UNIVERSIDAD
REY JUAN CARLOS

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERÍA INFORMÁTICA

Curso Académico 2011/2012

Proyecto Fin de Carrera

OPTIMIZACIÓN MULTI-GPU DE UN ALGORITMO DE DETECCIÓN DE COLISIONES. CASO DE ESTUDIO CUDA

Autor: Mar Maltese Girón

Tutores: Alberto Sánchez Campos
Marcos Novalbos Mendiguchía

Agradecimientos

Agradezco todos y cada uno de los momentos de atención dedicados por mis profesores de proyecto Alberto y Marcos. Sin su apoyo y decisión este proyecto no se hubiera llevado a cabo.

A mi pareja, por haberme ayudado cada día, por haberme encontrado tantas veces, cuando yo estaba tan perdida.

A mi familia y amigos, por haberme escuchado con paciencia en los momentos más difíciles, por sus palabras de apoyo incondicional, su cariño, su comprensión, su tiempo que aunque no lo tuvieran me dedicaron, su interés y por su confianza.

En especial a quien ya no está, por todo lo que hizo por mí, que aunque no sabía hasta donde iba a llegar, confío en mis capacidades y me animó.

Al que no hace más de cuatro meses que conozco, por haber ejercido de Pepito Grillo cada semana, y que con su constancia, su insistencia y sus peculiares ánimos, ha conseguido que este proyecto llegase a su fin.

Muchas gracias.

Resumen

La evolución en la computación gráfica ha sido notoria en los últimos años. Sin embargo a pesar de todos los avances, uno de los principales problemas que nos encontramos hoy en día para lograr simulaciones realistas es la necesidad de incluir en las mismas el funcionamiento de las leyes básicas de la física, como gravedad, roce, solidez. Así por ejemplo al proporcionar solidez, es necesario tener en cuenta la ley básica universal de que dos objetos no pueden ocupar el mismo espacio al mismo tiempo, lo cual a pesar de ser algo trivial en la vida diaria, resulta un problema complejo de implementar en tiempo real para sistemas complejos. Por tanto para proporcionar solidez, es necesario detectar colisiones entre los diferentes objetos de una simulación.

La detección de colisiones es una tarea costosa ya que se necesita analizar todos los objetos de la escena para buscar contactos entre ellos, y más si trabajamos con simulaciones de cuerpos móviles debido a su movimiento. Si los objetos son deformables y necesitamos resultados en tiempo real, el problema se complica, convirtiendo la detección de colisiones en una tarea de mayor peso computacional.

Por otro lado, actualmente las tarjetas gráficas han aumentado sus capacidades computacionales de manera asombrosa, debido a las exigencias del mundo de los videojuegos, añadiendo multitud de pequeños núcleos que trabajan de manera simultánea. A día de hoy las CPUs se están viendo relevadas a un segundo plano en la ejecución de algoritmos generales, por la utilización de las altas prestaciones de las tarjetas graficas.

Actualmente se están estudiando distintas alternativas para la detección de colisiones en paralelo para hacer uso de los nuevos dispositivos. Los algoritmos generalmente dependen de la infraestructura paralela que utiliza. En este proyecto se presenta un algoritmo escalable y paralelo para la detección de colisiones en simulaciones, que permite obtener un alto rendimiento utilizando tarjetas gráficas nVidia.

Índice general

Agradecimientos I

Resumen III

1. Introducción 1

1.1. Especificación del problema 2

1.2. Metodologías 4

1.3. Objetivos 5

I ESTADO DEL ARTE 7

2. Detección de colisiones 9

2.1. Definición de Detección de Colisión 9

2.1.1. Algoritmo básico 10

2.2. Técnicas básicas usadas en Detección de Colisión 10

2.2.1. Descomposición del espacio 11

2.2.1.1. Octrees 11

2.2.1.2. Arboles binarios 11

2.2.2. Volúmenes envolventes 12

2.2.2.1. Tipos de volúmenes envolventes 12

2.2.2.2. Jerarquía de Volúmenes Envolventes 13

2.2.3. Técnicas básicas usadas en la detección de auto-colisiones 15

2.2.3.1. Volúmenes envolventes 15

2.2.3.2. Jerarquía de Volúmenes Envolventes 16

2.2.3.3. Descomposición del espacio 16

2.2.4. Lista de test 17

3. Arquitecturas de computadores **19**
3.1. SISD (Single Instruction Single Data) 19
3.2. SIMD (Single Instruction Multiple Data) 19
3.3. MIMD (Multiple Instruction Multiple Data) 20
3.3.1. Sistemas de Memoria Compartida. 20
3.3.2. Sistemas de Memoria Distribuida 21
3.3.3. Sistemas de Memoria Compartida Distribuida 22
3.4. MISD (Multiple Instrucion Single Data) 23
3.5. Arquitecturas multicore de memoria compartida 23
3.5.1. Multiprocesadores 24
3.5.1.1. Cell 24
3.5.2. Trabajos relacionados 25
3.6. Arquitecturas GPU 25
3.6.1. GPU frente a CPU 26
3.6.2. Principales diferencias 26
3.6.3. Programación en dispositivos gráficos 28
3.6.4. Enfoque al problema 29
3.6.5. Trabajos relacionados 30
3.6.5.1. Detección de colisión de mallas de modelos 3D en entornos de memoria distribuida 30
3.6.5.2. Implementación de un algoritmo de detección de autocolisiones para objetos deformables 31

4. Representación de la animación de colisiones **33**
4.1. Frame 33
4.1.1. Definición 33
4.1.2. Representación de la colisión 34
4.2. Malla 3D 34
4.2.1. Representación 35
4.3. OpenGL 35

II ESTUDIO Y SOLUCIÓN DEL PROBLEMA **37**

5. Descripción Informática **39**
5.1. Algoritmo de detección de colisiones 41

5.1.1. Algoritmo de detección de colisiones 41
5.1.2. Algoritmo de detección de auto-colisiones 42
5.1.3. Jerarquía de volúmenes envolventes 42
5.1.3.1. Optimización 43
5.1.4. Detección de auto-colisiones 44
5.1.4.1. Test del producto cartesiano de vectores normales 44
5.1.4.2. Cono de normales 45
5.1.5. Test de triángulos 47
5.1.6. Jerarquía para la representación del espacio 47
5.2. Programación en CUDA 51
5.2.1. Detalles de la programación en CUDA 52
5.2.2. Versión multi-GPU 53
5.2.3. Adversidades 53

6. Resultados Experimentales 55
6.1. Resultados de la versión sobre una GPU 57
6.1.1. Tiempos tanto de la CPU como de la GPU en la versión single GPU 57
6.1.2. Tiempos de GPU de la versión single 59
6.2. Resultados de la versión multi-GPU 60
6.2.1. Tiempos completos de la versión multi GPU 60
6.2.2. Tiempos de GPU de la versión multi GPU 61
6.2.3. Comparativa entre la versión Multi GPU y la versión single GPU . 62
6.2.3.1. Speedup 64

7. Conclusiones y trabajos futuros 67
7.1. Conclusiones 67
7.2. Trabajos futuros 69

Bibliografía 72

Índice de figuras

1.1. Resultados de la comparativa realizada por AMD de los rendimientos de CUDA y OpenCL. 2

2.1. Detección, determinación y respuesta a una colisión. 9
2.2. Construcción de un *octree* 11
2.3. Tipos de volúmenes envolventes 12
2.4. Construcción *top-down* del árbol 14
2.5. Construcción *bottom-up* del árbol 14
2.6. Construcción por *inserción* del árbol 15
2.7. Construcción de un *BSP tree* 17

3.1. Modelo SISD 19
3.2. Modelo SIMD 19
3.3. Modelo MIMD 20
3.4. Sistemas de Memoria Compartida 21
3.5. Sistemas de Memoria Distribuida 22
3.6. Sistemas de Memoria Compartida Distribuida 22
3.7. MISD 23
3.8. CPUvsGPU 26
3.9. CPUvsGPU 27

4.1. Representación de una malla 3D 35

5.1. Comparativa de tiempos de ejecución dentro de la GPU y con el programa completo entre CUDA y OpenCL. 39
5.2. Comparativa de tiempos de ejecución dentro del kernel y tiempo de transferencia de datos entre CUDA y OpenCL. 40
5.3. Grid de bloques de threads 40
5.4. Distribución de memoria entre threads 41

5.5. Desglose de un *árbol binario* 43
5.6. Colisión de cajas envolventes 43
5.7. Test de normales, entre primitivas. 44
5.8. Test de normales erróneos 45
5.9. Representación de los vectores fijos 45
5.10. Desarrollo del árbol del cono de normales 46
5.11. Test de triángulos 47

6.1. Secuencia de la malla y la bola. 55
6.2. Secuencia de la manta y la bola. 56
6.3. Tiempos de ejecución obtenidos al ejecutar sobre una gráfica, configurado con el tamaño de bloque a 32 threads. 57
6.4. Tiempos de ejecución obtenidos al ejecutar sobre una gráfica, configurado con el tamaño de bloque a 64 threads. 58
6.5. Tiempos de ejecución obtenidos al ejecutar sobre una gráfica, configurado con el tamaño de bloque a 128 threads. 58
6.6. Tiempos de ejecución obtenidos al ejecutar sobre una gráfica, configurado con el tamaño de bloque a 256 threads. 58
6.7. Tiempos de GPU obtenidos al ejecutar sobre dos gráficas variando el número de threads. 59
6.8. Tiempos de ejecución obtenidos al ejecutar sobre una gráfica, configurado con el tamaño de bloque a 32 threads. 60
6.9. Tiempos de ejecución obtenidos al ejecutar sobre una gráfica, configurado con el tamaño de bloque a 64 threads. 60
6.10. Tiempos de ejecución obtenidos al ejecutar sobre una gráfica, configurado con el tamaño de bloque a 128 threads. 61
6.11. Tiempos de ejecución obtenidos al ejecutar sobre una gráfica, configurado con el tamaño de bloque a 256 threads. 61
6.12. Tiempos de GPU obtenidos al ejecutar sobre dos gráficas, variando el número de threads. 62
6.13. Tiempos de GPU obtenidos al ejecutar sobre una y dos gráficas, configurado con el tamaño de bloque a 32 threads. 62
6.14. Tiempos de GPU obtenidos al ejecutar sobre una y dos gráficas, configurado con el tamaño de bloque a 64 threads. 63

6.15. Tiempos de GPU obtenidos al ejecutar sobre una y dos gráficas, configurado con el tamaño de bloque a 128 threads. 63
6.16. Tiempos de GPU obtenidos al ejecutar sobre una y dos gráficas, configurado con el tamaño de bloque a 256 threads. 63

7.1. Comparativa de tiempos de todos los test que se generan en la secuencia completa, entre la versión de CPU, y el mejor y el peor resultado de Single-GPU. 68
7.2. Comparativa de tiempos de todos los test que se generan en la secuencia completa, entre la versión de CPU, y el mejor y el peor resultado de Multi-GPU. 68

Índice de algoritmos

2.1. Algoritmo básico para la detección de colisiones 10
4.1. Algoritmo para la representación de *frame* 34
5.1. Algoritmo básico de detección de colisión entre triángulos 47

Índice de tablas

6.1. Relación de tiempos y mejora de single-GPU y dual-GPU, con 32 threads por bloque. 64
6.2. Relación de tiempos y mejora de single-GPU y dual-GPU, con 64 threads por bloque. 64
6.3. Relación de tiempos y mejora de single-GPU y dual-GPU, con 128 threads por bloque. 65
6.4. Relación de tiempos y mejora de single-GPU y dual-GPU, con 256 threads por bloque. 65

Capítulo 1

Introducción

En el mundo real, los objetos se comportan de manera determinada debido a la materia que los compone, y a las propiedades físicas de la misma. Dentro de un ambiente simulado los objetos deben presentar los comportamientos similares a los objetos reales a los que representan. Existen muchas herramientas de simulación en el mercado, tanto en el entorno militar como el civil, para múltiples usos. Todas estas herramientas están basadas en formas tridimensionales creadas a partir de polígonos elementales. Además muchos sistemas de simulación están diseñados para representar el mundo real y, por consiguiente, requieren gran precisión, lo que implica utilizar cientos de millones de polígonos. Manejar un número tan grande de polígonos que representan los objetos del mundo simulado, requiere obviamente unos elevados requisitos computacionales.

La detección de colisiones es una de las tareas más costosas a la hora de representar escenarios simulados, consistente en evitar que los cuerpos simulados sólidos no se traspasen. El coste de estos cómputos se acentúa notoriamente si los cuerpos son móviles, deformables y se pretenden resultados en tiempo real, como es evidente.

Por otro lado, en los últimos años las tarjetas han sufrido grandes avances aumentando sus capacidades computacionales de manera asombrosa, debido a las exigencias del mundo del videojuego, añadiendo multitud de pequeños núcleos que trabajan de manera simultánea. De esta manera las CPUs de propósito general, se han visto relevadas a un segundo plano en el campo de la investigación computacional de altas prestaciones, utilizando las tarjetas graficas para ejecutar clásicos problemas computacionalmente costosos.

Frente a un problema de procesamiento gráfico, como al que nos enfrentamos,

detección de colisiones de objetos tridimensionales, se propone solucionarlo mediante la paralelización del problema utilizando tarjetas gráficas. NVIDIA empresa pionera en brindar la posibilidad de codificar algoritmos particulares de manera amigable, para una gran gama de tarjetas gráficas propias, creó en 2007 el SDK de CUDA (Compute Unified Device Architecture) con el objetivo de facilitar y standarizar la programación en sus propias tarjetas gráficas.

Si bien es cierto que CUDA es pionero, no es la única API de estas condiciones. La gran alternativa es OpenCL, creada inicialmente por Apple, y actualmente pertenece al grupo Khronos cuya principal ventaja es el ser una API abierta, capaz de ejecutar sobre cualquier tarjeta gráfica, sin importar el fabricante o el modelo. Teniendo en cuenta que las arquitecturas de las GPU pueden variar en función del fabricante y la tarjeta especifica, sería lógico que los resultados obtenidos por el software propietario sobre sus propias tarjetas gráficas sean mejores, como se ha demostrado en test realizados por AMD en [14]. A parte de las ventajas de rendimiento, CUDA permite la posibilidad de manejar la memoria compartida, soporte de enteros y operadores a nivel de bit.

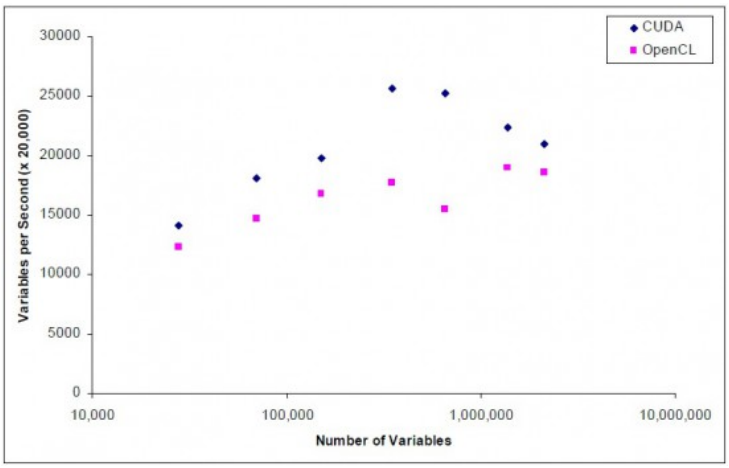


Figura 1.1: Resultados de la comparativa realizada por AMD de los rendimientos de CUDA y OpenCL.

1.1. Especificación del problema

En los ambientes simulados, varios objetos pueden interactuar, y estas interacciones pueden ser que un objeto empuje a otro. La movilidad de un objeto se debe limitar por los demás objetos de la escena, lo cual implica que se pueda tener una colisión con ellos. Si trabajamos con una simulación con N objetos en movimiento y M objetos fijos, cada uno de los N objetos en movimiento pueden chocar con los otros objetos en movimiento, así

como con los objetos fijos y consigo mismo. Para controlar (n /2) + NM pares de objetos en cada tiempo puede llegar a consumir mucho tiempo cuando N y M se incrementan [17], por lo que la detección de colisiones es un gran cuello de botella.

La detección de colisiones es un problema estudiado ampliamente, ya que es vital en varias áreas, como por ejemplo en robótica, realidad virtual, simuladores para formación, entornos virtuales, videojuegos... El método más directo y básico para determinar que dos objetos modelados creados a partir de triángulos colisionan, consiste en verificar si cada uno de los triángulos de un objeto intersecan con algunos de los triángulos de los que forman el otro objeto. Cuando la intersección de triángulos se presenta, se puede asegurar que se ha producido una colisión entre los objetos. Cuanto más realistas son los modelos, más fina es la malla de triángulos que los compone aumentando notoriamente el tiempo de detección.

Para mejorar los tiempos de los algoritmos, se propusieron representaciones basadas en jerarquías de volúmenes envolventes de los objetos, detectando de una forma rápida las áreas donde la colisión realmente se localiza. Algunos ejemplos son: "sphere tree" [21], "OBB- tree" [10], "shell trees" [16] y jerarquías de "k-Dops" [11].

Para la adecuada elección de un método para la detección de colisiones tendremos en cuenta los factores siguientes [6]:

1. El dominio de representación de la aplicación. Las representaciones geométricas que componen los objetos de las escenas, tienen una relación directa con el algoritmo usado. Para representaciones geométricas más sencillas, las soluciones de detección de colisiones a usar serán más sencillas. Por eso decimos que tanto el número de elementos a representar como la técnica usadas repercuten sobre el rendimiento.

2. Diferentes tipos de consultas. Generalmente el mayor esfuerzo computacional consiste en obtener el resultado de un tipo de consulta más detallada. Para la resolución de ciertas consultas se requieren estructuras de datos adicionales. No todas las representaciones soportan todo tipo de consultas. Las consultas más típicas son la detección de interferencias o test de intersección, los cuales incluyen la localización de intersecciones.

