Simulación de colisiones con deformación en paralelo

Victor León Higuita Cardona
Juan Diego Toro Cano

Universidad EAFIT
Escuela de ingeniería
Departamento de Ingeniería de Sistemas
Medellín
9 de junio de 2010

Simulación de colisiones con deformación en paralelo

Victor León Higuita Cardona
Juan Diego Toro Cano

Tesis de grado para optar por el título de Ingeniero de Sistemas

Asesor: Alejandro Montoya

Universidad EAFIT
Escuela de ingeniería
Departamento de Ingeniería de Sistemas
Medellín

9 de junio de 2010

Nota de aceptación

____

____

____

____

Presidente del jurado

____

Jurado

____

Jurado

Medellín, 9 de junio de 2010

“La total diferencia entre construcción y creación es exactamente esta: que una cosa construida sólo puede ser amada después de que es construida: pero, una cosa es amada antes de que exista.”

Gilbert Keith Chesterton

Agradecimientos

Victor León Higuita Cardona:
A Dios, a mi familia y a las personas que me ayudaron en la formación profesional y en el desarrollo del trabajo de grado , en especial al Dr. Helmuth Trefftz, al asesor de tesis y al equipo de realidad virtual de la Universidad EAFIT.

Juan Diego Toro Cano:
A mi familia y a todos los que hicieron posible la realización de este proyecto mediante su asesoría, ayuda y acompañamiento; a Alejandro Montoya, asesor de la tesis, Christian Diaz que nos apoyó durante todo el proceso y a los jurados Helmuth Trefftz y Christian Trefftz.

Resumen

Hoy en día las aplicaciones que utilizan gráficos en tercera dimensión han crecido tanto en demanda como en complejidad por lo que cada vez se hace mas necesario tener una mayor capacidad de procesamiento y una mayor velocidad en la presentación de gráficos.

Por otro lado, una gran cantidad de aplicaciones científicas requieren de una capacidad de memoria mínimo, pero una capacidad de procesamiento bastante importante. La idea de éste proyecto surge de la necesidad de optimizar dicha capacidad para las aplicaciones que utilicen gráficos en tercera dimensión. Lo que se pretende entonces es realizar una aplicación para la simulación de colisiones entre objetos, para evaluar el rendimiento de la misma en dos tipos de procesadores, CPU (Central Process Unit) y GPU (Graphics Process Unit). Dicha aplicación se desarrollará utilizando un API para gráficos 3D y programas para entornos GPU.

Palabras clave: Computación Gráfica, GPU, Procesamiento en Paralelo, CPU.

Abstract

Today the quantity of applications that use 3D graphics has grown in demand as well as in complexity. That's why there is more interest on having more prossesing capacity and more execution speed on graphics rendering.

On the other hand, there are a lot of scientific applications that require few quantities of main memory, but a lot of processing capacity. The idea for this project arises from the need of optimizes that capacity for applications that use 3D graphics. So, there is going to be an application that simulates the collision between different objects in order to evaluate the performance of two kinds of processors which are CPU (Central Process Unit) and GPU (Graphics Process Unit). This application uses 3D graphics API and utilities for GPU environments.

Key words: Graphics Computer, GPU, Parallel Processing, CPU.

Introducción

GPGPU o “General-Purpose Computing on Graphics Processing Units” es un término que hace referencia al uso de las capacidades de computo de la GPU, por ejemplo, en aplicaciones 3D interactivas haciendo un manejo de recursos óptimo, como: potencia de cálculo, paralelización y optimización para cálculos de tipo flotante, aplicaciones de simulación, base de datos, algoritmos clustering. Los algoritmos de procesamiento en GPU se programan en CUDA que se integra con el compilador C, lo cual funciona en nuevas tarjetas gráficas como GeForce Serie 8 y superior, Sistemas Operativos como Windows XP, Windows Vista, Mac OS X y Linux.
CUDA permite a los desarrolladores acceso directo a las API nativas y a la memoria de las poderosas y paralelizables GPU [6].

Todo comenzó con la necesidad de mejorar la capacidad de procesamiento de gráficos que tenían los diferentes dispositivos existentes, pero una vez alcanzada esta meta, se vio el potencial que tenían estos mismos para ser utilizados en aplicaciones diferentes a simplemente dibujar gráficos en el monitor.

GPGPU engloba todas las técnicas que se utilizan para aprovechar al máximo la gran capacidad de procesamiento que tienen las tarjetas gráficas, permitiendo entonces utilizarlas para llevar a cabo desde tareas sencillas tales como la búsqueda de valores o el ordenamiento de arreglos, hasta el procesamiento de cálculos científicos o la simulación del movimiento de una partícula.

El propósito de este trabajo es la creación de una aplicación de simulación de colisiones que sea paralelizable a través de GPU, es decir, que se pueda ejecutar tanto en la CPU del computador como en la GPU de la tarjeta gráfica. Tal desarrollo puede ser de gran interés en áreas tales como la Realidad Virtual, la Física, la telemedicina y algunas de las ramas de la ingeniería [6].

En el capítulo uno se hace la introducción del proyecto donde se plantean los objetivos, el alcance, los productos o entregables y la importancia en la carrera del proyecto realizado. En el capítulo dos se presentan algunos conceptos que faciliten la comprensión del proyecto (marco teórico); en éste capítulo se explica lo que es CUDA, el algoritmo de detección de colisiones y el modelo físico de deformación para la construcción de la aplicación. En el capítulo tres se presentan algunos trabajos relacionados con el proyecto. En el capítulo cuatro se muestra los resultados obtenidos en el experimento y en base a ellos se sacaran conclusiones. En el capítulo cinco se presentará la posibilidad de un trabajo posterior que pueda complementar este proyecto y finalmente en el capítulo seis se incluirán algunos anexos tales como manual de usuario de la aplicación y de CUDA en Windows.

Índice general

Resumen . . . 6
Abstract . . . 7
Introducción . . . 8

1. Proyecto **13**
Planteamiento del problema . . . 13
Justificación . . . 14
Importancia del proyecto para la carrera . . . 15
1.1. Objetivo . . . 16
1.2. Objetivos específicos . . . 16
1.3. Alcance . . . 16
1.4. Metodología . . . 16
1.4.1. Búsqueda de información . . . 16
1.4.2. Desarrollo del mundo 3D . . . 16
1.4.3. Construcción del modelo físico (deformación) . . . 17
1.4.4. Construcción del modelo físico (colisión) . . . 17
1.4.5. Cuantificación del desempeño . . . 17
1.4.6. Guías de la aplicación . . . 17
1.5. Entregables . . . 17
1.6. Verificación . . . 17
1.7. Fase de finalización . . . 18

2. Marco teórico **19**
2.1. Áreas beneficiadas . . . 20
2.2. Programación paralela . . . 22
2.2.1. Paradigmas de Programación Paralela . . . 24
2.3. Cuda . . . 28
2.3.1. Ventajas . . . 28
2.3.2. Limitaciones . . . 28
2.3.3. Arquitectura . . . 28
2.3.4. Ambiente de desarrollo en CUDA . . . 29
2.3.5. El modelo cuda . . . 29
2.3.6. Kernel . . . 30
2.3.7. Invocaciones a un Núcleo (Kernel) . . . 30
2.3.8. Sincronización . . . 30
2.4. Arquitectura GPU . . . 31
2.4.1. Microarquitectura de Soporte para la Ejecución Paralela [26] . . . 32
2.4.2. Espacios de Memoria [26] . . . 33
2.4.3. Optimización de la Aplicación [26] . . . 33

2.5. Estructuras de datos espaciales . . . 35
2.5.1. Bounding Volumes Hierarchies . . . 35
2.5.2. Árboles . . . 36
2.5.3. Árboles binarios . . . 36
2.5.4. Cálculo de árbol binario . . . 37
2.6. Técnicas usadas en la detección de colisiones . . . 40
2.6.1. Detección de colisiones con rayos . . . 40
2.6.2. Detección de colisiones usando estructuras jerárquicas . . . 41
2.7. Técnica usada para la detección de colisiones . . . 42
2.8. Modelo de deformación (mass-spring model) . . . 44
2.9. Método de deformación . . . 47
2.9.1. Pseudocódigo de deformación . . . 47
2.9.2. Calculo de la Deformación de la Figura a partir del Modelo Físico . . . 48

3. Estado del arte **49**

4. Resultados **53**
4.1. Pruebas . . . 53
4.2. Comparación de la ejecución del modelo Físico en la CPU y la GPU . . . 53
4.3. Resultados de acuerdo a la eficiencia del modelo físico. . . . 53
4.4. Paralelización del algoritmo propuesto . . . 55
4.5. Dificultades en la implementación . . . 56
4.6. Conclusiones . . . 57

5. Anexo 1. Trabajo futuro **59**

6. Anexo 2. Palabras clave **61**
6.1. Palabras clave . . . 61
6.2. Key words . . . 62

7. Anexo 3. Manuales de usuario **63**
7.1. CUDA . . . 63
7.2. Ejemplo en CUDA . . . 66
7.3. Manual de uso de la aplicación . . . 71

Bibliografía **75**

Índice de figuras

2.1. Funcionamiento del sistema . . . 19
2.2. Modelo de Michael Fynn. Tomado de [17] . . . 24
2.3. Modelo SISD (Single Instruction Single Data). Tomado de [17] . . . 24
2.4. Modelo SIMD (Single Instruction Multiple Data). Tomado de [17] . . . 25
2.5. Modelo MIMD (Multiple Instruction Multiple Data). Tomado de [17] . . . 25
2.6. Modelo Memoria Compartida. Tomado de [17] . . . 26
2.7. Modelo Memoria Distribuida. Tomado de [17] . . . 27
2.8. Modelo Multiple Instruction Single Data. Tomado de [17] . . . 27
2.9. Vista General de la Arquitectura CUDA. Tomado de 18. . . . 29
2.10. Comparación GPU - CPU Tomado de: [15]. . . . 31
2.11. Características del procesador y memoria de la Geforce 8800. El código ejecuta en el procesador puede ser accesible desde cualquier espacio de memoria. Las memorias mas cercas al procesador (columna izquierda) son más pequeña y rápidas que las más alejadas (columna derecha). Tomado de: [26]. . . . 32
2.12. La transmisión del valor entre todos los hilos se muestra por las flechas horizontales cruzando los hilos en la urdimbre. Tomado de: [26]. . . . 33
2.13. Ejemplo BVH. Tomado de [29] . . . 36
2.14. Ejemplo BVH . . . 37
2.15. Baricentro de un triángulo . . . 38
2.16. Construcción de árbol. El nodo raíz contiene todos los baricentros de la figura. Las hojas solo contienen un baricentro. . . . 39
2.17. Método de detección de colisiones con rayos. La colisión se produce cuando distancia entre el origen del rayo y el medio (carretera) es cero. Tomado de [29] . . . 41
2.18. Distancias P Q . . . 42
2.19. Búsqueda de colisión. . . . 43
2.20. Malla triangular que muestra un punto con sus respectivos vecinos, esto aplica a todos los puntos de la malla (Área de deformación). . . . 44
2.21. Objeto representado en puntos que muestra su área deformable (los puntos en rojo) . . 45
2.22. Esquema de resortes de un punto de la figura con masa m . . . 45
2.23. Momento a la respuesta de la colisión del objeto en caída al instante de chocar . . . 47
2.24. Estado inicial y final del cuerpo tridimensional al aplicar el modelo físico . . . 48

3.1. Deformaciones de objetos 3d usando métodos largos. Tomado de: [24]. . . . 51
3.2. La geometría que no está deformada q_i^0 es registrada con la geometría deformada p_i. La matriz A es computada para reducir el error. Tomado de: [21]. . . . 51

4.1. Tiempo en milisegundos de cada figura (representada por el No. Nodos) al probar el modelo físico en la CPU y GPU (usando 16 Bloques, 32 Bloques, 64 Bloques, 128 Bloques o 256 Bloques). 54
4.2. reconstrucción de los tiempos por Nodos de la malla en la CPU y GPU 54
4.3. SpeedUp(t_{cpu}/t_{gpu}) de cada figura por el numero de bloques 55
4.4. Speedup por bloque de cada figura 56

7.1. Archivos fuentes del proyecto 71
7.2. Propiedades del proyecto 71
7.3. Opciones de simulación 72
7.4. Malla de los objetos 3d 73
7.5. Objetos sólidos 3d 73
7.6. Archivo de texto con la lista de figuras disponibles 74
7.7. Menú con la lista de figuras disponibles 74

Capítulo 1

Proyecto

Planteamiento del problema

En física el tema de colisiones hace referencia al evento en el que dos o más objetos en movimiento (cuerpos colisionantes) ejercen fuerzas relativas entre sí por un periodo de tiempo generalmente corto. A continuación, y teniendo en cuenta si estos son deformables o no, su movimiento resultante tendrá determinada velocidad y dirección, y además ellos mismos podrán tener o no una nueva configuración o forma.

Por colisión elástica se entiende el tipo de colisión entre dos o más objetos en la que estos no sufren deformaciones permanentes durante el impacto. En una colisión elástica se conserva tanto el momento lineal como la energía cinética del sistema. No hay un intercambio de masas entre los cuerpos al separarse después del choque.
En mecánica se hace referencia a una colisión perfectamente elástica cuando se conserva la energía cinética del sistema formado por las dos masas (m1 y m2) que chocan entre sí. Los choques entre bolas de billar son un buen ejemplo, ya que la energía cinética allí se mantiene antes y después de la colisión [5].

Ahora bien, el objetivo del proyecto que pretendemos llevar a cabo es la creación de una aplicación para la simulación por computador de colisiones entre objetos 3D deformables. Para ello es necesario el diseño y la implementación de dos algoritmos principalmente. En primer lugar, uno que nos permita determinar en qué momento chocan los objetos y qué primitivas (triángulos, puntos, líneas, etc) están participando de la colisión. En segundo lugar, uno que se encargue de calcular cual será la nueva configuración de las mallas que representan los objetos tras la colisión.

Como es de prever, ambos algoritmos deben estar sincronizados entre sí y ser ejecutados secuencialmente, pues, a cada movimiento de un objeto, se debe determinar primero si este se está colisionando, y si lo está haciendo, se debe saber que parte es la que está en contacto para poder calcular más adelante cómo se deformará. Este modelo físico de deformación será paralelizado por la tarjeta gráfica con los cálculos de fuerza en los puntos de contacto en la figura deformable.

Cabe señalar que al momento de la colisión se tienen en cuenta el peso y el material del objeto deformable.

Justificación

Hoy en día, las aplicaciones que me permiten recrear determinada experiencia, ver el desarrollo de determinado evento "antes que este ocurra", o simplemente experimentar aquí lo que ocurre en otro lugar mediante simulaciones y ambientes de realidad virtual han crecido no solo en demanda sino también en complejidad.
Un ejemplo de ellas son los juegos, la telemedicina, la telepresencia, los simuladores de vuelo, entre otras.

En ese sentido nuestro proyecto es un aporte que permite hacer dichas simulaciones aun más reales, permitiendo que los usuarios de dichas aplicaciones tengan una experiencia más cercana a la realidad.

Por otro lado, cabe también notar nuestro interés en el tema y en que nuestro aporte pueda ser intervenido y/o mejorado por diferentes entidades como empresas desarrolladoras de videojuegos, instituciones académicas y otras personas interesadas en el tema.

Importancia del proyecto para la carrera

Uno de los objetivos de la carrera de ingeniería de sistemas es "aprender a crear y aplicar tecnologías informáticas para el beneficio de los individuos, de las organizaciones y del país". En ese sentido, nuestro proyecto es de gran importancia e interés pues sus posibles aplicaciones para la mejora de desarrollos y nuevos proyectos en áreas como la realidad virtual, las simulaciones, los videojuegos y la educación pueden beneficiar a gran cantidad de personas y ser útil en diferentes ramas tecnológicas.

Por otro lado, es importante en el sentido que estamos aprendiendo a usar una nueva tecnología como lo es la programación de GPUs de tarjetas gráficas NVIDIA a través de CUDA.

Es importante mencionar también que los principales temas tratados en este proyecto son la física implicada en las colisiones; necesaria para diseñar los algoritmos a utilizar, la computación gráfica; cuyos conceptos son necesarios para elaborar correctamente y de la manera más eficiente la simulación, la programación; puesto que se trata de una aplicación, y la ingeniería de software; utilizada para desarrollar de la manera más adecuada el proyecto. Todos estos temas, a excepción de la computación gráfica, son ampliamente utilizados a lo largo de la carrera y por lo tanto importantes.

1.1. Objetivo

Desarrollar una aplicación para la simulación en tercera dimensión de colisiones entre objetos deformables, que permita evaluar su rendimiento tanto en una CPU como en una GPU.

1.2. Objetivos específicos

- Crear objetos en tercera dimensión (3D) utilizando OpenGL y 3ds Max.
- Utilizar CUDA para la programación en GPU.
- Crear una interfaz adecuada para permitirle al usuario la simulación de colisiones.
- Realizar pruebas con la medición del tiempo para evaluar el rendimiento tanto en CPU como en GPU.

