFace.y
98         if (position == 3) then
99             return getVertColor tmesh colorFace.z
100        else
101            throw "No color associated with this vertex"
102    )

```