UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

IMPLEMENTACIÓN DE UN MODELO DEFORMABLE DE MASAS Y RESORTES EN UNA GPU PARA UN SIMULADOR DE CIRUGÍA DE PRÓSTATA

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

INGENIERO EN COMPUTACIÓN

PRESENTA:

ERIC LIRA BERRA

DIRECTOR DE LA TESIS:
M. C. MIGUEL ÁNGEL PADILLA CASTAÑEDA

MÉXICO D. F. JUNIO DE 2008

Agradecimientos

Agradezco a todas las personas, familiares, amigos, profesores, quienes de una forma u otra contribuyeron a la realización de este trabajo.

Agradezco a mi familia, por darme el apoyo incondicional, el ánimo y la seguridad para terminar lo que parecía no llegar a su fin.

Agradezco a mi director de tesis, M. C. Miguel Ángel Padilla Castañeda, por guiarme y asesorarme, por su paciencia y confianza.

Agradezco a la UNAM y a la Facultad de Ingeniería, por abrirme sus puertas y ser la sede de mi formación.

Gracias a mis amigos y compañeros de la facultad. A Jorge Luis y Raúl, quienes han sido y siguen siendo mis mejores amigos. A Irene, Carmen, Ericka, Rosaura, Héctor, Rafael y Paco, quienes me acompañaron en gran parte de la carrera, viviendo muchos momentos y experiencias, algunas más agradables que otras.

Gracias al Dr. Jorge Márquez Flores y al Dr. Fernando Arámbula Cosío, así como al CCADET y al Laboratorio de Imágenes y Visualización, en donde me brindaron los medios y los recursos necesarios para llevar a cabo esta tesis.

Agradezco a Isis, Manuel, Jorge y Jair, quienes han compartido conmigo el gusto por la música, pasando momentos de alegría y de tristeza, demostrando ser unos grandes amigos con los que he aprendido mucho y he logrado cosas que no hubiera imaginado.

Agradezco a Nancy, por estar conmigo y motivarme, porque en poco tiempo se ha convertido en una persona muy importante para mí, y que ha logrado hacerme experimentar gran variedad de emociones y sentimientos.

A mis padres…

Por su apoyo y motivación para la realización de mis estudios. Por su comprensión y paciencia. Por la educación que me han dado, la cual ha sido motivo de superación.

A mis hermanas…

Mariana y Miriam, quienes me han visto crecer y me conocen como nadie más. Por tener la confianza de que algún día terminaría esta tesis.

Índice General

Resumen

Los simuladores de intervenciones quirúrgicas son una herramienta que ofrece ciertas ventajas cuando se utilizan para fines didácticos, pues el entrenamiento de los futuros cirujanos puede realizarse de una forma más práctica, segura y económica que con otros métodos alternativos. Es importante que el simulador cuente con el mayor realismo posible para que sea verdaderamente funcional. Ésto implica que la construcción debe enfocarse a lograr los menores tiempos de ejecución posibles.

El desarrollo de un simulador de cirugía de próstata implica diversas etapas, entre las cuales el cálculo de deformaciones es entendido como un proceso que requiere de un gran tiempo de cálculo. Es por ello que en este trabajo se presenta una alternativa para el cálculo de dicho proceso, la cual consiste en aprovechar el poder de cálculo que han alcanzado los procesadores gráficos. De esta forma, el cómputo que realiza el CPU se reubica para ser ejecutado por un dispositivo de procesamiento de gráficos.

Como parte adicional de este trabajo de tesis, se utiliza la programación multihilos para separar los procesos de simulación independientes y así lograr que éstos trabajen de una forma concurrente.

Los resultados obtenidos muestran que la ejecución de algoritmos por parte de la tarjeta gráfica y la utilización de la programación multihilos disminuyen de forma significativa los tiempos de respuesta del proceso de simulación, propiciando que la calidad visual del simulador de cirugía de próstata mejore considerablemente.

Tabla de abreviaturas

API	Interfaz de programación de aplicaciones (Application Programming Interface)
CPU	Unidad de procesamiento central (Central Processing Unit)
CUDA	Arquitectura de dispositivos de cómputo unificado (Compute Unified Device Architecture)
GPGPU	Cómputo de propósito general utilizando dispositivos gráficos (General-Purpose Computation Using Graphics Hardware)
GPU	Unidad de procesamiento de gráficos (Graphics Processing Unit)
RTUP	Resección Transuretral de la Próstata

Capítulo 1

Introducción

En este capítulo se presenta un panorama del entorno que incita el desarrollo de este trabajo de tesis. La motivación del trabajo se genera a partir de las necesidades involucradas en la construcción de un sistema de simulación de cirugía de próstata, ya que para un correcto funcionamiento de este dispositivo se requiere que las diversas etapas que lo componen trabajen de una forma eficiente, proporcionando el realismo suficiente para ser utilizado como una herramienta de utilidad. En esta tesis se trabaja con la optimización de algunas de las etapas del proceso de simulación a partir de su ejecución en una tarjeta aceleradora de gráficos. Es por ello que en primer lugar se hace una descripción de dicho simulador, mostrando sus características, modo de operación, así como los avances alcanzados en su construcción. Además, se hace mención de las causas que incitan su construcción y los métodos que se han utilizado para su implementación. Enseguida, se hace una revisión de algunos trabajos previos relacionados, enfocándose en la utilización de tarjetas gráficas como aceleradoras de cálculos intensos. A partir de lo anterior se formula el objetivo de esta tesis, así como el método que será utilizado para la resolución del problema descrito.

1.1. Simulador de cirugía de próstata

El procedimiento estándar utilizado para tratar la Hiperplasia Benigna de la Próstata es la Resección Transuretral de la Próstata (RTUP). Los métodos tradicionales de entrenamiento para dicha cirugía se realizan *in-vivo* durante intervenciones reales, con el mecanismo de aprendizaje maestro-alumno. Debido a ello, las oportunidades que tiene el aprendiz para practicar son muy limitadas y en ocasiones se recurre a modelos artificiales que suelen ser costosos y no satisfacen las condiciones de realismo suficientes que el cirujano aprendiz residente de urología requiere para su correcto entrenamiento.

Una alternativa se basa en el desarrollo de un sistema de simulación por computadora para entrenamiento de cirugía que brinda a los residentes en urología

la oportunidad de practicar un número ilimitado de veces, sin riesgo para los pacientes; además, dicho entrenamiento resulta una forma económica, antes de practicar la cirugía con pacientes reales [24].

El grupo de Análisis de Imágenes y Visualización del Centro de Ciencias Aplicadas y Desarrollo Tecnológico (CCADET) de la UNAM está trabajando en el desarrollo un simulador de resección transuretral de próstata, el cual consiste en un ambiente virtual con un modelo en 3D de la próstata que permite simular las interacciones inherentes a la cirugía, como lo son las deformaciones y resecciones del tejido blando. Estas interacciones son provocadas por un resectoscopio, que es un instrumento quirúrgico con el cual se realiza la intervención. Dicho instrumento ha sido recreado a través de una interfaz electromecánica pasiva[1], semejante a un resectoscopio real, que interactúa con el modelo virtual.

Para obtener una simulación efectiva de los movimientos más importantes del cirujano durante la RTUP se diseñó un mecanismo basado en un arreglo de discos y anillos, con el cual se reproducen cinco de los siete grados de libertad que se presentan en la cirugía. Tres de esos ejes son rotacionales y los otros dos son desplazamientos lineales del resectoscopio. Este dispositivo cuenta con sensores ópticos que monitorean los movimientos rotacionales y lineales de la herramienta de cirugía a través de microcontroladores de sensado en tiempo real [24].

Por otro lado, se cuenta con el modelo virtual que simula las deformaciones del tejido provocadas por la interacción con los instrumentos quirúrgicos. En dicho modelo se representa al tejido como una malla discreta en 3D. De esta forma, el comportamiento deformable es obtenido considerando a la malla como un arreglo de nodos con masa puntual que se encuentran interconectados con sus vecinos por medio de resortes [22].

El modelo deformable de la próstata fue construido a partir de un conjunto de imágenes de ultrasonido, de las cuales se anotaron los contornos por un especialista en ultrasonido. Cada uno de los contornos fue muestreado y a partir de estas muestras, utilizando interpolación, se obtiene el modelo de la próstata como una malla de tetrahedros que representa un cuerpo sólido. Finalmente, este cuerpo sólido se estructura como una malla de masas y resortes, en donde cada nodo en el cuerpo es conectado con sus nodos adyacentes.

Para llevar a cabo la interacción entre la herramienta mecatrónica y el modelo virtual se utiliza una comunicación serial, en donde los microcontroladores trabajan en paralelo monitoreando los movimientos en tiempo real y el sistema envía dicha información en forma de comandos al modelo virtual. En la Figura 1.1 se pueden observar ambos elementos, los cuales que conforman al simulador de cirugía de próstata.

[1] La interfaz mecatrónica no genera retroalimentación de fuerzas.

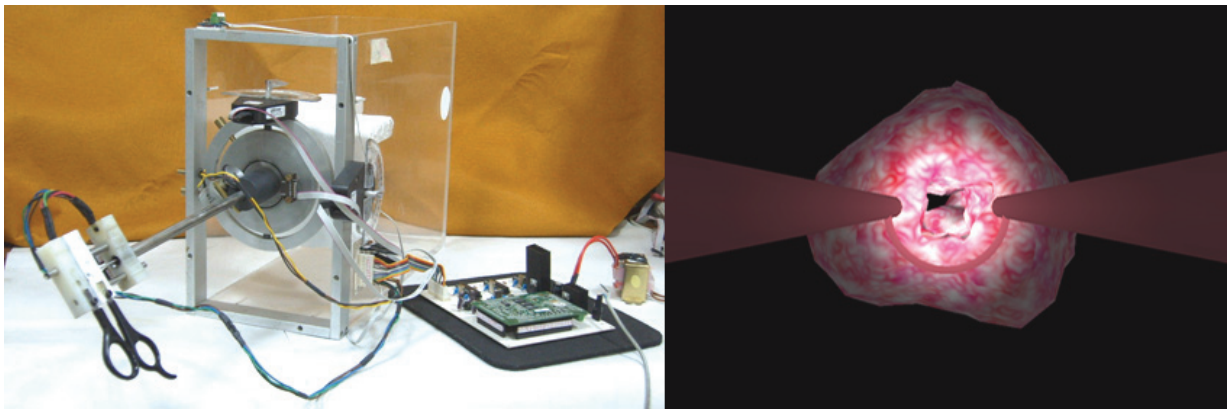

Figura 1.1. *Dispositivo mecatrónico y modelo virtual.*

1.2. Antecedentes

La utilización de tarjetas gráficas para aplicaciones de propósito general ha sido tema de interés en años recientes y existen numerosas investigaciones al respecto. Uno de los principales trabajos realizados en el campo de cómputo genérico con GPU es el desarrollado por el Laboratorio de Computación Gráfica de la Universidad de Stanford [30], pues además de contar con proyectos de investigación relacionados con la generación de imágenes, espacios interactivos y algoritmos para aplicación de efectos visuales, también han generado uno de los más famosos lenguajes para la programación genérica de tarjetas gráficas: BrookGPU.

Por otro lado, la compañía norteamericana NVIDIA ha desarrollado tarjetas gráficas programables que mejoran sus características en cada nueva entrega. Ésto propició la aparición de otros lenguajes de programación, igualmente creados por NVIDIA. En primer lugar se encuentra el lenguaje Cg, diseñado para la programación de efectos visuales utilizando tarjetas gráficas. Más recientemente ha aparecido la tecnología CUDA, que abre muchas posibilidades en cuanto a cómputo genérico con GPUs, pues elimina la necesidad de programar aplicaciones dentro de un contexto inmerso en el mundo de los gráficos, además de presentar mejoras en el acceso a la memoria [2].

Entre los trabajos realizados en el ámbito de simulación de procesos físicos utilizando tarjetas gráficas se encuentra el desarrollado en la Universidad de Carolina del Norte, donde presentan un proyecto enfocado a la simulación del comportamiento de las nubes en tiempo real [10]. Aquí, utilizan el hardware gráfico para realizar los cálculos relacionados a la dinámica de las nubes, que implica movimiento en el espacio, temperatura, presión, humedad, así como las interacciones con la luz, entre otros.

Si bien hay avances en el campo de cómputo genérico con GPUs, aún existen muchas posibilidades que prometen mayores logros, los cuales pueden verse reflejados en diversas aplicaciones. En esta tesis se trabaja con la utilización del cómputo genérico de GPUs aplicado a la construcción de simuladores quirúrgicos, aprovechando los recursos existentes y tomando en cuenta futuras implementaciones.

1.3. Descripción del problema

El simulador de próstata antes descrito se compone básicamente de dos partes: la herramienta mecatrónica y el modelo deformable virtual. El proceso general de simulación consta de diversas etapas, entre las cuales se encuentran el monitoreo de los movimientos de la herramienta quirúrgica, el envío de dicha información a través de comunicación serial, la lectura de los movimientos por parte del modelo virtual, el cálculo de colisiones, deformaciones y la generación de la escena final. Este proceso se realiza en aproximadamente 80 milisegundos, que corresponden a una tasa de 12.5 cuadros por segundo. Para obtener una escena con suficiente realismo visual es recomendable trabajar con 24 cuadros por segundo. Así, el problema consiste en enfocarse en las etapas que implican mayor tiempo de ejecución y disminuirlos de manera que el simulador sea capaz de trabajar en tiempo real con la suficiente calidad visual. Esta tesis se dirige principalmente a la etapa de cálculo de deformaciones, la cual es entendida como una de las etapas de mayor costo en cuanto a cómputo.

1.4. Objetivo

De acuerdo a lo establecido anteriormente, el objetivo de esta tesis es implementar un método de masas y resortes en una unidad de procesamiento de gráficos que sea capaz de realizar el cálculo de las deformaciones de tejido blando en tiempo real para un simulador de cirugía de próstata.

1.5. Método

Para disminuir el tiempo de cálculo que implica el método de masas y resortes, hasta el momento ejecutado en un procesador central, se pretende aprovechar el creciente poder de cálculo y la flexibilidad en la programación observados en las

últimas generaciones de GPUs, considerando también que la naturaleza paralelizable del método contribuye en gran medida a dicha implementación.

Por otro lado, siguiendo con el afán de mejorar los tiempos de ejecución, se implementará también un sistema de multihilos a manera de lograr un paralelismo entre las etapas que conforman el proceso de simulación.

1.6. Presentación de la tesis

A continuación se presenta el desarrollo de un modelo deformable de masas y resortes implementado para su ejecución en una tarjeta gráfica.

En el capítulo 2 se hace una revisión de los fundamentos teóricos en los que se basa la tesis, enfocándose en la descripción del método de masas y resortes, el modelo de programación para tarjetas gráficas y finalmente la programación multihilos.

En el capítulo 3 se reporta la implementación del método de masas y resortes. Se presentan las pruebas preeliminares realizadas sobre una tarjeta gráfica, la medición de su desempeño con respecto a un procesador central, y la capacidad de utilizar hardware gráfico para aplicaciones de propósito general. Enseguida, se presenta la programación del método de masas y resortes en una tarjeta gráfica, mostrando las técnicas y rutinas necesarias para dicho fin. De igual forma, se muestra la implementación en una GPU del método de diferenciación numérica de Newton-Euler. Por último, se presenta la paralelización de las etapas comprendidas en el proceso de simulación para lograr mayor eficiencia basándose en la aplicación de la programación multihilos.

El capítulo 4 muestra los resultados obtenidos tanto para la programación del método de masa y resortes en una GPU, como para la utilización de la programación multihilos para lograr paralelismo. Aquí mismo se exhiben diferentes configuraciones considerando la programación en GPU y multihilos mostrando los niveles de beneficio obtenidos en cada caso.

En el capítulo 5 se encuentran las conclusiones a las que se llega con el desarrollo de esta tesis, además de establecer las posibilidades futuras para el mejoramiento del trabajo realizado.