1.3. Alcance

Programa que corre en una maquina tipo PC con GPU (NVIDIA GeForce 9400 GT), donde se puede simular la colisión entre dos objetos. Se puede escoger en que "ambiente" se va a realizar la simulación, si en CPU o en GPU además de diferentes aspectos relativos a la simulación que son: movimientos de cámara, presentación de los objetos en mayas o con textura.

1.4. Metodología

La metodología propuesta consta de diferentes etapas que van desde la creación inicial de objetos en 3D en 3ds MAX, la familiarización con CUDA y las tarjetas gráficas hasta la finalización como tal de la aplicación.

Dichas etapas son:

1.4.1. Búsqueda de información

Se realizó una búsqueda y revisión bibliográfica en artículos científicos, libros y páginas Web con el fin de obtener las bases teóricas necesarias para el proyecto. Se hizo una aproximación a CUDA que consistió en la descarga de su compilador de la página de NVIDIA [1] y la puesta en marcha de diferentes ejemplos incluidos para aprender su uso.

1.4.2. Desarrollo del mundo 3D

Se empleó 3ds Max, Blender y OpenGL para C++. Con el objetivo de familiarizarse con OpenGL, se creó una aplicación simple que consta de un modelo del planeta tierra girando sobre su eje. Dicho modelo fue creado en Blender con formato .3ds que es el utilizado en 3ds Max. Se busco un cargador para los objetos .3ds (3ds Max) en el mundo en tercera dimensión y se implementó un mundo en tercera dimensión donde se puede observar los cuerpos y la escena. El mundo también permite visualizar los objetos a través de la cámara.

1.4.3. Construcción del modelo físico (deformación)

En esta etapa se buscó un modelo físico (Mass-Spring) basado en los simuladores quirúrgicos para la implementación de la deformación de la figura tanto en la CPU como la GPU. Se delimitó el espacio del trabajo y se empleó CUDA para el desarrollo del modelo en la GPU.

1.4.4. Construcción del modelo físico (colisión)

Se indagó acerca de los diferentes métodos existentes para la detección de colisiones entre los diferentes objetos que forman parte de la escena. Se implementó una aproximación utilizando Bounding Volumes Hierarquies (Jerarquías de volúmenes envolventes) en la CPU que emplea jerarquías para la subdivisión del espacio y una implementación propia para la resolución de la colisión, en la que se determinan los polígonos de la malla, en nuestro caso, triángulos, que entran en colisión.

1.4.5. Cuantificación del desempeño

Se evaluó el desempeño del programa usando la CPU y la GPU. Se llevaron a cabo las pruebas de desempeño donde se midió el tiempo que le tomaba el programa para realizar los cálculos de deformación. Además se observó la optimización de los cálculos del modelo físico aprovechando la potencia de la GPU. Las pruebas de desempeño arrojaron la necesidad de llevar a cabo las tareas más rápido en la GPU usando la programación paralela que en la CPU.

1.4.6. Guías de la aplicación

Por último se realizó el presente documento y un pequeño manual del funcionamiento de la aplicación.

1.5. Entregables

Se entregarán los siguientes productos:

- Anteproyecto.
- Trabajo escrito bajo los alineamientos de la universidad.
- Presentación oral con material de apoyo visual.
- Documentación del código fuente.
- Código fuente de la aplicación.
- Manual de usuario.
- CD con el proyecto.

1.6. Verificación

- Realizar pruebas de la aplicación.
- Identificar mejoras de la aplicación.
- Plantear una metodología para los experimentos.
- Optimizar el uso de la aplicación.

1.7. Fase de finalización

- Posibles mejoras a la aplicación.
- Realizar experimentos para recolectar datos.
- Sacar conclusiones a partir de los resultados.
- Entrega del informe final a los jurados.
- Lectura del informe final por parte de los jurados.
- Sustentación del trabajo de grado ante los jurados.
- Publicación del trabajo de grado.

Capítulo 2

Marco teórico

El marco teórico está dividido en dos secciones. En la primera, se revisan los conceptos necesarios a tener en cuenta para implementar el algoritmo de detección de colisiones y en la segunda se revisa de la misma forma el modelo físico de deformación.
A continuación se muestra un diagrama que conecta los principales elementos de la aplicación mediante el orden en que se presentan durante la ejecución:

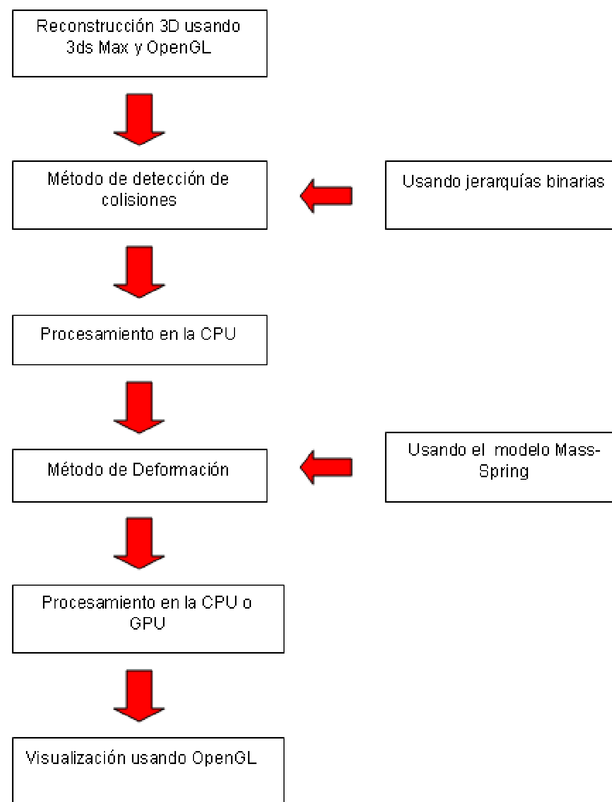


Figura 2.1: Funcionamiento del sistema

Por cada frame el proceso se repite partiendo del método de detección de colisiones hasta la deformación del objeto y posterior rebote.

2.1. Áreas beneficiadas

Este tipo de tecnología se aplica en diversas áreas tales como videojuegos, cálculos científicos, diseño asistido por computador, simulación, entre otras, lo cual ha ayudado a mejorar el rendimiento y la velocidad en el procesamiento de los cálculos más complejos. Existe un amplio rango de áreas que utilizan la GPU como medio para optimizar el rendimiento de muchas de sus aplicaciones. A continuación se mencionan algunas:

Medicina
Se utiliza CUDA en diversas aplicaciones ya que acelera el rendimiento de muchas aplicaciones que requieren de cálculos complejos. Por ejemplo AMBER es un programa de simulación de dinámica molecular utilizado por más de 60000 investigadores en el área académica y a nivel molecular para el descubrimiento de nuevos medicamentos [19].

Videojuegos
Se aplica para mejorar el rendimiento en los cálculos físicos y en el aceleramiento de los gráficos. Algunos son: Star Tales y PhysX Screensaver

Geología
Con Geogestar, un proveedor de tecnología geológica ha desarrollado un nuevo dispositivo, con su respectivo software, que está basado en las tarjetas de video NVIDIA TESLA. Dicho dispositivo se usa para manejar algoritmos sísmicos de gran complejidad para evaluar las condiciones de superficie de un área geológica determinada muy usada en la exploración de yacimientos de petróleo [11].

CAD/CAM/CAE
OptiTex en Israel ha tomado parte en el diseño asistido por computador en su nuevo nivel de tercera dimensión para la moda virtual usando CUDA para el desarrollo de ambientes de simulación para la reconstrucción del vestuario y la ejecución de los algoritmos en la GPU. También aplicable para los sistemas CAD en la precisión de las piezas mecánicas y en la fabricación de cualquier producto [12].

Realidad Virtual
En este caso, se aplica para el aumento en la aceleración y la mejora en la visualización de los gráficos para realidad virtual fotorrealística. Un ejemplo de esto es una aplicación en tercera dimensión (que utiliza los gráficos de NVIDIA) desarrollada en la NASA que permite a los científicos explorar Marte como si en realidad estuvieran sumergidos en el planeta. En un futuro se tiene previsto utilizar esta aplicación para simular el mundo real de la manera más vivida posible. [13].

Animación 3D
En este campo se han explorado técnicas avanzadas de animación que proporcionan un mayor grado de realidad en las animaciones y que han sido empleadas en videojuegos y en la industria del cine. Un ejemplo es la película “Cloudy With a Chance of Meatballs”, en la que se utilizaron tarjetas NVIDIA para la aceleración de gráficos y efectos visuales [20]. Por otro lado, se han desarrollado videojuegos con técnicas más realistas en la animación de personajes, usando las tarjetas NVIDIA para mejorar el rendimiento en la ejecución de cada uno de los escenarios de la aplicación.

Procesamiento de imágenes

Se emplea para la digitalización de las imágenes con una mayor calidad, acelerando el procesamiento de los cálculos en la GPU usando Connected Component Labeling (CCL), un algoritmo empleado para la digitalización de las imágenes [19].

2.2. Programación paralela

El paralelismo está relacionado a la simultaneidad, es decir, a las actividades que se realizan en forma paralela; por ejemplo, el cuerpo humano realiza actividades simultáneas como las señales eléctricas emitidas por las neuronas, lo mismo este fenómeno se presenta el ambiente que nos rodea. Cualquier cuerpo estelar realiza movimientos de forma permanente y la interacción con los otros objetos es simultánea ya que están sometidos a la fuerza de gravedad.
La programación paralela está asociada a la ejecución de un programa simultáneamente en varios procesadores, lo cual hace óptima la ejecución de tareas. En el otro extremo está la programación secuencial, en la que un sólo procesador ejecuta un programa instrucción por instrucción, siendo esto menos óptimo. [17].

En ese sentido, la programación paralela se utiliza en aplicaciones que necesiten un gran rendimiento o que necesiten manejar una gran cantidad de datos en poco tiempo. Sin embargo, la gran desventaja de este tipo de sistemas es su elevado costo, pese a su efectividad.

Dentro de las posibles aplicaciones para la programación o cómputo paralelo se encuentran: el procesamiento de imágenes, la predicción del clima, los sistemas de información geográficos, el modelado de corrientes marinas, entre otras [16].

Este modelo de programación ofrece:

- Alternativas de sincronizadores para mejorar el rendimiento.
- Aplicación a todos los niveles de un sistema de computación.
- Mayor rapidez en el procesamiento de los cálculos.
- Tendencias en la tecnología, arquitectura y economía.
- Apoyo al cálculo científico en áreas como la física, la química, la biología, la oceanografía y la astronomía.
- Apoyo al aceleramiento de cómputo de video, de aplicaciones CAD, gestión de base de datos, etc.
- Una forma de mejorar el rendimiento en las maquinas.
- En redes de multiprocesadores y computadores paralelos masivos.

Paralelismo y Cómputo Paralelo
El paralelismo es la realización de varias actividades relacionadas entre sí de manera simultánea mientras que el cómputo paralelo es la ejecución de más de un cálculo simultáneamente en diferentes procesadores [17], se trata de la ejecución de un trabajo distribuyendo las tareas entre distintos procesadores disponibles, para aumentar el desempeño del sistema y obtener una mayor velocidad en la ejecución de los cálculos [16].

Computación Paralela
Este paradigma de programación se enfoca en el procesamiento de datos concurrentes pertenecientes a uno o más procesos encaminados hacia un objetivo común. Puede ser definida como “Estrategia para la ejecución de tareas grandes y complejas con mayor rapidez” [17] e implica, grosso modo, los siguientes pasos:

- Descomponer un algoritmo en partes para ser ejecutadas por separado.
- Distribuir las partes como tareas que son llevadas a cabo por múltiples procesadores simultáneamente.
- Coordinar y sincronizar las tareas entre los procesadores

Por otro lado, entre los aspectos o temas más importantes que se deben tener en cuenta cuando se trabaja con este paradigma son:

- El diseño de algoritmos eficientes.
- El diseño de computadores paralelos.
- La evaluación de los algoritmos paralelos.
- Lenguajes de programación paralela.
- Portabilidad de los programas paralelos.

Computadores Paralelos

Un computador paralelo es un conjunto de procesadores encaminados a resolver un problema computacional. Estos sistemas son interesantes porque ofrecen recursos potenciales en materia de computación entre los que están las supercomputadores paralelas que tienen cientos o miles de procesadores, redes de estaciones de trabajo que manejan cientos de transacciones, etc. Se debe de considerar el tipo de arquitectura paralela y el tipo de comunicación entre los procesadores que está siendo usada [17].

2.2.1. Paradigmas de Programación Paralela

Basadas en Arquitecturas Paralelas

En 1966, Mychael Fynn propuso un método para clasificar las arquitecturas de los computadores que se basa en el numero de instrucciones y secuencias de datos que la maquina procesa. Puede haber secuencia de instrucciones sencillas o múltiples y secuencias de datos sencillas o múltiples dependiendo del modelo de computación [17].

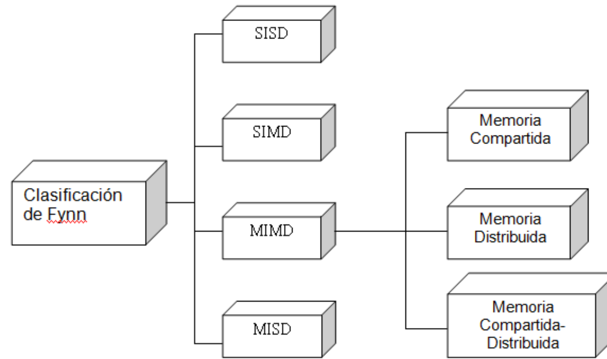


Figura 2.2: Modelo de Michael Fynn. Tomado de [17]

SISD (Single Instruction Single Data)

Modelo de computación secuencial, donde el procesador o la CPU recibe una secuencia de instrucciones que opera en una secuencia de datos [16].

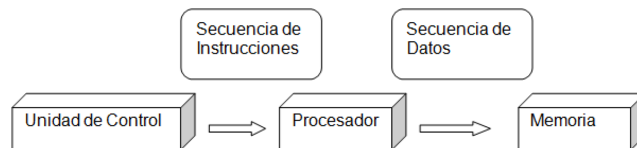


Figura 2.3: Modelo SISD (Single Instruction Single Data). Tomado de [17]

Ejemplo: Para la suma de N números $a, a, a, \ldots, a$, el procesador necesita acceder a memoria N veces consecutivas (para recibir un numero), ejecutando $N-1$ adiciones en secuencia, es decir, de forma secuencial ya que están contenidas dentro un solo procesador que es el que se encarga llevar a cabo la tarea completa [17].

SIMD (Single Instruction Multiple Data)

A diferencia del método SISD, existen multiples procesadores que se encargan de sincronizar la ejecución de una misma tarea. Esto se hace repartiendo cada instrucción de dicha tarea entre los diferentes procesadores y utilizando un reloj o temporizador global que permite coordinar la operación [17].

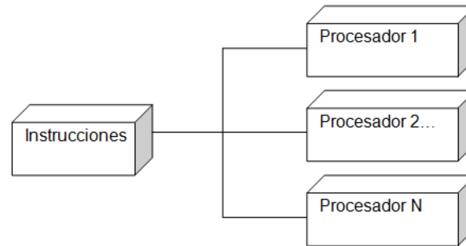


Figura 2.4: Modelo SIMD (Single Instruction Multiple Data). Tomado de [17]

Ejemplo: Sumar las matrices A y B de orden 2, si la suma se realiza con cuatro procesadores, entonces se ejecutan las siguientes instrucciones simultáneamente:

$$A_{11} + B_{11} = C_{11},\ A_{12} + B_{12} = C_{12}$$
$$A_{21} + B_{21} = C_{21},\ A_{22} + B_{22} = C_{22}$$

En una maquina secuencial esta suma de matrices se realiza en 4 pasos mientras con este modelo se hace en un solo paso. [17]

MIMD (Multiple Instruction Multiple Data)

Tiene las mismas características que la SIMD, pero a diferencia de esta es asincrónica. No tiene un temporizador central. En este sistema cada procesador puede ejecutar su propia secuencia de instrucciones y poseer sus propios datos [17].

