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 OPENCL

Autor: Sergio López Arias

Tutores: Alberto Sánchez Campos
Marcos Novalbos Mendiguchía

For you,

Thanks

Agradecimientos

Agradezco toda la ayuda prestada a Alberto y Marcos, sin vosotros ésto no habría sido posible. Gracias por esas últimas semanas de duro trabajo.

A mi familia, pareja y amigos, por estar a mi lado en los duros momentos y dar siempre los ánimos necesarios para seguir hacia adelante.

A la gente que hace que cada día sea diferente y que en las últimas fechas han hecho posible que todo ésto sea más llevadero.

Muchas gracias.

Resumen

En la actualidad la representación gráfica es un elemento ampliamente utilizado y en constante crecimiento, intentando llegar a conseguir tal semejanza con la realidad que el ojo humano no sea capaz de detectar.

Para conseguir dotar de realismo a estas representaciones gráficas, hay que tener en cuenta gran cantidad de parámetros, uno de ellos son las leyes físicas, más específicamente la de las colisiones o choques, que impide a dos o más objetos ocupar la misma posición espacial en el mismo instante de tiempo.

En este problema es en el que se centra este proyecto, en la búsqueda de un método de detección de colisiones que pueda llegar a conseguir un cálculo de ellas en tiempo real o que se aproxime lo máximo posible a éste.

Para llevar a cabo esta tarea, se propone una jerarquía de volúmenes envolventes, unido al uso de la técnica de conos de normales descrita por Volino[23], que ayudará en gran medida a la reducción del número de tests a realizar, conllevando a una mejora en el tiempo.

Además, se propone el cálculo final de las colisiones dentro de la GPU (graphics processing unit), aprovechando de este modo el gran paralelismo de este tipo de procesador, pudiendo realizar gran cantidad de operaciones en un mismo instante de tiempo. Para este paso se hará uso de openCL (open computing language), lenguaje estándar y libre.

Mediante las técnicas descritas, los resultados han sido los esperados, se ha conseguido una buena mejora en el tiempo del cálculo de las colisiones en la animación empleada como prueba.

Índice general

Agradecimientos I

Resumen III

1. Introducción 1

1.1. Objetivos 2

2. Estado del arte 5

2.1. Definición y dominio del problema 5

2.2. Técnicas usadas en la detección de colisiones 6

2.2.1. Volúmenes envolventes 6

2.2.1.1. Esferas envolventes 8

2.2.1.2. Caja envolvente alineada con los ejes 8

2.2.1.3. Cajas envolventes orientadas 9

2.2.1.4. Envolventes de orientación fija 9

2.2.2. Jerarquía de volúmenes envolventes 9

2.2.2.1. Construcción de un árbol de jerarquías de volúmenes envolventes 10

2.2.3. Técnicas de descomposición espacial 11

2.2.3.1. Octrees 11

2.2.3.2. BSP trees 12

2.3. Representación de la animación 13

2.3.1. Malla 3D 13

2.3.1.1. Elementos para la representación de las mallas 3D 14

2.3.1.2. Representación 14

2.3.2. Frame 15

2.3.2.1. Definición 15

2.4. Sistemas con procesadores paralelos masivos 15

2.4.1. GPU 16
2.4.2. GPGPU o GPU Computing 16
2.4.3. GPU vs CPU 17
2.5. Lenguajes de programación 18
2.5.1. CUDA 18
2.5.1.1. Modelo de memoria 18
2.5.2. OpenCL 19
2.5.2.1. Modelo de memoria 20

3. Descripción Informática 21
3.1. Volumen envolvente 21
3.2. Jerarquía de volúmenes envolventes 22
3.3. Jerarquía de detección de autocolisiones 23
3.3.1. Cono de normales 25
3.3.1.1. Construcción y actualización 26
3.3.1.2. Detección de autocolisión 28
3.3.2. Cálculo de adyacencias 29
3.3.2.1. Construcción de las listas de adyacencias 29
3.3.2.2. Comprobación de adyacencia entre nodos 30
3.4. Determinación de autocolisión 30
3.5. Determinación de colisión 33
3.6. OpenCL 34
3.6.1. Definición del problema 34
3.6.2. Evaluación de los test de triángulos 35
3.6.3. Ejecución en un entorno con una sola GPU 36
3.6.4. Ejecución en un entorno con varias GPUs 37
3.6.5. Proceso de ejecución en la tarjeta gráfica 37
3.7. Estructura usada para el cálculo de colisiones 38
3.7.1. Array 39
3.7.2. Lista 39
3.7.3. Tabla Hash 40
3.7.4. Comparativa entre las distintas estructuras 40