3. Parámetros de entorno de la simulación. Las simulaciones en sí mismas contienen

distintos parámetros que tienen un impacto directo en el sistema de detección de colisiones. Estos parámetros incluyen cuántos objetos existen en la escena, sus posiciones relativas, su tamaño, cómo se mueven, si son penetrables, si son rígidos, flexibles, etc.

4. Ejecución en tiempo real. El sistema de detección de colisiones trabaja con un estricto periodo de tiempo y tamaño de memoria. Con el espacio y tiempo siempre existe un intercambio para alcanzar los requerimientos de rendimiento.

5. Robustez. No todas las aplicaciones requieren el mismo nivel de simulación física. Por ejemplo apilar bloques unos encima de otros necesita más trabajo en la detección de colisiones que si se bota un balón en una cancha de baloncesto. Si la pelota rebota un poco antes o un poco después es irrelevante, en cambio en el mundo de los bloques, los bloques podrían comenzar a penetrarse unos a otros.

6. Facilidad de uso e implementación. Muchos proyectos trabajan sobre un marco de tiempo. La programación característica de detección de colisiones no sirve de nada si no se puede llevar a completarse y finalizar en tiempo real. Las decisiones relativas a la simplicidad de la implementación, desempeñan un papel importante en la forma de enfocar el problema.

1.2. Metodologías

Las metodologías en este área plantean diversas cuestiones como, por ejemplo, cuál es la manera óptima de implementar estas funciones en un entorno computacional. Cierto hardware especial como las aceleradoras gráficas, tarjetas gráficas con dos procesadores, los ordenadores con multiprocesador o los *cluster* de computadores, tienen una significativa influencia en la implementación de esas funciones.

Para alcanzar los requerimientos de las funciones geométricas (colisiones polígono-polígono en el espacio donde cada polígono tiene su propia velocidad y aceleración) conviene utilizar:

1. Algoritmos inteligentes para reducir la complejidad de las funciones.

2. Un número elevado de procesadores para utilizarlos en paralelo o de manera distribuida.

En este proyecto se presenta un algoritmo paralelo para la detección de colisiones. Dicho algoritmo es apropiado tanto para modelos geométricos complejos como simples, aunque destaque en trabajos complejos.

1.3. Objetivos

El objetivo de este proyecto consiste en la implementación de un algoritmo para la detección de colisiones y auto colisiones mediante la utilización de tarjetas gráficas como elementos de cómputo y no sólo de visualización.

La implementación del algoritmo se basa en la utilización de CUDA (Compute Unified Device Architecture) como API para interactuar con la tarjeta gráfica, el cuál hoy en día es la API de este tipo más popular para programación en GPUs. Los puntos fuertes de CUDA son los siguientes:

1. La capacidad de realizar lecturas dispersas, es decir que se puede consultar cualquier posición de memoria.
2. La memoria compartida de alta velocidad disponible para el programador que se comparte entre los threads, y que puede ser utilizada como caché.
3. La capacidad de realizar lecturas rápidas desde y hacia la GPU
4. Soporte de enteros y operadores de bit.

No obstante, CUDA también ofrece ciertas limitaciones:

1. La incapacidad de ejecutar código recursivo o con punteros a funciones externas. Tampoco permite el uso de variables estáticas dentro de funciones, ni la sobrecarga de funciones.
2. No soporta renderizado de texturas.
3. Los números desnormalizados o NaNs no son soportados en precisión simple.

La elección de la utilización de tarjetas gráficas como soporte para la ejecución de este proyecto, se debe a su mayor capacidad de cómputo sobre operaciones matemáticas. Se ha asumido la necesidad de aprender una manera diferente de programar en comparación, tanto con las alternativas tradicionales HPC, como con la programación tradicional de CPU. Y finalmente después espués de valorar sus ventajas y desventajas la elección de

CUDA frente a otras APIs del tipo se debe, a su extensión, rendimiento, popularidad y a la disponibilidad de arquitecturas Nvidia para las pruebas.

Parte I

ESTADO DEL ARTE

Capítulo 2

Detección de colisiones

A continuación se expondrá los detalles de la detección de colisiones, y los métodos usados habitualmente para su resolución.

2.1. Definición de Detección de Colisión

El término *detección de colisión* es parte del término general *manipulación de la colisión*, que se divide en tres partes[4]:

- *detección de colisión.* El resultado de la *detección de colisión (DC)* es un valor booleano que indica si dos o más objetos colisionan.
- *determinación de colisión.* La *determinación de colisión* es el cálculo de las intersecciones entre objetos.
- *respuesta a la colisión* (ver figura 2.1). La *Respuesta a Colisión* son las acciones a realizar en respuesta a la colisión.

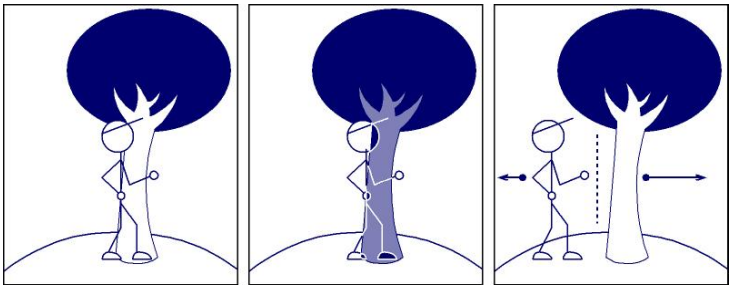


Figura 2.1: Detección, determinación y respuesta a una colisión.

La detección de colisiones el primer paso para solventar diferentes problemás:

- Que dos objetos, o uno consigo mismo colisionen y no deseemos la penetración.

- Que un objeto necesite colisionar con otro en ciertas condiciones y de una manera determinada, para estudiar el comportamiento o simplemente registrar el incidente.
- Que el objeto que realice un recorrido, evitando colisionarse con otros objetos, como un robot, transitando por un pasillo de obstaculos.
- Simulaciones para descubrir, como dos objetos simulables encajarian.
- Simulaciones de objetos que interactuan sobre otros de maneras complejas. Provocando trayectorias complicadas de predecir, y que necesitan de costosos calculos matemáticos.

2.1.1. Algoritmo básico

Veamos a continuación un algoritmo genérico de detección de colisiones [13] (Algoritmo 2.1). El Algoritmo 2.1 trabaja sobre una simulación en la que existen M objetos estáticos y N en movimiento para obtener las colisiones entre los objetos.

```
for t = t_i to t_f do
  for all Objeto_i in {ObjetoDinam_1, ..., ObjetoDinam_N} do
    MOVER Objeto_i A SU POSICIÓN EN EL TIEMPO t
    for all Objeto_j in {ObjetoDinam_1, ..., ObjetoDinam_N}, j ≠ i do
      MOVER Objeto_j A SU POSICIÓN EN EL TIEMPO t
      if interpenetra (Objeto_i, Objeto_j) then
        return Colisión en el tiempo t
      end if
    end for
    for all Objeto_k in {ObjetoDinam_1, ..., ObjetoDinam_M} do
      if interpenetra (Objeto_i, Objeto_k) then
        return Colisión en el tiempo t
      end if
    end for
  end for
end for
```

Algoritmo 2.1: Algoritmo básico para la detección de colisiones

2.2. Técnicas básicas usadas en Detección de Colisión

En aplicaciones en las que es necesario conocer la colisión entre objetos en un tiempo reducido , es necesario reducir el número de test de colisión. Para ello se hacen necesarias estructuras de datos especiales que reduzcan estos $O(n^2)$ pares de tests que se deberían

de realizar en el peor de los casos.

En esta sección se habla de algunas de las estrategias para la reducción de la complejidad.

2.2.1. Descomposición del espacio

La descomposición espacial es una técnica utilizada para descartar parejas de objetos de una manera rápida. Concretamente se divide el espacio en zonas, de manera que sólo se realizan tests de colisiones entre los objetos, que se encuentren en la misma zona.

A continuación se expondrá las técnica para la descomposición espacial, más empleadas.

2.2.1.1. Octrees

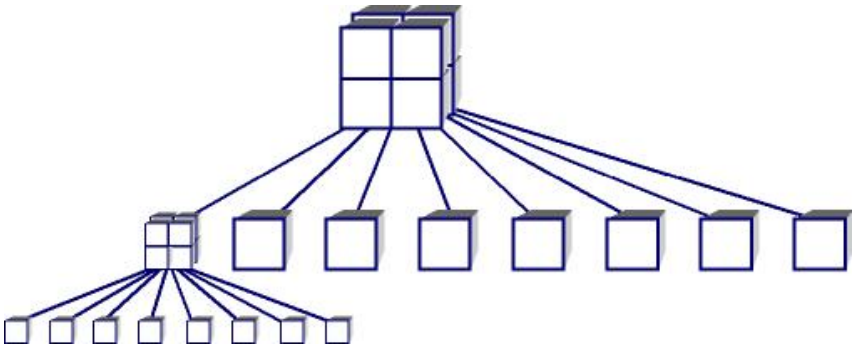


Figura 2.2: Construcción de un *octree*

Un *octree* consiste en una división jerárquica del espacio mediante cajas alineadas con los ejes. Su representación consiste en un árbol octario en el que cada nodo padre se divide en ocho hijos. El nodo raíz representa el volumen que recubre al objeto por completo. En la Figura 2.2 se puede ver gráficamente el proceso de construcción de la jerarquía.

2.2.1.2. Arboles binarios

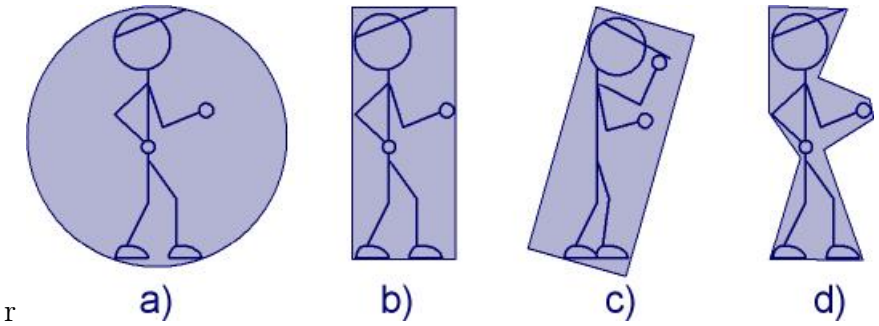
Un *octree* consiste en una división jerárquica del espacio mediante cajas alineadas con los ejes. Su representación consiste en un árbol octario en el que cada nodo padre se divide en ocho hijos. El nodo raíz representa el volumen que recubre al objeto por completo. En la Figura 2.2 se puede ver gráficamente el proceso de construcción de la jerarquía.

2.2.2. Volúmenes envolventes

Comprobar la colisión entre dos objetos de manera tradicional es muy costoso, sobre todo cuando hablamos de alto detalle, donde entran en juego miles de polígonos. Para reducir estos costes, se aplica la técnica de volúmenes envolventes, consistente en el recubrimiento del modelo, por formás geométricas sencillas. La principal ventaja de esta técnica es el bajo coste computacional de las colisiones entre los volúmenes.

Para crear el volumen envolvente óptimo, necesitamos considerar [3]:

1. Debe ajustarse al objeto tanto como sea posible.
2. El cálculo de colisión entre volúmenes debe de ser rápida.
3. El cálculo y recalculamiento de las dimensiones del volumen envolvente debe ser sencillo.
4. Se debe poder rotar y transformar los volúmenes fácilmente.
5. El almacenamiento de los volúmenes debe consumir poca memoria.



r

Figura 2.3: Tipos de volúmenes envolventes. a) Esfera. b) Caja envolvente alineada con los ejes. c) Caja envolvente orientada. d) k-DOP

En la Figura 2.3 se muestran los volúmenes envolventes más representativos a) *esferas envolventes (BS)*, b) las *cajas envolventes alineadas con los ejes (AABB)*, c) las *cajas envolventes orientadas (OBB)* y d) los *k-DOPs*.

2.2.2.1. Tipos de volúmenes envolventes

- Esfera envolvente Las principales ventajas de la esfera envolvente es que tiene un test de colisión poco costoso, no necesita rotaciones, ya que es completamente simétrica, y utiliza poca memoría para su almacenamiento. El inconveniente es que no se ajusta

adecuadamente a ciertos objetos, incumpliendo una de las características básicas para un buen volumen envolvente.

- Caja envolvente alineada con los ejes Es una caja poliédrica, alineada con los ejes de coordenadas. La ventaja de este tipo es que también tiene un test de colisión poco costoso y más rápido que el de esferas, ya que solo se comparan los valores de las coordenadas. Tampoco se ajusta a todos los objetos y precisa de rotación.

- Caja envolvente orientada Este tipo de volumen envolvente es representado por una caja rectangular, como en el anterior caso, aunque añadiendo que la orientación varía (ya no es paralelo a los ejes). Al contrario que las cajas alineadas, éste tiene un test de colisión mucho más costoso y necesita guardar más información para la representación, ya que se necesita almacenar la caja y una matriz de rotación. La principal ventaja es que dispone de un mejor ajuste a los objetos y una rápida actualización en las rotaciones.

- Politopo de orientación discreta Este volumen es similar a una caja envolvente orientada, la diferencia es que este tipo de volumen secciona las esquinas de la caja, permitiendo un mejor ajuste, complicando el proceso de detección y añadiendo datos para guardar en memoria.

2.2.2.2. Jerarquía de Volúmenes Envolventes

La utilización de volúmenes envolventes, puede disminuir el tiempo del test de colisión entre objetos. El número de parejas de objetos sobre el que realizar este test no se reduce, por tanto, el tiempo de ejecución es el mismo menos una constante. Al unir los volúmenes envolventes y una jerarquía, en una jerarquía de volúmenes envolventes, podemos llegar a reducir la complejidad a logarítmica.

Para cualquier objeto podemos construir una jerarquía de volúmenes envolventes en forma de árbol. Para ello se obtiene el volúmen envolvente del objetos entero, y se coloca en el nodo hoja del árbol. Los nodos raíz se van dividiendo de manera que los hijos de un nodo, tengan las divisiones del mismo. Con la jerarquía lo que se consigue es ahorrarse teste, ya que solo se comprobará la colisión entre hijos de un nodo si se produce colisión entre los nodos padre.

Las jerarquías de volúmenes envolventes incluyen entre otros: AABB-Trees, OBB-Trees, Box-Trees, Sphere-Trees, k-DOP trees, Shell-Trees, R-Trees y QuOSPO-Trees.

La construcción de un árbol de volúmenes envolventes es un proceso complejo, sobretodo si se quiere obtener un árbol óptimo. Las 3 estrategias principales son:

- Top-down. Éste es el método más usado, consiste en partiendo de gran un volumen envolvente que contenga a todos los objetos de la escena ir dividiéndolo en volúmenes más pequeños, a la vez que se divide el árbol. Este método es rápido y fácil de implementar aunque no genere el árbol óptimo.

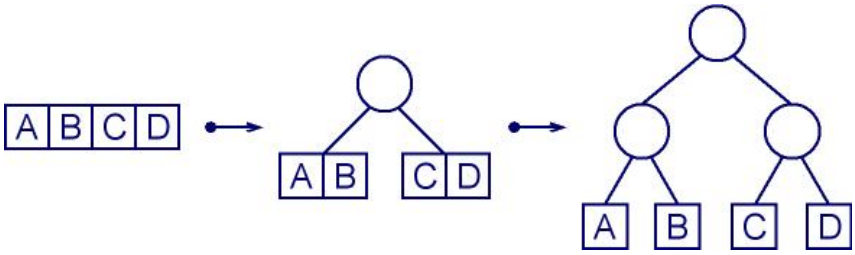


Figura 2.4: Construcción *top-down* del árbol

- Bottom-up . En este caso se parte de los objetos, los triángulos, y se va generando los volúmenes envolventes más grandes uniendo los más cercanos. Con este método si generamos el árbol óptimo aunque la complejidad de implementación es mucho mayor y consume más tiempo de ejecución que el anterior.

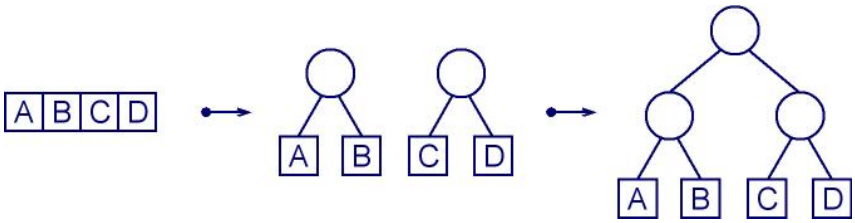


Figura 2.5: Construcción *bottom-up* del árbol

- Inserción. A partir de un árbol vacío se van insertando los nodos de forma que se mantenga el equilibrio entre las ramás. Este método tiene tiempos equivalentes a Top-down y ofrece un mejor resultado ya que los arboles balanceados son más rápidos de explorar y manejar.

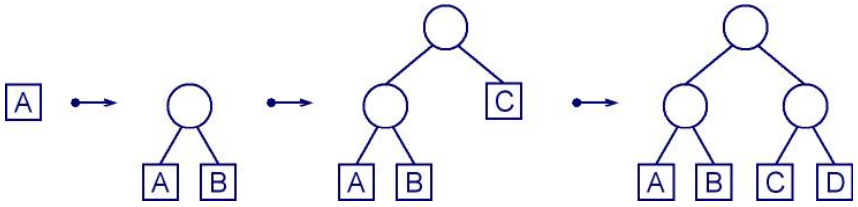


Figura 2.6: Construcción por *inserción* del árbol

2.2.3. Técnicas básicas usadas en la detección de auto-colisiones

Como auto-colisión se entiende la penetración de un objeto consigo mismo, muy habitual en simulados de telas y elementos flexibles. Como es evidente si nos enfrentamos a un problema de auto-colisiones se busca reducir al máximo el numero de test de auto-colisión para resolverlo lo antes posible. Existen diversas estructuras de datos específicas para este problema así como heurísticas, con el único fin de mitigar este problema. En el peor de los casos el problema se ajustaría a una complejidad proporcional a $O(n^2)$ pares de test. A continuación se presentaran las estrategias más comunes para la reducción de esta complejidad.