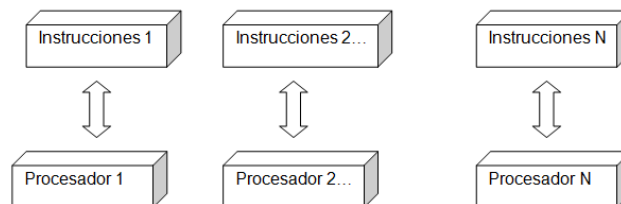


Figura 2.5: Modelo MIMD (Multiple Instruction Multiple Data). Tomado de [17]

Se tiene N procesadores, N secuencias de instrucciones y N secuencias de datos. Cada procesador ejecuta su propia secuencia de instrucciones con diferentes datos, es decir de forma asincrónica [17]. Se clasifican en:

- Sistemas de Memoria Compartida

- Sistemas de Memoria Distribuida
- Sistemas de Memoria Compartida-Distribuida

Sistemas de Memoria Compartida

En este sistema el procesador tiene acceso al área de memoria que esta compartido. Los tiempos son uniformes, pues todos los procesadores están igualmente comunicados con la memoria principal, las lecturas y escrituras tienen la misma latencia, el acceso a memoria se hace por medio de un canal común. La principal desventaja se da cuando se intenta un acceso simultáneo a memoria y no se tiene un mecanismo de sincronización. Una de las ventajas principales es que son más fáciles de diseñar que los sistemas de memoria distribuida [17].
Las MIMD con memoria compartida son conocidas como sistemas de multiprocesamiento simétrico (SMP), donde múltiples procesadores comparten un mismo sistema operativo o memoria.

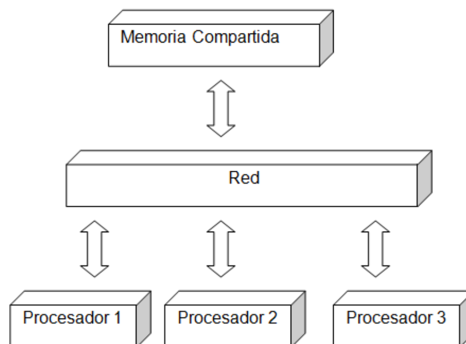


Figura 2.6: Modelo Memoria Compartida. Tomado de [17]

Sistemas de Memoria Distribuida

En estos sistemas, cada procesador tiene asignada su propia memoria local y además están separados físicamente uno de otro, pero todos están conectados y se comunican a través de una red. En el caso que uno requiera los datos contenidos en la memoria de otro, se hace una petición solicitándolos; a esto se le conoce como paso de mensajes.
Ejemplos de estos sistemas son: sistemas Cliente/Servidor, RPC, Sockets, entre otros [16].

Los sistemas MIMD de memoria distribuida son conocidos como sistemas de procesamiento masivamente paralelos (MPP).

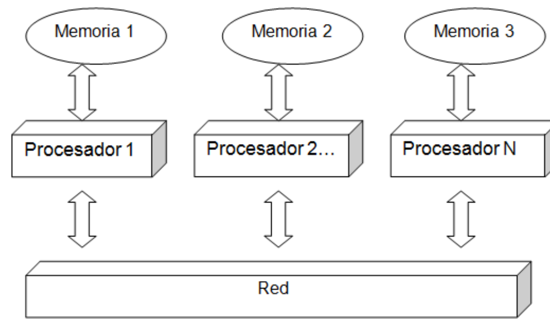


Figura 2.7: Modelo Memoria Distribuida. Tomado de [17]

Sistemas de Memoria Compartida-Distribuida

A diferencia del anterior, aunque los diferentes procesadores tengan su propio espacio de memoria, todos ven un único espacio de direcciones como una memoria global compartida [16].

MISD (Multiple Instruction Single Data)

En este modelo las secuencias de instrucciones pasan a través de N procesadores que comparten una memoria global [17].

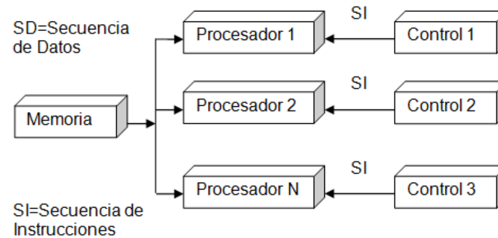


Figura 2.8: Modelo Multiple Instruction Single Data. Tomado de [17]

Existen N programas/algoritmos y una secuencia de datos. Cada procesador ejecuta diferentes secuencias de un algoritmo con diferentes datos en cada caso.[17].

2.3. Cuda

NVIDIA® CUDATM es un acrónimo para Compute Unified Device Arquitecture (CUDA). CUDA es un modelo de programación en paralelo y ambiente software que aumenta la potencia computacional de la GPU (Graphics Processor Unit).

La tecnología CUDA ha sido diseñada con varias metas dentro de las que se encuentran:

- Permitir a los programadores usar una variación del lenguaje de programación C para codificar algoritmos y ejecutarlos en GPUs nvidia.
- Permitir la implementación de algoritmos en paralelo [8].
- Soportar computación heterogénea en la que las aplicaciones puedan usar tanto la GPU como la CPU. Porciones seriales de las aplicaciones pueden estar corriendo en la CPU mientras que porciones paralelas se descargan en la GPU para ser procesadas. La CPU y la GPU son tratadas como dispositivos separados que tienen su propio espacio en memoria (de manera semejante a los sistemas de memoria distribuida). Esta configuración permite la computación simultánea en la CPU y GPU sin una contención de recursos en memoria [8].
- Los hilos ejecutan partes de un programa que están gestionadas por tareas paralelas (núcleos o kernels, secuencias de trabajos hechas por cada hilo) mapeadas en un dominio (hilos que están preparados para su ejecución).

2.3.1. Ventajas

- Memoria Compartida: CUDA dispone de mecanismos de sincronización de procesos usando el área de memoria de 16KB compartida entre los threads. Su tamaño y rapidez puede ser utilizada como cache.
- Lecturas más rápidas entre la CPU y la GPU.
- Soporte de enteros y operadores a nivel de bit.

2.3.2. Limitaciones

- No se puede utilizar recursividad, punteros a funciones, variables estáticas dentro de funciones con número de parámetros variables.
- En precisión simple no soporta números indefinidos como NaNs (No definidos).
- Puede existir cuellos de botella entre la CPU y GPU debido a los anchos de bandas y a los buses.

2.3.3. Arquitectura

Consta de varios componentes:

- Motores de computación paralela dentro de la GPU de NVIDIA [18].
- Nivel del Kernel OS para el soporte de la inicialización del hardware, configuración, etc. [18].
- Driver de modo usuario, lo que provee un API al nivel del dispositivo para los programadores [18].
- Arquitectura del Conjunto de Instrucciones PTX (ISA) para funciones y núcleos de computación paralela [18].

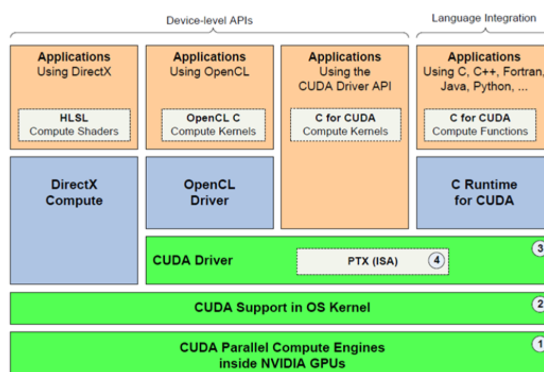


Figura 2.9: Vista General de la Arquitectura CUDA. Tomado de 18.

2.3.4. Ambiente de desarrollo en CUDA

La página de Nvidia (http://www.nvidia.es/page/home.html) provee las herramientas, ejemplos y documentación para el desarrollo de las aplicaciones como se muestra a continuación:

Librerías Se utilizan las que incluyen BLAS, FFT y otras funciones especiales para la optimización de la arquitectura CUDA [18].

C Runtime El C Runtime para CUDA provee la ejecución de funciones standard en la GPU [18].

Herramientas El compilador NVIDIA C (nvcc), debugger de CUDA (cudagdb) y otras herramientas que enriquecen el entorno de programación [18].

Documentación La guía de programación, las APIs y otras ayudas útiles para la implementación [18].

Ejemplos: el SDK incluye los códigos ejemplos y documentación de una amplia variedad de algoritmos y aplicaciones implementadas en la GPU.

2.3.5. El modelo cuda

CUDA puede aprovechar el paralelismo y el alto ancho de banda de la GPU al máximo en aplicaciones prácticas de gran cálculo numérico y de abundantes accesos a memoria, lo que aumenta el rendimiento del sistema. Es usado en una gran variedad de sistemas informáticos para incrementar la velocidad de procesamiento de los cálculos de los algoritmos implementados en la GPU [18].

El modelo CUDA está diseñado para el procesamiento en cómputo de las aplicaciones que escalen aumentando la potencia en el número de núcleos computacionales.
Usando la interfaz de programación del nivel del dispositivo se puede codificar archivos separados para procesar los cálculos usando el lenguaje de los núcleos (kernels) soportado por la API de selección. Por ejemplo los Kernels OpenCL están escritos en un lenguaje muy similar a C llamado OpenCL C.
Usando la interfaz de programación para la integración del lenguaje, los programadores pueden escribir funciones C y de C Runtime para CUDA lo que automáticamente permite ejecutar las funciones

computadas en la GPU. La interfaz de programación permite la integración de diferentes lenguajes de soporte nativo a la aplicación como C, C++, Fortran, Java, etc. Con el tipo de integración se permite que tipos de datos como vectores y strutcs sean usados de manera similar en funciones que se ejecutan tanto en la CPU como en la GPU. La integración del código permite que una misma función sea llamada de la CPU o de la GPU.
La estructura que utiliza está definida por un grid, dentro del cual hay bloques de hilos distintos formados por un máximo de 32 de ellos.
Cada hilo esta precisado por un identificador único que se denomina threadIdx. Esta variable se usa para repartir el trabajo entre distintos hilos.
Cada threadIdx puede tener dos componentes (x, y) o tres componentes(x, y, z), dependiendo de las dimensiones de los hilos.
Al igual los que hilos, los bloques se pueden identificar por medio de un blockIdx.x para x y blockIdx.y para y. Para acceder al tamaño de bloque se usa blockDim, también se puede usar para x e y [7].

2.3.6. Kernel

Un Kernel es una función que se define como _global_ mediante el cual CUDA hará el trabajo en N threads al ejecutarse las instrucciones en forma paralela. Su estructura es la siguiente en la aplicación:

```
_global_  void f(int a, int b, int c)
{
}
```

2.3.7. Invocaciones a un Núcleo (Kernel)

Para invocar el kernel, se le ha de pasar el tamaño del grid y el bloque, en el main del programa principal en CUDA se le agrega el siguiente código:

```
dim3 block(N, N); //Definimos un bloque de hilos de NxN
dim3 grid(M, M)    //Grid de tama o MxM

f<<<grid, block>>>(A, B, C);
```

EEn el momento que se invoque esta función, los bloques de un grid se enumeraran y distribuirán por diferentes procesadores libres.

2.3.8. Sincronización

Como los hilos comparten datos y trabajan simultáneamente requieren de directivas de sincronización. En un kernel, se puede aplicar una barrera incluyendo una llamada a __syncthreads(), en la que todos los hilos esperan a que los demás lleguen a este mismo punto.

2.4. Arquitectura GPU

Los científicos se han interesado por esta tecnología debido a su bajo costo y potencia. A diferencia de los multiprocesadores convencionales, los núcleos de los procesadores GPU's son especiales para procesar gráficos y proveen excelentes aceleraciones en gran variedad de aplicaciones.
Cada GPU es un conjunto de núcleos de procesamiento con la capacidad de dirigir directamente una memoria global mediante el uso del cómputo paralelo, lo que hace posible la implementación de aplicaciones elaboradas con programación paralela que se ejecuten utilizando hasta cientos de hilos [26].

Cada núcleo comparte recursos, incluyendo registros de memoria los cual permite que varias tareas ejecutadas en estos núcleos compartan datos sin enviarlos a la memoria principal. [8]

Los diferentes modelos de GPU's tienen su propia configuración y arquitectura de núcleos para paralelizar una tarea.

Hay una parte la secuencial que es la que se ejecuta en la CPU y las partes de mayor carga computacional se ejecutan en la GPU lo que multiplica el rendimiento en el sistema.

El siguiente gráfico hace una comparación entre una CPU actual (de 4 núcleos) y una GPU (de 240) [15].

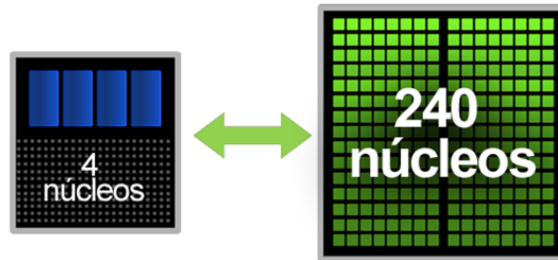


Figura 2.10: Comparación GPU - CPU Tomado de: [15].

Se asigna los núcleos (Kernerls) a la GPU para la parte que se va a correr en paralelo y el resto del programa se ejecuta en forma secuencial en la CPU. Al Escribir un programa en la GPU se aprovecha el rendimiento en términos de paralelismo en la tarjeta gráfica. La computación GPU implementa una arquitectura paralela incluida en las GPUs NVIDIA que se denomina CUDA. Las GPUs Tesla Serie 10 se basan en la arquitectura CUDA de segunda generación que contiene funciones optimizadas para cálculos científicos en la GPU (punto flotante o doble precisión), memoria compartida o acceso coalescente a memoria [15].

2.4.1. Microarquitectura de Soporte para la Ejecución Paralela [26]

La GPU es un tipo de procesador especializado para gráficos. La siguiente figura muestra la microarquitectura de la Geforce 8800 (GPU Nvidia) y describe los distintos accesos de memoria:

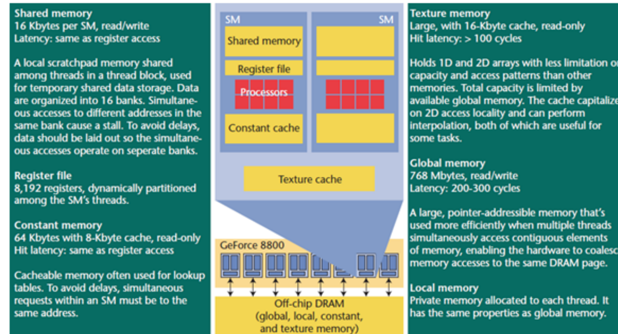


Figura 2.11: Características del procesador y memoria de la Geforce 8800. El código ejecuta en el procesador puede ser accesible desde cualquier espacio de memoria. Las memorias mas cercas al procesador (columna izquierda) son más pequeña y rápidas que las más alejadas (columna derecha). Tomado de: [26].

Esta Arquitectura consiste en 16 flujos de multiprocesadores (SMs), cada uno contiene 8 flujos de procesadores (SPs), o procesadores núcleos, que corren a 1.35 HZ. Los núcleos en un SM ejecutan instrucciones en SIMD (Single Instruction Multiple Data), con la unidad de instrucción SM transmite la instrucción actual a los núcleos. Cada núcleo tiene una precisión simple de 32 bits. Además cada SM tiene dos unidades super funcionales que ejecutan las operaciones más complejas de punto flotante, como las operaciones trigonométricas, con un alto rendimiento. La Nvidia Tesla GPUs soporta precisión doble de punto flotante. La unidad de la Geforce 8800 de ejecución SIMD es una urdimbre de 32 hilos en paralelo. Cada bloque se forma a partir de grupos contiguos de hilos, los 32 primeros hilos forman el primer urdimbre, los 32 siguientes el segundo urdimbre y así sucesivamente. CUDA no declara explícitamente las urdimbres de los hilos. Cuando dichos hilos dentro la urdimbre se ejecutan a diferentes flujos de control, se conoce como rama de divergencia, porque el hardware ejecuta múltiples pasos a través del código del programa con la supresión de trayectos. La ejecución es lenta dependiendo de que si cada hilo ejecuta todos los flujos de control del sistema. Esto hace que los núcleos con una gran carga de número de flujos de control produzcan efectos inadecuados en los datos durante el procesamiento en la GPU.
La GPU produce la ejecución SIMD de hilos en el trayecto, lo que permite a los programadores salvar el esfuerzo manual de reestructuración del flujo de control y datos en el modelo SIMD. Esto se traduce en un buen desempeño para las gráficas shader y núcleos de datos paralelos, lo cual los hilos de la aplicación ejecutan la misma secuencia de instrucciones. En las arquitecturas de varios núcleos, como Larrabee, se debe transmitir el flujo de control dentro de la unidad de ejecución SIMD. Por otro lado, el sistema de ejecución AMD/ATI se compila los shaders o la computación de núcleos para predecir el código SIMD para el flujo AMD [26].

La libertad en la programación es alta en muchas aplicaciones debido a que los hilos en diferentes urdimbres son independientes, con la excepción de la barrera explicita en la sincronización entre los hilos del mismo bloque. Esto hace que la GPU se mantenga altamente paralela a la ejecución usando una pequeña porción del área del chip. Cada SM soporta un máximo de 768 contextos de hilos activos.