4. Resultados Experimentales 43
4.1. Entorno de pruebas 43

4.1.1. Máquina 1 43
4.1.2. Máquina 2 44
4.1.3. Animación 44
4.1.4. Leyendas de las gráficas 45
4.2. Resultados de tests de triángulos usando únicamente una GPU 46
4.2.1. Resultados enviando vértices y aristas 47
4.2.1.1. Máquina 1 47
4.2.1.2. Máquina 2 48
4.2.2. Resultados enviando vértices 49
4.2.2.1. Máquina 1 50
4.2.2.2. Máquina 2 51
4.3. Resultados de tests de triángulos usando dos GPUs 52
4.3.1. Resultados enviando vértices y aristas 52
4.3.2. Resultados enviando vértices 52
4.4. Resultados de la ejecución completa usando únicamente una GPU 53
4.4.1. Resultados enviando vértices y aristas 53
4.4.1.1. Máquina 1 53
4.4.1.2. Máquina 2 54
4.4.2. Resultados enviando vértices 55
4.4.2.1. Máquina 1 55
4.4.2.2. Máquina 2 56
4.5. Resultados de la ejecución completa usando dos GPUs 57
4.5.1. Resultados enviando vértices y aristas 57
4.5.2. Resultados enviando vértices 58
4.6. Comparativa de resultados 59
4.6.1. Máquina 1 59
4.6.2. Máquina 2 59

5. Conclusiones y trabajos futuros 61
5.1. Conclusiones 61
5.2. Trabajos futuros 61

Bibliografía 65

Índice de figuras

1.1. Detección de colisión en un entorno simulado 2

2.1. Mapa Conceptual de los Sistemas de Detección de Colisiones y Autocolisiones 7
2.2. Tipos de volúmenes envolventes 8
2.3. Construcción *top-down* del árbol 10
2.4. Construcción *bottom-up* del árbol 10
2.5. Construcción por *inserción* del árbol 11
2.6. Construcción de un octree 12
2.7. Construcción de un BSP tree 12
2.8. Representación de una malla 3D de triángulos 13
2.9. Elementos para la representación de una malla 3D 14
2.10. Representación de una malla 3D mediante vértices y caras 14
2.11. Diferencia de cantidad de cores entre la CPU y la GPU 17
2.12. Modelo de memoria de CUDA 19
2.13. Modelo de memoria de openCL 20

3.1. Primera condición de la jerarquía de detección de colisiones 23
3.2. Segunda condición de la jerarquía de detección de colisiones 24
3.3. Ejemplo de una malla con un caso patológico 24
3.4. Cálculo del vector normal de un triángulo 25
3.5. Vectores para el cálculo del cono de normales 26
3.6. Proceso del cálculo del DNC 27
3.7. Actualización de la jerarquía de los DNCs 28
3.8. Correspondecia entre un *work item* y una posición de la estructura de test 36
3.9. Inserciones en cada una de las estructuras 41
3.10. Extracciones en cada una de las estructuras 42

4.1. Autocolisiones en la animación 45

4.2. Colisiones y autocolisiones en la animación 45
4.3. Resultados de la ejecución de tests de triángulos sobre una GPU en la máquina 1 (enviando el array con vértices y aristas) 47
4.4. Comparativa de tiempos de ejecución entre, sólamente, la CPU e incorporando la GPU (enviando aristas y vértices) 48
4.5. Resultados de la ejecución de tests de triángulos sobre una GPU en la máquina 2 (enviando el array con vértices y aristas) 48
4.6. Comparativa de tiempos de ejecución entre, sólamente, la CPU e incorporando la GPU (enviando aristas y vértices) 49
4.7. Resultados de la ejecución de tests de triángulos sobre una GPU en la máquina 2 (enviando el array con vértices únicamente) 50
4.8. Comparativa de tiempos de ejecución entre, sólamente, la CPU e incorporando la GPU en la máquina 1(enviando vértices solamente) 50
4.9. Resultados de la ejecución de tests de triángulos sobre una GPU en la máquina 2 (enviando el array con vértices únicamente) 51
4.10. Comparativa de tiempos de ejecución entre, sólamente, la CPU e incorporando la GPU en la máquina 2(enviando vértices solamente) 51
4.11. Resultados de la ejecución de tests de triángulos sobre dos GPUs (enviando el array con vértices y aristas) 52
4.12. Resultados de la ejecución de tests de triángulos sobre dos GPUs (enviando el array con vértices) 52
4.13. Resultados de la ejecución completa sobre una GPU en la máquina 1 (enviando el array con vértices y aristas) 53
4.14. Resultados de la ejecución completa sobre una GPU en la máquina 1 (enviando el array con vértices y aristas) 53
4.15. Resultados de la ejecución completa sobre una GPU en la máquina 2 (enviando el array con vértices y aristas) 54
4.16. Resultados de la ejecución completa sobre una GPU en la máquina 2 (enviando el array con vértices y aristas) 54
4.17. Resultados de la ejecución completa sobre una GPU en la máquina 1 (enviando el array con vértices) 55
4.18. Resultados de la ejecución completa sobre una GPU en la máquina 1 (enviando el array con vértices) 55