Capítulo 2

Fundamentos teóricos

En este capítulo se presenta la revisión bibliográfica de los aspectos teóricos en los cuales está cimentado el desarrollo de esta tesis. Se aborda en primer lugar la definición del método de masas y resortes, sus usos más prácticos, así como sus ventajas y desventajas. Enseguida, se hace una descripción de la arquitectura y el modelo que se utiliza para la programación de tarjetas gráficas, así como su evolución y algunos aspectos del porqué de su rápido crecimiento. Además, se muestran varios de los trabajos que se han realizado en el área de programación genérica de tarjetas gráficas, dando un panorama general de los avances en dicho ámbito. Por último, se presentan algunas generalidades de la programación multihilos, mostrando la conveniencia de su utilización como un medio para lograr mayor eficiencia en determinadas aplicaciones.

2.1. Método de masas y resortes

El modelado de tejido blando para sistemas de simulación de cirugía asistida por computadora involucra varios retos. Uno de ellos se refiere a los tejidos blandos, los cuales tienen propiedades biomecánicas complejas que pueden ser difíciles de simular. Los modelos matemáticos destinados a reproducir la naturaleza del tejido deben tener un comportamiento aceptable con respecto a la realidad. Además, las deformaciones de los tejidos deben calcularse en tiempo real. Uno de los métodos más frecuentemente usados para simulación de tejido blando es el método de masas y resortes, pues es relativamente sencillo de implementar y además facilita los cálculos en tiempo real [23].

Contando con una representación geométrica del cuerpo deformable, cada vértice en la malla está sujeto a fuerzas elásticas que lo unen con los demás vértices. Por otra parte, cada vértice puede ser influido por fuerzas externas como gravedad, presión, etc. Dichas fuerzas externas también intervienen en la deformación del cuerpo. De acuerdo a ésto, el tejido se representa con una malla visco-elástica en

3D, la cual está conformada por nodos de masa puntual, interconectados con sus nodos adyacentes por medio de resortes virtuales.

El comportamiento del cuerpo deformable corresponde a un comportamiento dinámico, en el cual están presentes características físicas como masa, elasticidad y viscosidad, todo ésto regido por una ecuación diferencial de segundo orden.

$$m_i \frac{\mathrm{d}^2 \mathrm{x}_i}{\mathrm{d}t^2} + \gamma_i \frac{d\mathrm{x}_i}{dt} + \mathrm{g}_i(t, \mathrm{x}_i) = \mathrm{f}_i(t, \mathrm{x}_i) \qquad (2.1)$$

Aquí x_i denota las coordenadas cartesianas del nodo *i*, m_i la masa del nodo *i*, γ_i el coeficiente de viscosidad, g_i la suma de todas las fuerzas internas en el nodo *i*, y f_i todas las fuerzas externas en el nodo *i* [18].

Para el cálculo de la fuerza elástica se considera la suma de todas las fuerzas internas que actúan en el nodo *i*, para lo cual se utiliza el siguiente conjunto de ecuaciones:

$$g_i = \sum s_k \,, \qquad (2.2)$$

$$s_k = \frac{c_k \cdot e_k}{\|r_k\|} \cdot r_k \,, \qquad (2.3)$$

$$e_k = \|r_k\| - l_k \,, \qquad (2.4)$$

$$r_k = x_j - x_i \,, \qquad (2.5)$$

Donde s_k es la fuerza interna en el nodo *i* causada por el resorte *k*, c_k denota el coeficiente de elasticidad del resorte *k*, e_k es el valor absoluto de la deformación desde su posición inicial (l_k), r_k es el vector de distancia entre el nodo *i* y su vecino *j*, $\|r_k\|$ es el valor absoluto de la distancia del resorte *k*, y x_j denota la posición cartesiana del vector para el nodo *j*, el cual está conectado por el resorte *k* con el nodo *i* [23].

Entre las ventajas que se obtienen al trabajar con este método se encuentran la posibilidad de asignar diferentes propiedades mecánicas a diferentes partes del modelo geométrico; además, el modelo permite realizar cortes y suturas, removiendo o agregando conexiones entre vértices. Aunque, por otro lado, si el número de resortes por vértices es muy pequeño, el sistema eventualmente podrá caer en mínimos locales no deseados; si son muchos los vértices, el sistema tiende a reducir el grado de deformación. Además, las deformaciones producidas no son fácilmente comparables a las definidas en estudios biomecánicos, ya que la elasticidad no está definida desde el punto de vista de mecánica del medio contínuo, por lo que para deformaciones grandes el sistema tiene poca precisión. Para aproximar el comportamiento a un modelo elástico no lineal se deben variar los valores de los parámetros, lo cual puede ser difícil y además el sistema podría volverse inestable.

2.2. Arquitectura y modelo de programación GPU

Una GPU es un dispositivo dedicado al procesamiento de gráficos que surge como una opción para aligerar la carga de trabajo del procesador central en aplicaciones como videojuegos, navegadores virtuales, ambientes en 3D interactivos, etc.

La arquitectura de una GPU se puede ver como una línea de ensamble con diferentes etapas en donde se aplican operaciones gráficas. Los primeros modelos de GPUs sólo eran capaces de manejar un número restringido de operaciones gráficas, las cuales estaban predefinidas en un proceso riguroso y muy poco flexible. Un factor determinante en el desarrollo de las actuales arquitecturas de GPUs fue la aparición de tarjetas gráficas con características programables, pues con ello se ofrecía la posibilidad de ejecutar código de usuario. Como consecuencia, los usuarios, y en particular programadores gráficos, podían escribir sus propias operaciones gráficas en un lenguaje de bajo nivel [34].

En este apartado se presenta una descripción de las transformaciones que se han presentado en el desarrollo de dispositivos procesadores de gráficos y los motivos que han incitado tales cambios. Además, se explica el proceso general sobre el cual trabajan las tarjetas gráficas y las operaciones internas que se realizan dentro de ellas. Por otra parte, se presentan las dificultades que conlleva la rápida evolución de las características de las GPUs, haciendo obvia la necesidad de programación en alto nivel, y por lo tanto la aparición de nuevos lenguajes diseñados para utilizar las características de los procesadores gráficos de una manera eficiente.

2.2.1. Evolución de los dispositivos gráficos

El hardware gráfico ha avanzado a pasos muy grandes. Existen tres factores principales que motivan este ritmo de innovación, como se muestra en la Figura 2.1. En primer lugar, la industria de los semiconductores se ha comprometido a duplicar cada 18 meses el número de transistores que se alojan en un microchip. Este constante crecimiento de poder de cómputo, conocido como Ley de Moore, provoca la aparición de hardware gráfico más rápido y barato [27].

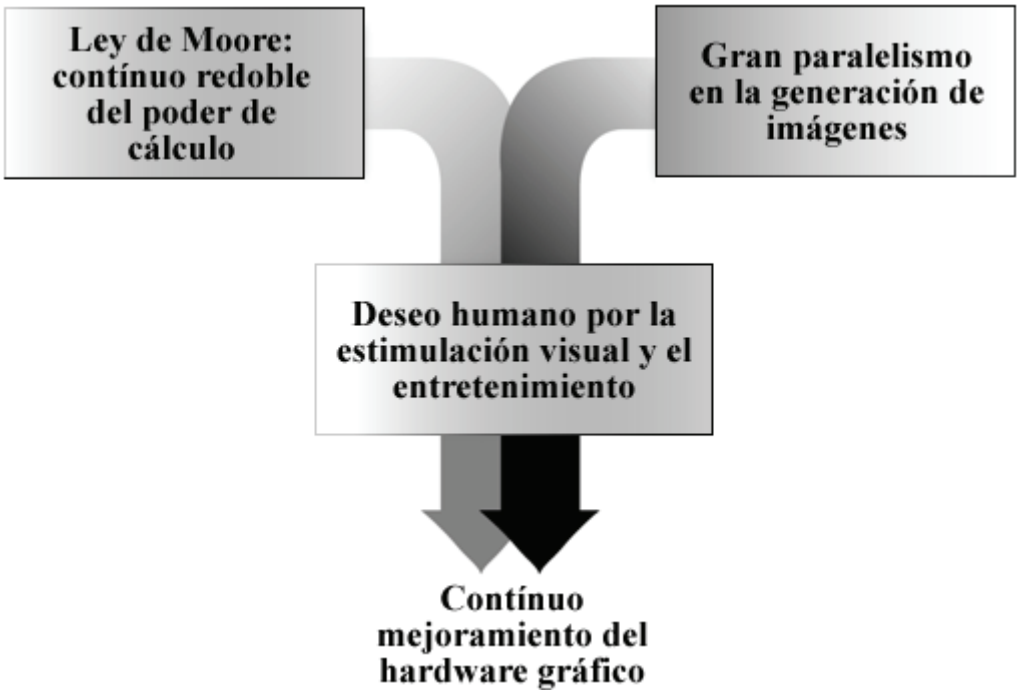

Figura 2.1. *Razones que incitan la innovación en los dispositivos gráficos [27].*

La segunda razón es la gran cantidad de cómputo requerida para simular el mundo que nos rodea. Es poco probable que se pueda alcanzar un punto en el que los gráficos por computadora lleguen a sustituir a la realidad, pero los practicantes de la computación gráfica siguen empeñados y han logrado grandes avances para cumplir dicho reto. Por fortuna, los diseñadores de dispositivos gráficos pueden repetidamente separar el problema de crear imágenes realistas en subproblemas que son más pequeños y fáciles de atacar [27].

El tercer factor es el deseo sostenido que todos tenemos de ser estimulados y entretenidos visualmente. Esta es la fuerza que conecta la fuente del contínuo crecimiento de poder de cómputo con la tarea de alcanzar un ambiente visual cada vez más realista.

Estas tres fuerzas son primordiales en la innovación de dispositivos gráficos. Como reflejo de todo ello se encuentra la demanda en el mercado de videojuegos, pues ésta ha incitado el desarrollo de mejores y más rápidas arquitecturas de GPUs.

2.2.2. Generaciones de GPUs

A mediados de la década de los 90s, los dispositivos gráficos más rápidos se componían de múltiples chips que trabajaban juntos para renderizar[2] imágenes y desplegarlas en pantalla. Los más complejos sistemas de computación gráfica se integraban por docenas de chips colocados en varias tabletas. Con el paso del tiempo la tecnología en semiconductores mejoró. Se logró incorporar la funcionalidad de complicados diseños multichip dentro de un solo chip gráfico. Este desarrollo resultó en una gran economía de integración y escala [27]. NVIDIA introdujo el término "GPU" a finales de 1990 cuando el término "controlador VGA" ya no coincidía con la descripción del hardware gráfico en una PC.

Las GPUs ahora superan a los CPUs en el número de transistores presentes en cada microchip. Por ejemplo, Intel construyó el procesador Pentium IV de 2.4 GHz. con 55 millones de transistores; NVIDIA usó cerca de 125 millones de transistores en la tarjeta GeForce FX [27].

Generación	Año	Nombre	Número de Transistores [1]	Tasa de relleno *antialiasing* [pixeles/s]	Tasa de dibujo de polígonos [triángulos/s]
Primera	Finales de 1998	RIVA TNT	7 M	50 M	6 M
Primera	Principios de 1999	RIVA TNT2	9 M	75 M	9 M
Segunda	Finales de 1999	GeForce 256	23 M	120 M	15 M
Segunda	Principios de 2000	GeForce2	25 M	200 M	25 M
Tercera	Principios de 2001	GeForce3	57 M	800 M	30 M
Tercera	Principios de 2002	GeForce4 Ti	63 M	1200 M	60 M
Cuarta	Principios de 2003	GeForce FX	125 M	2000 M	200 M

Tabla 2.1. Características y evolución de GPUs de NVIDIA.

Dentro de la industria se han definido cuatro generaciones de GPUs. Cada generación presenta mejor desempeño en programabilidad; además, también influye y se incorpora la funcionalidad de las dos grandes interfaces de programación en 3D, OpenGL y DirectX. OpenGL es un estándar de programación 3D para computadoras que corren bajo Windows, Linux, UNIX y Macintosh. DirectX es un conjunto de interfaces de programación multimedia de Microsoft, incluyendo

[2] El término "renderizar" se aplica al proceso en el cual se genera una imagen en dos dimensiones a partir de un modelo en tres dimensiones.

Direct3D para programación en 3D. En la Tabla 2.1 se lista una selección de tarjetas gráficas de NVIDIA representantes de las cuatro generaciones de GPUs.

En la primera generación, las GPUs eran capaces de *rasterizar*[3] triángulos pre-transformados y aplicar una o dos texturas. Estas GPUs liberaban al CPU de actualizar *pixeles*[4] individuales. Sin embargo, no podían transformar vértices de objetos en 3D y tenían un limitado conjunto de operaciones matemáticas. La segunda generación podía aplicar iluminación y realizar transformaciones de vértices en 3D de una manera ágil; esta generación era más configurable, pero aún no verdaderamente programable. La tercera generación fue transicional, pues ya se contaba con programabilidad para vértices, pero aún no para *pixeles*. La cuarta y actual generación de tarjetas gráficas provee programación a nivel de vértices y *pixeles*. Este nivel de programabilidad abrió la posibilidad de trasladar complicadas transformaciones de vértices y *pixeles* del CPU a la GPU.

2.2.3. Cómputo en paralelo

Ya hemos visto que desde hace pocos años, la unidad programable de procesamiento de gráficos ha incrementado de forma considerable el poder de cálculo. Con múltiples núcleos provistos de una memoria con alto ancho de banda, hoy las GPUs ofrecen muchos recursos para el procesamiento de gráficos. Las GPUs están especializadas en cómputo intensivo, cálculo altamente paralelo, y por lo tanto se diseñan de tal forma que se destinan más transistores al procesamiento de datos, como se muestra en la Figura 2.2.

Las tarjetas gráficas están especialmente adecuadas para resolver problemas que pueden ser expresados como cálculo de datos en paralelo, en donde el mismo programa es ejecutado sobre muchos elementos al mismo tiempo y con una alta intensidad aritmética. Ya que el mismo programa se ejecuta por cada elemento, no se requiere de un sofisticado control de flujo, además de que la latencia del acceso a memoria no es un factor que provoque intermitencia, pues hay un contínuo cálculo de datos en las demás unidades y ésto evita el uso de grandes *caches* de datos [21].

[3] La rasterización es el proceso de determinar el conjunto de *pixeles* cubiertos por una primitiva geométrica.
[4] Un *pixel*, cuyo nombre proviene del inglés *picture element*, es la unidad más pequeña de la que están compuestas las imágenes digitales. En el contexto de los procesadores gráficos, es común referirse al término "fragmento", que es el elemento que se obtiene al aplicar un proceso de rasterización y que después corresponderá a un *pixel*. En adelante se utilizarán estos dos términos indistintamente, aunque en una forma estricta los dos conceptos difieran ligeramente en su significado.

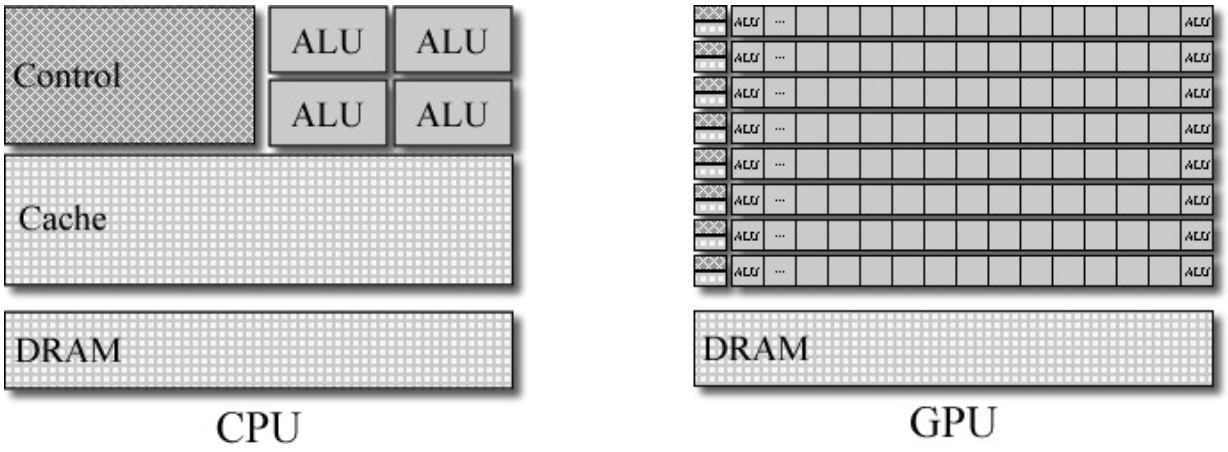

Figura 2.2. *Comparación entre GPU y CPU. La GPU otorga mayor prioridad al procesamiento de datos [21].*

El manejo de información en paralelo distribuye los datos en hilos de procesamiento. Así, muchas aplicaciones que procesan grandes cantidades de datos pueden utilizar un modelo de programación de datos en paralelo para incrementar la velocidad de cálculo. Por ejemplo, en el proceso de *renderizado* en 3D, grandes cantidades de vértices y fragmentos son dirigidas a hilos en paralelo para su procesamiento.