El código CUDA en ejecución asigna un número integral de ocho hilos en un SM en cualquier tiempo para completar los contextos del hilo. Cuando se asignan los bloques a la SM, CUDA automáticamente reserva espacio en memoria para los recursos de hardware tales como los contextos del hilo, memoria compartida y registros. En la optimización se debe tener en cuenta el límite de hilos paralelos que pueden correr en el dispositivo. Algunas veces esto tiene efectos negativos debido a pequeños incrementos en el uso de los recursos lo que puede suceder que menos bloques y menos hilos se ejecuten simultáneamente.

2.4.2. Espacios de Memoria [26]

Los datos pueden ser desplazados en la memoria global, compartida, local, constante o de textura. Las memorias GPU's son especializadas; tienen diferentes accesos durante el tiempo de ejecución. Algunas memorias son de rápido acceso solo para algunos patrones límites de referencias de memoria. Para el alto desempeño en los cálculos paralelizados, se debe tener en cuenta la estructura de la memoria de la tarjeta para disponer los datos.

La memoria global es grande en tamaño de latencia pero la memoria que existe físicamente fuera del chip dinámico RAM es la (DRAM), que la memoria principal en un chip multiprocesador. Se debe escribir desde la salida del núcleo (kernel) a la memoria global que sea de acceso de lectura después de que el núcleo termina la ejecución.
La memoria constante sirve como de acceso simultaneo a varios hilos como de lectura, el valor de la cache es transmitido a todos los hilos de la urdimbre. Lo cual hay 32 cargas de memoria con un simple acceso a la cache. Al cargar todas las instancias de las instrucciones al mismo acceso de la memoria el valor puede ser transmitido a todos los hilos. A continuación se describe el esquema de la memoria constante:

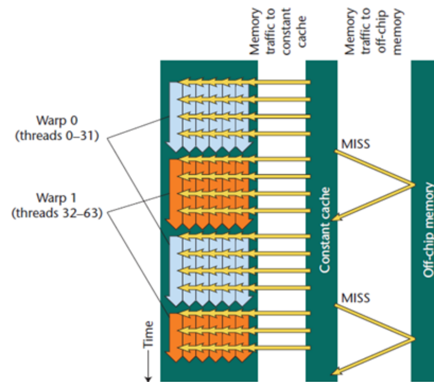


Figura 2.12: La transmisión del valor entre todos los hilos se muestra por las flechas horizontales cruzando los hilos en la urdimbre. Tomado de: [26].

2.4.3. Optimización de la Aplicación [26]

La computación intensiva de los núcleos con pocos accesos a la memoria global mejora el rendimiento de la aplicación. Las Geforce 8800 tienen la habilidad de ejecutar gran cantidad de hilos simultáneamente, esto se debe a la correcta distribución de los recursos de memoria e hilos que generalmente se establece en la fase inicial de optimización. La granularidad del hilo puede ser tan pequeña que

toma la ventaja del reuso de los datos o demasiado grande para encajar simultáneamente varios hilos en el hardware. Este modelo de programación es comúnmente utilizado para distribuir gran carga de trabajo entre los hilos aprovechando el uso de la memoria a través de la estrategia de intercambio de lazo (cambio de la jerarquía de lazos), los núcleos MRI son un ejemplo de esto.

2.5. Estructuras de datos espaciales

Una estructura de datos espacial es una que se utiliza para organizar los elementos que se encuentran en un espacio de dimensiones. Estas estructuras pueden ser usadas para acelerar la respuesta a queries o consultas acerca de la colisión (o intersección) de diferentes geometrías, es decir, entre diferentes objetos.

La construcción de las estructuras de datos espaciales generalmente se hace de manera jerárquica, esto quiere decir, que el nivel con más jerarquía encierra o contiene el nivel inmediatamente inferior, y a su vez este contiene el nivel inmediatamente inferior a este y así sucesivamente, de esta forma, se tiene una estructura anidada y recursiva. La razón principal para utilizar una jerarquía es que el tiempo de respuesta a diferentes tipos de consultas se vuelve significativamente corto; generalmente se obtiene una mejoría de $O(n)$ a $O(logn)$ [29].

Por otro lado es de notar que la construcción de dichas estructuras de datos es un proceso costoso que generalmente se realiza antes de hacer alguna consulta sobre ellas y no al momento mismo de realizarlas, es decir, se realiza como un preproceso.

Existen diferentes tipos de estructuras que son: jerarquías con volúmenes envolventes (BVH), arboles BSP (Binary space partitioning) y de árboles octales (Octrees). Por un lado, los BSP y los árboles octales están basados en la subdivisión del espacio, es decir, el espacio completo de la escena es subdividido y codificado para poder manejarlo más adecuadamente mediante la estructura. Por otro lado, los BVH, en lugar de dividir el espacio completo, se concentran en dividir y codificar únicamente objetos geométricos para poder manipularlos más adecuadamente. En este trabajo se usa una implementación propia de las BVH que se explicará con detalle más adelante.

2.5.1. Bounding Volumes Hierarchies

Un volumen envolvente (BV) es un volumen que envuelve un conjunto de objetos, y su principal característica es que es una forma geométrica más simple de lo que son las figuras que envuelven; de esta forma, los queries realizados sobre dichos BV son más eficientes que los que se harían sobre las figuras originales.

Cabe notar que los BV por si solos no contribuyen a mejorar la calidad de la imagen como tal, en su lugar, contribuyen en el proceso de renderizado al optimizar y acelerar los diferentes cálculos y queries que este requiere [29].

El objeto es organizado en una estructura jerárquica (árbol) que está organizado de la siguiente manera: el nodo con más jerarquía es la raíz, es un BV que encierra a toda la figura y no tiene padres. Por el contrario, los nodos con menos jerarquía son las hojas y son BV que encierran la unidad más pequeña posible de la "representación" de la figura (polígonos de los que se compone la malla que se utiliza para dar forma al objeto en el espacio tridimensional) tienen un padre y no tienen hijos. Por último, los nodos intermedios tienen un padre e hijos. Cada nodo es, como se dijo, un BV que envuelve una porción de un objeto.

A continuación se muestra un ejemplo de una jerarquía de volúmenes envolventes:

En la parte izquierda de la figura se muestra una escena en la que cada objeto es encerrado en una esfera. Cada grupo de esferas está agrupado dentro de otra esfera mayor. En la parte derecha se muestra la jerarquía formada por dicho grupo de esferas.

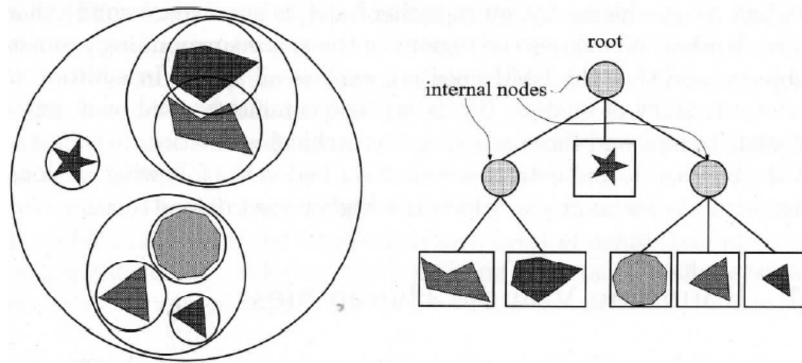


Figura 2.13: Ejemplo BVH. Tomado de [29]

2.5.2. Árboles

Los árboles son estructuras de datos jerárquicas ampliamente usados en computación gráfica debido a la optimización que proporciona en una gran cantidad de procesos. Dicha optimización se debe precisamente a su naturaleza que permite descartar en un solo paso una gran cantidad de cálculos, pues, por ejemplo, si se está determinando la colisión de un objeto con una figura representada por una BVH binario, se podría descartar el análisis de la mitad de la figura solo en el primer paso si se utiliza la condición adecuada.

La altura de un árbol se refiere a la cantidad de jerarquías que se forman; por ejemplo, los hijos del nodo raíz están a una altura 1, los hijos de estos a una altura 2 y así sucesivamente.

Los árboles balanceados son aquellos en los que todas las hojas están a una altura h o por lo menos a una altura $h-1$. Un árbol completo es aquel en el que todas sus hojas están a la misma altura [29].
El número de nodos de un árbol puede ser determinado mediante:

$$n = k^0 + k^1 + \ldots + k^h = \frac{k^{h+1} - 1}{k - 1}$$

De la misma forma el número de hojas l es $l = k^h$ y el número de nodos internos i es igual a $i = n - l = \dfrac{k^h - 1}{k - 1}$ En el caso de árboles binarios, donde $k = 2$, $n = 2l - 1$.
Los árboles binarios generalmente son la mejor opción dado su buen funcionamiento [29].

2.5.3. Árboles binarios

Un árbol BSP (Binary space partitioning) es una estructura de datos que permite dividir el espacio en partes más pequeñas. El beneficio de esto es que cuando se está trabajando con este tipo de elementos, se deben considerar una menor cantidad de datos a la vez [30].

La importancia de este tipo de árboles radica en que proporcionan un balance adecuado entre elementos generalmente presentes en una estructura de datos útil en computación gráfica, y que generalmente están unos en contra de otros, como son una alta capacidad de almacenamiento contra un tiempo de acceso corto en procesamiento secuencial y aleatorio, y facilidad de modificación contra

usabilidad. Aunque no es óptima en ninguno de estos aspectos, es lo suficientemente bueno en todos ellos a la vez [32].

Tradicionalmente los árboles binarios se han usado y explotado e sistemas gráficos, en juegos (dada su eficiencia para resolver la visibilidad), la generación de sombras y la detección de colisiones [31]. Aquí se usan para optimizar los cálculos de colisión y deformación de objetos.

2.5.4. Cálculo de árbol binario

En esta aplicación se utilizan los árboles binarios para almacenar o cargar los objetos con el fin de optimizar, primero, el cálculo correspondiente a la detección de colisiones con otros objetos, y segundo, para mejorar el rendimiento del cálculo de la deformación.

Antes de exponer el algoritmo empleado para la creación del árbol, es necesario tener en cuenta que los objetos dentro de la aplicación se representan con mallas formadas por triángulos como se muestra en la siguiente imagen:

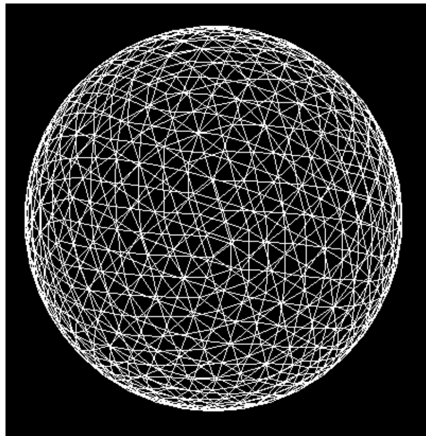
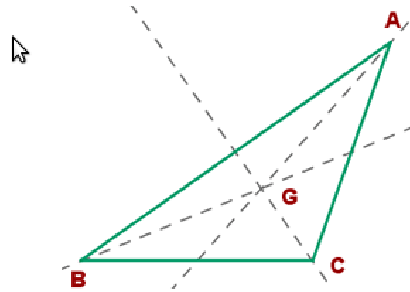


Figura 2.14: Ejemplo BVH

Como parte de un preproceso, se calculan los baricentros de todos los triángulos de dicha malla y se almacenan en un vector. Tales baricentros se ubican en el espacio mediante coordenadas x, y y z. Esto se hace con el objetivo de representar cada uno de esos triángulos con un único punto, para lo cual se usa un mapa que indica que baricentro corresponde a que triángulo, lo que permite finalmente optimizar tanto el proceso de creación del árbol de la figura, como los cálculos relacionados a la colisión y a la deformación.

Coordenadas del baricentro de un triángulo en el espacio

Sean A (x_1, y_1, z_1), B (x_2, y_2, z_2) y C (x_3, y_3, z_3) los vértices de un triángulo, las **coordenadas del baricentro** son:



$$G\left(\frac{x_1+x_2+x_3}{3}, \frac{y_1+y_2+y_3}{3}, \frac{z_1+z_2+z_3}{3}\right)$$

Figura 2.15: Baricentro de un triángulo

En este punto es importante hacer notar que el uso de un mapa para almacenar la correspondencia triangulo - baricentro tiene como objetivo reducir el tiempo de acceso a los triángulos, pues esta estructura permite un acceso directo a cada uno. Por otro lado, teniendo en cuenta que para la creación del árbol se deben procesar los baricentros uno por uno, se utilizó un vector para acceder secuencialmente a ellos.

Finalmente, el algoritmo es el siguiente:

```
crearArbol(vector baricentros, Nodo n){

        si (tama o de baricentros es > 1) entonces
        {
                vector p;
                vector q;

                dividirVectorBaricentros(n.ObtenerP,n.ObtenerQ,
                  baricentros,p,q);


                Nodo izq = nuevo Nodo a partir de p;
                Nodo der = nuevo Nodo a partir de q;

                Agregar Nodo izq a n;
                Agregar Nodo der a n;

                crearArbol (bari1,izq);
                crearArbol (bari2,der);
        }
}
```

El tipo de dato Nodo se usa para representar los nodos que conformarán el árbol. Como se mostró anteriormente, para una BVH (Bounding Volume Hierarchy) cada nodo estaría constituido por una BV

(Bounding Volume), es decir, una figura geométrica que envuelve una parte de la figura. Aquí por el contrario, un nodo es un conjunto de baricentros-triángulos que hacen parte de la figura.
Comparando un método con el otro vemos que la única diferencia consiste en que el algoritmo empleado en esta aplicación no utiliza volúmenes envolventes para encerrar partes de la figura, sino que toma directamente trozos de la misma. Por lo demás, funcionan de la misma manera.

Teniendo esto en cuenta, vemos que el método crearArbol recibe un vector de baricentros y un nodo n. La primera vez que se llama esta función, el vector de baricentros contiene los baricentros de todos los triángulos de la figura, y el nodo n está formado a su vez por todos los baricentros de la figura, es decir, es el nodo raíz.
Cada nodo tiene asociado un par de puntos P,Q que corresponden al par de puntos más lejanos entre sí dentro de los que conforman el nodo en cuestión; estos se utilizan para dividir el nodo en dos partes, guardando en dos vectores los puntos más cercanos a P o a Q, es decir, para cada punto que hace parte del nodo, se calcula su distancia a P y a Q (que tienen cada uno un vector asociado), y se mente en el vector del punto más cercano a él.

Con los vectores resultantes se crean dos nuevos nodos, se asocian al nodo n (como sus hijos) y se llama recursivamente el método crearArbol con cada uno de ellos. El método para cuando el vector de baricentros que recibe es igual a uno, es decir, cuando ha llegado a una hoja.

El árbol resultante es algo como esto:

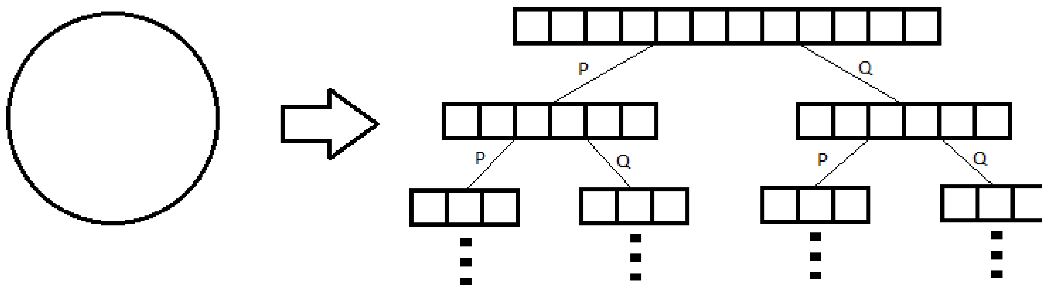


Figura 2.16: Construcción de árbol. El nodo raíz contiene todos los baricentros de la figura. Las hojas solo contienen un baricentro.

2.6. Técnicas usadas en la detección de colisiones

La detección de colisiones es una parte fundamental en muchas aplicaciones gráficas y en áreas como la realidad virtual. Algunas de las áreas en las que la detección de colisiones juega un papel fundamental son la manufactura virtual, aplicaciones CAD/CAM, animación por computador, juegos, simuladores, robótica y realidad virtual, entre otras.

La detección de colisiones hace parte de lo que se llama manejo de colisiones, que puede ser dividido en tres partes fundamentalmente: detección de colisiones, determinación de colisiones y respuesta a colisiones. El resultado de la primera es un booleano indicando cuando dos o más objetos colisionan, el de la segunda es la parte de los objetos que está colisionando y el de la última es la respuesta que se va a dar a dicha colisión [29]. En esta aplicación se tendrán en cuenta todas ellas y se explicarán con mayor detalle más adelante.