2.2.3.1. Volúmenes envolventes

Como en la detección de colisiones ordinaria la realización de las comprobaciones sobre si colisionan o no, desde los triángulos que forman los modelos, es muy costoso. Por esta razón y con el fin de preservar los recursos se crean volúmenes geométricos que recubran los objetos, de manera que con estos objetos se simplifique la detección y sobretodo se diferencien las zonas conflictivas de las que no lo son. Así que de la misma manera que para la detección de colisiones entre diferentes objetos se utiliza esta estructura para la localización de las zonas problemáticas y candidatas a tener colisiones, se aplica de la misma forma para el problema de un objeto consigo mismo.

La adición de este proceso supondrá un coste en tiempo computacional, muy notable si vamos a tratar una malla completamente colisionada. No obstante si la malla colisiona solo en unos pocos puntos localizados ahorraremos mucho tiempo en realizar test de triángulos que en realidad no se tocan.

Esfera envolvente Este tipo de volumen envolvente es utilizado en colisiones entre varios objetos como hemos podido ver anteriormente, también es adecuado para auto-colisiones, sobre todo por su simplicidad a la hora de detectar una colisión. Simplemente

con dos parámetros el centro y el radio, tenemos todo lo necesario para realizar el calculo con total precisión.

Caja envolvente alineada con los ejes Este volumen, consiste como ya hemos visto con anterioridad en poliedros alineados a los ejes de manera que con una simple comprobación de las coordenadas de ambas cajas esta resuelto el test.

Caja envolvente orientada A diferencia de su hermana la caja envolvente alineada con los ejes, es justamente la orientación de la caja, ya que esta puede variar, por lo que, como es evidente se necesita guardar el grado de rotación, y aunque se ajuste mejor a los objetos proporciona un test más costoso de comprobar.

Politopo de orientación discreta El considerado politopo es descendiente directo de la caja envolvente orientada, con la evolución de que no se trata de una caja si no que tiene las esquinas con la capacidad de ser seccionadas o no, según se adapte mejor al modelo. Su principal ventaja es un mejor ajuste y como es lógico la desventaja es que es un test costoso, porque si en las cajas envolventes orientadas, el test se complicaba ahora que no se tratan de poliedros perfectos aun se complica más.

2.2.3.2. Jerarquía de Volúmenes Envolventes

Como ya se vio en el apartado 2.2.2.2 las jerarquías de volúmenes envolventes son muy útiles, ya que es gracias a estas que se eliminan las zonas fuera de conflicto, reduciendo el número de test a realizar.

Construcción de un árbol de jerarquías envolventes La construcción de un árbol de volúmenes envolventes es la manera más eficiente de organizar los volúmenes, ya que no solo conseguimos ordenación si no que es la estructura más fácil de explorar. Conseguir el árbol optimo no es un trabajo trivial o sencillo. Requiere de un estudio de las necesidades así como de las posibilidades a barajar, las cuales ya han sido explicadas con anterioridad en el apartado 2.2.2.2.

2.2.3.3. Descomposición del espacio

Dividiendo el espacio en zonas más pequeñas se obtendrán mejoras en el tiempo de ejecccución ya que se eliminaran un gran número de test. Para hacer estas divisiones de

manera adecuada al problema concreto se deben explorar al menos los siguientes métodos de descomposición del espacio utilizando árboles.

Octrees Como se vio en el apartado 2.2.1.1 la utilización de octrees no solo es adecuada para la división del espacio para la detección de colisiones entre varios objetos, si no también para la detección de auto-colisiones. Lo más especial de este tipo de jerarquías es que el nodo raíz es el que incluye tridimensionalmente a sus hijos, como se puede ver en la figura 2.2, y así recursivamente hasta llegar a los nodos hojas compuestos de dependiendo del caso, de un tamaño mínimo a dividir o según la profundidad máxima que se desee para el árbol.

BSP trees A diferencia de los anteriores, esta jerarquía no ha sido mencionada con anterioridad, pero a la hora de las auto-colisiones es más relevante, ya dentro de un mismo objeto hay zonas más candidatas a la colisión que otras, por eso puede ser más interesante dividir más unas zonas que otras. Su funcionamiento es muy similar a los octrees, la única diferencia clara es que este tipo de jerarquías solo se puede dividir en dos regiones, también se diferencia en que la orientación y la posición de esas regiones no se rigen a ninguna norma, como se puede ver en la figura 2.7.

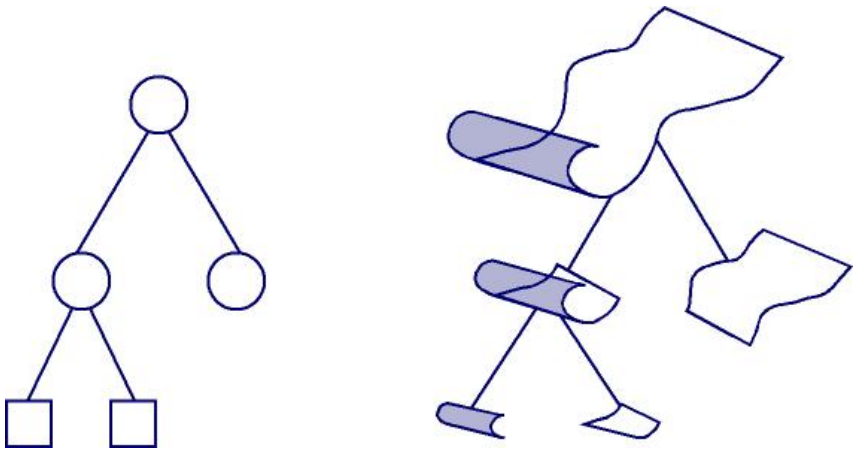


Figura 2.7: Construcción de un *BSP tree*

2.2.4. Lista de test

En vez de utilizar el frente de colisiones, el cual consiste en dos listas para almacenar los test de colisiones que han dado negativo, se utilizará una lista en la que se almacenarán los test de los triángulos candidatos a colisionar. Es decir, depués de

realizar las comprobaciones necesarias y los test correspondientes sobre la/las jerarquía/as correspondientes, se realizará dicha lista con los test de triángulos candidatos, para ser enviada a la gráfica para ser ejecutada de manera paralela.

Capítulo 3

Arquitecturas de computadores

A continuación se muestra la taxonomía presentada por Flynn (1972), que considera dos características esenciales: el número de flujo de instrucciones y el de flujo de datos.

3.1. SISD (Single Instruction Single Data)

Este es el modelo tradicional de computación secuencial donde una unidad de procesamiento recibe una sola secuencia de instrucciones que operan en una secuencia de datos.

Figura 3.1: Modelo SISD (Single Instruction Single Data)

3.2. SIMD (Single Instruction Multiple Data)

En este caso se tienen múltiples procesadores que sincronizadamente ejecutan la misma secuencia de instrucciones, pero en diferentes datos utilizando memoria distribuida.

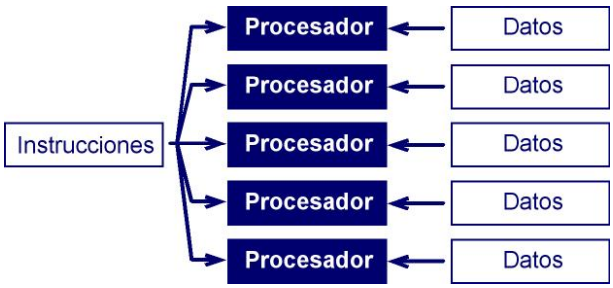


Figura 3.2: Modelo SIMD(Single Instruction Multiple Data)

Aquí se tienen N secuencias de datos, una por procesador, de manera que hay diferentes datos usados por cada procesador. Los procesadores operan sincronizadamente y un reloj

global se utiliza para asegurar esta operación. Es decir, en cada paso todos los procesadores ejecutan la misma instrucción, cada uno en diferente dato. Todos los procesadores ejecutan las instrucciones simultáneamente. Esto tarda un tiempo menor en comparación con una máquina secuencial.

3.3. MIMD (Multiple Instruction Multiple Data)

Este tipo de arquitectura es paralela igual que SIMD, la diferencia es que MIMD es asíncrono. No tiene un reloj central. Cada procesador en un sistema MIMD puede ejecutar su propia secuencia de instrucciones y tener sus propios datos.

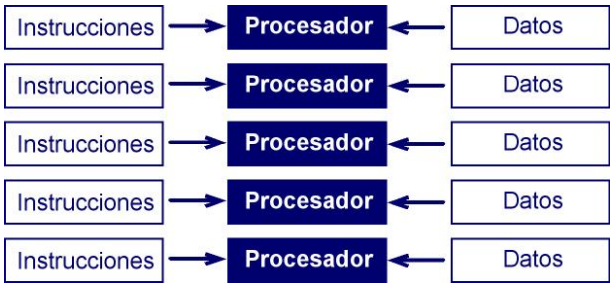


Figura 3.3: Modelo MIMD (Multiple Instruction Multiple Data)

Se tienen N procesadores, secuencias de instrucciones y de datos. Cada procesador opera bajo el control de una secuencia de instrucciones, ejecutada por su propia unidad de control, es decir cada procesador ejecuta su propio programa con diferentes datos. Esto significa que los procesadores pueden estar haciendo diferentes cosas en diferentes datos al mismo tiempo.

Los sistemas MIMD se clasifican en tres grupos:

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

3.3.1. Sistemas de Memoria Compartida.

En este tipo de sistemas cada procesador tiene acceso a toda la memoria, es decir, hay un espacio de direccionamiento compartido. Se tienen tiempos de acceso a memoria uniformes, ya que todos los procesadores se encuentran igualmente comunicados con la

memoria principal, y las lecturas y escrituras de todos los procesadores tienen exactamente las mismas latencias. No obstante, debe asegurarse que los procesadores no tengan acceso simultáneamente a regiones de memoria de una manera descontrolada. La principales desventajas de esta configuración son dos, la primera consiste en que el acceso simultáneo a memoria siempre es un problema, y la segunda, que no se puede incrementar en exceso el número de CPU's, ya que podrían producir un cuello de botella. La principal ventaja es la facilidad de la programación, mucho más fácil que en sistemas de memoria distribuida.

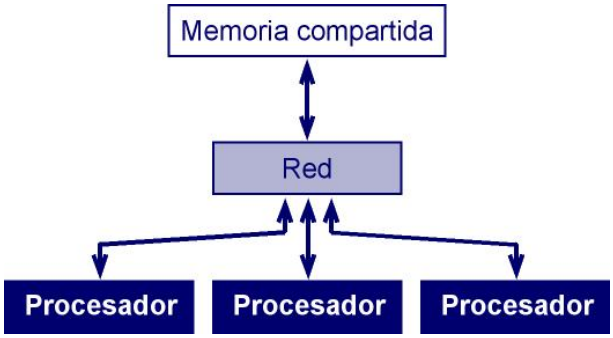


Figura 3.4: Sistemas de Memoria Compartida

Las computadoras MIMD con memoria compartida son sistemas conocidos como de multiprocesamiento simétrico (SMP), donde múltiples procesadores comparten un mismo sistema operativo y memoria. Otro término con el que se les conoce es multiprocesadores.

3.3.2. Sistemas de Memoria Distribuida

Estos sistemas tienen su propia memoria local. Los procesadores pueden compartir información solamente enviando mensajes, es decir, si un procesador requiere los datos contenidos en la memoria de otro procesador, deberá enviar un mensaje solicitándolos. Esta comunicación se conoce como paso de mensajes. La principal ventaja es la escalabilidad, ya que si la demanda de los recursos crece. siempre se puede agregar más memoria y procesadores.

Las desventajas son principalmente dos, que el acceso remoto a memoria es lento y que la programación se complica.

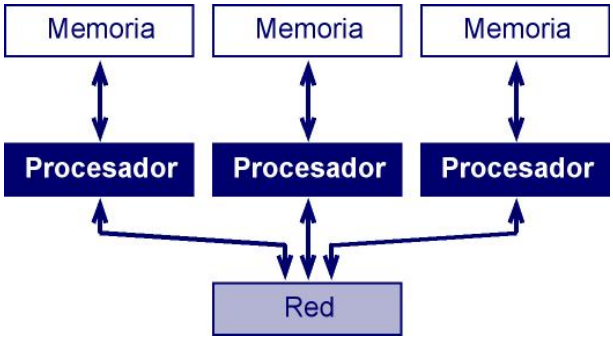


Figura 3.5: Sistemas de Memoria Distribuida

Las computadoras MIMD de memoria distribuida son conocidas como sistemas de procesamiento en paralelo masivo (MPP), multicomputadoras o *cluster*, donde múltiples procesadores trabajan en diferentes partes de un programa, usando su propio sistema operativo y memoria.

3.3.3. Sistemas de Memoria Compartida Distribuida

Es una partición de procesadores que tienen acceso a una memoria compartida común pero sin un canal compartido. Esto significa que físicamente cada procesador posee su memoria local y se interconecta con otros procesadores por medio de un dispositivo de alta velocidad, y todos ven las memorias de cada uno como un espacio de direcciones globales. La configuración de memoria de este modelo, ofrece una penalización de tiempo de acceso a la memoria local de otro procesador. Las ventajas son: la escalabilidad como en los sistemas de memoria distribuida, que es fácil de programar, y que no existe el cuello de botella de las máquinas de sólo memoria compartida.

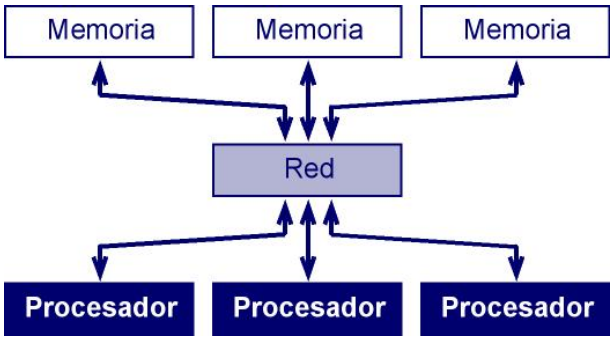


Figura 3.6: Sistemas de Memoria Compartida Distribuida

3.4. MISD (Multiple Instrucion Single Data)

En este modelo, secuencias de instrucciones pasan a través de múltiples procesadores. Diferentes operaciones son realizadas en diversos procesadores. N procesadores, cada uno con su propia unidad de control comparten una memoria común.

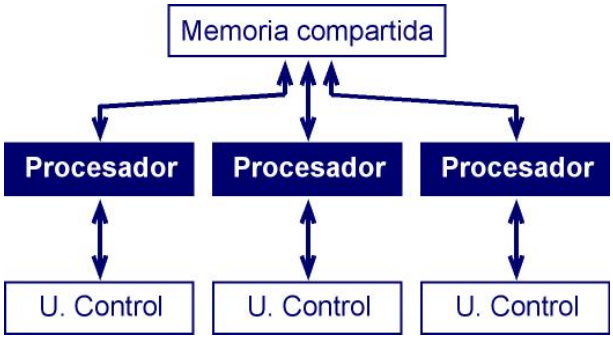


Figura 3.7: MISD (Multiple instruction Single Data)

Aquí hay N secuencias de instrucciones y una secuencia de datos. Los procesadores realizan diferentes acciones al mismo tiempo en el mismo dato. Son útiles sobretodo en cómputos donde la misma entrada está sujeta a diferentes operaciones.

3.5. Arquitecturas multicore de memoria compartida

En esta arquitectura existen varios procesadores dentro del mismo núcleo, que ejecutan de manera independiente, pero que acceden a la misma memoria. Los sistemas de memoria compartida distribuida (DSM, distributed shared memory) dan un espacio de direcciones virtual entre los procesadores aunque estos estén débilmente acoplados. Las principales ventajas de este tipo de sistema es su facilidad de programación y la escalabilidad que depende directamente del ancho de banda de la red de comunicaciones. Los sistemas se pueden clasificar en lo siguiente:

- DVSM (Distributed Virtual Shared Memory). En los sistema DVSM la capa de software, desde el sistema operativo o desde compilación, suministra y gestiona un espacio compartido de direcciones.

- NUMA (Non-Uniform Virtual Shared Memory). En los sistemas NUMA, cada nodo contiene uno o varios procesadores, con caches privadas y un módulo de memoria, que almacena una parte determinada del espacio compartido de direcciones.

- CC-NUMA (Cache-coherent Non Uniform Memory Access). Los sistemas CC-NUMA pretenden reducir el tiempo de acceso a los datos residentes, en un nodo remoto, utilizando cachés que almacenen los datos remotos.

- COMA (Cache Only Memory Architecture). En los sistemas COMA los módulos de memoria se comportan como grandes caches, que almacenan el conjunto de trabajo de cada procesador.

3.5.1. Multiprocesadores

Así, se denomina multiprocesador a un computador que cuenta con dos o más CPUs. Lo cual permite al multiprocesador ejecutar simultáneamente varios hilos pertenecientes a uno o varios procesos. Los ordenadores multiprocesador presentan problemas de diseño que no se encuentran en ordenadores monoprocesador. Estos problemas derivan del hecho, de que dos programas pueden ejecutarse simultáneamente y, potencialmente, pueden interferirse entre sí. Concretamente, en lo que se refiere a las lecturas y escrituras en memoria.

Existen dos arquitecturas que resuelven estos problemas:

- La arquitectura NUMA, donde cada procesador tiene acceso y control exclusivo a solo una parte de la memoria. O bien, un procesador puede acceder a su propia memoria local de forma más rápida que a la memoria no local.

- La arquitectura SMP, donde todos los procesadores comparten toda la memoria. Esta última es la típica y tiene el problema de la coherencia de caché. Cada microprocesador cuenta con su propia memoria caché local. De manera que cuando un microprocesador escribe en una dirección de memoria, lo hace únicamente sobre su copia local en caché. Si otro tiene almacenada la misma dirección de memoria en su caché, resultará que trabaja con una copia obsoleta del dato almacenado. Para el correcto funcionamiento y evitar que los procesos intercedan entre sí se delega en el sistema operativo.

3.5.1.1. Cell

El procesador Cell es obra de IBM, Sony y Toshiba. Es un procesador capaz de alcanzar el alto rendimiento necesario para tareas matemáticas intensivas, tales como decodificar

o codificar secuencias MPEG, generar o transformar datos 3D.