2.2.4. La línea de ensamble gráfica

Una línea de ensamble (o *pipeline*) es una secuencia de etapas operando en paralelo y en un orden fijo. Cada etapa es alimentada por la etapa anterior y a su vez alimenta a la siguiente etapa. Una línea de ensamble gráfica convencional procesa una multitud de vértices, primitivas geométricas y fragmentos de la misma forma.

Los CPUs normalmente cuentan solo con un procesador programable. En contraste, las GPUs tienen al menos dos procesadores programables: el procesador de vértices y el procesador de fragmentos (o procesador de *pixeles*), además de otras unidades no programables. Los procesadores, las partes no programables del hardware gráfico y las aplicaciones están todas vinculadas a través de flujos de datos [33].

En la Figura 2.3 se muestra la organización de la línea de ensamble gráfica utilizada por las GPUs actuales. La aplicación en 3D manda a la tarjeta una secuencia de vértices agrupados dentro de primitivas geométricas, típicamente polígonos, líneas y puntos [27].

Cada vértice tiene una posición, pero también es usual que tenga algunos otros atributos como color, un color secundario (o especular), uno o múltiples conjuntos

de coordenadas de texturas y un vector normal, el cual es usado para calcular la iluminación.

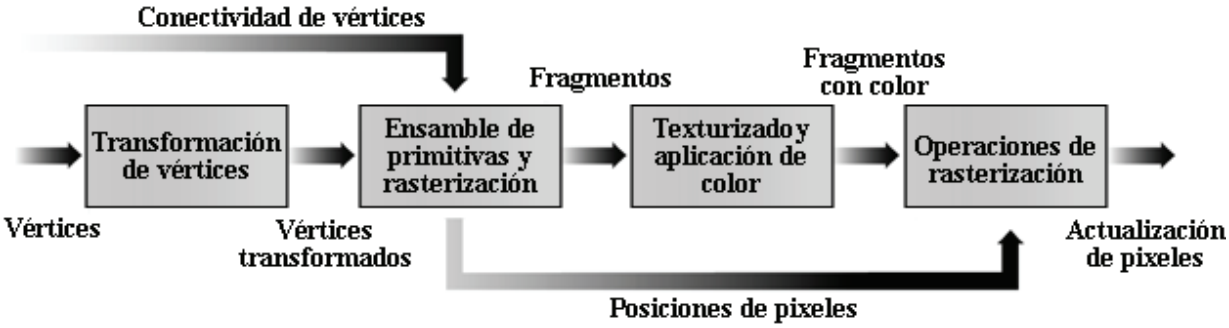

Figura 2.3. *Línea de ensamble gráfica [27].*

En la primera etapa de la línea de ensamble se realiza la transformación de vértices, en donde se ejecuta una serie de operaciones aritméticas sobre cada vértice, entre las cuales se encuentran la transformación de posiciones de vértices a posiciones en pantalla, generación de coordenadas de textura para el texturizado e iluminación del vértice para determinar su color. Estos vértices transformados fluyen hacia la siguiente etapa, en donde se ensamblan dentro de primitivas geométricas, con lo cual se obtiene una secuencia de triángulos, líneas o puntos. En esta etapa también se recortan las primitivas que quedan fuera del área de la región visible del espacio en 3D. Los polígonos que sobreviven a este proceso deben ser *rasterizados*. La *rasterización* es el proceso de determinar el conjunto de *pixeles* cubiertos por una primitiva geométrica. Después de la *rasterización*, viene la etapa de interpolación y aplicación de texturas y color. Finalmente, se realiza una secuencia de operación por fragmentos antes de actualizar el *framebuffer*. En esta etapa se eliminan las superficies ocultas (proceso conocido como *depth testing*), además de aplicar sombreado y operaciones de *rasterización* para determinar el color y posición final del *pixel*, después de ello se actualiza el *framebuffer* con los nuevos *pixeles*. En la Figura 2.4 se observa el proceso descrito aplicado a un conjunto de vértices.

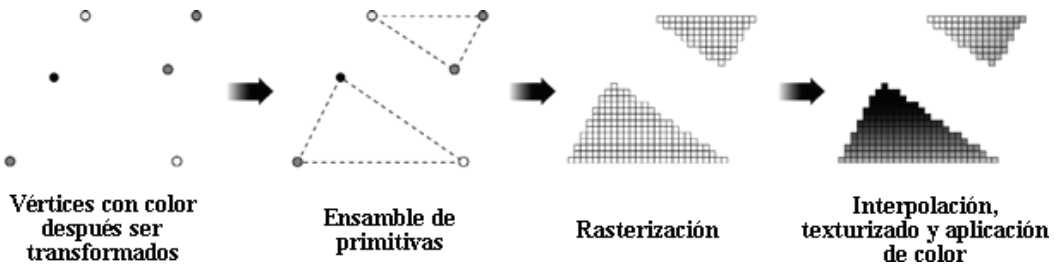

Figura 2.4. *Procesamiento en la línea de ensamble grafica [27].*

2.2.5. Programación en alto nivel

Históricamente, el hardware gráfico se ha programado a un muy bajo nivel. Más recientemente, los programadores configuraban líneas de ensamble usando interfaces programables en lenguaje ensamblador. En teoría, dichas interfaces de programación de bajo nivel proveían gran flexibilidad. En la práctica, eran de uso complicado y presentaban una seria barrera para el efectivo uso del hardware gráfico [17]. Por otra parte, dado que el desarrollo de las GPUs crecía constantemente, era más frecuente la aparición de nuevas características. Con todo ésto, resultaba difícil una adaptación de los programadores a tales cambios, además de que el manejo de mayor cantidad de características con un nivel de programación bajo se volvía cada vez más tedioso. Aunado a lo anterior, la reutilización de código y portabilidad resultaron una tarea con muy pocas posibilidades. Para la solución de todos esos problemas surgieron nuevos lenguajes de programación. El uso de un lenguaje de programación de alto nivel conlleva varias ventajas:

- El tiempo de desarrollo de *shaders*[5] disminuye considerablemente, pues un alto nivel de programación permite la rápida modificación del prototipo y así obtener efectos con alta calidad.
- El compilador optimiza código automáticamente, y realiza tareas a bajo nivel como almacenamiento en registros, que son tediosos y propensos a errores.
- El código escrito en lenguaje de alto nivel es mucho más fácil de leer y de entender. Ésto también permite la creación de nuevos *shaders* a partir de la modificación de anteriores.
- *Shaders* escritos con un lenguaje de alto nivel son más portables a un amplio rango de plataformas que los escritos en un lenguaje de bajo nivel.

El uso de un lenguaje de alto nivel, en conjunto con el inherente poder de las GPUs son las herramientas con las que se cuenta actualmente para el desarrollo de *shaders* y procesamiento de gráficos.

Existen varios lenguajes para la programación de hardware gráfico. Entre los más populares se encuentran: *High Level Shading Language* (HLSL), OpenGL Shading Language (GLSL) y Cg (C para gráficos) [4].

[5] Un *shader* es un programa que se ejecuta dentro de una GPU, usualmente para aplicar algún efecto visual.

2.3. Antecedentes del uso de GPU para propósito general

Una de las tendencias que se han venido desarrollando en años recientes es el uso de tarjetas gráficas para aplicaciones de propósito general. Esta relativamente nueva perspectiva se conoce como Cómputo de Propósito General en GPUs (GPGPU, por sus siglas en inglés) [8]. Debido al creciente nivel de programabilidad presente en las recientes generaciones de GPUs, se ha alcanzado un punto en el cual es posible utilizar dispositivos gráficos para aplicaciones de propósito general que no precisamente se relacionan con el ambiente de los gráficos. Existen numerosos trabajos que han tenido resultados satisfactorios en el ambiente de cómputo genérico con GPUs. Este apartado presenta el desarrollo de esta tendencia. Además, se exhiben algunos de los trabajos de GPGPU, sus alcances y sus perspectivas a futuro.

2.3.1. Aplicaciones del cómputo genérico con GPU

Ya sea que una computadora personal o una consola de videojuegos tenga o no una unidad de procesamiento de gráficos, debe contar con un procesador central sobre el cual corra el sistema operativo y los programas de aplicación. Los CPUs son, por diseño, de propósito general. En ellos se ejecutan aplicaciones escritas en lenguajes de propósito general, tales como C++ o Java. Las recientes GPUs procesan decenas de millones de vértices por segundo y *rasterizan* miles de millones o hasta billones de fragmentos por segundo. Las futuras GPUs serán mucho más rápidas. Estas cantidades son enormemente mayores que la tasa a la cual un CPU puede procesar un número similar de vértices y fragmentos. Sin embargo, una GPU no puede ejecutar los mismos programas de propósito general que sí realiza el CPU [27].

Debido a las diferencias entre las arquitecturas GPU y CPU, no cualquier problema se puede beneficiar de una implementación en GPU. Una de las principales razones es que el acceso a memoria es más restringido en una GPU, por lo que al momento de diseñar algoritmos se deben adaptar el acceso a memoria y las estructuras de datos para que puedan ser procesados correcta y eficientemente en una tarjeta de gráficos. Es claro que sólo ciertas aplicaciones son aptas para ejecutarse en una GPU, pues el hablar de la implementación de una hoja de cálculo o un procesador de palabras no tendría sentido. Sólo aquellas aplicaciones que procesan grandes cantidades de datos pueden usar el modelo de programación de datos en paralelo utilizado en las GPUs para agilizar los cálculos [21]. Por ejemplo, las aplicaciones de procesamiento de imágenes y multimedia tales como el post-procesamiento de generación de imágenes, codificación y decodificación de video, escalamiento de imágenes, estereovisión y reconocimiento de patrones son bastante adecuadas para el cómputo con GPUs. Además, muchos otros algoritmos fuera del ambiente del procesamiento de imágenes pueden ser acelerados por un

procesamiento de datos en paralelo, como lo son el procesamiento de señales en general, simulaciones físicas, cálculo financiero o aplicaciones en biología, entre otras.

2.3.2. Evolución de GPGPU

El uso de dispositivos gráficos para cálculo de propósito general ha sido un área de activa investigación desde hace muchos años. Los comienzos de tales investigaciones se remontan a finales de los años 70s, con la máquina Ikonas, la cual era un sistema programable de *rasterización* para la visualización de instrumentación en la cabina de aeronaves [5]. En 1989, en la Universidad de Carolina del Norte, EUA, apareció el Proyecto *Pixel-Planes*, dedicado a construir motores gráficos con énfasis en renderizado en tiempo real [28].

Otras investigaciones encontraron la forma de utilizar hardware gráfico para más diversas aplicaciones. El uso de dispositivos de *rasterización* para la planeación de movimiento de un robot se describe en [19]. Los dispositivos gráficos también se han utilizado para implementar el método de interpolación basado en distancia euclidiana, conocido como Diagramas de Voronoi, el cual es descrito en [14]. En 1999, Ishihara y Kedem [15], dentro del proyecto llamado *CipherFlow*, utilizaron una arquitectura de alta velocidad y gran realismo en la generación de imágenes, conocida como *PixelFlow* [6], para implementar técnicas de criptoanálisis que rompían contraseñas en sistemas UNIX a base de fuerza bruta.

Ya para el año 2000 se incrementó notablemente la investigación sobre hardware gráfico. En la Universidad de Toronto, Canadá, se hacía investigación para demostrar la capacidad de los dispositivos gráficos para realizar cómputo general, tomando como ejemplo el cálculo en tiempo real del efecto de refracción de la luz [32]. En 2002 ya eran frecuentes los trabajos que se enfocaban a la visualización de sistemas físicos, como el descrito por Harris [11], en donde se presentaba un método para la simulación en tiempo real de fenómenos dinámicos como calentamiento, convección y reacción-difusión. Harris también realizó investigación para la simulación del comportamiento de nubes utilizando GPUs [12].

En años recientes resultan vastos los trabajos realizados en el área de GPGPU con diversas aplicaciones, entre las que se encuentran la simulación de crecimiento de cristales de hielo [16], algoritmos para multiplicaciones matriciales [7], simulación de fluídos [4], implementación de algoritmos genéticos [26], algoritmos numéricos para simulación de superficies de agua [35], entre otros.

Los resultados obtenidos en gran parte de las investigaciones en cálculo genérico con GPUs muestran, en primer lugar, que las tarjetas gráficas han evolucionado para convertirse en procesadores con un gran poder de cálculo que puede ser aprovechado para diversidad de aplicaciones. En segundo lugar, si bien es

posible controlar las etapas de procesamiento, aún en las recientes versiones de GPUs existen características que son propensas a mejoras para tener una mayor flexibilidad y menores restricciones para la programación genérica. Ésto es punto de opiniones encontradas, pues hay quienes argumentan que el diseño de las GPUs se enfoca al procesamiento de gráficos, y que su aplicación para otros fines resulta en una distorsión de su función original.

En el desarrollo de cálculo genérico con GPU se encuentra implícito el tener que trabajar en un ambiente envuelto en el mundo de los gráficos, debido precisamente al propósito original de las GPUs; ésto implica que debe existir un esfuerzo no trivial para adaptarse al medio cuando se programan aplicaciones de GPGPU. Una de las metas que se están llevando a cabo es construir ambientes y herramientas que hagan que la programación con GPUs sea más sencilla para aplicaciones de propósito general. Existen varios trabajos al respecto, entre los que se encuentran el proyecto BrookGPU desarrollado desde 2003 en la Universidad de Stanford, y más recientemente, la tecnología CUDA, desarrollada por Nvidia en 2007.

2.3.2.1. BrookGPU

El proyecto BrookGPU desarrollado en la Universidad de Stanford [30], es una extensión del estándar ANSI C y está diseñado para incorporar las ideas del cómputo de datos en paralelo e intensidad aritmética dentro de un lenguaje familiar y eficiente. El modelo computacional general provee dos beneficios principales sobre los lenguajes convencionales:

- Paralelismo de datos, que permite al programador especificar la forma de realizar las mismas operaciones en paralelo sobre diferentes datos.
- Intensidad aritmética, que incita a los programadores a especificar operaciones sobre datos que minimicen la comunicación global y maximicen el cómputo local.

La arquitectura de BrookGPU se basa en dos componentes: BRCC (*Brook C Compiler*), un compilador que trabaja sobre archivos fuente de Brook (.br) y genera archivos en lenguaje C++; la librería de tiempo de ejecución *Brook Runtime* (BRT), que implementa el soporte de las instrucciones de Brook para un hardware en particular.

2.3.2.2. CUDA

La tecnología *Compute Unified Device Architecture* (CUDA) representa una nueva arquitectura de hardware y software para la generación y manejo de cómputo en una GPU, utilizándola como un dispositivo de cálculo en paralelo sin la necesidad de adaptación a una interfaz de programación de aplicaciones (API) gráfica. CUDA está disponible en la serie de tarjetas gráficas GEForce 8, en el procesador Tesla y algunos procesadores Quadro [21].

Esta tecnología permite utilizar un entorno de desarrollo en lenguaje de programación C con algunas extensiones para codificar algoritmos a ejecutarse en una GPU. Por otra parte, la arquitectura CUDA presenta mejoras en los accesos a memoria, permitiendo leer y escribir datos en cualquier localidad, así como se realiza en un CPU; ésto conlleva una mayor flexibilidad de programación. Por otra parte, la arquitectura también ofrece mayor velocidad en la transmisión de datos del CPU a la GPU y viceversa. Las herramientas de desarrollo de CUDA son proporcionadas de forma libre por Nvidia y son compatibles con sistemas operativos Linux y Windows [20].