4.19. Resultados de la ejecución completa sobre una GPU en la máquina 2 (enviando el array con vértices) 56
4.20. Resultados de la ejecución completa sobre una GPU en la máquina 2 (enviando el array con vértices) 56
4.21. Resultados de la ejecución completa sobre dos GPUs (enviando el array con vértices) 57
4.22. Tiempos de ejecución de las GPUs y tiempos de copiado del array desde la tarjeta gráfica (enviando el array con vértices) 58
4.23. Resultados de la ejecución completa sobre dos GPUs (enviando el array con vértices) 58
4.24. Tiempos de ejecución de las GPUs y tiempos de copiado del array desde la tarjeta gráfica (enviando el array con vértices) 58
4.25. Comparativa entre los tiempos obtenidos en la máquina 1 59
4.26. Comparativa entre los tiempos obtenidos en la máquina 2 59
4.27. Comparativa entre los tiempos single GPU y dual GPU (enviando aristas y vértices) 60

Índice de algoritmos

3.1. Determinación de colisión entre una caja A y una caja B 21
3.2. Creación y actualización de los conos de normales 27
3.3. boolean normal_indicates_autocollision(cone) 29
3.4. boolean normal_indicates_autocollision(coneA, coneB) 29
3.5. selfCollide(node) 31
3.6. selfCollide(nodeA, nodeB) 32
3.7. collide(nodeA, nodeB) 33

Índice de tablas

2.1. Diferencia de ancho de banda entre tarjetas gráficas del mismo modelo con distinta tecnología de memoria 18

4.1. Tiempos destinados a la ejecución de la GPU y de la copia del array desde la tarjeta gráfica, para el caso del envío del array con aristas y vértices, usando únicamente una GPU en la máquina 1 47
4.2. Tiempos destinados a la ejecución de la GPU y de la copia del array desde la tarjeta gráfica, para el caso del envío del array con aristas y vértices, usando únicamente una GPU en la máquina 2 48
4.3. Tiempos destinados a la ejecución de la GPU y de la copia del array desde la tarjeta gráfica, para el caso del envío del array con vértices, usando únicamente una GPU en la máquina 1 50
4.4. Tiempos destinados a la ejecución de la GPU y de la copia del array desde la tarjeta gráfica, para el caso del envío del array con vértices, usando únicamente una GPU en la máquina 2 51
4.5. Relación de tiempos en la ejecución completa de la animación en la máquina 1, usando únicamente una gráfica 54
4.6. Relación de tiempos en la ejecución completa de la animación en la máquina 2, usando únicamente una gráfica 55
4.7. Relación de tiempos en la ejecución completa de la animación en la máquina 1, usando únicamente una GPU 56
4.8. Relación de tiempos en la ejecución completa de la animación en la máquina 2, usando únicamente una GPU 57
4.9. Speedups obtenidos en la máquina 2 59
4.10. Speedups obtenidos en la máquina 2 60

Capítulo 1

Introducción

De años atrás hasta los tiempos actuales se ha pretendido alcanzar la absoluta semejanza entre lo real y lo simulado. Conseguir una apariencia similar entre ambos escenarios no debería de ser un gran reto. El problema se presenta cuando se quieren dotar de las mismas propiedades y comportamientos físicos al escenario simulado.

Con la intención anteriormente explicada, los objetos representados requieren de gran precisión, haciendo que éstos estén formados por gran cantidad de polígonos, llegando a alcanzar los cientos de millones. Tanto detalle en los objetos implica gran cantidad de cálculos, lo que deriva en un gran coste computacional, sin hablar de la cantidad de objetos que la escena tendrá.

Puesto que las animaciones y aplicaciones gráficas son exigentes con el tiempo, como ya se ha comentado, se han buscado métodos para disminuir la carga que conllevan las detecciones de colisiones y autocolisiones. Algunos métodos se basaban en las jerarquías de volúmenes envolventes, ayudando a localizar las áreas donde realmente se ocasionaban, consiguiendo ahorrar el número de comprobaciones innecesarias. Así se puede emplear este tiempo para otras tareas o, simplemente, para agilizar el proceso.