Está formado por una arquitectura IBM Power y múltiples unidades de cálculo vectorial del tipo SIMD.

3.5.2. Trabajos relacionados

En el articulo [25] se describe una aproximación del algoritmo para la detección de colisiones enfocado a simuladores de vuelo. El algoritmo invoca previamente a una función secuencial previamente definida en paralelo, usando información espacial adquirida durante el proceso del árbol cuaternario, para tener separadas las invocaciones de las funciones lo más independientes posible. El método usado es aplicable a cualquier función, no solo al conflicto de la detección de colisiones, mientras que los argumentos estén organizados espacialmente. Los resultados presentan notables mejoras con respecto a una ejecución sin *threads* pero lo más notable es que la simulación permanece intacta, solo se cambia el mecanismo de invocación de una función. La clave consiste en llamar a la función secuencial desde varios *threads* dándoles trabajos independientes.

En el artículo [22] también se presenta un algoritmo paralelo, en concreto es denominado CCD (continuous collision detection) para modelos deformables, usando CPUs con multi-core. Su formulación mantiene el frente de colisiones, el cual tenga en cuenta tanto las colisiones entre objetos como las auto-colisiones mediante una reconstrucción adaptativa, mediante una distribución bien balanceada entre los múltiples cores usando, una descomposición de grano fino basada en "Front-Based". También lo combina con el conjunto de huérfanos con la finalidad de reducir el número de test elementales. En la práctica el algoritmo FBD puede considerarse una mejora en el cálculo de colisiones, usando múltiples cores. No obstante se podría mejorar el rendimiento, ya que no funciona demasiado bien cuando el número de procesadores es elevado.

3.6. Arquitecturas GPU

Antes de nada se debe tener siempre presente que a la hora de elegir entre GPU y CPU dependemos de la fisiología concreta del problema a resolver, ya que la CPU permite resolver un mayor espectro de problemas. Partiendo de este punto y de que las nuevas generaciones de hardware grafico, están un punto por delante incorporando más capacidad computacional que la CPU principal del PC. Este cambio junto a que sea altamente

paralelizables, ha disparado el interés en adaptar problemas costosos computacionalmente a tarjetas gráficas. Aunque tenga desventajas como que son unidades muy especializadas, y que el dominio sobre el cual trabajan es muy restringido, las ventajas que ofrecen, permiten que se desarrollen algoritmos de diferentes cometidos como bases de datos, procesado de imágenes y algebra lineal.

3.6.1. GPU frente a CPU

Tanto la CPU como la GPU son unidades de procesamiento, no obstante tienen orígenes, finalidades y capacidades muy diversas entre sí, sobretodo en sus comienzos.

- La CPU (Central Processing Unit), es el procesador, es decir, el corazón y el cerebro del computador. Normalmente es quien realiza las instrucciones y maneja los datos. Por regla general, cualquier computador necesita al menos una CPU como elemento indispensable.

- El término GPU (Graphics Processing Unit) se utiliza para unidades de procesamiento gráfico, las cuales nacieron con la finalidad de liberar de carga a la CPU en cuestión de operaciones gráficas. Los computadores no disponían de GPU hasta finales de los 70.

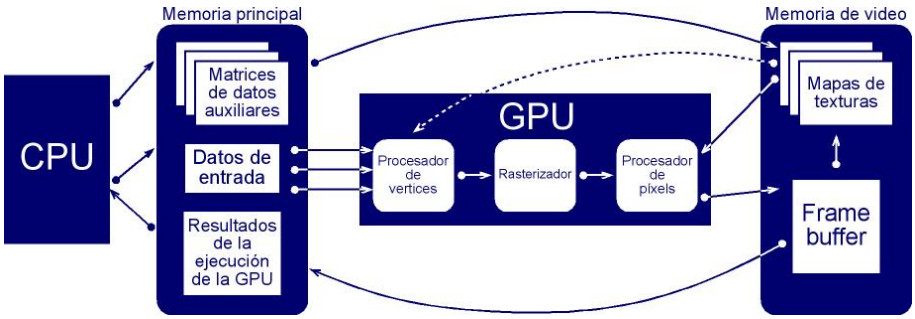


Figura 3.8: Relación de datos entre memorias cuando se ejecutan algoritmos de proposito general en GPU

3.6.2. Principales diferencias

Normalmente no es posible reemplazar la CPU por la GPU, a pesar de que las GPUs sean muy potentes hoy en día. No se puede olvidar que las GPU son procesadores altamente especializados, principalmente destinados a realizar una sola tarea, lo cual hace que sea más fácil su optimización. Concretando las GPUs están orientadas a la realización de operaciones en coma flotante las cuales son las más comunes en gráficos

tridimensionales.

Además, el procesamiento gráfico es muy costoso, y requiere de un paralelismo inherente debido a que se opera sobre miles de vértices y píxeles. Por tanto, la arquitectura de las GPUs es diferente a la de las CPUs. La CPU por regla general tiene una arquitectura Von Neumann, mientras que la GPU sigue el Modelo Circulante, sobre una arquitectura 3.2 SIMD (Single Instruction Multiple Data). De esta manera se dispone de un procesador con un ciclo de reloj menor pero con un alto grado de paralelismo, de manera que su potencia de cálculo es mucho mayor.

Como modelo de ejecución circulante, se entiende aquel que actúa sobre un flujo de datos de entrada y genera una salida sin guardar valores. Como ya se dijo anteriormente, se realizan operaciones tanto sobre vértices como sobre pixeles. Por lo que algunas unidades funcionales estarán dedicadas a operaciones de vértices y otras a las de los pixeles.

También tenemos que tener en cuenta como factor diferenciador, la memoria condiciona a la hora del almacenamiento de los resultados intermedios. La memoria de la tarjeta gráfica es de alta velocidad y permite realizar accesos con mayor frecuencia que si lo ejecutáramos en la CPU, el gran problema que ralentizará todo el proceso es el transporte de los datos de la memoria general a la de la GPU, ya que el bus es limitado. En cuanto a segmentación, ambos son procesadores segmentados sin embargo son muy diferentes. En la GPU no fluyen las instrucciones sino los datos, tiene operadores unitarios lo que previene de dependencias y favorece el paralelismo. También se dispone de un cauce con menos etapas ya que son procesadores con menos frecuencia, de manera que se fomenta procesar de manera vectorial y superescalar para así aprovechar al máximo el paralelismo de los datos.

	Computación	Mov. datos
CPU	10%	90%
GPU	30%	70%

Figura 3.9: Relación de consumos medios

3.6.3. Programación en dispositivos gráficos

En un inicio para programar la GPU se necesitaba realizar llamadas a la BIOS. Posteriormente se empezó ha utilizar el lenguaje ensamblador especifico de cada GPU. Más tarde llegaron las APIs especificas para tarjetas graficas que permiten un lenguaje más homogéneo y menos dependiente de la arquitectura. La evolución de estas APIs darían un lenguaje de alto nivel denominado Cg propietario de NVIDIA, y alrededor del 2003 se populariza el uso de los *shader*. Como *shader* se entiende un programa para interactuar con la GPU, sobre su cometido original era obtener diferentes estilos de sombreados. La programación en Cg provoca el nacimiento de los códigos acelerados en GPU, aunque la implementación sigue siendo un trabajo complejo y tedioso que requiere el conocimiento del cauce gráfico.

Con la extensión de las APIs llega el concepto GPGPU, o GPU de propósito general, en el que se pretende explotar las altas prestaciones de las arquitecturas gráficas para diferentes entornos. En 2004 se comienza experimentar, sobretodo en el entorno académico, sobre estas arquitecturas llegando a implementar simulaciones físicas, computación numérica e incluso gestión de base de datos. Finalmente en 2006 nació CUDA, aunque con ausencia de *floats* y manteniendo cierta complejidad de programación. En el momento en que se lanza la serie 8 de NVIDIA a finales del 2006 se resuelven este problema, ofreciendo eficiencia, versatilidad, precisión y sencillez en dispositivos avanzados.

Si bien es cierto que cuando se decide adaptar una implementación a la GPU, lo mejor sería tener la certeza de que se aprovecharán las ventajas de la misma y se ejecutara de manera eficiente. Normalmente no todo el código se ha de incluir en la tarjeta gráfica, ya que cierto trabajo es necesario realizarlo en la CPU. El principal problema se encuentra en la distinción de que parte del código se ejecuta más eficientemente en la CPU y en que partes se podría aprovechar la GPU, reduciendo el tiempo de ejecución. Por ejemplo, es bastante evidente que el acceso a memoria es un cuello de botella para la CPU, a pesar de las posibles optimizaciones de los compiladores. En este sentido, si se requiere un acceso consecutivo rápido, sería mejor utilizar una GPU ya que su memoria resuelve indirecciones de manera directa por hardware. En contraposición está el hecho de que aunque los operadores gráficos tengan una gran complejidad matemática, es complicado que la operación que se necesite en nuestro código este disponible, de manera que tendremos

que implementarla manualmente, lo que hace la realización de código aun más tediosa.

3.6.4. Enfoque al problema

Para la detección de colisiones existen dos caminos principales en la utilización de GPUs para acelerar los cálculos matemáticos y geométricos:

- Las técnicas de intersección basadas en espacio de imágenes se basa en realizar renders de los modelos, para después interpretar la imagen resultante. Son fáciles de interpretar y no necesitan estructuras de datos complejas. Los resultados son aproximados, ya que se hace un render de lo que está cerca del viewport[1], pero en general podemos decir que funcionan bastante bien.

- La detección de colisiones utilizando la GPU como co-procesador es un area menos desarrollada que la anterior y se subdivide en dos:

  – Utilización de GPGPU (General Purpose Computing on Graphics Processing Units) lo cual se podría traducir como computación de propósito general en unidades de procesamiento gráfico. La entrada debe ser empaquetada en vértices o texturas. Si tenemos la entrada dividida en texturas entonces en el acceso a memoria tendremos *Pixel Shader*, con el que se lee de varias posiciones arbitrarias pero se escribe en una determinada. Mientras que si tenemos como entrada vértices tendremos como acceso a memoria *Vertex Shader* que lee de una posición determinada y escribe en posiciones arbitrarias. El inconveniente de este método lo encontramos al leer desde la GPU, ya que produce una penalización debido a pérdidas en el paralelismo.

  – Utilización de CUDA (Computer United Device Architecture), que permite escribir en direcciones arbitrarias de memoria. Tiene una región de memoria rápida compartida para que las lecturas desde la GPU sean más rápidas, pero requiere una tarjeta de gráfica relativamente actual.

[1]"Viewport se considera al area rectangular correspondiente a la región que se visualiza dentro de un entorno grafico tridimensional completo."

3.6.5. Trabajos relacionados

3.6.5.1. Detección de colisión de mallas de modelos 3D en entornos de memoria distribuida

En [19] se expone solventar el problema de la detección de colisiones de objetos tridimensionales, explotando la alternativa del paralelismo, sobre una arquitectura de memoria distribuida, compuesta por diversas computadoras completas que podían ser heterogéneas tanto arquitectónicamente como en cuanto a software. En este proyecto se utilizaba MPI[2] para la comunicación entre los nodos, ya que estos se comunicaban mediante red, ya que no tienen memoria compartida.

El algoritmo contiene el uso de jerarquias de volumenes envolventes tipo Octree **??** con el fin de evitar la realización de tests de primitivas de manera inutil. También incorpora el llamado frente de colisiones, el cual es un buffer donde se almacenan los ultimos test sin colision, de manera que se supone que en el siguiente frame, tendremos que las colisiones se iran incrementando, de forma que los tests guardados en el frente son unos buenos candidatos a dar colisión en los siguientes frames.

Al tener este tipo de arquitectura, se dispone de tres tipos de maquinas con una funcionalidad completamente diferenciada:

- Master, es único y se encarga de cargar las mallas y enviar la parte necesaria a cada esclavo.
- Servidor, puede haber varios, y gestiona los esclavos, llevando la cuenta de quien esta trabajando en que, y cuando se vuelven ociosos se encargará de enviarles nuevo trabajo.
- Esclavo, el que ejecuta la detección de colisiones.

Al utilizar un cluster, su principal condicionamiento es la red por la cual se comunican, de manera que la mayor parte de las experimentaciones de este proyecto se centraron en buscar el equilibrio entre el número de nodos y el coste que supone el envio de los mensajes que son necesarios para que este nodo realice el trabajo.

[2]"MPI (Message Passing Interface) es un Interfaz estandarizado para aplicaciones paralelas basadas en paso de mensajes. es un software instalable que permite comunicar nodos distribuidos proporcionando versatilidad, escalabilidad y portabilidad."

3.6.5.2. Implementación de un algoritmo de detección de autocolisiones para objetos deformables

En [18] el principal objetivo consiste en la estimación de la posición donde se produce una autocolisión dentro de una animación en tiempo real. Se presenta un algoritmo implementado que utilizando jerarquias de volumenes envilventes basándose en los conos de normales, consigue la detección de las autocolisiones de manera más eficiente que mediante test de triangulos.

Capítulo 4

Representación de la animación de colisiones

Para representar una animación, tenemos que crear *frames* que definan la posición de los triángulos de los objetos de la misma a lo largo de ésta.

Con el fin de comprender cómo está realizada nuestra animación, procederemos a explicar qué es una *malla 3D* y los *frames* que constituyen a ésta.

4.1. Frame

Una vez comprendido el término de malla 3D, necesitamos determinar el movimiento de la misma para convertirse de un elemento inmóvil en una animación. Para ello explicaremos qué es un *frame* dentro de un escenario simulado y las analogías que éste tiene en una imagen en un vídeo.

4.1.1. Definición

Se denomina *frame*, a un fotograma, una imagen concreta dentro de una sucesión de imágenes que componen una animación. La continua sucesión de estos fotogramas provocan la sensación de movimiento.

La manera de hacer esto dentro de una malla 3D, es modificar la localización de los triángulos que forman nuestro modelo, con el fin de que al transitar de un *frame* a otro se cree el efecto de movimiento.

4.1.2. Representación de la colisión

Una vez definido lo que es un *frame* dentro de la animación vamos a exponer los pasos más generales realizados para la detección de colisiones.

Cuando un *frame* es cargado, se crean todas las instancias a la clase que almacena los triángulos, dentro de esta clase tenemos un indicador de si se ha producido colisión en dicho triángulo. Después de la ejecución del algoritmo de detección de colisiones, en caso de que se haya detectado alguna, se activa el indicador para a la hora de mostrarlo con ayuda de openGL mostremos este triángulo del color rojo. A continuación se presenta el algoritmo para para la detección de la colisión 4.1.

```
        ⋮
cargarSiguienteFrame()
        ⋮
calculaColision()
        ⋮
pintarEscena()
        ⋮
```

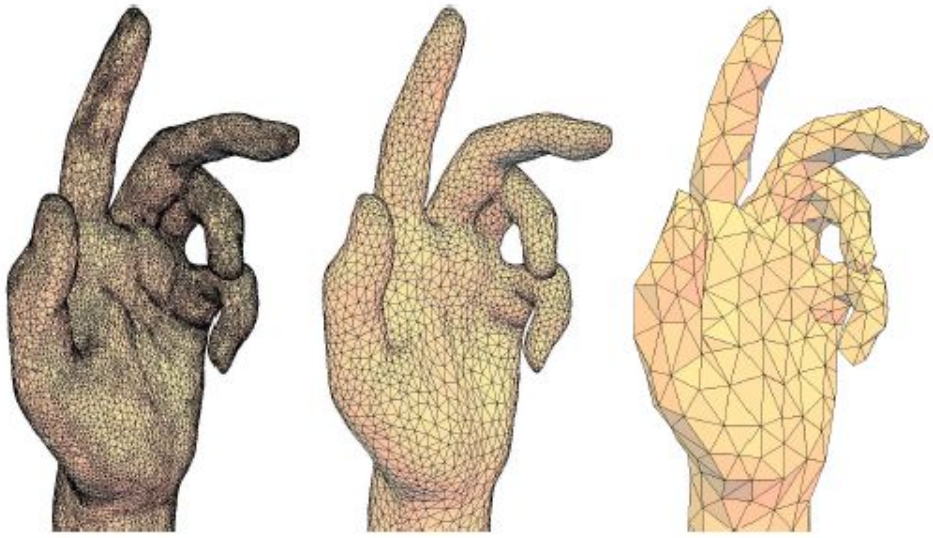
Algoritmo 4.1: Algoritmo general para la representación de los *frames* con o sin autocolisión

4.2. Malla 3D

Una malla poligonal por regla general es un conjunto de vértices, aristas y caras que definen la forma de un objeto en gráficos 3D y en el modelado de sólidos. Las caras normalmente son un conjunto de triángulos, cuadriláteros o polígonos convexos simples, ya que éstos facilitan la representación, pero también pueden estar compuestos por polígonos cóncavos o por polígonos con agujeros.

También existen las mallas volumétricas que son representan explícitamente tanto la superficie como en el volumen de la estructura.

Nuestras representaciones de mallas 3D están realizadas mediante triángulos, tal y como se muestra en la Figura 4.1.



(a) 25,000 vertices. (b) 5,000 vertices. (c) 500 vertices.

Figura 4.1: Representación de una malla 3D de triángulos. Figura extraida de [5]

4.2.1. Representación

Para representar nuestros objetos solo almacenamos vértices y caras para definir nuestras mallas 3D.

En nuestro implementación, cada una de estas listas que a continuación se exponen están almacenadas en un fichero plano de texto. La primera lista contiene los vértices, mediante tres dimensiones dentro del eje de coordenadas, y en segundo lugar una lista para almacenar las caras, las cuales referencian a los vértices que las componen.

4.3. OpenGL

La biblioteca gráfica OpenGL[1] (Open Graphics Library) nos da la oportunidad de poder dibujar escenas tridimensionales complejas partiendo de primitivas geométricas simples con gran interactividad y realismo. OpenGL ofrece una imagen sintética de la escena 3D, desde una cámara virtual trabajando con el procesador gráfico y no con la CPU.