Debido a que es una tecnología nueva, CUDA no se puede utilizar en la gran mayoría de las tarjetas gráficas existentes en el mercado, y una de sus desventajas es que únicamente las GPUs más recientes desarrolladas por Nvidia son aptas para trabajar con dicha tecnología.

2.4. Programación multi-hilos

Tradicionalmente, el desarrollo de programas se ha realizado en base a cómputo en serie, en donde se cuenta con una máquina con un solo procesador central; el problema que se pretende resolver es seccionado en una serie de instrucciones, las cuales son ejecutadas una después de la otra; de esta forma, sólo una instrucción es ejecutada a la vez. Desde el punto de vista de la programación en paralelo, un problema se puede resolver con el uso simultáneo de múltiples recursos de cómputo. Trabajando con múltiples CPUs, un problema es seccionado en partes que pueden ser resueltas concurrentemente; cada parte es separada en una serie de instrucciones y las instrucciones de cada parte son ejecutadas simultáneamente en diferentes CPUs. Ésto puede involucrar varios esquemas, ya sea el uso de una única computadora con múltiples procesadores, un conjunto de computadoras conectadas por una red, o una combinación de ambos [36].

El cómputo en paralelo provee concurrencia y por lo tanto reduce el tiempo de resolución de un problema, ya sea que éste sea pequeño o de mucha mayor escala. Uno de los modelos de programación en paralelo es el modelo en hilos, en donde un único proceso puede tener rutas de ejecución múltiples y concurrentes. En esta sección se explican algunas generalidades de dicho modelo de programación.

2.4.1. Diseño de programas en paralelo

Es claro que en el desarrollo de todo programa un primer paso es comprender el problema que se quiere resolver. Una vez hecho ésto, se debe determinar si el

problema es o no apto para implementarse en paralelo, lo cual implica identificar si es posible separar el problema general en un conjunto de subproblemas, los cuales deben ser independientes entre sí (ver Figura 2.5). La independencia radica principalmente en la nula necesidad de esperar la finalización de un subproblema para la iniciación de otro. Por otro lado, además de identificar la independencia entre subproblemas, también se deben tomar en cuenta algunos otros aspectos, entre los cuales se encuentran:

- Comunicación;
- Dependencia de datos;
- Sincronización;
- Complejidad del programa;
- Esfuerzo, costo y tiempo de programación.

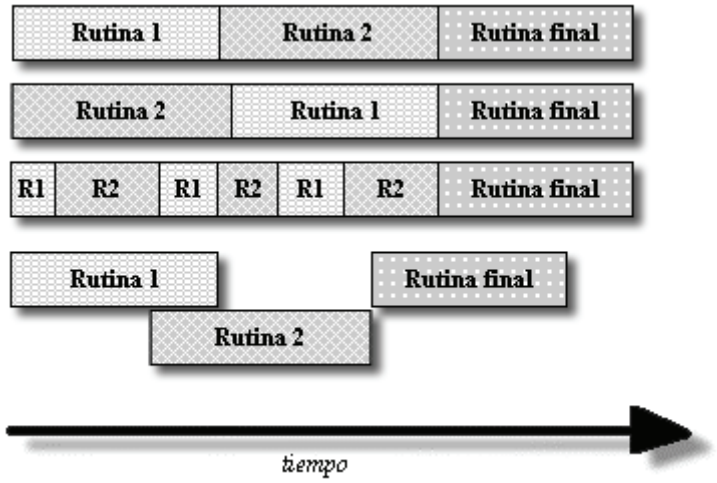

Figura 2.5. *Paralelismo en un programa secuencial.*

La implementación de paralelismo con multihilos comprende todos estos aspectos, con lo que para tomar completa ventaja del cómputo en paralelo se ha estandarizado una interfaz de programación. Esta interfaz ha sido especificada por el estándar IEEE POSIX 1003.1c (1995). Las implementaciones adheridas a este estándar son referidas como hilos POSIX, o Pthreads. Los hilos POSIX están definidos como un conjunto de tipos y procedimientos en lenguaje C, implementados en un archivo de cabecera y una librería de hilos [13].

2.4.2. Definición de hilo

Un método para lograr paralelismo consiste en hacer que varios procesos cooperen y se sincronicen mediante memoria compartida. Una alternativa es el empleo de múltiples hilos de ejecución en un solo espacio de direcciones.

Cuando se ejecuta un programa, el CPU utiliza el valor del contador del programa para determinar cuál instrucción debe ejecutar a continuación. El flujo de instrucciones resultante se denomina *hilo de ejecución* del programa. Desde el punto de vista del programa, la secuencia de instrucciones de un *hilo de ejecución* es un flujo ininterrumpido de direcciones. En cambio, desde el punto de vista del procesador, los *hilos de ejecución* de diferentes procesos están entremezclados, y el punto en que la ejecución cambia de un proceso a otro se denomina conmutación de contexto [29].

Una extensión natural del modelo de proceso es permitir la ejecución de varios hilos dentro del mismo proceso. Esta extensión provee un mecanismo eficiente para controlar hilos de ejecución que comparten tanto código como datos, con lo que se evitan comunicaciones de contexto.

Cada *hilo de ejecución* se asocia con un *hilo*, un tipo de datos abstracto que representa el flujo de control dentro de un proceso. Cada hilo tiene su propia pila de ejecución, valor de contador del programa, conjunto de registros y estado [29].

2.4.3. API de hilos POSIX

Existe una API que está definida en el estándar ANSI/IEEE POSIX 1003.1. Las subrutinas que comprende la API de hilos POSIX se pueden agrupar en tres clases: manejo de hilos, *mutexes* y variables de condición.

El manejo de hilos comprende diversas actividades, entre las que se encuentran la creación, terminación, inicialización de atributos y destrucción de hilos. Además, también es posible la unión de hilos y el manejo de sus pilas. POSIX maneja un mecanismo de sincronización de hilos llamado *mutex* para el uso de candados a corto plazo y maneja variables de condición para esperar sucesos de duración ilimitada.

2.4.4. Sincronización de hilos

Los hilos se crean dentro del espacio de direcciones de un proceso y comparten recursos con las variables estáticas y descriptores de archivos abiertos, por lo cual deben sincronizar su interacción a fin de obtener resultados consistentes. Existen dos tipos de sincronización: candados y espera. En el uso de candados, un hilo tiene acceso exclusivo a los recursos por un corto tiempo. La espera, que puede tener duración ilimitada, se refiere al bloque del hilo hasta que ocurra cierto suceso [29].

2.4.4.1. Mutexes

El mutex, abreviación de exclusión mútua, es el mecanismo de sincronización de hilos más sencillo y eficiente. Los programas emplean mutex para preservar secciones críticas[6] y obtener acceso a los recursos. Un mutex se maneja a partir de una variable que se declara e inicializa antes de utilizarse para sincronización. Por lo regular, las variables de mutex son accesibles para todos los hilos del proceso.

Un hilo puede bloquear un mutex para proteger sus secciones críticas, como el acceso a memoria, o la lectura de un puerto. De esta forma, los demás hilos que quieran hacer uso de los recursos tendrán que esperar a que el mutex bloqueado sea liberado por el hilo bloqueante y así poder competir por el acceso a los recursos [9].

Sólo se logra la exclusión mútua cuando todos los hilos manejan el bloqueo de candado antes de entrar en sus secciones críticas y el desbloqueo al salir de ellas. No hay nada que impida a un hilo ingresar en una sección crítica sin haber bloqueado un mutex [29].

2.4.4.2. Variables de condición

Las variables de condición proveen otra forma de sincronización. Mientras los mutex implementan la sincronización de hilos controlando el acceso a datos, las variables de condición permiten la sincronización de hilos basándose en el actual valor de los datos. Sin las variables de condición se requeriría que un hilo estuviera continuamente bloqueando y desbloqueando un mutex hasta que se alcance una condición específica. Ésto puede llegar a consumir muchos recursos ya que el hilo estaría contínuamente ocupado en esa actividad. Una variable de condición es un mecanismo con el cual se puede lograr el mismo propósito sin bloquear y desbloquear el mutex contínuamente [13].

De igual forma que los mutex, las variables de condición deben ser declaradas e inicializadas antes de poder ser utilizadas. Una variable de condición es siempre utilizada con un mutex. Suponiendo que un hilo necesita esperar hasta que se cumpla una condición en la que interviene un conjunto de variables compartidas, éstas se protegerán con un mutex. Una variable condicional ofrece un mecanismo

[6] Una sección crítica es un fragmento de código en el cual se accede a los recursos del sistema.

para que los hilos esperen el cumplimiento de predicados en los que intervienen las variables compartidas. Cada vez que un hilo modifica dichas variables compartidas, señaliza mediante la variable de condición que se ha realizado un cambio. Esta señal activa un hilo en espera, que determinará si su predicado se cumple. Cuando se envía una señal a un hilo en espera, éste debe cerrar el candado antes de probar su predicado. Si el predicado es falso, el hilo deberá desbloquear el candado y continuar en espera. La operación de espera bloquea el hilo y libera el mutex. Así, el mutex se libera explícitamente si el predicado se cumple e implícitamente si no se cumple.

2.4.5. Ventajas de la programación con hilos

La programación con multihilos involucra una serie de ventajas respecto a la utilización del modelo tradicional de programación en serie. Una de ellas se basa en el hecho de que un hilo puede ser ejecutado en diferentes procesadores; de esta forma, cuando se cuenta con una computadora con múltiples procesadores, se pueden tener múltiples hilos ejecutándose simultáneamente. Además, un hilo puede realizar una petición al sistema esperando hasta que el servicio se complete; mientras tanto, otros hilos pueden continuar trabajando y realizando varias peticiones en un proceso sin que éste se bloquee. Así, una aplicación que requiera de operaciones extensas puede separarse en la ejecución de múltiples hilos independientes, lo cual permite que la aplicación esté siempre en actividad, disminuyendo los tiempos de respuesta. Por otra parte, las aplicaciones que mantienen varios procesos con acceso a memoria compartida requieren de un control y sincronización mucho más costosos en tiempo, ya que requiere de más recursos del sistema; en un sistema multihilos la sincronización requiere menor uso de recursos, pues los hilos comparten datos en un mismo espacio de direcciones [25].

Capítulo 3

Método de masas y resortes en una GPU

Este capítulo muestra la implementación del método de masas y resortes en la GPU. En primera instancia se abordan las pruebas preeliminares realizadas con la tarjeta para determinar su eficiencia respecto a la utilización del CPU. Después se presenta el desarrollo de la programación del algoritmo de masas y resortes en la tarjeta gráfica. Por último se aborda la implementación de multihilos como medio para lograr paralelismo entre las diferentes etapas del proceso de simulación.

3.1. Pruebas preeliminares

A continuación se presentan una serie de pruebas que están dirigidas a evaluar el eventual uso de la GPU para aplicaciones de cálculo genérico.

Las pruebas se realizaron con base en la ejecución de operaciones aritméticas utilizando el procesador de fragmentos[7] de la GPU, pues es más adecuado para el cómputo exhaustivo. Se utilizó la tarjeta de gráficos NVidia GE Force 7900GS y un procesador central Intel Core 2 DUO 6400 @ 2.13GHz con 1 GB de RAM. Se hace uso del lenguaje Cg para la programación del procesador de fragmentos dentro de la GPU; además, se utiliza el lenguaje C y la librería OpenGL en el ambiente de desarrollo DEV C++ [1] bajo el sistema operativo Windows XP Profesional.

Como primer paso para poder trabajar con la tarjeta gráfica, es necesario establecer la forma en que se efectuará la comunicación con ella. El lenguaje utilizado para la programación es Cg [27], el cual está provisto de un paquete de herramientas en versión para DirectX y OpenGL. En dicho paquete se proporcionan librerías que especifican las rutinas necesarias tanto para el manejo de archivos fuente de Cg como para la interacción y manejo de los *shaders* en la aplicación principal.

[7] Las tarjetas gráficas cuentan con dos unidades programables: el procesador de vértices y el procesador de fragmentos (o pixeles).

3.1.1. *Shaders* con el lenguaje Cg

El manejo de *shaders* dentro del entorno de la aplicación principal implica la compilación y administración de los programas en Cg, así como el suministro de información hacia los *shaders*. A continuación se presenta cada una de estas etapas.

3.1.1.1. Compilación

Los *shaders* de Cg deben ser compilados y cargados en la tarjeta gráfica. Para lograr tal fin es necesario efectuar una secuencia de pasos de inicialización del ambiente de Cg dentro de la aplicación principal.

El primer paso consiste en generar un contexto, que es un contenedor que almacena múltiples programas de Cg, así como sus datos compartidos. El contexto se crea de la siguiente forma:

```
CGcontext contexto = cgCreateContext();
```

Ya que no todas las tarjetas gráficas proveen las mismas características, es necesario especificar un perfil, el cual indica el subconjunto del lenguaje Cg que es soportado por el hardware que se utiliza. Para obtener el mejor perfil disponible se recurre a la rutina:

```
CGprofile perfil = cgGLGetLatestProfile(CG_GL_FRAGMENT);
```

Esta función determina el perfil más adecuado a usarse para el procesador de fragmentos. Para conocer las capacidades de cada uno de los diferentes perfiles existentes se puede consultar [17].

Con el contexto y el perfil ya especificados, se procede a crear el programa. El código puede alojarse en la memoria de la aplicación principal o fuera de ella, almacenado en un archivo. Para este caso, el código se encuentra en un archivo de Cg (shader.cg), por lo que se utiliza la siguiente función:

```
CGprogram programa = cgCreateProgramFromFile(contexto,
                                             CG_SOURCE,
                                             "shader.cg",
                                             perfil,
                                             "main",
                                             NULL);
```

Dicha función crea un programa objeto, lo agrega al contexto especificado y compila el código fuente asociado de acuerdo al perfil indicado. Además, se especifica el tipo de programa, el nombre del archivo fuente y la función de entrada

del programa, que en este caso es “main”. También es posible pasar argumentos al compilador, aunque aquí se omiten con NULL.

Después de compilar un programa, es necesario cargar el código objeto a OpenGL. Ésto se hace con:

```
cgGLLoadProgram(programa);
```

Antes de poder ejecutar un programa con OpenGL, se debe habilitar su perfil correspondiente:

```
cgGLEnableProfile(perfil);
```

Por último, el programa creado con Cg debe acoplarse al estado actual de la aplicación principal, lo cual indica que se ejecutará cada que se realicen llamadas a funciones de OpenGL que dibujen en la pantalla.

```
cgGLBindProgram(programa);
```

3.1.1.2. Manejo de variables

Las variables declaradas en un programa de Cg pueden ser manipuladas desde la aplicación principal. Para ello se utiliza el concepto de *parámetro*. Un *parámetro* proporciona un vínculo entre variables del *shader* y del programa principal. Para crear un parámetro se ejecuta la siguiente función:

```
CGparameter parametro = cgGetNamedParameter( programa, "var1");
```

Esta función crea el parámetro para manejar la variable `var1` declarada dentro del programa de Cg especificado. Para cargar un valor a dicha variable se utiliza:

```
cgGLSetParameter1f(parametro, val);
```

Aquí, `val` es una variable declarada en la aplicación principal que almacena el valor a ser cargado a la variable del *shader*. Los tipos de datos deben corresponder tanto en la declaración del *shader* como el valor cargado desde la aplicación principal, ya sea que los dos sean enteros o de punto flotante, principalmente.

El manejo de parámetros proporciona una manera de inicializar las variables del *shader* desde la aplicación principal. Lo mismo es posible en el caso de texturas, creando el parámetro:

```
CGparameter texparametro = cgGetNamedParameter( programa, "texvar1");
```

En este caso, `texvar1` es una variable de textura de tipo `sampler2D`[8], para el caso de texturas de dos dimensiones, que es declarada en el programa de Cg. Por otro lado, en la aplicación principal se especifica la textura:

```
cgGLSetTextureParameter(texparametro, textura);
```

Aquí, en vez de un valor, se especifica un manejador de textura, que es un número entero generado con:

```
glGenTextures(1,&textura);
```

[8] Los tipos de datos soportados por Cg pueden consultarse en [17].

Además, en el caso de texturas, es necesario activar el parámetro:

```
cgGLEnableTextureParameter(texparametro);
```

Así, OpenGL trasladará la textura a la tarjeta gráfica y se accederá a ella a partir de la variable de textura indicada en el *shader*.