Actualmente existen diversos métodos que se emplean para detectar colisiones en aplicaciones gráficas, y cada uno difiere de los demás en cuanto a precisión, complejidad y rendimiento, pero en cualquiera de los casos se espera que tengan por lo menos las siguientes características:

- Que proporcionen diferentes grados de interactividad entre modelos de objetos que consistan en una gran cantidad de polígonos, tanto si ambos modelos están cerca como si están lejos uno del otro.
- Que manejen modelos de objetos basados en polígonos, sin importar la configuración de los mismos o si se posee la adyacencia de dichos polígonos.
- Que se puedan aplicar para diferentes tipos de movimientos de los objetos.
- Que proporcionen volúmenes envolventes ajustados a la forma de cada objeto.

Un escenario puede contener cientos de objetos moviéndose, por lo que un buen algoritmo de detección de colisiones debe tener esto en cuenta.
En una situación con n objetos moviéndose y m objetos estáticos, un algoritmo sencillo ejecutaría $nm + \frac{n}{2}$ cálculos de colisión por cada frame [29]. El primer término corresponde a las colisiones entre objetos estáticos (m) y dinámicos (n) y el segundo a las colisiones entre objetos dinámicos solamente. Se puede ver entonces que a medida que n y m crecen un método sencillo se vuelve bastante costoso por lo que se hacen necesarios métodos más "inteligentes", sin embargo, es necesario tener en cuenta que la evaluación del rendimiento de un método de detección de colisiones es bastante difícil, ya que los diferentes algoritmos son sensibles al estado actual del escenario, es decir, su rendimiento es diferente en diferentes escenarios, y no existe un método que trabaje mejor en todos los casos o escenarios.
A continuación se presentan algunos algoritmos que comúnmente se usan en la detección de colisiones.

2.6.1. Detección de colisiones con rayos

Este es un método sencillo que consiste en dirigir rayos, que salen de la superficie de la figura, hacia los demás objetos de la escena. Luego cada rayo se analiza en busca de intersección con otros elementos de la escena, de tal manera que cuando el rayo se intercepta con otro elemento del ambiente, se aumenta la probabilidad de que la figura colisione en esa dirección. Finalmente, si la distancia entre el origen del rayo y cualquier elemento del ambiente es cero, es porque se ha producido una colisión.

Por ejemplo, la interacción/colisión de un carro y el piso se puede modelar de esta manera.

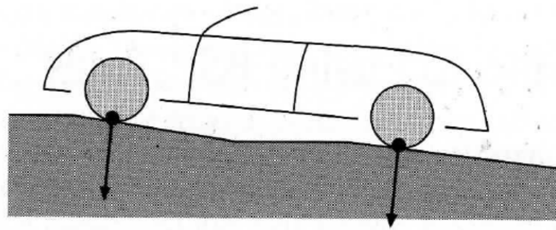


Figura 2.17: Método de detección de colisiones con rayos. La colisión se produce cuando distancia entre el origen del rayo y el medio (carretera) es cero. Tomado de [29]

2.6.2. Detección de colisiones usando estructuras jerárquicas

La mayoría de los algoritmos más eficientes para la detección de colisiones hacen uso de la descomposición jerárquica de escenas complejas, pues permite evitar la evaluación extensiva de todas las posibles parejas de primitivas con que están construidos los objetos.

Existen dos formas de construir tales jerarquías, la primera es usando esquemas de subdivisión del espacio, donde el espacio es dividido en una estructura jerárquica, como los árboles BSP. Esta forma permite encontrar un número reducido de vecinos geográficos en un tiempo $log(n)$ (donde n es el número de objetos) y así verificar colisiones contra dicho objeto.
La segunda forma es usando esquemas de subdivisión de los objetos, donde las primitivas de los mismos son agrupadas jerárquicamente en una estructura, como se hace en los BVH. De esta forma, grandes partes de una figura pueden ser descartadas en un tiempo $log(n)$ (donde n es el número de primitivas del objeto) durante la evaluación de colisión [32].

La ventaja de usar la segunda aproximación consiste en que, primero, las mallas de los objetos definen una estructura jerárquica que en la mayoría de los casos no necesita ser actualizada (siempre y cuando no hallan colisiones) y segundo, que la topología de la jerarquía refleja la adyacencia de las regiones descritas en ella. Tal adyacencia puede ser usada en varios tipos de optimización.

2.7. Técnica usada para la detección de colisiones

La técnica usada en este trabajo para la detección de colisiones es una variación de la aproximación por BVH, pues como se mencionó anteriormente, no se utilizan BV para representar partes de la figura sino que en su lugar se toman partes de la mima, es decir, se toman conjuntos de primitivas adyacentes para formar los nodos de la estructura jerárquica.

Se mencionó también que cada nodo tiene asociados un par de primitivas P y Q que son las más lejanas entre sí de las que lo componen.

Pues bien, en primer lugar se comparan los P y los Q de los nodos raíz de los objetos con el fin de determinar cuáles son los más cercanos, como se muestra en la figura:

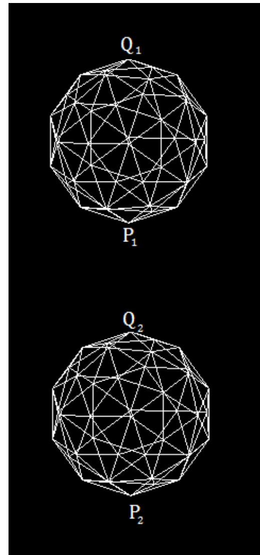


Figura 2.18: Distancias P Q

En este caso, se puede ver que las primitivas más cercanas son P1 y Q2 por lo que lo más probable es que ambos objetos colisionen por ese lado.
Con esto ya determinado, se obtienen los nodos hijos asociados a P1 y Q2 y se repite el proceso hasta llegar a las hojas:

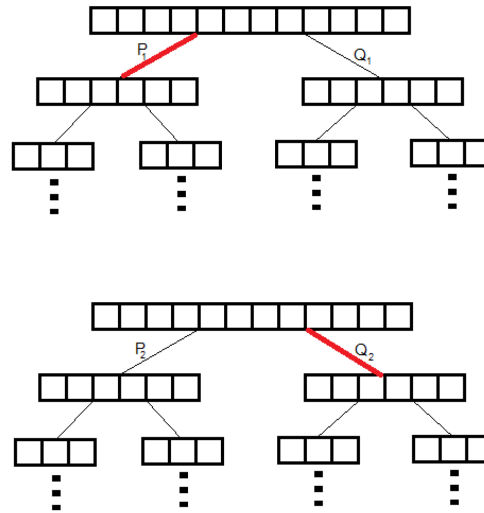


Figura 2.19: Búsqueda de colisión.

Cuando se llega a las hojas en ambas jerarquías se verifica si se están interceptando o no para determinar si las figuras están colisionando.

Este algoritmo tiene una complejidad de $O(logn)$ gracias a la optimización que se logra usando una estructura jerárquica (como que se mencionó anteriormente).

2.8. Modelo de deformación (mass-spring model)

Existen varios modelos físicos para el cálculo de la deformación de un cuerpo en tercera dimensión, como el método de elementos largos, mínimos cuadrados, entre otros, pero la base de este proyecto es el modelo Mass-Spring que se basa en la ley de los resortes para medir la fuerza aplicada a una masa. Este modelo se aplica principalmente para objetos tridimensionales deformables que se definen aquí como varios puntos con masa, conectados por enlaces elásticos. En esta implementación, cada nodo de la malla ejerce una fuerza resultante en el lugar que se produce la colisión.
Mass-Spring ha sido implementado en diversos campos como aplicaciones quirúrgicas, animaciones faciales y objetos deformables.

A diferencia de FEM, esta técnica de ingeniería y física permite resolver cálculos por medio de una aproximación numérica usando ecuaciones diferenciales parciales en un cuerpo representado por una malla de puntos (llamados nodos), lo cual usa una malla descompuesta en un dominio sobre el cual las ecuaciones diferenciales de movimiento son resueltas y se computa un vector que representa el desplazamiento de cada punto en el dominio, pero puede resultar más preciso que el modelo de Mass-Spring. Asumiendo que en la malla pueden interactuar fuerzas externas que muchas veces, dependiendo de la situación, son diferentes a la fuerza de la gravedad; estas fuerzas permiten la deformación del objeto.

El modelo representa un objeto deformable mediante una malla tridimensional M donde los puntos pt, $i = 0, 1, 2, \ldots N$ con masa $m1$ están conectados por enlaces $L_{ij} i, j \in [1, n], i \neq j$.

Cada punto contiene la relación de sus vecinos y cada partícula se desplaza debido a la fuerza inducida por sus resortes en el modelo físico. Los nodos y los enlaces de los objetos están triangulados, es decir, forman triángulos. Cada punto de conexión de la malla tiene vecinos y con relación a ellos se calcula tanto la fuerza resultante como el área deformable. A continuación se muestra una malla triangular de un objeto tridimensional. [9]

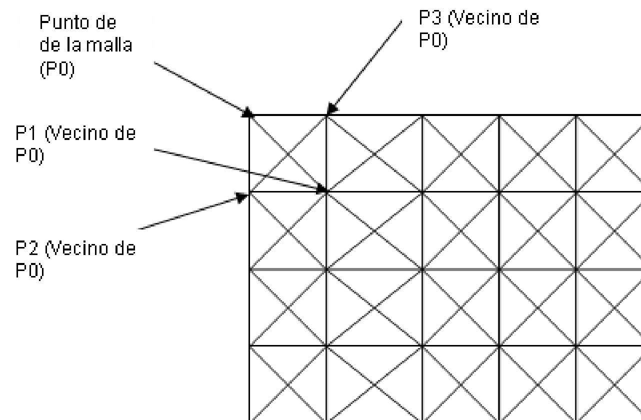


Figura 2.20: Malla triangular que muestra un punto con sus respectivos vecinos, esto aplica a todos los puntos de la malla (Área de deformación).

La siguiente figura muestra el área de deformación de una esfera:

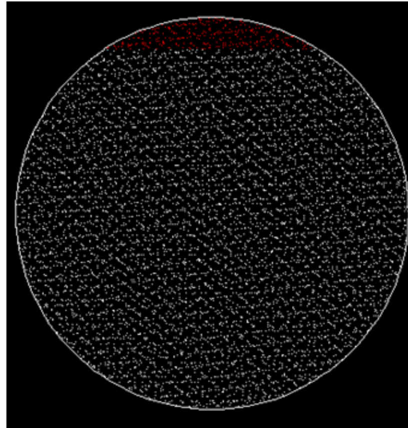


Figura 2.21: Objeto representado en puntos que muestra su área deformable (los puntos en rojo)

El algoritmo de respuesta a la colisión determina la fuerza externa en los puntos de la región de deformación de la figura.

Cada punto de la malla conectado con respecto a sus vecinos es un resorte como se muestra el siguiente diagrama de resortes para un punto en particular.

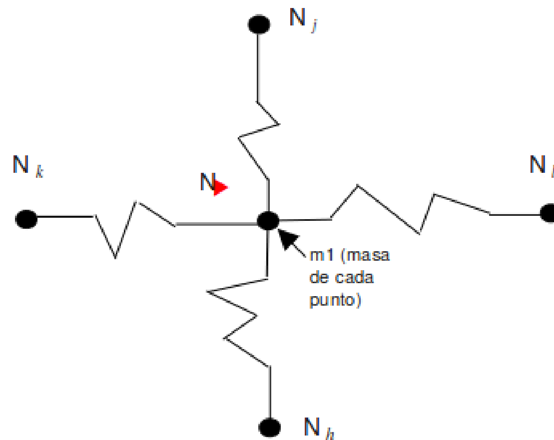


Figura 2.22: Esquema de resortes de un punto de la figura con masa m

En las propiedades mecánicas, como la visco-elasticidad en la mayoría de aplicaciones quirúrgicas, los objetos almacenados están representados por los nodos y enlaces en la malla M.
$M1$ representa la masa de cada uno de los puntos y se obtiene dividiendo la masa total por el número de vértices de la figura; C_i es el coeficiente de viscosidad y k_{ij} que esta asociado al enlace L entre i y j.
La fuerza interna $\vec{F}_{ij} = -k\triangle_{ij}\vec{u}_{ij}$ es un vector entre N_i y N_j, donde $\triangle_{ij} = L_{ij} - \beta L_{ij}$, L_{ij} es la distancia entre N_i y N_j(el enlace entre i y j), β es un parámetro de incremento con respecto a L_{ij},

$\triangle_{ij}$ es la longitud actual del enlace menos el enlace asociado a la distancia de sus vecinos y $\vec{u}_{ij}$ es el vector unitario entre N_i y N_j. En algún instante de tiempo el movimiento/deformación de la malla M esta descrito por un sistema de n ecuaciones diferenciales que expresa el cambio en la coordenada de cada nodo N_i en el objeto deformable[9].

$$m_i + a_i + c_i v_i + \sum_{j \in \sigma(i)} F_{ij}(x_i, y_j) = m_i g + F_i^{ext}$$

m_i masa de cada punto
a_i es la velocidad y los vectores de aceleración
$m_i g$ es la fuerza gravitacional
F_i^{ext} es la fuerza externa aplicada a N_i
$\sigma(i)$ denota los nodos vecinos conectados a N_i en la malla.

La fuerza externa (F_i^{ext}) se aplica con un parámetro de convergencia que es otra fuerza que actúa independientemente en el objeto.
Esta ecuación diferencial se aplica a sistemas quirúrgicos que se puede resolver utilizando cualquier método como Newton o Euler-Chauchy.

2.9. Método de deformación

El modelo físico de deformación rodea el área de los puntos al que se le aplica la fuerza externa total calculada a partir del objeto que produce la colisión dada por la siguiente fórmula: $F_i^{extTotal} = ma$ donde m es la masa y a es la aceleración con la que choca el objeto.
La complejidad que se tiene del modelo es O(n*e) donde n es el numero de vértices y e el numero de bordes de la figura, que es mucho más lento que el algoritmo de colisiones pues el de deformación se define a partir del producto de los puntos por el lado de las aristas de la figura que participan directamente en la colisión y determinan la nueva configuración del objeto.
En esta aplicación se simulará un objeto en caída libre que, al chocar con un segundo objeto debajo de él, rebota con la misma velocidad y en el sentido contrario a la dirección inicial. La energía del sistema se conserva. En la siguiente figura se ve el trabajo realizado por una fuerza conservativa en el cambio del movimiento del objeto.

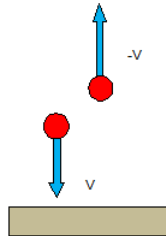


Figura 2.23: Momento a la respuesta de la colisión del objeto en caída al instante de chocar

2.9.1. Pseudocódigo de deformación

N_i: cada nodo (vértice) de la figura en las componentes x, y, z.
V_j: cada vértice vecino del nodo N_i.
t_i: fuerza acumulada que interactúa con el punto N_i y los vecinos de N_i.
β: parámetro de convergencia que se aplica para la deformación del figura.
$\vec{u}_{ij}$: vector unitario entre el nodo N_i y el vecino V_j.
F_i^{ext}: fuerza externa que actúa en el nodo N_i.
$Fuerza_i$: fuerza resultante que actúa en el nodo N_i.

Método Calcular deformación:

```
Para cada Nodo Ni ... Nn de la figura

     Si n mero Vecinos de Ni > 4

             Para cada  Vecino Vj...Vm de Ni

                    //para cada punto en x, y, z
                    Calcular la Distancia entre Ni y Vj

        t   t+    uij

      Fuerza_i     ti

     {Calcular la fuerza para el nodo Ni en x, y, z}

      Ni      Ni + Fuerza_i +    F_i en el V rtice
```

2.9.2. Calculo de la Deformación de la Figura a partir del Modelo Físico

Cada Nodo N_i tiene un conjunto de nodos vecinos V_j, inicialmente se calcula los vecinos del vértice i para saber en qué puntos actúan las fuerzas externas al detectar la colisión entre las figuras a partir de la configuración actual de los resortes de atracción entre los puntos; la fuerza externa cambia dependiendo del área de la figura deformable que entra en contacto a medida que esta se deforma. La fuerza que actúa en el punto se calcula en función del resorte que es el que determina la fuerza resultante dada por la acumulación de las fuerzas con respecto a los vecinos. La elongación de las aristas está determinada por el coeficiente de elasticidad que se determinó previamente (la cual depende del material del objeto) y la configuración de los puntos vecinos luego de un tiempo t de deformación de la figura. Luego de la colisión, el objeto deformable recupera su forma inicial ya que no se supera su límite estático. En este punto la fuerza externa que actúa sobre todos los puntos es la misma y es igual a 0.

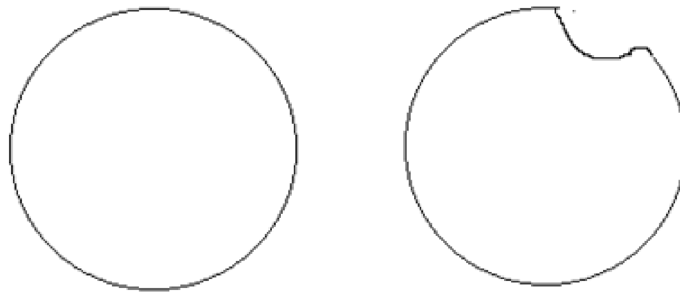


Figura 2.24: Estado inicial y final del cuerpo tridimensional al aplicar el modelo físico

Capítulo 3

Estado del arte

Aquí se muestran una serie de trabajos que se han realizado en diferentes lugares en las áreas de detección de colisiones y del modelo físico de deformación. Se hablará un poco al respecto de los mismos.