OpenGL permite:

[1] http://www.opengl.org/

1. Modelar figuras a partir de las primitivas.

2. Situar los objetos en el espacio y cambiar el punto de vista desde el que enfocarlos.

3. Calcular el color, textura e iluminación de los objetos.

4. Convertir las primitivas y los datos de color en objetos dentro de imagen virtual 3D.

OpenGL desarrolla la eliminación de partes de objetos que no son visibles para el usuario por cuestiones del entorno.

Parte II

ESTUDIO Y SOLUCIÓN DEL PROBLEMA

Capítulo 5

Descripción Informática

El objetivo de este proyecto es diseñar un algoritmo de detección de colisiones utilizando GPUs, las cuales son cada vez más máquinas multi-core abundantes. Las ventajas que ofrece una solución así es alto rendimiento y escalabilidad sobretodo si se implementa una versión para múltiples tarjetas.

Hoy en día, casi cualquier computador tiene una tarjeta gráfica potente. Actualmente sin lugar a duda el líder del mercado en este sector es NVIDIA. Como se puede ver en la figura 5.1 obtenida de [14] los tiempos de ejecución completa y dentro de la GPU para la misma prueba en CUDA y OpenCL son inferiores en CUDA. La figura 5.2 se muestra el tiempo de las trasferencias de datos y el tiempo dentro del kernel, los cuales siempre son inferiores en CUDA que en OpenCL. Debido a esto se ha elegido CUDA como API. Además, para desarrollo del mismo en diciembre del 2007 existían 40 millones de dispositivos capaces de utilizar CUDA, por tanto está asegurado el soporte futuro de esta API.

Qubits	GPU Operations Time				End-To-End Running Time			
	CUDA		OpenCL		CUDA		OpenCL	
	value	stdev	value	stdev	value	stdev	value	stdev
8	1.97	0.030	2.24	0.006	2.94	0.007	4.28	0.164
16	3.87	0.006	4.75	0.012	5.39	0.008	7.45	0.023
32	7.71	0.007	9.05	0.012	10.16	0.009	12.84	0.006
48	13.75	0.015	19.89	0.010	17.75	0.013	26.69	0.016
72	26.04	0.034	42.32	0.085	32.77	0.025	54.85	0.103
96	61.32	0.065	72.29	0.062	76.24	0.033	92.97	0.064
128	101.07	0.523	113.95	0.758	123.54	1.091	142.92	1.080

Figura 5.1: Comparativa de tiempos de ejecución dentro de la GPU y con el programa completo entre CUDA y OpenCL.

Qubits	Kernel Running Time				Data Transfer Time			
	CUDA		OpenCL		CUDA		OpenCL	
	value	stdev	value	stdev	value	stdev	value	stdev
8	1.96	0.027	2.23	0.004	0.009	0.007	0.011	0.007
16	3.85	0.006	4.73	0.013	0.015	0.001	0.023	0.008
32	7.65	0.007	9.01	0.012	0.025	0.010	0.039	0.010
48	13.68	0.015	19.80	0.007	0.061	0.010	0.086	0.008
72	25.94	0.036	42.17	0.085	0.106	0.006	0.146	0.010
96	61.10	0.065	71.99	0.055	0.215	0.009	0.294	0.011
128	100.76	0.527	113.54	0.761	0.306	0.010	0.417	0.007

Figura 5.2: Comparativa de tiempos de ejecución dentro del kernel y tiempo de transferencia de datos entre CUDA y OpenCL.

El *grid de bloques de threads*, ver figura 5.3, es la abstracción de la arquitectura básica en la que CUDA organiza el paralelismo[1]. El principio básico se centra en que el *grid computacional* es un conjunto de *bloques de threads*. El chip de la GPU se divide en una colección de *multiprocessors (MPs)* capaces de ejecutar más de un bloque. A su vez, cada multiprocesador se divide en una serie de *stream processors (SPs)*, los cuales son capaces de manejar uno o más threads de un bloque. La arquitectura se considera una arquitectura SIMD, arquitectura explicada en el apartado 3.2 ya que es como realmente se puede explotar el paralelismo.

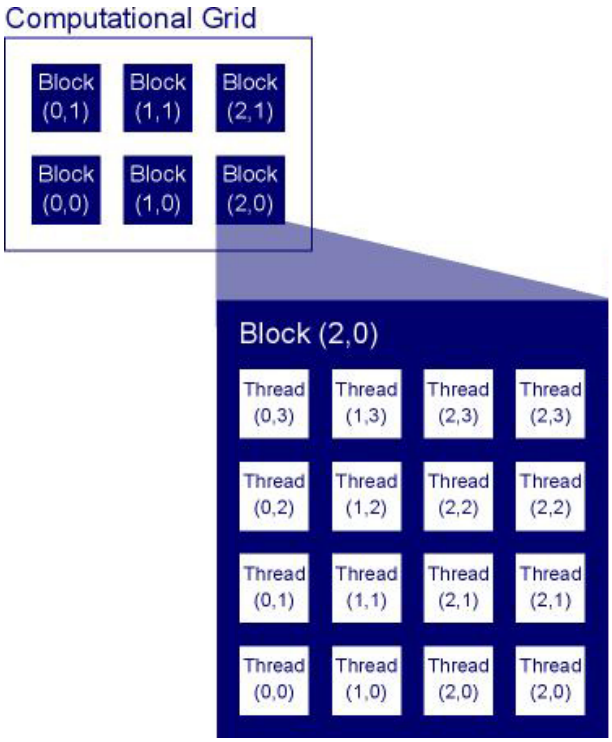


Figura 5.3: Grid de bloques de threads

Para ayudar a entender este concepto mejor se necesita de las siguientes definiciones [2]:

- Kernel, consiste en el elemento unitario consistente en el conjunto de codigo referente a unos datos, capaz de ser ejecutado por si solo.
- Thread, se considera a cualquier ejecución de un kernel con un índice dado. Cada kernel usa este índice para acceder a los elementos del array de datos al cual tiene acceso todos los threads que están ejecutando.
- Bloque, se refiere al conjunto de threads, que ejecutan de manera concurrente o no, el mismo código pero con diferentes datos como se puede ver en la figura 5.4.

  Cada bloque es ejecutado por un único multiprocesador, nunca se dividen en varios.
- Grid, lo cual incluye un grupo de bloques. No existen herramientas de sincronización entre los bloques. Un grid es manejado por solo un chip de GPU.

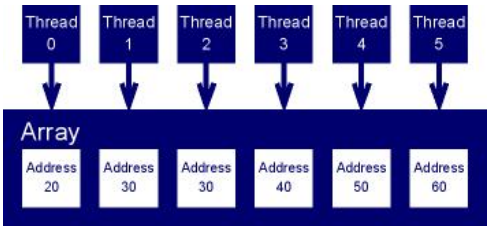


Figura 5.4: Distribución de memoria entre threads

El balanceo de la carga, es un problema tradicionalmente importante a la hora de paralelizar un algoritmo, pero gracias a esta API y a la manera de programar que nos brinda, es transparente al usuario ya que este solo debe elegir el número de threads que quiere utilizar, y las secciones de memoria a copiar a la memoria de la grafica. Es el software de CUDA el que pone a cada thread a trabajar sobre una zona de memoria determinada.

5.1. Algoritmo de detección de colisiones

A continuación, mostraremos de forma breve los elementos que hemos utilizado para esta implementación de la detección de colisiones y auto-colisiones.

5.1.1. Algoritmo de detección de colisiones

Para la detección de colisiones se han utilizado los siguientes procedimientos:

- Jerarquía de volúmenes envolventes.

- Test de triángulos.

5.1.2. Algoritmo de detección de auto-colisiones

Para la detección de auto-colisiones se han utilizado los siguientes procedimientos:

- Jerarquía de volúmenes envolventes.
- Conos de normales.
- Test de triángulos.

5.1.3. Jerarquía de volúmenes envolventes

En esta implementación se ha elegido una división utilizando un árbol binario 2.2.1.2 por su fácil comprensión y utilización.

Los pasos a seguir para su utilización son los siguientes:

1. Se recubre por completo el objeto con un poliedro tridimensional, que se ajuste.
2. Se localiza el centro exacto del cubo.
3. Utilizando el punto central del eje x y se divide el poliedro tridimensional[1]. en 2 partes iguales.
4. Se continuará dividiendo de esta manera se haran divisiones hasta que lleguemos a que alguno de los hijos este vacío, a partir de donde se realizaran el test de triangulos si el volumen colisiona con otro y no está vacio.

Para la detección de colisiones, primero se envuelven las dos figuras sobre las que trabajamos, con dos cubos lo más ajustados posibles a los objetos y se realiza un test para ver si están lo suficientemente cerca para colisionar. Si se solapan en algún punto (ver figura 5.6), tenemos que seguir trabajando. En segundo lugar se deben hacer todas las parejas posibles entre los sub-poliedros, de los dos poliedros originales, y realizar un test en el que si el espacio de uno esta invadido por el de otro, podría haber colisión y hay que seguir explorando esa rama.

[1]Hablamos de poliedros tridimensionales de seis caras ya que los volúmenes envolventes utilizados no tienen por qué ser regulares, cubos, si no que son variables y su misión es adaptarse lo más posible al objeto tridimensional a analizar, para conseguir más efectividad.

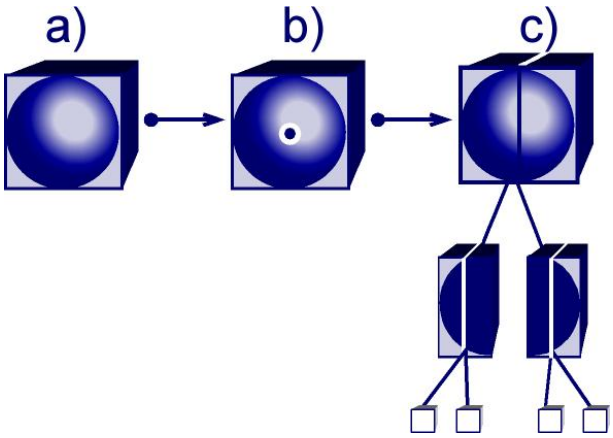


Figura 5.5: Desglose de un *árbol binario*

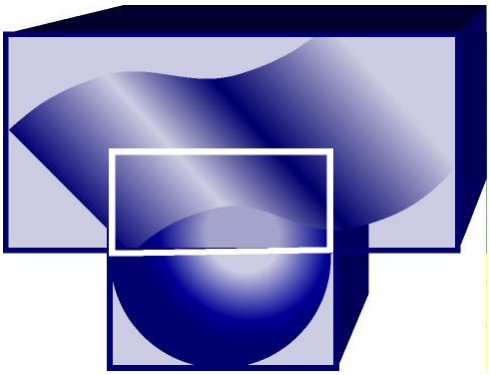


Figura 5.6: Colisión de cajas envolventes

5.1.3.1. Optimización

Bien es cierto que como base se utiliza un octree, no obstante se han realizado un par de modificaciones para optimizarlo. Estas optimizaciones consisten en:

- No siempre tenemos que trabajar con poliedros del mismo tamaño, es decir al dividir un poliedro, los sub-poliedros resultantes pueden ser diferentes ya que se busca que se ajusten lo máximo posible al objeto. Esto provoca que haya menos espacio inutil cubierto por el poliedro, pudiendo provocar falsos positivos.

- Si al subdividir un poliedro, entre sus hijos encontramos alguno vacío se elimina, de manera que no siempre se tendrán ocho hijos por cada nodo no hoja si no que es algo variable dependiendo del modelo. Esto provoca ahorro, tanto de espacio en memoria guardando un poliedro inutil como en tiempo de ejecución explorando ramás nulas..

5.1.4. Detección de auto-colisiones

Como ya se ha mencionado antes para la detección de auto-colisiones también se usan jerarquías de volúmenes envolventes, en este caso se dividirá, cada objeto para analizarlos consigo mismo, realizando todas las conbinaciones posibles entre los sub-poliedros, eliminando el test que incluye una zona consigo misma.

5.1.4.1. Test del producto cartesiano de vectores normales

Para determinar que una superficie esta libre de auto-colisiones se utiliza un cono de normales, el cual consiste en que el producto cartesiano sea positivo con cada uno de los vectores normales de todas las caras. Como se puede ver en la figura 5.7 en el ejemplo b), se obtiene un producto cartesiano negativo, por lo que se consideraría que estamos ante una aut-colisión, mientras que en el ejemplo a), el producto cartesiano es positivo, por lo que no se consideraria que tiene colisión.

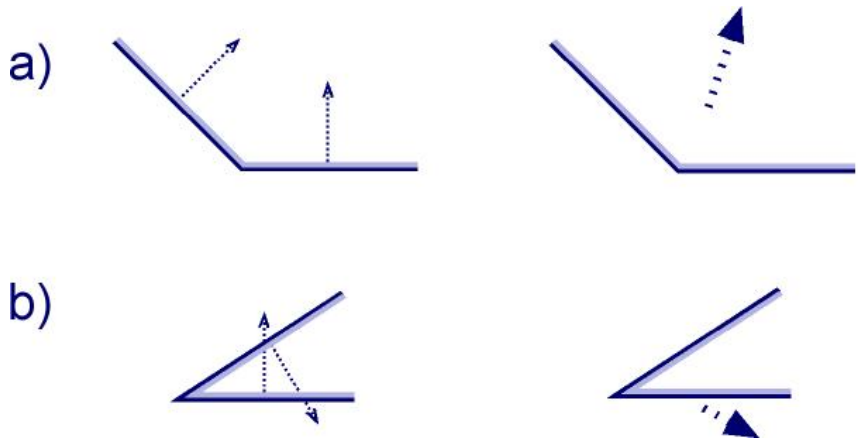


Figura 5.7: Test de normales, entre primitivas.

No obstante esto no siempre es real, de hecho solo es real si los test son sobre las primitivas que forman la malla del objeto tridimensional, ya que de no ser así podríamos estar ante un resultado falso, como se puede ver en la figura 5.8. En esta figura se puede ver como el test de auto-colisión basado en el producto cartesiano positivo de un conjunto de vectores normales frente a otro grupo de vectores normales, puede dar tanto falsos positivos como en a) como falsos negativos en b). Por esta razón este test se tiene que hacer desde la primitiva, de manera que se utiliza el *cono de normales*.

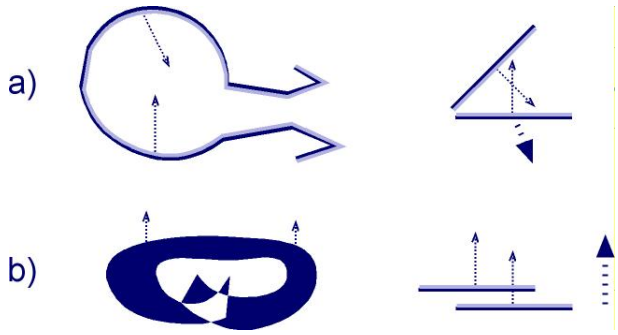


Figura 5.8: Test de normales erróneos

5.1.4.2. Cono de normales

Un cono de normales, también llamado DNC (Discrete Normal Cones) es una estructura de datos para el almacenamiento de regiones angulares [18] de los vectores normales que están dentro. No obstante es irrelevante el conocimiento de las coordenadas en las cuales se sitúa este vector normal, pues solo es relevante su existencia.

El cono de normales comienza por las hojas de la jerarquía, de las primitivas de las mallas, y a partir de aquí se construyen los niveles superiores. Para la construcción se necesita un array de 14 bits, en el que cada bit representa el producto cartesiano entre un vector fijo y el vector normal del triangulo. Si el producto cartesiano da positivo se rellena esa posición del array con unos mientras que si da negativo se rellena con ceros. De manera que al representar gráficamente el cono de normales, solo serán visibles los vectores fijos que en el array tuvieran un 1.

Los vectores fijos existentes son los respectivos a los ejes tanto en versión positiva como negativa, es decir (x, -x, y, -y, z, -z), y los vectores que apuntan desde el centro del cubo hasta sus vértices, como se puede ver en las segunda representación de la figura 5.9.

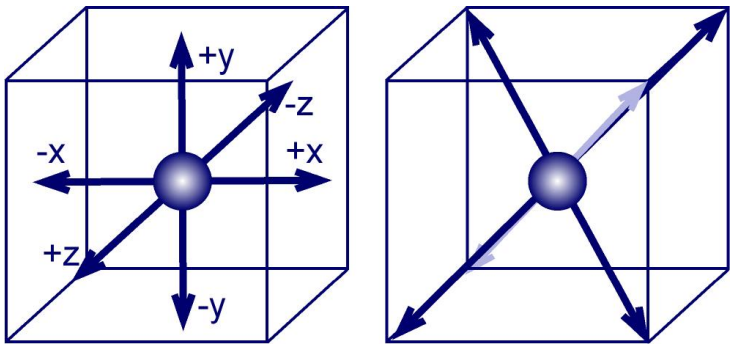


Figura 5.9: Representación de los vectores fijos

Para la construcción del cono de normales, se irán calculando las intersecciones de

los hijos de manera que calculando la intersección de los array obtenemos el array del padre, y así sucesivamente hasta llegar al nodo raíz, es decir seguiremos una construcción bottom-up, ver sección 2.2.2.2.

Una vez se tiene el árbol se puede considerar que hay riesgo de auto-colisión si el nodo de un cono de normales contiene todo ceros en su array, como se puede ver en la figura 5.10. El significado de esto es que no hay normales comunes, lo que supone un riesgo de colisión por lo que habrá que realizar un test más exacto, es decir test de triángulos, sobre la zona que cubría este nodo.