3.1.2. GPU como procesador de propósito general

La primera prueba se enfoca en la utilización de la tarjeta gráfica como procesador de propósito general. Para ello, se pretende que el procesador de fragmentos realice una operación aritmética sencilla y proporcione un resultado que se pueda utilizar para cálculos posteriores.

Como punto inicial, se debe especificar la forma en que se realiza el envío de datos a la tarjeta y la lectura de los resultados provenientes de ella. El envío de datos se efectúa con texturas que almacenan la información que será leída por el *shader*. Los *pixeles* finales calculados por la tarjeta usualmente son almacenados en un *framebuffer*[9]. Para poder leer dichos *pixeles*, se aplica la técnica denominada *render to texture*, en la cual los datos de retorno de la tarjeta se almacenan en una textura, la cual puede ser leída y reutilizada tanto por el *shader* como por la aplicación principal para futuras operaciones. Para poder realizar la técnica de *render to texture* es necesario utilizar la extensión de OpenGL denominada *EXT_framebuffer_object* (FBO). Esta extensión permite escribir sobre un *framebuffer* diferente al utilizado para actualizar la pantalla, además de que provee precisión de punto flotante de 32 bits, necesaria en cómputo genérico. Para el manejo de extensiones se hace uso de la librería GLEW (OpenGL Extension Wrangler Library) [31].

Los datos recibidos de la tarjeta se almacenarán en un *framebuffer*, el cual se crea con:

```
GLuint fb;
glGenFramebuffersEXT(1,&fb);
```

También debe activarse la escritura sobre el *framebuffer* creado:

```
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT,fb);
```

La operación aritmética a calcularse en el procesador de fragmentos es:

```
y = alpha*x
```

[9] El *framebuffer* es un espacio en la memoria gráfica desde donde se lee la imagen que aparecerá en pantalla. Puede llegar a tener 8 bits de profundidad por cada uno de los canales RGBA, por lo que cada canal trabaja en el rango de 0 a 255.

donde `x` es un vector que almacena los datos de entrada[10], cuya dimensión se expresa en función del tamaño de textura `texSize*texSize*4`, `alpha` es un escalar, y `y` es el vector de resultado.

Para hacer llegar los datos a la tarjeta gráfica se requiere almacenarlos dentro de una textura. En primer lugar, los datos se almacenan en un arreglo y son cargados a la textura por medio de la función `glTexSubImage2D` de OpenGL. Para ello, se crea la textura:

```
GLuint textura;
glGenTextures(1, &textura);
```

Se especifica el tipo de textura[11] y sus propiedades de manera que no se aplique ningún tipo de filtro que pueda alterar la información contenida en ella:

```
    glBindTexture(GL_TEXTURE_RECTANGLE_ARB,textura);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_MIN_FILTER,
                                          GL_NEAREST);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_MIN_FILTER,
                                          GL_NEAREST);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_MAG_FILTER,
                                          GL_NEAREST);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_WRAP_T, GL_CLAMP);
```

La textura aún no contiene la información necesaria. Para cargar datos se utiliza la siguiente función, donde `datatex` es un arreglo unidimensional que almacena la información:

```
glTexSubImage2D(GL_TEXTURE_RECTANGLE_ARB,0,0,0,texSize,texSize,GL_RGB,
                                          GL_FLOAT,datatex);
```

Una vez que la textura contiene los datos, se dirige la escritura del *FrameBuffer* hacia la textura de salida `ytex`; ésto se realiza con la función:

```
   glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,GL_COLOR_ATTACHMENT0_EXT,
                                    GL_TEXTURE_RECTANGLE_ARB,ytex,0);
```

Donde `ytex` es una textura en la cual se almacenaran los resultados calculados con la GPU.

Para que se ejecute el cálculo se requiere que la tarjeta procese los datos cargados en la textura, para ello basta con dibujar un cuadro con las mismas dimensiones de la textura, de tal forma que el procesador de fragmentos opere exactamente sobre todos los *texeles*[12].

---

[10] Los datos son unidimensionales y tendrán una correspondencia con una textura bidimensional de tamaño `texSize*texSize`; en este caso se manejan texturas con formato de cuatro componentes: RGBA.

[11] Las extensiones de OpenGL permiten generar texturas sin restricción de tamaño a potencias de 2, con lo cual se pueden crear texturas que se acoplen de forma más exacta a los datos que se almacenan en ella.

[12] Un *texel* es la unidad elemental de la textura, como lo es un *pixel* en una imagen.

```
glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0);
    glVertex2f(0.0, 0.0);
    glTexCoord2f(texSize, 0.0);
    glVertex2f(texSize, 0.0);
    glTexCoord2f(texSize, texSize);
    glVertex2f(texSize, texSize);
    glTexCoord2f(0.0, texSize);
    glVertex2f(0.0, texSize);
glEnd();
```

Cabe mencionar que se debe mantener una relación de 1:1 entre la ventana de OpenGL y la textura, ya que de otra forma los datos se verían alterados; para ésto se requiere una proyección ortogonal y un puerto de vista del mismo tamaño que la textura:

```
gluOrtho2D(0.0,texSize,0.0,texSize);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glViewport(0,0,texSize,texSize);
```

Al momento en que se dibuja el cuadro, se ejecuta en la GPU el *shader*; el procesador de fragmentos recorre toda la textura y la aplica al cuadro dibujado, operando de acuerdo a lo establecido en el programa cargado a la GPU. El código fuente del *shader* de esta primera prueba se puede observar en el Apéndice A.1.

Para recuperar los datos calculados se lee sobre la textura de salida. Ésto se realiza con la función *glReadPixels* de OpenGL, la cual lee los pixeles de la textura y los almacena en memoria de CPU por medio de un arreglo. Antes, se selecciona la sección del *buffer* desde donde se va a leer.

```
glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
glReadPixels(0,0,texSize,texSize,GL_RGB,GL_FLOAT,result0tex);
```

Aquí, `result0tex` es un arreglo unidimensional que almacena los datos leídos.

El resultado de esta prueba, realizando la operación `y=alpha*x`, con `texSize=4`, un vector de tamaño `texSize*texSize*4=64` y `alpha=0.5` fue el siguiente:

```
Datos enviados:
1.00; 2.00; 3.00; 4.00; 5.00; 6.00; 7.00; 8.00; 9.00; 10.00; 11.00; 12.00; 13.00
; 14.00; 15.00; 16.00; 17.00; 18.00; 19.00; 20.00; 21.00; 22.00; 23.00; 24.00; 2
5.00; 26.00; 27.00; 28.00; 29.00; 30.00; 31.00; 32.00; 33.00; 34.00; 35.00; 36.0
0; 37.00; 38.00; 39.00; 40.00; 41.00; 42.00; 43.00; 44.00; 45.00; 46.00; 47.00;
48.00; 49.00; 50.00; 51.00; 52.00; 53.00; 54.00; 55.00; 56.00; 57.00; 58.00; 59.
00; 60.00; 61.00; 62.00; 63.00; 64.00;

Datos recibidos:
0.50; 1.00; 1.50; 2.00; 2.50; 3.00; 3.50; 4.00; 4.50; 5.00; 5.50; 6.00; 6.50; 7.
00; 7.50; 8.00; 8.50; 9.00; 9.50; 10.00; 10.50; 11.00; 11.50; 12.00; 12.50; 13.0
0; 13.50; 14.00; 14.50; 15.00; 15.50; 16.00; 16.50; 17.00; 17.50; 18.00; 18.50;
19.00; 19.50; 20.00; 20.50; 21.00; 21.50; 22.00; 22.50; 23.00; 23.50; 24.00; 24.
50; 25.00; 25.50; 26.00; 26.50; 27.00; 27.50; 28.00; 28.50; 29.00; 29.50; 30.00;
 30.50; 31.00; 31.50; 32.00;
```

Estos resultados implican la capacidad de utilizar la tarjeta gráfica no necesariamente para aplicaciones gráficas, sino para realizar cálculo numérico de propósito general; además, como resultado de ésta prueba se conoce la configuración y los requerimientos necesarios para operar la tarjeta y hacerla funcionar como un co-procesador.

3.1.3. Medición de desempeño de la GPU

Después de lograr el funcionamiento de la tarjeta gráfica como una unidad de cálculo numérico de propósito general, se pretende entonces tener una idea clara de las ventajas de su uso con respecto al del procesador central.

La segunda prueba se basa en los resultados obtenidos en la primera y tiene como objetivo medir la eficiencia en poder de cálculo de la GPU con respecto al CPU. Para dicho fin, se realiza una serie de operaciones aritméticas (compuestas por sumas y multiplicaciones) utilizando el CPU y se efectúa el mismo procedimiento en la GPU, en concreto, en el procesador de fragmentos.

El cálculo en el CPU se realiza programando las operaciones aritméticas en lenguaje C. Por otra parte, el cálculo en GPU se realiza utilizando Cg. El código fuente para cada una de estas implementaciones se encuentra en los apéndices A.2 y A.3.

Para medir el tiempo de procesamiento se hace uso de la función *clock*, incluída en el archivo de cabecera time.h de C.

Al igual que en la primera prueba, la configuración de las texturas y la aplicación de la técnica *render to texture* también son empleadas, con la diferencia de que los cálculos son ahora más complejos y se aplican a una mayor cantidad de datos.

Para esta segunda prueba se trabaja con `texSize= 1700`, un vector de tamaño

`texSize*texSize*4 = 11 560 000`, y se obtiene:

```
Calculo con CPU:
Tiempo: 0.297000

Calculo con GPU:
Tiempo: 0.062000
```

Se observa que para este caso, el tiempo de cálculo realizado por la GPU es casi cinco veces menor al obtenido en CPU.

De igual forma se realizó la medición de tiempos de ejecución en CPU y GPU, variando la cantidad de los datos. Los resultados se muestran en la Figura 3.1.

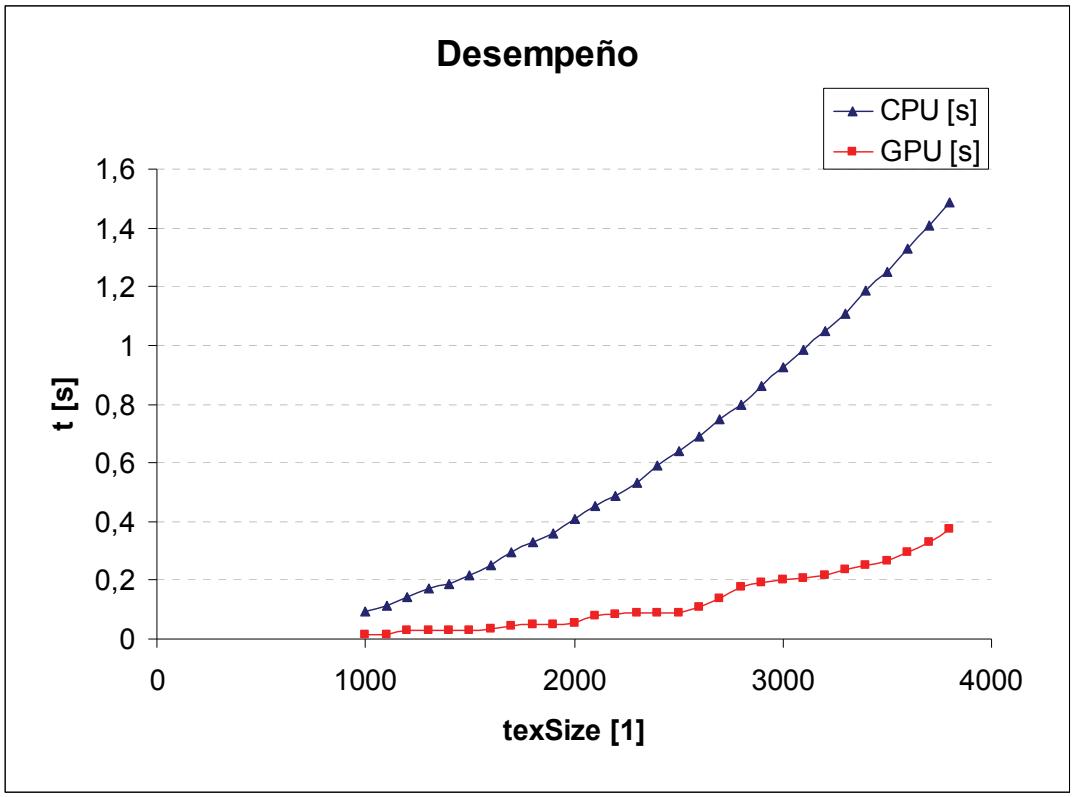

Figura 3.1. *Comparación de desempeño entre CPU y GPU. Se muestra la gráfica de comparación de tiempos de cálculo en función de la cantidad de datos, determinados por* `texSize*texSize*4`*. Se parte de* `1000` *(que representa 1 millón de pixeles en formato RGBA), hasta* `3800` *(14.44 millones de pixeles).*

Se observa que los tiempos obtenidos con el CPU son mayores a los medidos para la GPU. Con ésto se establece una clara ventaja al momento de utilizar la tarjeta gráfica para realizar cálculo que requiere de cómputo exhaustivo, como es el caso del método de masas y resortes.

3.2. Programación de la GPU

En este apartado se presenta el desarrollo de la implementación del método de masas y resortes utilizando la GPU.

En el modelo original del simulador de cirugía de próstata la información de las propiedades de los vértices con la que opera el CPU se almacena en una estructura de datos en donde cada uno de los vértices se representa con un elemento compuesto por propiedades como masa, posición, velocidad, aceleración, entre otras.

Ya se ha establecido que debido a su arquitectura, la GPU tiene un gran poder de cálculo, pero para poder aprovecharlo es necesario que los datos que le sean suministrados se encuentren vectorizados[13], pues la manera de operación de la tarjeta es a través del flujo de datos. Es por ello que se requiere una transformación en la estructura de almacenamiento de la información, de tal forma que todos los datos que intervienen en el cálculo puedan ser trasladados a una o varias texturas, las cuales son fácilmente digeribles por la tarjeta.

La estructura agrupa todas las propiedades de los vértices, y en conjunto se vinculan a través de una lista ligada. Para llevar a cabo la transición en la estructura de almacenamiento, los datos son recolectados recorriendo la lista ligada y almacenando cada una de las propiedades en un arreglo[14], de manera que se tienen tantos arreglos como propiedades de vértice; de esta forma se cuenta con una organización óptima para que la información sea transferida a texturas que posteriormente leerá la tarjeta gráfica (ver Figura 3.2). Además, una de las ventajas de manejar los datos de esta forma radica en que al momento de trasladarse a una textura, implícitamente la información será interpretada como un píxel, y ya que la mayor parte de los datos son de carácter vectorial de tres componentes, resulta muy conveniente, pues los pixeles son el tipo de datos con los que opera la tarjeta de forma nativa[15].

Es necesario tener una visión clara de cómo serán interpretados los datos una vez que se hayan trasladado a texturas y al momento de ser leídos por la GPU. La importancia de almacenar cada propiedad en una textura independiente radica en el hecho de que el mapeo que ejecuta la tarjeta sobre las texturas se realiza en base a un solo par de coordenadas (para el caso de texturas bidimensionales). Entonces, la tarjeta recorrerá cada una de las texturas, pero siempre accediendo a una sola coordenada por vez.

---

[13] Refiriéndose a "vectorizados" como una representación en forma de arreglo unidimensional.

[14] Dado que todos los datos son reagrupados, es importante respetar el orden en que cada propiedad toma su lugar, pues será su posición la que determine el pertenecer a un vértice o a otro.

[15] La vectorización de los tipos de datos permite realizar operaciones con una sola instrucción aplicada al vector, en vez de operar individualmente sobre cada uno de los componentes, como se hace tradicionalmente en lenguajes como C.

De aquí la importancia de que los datos de cada vértice estén correctamente ordenados en la misma posición dentro de cada textura, pues de otra forma serían interpretados como pertenecientes a otro vértice. Este esquema es la base del manejo de lectura de información utilizado en la implementación del método de masas y resortes.