Deformaciones Interactivas en la GPU [22]

En dicho trabajo, el método que se utilizó fue, en primer lugar, implementado tanto en la CPU como en la GPU para comparar el rendimiento de este en ambos procesadores. Con la GPU se obtuvo un rendimiento de 100 veces mayor, usando el método "cuadrado móviles" para la deformación de las figuras. En segundo lugar, fue utilizado para realizar varios experimentos con el fin de medir la eficiencia en la CPU y GPU. Se comprobó que la GPU tiene una capacidad de computo mucho mayor que la CPU en términos del óptimo aprovechamiento que hacer de sus propios recursos en tareas tales como las deformaciones interactivas.

Sistemas Mass-Spring en la GPU [23]

En este trabajo se describen las diferentes implementaciones de este modelo físico (Mass-Spring) en las GPU's. Aquí la CPU se encarga de la simulación mientras que la GPU se encarga de los cálculos necesarios para el renderizado de la figura. Se implementó el modelo basado en mallas triangulares en donde las aristas son resortes conectados a pares de masas puntuales (puntos). Las fuerzas externas que actúan sobre el modelo pueden ser propiciadas por la interacción del usuario, la gravedad o una colisión. En casos muy específicos, los resortes o aristas aplican una cierta cantidad de fuerza en el sentido contrario a su elongación con el fin de permitirle al objeto recuperar su forma original luego de una deformación.
Por otro lado, los diferentes métodos usados para la representación de los objetos (mediante puntos y aristas o mediante un punto) son manejados según la configuración de la memoria, el rendimiento y la precisión numérica de la GPU. En este trabajo se probó que el rendimiento del método basado en aristas y puntos es mayor que el que se basa en un solo punto.

Herramientas algorítmicas para la simulación de microcirugías en tiempo real [9]

Este trabajo muestra las diferentes implementaciones de objetos deformables en las microcirugías. Se realiza un enfoque de la deformación a partir de las fuerzas externas que actúan en el modelo de vasos sanguíneos y sutura, la detección de las colisiones entre objetos rígidos y deformables. Se realizo una serie de experimentos con algoritmos rápidos para la simulación de las deformaciones de objetos blandos y detección de colisiones entre objetos deformables y rígidos en la aplicación usando actores

reales. Con estos algoritmos se obtuvo mayor realismo y precisión, un movimiento lento de los instrumentos quirúrgicos, una técnica de deformación local que minimiza el número de actualizaciones en las representaciones jerárquicas de los objetos deformables. Los algoritmos fueron integrados a un sistema de realidad virtual para la simulación la sutura de vasos sanguíneos pequeños.

Exploración de algoritmos paralelos para sistemas de modelos volumétricos masa-resorte-amortiguado en CUDA [27]

El poder computacional de la GPU se ha venido utilizado para la computación de propósito general desde hace algún tiempo. Este trabajo se basa en la implementación de sistemas volumétricos masa-resorte-amortiguador en CUDA; que difiere relativamente de la misma implementación usando OpenGL.
Se utilizaron dos formas de implementación paralela, una explícita, en la que cada partícula está conectada a otras partículas identificadas por medio de un índice MSDM, y otra implícita, en la que cada figura u objeto está referenciado por datasets que son utilizados para localizar cada partícula dentro de una rejilla regular de tercera dimensión (estructura en malla).

Objetos Deformables Complejos en Realidad Virtual [28]

Aquí se presenta, por un lado, una técnica de animación en tiempo real para objetos deformables dentro de mundos virtuales usando mass-spring. Por otro lado, se proponen varios métodos para la representación de los movimientos y la apariencia de personajes virtuales.
Con respecto a lo último, se muestra que lo más complejo es la representación del vestuario, pero se propone al mismo tiempo una técnica muy eficiente de animación de objetos complejos deformables que hace posible la integración de dichos personajes al sistema de realidad virtual sin que este se vea comprometido en su rendimiento.
La estabilidad de este método se obtuvo a partir de la integración implícita para la derivación del estado de actualización en los puntos mass-spring de la figura. Este método puede producir la animación del vestuario en lapsos de tiempos grandes. El modelo lineal es representado por medio de simples módulos para la computación de matrices de 3x3 y la multiplicación de matrices 3x3 y vectores en espacios tridimensionales, lo que optimiza más el modelo y requiere el mínimo esfuerzo en los cálculos.

Simulación Dinámica de Deformaciones usando Elementos Largos LEM [24]

La idea central en esta clase de algoritmos es que se basan en una estrategia de endentar a partir de elementos largos. Los elementos largos definen una estrategia para mallas en 3d que permiten una aproximación al estado del objeto por medio de un punto en el volumen de la malla a partir de la actualización de un conjunto de puntos reducidos. Lo que hace el método es dividir un objeto por la proyección en un espacio de tercera dimensión ortogonal. La descomposición usa tres planos de referencia perpendiculares entre sí que intersecan el objeto, las posiciones relativas de los puntos dentro del objeto con respecto a la referencia son simuladas en el espacio [24].

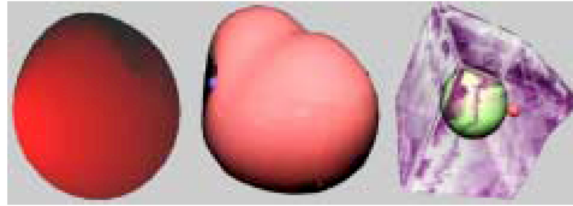


Figura 3.1: Deformaciones de objetos 3d usando métodos largos. Tomado de: [24].

Deformación Vía Cuadrados Móviles "Deformaciones Interactivas en la GPU" [22]

Consiste en hallar la fuerza transformación rígida aplicando la elasticidad que minimice $\sum_i w_i|T_x(p_i) - q_i|^2$ donde $w_i(x) = |p_i - x|^-2$ es la función del peso al hallar las nuevas posiciones q a partir de las posición (x, y, z) en R3 y las posiciones p_i en el espacio formado por el contorno de la figura.

Para el algoritmo una sección de deformación recibe una malla en 3d representada por polígonos de n vértices, y la configuración de los puntos p_i y q_i lo que representa los puntos iniciales y deformados de los puntos de control. La deformación de la figura ocurre cuando hay un cambio o modificación en los puntos q_i por parte del usuario.

Actualizaciones eficientes de los límites de jerarquías de esferas para modelos de deformación geométricos [21]

Esta técnica está basada en los límites de las jerarquías de esferas; el costo de la actualización de la jerarquía depende del número de primitivas más próximas, pero no del número total de primitivas. Las jerarquías de volumen delimitadas han sido muy eficientes para la detección y la aceleración de colisiones entre objetos rígidos. La información sobre el modelo de deformación geométrica es utilizada de forma eficiente, se vuelven a adecuar los límites del volumen de la jerarquía. La geometría del modelo físico proyecta unos puntos hacia las posiciones objetivo del objeto deformable. Para calcular estas posiciones de objetivo, la deformación de la nube de punto es estimada por una matriz de transformación que reduce el mínimo error $\sum ||Aq_i^0 - p_i||2$ entre la deformación y las posiciones transformadas iniciales. El resultado de la matriz se emplea para acelerar la jerarquía.

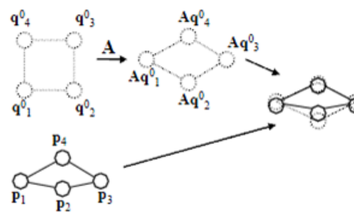


Figura 3.2: La geometría que no está deformada q_i^0 es registrada con la geometría deformada p_i. La matriz A es computada para reducir el error. Tomado de: [21].

Deformaciones lineales: estas deformaciones son estimadas con una matriz de transformación linear aplicada a la nube de puntos que minimiza $\sum ||Aq_i^0 - p_i||2$, estas transformaciones tienen la

capacidad de representar cortes y alargamientos [21].

Deformaciones cuadráticas: este rango de deformaciones se definen a partir de la torcedura y el modo de doblamiento de la figura. Lo que se busca es minimizar i_2^2 , con $A = [AQM] \in R_{3x9}$ donde A, Q, M son matrices de R_{3x3} y que representan términos cuadráticos, lineales y mixtos, teniendo $= (q_x, q_y, q_z, q_x^2, q_y^2, q_z^2, q_x, q_y, q_y, q_z, q_z, q_x)^T \in R^9$.[21]

Objetos Rígidos: este caso es similar a la deformación lineal, solo la rotación $R \in R_{3x3}$ de la matriz A es considerada. R es computado a partir de una descomposición polar. Para extender aun más la gama de deformaciones, la geometría debe ser compuesta en grupos. Los objetos típicos consisten en una agrupación máxima de 10 [21].

Deformaciones definidas a partir de objetos heterogéneos volumétricos [25]

Este trabajo se basa en la técnica de modelamiento FRep (función de representación) para la deformación a partir de objetos heterogéneos volumétricos. Los objetos volumétricos tienen un número de atributos asignados a cada punto y los atributos son tratados como modelos matemáticos. Cada atributo tiene una propiedad característica de la naturaleza arbitraria (el material, fotométrico, físico, estadístico etc.), lo cual varia en el espacio en tercera dimensión. Cada objeto volumétrico no tiene la necesidad de que los atributos estén distribuidos de forma uniforme en el espacio. La deformación se considera como el estado final del modelo de un objeto en tercera dimensión. Se tienen unos puntos objetivos que son desplazados de forma arbitraria hacia otro punto objetivo en el espacio, lo que define el estado final. Esta técnica puede ser fácilmente aplicada a objetos implícitos. También se usan los árboles (FRep tree) para la deformación interactiva de los objetos, que está representado por las deformaciones pequeñas correspondientes a los nodos del árbol, construido a partir de los atributos de la geometría de los objetos volumétricos.

Deformaciones basadas en curvas B-Spline [25]

La función B-Spline es una función $f(t)$ de una sola variable definida por un conjunto de puntos de control P_i, el parámetro P_i de la función pertenece a un espacio paramétrico [0,1]. Los puntos de control se definen en un espacio bidimensional. La primera coordenada es usada para calcular el punto en el espacio, que es reemplazada en el eje x para asegurar que $f(t) = t$, la segunda coordenada sería un escalar. Fuera del dominio de la B-Spline, el valor del escalar que se obtiene es de 0. Para determinar la nueva configuración de puntos se hace la interpolación en las B-Splines, lo que produce cambios en los valores escalares de los puntos de control, lo que lleva a diferentes deformaciones y formas en las figuras. Para la deformación se requiere de un coeficiente escalar para la función para así determinar la configuración de los puntos de control, pero cuando se obtiene un valor de 0 en el escalar, el punto dado queda muy alejado del punto objetivo, lo cual es muy restringido a una deformación de la figura basada en B-Splines. Las deformaciones deben estar centradas en el vector de desplazamiento descrito por la función B-Spline.

Capítulo 4

Resultados

4.1. Pruebas

Las pruebas de rendimiento se realizaron en un equipo que cuenta con una tarjeta de Nvidia Geforce 8800 GT, en el que se tenía previamente instalada y configurada la aplicación (según el manual de usuario de CUDA de este documento). En este capítulo se muestran los resultados de dichas pruebas y algunas conclusiones que se pudieron sacar de las mismas.

4.2. Comparación de la ejecución del modelo Físico en la CPU y la GPU

Los algoritmos que fueron presentados en el capítulo 2 (detección de colisiones y modelo físico de deformación), fueron implementados en C++ usando librerías de OpenGL. Las pruebas se realizaron en la plataforma Windows XP utilizando el Visual Studio 2005 como IDE para el desarrollo.

4.3. Resultados de acuerdo a la eficiencia del modelo físico.

La siguiente tabla muestra el promedio del tiempo de ejecución del modelo físico en ambos procesadores utilizando figuras de diferente número de nodos, así como diferentes tamaños de bloques para el caso de la ejecución sobre la GPU.

Antes de realizar las pruebas se esperaba que al aumentar el tamaño del bloque el rendimiento fuera mayor, y aunque en la mayoría de los casos ocurría, se vio que en algunos el desempeño era inclusive menor que en un caso con menor número de bloques.

En general, el menor tiempo de procesamiento del modelo físico se obtuvo con un tamaño de bloque de 64, cuyo promedio general de rendimiento fue de 0.01263281.

El comportamiento del modelo al cambiar el tamaño del bloque es muy importante porque de eso depende el tiempo de respuesta en cualquier instante de tiempo.

Tpro / No. Nodos						
	CPU	16B	32B	64B	128B	256B
4034	0.010732143	0.005443478	0.00451408	0.00496032	0.0042459	0.00847479
5114	0.010418079	0.004139535	0.00964706	0.00642748	0.00372294	0.00691509
6322	0.016946	0.010863636	0.01029609	0.00661943	0.00886207	0.01310897
8912	0.021374	0.01588	0.01028571	0.00772549	0.01457778	0.00968675
9122	0.020760563	0.02075	0.00664596	0.01326619	0.01742857	0.01820144
10242	0.038816667	0.00755287	0.01903974	0.01659509	0.00969655	0.00561633
22352	0.117305699	0.067503546	0.0590875	0.03283566	0.06366216	0.044

Figura 4.1: Tiempo en milisegundos de cada figura (representada por el No. Nodos) al probar el modelo físico en la CPU y GPU (usando 16 Bloques, 32 Bloques, 64 Bloques, 128 Bloques o 256 Bloques).

Se observo el comportamiento del algoritmo y se pudo determinar, como se preveía, que el tiempo de respuesta es mucho mayor en la CPU que en la GPU. El tiempo de respuesta disminuyó notablemente en la GPU para la mayoría de los tamaños de bloque que se usaron, aunque en la malla de 9122 puntos se obtuvieron resultados relativamente similares tanto en la CPU como en la GPU. Sin embargo fue mucho mayor que todos los tamaños de bloque en la malla más grande (22352 vértices).

A continuación se presenta una gráfica con estos datos:

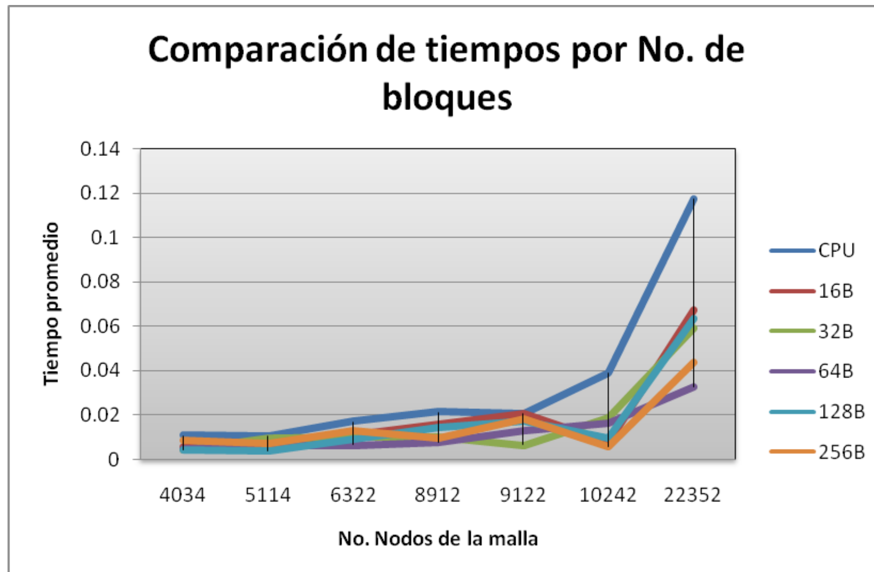


Figura 4.2: reconstrucción de los tiempos por Nodos de la malla en la CPU y GPU

Para optimizar aun más el desempeño del modelo físico propuesto es muy complejo porque requiere de la memoria compartida que es precisamente dividir la memoria global de la GPU en varios bloques mediante el cual los hilos se ejecutarían, pero implicaría reconstruir la aplicación desde 0, ya que para ello se necesita de una nueva configuración de puntos de la malla para organizarlos en los bloques.

4.4. Paralelización del algoritmo propuesto

La finalidad de la paralelización es el de ejecutar el algoritmo más rápido y reducir el tiempo de ejecución de los cálculos de la aplicación, para esto se mide un factor que se denomina Aceleración en inglés "SpeedUp" que determina la rapidez del algoritmo al ser ejecutado en forma paralela, se calcula de la siguiente forma:

$$Aceleracion = \frac{T_s}{T_p} \text{ Tomado de [34]}$$

T_p: Tiempos de ejecución en paralelo.
T_s: Tiempo de ejecución secuencial.