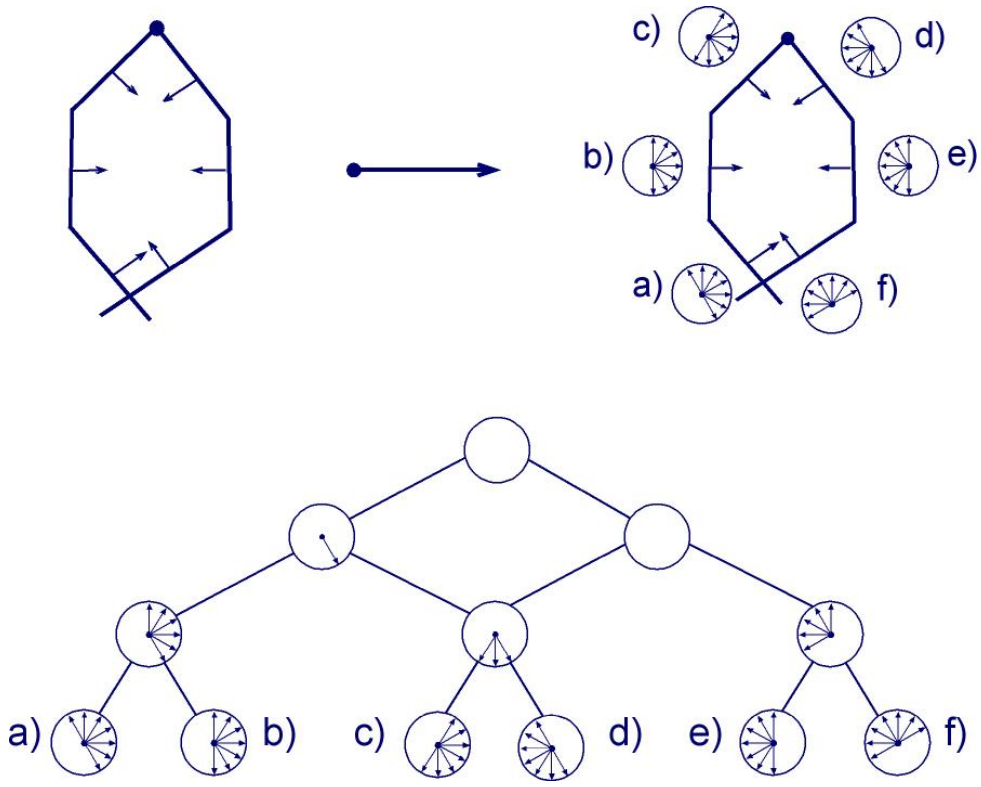


Figura 5.10: Desarrollo del árbol del cono de normales

En este proceso es necesario saber si dos nodos son adyacentes o no, ya que solo si lo son podemos realizar el test. Para saberlo de manera certera se utilizan los dos métodos expuestos a continuación:

- Recuperando todos los nodos hojas del árbol de cajas envolventes, y se comprueba si los vértices de los volúmenes se tocan entre sí, en caso afirmativo estaremos ante una adyacencia.

- Por otro lado cada nodo dispone de una lista de sus adyacentes de manera que un nodo es adyacente a otro si este tiene en su lista de adyacentes al otro. Para evitar

la sobrecarga de memoria en modelos complejos, solo los modelos de dentro de los volúmenes envolventes con colisión y candidatos a su posterior análisis serán los que tengan las listas de adyacentes.

5.1.5. Test de triángulos

Se considera que dos triángulos tienen colisión si al menos una de las aristas interseca con la de otro triángulo como se muestra en la figura 5.11

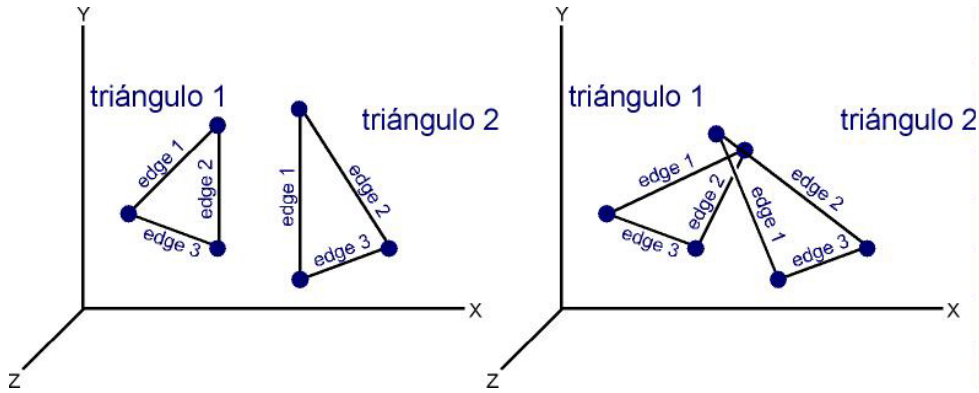


Figura 5.11: Test de triángulos

Se utilizarán las aristas y se comprobarán si estas intersecan entre si, antes de nada se comprueba que el test no sea una auto-colisión, ya que para este tipo se realizará un test diferente (ver sección 5.1.2), si no son triángulos paralelos. Después si las aristas intersecan se comprobará si están dentro de la región de los triángulos o la intersección se encontraría en las prolongaciones de las aristas, más allá de los triángulos a analizar.

```
if triangulosDiferentes (t1, t2) then
  if noSonParalelos(arista, triangulo) then
    if intersecan(edgev1x, edgev1y, edgev1z, edgev2x, edgev2y, edgev2z) then
      return test.collision = true;
    end if
  end if
end if
```

Algoritmo 5.1: Algoritmo básico de detección de colisión entre triángulos

5.1.6. Jerarquía para la representación del espacio

Dentro de las posibilidades que se presentan dentro de las posibles representaciones del espacio y las jerarquías para almacenar el modelo 3D, se ha optado por utilizar un

volumen envolvente.

Sobre a las táctica de representación que ha sido usada después de analizar algunas de las investigaciones como representación a base de volúmenes envolventes a base de esferas envolventes [12], K-DOPs – Discrete Orientantion polytopes Polígonos orientados discretos [15], OBB – Oriented Bounding Boxes Cajas envolventes orientadas [9], AABB – Axis Aligned Bounding Boxes, Cajas envolventes alineadas con el eje de coordenadas natural [23] y Hierarchical Spherical Distance Fields [8].

En conjunto el coste de la detección de colisiones de dos objetos geométricos, que son representados con jerarquía de volúmenes envolventes puede ser formulado por la siguiente ecuación 5.1:

$$Ttotal = Nb * Cb + Np * Cp \tag{5.1}$$

Donde:

- Ttotal: tiempo total de cálculo de las intersecciones entre dos modelos.
- Nb: número de los pares de volúmenes envolventes que son testeados buscando una intersección.
- Cb: el coste de un test de intersección entre pares de volúmenes envolventes.
- Np: número de pares de polígonos primitivos que son testeados para buscar una intersección.
- Cp: coste de un test de intersección entre pares de polígonos primitivos.

Los parámetros que son afectados por los tipos de los volúmenes envolventes son Nb, Np y Cb. El ajuste de los volúmenes envolventes, ver figura 2.3, como el que se tiene con un algoritmo tipo OBB, produciría un valor bajo Nb y Np, pero producirá un Cb relativamente alto. En cualquier caso AABB provocará más tests a realizar, pero el valor de Cb suele ser más bajo.

El potencial de los algoritmos secuenciales es sin lugar a dudas limitado, y el paralelismo se vuelve una mejora esencial si el algoritmo tiene que ejecutar rápidamente. Algunas investigaciones han trabajado para poner en práctica una versión paralela del

problema de detección de colisiones. En este caso se estudió las diferentes aproximaciones de estos trabajos para comprender cuál es la dirección que se debe tomar.

Sobre anteriores expuestos, se ha seleccionado la táctica AABB. Esta táctica representa los volúmenes envolventes con el valor mínimo y máximo del modelo por cada uno de sus ejes, horizontal y vertical. De este modo, se crea cada volumen envolvente.

La construcción de un volumen envolvente con la técnica AABB es considerablemente más rápida [24]. El algoritmo procesa a través de cada uno de los elementos básicos contenidos en el volumen envolvente y proyecta por cada uno de los ejes.

Los volúmenes envolventes son usados de dos maneras. La primera manera es reducir el número de comprobaciones de intersección de objetos, tanto estáticos como dinámicos, en el espacio dado. Para n objetos, entonces tendremos [n/2] objetos, que tienen una alta probabilidad de tener una intersección. Este número es obviamente muy grande y depende de la resolución de los modelos, definiendo el número de test de intersección.

La segunda manera consiste en reducir el número de test de intersecciones por cada par de polígonos primitivos, triángulos en nuestro caso.

Esto no consiste en detectar las colisiones entre dos objetos complejos, ya que se han realizado un test de intersección entre cajas continentes. La estructura espacial de datos es creada en pre-proceso y permanece estática durante todo el proceso. Así se envuelve dentro de un volumen una parte de un objeto y se computa por cada uno de los subconjuntos, 8 en nuestro caso. Cada volumen envolvente se va ajustando para que cuando las intersecciones de los subconjuntos sean testeadas, esos subconjuntos sean envueltos de forma ajustada.

Como se ha visto, al igual que [7], nosotros también elegimos AABB para representar el modelo geométrico para la detección de colisiones. No construiremos un árbol de grandes dimensiones que contenga hojas en las cuales solo encontremos un volumen envolvente con un solo polígono. En estas hojas, el tiempo de ejecución es constante ya que en la mayoría el número de polígonos que contienen es similar.

El mayor inconveniente del uso de esta técnica es el tiempo de ejecución que el algoritmo necesita de la geometría, para construir el árbol de representación, por lo tanto

las actualizaciones que el algoritmo necesita son numerosas cuando se usan modelos en movimiento.

El test de colisiones entre dos modelos geométricos está realizado de manera recursiva, por cada dos nodos que son obtenidos de cada uno de los árboles, empezando por los padres. Del mismo modo, para la detección de auto-colisiones se explorará cada árbol consigo mismo. A continuación se presentan los casos necesarios para el manejo de los test de intersección:

- Si la representación de los volúmenes envolventes de los nodos tienen una intersección se retornara verdadero y se seguirá analizando, dividiendo estos volúmenes.
- En todos los nodos hoja, las primitivas geométricas contenidas en ella, serán comprobadas para ver si tienen intersección, pero no en la CPU este test se reserva para ser enviado en masa a la GPU. Este test se reserva para ser enviado en masa a la GPU.
- Si uno de los nodos es una hoja y otro no, y se considera la posibilidad de colisión, se desglosará el nodo hasta sus hojas y se guardara un test de cada combinación de nodos hoja.
- Si los dos nodos no son hojas, el nodo al que le representa una caja envolvente menor, será testeada buscando una intersección con cada uno de los hijos del otro nodo (todos con todos).

La metodología que se ha seguido para la construcción del árbol es Top-down ya que éste es el método más usado, y une rapidez y sencillez a la hora de la implementación. Aunque esta técnica no genere el árbol óptimo, compensa. Entrando en detalles este método parte de un gran un volumen envolvente que contenga a todos los objetos de la escena, dividiéndose en volúmenes más pequeños, a la vez que se divide el árbol, en ocho partes ya que trabajamos sobre cubos o hexaedros tridimensionales ajustados al objeto.

La división en dos partes un árboles binarios ha sido elegida ya que es la más básica y rápida, para dividir un poliedro de seis caras, por un plano vertical que divide por la mitad del eje de abcisas. Es evidente que es un cálculo muy sencillo computacionalmente hablando, frente a la complicada división de tres planos a la que nos enfrentaríamos si utilizáramos un octree.

Hablamos de poliedros de seis caras ya que los volúmenes envolventes utilizados no tienen por qué ser regulares, cubos, si no que son variables y su misión es adaptarse lo más posible al objeto tridimensional a analizar, para conseguir más efectividad.

5.2. Programación en CUDA

El algoritmo funciona con todos los "frames" de una secuencia, debido a esto el algoritmo dispone de un bucle, del que solo se sale cuando se haya terminado la secuencia y su análisis. Dentro del bucle la primera tarea a realizar en acotar el problema, es decir por cada "frame" se evaluara una jerarquía de volúmenes envolventes para delimitar el problema, y se almacenará finalmente en una tabla Hash. Esta tabla contendrá todos los test candidatos a tener una colisión y/o auto-colisión. Una vez preparada la tabla Hash con todos los test se introduce en la memoria de la GPU junto con todo el código necesario para su ejecución.

Desde un inicio, se tenia claro que sólo se enviarían los tests de triángulos candidatos a la GPU y no los tests de los volúmenes envolventes. El problema erradicó en la estructura para almacenarlos. Inicialmente se pensó en una lista, por ser la manera más simple y bastante similar a la utilizada en [19]. No obstante era poco eficiente ya que obligaba a realizar una búsqueda iterativa de cada test, ya que se desconocía totalmente la posición. Más tarde se descubriría que la tabla hash tampoco era la estructura más adecuada, ya que utilizaba memoria dinámica, de manera que teníamos una colección de punteros relacionados entre si que apuntaban a zonas de memoria localizadas. Por lo tanto era prácticamente imposible concentrar toda la tabla hash en una zona de memoria limitada para copiarla en la GPU.

A la GPU se le envía una estructura que contiene una variable para guardar el tamaño del array completo, otra variable para determinar hasta donde esta lleno el array, ya que al depender de los tests previos de volúmenes envolventes hemos podido ahorrar muchos tests, y por último el array de tests.

La estructura de los test es la siguiente:

- La estructura de test consiste en una variable llamada *collision*, que si es 0 significará que no hay colisión y un 1 si sí que la hay.

- Por cada uno de los triángulos hay un identificador, las coordenadas tanto de sus vértices, y las coordenadas de sus aristas.

La inclusión del código dentro del espacio de memoria que se envía a la GPU es un requisito indispensable para la programación de cualquier algoritmo en CUDA, ya que no admite la importación de librerías externas, así que la más minima de las funciones auxiliares, tiene que ser implementada de cero.

La zona de memoria copiada, contendrá:

- Código necesario para la realización del test de triángulos.
- Estructura que contiene el array de los tests candidatos a colisión, es decir solo se emparejan triángulos de una malla con los de la otra.
- Estructura que contiene el array de los tests candidatos a auto-colisiones, es decir solo se emparejan triángulos de cada una de las mallas consigo mismos.

Finalmente cuando la GPU termina la evaluación de colisiones devuelve la estructura, de manera que de vuelta a una ejecución normal se pintara el resultado, recorriendo las listas devueltas para comprobar si nos encontramos ante una colisión, en cuyo caso se pintara de rojo, o frente a una auto-colisión que se pintará de amarillo.

5.2.1. Detalles de la programación en CUDA

Nociones básicas utilizadas, han sido enfocadas para el uso general de CUDA, desde un nivel inicial hasta el necesario para conseguir algo productivo. Los principales cambios a destacar frente a la programación imperativa normal, consisten en como ya se explicará con posterioridad, la limitación sobre la importación de librerías externas. CUDA tiene una manera peculiar de introducir los datos, ya que necesita que se cree un kernel que incluye tanto el código a ejecutar como los datos sobre los que ejecutar. En la zona de memoria de los datos dentro de la GPU, a cada thread se le asigna una porción definida a partir de su índice. El valor del índice es asignado según la posición que ocupa ese thread dentro de un bloque (ver figura 5.3). Tipos de funciones:

- Precediendo a la función de la palabra reservada *__global__* se marcan las funciones que se desean llamar desde el código de la CPU.
- Mientras que el resto de funciones deben de ir precedidas de *__device__* para indicar que es una función ejecutable por los threads de la GPU.

A parte de las evidentes funciones para la copia de la memoria de CPU a GPU y viceversa, se han implementado diversas funciones necesarias para la gestión de la memoria, más concretamente funciones encargadas de definir que parte de memoria pertenece o es sobre la que trabaja cada thread.

5.2.2. Versión multi-GPU

Para la versión de multi-GPU, únicamente se necesita conocer el número exacto de GPUs que vamos a tener para ello se utiliza la función *cudaGetDeviceCount(&GPU_N)* , que modifica la variable devolviendo de cuantas GPUs se dispone, el trabajo se distribuye de la misma manera que para solo una GPU, asignando un trozo de memoria sobre la que trabajar a cada thread.

Al tener más de una GPU, se necesita replicar el código en ambas, y asignar una parte de datos a cada una, para no realizar dos veces los mismos test, de manera que es completamente necesario la creación de tantos *Kernels* como GPUs sobre las que se vaya a ejecutar.

Al utilizar un *Kernel* con datos específicos en cada una de las GPUs se consigue de las mayores paralelizaciones posibles, ya que con gran cantidad de threads, de n tarjetas gráficas diferentes, trabajando al unísono sobre diferentes datos dentro del gran array de test.

5.2.3. Adversidades

Durante la realización del proyecto se dieron unas cuantas incidencias de las que aprender:

- En primera instancia, CUDA no permite la importación de librerías externas obligando a incluir la implementación dentro de la gráfica.
- La capacidad de las actualizaciones del sistema operativo, para generar fallos de segmentación inexistentes.
- El problema de utilizar estructuras de memoria dinámica, y como agruparlas para poder copiar una zona de memoria concreta y localizada.
- Problemas de espacio en memoria, en la parte multi-GPU, al destruir y volver a crear un buffer que contuviese los test de cada *frame* en concreto.

Capítulo 6

Resultados Experimentales

En este capitulo se presentan los resultados de las pruebas realizadas con el algoritmo anteriormente presentado utilizando GPUs. Los resultados finales se presentan a modo de gráficas para poder realizar análisis más rápidamente. Antes de comenzar con los datos se van a definir las condiciones en las que se han realizado las pruebas.

En este proyecto la implementación se ha realizado en CUDA, por lo tanto necesitamos tener tarjetas gráficas NVIDIA, ya que CUDA es propietario de NVIDIA y solo funcionará en su arquitectura. La máquina utilizada para obtener los resultados es una máquina muy potente que dispone de dos tarjetas gráficas. Más concretamente dispone de dos tarjetas gráficas NVIDIA GTX 580, una CPU con d2 procesadores Intel R Xeon R E5645 2.4 GHz (6 cores cada procesador, es decir 12 cores con hiperthreading) y 48 Gb de RAM.