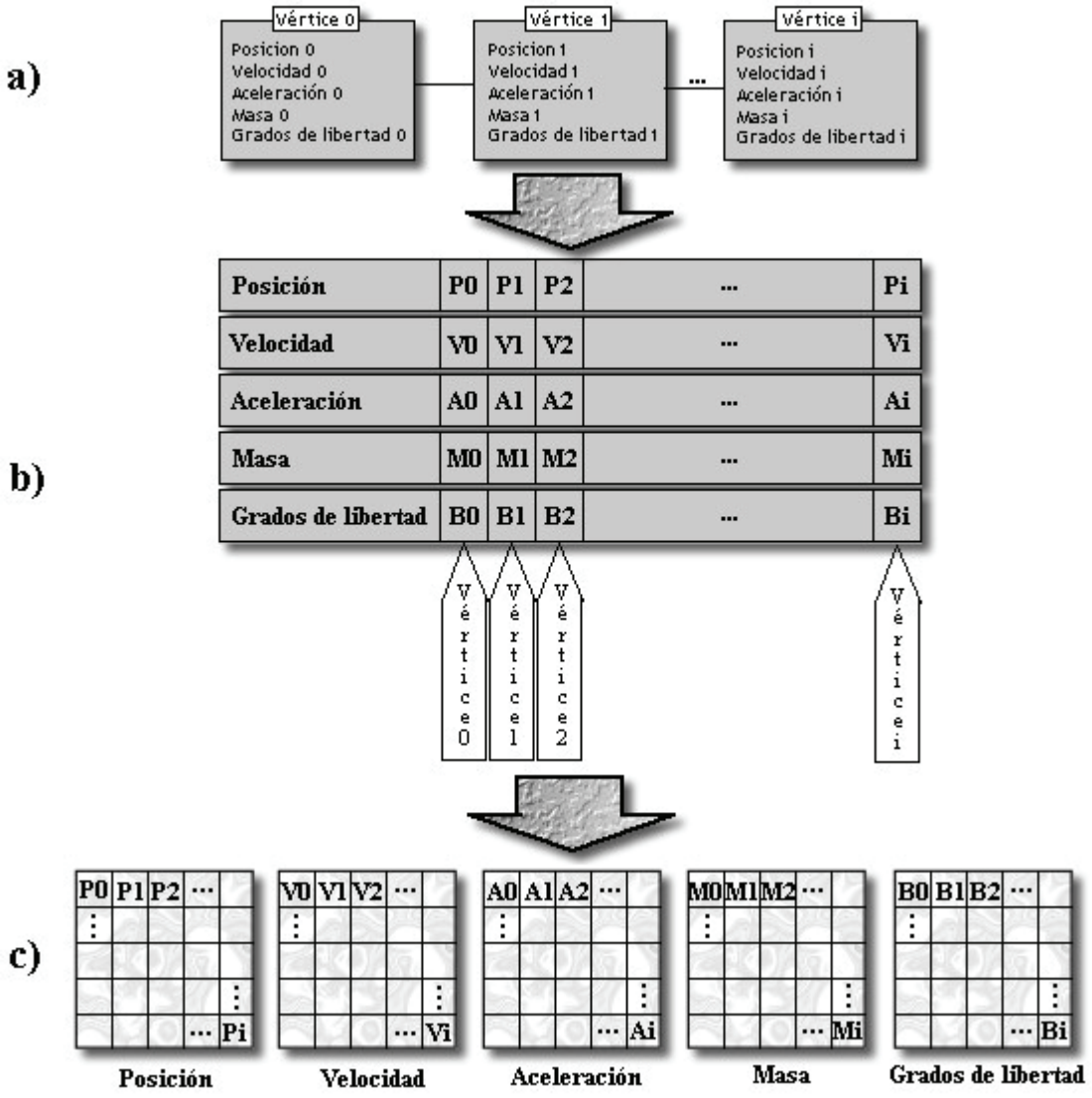

Figura 3.2. *Transformaciones en la estructura de almacenamiento de información. a) Organización original de la información. b) Transformación en arreglos unidimensionales. c) Texturas contenedoras de la información leída por la tarjeta.*

Como se vió en el capítulo 2.1, el comportamiento del cuerpo deformable obedece a un comportamiento físico determinado por una ecuación de Lagrange, en la cual intervienen fuerzas internas, que son las interacciones entre cada nodo y sus respectivos vecinos, las fuerzas externas, producidas por elementos ajenos a la

estructura que conforma el modelo de la próstata, además de la masa, viscosidad, velocidad y posición de cada vértice.

A continuación se presentan las dos secciones que son de vital importancia en la implementación del método: el cálculo de fuerzas elásticas y la derivación numérica por el método de Newton-Euler para la resolución de la ecuación de Lagrange.

3.2.1. Fuerza elástica

El cálculo de la fuerza elástica es una de las etapas que involucra mayor cantidad de procesamiento, pues para ello se requiere realizar el cómputo de las interacciones entre todos los nodos de la malla que conforma el modelo de la próstata.

La malla se compone por vértices que tienen propiedades individuales como masa, posición y velocidad, entre otras. Por otra parte, cada vértice, de acuerdo con el método de masas y resortes, está interconectado con sus vértices adyacentes, llamados vecinos, dando lugar a propiedades como el coeficiente de elasticidad y longitud inicial. De esta forma, el cambio en la posición de un vértice induce la modificación de las propiedades de sus vecinos.

Como primer paso para el cálculo de fuerzas internas es necesario ordenar toda la información en arreglos unidimensionales, ésto, como se ha mencionado, se realiza recorriendo la lista ligada y almacenando cada propiedad en su respectivo arreglo para después trasladarse a texturas. Este proceso considera que la información a almacenar por cada vértice va a corresponder e interpretarse como un píxel en formato RGB. Por ejemplo, la posición de cada vértice está siempre conformada por tres componentes, esto mismo sucede con otras propiedades vectoriales como la velocidad y la fuerza.

El cálculo de las fuerzas internas implica la interacción de todos los vértices con cada uno de sus vecinos, de manera que se requiere acceso tanto a la información del vértice en turno, como a la de cada uno de sus vecinos. Aquí hay una particularidad: el número de vecinos por vértice no es constante. A diferencia de otras propiedades vectoriales como la posición y velocidad, que siempre se representan con tres componentes, el número de vecinos es variable respecto a cada vértice. La implementación del acceso a vecinos en el modelo original del simulador se realiza por medio de apuntadores que vinculan a cada vértice con una lista de vecinos desde donde se accede a sus propiedades. Dado que el lenguaje de programación Cg no admite el uso de apuntadores, no es posible aplicar la misma forma de acceso a vecinos que se realiza con el CPU, y ésto conlleva a no poder realizar el texturizado como se realiza con la demás información. Para resolver este problema se planteó una organización alternativa, que se presenta a continuación.

El modelo de la próstata cuenta con aproximadamente 8600 vértices, entre los cuales, el número de vecinos varía desde 1 hasta 30. Con base en ésto, es posible considerar un límite máximo de vecinos y manejar un número constante para todos los vértices, aunque en la práctica el número sea siempre menor. De acuerdo a tal perspectiva, se construye una textura especial, la cual contiene el listado de vecinos por cada vértice. Hay que recordar que un vértice está compuesto por tres coordenadas, las cuales se almacenan en la textura de vértices. Para evitar redundancia de información, el listado de vecinos se compone por IDs, y no por las coordenadas en sí. Al momento de recuperar las coordenadas de cada vecino se hace uso de su ID, el cual "apunta" a la posición de un vértice dentro de la textura de vértices.

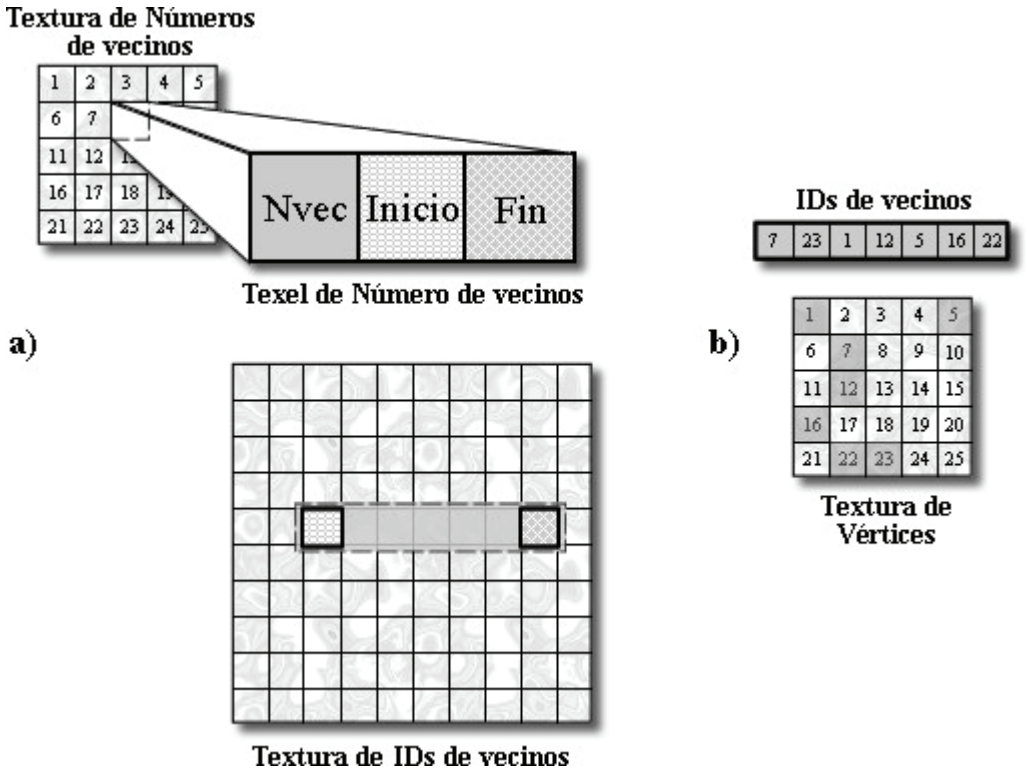

Figura 3.3. *Representación del acceso a vecinos por medio de texturas. a) La textura de números de vecinos contiene la información del número, posición inicial y final en la textura de IDs de los vecinos del vértice en turno. b) El bloque señalado en la textura de IDs hace referencia a las posiciones de los vecinos en la textura de vértices.*

Un inconveniente más que se presenta a la hora de programar sobre la tarjeta gráfica es que dado que los *shaders* son programas que se ejecutan una vez por cada vértice o fragmento, las variables que se declaran son de alcance local para cada vértice. Ésto implica que las variables no pueden ser utilizadas como un contador general, pues serían reinicializadas con cada ejecución por vértice (aún si son

declaradas con carácter global). Ya que es necesario mantener un registro de la posición inicial de las listas de vecinos para cada vértice y dado que no se puede hacer uso de variables declaradas en el *shader*, se recurre nuevamente al uso de texturas, de manera que se tiene una textura que indica por cada vértice el número y las posiciones inicial y final del primer y último vecino dentro de la textura de vecinos. El método descrito se representa en la Figura 3.3.

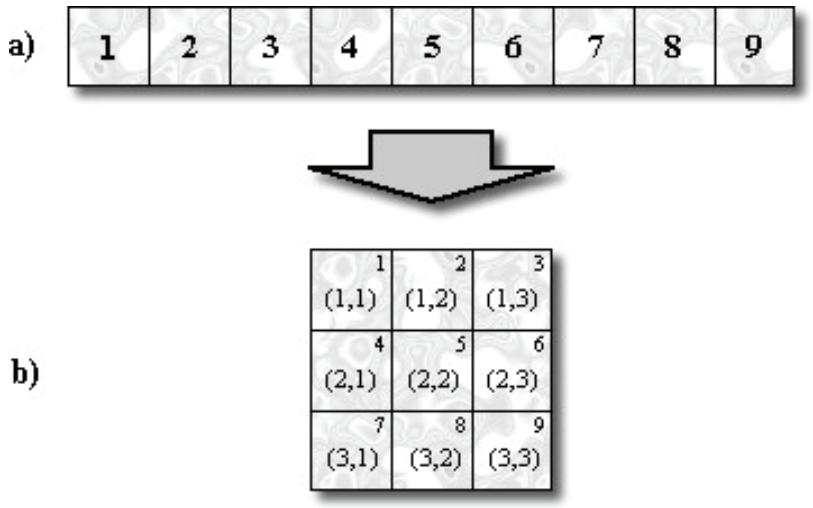

Figura 3.4. *Mapeo de coordenadas de 1D a 2D. a) Arreglo unidimensional. b) textura de tamaño 3. Se muestra la correspondencia de los elementos del arreglo con las coordenadas de la textura.*

Se puede observar que las texturas son el principal y más importante elemento para el almacenamiento de información cuando se hace programación genérica de la GPU. De igual importancia es el mapeo que se realiza para el acceso a cada elemento de la textura. En el método recién descrito se hace uso de dos nuevas texturas para el manejo de acceso a vecinos, una de las cuales (la textura de IDs de vecinos) difiere en tamaño del resto de las texturas que almacenan las propiedades de los vértices. El mapeo de las texturas de propiedades lo realiza automáticamente la GPU, utilizando un par de coordenadas, conocido como TEXCOORD0, y variándolo de acuerdo al tamaño de la textura, de manera que se logre un barrido secuencial y total. Para el caso de las texturas que contienen la información de los vecinos, el acceso dependerá de las posiciones inicial y final de las listas para cada vértice, además del ID con el cual serán mapeadas la coordenadas de cada vecino. Esto implica la implementación de una función de mapeo independiente a la utilizada inherentemente por la GPU. Así, es necesario crear una función que a partir del ID de un vértice se obtengan las coordenadas correspondientes en la textura de vértices. Ésto resulta relativamente sencillo, pues los vértices en la textura se encuentran ordenados por sus IDs, de manera que el problema se traduce en un mapeo de coordenadas de 1D a 2D, como se muestra en la Figura 3.4.

En el Programa 3.1 se muestra implementada en Cg la función de trasformación de coordenadas de 1D a 2D. Como parámetros de entrada se encuentran el ID del vértice vecino, así como el tamaño de la textura de vecinos. La función regresa el par de coordenadas correspondientes al ID especificado.

Programa 3.1

```
1  //Coordenadas de textura
2  float2 getCoord(float ID, float texSize)
3  {
4  float2 coord;
5  coord =float2(fmod(ID-0.5,texSize), floor((ID-0.6)/texSize)+0.5);
6  return (coord);
7  }
```

Una vez que se tiene establecido el manejo de la información necesaria, se procede a la programación del algoritmo para el cálculo de fuerzas elásticas internas, el cual, como se vio en el capítulo 2.1, es el siguiente:

$$g_i = \sum s_k \,, \tag{3.1}$$

$$s_k = \frac{c_k \cdot e_k}{\|r_k\|} \cdot r_k \,, \tag{3.2}$$

$$e_k = \|r_k\| - l_k \,, \tag{3.3}$$

$$r_k = x_j - x_i \,, \tag{3.4}$$

Donde s_k es la fuerza interna en el nodo *i* causada por el resorte *k,* c_k denota el coeficiente de elasticidad del resorte *k*, e_k es el valor absoluto de la deformación desde su posición inicial (l_k), r_k es el vector de distancia entre el nodo *i* y su vecino *j*, $\|r_k\|$ es el valor absoluto de la distancia del resorte *k*, y x_j denota la posición cartesiana del vector para el nodo *j*, el cual está conectado por el resorte *k* con el nodo *i*.

A continuación se presenta en el Programa 3.2 el código fuente en lenguaje Cg de la función que calcula la fuerza elástica. En las líneas 4 a 7 se declaran las variables utilizadas en el algoritmo.

Las líneas 8 a 19 recuperan los IDs de los vecinos y las longitudes iniciales de los resortes como se explica enseguida: en la línea 10 se obtiene las coordenadas para mapear a las texturas de números de vecinos y posiciones iniciales; `nVec`, almacena el número de vecinos por vértice y es declarado en la función de entrada del *shader*, `IDveci` e `initleni` son los manejadores de las texturas de vecinos y longitudes iniciales de los resortes, que son parámetros de entrada del *shader*; 205 es el tamaño de dichas texturas. La línea 11 realiza el mapeo de la textura de posiciones iniciales para recuperar la posición inicial del vecino en turno. Cada

coordenada se almacena en variables locales de la tarjeta. El mismo proceso se realiza de las líneas 15 a 18, esta vez para recuperar los IDs de cada vecino.