El tiempo secuencial es el empleado por un equipo de un solo procesador y en paralelo por varios. Pero en este caso el tarjeta gráfica cuenta con varios núcleos que son los encargados de realizar esta tarea de procesamiento en paralelo.

A continuación se muestra el Speedup que indica las veces que fue más rápido el modelo en la GPU por el número de bloques de cada figura:

SpeedUp

	16B	32B	64B	128B	256B
4034	1.97155979	2.37747938	2.1636	2.52764755	1.26636093
5114	2.51672697	1.07992283	1.62086504	2.7983445	1.50657078
6322	1.55986288	1.64584654	2.56000516	1.91217008	1.29268566
8912	1.34598196	2.07804659	2.76671033	1.46621755	2.20653988
9122	1.00050908	3.1237857	1.56492316	1.19117987	1.14060012
10242	5.13932667	2.03871884	2.33904498	4.0031413	6.91139777
22352	1.73777092	1.98528791	3.57250879	1.84262827	2.66603862

Figura 4.3: SpeedUp(t_{cpu}/t_{gpu}) de cada figura por el numero de bloques

Al calcular el SpeedUp, se pudo determinar por cada malla cuantas veces fue más rápido el modelo físico en la GPU que en la CPU.

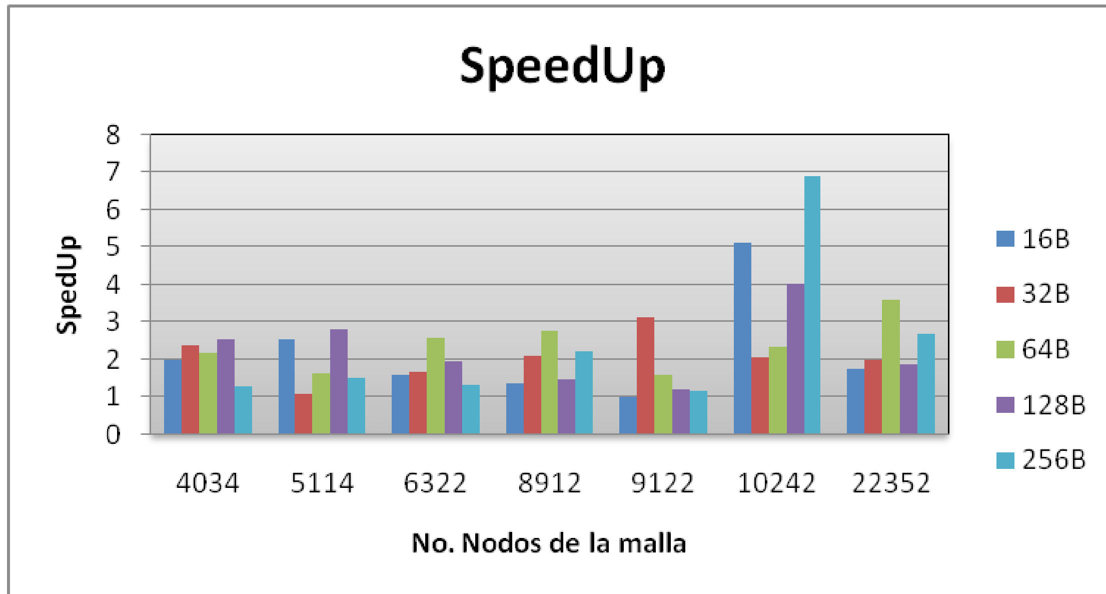


Figura 4.4: Speedup por bloque de cada figura

4.5. Dificultades en la implementación

- Para cargar el árbol de colisiones de cada figura se tenía un arreglo bidimensional pero para una figura de más de 8900 puntos lanzaba un error de ejecución debido a la RAM de la maquina no soportaba la estructura por el tamaño pero se soluciono quitando el arreglo que no era necesario para cargar el árbol.
- Al construir el algoritmo de detección de colisiones se presento algunas dificultades en la implementación ya que se tenía que adecuar las estructuras de datos del modelo físico a las estructuras de datos del algoritmo de detección de colisiones.
- Dada la recursividad del algoritmo de detección de colisiones, el seguimiento de errores se hizo algo complicado pues en tales casos es difícil en que estado se encuentra la aplicación.

4.6. Conclusiones

Los tiempos obtenidos en la GPU por cada bloque no varían mucho pero en algunos casos se obtuvo un rendimiento muy óptimo con respecto a la CPU. Se observaron resultados que sobrepasaban más del doble de lo obtenido en la CPU para algunos bloques, sin embargo, con algunos otros el desempeño era casi igual.
El modelo de procesamiento en paralelo que se está usando es SIMD (Single Model Instruction Mutiple Data), anteriormente mencionado, que consta de un conjunto de procesadores al que a cada uno se le delega una parte del programa que se encargan de sincronizar en un tiempo dado. Lo que se esperaba al final, es que al aumentar el tamaño del bloque el rendimiento aumentara, pero según los resultados, se presentó todo lo contrario, lo cual puede ser explicado por la ley de Amdahl, que afirma que la aceleración máxima está dada por 1/x, si un programa tiene una porción secuencial del 10 % entonces la aceleración máxima es 1/0.1=10 lo que quiere decir que al aumentar el número de procesadores no implica necesariamente aumenta el desempeño [34].
Por los resultados obtenidos, se puede concluir que el trabajo en esta área aporta verdaderos beneficios, por lo que se debe seguir con el trabajo en la aplicación para mejorar aun más el rendimiento con otros modelos de programación paralela más eficientes. El entorno de ejecución y pruebas de la aplicación debe ser adecuado, esto es, que disponga de una unidad que permita la paralelización, en este caso, una tarjeta gráfica que disponga de una GPU de varios núcleos. También se concluye que los entornos, librerías y API'S de CUDA usadas para el desarrollo de la aplicación están bien documentadas y es de fácil uso para el usuario, es importante también recalcar el uso de las herramientas de modelación en 3D dimensión como lo es OpenGL y los programas diseño como lo son Blender y 3ds max para crear las mallas en 3D, son un poco complicados en emplearlos si no se cuenta con algún conocimiento previo en computación gráfica.
Finalmente es importante concluir, que los modelos físicos de deformación pueden tener multiples usos en diversos campos de vida diaria como en los videojuegos, la realidad virtual, la animación en 3D, la simulación en la física de colisiones con deformación entre otros.

Capítulo 5

Anexo 1. Trabajo futuro

La implementación del modelo físico de deformación presentado en este trabajo probo ser más eficiente al emplearlo en una maquina de procesamiento secuencial, esto gracias a las comparaciones realizadas durante las pruebas del modelo. Pues según los resultados que se muestran puede ser más rápido o efectivo el modelo según los bloques. Para un optimizar mas la simulación se podría aplicar la memoria compartida que es aun más rápida que el modelo de programación paralelo que se presenta aquí para que el algoritmo tenga mayor rendimiento, lo que se haría es medir los resultados entre ambos modelos por bloque, lo cual se obtendría resultados más fiables.
Se podría proponer un modelo de vista más realístico para la escena y objetos en tercera dimensión, es decir que las colisiones y deformación sean entre objetos más irregulares y complejos que los que se muestran aquí, implementando un método más eficaz. Lo más importante de esta propuesta es que pueda llegar a ser parte de la industria para probar modelos de carácter más real, lo cual el algoritmo debe ser puesto en prueba en estos tipos de escenarios.

Capítulo 6

Anexo 2. Palabras clave

6.1. Palabras clave

- **CUDA**: es un compilador y un conjunto de herramientas que permiten a los programadores usar una variación del lenguaje de C para codificar algoritmos en GPUs, fue desarrollado por NVidia.
- **CPU (Unidad Central de Proceso)**: AA veces es referido simplemente como el procesador o procesador central, cerebro de un computador que interpreta las instrucciones y procesa los datos de los programas en el computador.
- **GPU (Unidad de Procesamiento Gráfico)**: es un procesador dedicado exclusivamente al procesamiento gráfico, lo cual permite mayor procesamiento de aplicaciones gráficas como videojuegos y 3D interactivas.
- **Gráficos 3D por Computador**: se refiera a una serie trabajos de arte gráfico creados con ayuda del computador, técnicas matemáticas y programas de 3D.
- **NVidia**: es un fabricante estadounidense de procesadores gráficos (GPUs), chipsets, tarjeta gráficas y dispositivos para consolas (Play Station 3).
- **GeForce**: es la denominación que tienen las tarjetas gráficas que cuentan con unidades de procesamiento gráfico (GPU) desarrolladas por la empresa estadounidense nVidia. 3ds Max: programa para la creación de animaciones, renderizado y gráficos 3D desarrollado por Autodesk.
- **Blender**: programa multiplataforma dedicado especialmente a la animación, modelado de escenas y creaciones de gráficos 3D.
- **Árbol**: estructura de datos dinámica compuesta por un nodo especial (v) que es la raíz del árbol y los nodos restantes (v, v, v,. . . , v), que se agrupan en conjuntos de arboles (A, A, A,. . . , A), los cuales son los subárboles del nodo principal (Raíz).
- **Programación en paralelo o paralelismo**: ejecutar un programa por más de un procesador de forma simultánea con la estrategia “divide y vencerás”, los cálculos son realizados en forma paralela.
- **Método de Detección de Colisiones**: algoritmo que se encarga de determinar a cada instante cuando dos objetos entran en contacto. Existen diversos métodos; en este trabajo se utilizarán arboles para optimizar la búsqueda de la colisión.

- **Modelo Físico de Deformación**: algoritmo que se encarga de calcular la deformación de un objeto tridimensional, determinando la nueva configuración de los puntos en cada instante de tiempo, teniendo en cuenta una fuerza externa que actúa con el modelo.
- **Resorte**: operador elástico que se encarga de almacenar la energía y desprenderse sin sufrir mayor deformación cuando cesan las fuerzas o las tensiones al que está sometido el modelo.

6.2. Key words

- **CUDA**: It is a compiler and a collection of tools that allows to the programmers to use a variation of the C programming language to encode algorithms in GPUs, It was developed by nvidia.
- **CPU (Central Processing Unit)**: sometimes it is recounted only as the processor or Central processor, central unit of a computer that encodes the instructions and processes the programs data in the computer.
- **GPU (Graphics Processing Unit)**: It is a processor for graphics processing that allows major processing of graphics applications as video games and 3D interactive environments.
- **3D Computer Graphics**: It refers to works of graphic arts that were created by computer, math technicals and 3D programs.
- **NVidia**: It is an American distributor of graphics processors(GPUs), chipsets, graphics cards and devices for console (Play Station 3).
- **GeForce**: They are the graphics cards that have graphics processing unit (GPU) developed by nVidia.
- **3ds Max**: program for the creation of animation, render and 3D graphics developed by Autodesk.
- **Blender**: multiplataform program dedicated specially to the animation, scene modelling and 3D graphics creation.
- **Tree**: Structure of dynamic data composed by a special node (v) that is the tree's root and the remaining nodes (v, v, v,. . . , v), that are grouped by tree's collections (A, A, A,. . . , A), these are de subtrees of the main node(Root).
- **Parallel Programming or Parallelism**: execute a program by more a processor in simultaneous way with the strategy "divide and conquer", the operations are done in parallel form.
- **Method of Detect Collision**: algorithms that determines each instant when the two objects are in contact. There are a lot of methods: in this project will use trees to optimize the collision's finding.
- **Physic Model of Deformation**: algorithm that calculate the deformation of a tridimensional object, determines the new configuration of points in each instant of time, an extern force acts with the model.
- **Spring**: elastic operator that keeps the energy and drop off without suffering greater deformation when stops the forces or the tensions what the model is subjected.

Capítulo 7

Anexo 3. Manuales de usuario

7.1. CUDA

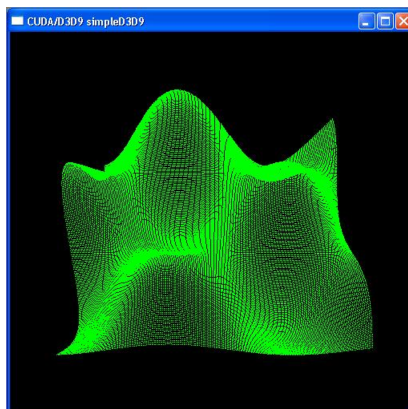
A continuación se presenta un pequeño manual para instalar

1. Se descarga CUDA de la pagina *http : //www.nvidia.es/object/cuda_get_es.html*, el equipo debe contar con una tarjeta que soporte CUDA
2. Se selecciona el sistema operativo.
3. Se seleccionar la versión de CUDA a descargar (2.1, 2.2 o 2.3).
4. Cuando se completa la descarga se instala el programa.
5. En el escritorio se crea un acceso directo a los ejemplos que vienen por defecto.

6. Se puede seleccionar uno de los ejemplos. (Simple Direct3D)

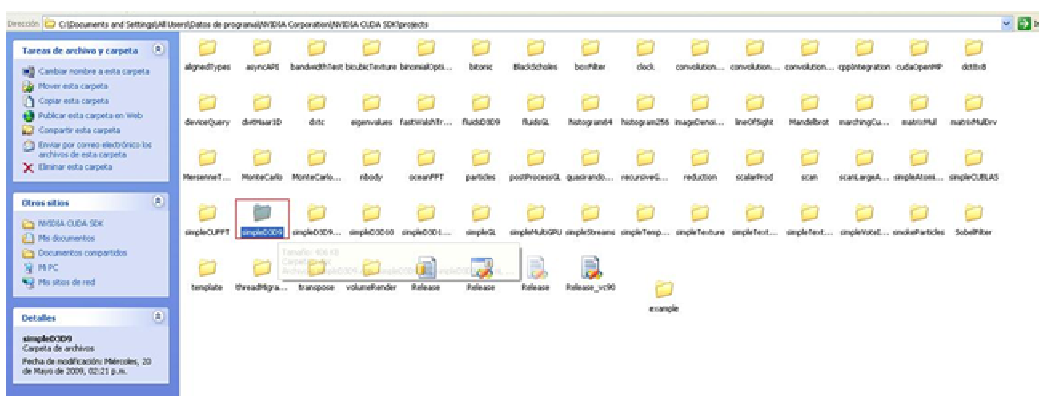
7. Presione run como se muestra a continuación

8. Se Visualiza la aplicación como se muestra a continuación



9. Para ver el código de alguno de los ejemplos seleccione Files:

10. En la carpeta que aparece se selecciona alguno de los ejemplos. Como se muestra en la siguiente imagen, se puede correr directamente en el browser o en Visual Studio .Net.



.

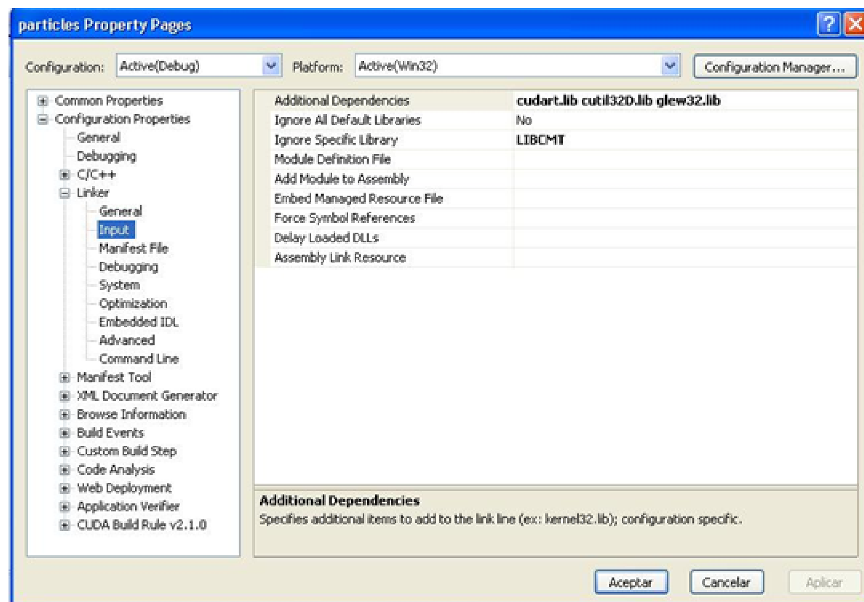
7.2. Ejemplo en CUDA

A continuación muestra cómo crear y correr un ejemplo sencillo que consiste en llenar un arreglo. Lo primero que se debe hacer es configurar el IDE de desarrollo (Visual Studio 2005 en este proyecto) para poder correr CUDA:

1. En la carpeta C:/Documents and Settings/All Users/Datos de programa/NVIDIA Corporation/NVIDIA CUDA SDK/projects se crea una nueva carpeta que se llame Example, después se copia el código completo de cualquiera de los ejemplos (simpleMultiGPU).