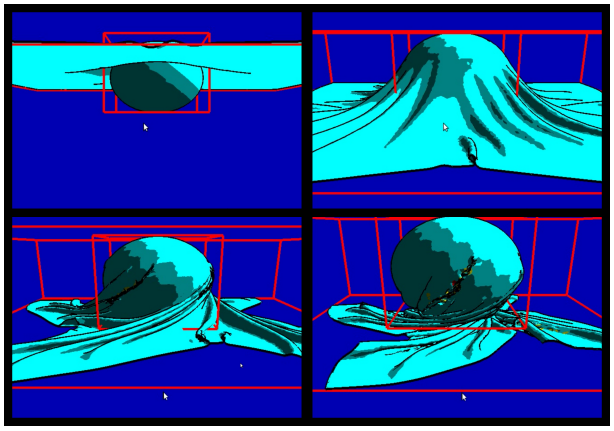


Figura 6.1: Secuencia de la malla y la bola.

La secuencia que se utiliza en las pruebas consiste en una manta y una bola que interactúan como se muestra en la Figura 6. La manta cae sobre la bola que gira sobre si misma, de manera que tendremos tanto colisiones entre los dos objetos como auto-

colisiones en la manta cuando esta se enrede consigo misma generando una multitud de pliegues. Las pruebas se han realizado para la secuencia completa, para cada uno de sus *frames*, y con la perspectiva de que se dibujen las colisiones, marcando las colisiones en rojo y las auto-colisiones en amarillo verdoso (ver Figura 6.2), en cada *frame* a medida que vayan transcurriendo. Con el fin de facilitar la ejecución remota se ha omitido la parte gráfica así que los tiempos son recogidos sobre la ejecución de la detección de colisiones pero no del marcado de las mismas.

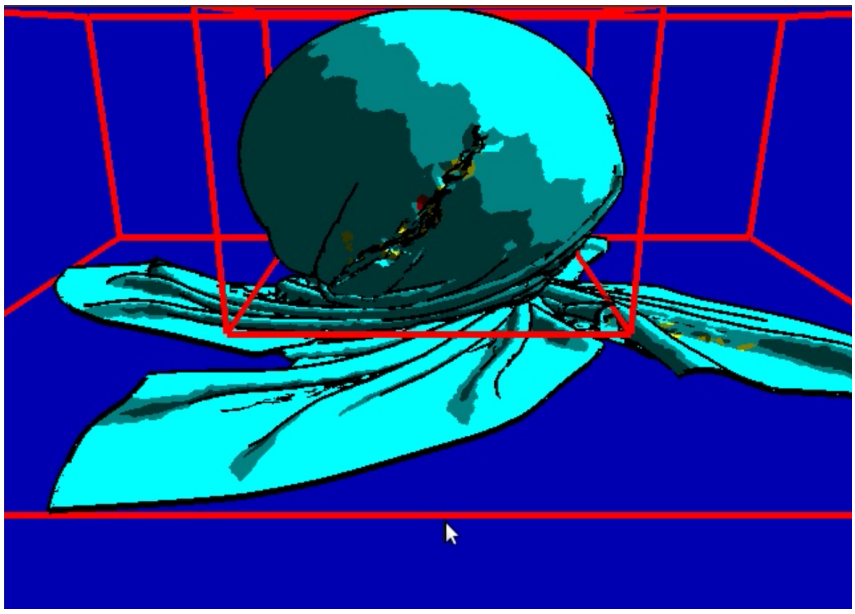


Figura 6.2: Secuencia de la manta y la bola.

El algoritmo se plantea de la siguiente manera, la CPU realiza la carga de las mallas, y en cada *frame* realiza la jerarquía de volúmenes envolventes, obteniendo los test de triángulos candidatos a colisionar, tanto para colisiones como para las auto-colisiones. Los tests se guardan en el array de test, que se envía a la GPU para que ejecute el test. Cada test se compone de los datos necesarios para la ejecución de la detección de colisiones entre primitivas, es decir los datos correspondientes a dos triángulos. Una vez la GPU tenga los resultados se los devolverá a la CPU, que pintará los resultados, si estuviera habilitada la visualización, y continuará con el siguiente *frame* hasta acabar la secuencia.

CUDA permite elegir el número de threads que tendremos en cada bloque, lo cual condicionará el número de bloques final, ya que threads tendremos tantos como tests de triángulos. Por tanto se utiliza este parámetro para investigar cual será el valor que

mejores tiempos obtiene. Los valores que se prueban son de 32, 64, 128 y 256 threads.

Los resultados que se muestran a continuación, se encuentran condicionados por el tamaño del array, el cual restringe el número de test a ejecutar por la GPU. En las pruebas realizadas se han tomado medidas en las que se incrementa en intervalos de 250.000 del 0 al 1.500.000, para las pruebas de single y de 250.000 a 1.500.000 para las pruebas de dual GPU, ya que en estas últimas no permite la ejecución de un array vacío ya que no se puede dividir. Para obtener las gráficas con el eje de ordenadas desde el origen, se utiliza un parámetro de control presente en todas las gráficas. El número de test nunca sobrepasará el millón y medio ya que las mallas sobre las que trabajamos en el peor frame generar un 1.470.000 tests aproximadamente.

6.1. Resultados de la versión sobre una GPU

En este apartado se han realizado las siguientes pruebas de la versión single-GPU, es decir utilizando sólo una de sus gráficas NVIDIA 580 GTX.

6.1.1. Tiempos tanto de la CPU como de la GPU en la versión single GPU

A continuación se presentan los resultados del tiempo de ejecución dentro de la GPU y el tiempo global de GPU+CPU de manera gráfica. Los datos han sido obtenidos con la asistencia de CUDA Profiler [20].

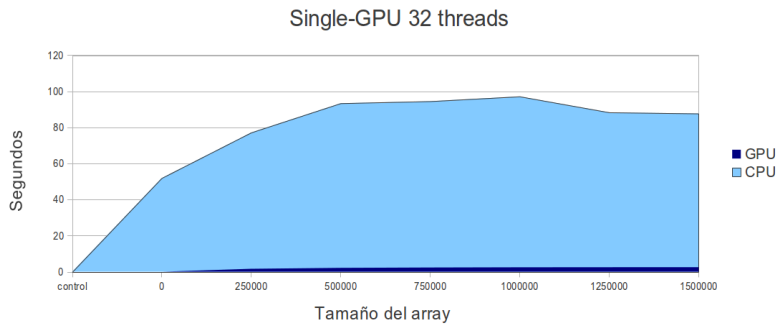


Figura 6.3: Tiempos de ejecución obtenidos al ejecutar sobre una gráfica, configurado con el tamaño de bloque a 32 threads.

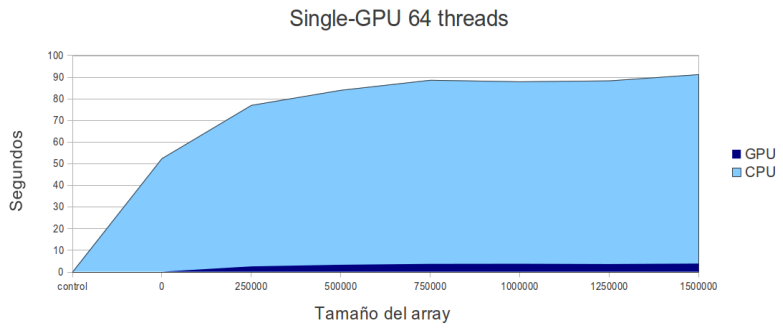


Figura 6.4: Tiempos de ejecución obtenidos al ejecutar sobre una gráfica, configurado con el tamaño de bloque a 64 threads.

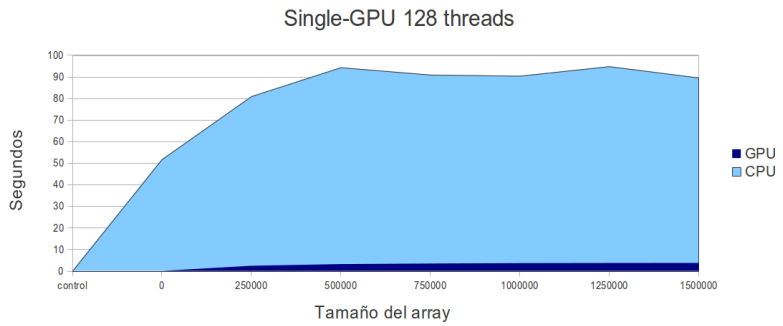


Figura 6.5: Tiempos de ejecución obtenidos al ejecutar sobre una gráfica, configurado con el tamaño de bloque a 128 threads.

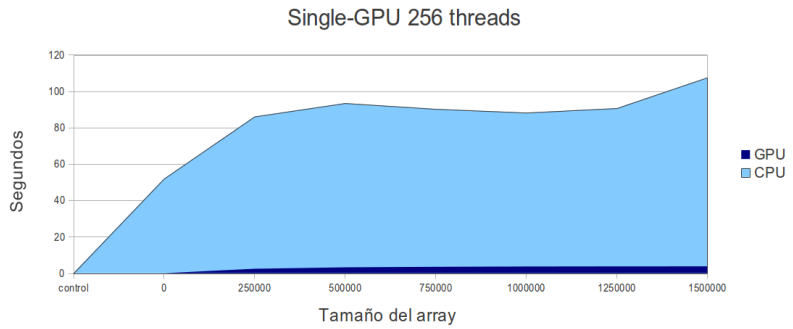


Figura 6.6: Tiempos de ejecución obtenidos al ejecutar sobre una gráfica, configurado con el tamaño de bloque a 256 threads.

La principal función de estos gráficos es la comparativa de los tiempos que se dedican a los test de triángulos en la GPU, con el tiempo que emplea la CPU en el resto del proceso. De estos gráficos, se llega a la deducción que a medida que el tamaño del array de test introducido crece, también crecen los tiempos. No obstante llega un punto en el que los tiempos se estabilizan, realizando pequeñas oscilaciones sobre el mismo rango.

6.1.2. Tiempos de GPU de la versión single

A continuación se han aislado únicamente los tiempos de GPU, para observar con mayor precisión los tiempos y su progresión. Analizando la Figura 6.7 podemos afirmar, que nos encontramos ante progresos incrementales en tiempo, al menos durante la primera parte de la ejecución hasta que el array llega al tamaño de un millón de test, momento en el que se estabiliza el crecimiento del tiempo. La parte incremental es producida porque no estamos llenando la capacidad de la GPU. La zona estable se debe a que estamos ejecutando de manera paralela al mismo tiempo, es decir mientras tenemos llenos unos procesadores trabajando al máximo hay otros ejecutando de igual modo de manera que se lleva a cabo más trabajo en el mismo tiempo.

En el siguiente gráfico se presenta la diferencia en tiempos de todas las ejecuciones anteriores sobre la GPU, con los diferentes valores de threads elegidos para las pruebas. Como se puede observar en la Figura 6.7 en la versión single, a menor número de threads, menores son los tiempos de ejecución, de estos resultados se deduce que si disponemos de número elevado de threads, se produce una sobrecarga en su creación y gestión, lo que nos lleva a la conclusión de que no es rentable la utilización de muchos threads en esta versión.

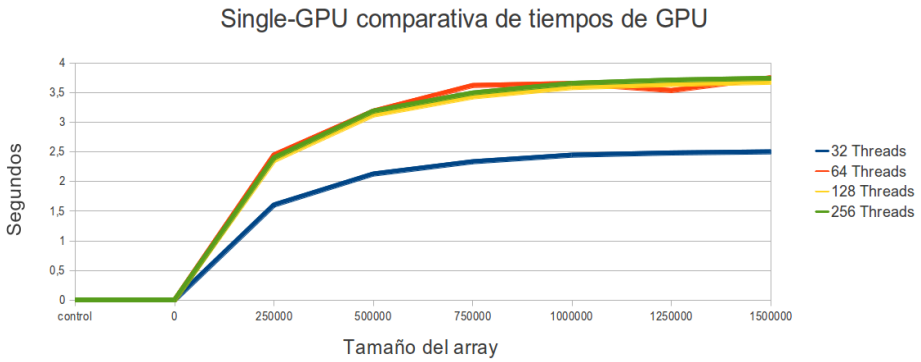


Figura 6.7: Tiempos de GPU obtenidos al ejecutar sobre dos gráficas variando el número de threads.

6.2. Resultados de la versión multi-GPU

6.2.1. Tiempos completos de la versión multi GPU

En este apartado se han realizado las pruebas utilizando las dos tarjetas gráficas NVIDIA 580 GTX, sobre las que se han realizado las ejecuciones pertinentes para la obtención de los resultados que se muestran a continuación. Los resultados son una comparación del tiempo de ejecución dentro de la GPU y el tiempo global de GPU+CPU de manera gráfica. Los datos han sido obtenidos con la asistencia de CUDA Profiler [20].

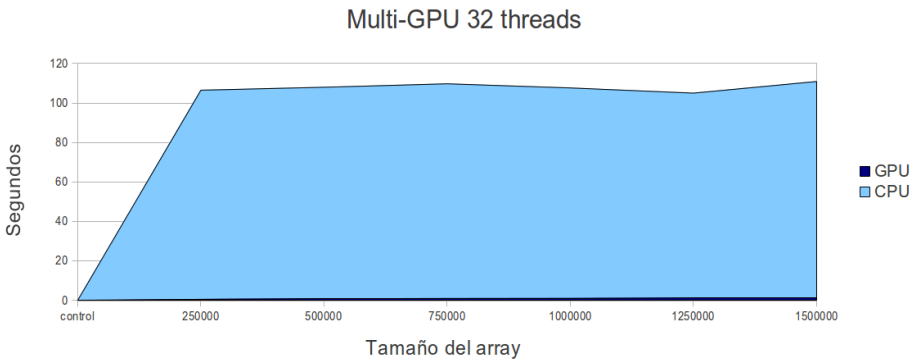


Figura 6.8: Tiempos de ejecución obtenidos al ejecutar sobre una gráfica, configurado con el tamaño de bloque a 32 threads.

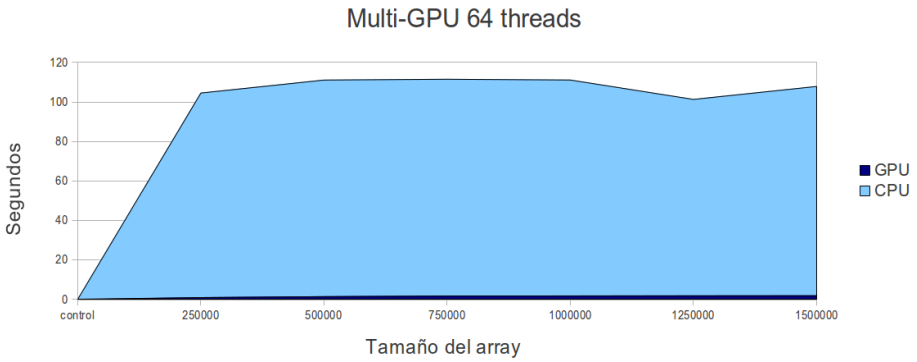


Figura 6.9: Tiempos de ejecución obtenidos al ejecutar sobre una gráfica, configurado con el tamaño de bloque a 64 threads.

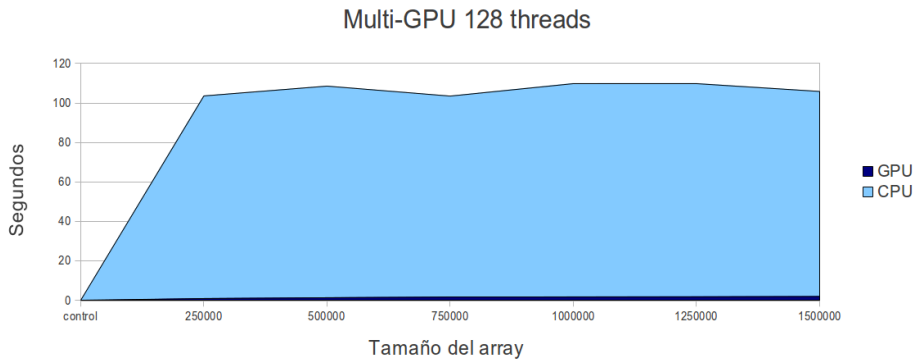


Figura 6.10: Tiempos de ejecución obtenidos al ejecutar sobre una gráfica, configurado con el tamaño de bloque a 128 threads.

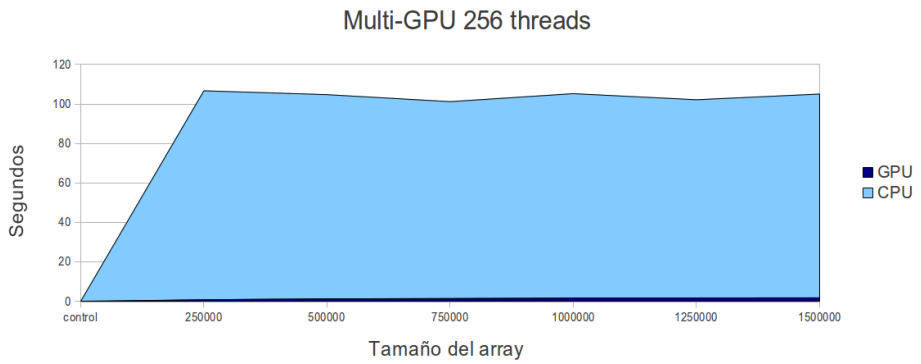


Figura 6.11: Tiempos de ejecución obtenidos al ejecutar sobre una gráfica, configurado con el tamaño de bloque a 256 threads.

6.2.2. Tiempos de GPU de la versión multi GPU

A continuación se han aislado únicamente los tiempos de GPU, para observar con mayor precisión los tiempos y su progresión. En la Figura 6.12 se presenta todas los tiempos de GPU de las ejecuciones anteriores de la versión multi GPU, variando el número de threads. Como se puede observar en este caso, los resultados van creciendo a medida de que el número de threads disminuye. De estos datos se deduce que es más costosa la creación de pocos bloques con muchos threads que de muchos bloques con pocos threads.