En las líneas 20 a 32 se implementa el algoritmo de cálculo de fuerzas internas descrito anteriormente. En la línea 24 se calcula la coordenada del vecino en turno en la textura de vértices, cuyo tamaño es 93. La línea 25 se realiza el mapeo a la textura de vértices para extraer la posición del vecino en turno. Enseguida se implementa el algoritmo de cálculo de fuerzas internas y finalmente la función regresa el valor de la fuerza interna total obtenida.

Programa 3.2

```
 1  //Fuerza elástica
 2  float3 ElasticForce()
 3  {
 4  float3 vj, vaux, Sk=float3(0.0,0.0,0.0), Rk, G=float3(0.0,0.0,0.0);
 5  float2 coordi=float2(0.0,0.0);
 6  float ark, ek, IdVec[50],initlen0[50], idin[50], id=0.0;
 7  int i=0,j=0;
 8  for (j=0;j<13;i+=3,j++)
 9        {
10        coordi=getCoord(nVec.y+j, 205);
11        vaux=float4(texRECT(initleni,coordi)).xyz;
12        initlen0[i] =  vaux.x;
13        initlen0[i+1]= vaux.y;
14        initlen0[i+2]= vaux.z;
15        vaux=float4(texRECT(IdVeci,coordi)).xyz;
16        IdVec[i] =  vaux.x;
17        IdVec[i+1]= vaux.y;
18        IdVec[i+2]= vaux.z;
19        }
20  for (i=0;i<39;i++)
21        {
22        if (IdVec[i] > 0)
23              {
24              coordi=getCoord(IdVec[i], 93);
25              vj=float4(texRECT(Pi,coordi)).xyz;
26              Rk = P-vj;
27              ark=distance(P, vj);
28              ek= ark - initlen0[i];
29              Sk= (Rk*stiff*(ark - initlen0[i]))/ark;
30              G+= Sk;
31              }
32        }
33  return G;
34  }
```

3.2.2. Método de Newton-Euler

La utilización de este método tiene como finalidad solucionar por derivación numérica la ecuación de movimiento del sistema de masas y resortes.

La ejecución de este algoritmo por parte de la GPU no conlleva ningún proceso diferente al utilizado en la implementación del método de masas y resortes. De igual forma se requiere una transformación en la estructura de los datos, y también se utilizan texturas contenedoras de la información de las propiedades de los vértices. Ya con todo ésto, basta con el simple barrido natural que hace la tarjeta grafica para acceder a toda la información. Una vez organizados todos los datos, considerando la correcta estructura de almacenamiento anteriormente descrita, resta únicamente la codificación en lenguaje Cg. Dicho método de solución numérica se muestra a continuación.

$$\mathrm{h}_i^{\,t} = \mathrm{f}_i^{\,t} - \gamma_i \cdot \mathrm{v}_i^{\,t} - g_i^{\,t}, \tag{3.5}$$

$$\left(\mathrm{v}_i^{\,t} = \frac{\mathrm{dx}_i^{\,t}}{\mathrm{d}t} \right), \tag{3.6}$$

$$\mathrm{a}_i^{\,t} = \frac{\mathrm{d}^2 \mathrm{x}_i^{\,t}}{\mathrm{d}t^2} = \frac{\mathrm{B}_i \cdot \mathrm{h}_i^{\,t}}{m_i}, \tag{3.7}$$

$$\mathrm{v}_i^{\,t+\Delta t} = \mathrm{v}_i^{\,t} + \Delta t \cdot \mathrm{a}_i^{\,t}, \tag{3.8}$$

$$\mathrm{x}_i^{\,t+\Delta t} = \mathrm{x}_i^{\,t} + \Delta t \cdot \mathrm{v}_i^{\,t+\Delta t}, \tag{3.9}$$

Donde h_i es la suma de todas las fuerzas (internas y externas) en el nodo *i*, a_i es el vector de aceleración del nodo *i*, v_i el vector de velocidad del nodo *i*, Δt es el tiempo de integración, y B_i denota la matriz de grados de libertad del nodo *i*.

El Programa 3.3 muestra el *shader* que calcula el método de masas y resortes, en donde se incluye el cálculo de fuerzas internas y el método de Newton-Euler. La línea 2 declara la estructura de salida de datos del *shader*, compuesta en este caso por un vector que representa la velocidad, asociado con la semántica de color de Cg[16]. En las líneas 3 a 18 se define la función de entrada del *shader*, además de sus parámetros de entrada, entre los que se encuentran los manejadores de las texturas que contienen la información. De las líneas 19 a 29 se declaran las variables utilizadas en el método de Newton-Euler, además, se realiza el mapeo de las texturas para obtener las propiedades del vértice en turno. En las líneas 32 a 37 se implementa el método de Newton-Euler y finalmente se almacena el resultado en la estructura de salida.

[16] La semántica asocia el valor de una variable con las variables de entrada y salida del *shader*.

Programa 3.3

```
 1  //Metodo de masas y resortes
 2  struct vout    { float3 Vn : COLOR0; };
 3  vout myr(float2 coords : TEXCOORD0,
 4               uniform samplerRECT xi,
 5               uniform samplerRECT xiold,
 6               uniform samplerRECT G2i,
 7               uniform samplerRECT Bi,
 8               uniform samplerRECT mi,
 9               uniform samplerRECT Pi,
10               uniform samplerRECT Fsi,
11               uniform samplerRECT vi,
12               uniform samplerRECT nVeci,
13               uniform samplerRECT IdVeci,
14               uniform samplerRECT initleni,
15               uniform float  dt,
16               uniform float  damp,
17               uniform float  stiff)
18  {
19  vout OUT;
20  float3 H,a,VPEN,V,F;
21  float3 x =    float4(texRECT(xi,coords)).xyz;
22  float3 xold = float4(texRECT(xiold,coords)).xyz;
23  float3 G2 =   float4(texRECT(G2i,coords)).xyz;
24  float3 B =    float4(texRECT(Bi,coords)).xyz;
25  float3 m =    float4(texRECT(mi,coords)).xyz;
26  float3 P =    float4(texRECT(Pi,coords)).xyz;
27  float3 Fs=    float4(texRECT(Fsi,coords)).xyz;
28  float3 v =    float4(texRECT(vi,coords)).xyz;
29  float3 nVec = float4(texRECT(nVeci,coords)).xyz;
30
31  G = ElasticForce();
32  VPEN = (x-xold)/dt;
33  F = -( stiff*x+damp*VPEN*x );
34  F = F*Fs;
35  H = F-damp*v-G;
36  a = (B*H)/(m.x);
37  V = v+dt*a;
38  OUT.Vn = V;
39  return OUT;
40  }
```

3.3. Paralelismo con multihilos

En esta sección se presenta el proceso de paralelización por medio de multihilos aplicado al modelo virtual del simulador de cirugía de próstata.

Como se indica en el apartado 2.4, para aplicar paralelismo en una aplicación, es necesario identificar los subproblemas y determinar si existe independencia entre ellos. En este caso, el proceso de simulación presenta diversas etapas, de las cuales se identifican dos secciones independientes: la lectura de datos de la comunicación serial y el proceso de transformaciones y visualización. De esta forma, se

contemplan dos hilos en los que se alojan dichas secciones (ver Figura 3.5). Para trabajar con aplicaciones multihilos se utilizó la API de código abierto del estándar POSIX 1003.1-2001.

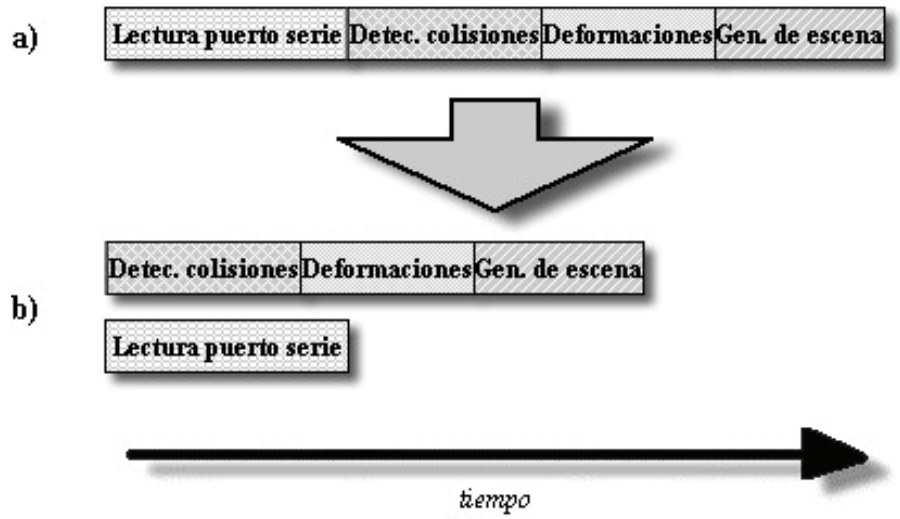

Figura 3.5. *Paralelización de las etapas principales comprendidas en el proceso de simulación. a) Procesamiento lineal: Lectura del puerto serie, detección de colisiones, cálculo de deformaciones y generación de la escena. b) Procesamiento en paralelo de etapas independientes.*

Como primer paso para la implementación multihilos se establecen las variables de hilos y las rutinas asociadas a ellos para después proceder a crearlos. El siguiente es un fragmento de código que ilustra dichas operaciones.

Programa 3.4

```
#include<pthread.h>

void *threadMain(void *arg1);
void *threadSerial(void *arg1);

void main (){
     pthread_t pthreadMain,pthreadSerial;

     pthread_create(&pthreadMain, NULL, threadMain, NULL);
     pthread_create(&pthreadSerial, NULL, threadSerial, NULL);
     pthread_join(pthreadMain, NULL);
     return 0;
}
```

El hilo denominado `pthreadSerial` contendrá las operaciones de lectura del puerto serial, y `pthreadMain` se destina a la ejecución de procedimientos de transformaciones, interacciones y visualización de la escena final. En este hilo también se incluye el manejo del cálculo de deformaciones en la GPU.

Las operaciones efectuadas en los dos hilos son bastante independientes unas de la otras. Sin embargo, es necesaria una sincronización entre hilos, pues ambos acceden a variables compartidas: `pthreadSerial` lee los datos del puerto serial y los almacena en variables. Por otro lado, `pthreadMain` lee dichas variables y en base a ellas calcula las transformaciones que se reflejarán en el siguiente cuadro de la escena. Por lo tanto, se deben proteger dichas secciones críticas para que sólo un hilo pueda acceder a la vez a las variables compartidas.

La sincronización se realiza con un candado, por lo que se define una variable tipo *mutex* y se utilizan las rutinas de bloqueo y desbloqueo en las secciones críticas pertinentes, como se muestra en el código del Programa 3.5.

Programa 3.5

```
#include<pthread.h>

pthread_mutex_t mutexvar = PTHREAD_MUTEX_INITIALIZER;

void *threadMain(void *arg1){
       /*...*/
       pthread_mutex_lock(&mutexvar);
       /* SECCION CRITIICA */
       pthread_mutex_unlock(&mutexvar);
       /*...*/
}

void *threadSerial(void *arg1){
       /*...*/
       pthread_mutex_lock(&mutexvar);
       /* SECCION CRITIICA */
       pthread_mutex_unlock(&mutexvar);
       /*...*/
}
```

De acuerdo a lo anterior, se cuenta con una implementación de paralelismo en el simulador virtual. En primer lugar, la organización de las etapas en hilos independientes propicia que la ejecución del proceso de simulación sea más ágil, pues mientras que por un lado se realiza la lectura del puerto serie, por otro se están ejecutando las rutinas de cálculo y generación de la escena, lo cual conlleva a no tener que esperar por información del puerto serie para continuar con el proceso general. Además, esta implementación con multihilos permite que tanto el CPU como la GPU mantengan un trabajo en paralelo, pues mientras que en un hilo la tarjeta se encuentra procesando la información que le fue suministrada, en otro hilo el CPU continúa trabajando, en este caso en la lectura del puerto serie. Por lo tanto, con la aplicación de la programación en paralelo se obtiene una disminución en los tiempos de espera, lo cual se ve reflejado en la disminución de los tiempos de ejecución.

Capítulo 4

Experimentos y resultados

Este capítulo muestra los resultados obtenidos en la implementación del método de masas y resortes utilizando la GPU, así como los obtenidos a partir de la aplicación de la programación multihilos para lograr paralelismo entre las diferentes etapas del proceso de simulación.

Como se mostró en el apartado 3.1, el primer resultado de importancia fue lograr que la tarjeta gráfica funcionara como un procesador de propósito general. A partir de ello se desprende la ejecución del método de masas y resortes en la tarjeta gráfica.

Dentro de la implementación de la programación multihilos y del método de masas y resortes con la GPU se comprenden algunas variantes para analizar en qué medida repercute cada una de estas etapas sobre los tiempos de respuesta del simulador. Es por ello que se generaron algunas versiones del simulador para comparar tiempos de respuesta. Para ello, se consideran principalmente dos factores que repercuten en gran medida en el proceso de simulación. Por un lado, la implementación del método de masas y resortes en la GPU se realizó considerando que la malla que conforma el modelo virtual no sufre ninguna modificación, por lo tanto los nodos son siempre los mismos y una vez que se envían a la tarjeta, no es necesario volver a enviarlos por cada ciclo. En futuras implementaciones está considerado el hecho de poder realizar cortes al modelo virtual, por lo que el número de nodos se vería afectado; es por ello que se debe considerar el constante envío de información de nodos y de cada una de sus propiedades hacia la tarjeta por cada ciclo de ejecución, lo cual produce un gasto extra en el tiempo de ejecución del cálculo de deformaciones. Por otro lado, en la implementación de paralelismo con multihilos se considera también la inclusión de un hilo extra que por el momento no realiza ninguna función pero que estará destinado a la implementación de otras funciones que pueden ser añadidas al simulador, como lo es la interacción del modelo virtual con un mecanismo de retroalimentación de fuerzas como un medio para lograr que el simulador presente mayor realismo.

De acuerdo a lo anterior, se generaron cuatro versiones del simulador de cirugía de próstata, en las cuales se consideran las variantes de aplicar o no los

aspectos antes mencionados: el envío total de datos a la tarjeta y la ejecución de un hilo extra.

El resultado de los tiempos de ejecución de cada una de las versiones se muestra en la Figura 4.1, en donde se comparan dichas versiones de acuerdo al número de cuadros por segundo sobre los que trabajan. Aquí se puede observar en primer lugar que en comparación con la versión original del simulador de cirugía de próstata, que trabaja a 13.1 Hz., las versiones implementadas con multihilos y cálculo con GPU se ejecutan con mayor rapidez. Por otro lado, es posible identificar que para el caso de las versiones 2 y 3, en donde se realiza el envío de vértices y propiedades por cada ciclo, la velocidad en la generación de un cuadro es considerablemente menor que cuando se envía dicha información una sola vez, como lo es en las versiones 4 y 5. De éstas últimas, se obtiene una mayor eficiencia cuando no se aplica la ejecución de un hilo extra. La implementación de la versión 5 del simulador resulta la más ágil, alcanzando los 57.4 Hz., seguida por la versión 4 con 51.3 Hz., la versión 3 con 37.2 Hz. y la versión 2 con 36.3 Hz.