2. Se abre el proyecto en el IDE y se eliminan todos los archivos c, c++ o cu que hayan en el proyecto.

3. En propiedades del proyecto debe quedar tal como se muestra a continuación en Linker-¿Input.



4. Se crea un archivo .C (add -¿New Item-¿Templates¿archivo C++).

5. Se escribe el código del programa como se muestra a continuación:

```
#include <stdio.h>
#include <stdlib.h>
#define SIZEOFARRAY 64

extern void fillArray(int *a,int size);

int main(int argc,char *argv[])
{
 int a[SIZEOFARRAY];
 int i;
 for(i=0;i < SIZEOFARRAY; i++) {
        a[i]=0;
 }
  printf("Initial state of the array:\n");
 for(i = 0;i < SIZEOFARRAY; i++) {
   printf("%d ",a[i]);
 }
 printf("\n");
 fillArray(a,SIZEOFARRAY);

 printf("Final state of the array:\n");
 for(i = 0;i < SIZEOFARRAY; i++) {
   printf("%d ",a[i]);
 }
 printf("\n");
 return 0;
}
```

Notas:

- Inicialmente se declaran los header como se hace normalmente en c++.
- La función extern void fillArray(int *a,int size), se define afuera ya que esta función permite comunicar C con CUDA
- Se inicia el arreglo en 0 y se imprime en forma secuencial.
- Después se llama la función que se creo fillArray(a,SIZEOFARRAY) de la GPU que es la encargada de llenar el arreglo.

6. A continuación se crea el .cu sea en el bloc de notas o en cualquier editor de texto:

```
##include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cuda.h>
#include <cstdlib>
#include <cstdio>
#define BLOCK_SIZE 32
__global__ void cu_fillArray(int *array_d){
        int x;

        x=blockIdx.x*BLOCK_SIZE+threadIdx.x;
        array_d[x]=x;
}


extern "C" void fillArray(int *array,int arraySize){
        //a_d is the GPU counterpart of the array that exists
        //on the host memory
        int *array_d;
        cudaError_t result;

        // allocate memory on device
        // cudaMalloc allocates space in the memory of the GPU card
        result = cudaMalloc((void**)&array_d,sizeof(int)*arraySize);

        if (result != cudaSuccess) {
                printf("cudaMalloc failed.");
                exit(1);
        }

        // copy the array into the variable array_d in the device
        // The memory from the host is being copied to the
        // corresponding variable in the GPU global memory
        result = cudaMemcpy(array_d,array ,sizeof(int)*arraySize,
                                cudaMemcpyHostToDevice);
        if (result != cudaSuccess) {
                printf("cudaMemcpy failed.");
                exit(1);
        }

        //execution configuration...
        // Indicate the dimension of the block
        dim3 dimblock(BLOCK_SIZE);

        // Indicate the dimension of the grid measured in blocks
        dim3 dimgrid(arraySize/BLOCK_SIZE);

        //actual computation: Call the kernel, the function that is
```

```
        // executed by each and every stream processor on the GPU card
        cu_fillArray<<<dimgrid,dimblock>>>(array_d);

        //read results back:
        // Copy the results from the memory in the GPU back to the
        //memory on the host
        result = cudaMemcpy(array,array_d,sizeof(int)*arraySize,
                           cudaMemcpyDeviceToHost);
        if (result != cudaSuccess) {
                printf("cudaMemcpy failed.");
                exit(1);
        }

        // Release the memory on the GPU card
        result = cudaFree(array_d);
        if (result != cudaSuccess) {
                printf("cudaFree failed.");
                exit(1);
        }
}
```

Notas:

- Se incluyen los headers para CUDA inicialmente:$< stdio.h >$, $< stdlib.h >$, $< string.h >$, $< cuda.h >$, $< cstdlib >$ y $< cstdio >$
- Se declara la funcion $__global__voidcu_fillArray(int{*}array_d)$ que es la encargada de procesar los datos en la GPU, aquí se asigna blockIdx.x que va a ejecutar el bloque del codigo que retorna el blockId en el eje x y el threadIdx.x que retorna el threadId en el eje x.
- Se implementa la funcion extern "C" void $fillArray(int * array, intarraySize)$ que es la que retorna los calculos a C.
- Con $result = cudaMalloc((void**)\&array_d, sizeof(int)*arraySize)$ se asigna la memoria del dispositivo.

Por último se imprime los resultados que arroja la función fillArray que retorna de CUDA como se muestra a continuación después compilar en Build y de ejecutar con Run en Visual Studio:

```
c:\Documents and Settings\All Users\Datos de programa\NVIDIA Corporation\NVIDIA CUDA S...
Initial state of the array:
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Final state of the array:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
 57 58 59 60 61 62 63
```

7.3. Manual de uso de la aplicación

Para configurar la aplicación se incluyen todos los archivos fuentes que se muestran a continuación en el proyecto de Visual Studio.

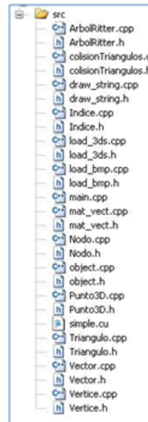


Figura 7.1: Archivos fuentes del proyecto

En propiedades del proyecto debe configurar tal como se muestra a continuación en Linker-¿Input.

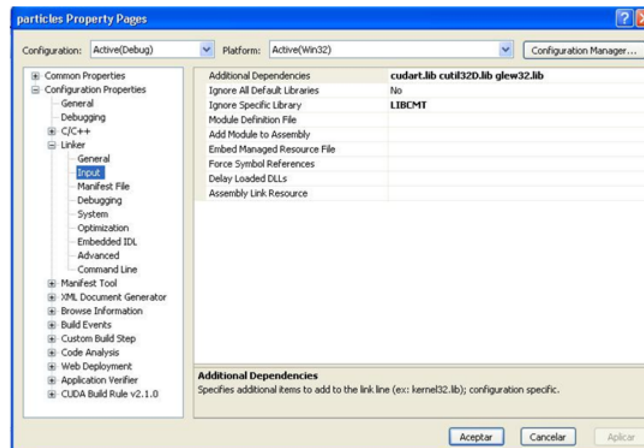
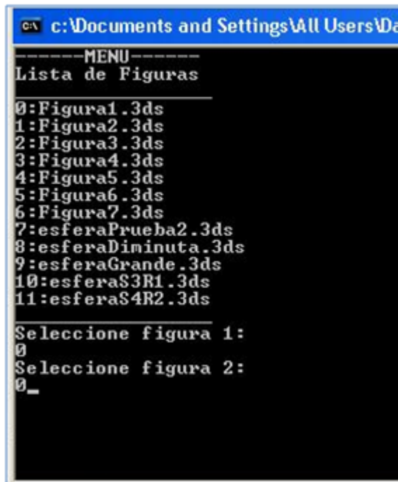


Figura 7.2: Propiedades del proyecto

Después de tener la aplicación configurada con el Visual Studio se ejecuta con continue.

La aplicación desarrollada en este proyecto se debe ejecutar y en el menú se selecciona las figuras (se digita el número del menú de figuras y luego enter) para cargarlas (Figura):



Nota: Si se presiona un número diferente al que aparece en el menú, la aplicación vuelve a pedir las figuras ya que la opción no es válida.

Luego la opción de simulación sea CPU o GPU (Se digita 1 o 2 dependiendo del caso), para salir se presiona 0 (Figura).

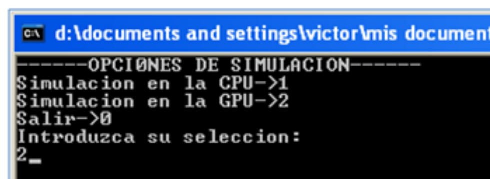


Figura 7.3: Opciones de simulación

Después de que termine de cargar todo en el mundo en tercera dimensión hay unas opciones para la simulación al presionar el botón derecho del mouse como se muestra a continuación.

Hay dos formas de visualización:

1. Wireframe : Muestra las mallas de los objetos 3d.

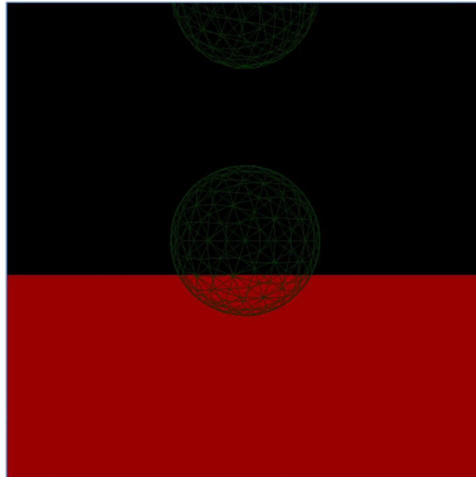


Figura 7.4: Malla de los objetos 3d

2. Smooth: Muestra los objetos sólidos o con textura.

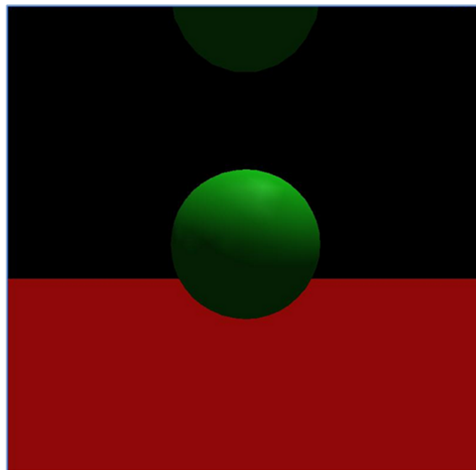


Figura 7.5: Objetos sólidos 3d

Para arrancar la simulación (Run), para detener la simulación (Stop), para salir de la aplicación (Quit).

La aplicación también permite mover la cámara con tres movimientos básicos: acercar, alejar y rotar (sobre el objeto). Se acerca con la flecha arriba, se aleja con la flecha abajo, se rota hacia la derecha o izquierda con las flechas derecha o izquierda respectivamente. Se sube la cámara con la tecla inicio, y se baja con la tecla end.

Archivo de Figuras: Cuando se crean las figuras en 3ds max y en Blender se exportan como .3ds, se copian a la carpeta del proyecto y en el archivo de configuración figures.txt se escribe el nombreDeLaFigura.3ds como se muestra en la siguiente figura, luego se salva el archivo.

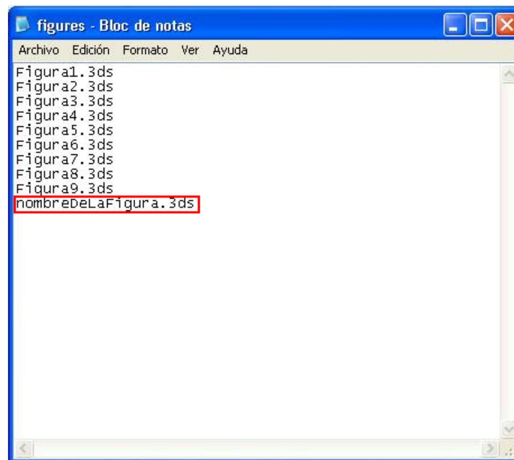


Figura 7.6: Archivo de texto con la lista de figuras disponibles

Al ejecutar la aplicación se tiene:

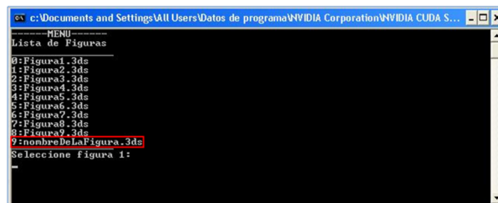


Figura 7.7: Menú con la lista de figuras disponibles

Bibliografía

[1] Página Oficial de CUDA.
Programación lineal y flujo en redes
http://www.nvidia.com/object/cuda_home.html

[2] Open Graphics Library.
http://www.opengl.org/

[3] Página de Oficial de Autodesk 3ds Max.
http://usa.autodesk.com/adsk/servlet/pc/index?siteID=123112&id=13567410

[4] Página oficial de Blender
http://www.blender.org/

[5] SERWAY y BEICHNER.
Tomo I de Física para ciencias e ingeniería
Quinta Edición.
Capitulo 9: Momento lineal y choques.

[6] CHE, BOYER, MENG, TARJAN, SHEAFFER y SKADRON.
A performance study of general-purpose applications on graphics processors using CUDA
(2008)

[7] PÁGINA OFICIAL DE CUDA: Guía de programación.
Guía de programación de CUDA para Windows.
http://www.nvidia.com/object/cuda_home.html.

[8] PÁGINA OFICIAL DE CUDA: Guía de instalación.
NVIDIA CUDA Guía de instalación y verificación en Windows XP y Windos Vista (Edicion C)
http://www.nvidia.com/object/cuda_home.html.

[9] BROWN, SORKIN, LATOMBE, MONTGOMERY y STEPHANIDES
Algorithmic tools for real-time for microsurgery simulation.
(2002)

[10] PÁGINA OFICIAL DE CUDA: Geforce serie 8.
Aplicaciones usando las tarjetas Geforce serie 8 o después con un mínimo de 32 núcleos y 256 MB con memoria dedicada.
http://www.nvidia.com/content/graphicsplus/us/download.asp.

[11] PÁGINA OFICIAL DE CUDA: Área geofísica.
Aplicación de CUDA al área CAD/CAM/CAE.
http://www.nvidia.com/object/optitex.html.

[12] PÁGINA OFICIAL DE CUDA: CAD/CAM.
Aplicación de CUDA al área geofísica.
http : //www.nvidia.com/object/geostar$_o$*il*$_g$*as.html*.

[13] NASA
Proyecto de Realidad Virtual de la NASA usando CUDA.
http://www.prnewswire.co.uk/cgi/news/release?id=115409

[14] PÁGINA OFICIAL DE CUDA: Que es cuda.
Qué es CUDA?.
www.nvidia.es/object/what_is_cuda_new_es.html.

[15] PÁGINA OFICIAL DE CUDA: Computación en la GPU.
Computación en la GPU.
http : //www.nvidia.es/page/gpu_computing.html.

[16] Programación Paralela.
Computación en la GPU.
http : //www.mhpcc.edu/training/workshop/parallel_intro/MAIN.html#what%20*is*%20*parallelism*.

[17] OSCAR RAFAEL GARCIA REGIS Y ENRIQUE CRUZ MARTINEZ.

[18] PÁGINA OFICIAL DE CUDA: Arquitectura.
Arquitectura CUDA de NVIDIA.
http : //www.nvidia.com/object/cuda_home.html.

[19] PÁGINA OFICIAL DE CUDA: Aplicaciones.
Aplicaciones de CUDA en distintas áreas.
http : //www.nvidia.es/object/cuda_in_action_es.html.

[20] PÁGINA OFICIAL DE CUDA: Película.
Película animada usando la tecnología GPU NVIDIA.
http : //www.zanir.szm.sk/10 − 19.*html*.

[21] J. Spillmann, M. Becker, M. Teschner
Efficient Updates of Bounding Sphere Hierarchies for Geometrically Deformable Models.
(2006)

[22] Alvaro Cuno, Claudio Esperanc
Deformaciones Interactivas en GPU.

[23] Joachim Georgii, Rüdiger Westermann
DMass-spring systems on the GPU.
(2005)

[24] REMIS BALANIUK, KENNETH SALISBURY
Dynamic simulation of deformable objects using the Long Elements Method (LEM).
(2002)

[25] SCHMITT B., PASKO A., SCHLICK C
Shape-driven deformations of functionally defined heterogeneous volumetric objects.
(2003)

[26] Wen-Mei Hwu, Christopher Rodrigues, Shane Ryoo and John Stratton
Compute Unified Device Architecture Application Suitability.
University of Illinois, Urbana-Champaign.

[27] Allan Rasmusson, Jesper Mosegaard, and Thomas Sangild Sørensen
Exploring Parallel Algorithms for Volumetric Mass-Spring-Damper Models in CUDA.
(2008)

[28] Young-Min Kang, Hwan-Gue Cho
Complex Deformable Objects in Virtual Reality.
GALab, Department of Computer Science, Pusan National University
2002

[29] Tomas Akenine, Eric Haines, Naty Hoffman
Real Time Rendering, A.K. Peters Ltd.
(2008)

[30] Samuel Ranta-Eskola
Binary Space Partioning Trees and Polygon Removal in Real Time 3D Rendering, Uppsala University
(2001)

[31] Chi-Wing Fu, Tien-Tsin Wong, Wai-Shun Tong, Chi-Keung Tang, Andrew J. Hanson,
Binary-Space-Partitioned Images for Resolving Image-Based Visibility
IEEE Transactions on visualization and computer graphics

[32] J. Nievergelt
Binary Search Trees and File Organization
Universidad de Illinois.

[33] Pascal Volino
Collision Detection for Deformable Objects
University of Geneva.

[34] Christian Trefftz Gómez
Procesamiento Paralelo en EAFIT
$http://www1.eafit.edu.co/drupal/?q=node/549$
Universidad EAFIT, Medellín
Abril, Mayo, Junio de 1998