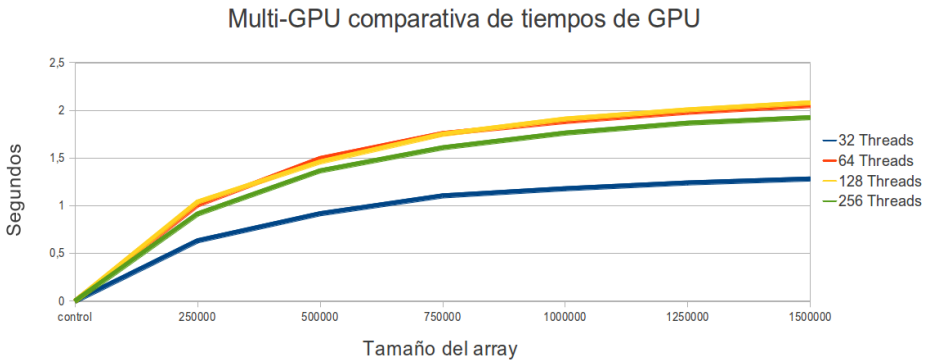


Figura 6.12: Tiempos de GPU obtenidos al ejecutar sobre dos gráficas, variando el número de threads.

6.2.3. Comparativa entre la versión Multi GPU y la versión single GPU

A continuación, se presentan las diferencias de tiempos de GPU/GPUs entre la versión que ejecuta sobre las dos tarjetas gráficas y la que ejecuta sobre solo una. Como se puede ver en todas las representaciones de la Figura 6.13 a la 6.16 la versión que ejecuta sobre una sola gráfica tiene tiempos superiores. Cabría esperar haber obtenido tiempos en la versión single que duplicasen los resultados de la multi GPU, y es prácticamente lo que se ha conseguido, con pequeñas oscilaciones.

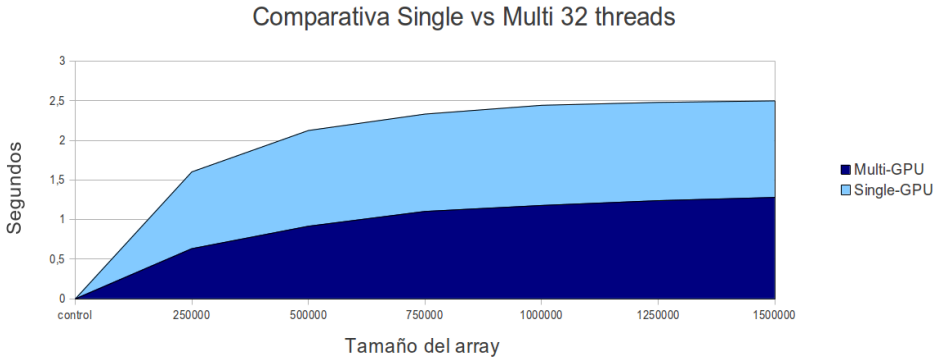


Figura 6.13: Tiempos de GPU obtenidos al ejecutar sobre una y dos gráficas, configurado con el tamaño de bloque a 32 threads.

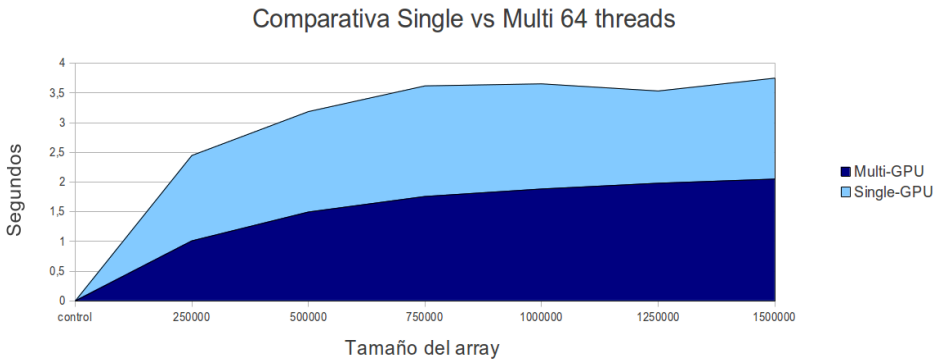


Figura 6.14: Tiempos de GPU obtenidos al ejecutar sobre una y dos gráficas, configurado con el tamaño de bloque a 64 threads.

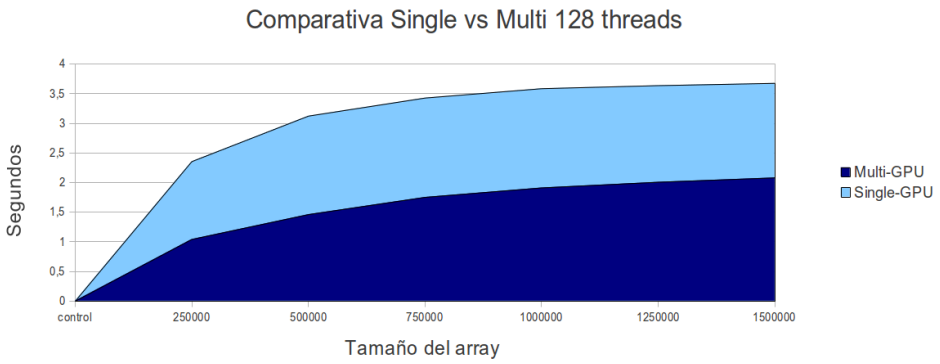


Figura 6.15: Tiempos de GPU obtenidos al ejecutar sobre una y dos gráficas, configurado con el tamaño de bloque a 128 threads.

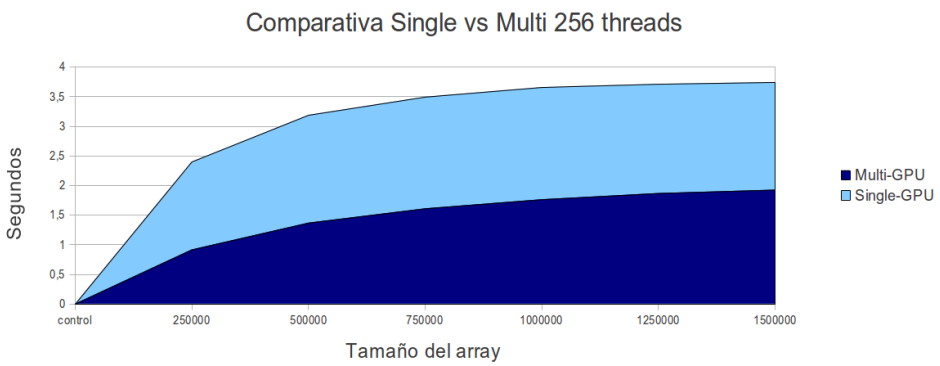


Figura 6.16: Tiempos de GPU obtenidos al ejecutar sobre una y dos gráficas, configurado con el tamaño de bloque a 256 threads.

En conclusión después de estos gráficos no cabe duda de que la mejora en tiempo frente a la versión single es notoria. Era de esperar que al utilizar dos gráficas y dividir el trabajo a la mitad se tardara la mitad del tiempo en ejecutar el mismo número de test.

6.2.3.1. Speedup

Llegado este punto, queda determinar numéricamente cual es la mejora obtenida en las pruebas entre la versión Multi GPU con dos tarjetas gráficas y la versión single GPU. Para llevarlo acabo utilizaremos el speedup como medida, el cual nos dará el valor exacto de mejora que obtenemos en la versión dual frente a la single.

$$S = \frac{T_{single}}{T_{dual}}$$

A continuación, se presentan la relación de resultados obtenidos, divididos segun el número de threads, con el fin de mostrar los speedups obtenidos en todas las pruebas.

Tamaño del array	Tiempo Single	Tiempo Dual	Speedup
250.000	1,603	0,633	2,532
500.000	2,126	0,917	2,318
750.000	2,333	1,104	2,113
1.000.000	2,444	1,179	2,073
1.250.000	2,481	1,24	2,001
1.500.000	2,5	1,281	1,952

Tabla 6.1: Relación de tiempos y mejora de single-GPU y dual-GPU, con 32 threads por bloque.

Tamaño del array	Tiempo Single	Tiempo Dual	Speedup
250.000	2,447	1,009	2,425
500.000	3,186	1,495	2,131
750.000	3,619	1,757	2,059
1.000.000	3,653	1,883	1,94
1.250.000	3,533	1,98	1,784
1.500.000	3,75	2,05	1,829

Tabla 6.2: Relación de tiempos y mejora de single-GPU y dual-GPU, con 64 threads por bloque.

Tamaño del array	Tiempo Single	Tiempo Dual	Speedup
250.000	2,353	1,039	2,265
500.000	3,121	1,459	2,139
750.000	3,426	1,75	1,958
1.000.000	3,584	1,909	1,877
1.250.000	3,636	2,005	1,813
1.500.000	3,674	2,079	1,767

Tabla 6.3: Relación de tiempos y mejora de single-GPU y dual-GPU, con 128 threads por bloque.

Tamaño del array	Tiempo Single	Tiempo Dual	Speedup
250.000	2,397	0,914	2,624
500.000	3,186	1,366	2,332
750.000	3,492	1,608	2,172
1.000.000	3,655	1,762	2,074
1.250.000	3,71	1,866	1,988
1.500.000	3,74	1,924	1,944

Tabla 6.4: Relación de tiempos y mejora de single-GPU y dual-GPU, con 256 threads por bloque.

En conclusión, en las anteriores tablas tenemos speedups del orden de 1,767 al 2,624, es decir en general lo que estos datos reflejan es que se ha reducido los tiempos en la versión de dual, del 40 % al 56 % del tiempo de la versión single.

Capítulo 7

Conclusiones y trabajos futuros

7.1. Conclusiones

El objetivo planteado para este proyecto consistía en paralelizar utilizando la tarjeta gráfica, un algoritmo de detección de colisiones y auto-colisiones. El problema de este algoritmo concretamente, es que es muy costoso computacionalmente, por esa razón se ha planteado la explotación de las tarjetas gráficas de forma paralela. Las GPUs son capaces de realizar cierto tipo de operaciones, sobretodo las matemáticas de manera muy eficiente y rápida. Por eso mismo, realizar el test de colisiones en la GPU era una buena manera de reducir tiempo, al menos parcialmente.

La principal finalidad era la comparación de los resultados con respecto a la ejecución en CPU. Una vez se planteó los prometedores resultados se amplio a la versión multi-GPU, para aprovechar al máximo la máquina de la que se disponía.

La implementación se ha realizado utilizando CUDA como API para programar en la tarjeta gráfica. La elección de CUDA presentó la posibilidad de un análisis comparativo de gran interés.

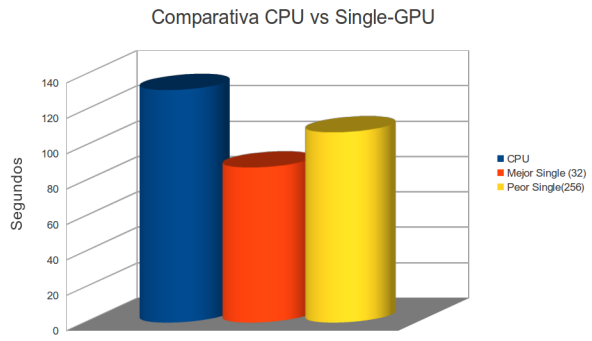


Figura 7.1: Comparativa de tiempos de todos los test que se generan en la secuencia completa, entre la versión de CPU, y el mejor y el peor resultado de Single-GPU.

Los resultados obtenidos con respecto a la versión de CPU, son altamente satisfactorios como se puede ver en la figuras 7.1 y 7.2, ya que existe una mejora en tiempo. En ambas gráficas se puede ver que no hay gran diferencia con respecto al tiempo de solo CPU y eso se debe a que aún la mayor parte del trabajo lo sigue realizando La diferencia entre las versión single-GPU es mayor, ya que no perdemos tiempo en duplicar las estructuras para las dos GPUs.

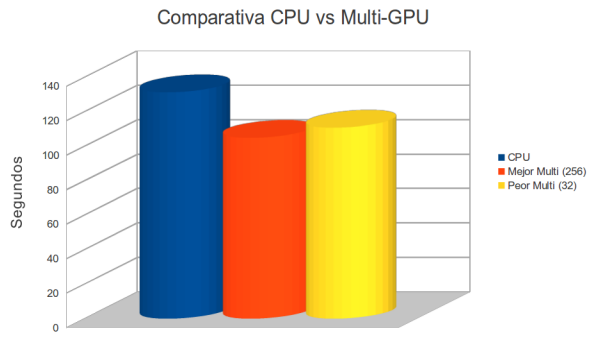


Figura 7.2: Comparativa de tiempos de todos los test que se generan en la secuencia completa, entre la versión de CPU, y el mejor y el peor resultado de Multi-GPU.

Refiriéndonos a la versión para múltiples tarjetas gráficas, los resultados han sido los esperados, ya que al usar dos tarjetas reducimos el tiempo a la mitad pero si tenemos en cuenta los tiempos completos, los resultados están demasiado penalizados por las copias y reservas de memoria. Ya que lo que en Single es crear y mover un array, ahora nos vemos obligados a crear y mover dos de la mitad de tamaño, pero que conllevan el mismo proceso.

En conclusión como manera de paralelizar, las GPUs y este tipo de APIs son muy efectivas, paralelizan el trabajo dando unos resultados en tiempos mucho inferiores con

respecto a ejecuciones en CPU.

7.2. Trabajos futuros

Como inconvenientes en los resultados, tenemos que el entorno no es el más adecuado para explotar el paralelismo de las tarjetas gráficas, ya que la mayor parte del cálculo aún se realiza en la CPU. Cada *frame* necesita toda la ejecución, tanto de la parte ejecutada por la CPU como de la GPU. Así la CPU es la responsable de cargar mallas, evaluar la jerarquía de volúmenes envolventes y preparar el array de test. Bien es cierto, que la parte de test de triángulos, es un cómputo tedioso para la CPU y ahora está en la GPU. No obstante sería interesante, incorporar la funcionalidad que actualmente aun reside en la CPU en la GPU. Cambiar todo el código que ejecuta la CPU a la GPU no es trivial, sobretodo si tenemos en cuenta las limitaciones de CUDA como que el cargar mallas desde disco no es algo habilitado, al igual que el uso de punteros y memoria dinámica.

La posibilidad de incorporar todo el algoritmo a la GPU como se ha mencionado, podría ofrecer grandes beneficios en tiempo. Utilizar mallas más complejas con mayor número de test de triángulos podría exprimir en mayor medida las posibilidades de este algoritmo y de las GPUs para las que esta diseñado.

Bibliografía

[1] http://heim.ifi.uio.no/ knutm/geilo2008/seland.pdf.

[2] http://llpanorama.wordpress.com/2008/06/11/threads-and-blocks-and-grids-oh-my/.

[3] Detección de colisiones mediante recubrimientos simpliciales. *Tésis doctoral*, 2006.

[4] T. Akenine-Möller and E. Haines. *Real-Time Rendering.* 2nd edition A K Peters, 2002.

[5] Ramsay Dyer, Hao Zhang, and T. Möller. Delaunay mesh construction. In *Proc. of Eurographics Symposium on Geometry Processing*, pages 273–282, 2007.

[6] C. Ericson. *Real-Time Collision Detection.* Morgan Kaufmann Publishers, 2005.

[7] M. Figueiredo and T. Fernando. An efficient parallel collision detection algorithm for virtual prototype environments. *ICPADS'04*, 2004.

[8] C. Funfzig, T. Ullrich, and D. W. Fellner. *Hierarchical spherical distance fields for collision detection Vol. 26(1).* IEEE Computer Graphics and Applications, 2006.

[9] Gottschalk. Collision queries using oriented bounding boxes, 2000.

[10] S. Gottschalk, M. Lin, and D. Manocha. Obb-tree: A hierarchical structure for rapid interference detection. *Proceedings of ACM SIGGRAPH '96*, pages 171–180, 1998.

[11] P. M. Hubbard. *Collision Detection: Algorithms and Applications. Algorithms for Robot Motion and Manipulation.* Jean-Paul Laumond and M. Overmars, A.K. Peters (Invited Submission), 1996.

[12] D. L. James and D. K. Pai. Bd-tree: Output-sensitive collision detection for reduced deformable models. *ACM Transactions on Graphics (TOG), Vol 23(3)*, pages 391–396, 2004.

[13] J.J. Jiménez, F.R. Segura, and F.R. Feito. Tutorial sobre detección de colisiones en informática gráfica. *Novática (ATI)*, pages 55–58, Mayo–Junio 2002.

[14] Firas Hamze Kamran Karimi, Neil G. Dickson. A performance comparison of cuda and opencl. *D-Wave Systems Inc., 100-4401 Still Creek Drive, Burnaby, British Columbia, Canada, V5C 6G9*, 2009.

[15] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan. *Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs vol 4*. IEEE Transactions on Visualization and Computer Graphics, 1998.

[16] S. Krishnan, M. Gopi, M. Lin, D. Manocha, and A. Pattekar. Rapid and accurate contact determination between spline models using shelltrees. *Proceedings of Eurographics '98*, pages 315–326, 1998.

[17] M. Lin, D. Manocha, J. Cohen, and S. Gottschalk. *Collision Detection for Interactive Graphics Applications*. IEEE Transactions on Visualization and Computer Graphics, 1995.

[18] Sergio López Arias. Implementación de un algoritmo de detección de autocolisiones para objetos deformables, January 2010.

[19] Mar Maltese Girón. Detección de colisión de mallas de modelos 3d en entornos de memoria distribuida, January 2010.

[20] NVIDIA Mark Harris. Cuda debugging and profiling tools.

[21] S. Quinlan. Efficient distance computation between non-convex objects. in proceedings of international conference on robotics and automation, 1994.

[22] M. Tang, D. Manocha, and R. Tong. Mccd: Multi-core collision detection between deformable models using front-based decomposition.

[23] G. van den Bergen. *Efficient collision detection of complex deformable models using AABB trees. J. Graph. Tools Vol. 2(4)*. 1998.

[24] G. van der Bergen. Collision detection in interactive 3d environments. *Spatial Data Structures*, pages 171–217, 2004.

[25] F. Wieland, D. Cames, and G. Schultz. Using quad trees for parallelizing conflict detection in a sequential simulation, 2001.