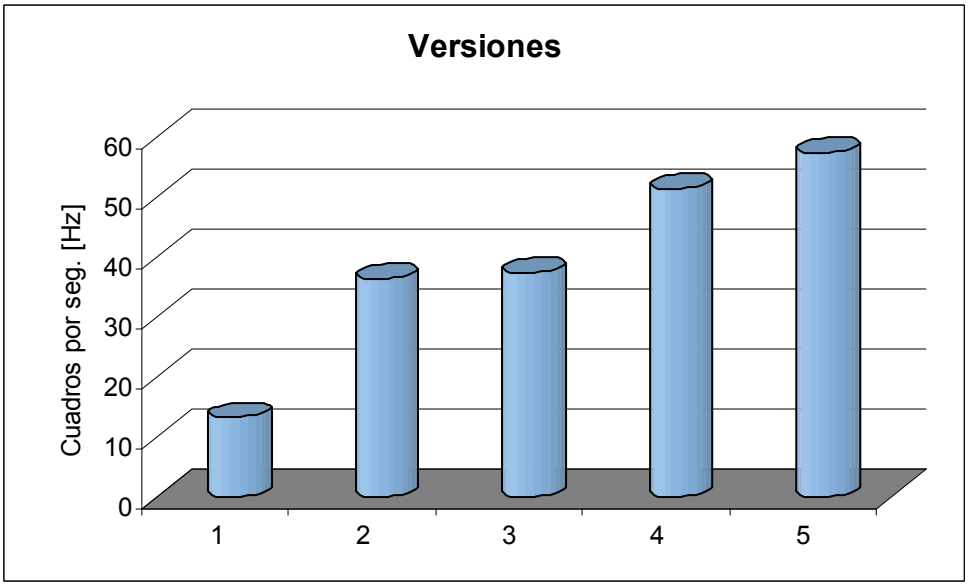

Figura 4.1. *Comparativa entre versiones del simulador. Versión1: versión original del simulador. Versión 2: Ejecución de hilo extra con envío total de datos. Versión 3: Envío total de datos sin ejecución de hilo extra. Versión 4: Ejecución de hilo extra con envío parcial de datos. Versión 5: Envío parcial de datos sin ejecución de hilo extra.*

Para fines de visualización, la versión 5 es la óptima, pues se ejecuta 4.3 veces más rápido que la versión original del simulador. En implementaciones posteriores se pueden agregar más funciones al simulador, con lo que aumentará el tiempo de cálculo de cuadros por segundo y, de acuerdo a lo observado en las versiones 2 y 3, se espera que la ejecución de funciones extras trabaje alrededor de los 35 Hz., lo cual aún es aceptable en función de la calidad visual.

Capítulo 5

Conclusiones y perspectivas

En este trabajo se presentó la implementación del método de masas y resortes ejecutado en una unidad de procesamiento de gráficos para realizar el cálculo de deformaciones en un modelo virtual para un simulador de cirugía de próstata. Además, se aplicó la programación con multihilos para lograr paralelismo en la ejecución de las etapas comprendidas en el proceso de simulación.

En primer lugar se mostró la creciente tendencia que existe desde años recientes para utilizar a las tarjetas gráficas como procesadores de propósito general. El porqué de dicha tendencia tiene que ver principalmente con el gran poder de cálculo presente en las últimas generaciones de GPUs. En este trabajo se realizaron algunas pruebas para determinar la eficiencia del uso de una tarjeta gráfica con respecto al procesador central. Se encontró que las GPUs han sobrepasado el poder de cálculo de un CPU, siendo aproximadamente cinco veces más rápidas, y esta cantidad va en aumento para los más recientes procesadores de gráficos.

Por otro lado, la implementación del método de masas y resortes en una GPU resultó ser una tarea que solicitó la resolución de diversos problemas que aparecieron desde las primeras etapas de esta implementación. En primer lugar, la utilización de la tarjeta gráfica para aplicaciones de cálculo de propósito general obligó a encontrar una forma adecuada de manejar la información por parte de la tarjeta gráfica, de manera que ésta pudiera interpretarla correctamente, ésto es, vectorizando la información para poder trasladarla a texturas. Además, se establecieron técnicas y la configuración de la aplicación principal para poder trabajar correctamente con la GPU. A partir de ello, se generó la necesidad de organizar la información en una nueva estructura de datos de manera que pudieran ser procesados eficientemente por la tarjeta gráfica. Por otra parte, las limitantes en las capacidades del lenguaje Cg, principalmente la ausencia de manejo de apuntadores, obligó a crear nuevas formas de almacenamiento de datos, por lo que fue necesaria la implementación de funciones de mapeo para el acceso a esta información.

La aplicación de la programación multihilos en el simulador de cirugía de próstata fue muy conveniente, pues ello permitió determinar las etapas de

simulación que mantenían una dependencia y en base a ésto se logró generar hilos de ejecución independientes dentro de los cuales se realizan procesos de forma paralela, lo cual resultó ser más eficientes que su correspondiente implementación en forma secuencial.

De acuerdo a los resultados obtenidos con el desarrollo de esta tesis, se puede establecer que el objetivo primordial del trabajo se alcanzó satisfactoriamente, pues se logró reducir el tiempo de ejecución de algunas de las etapas involucradas en el proceso de simulación. En comparación con la versión original del simulador, que trabajaba aproximadamente a 13 cuadros por segundo, la implementación de la programación en GPU y la aplicación de multihilos permitió que el simulador trabajara cuatro veces más rápido, alcanzando los 52 cuadros por segundo. Con esta cantidad se cuenta con una generación de cuadros bastante aceptable para que el simulador proporcione el suficiente realismo visual. Con base en estos resultados, es muy factible pensar en la disminución de tiempos de respuesta de otros modelos de simulación basando su ejecución en el cálculo con una GPU, como pueden ser el FEM (Método de Elemento Finito) o SPH (Hidrodinámica de Partículas Suavizadas).

Si bien es cierto que el simulador de cirugía de próstata aún se encuentra en fase de desarrollo, ha sido considerada la inclusión de nuevas funciones que generarán mayor tiempo de cómputo, y se espera que la tasa de cuadros por segundo no disminuya a menos de aproximadamente 30-35; estas cantidades aún representan una tasa favorable para fines de visualización.

Como se sabe, la programación de la tarjeta gráfica en este trabajo se realizó con el lenguaje de programación Cg. Dicho lenguaje ha venido actualizándose y se le han añadido nuevas funciones, y considerando que de igual forma las capacidades de las tarjetas gráficas van en aumento, se espera que la adaptación del trabajo realizado pueda generar tiempos de ejecución menores a los obtenidos a la fecha.

Por otra parte, en este trabajo también se mencionó la aparición de nuevas tecnologías que van de la mano con la aparición de nuevas tarjetas gráficas con mayores capacidades. Una de ellas es la tecnología CUDA. Parte del trabajo futuro en el desarrollo del simulador de cirugía de próstata será hacer uso de dicha tecnología y determinar la conveniencia de adoptarla como base para la utilización de la tarjeta gráfica para aplicaciones de propósito general, pues la tecnología CUDA esta diseñada para desenvolverse en dicho ámbito, a diferencia de Cg, que es entendido como un lenguaje para la generación de *shaders* para la aplicación de efectos visuales.

Apéndices

Apéndice A. Pruebas preeliminares. Código fuente.

A.1. Programa en Cg, prueba 1.

```
float4 saxpy(float2 coords : TEXCOORD0,
             uniform samplerRECT textureX,
             uniform float alpha
             ) : COLOR
{
float4 result;
float4 x = texRECT(textureX,coords);
result = alpha*x;
return result;
}
```

A.2. Programa en Cg, prueba 2.

```
float4 saxpy(float2 coords : TEXCOORD0,
             uniform samplerRECT textureX,
             uniform float alpha
             ) : COLOR
{
float4 result;
float4 x = texRECT(textureX,coords);
result = alpha*x + alpha*2*x + alpha*3*x + alpha*0.4*x + alpha*0.5*x + alpha*0.6*x +
0.5*(alpha*x+alpha*2*x+alpha*3*x+alpha*0.4*x+alpha*0.5*x+alpha*0.6*x);
return result;
}
```

A.3. Código en C, prueba 2.

```
#include <time.h>
#include <stdio.h>
#include <conio.h>
#include <windows.h>
#include <GL/glew.h>
#include <GL/glut.h>
#include <cg/cg.h>
#include <cg/cggl.h>
#define texSize 1700

CGcontext cgContext;
CGprofile fragmentProfile;
CGprogram fragmentProgram;
CGparameter yParam, xParam, alphaParam;
GLuint y_newTexID,xtex;
float data[texSize*texSize*4];
float result[texSize*texSize*4];
GLfloat alpha=2.0f;
clock_t ini, fin;
```

```
bool initCG(void);
void texturas();
void performComputation();

int main(int argc, char **argv)
{   int op;
    glutInit (&argc, argv);
       for (int i=0; i<texSize*texSize*4; i++)
              data[i] = i+1.0;
       printf("1. Calculo con CPU\n2. Calculo con GPU\nIngresa opcion: ");
       scanf("%d",&op);
       printf("\nNo. de datos: %d",texSize*texSize*4);
       switch(op)
       {
              case 1:{
                           ini = clock();
                           for(int i=0; i<texSize*texSize*4; i++)
                              data[i] = data[i]*alpha + 2*data[i]*alpha +
3*data[i]*alpha + 0.4*data[i]*alpha + 0.5*data[i]*alpha + 0.6*data[i]*alpha +
0.5*(data[i]*alpha + 2*data[i]*alpha + 3*data[i]*alpha + 0.4*data[i]*alpha +
0.5*data[i]*alpha + 0.6*data[i]*alpha);
                           fin = clock();
                           printf("\n\nCalculo con CPU:");
                        break;}
              case 2:{
                     glutCreateWindow("TEST1");
                     glewInit();
                     initCG();
                     glMatrixMode(GL_PROJECTION);
                     glLoadIdentity();
                     gluOrtho2D(0.0,texSize,0.0,texSize);
                     glMatrixMode(GL_MODELVIEW);
                     glLoadIdentity();
                     glViewport(0,0,texSize,texSize);
                     // create FBO and bind it (that is, use offscreen render target)
                     GLuint fb;
                     glGenFramebuffersEXT(1,&fb);
                     glBindFramebufferEXT(GL_FRAMEBUFFER_EXT,fb);
                     texturas();
                     ini = clock();
                     performComputation();
glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
                     glReadPixels(0,0,texSize,texSize,GL_RGBA,GL_FLOAT,result);
                     fin = clock();
                     printf("\n\nCalculo con GPU:");
                     glDeleteFramebuffersEXT(1,&fb);
                     glDeleteTextures (1,&xtex);
                     glDeleteTextures (1,&y_newTexID);
                     break;}//fin case 2
              }//fin switch
       float a=fin-ini;
       printf("\nRetardo: %f\n",(a)/CLOCKS_PER_SEC);
       getch();
    return 0;
}
```

Bibliografía

[1] Bloodshed Software (www.bloodshed.net)

[2] Brandvik, Tobias, Pullan, Graham, "Acceleration of a 3D Euler Solver Using Commodity Graphics Hardware", 46th AIAA Aerospace Sciences Meeting and Exhibit. January, 2008. Whittle Laboratory, Department of Engineering, University of Cambridge, Cambridge, UK, 2008.

[3] Cuntz, N., Strzodka, R., Kolb, A., "Real-Time Particle Level Sets with Application to Flow Visualization", Technical report, University of Siegen, Germany, 2007.

[4] Dokken ,T., Hagen, T.R., Hjelmervik, J. M. "The GPU as a high performance computational resource", In B.~Jüttler, editor, Spring Conference on Computer Graphics SCCG 2005, pp. 21–26, ACM Press, New York, 2005.

[5] England, J.N. "A system for interactive modeling of physical curved surface objects". SIGGRAPH 78 1978, 336-340. 1978.

[6] Eyles, J., Molnar, S., Poulton, J., Greer, T. and Lastra, A. "PixelFlow: The Realization". 1997 SIGGRAPH / Eurographics Workshop on Graphics Hardware 1997, ACM Press, 57-68. 1997.

[7] Fatahalian, K., Sugerman, J. y Hanrahan, P., "Understanding the Ef□ciency of GPU Algorithms for Matrix-Matrix Multiplication", Graphics Hardware, Stanford University, 2004.

[8] General-Purpose Computation Using Graphics Hardware (www.gpgpu.org)

[9] Gentoo Linux (www.gentoo.org)

[10] Harris, Mark J., "Real-Time Cloud Simulation and Rendering", Ph.D. Dissertation. UNC Technical Report #TR03-040. University of North Carolina at Chapel Hill, USA, September, 2003.

[11] Harris, Mark J., Coombe, G., Scheuermann, T. and Lastra, A. "Physically-Based Visual Simulation on Graphics Hardware". SIGGRAPH / Eurographics Workshop on Graphics Hardware 2002.

[12] Harris, Mark J., William V. Baxter III, Thorsten Scheuermann, Anselmo Lastra, "Simulation of Cloud Dynamics on Graphics Hardware", Graphics Hardware, USA, 2003.

[13] High Performance Computing. Lawrence Livermore National Laboratory (computing.llnl.gov)

[14] Hoff, K.E.I., Culver, T., Keyser, J., Lin, M. and Manocha, D. “Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware”. SIGGRAPH 1999, ACM / ACM Press, 277- 286. 1999.

[15] Kedem, G., Ishihara, Y. “Brute Force Attack on UNIX Passwords with SIMD Computer”. 8th USENIX Security Symposium 1999.

[16] Kim, Theodore, Ming C. Lin, "Visual Simulation of Ice Crystal Growth", Eurographics/SIGGRAPH Symposium on Computer Animation, USA, 2003.

[17] Kirk, David, “Cg toolkit user’s manual. A developer’s guide to programmable graphics”, Santa Clara, California, USA. 2005.

[18] Kühnapfel, U., Çakmak, H. K., Maaß, H. “Endoscopic surgery training using virtual reality and deformable tissue simulation”. Computer & Graphics, 24, pp 671-682, 2000.

[19] Lengyel, J., Reichert, M., Donald, B.R., Greenberg, D.P., "Real-time Robot Motion Planning Using Rasterizing Computer," Computer Graphics, ACM, Vol. 24, No.4, pp. 327-335. 1990

[20] NVIDIA (www.nvidia.com)

[21] NVIDIA Compute Unified Device Architecture Programming Guide, Version 1.1. Noviembre de 2007.

[22] Padilla, Castañeda M., Arámbula, Cosio F., “Deformable model of the prostate for turp surgery simulation”. Computers and Graphics 28, 2004.

[23] Padilla, Castañeda, M. "Simulación del comportamiento de tejido suave para aplicaciones de cirugía asistida por computadora". Tesis de maestría. Posgrado en ciencia e ingeniería de la computación. México, D. F., 2001.

[24] Padilla, Castañeda, M., Altamirano, Felipe, Arámbula, Fernando, Márquez, Jorge. “Mechatronic resectoscope emulator for a surgery simulation training system of the prostate”. Proceedings of the 29th Annual International Conference of the IEEE EMBS. 2007.

[25] Padilla, Castañeda, M., Pérez, Sánchez, D, “Sistema de monitoreo y análisis de datos (SMAD), utilizando la tecnología multihilos y el diseño orientado a objetos". Tesis de licenciatura. Facultad de Ingeniería, UNAM. México, D. F., 2000.

[26] Qizhi Yu, Chongcheng Chen, and Zhigeng Pan, "Parallel Genetic Algorithms on Programmable Graphics Hardware", ACM International Conference Proceeding Series; Vol. 265, Zhejiang University, China, 2005.

[27] Randima, Fernando, Kilgard, Mark J. , “The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics”. Addison-Wesley Professional, 2003.

[28] Rhoades, J., Turk, G., Bell, A., State, A., Neumann, U. and Varshney, A. “Real-Time Procedural Textures”. In Proceedings of Symposium on Interactive 3D Graphics, 1992, ACM / ACM Press, 95-100. 1992.

[29] Robbins, Kay A., Robbins, Steven, “Unix Programación práctica. Guía para la comunicación y los multihilos”. Prentice Hall, 1ª. Edición, México, 1996.

[30] Stanford Computer Graphics Laboratory (graphics.stanford.edu)

[31] The OpenGL Extension Wrangler Library (glew.sourceforge.net)

[32] The University of Toronto's Dynamic Graphics Project (www.dgp.utoronto.ca)

[33] Thompson, Chris J., Sahngyun Hahn, Mark Oskin, "Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis", International Symposium on Microarchitecture (MICRO), Turkey, Nov. 2002.

[34] Trancoso, Pedro, Charalambous, Maria, "Exploring Graphics Processor Performance for General Purpose Applications", Proceedings of the Eighth Euromicro Conference on Digital System Design, 2005.

[35] Trendall, C. and Stewart, A. J., "General calculations using graphics hardware, with application to interactive caustics", Proc. Eurographics Rendering Workshop, pp. 287-298. Springer, June 2000.

[36] YoLinux Information Portal (www.yolinux.com)