Con la evolución de la tecnología, cada vez se consiguen resultados más reales y se puede llegar a manejar mayor cantidad de datos, pero aun así se siguen buscando métodos que agilice el proceso que nos atañe. Una alternativa que se propone es el uso de las tarjetas gráficas, pues dispone de un procesador, llamado *GPU*, en el cual, en los últimos años, se ha añadido soporte para poder ser programados con operaciones de propósito general, aprovechando de este modo la optimización de los cómputos en coma flotante y la gran capacidad de realizar cálculos de manera paralela.

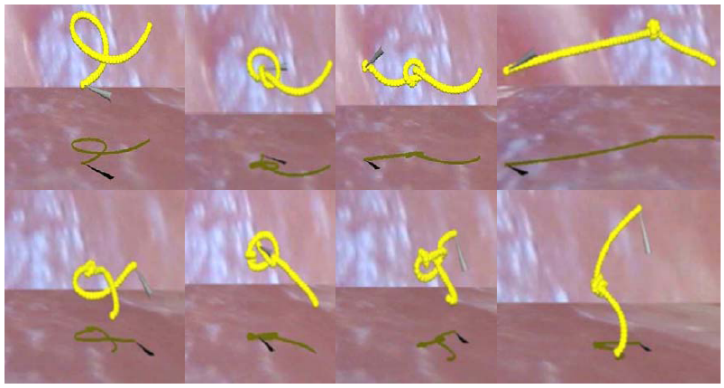

Figura 1.1: Detección de colisiones y autocolisiones en un entorno simulado, utilizando un dispositivo háptico, donde se requiere una gran precisión en los resultados. Figura extraída de [24]

Este proyecto se centrará es en la detección de colisiones y autocolisiones, un problema ampliamente estudiado. Es de alta relevancia en un gran abanico de áreas, como puede ser en robótica, en realidad virtual, en simuladores para formación o en videojuegos. Algunos de los ejemplos citados no requieren de unos resultados demasiado exactos, pero otros requieren de una precisión que supone realizar demasiados cálculos como para conseguir una ejecución fluida en tiempo real.

Para la programación de estas operaciones en la tarjeta gráfica existen varios lenguajes, como puede ser CUDA (nVidia), openCL (estándar abierto y libre) o Stream (ATI/AMD), similares a lenguajes de programación tradicionales, por lo que la curva de aprendizaje es alta, en poco tiempo se pueden comenzar a obtener resultados visibles.

En este caso, se realizará de manera híbrida la detección de las colisiones y autocolisiones, es decir, se hará uso tanto de la CPU como de la GPU para llegar al resultado final.

1.1. Objetivos

El objetivo del proyecto es la estimación de la posición exacta, en tiempo real y buscando la mayor precisión posible, donde se producen colisiones. Estas colisiones podrán producirse entre varios objetos o en uno mismo (autocolisión), dentro de una animación específica. Para llevar a cabo esta tarea se utilizará como único medio la descripción de vértices y aristas de las mallas tridimensionales que componen la representación gráfica de los objetos simulados.

En cuanto al algoritmo de detección de las colisiones, se llevará a cabo una implementación de una jerarquía de volúmenes envolventes, apoyándose en conos de normales para la parte de las autocolisiones. En este caso se obtienen resultados notablemente mejores que si se realizara de la forma tradicional (fuerza bruta), es decir, la comprobación uno a uno de todos los triángulos que conforman los modelos de la animación.

Como parte novedosa y con el fin de conseguir acercarse al objetivo de la detección de las colisiones y autocolisiones en tiempo real, se realizará una implementación que hará uso de la GPU para la realización del cómputo de estas. La razón por la cuál se hará uso del procesador de la tarjeta gráfica es por la principal característica que éstas presentan, la posibilidad de realizar gran cantidad de cálculos de manera paralela.

Como se ha dicho anteriormente, la animación estará formada por objetos, uno de los cuáles será deformable, en el que se estudiarán las autocolisiones, ya que éstos introducen una mayor libertad de movimiento respecto a un objeto rígido.

Capítulo 2

Estado del arte

En este capítulo se definirán los conceptos básicos para entender el concepto de detección de colisiones, así como métodos y técnicas expuestas para la mejora en su cálculo. Además de explicar el concepto de detección de colisiones y autocolisiones, se describirá cómo se representará la animación y aquellos elementos necesarios para ésta. Por último se hablará de las tarjetas gráfica y del concepto que emplea las tarjetas gráficas para cálculos de propósito general (conocido como *GPU Computing*).

2.1. Definición y dominio del problema

El problema a tratar es la detección de colisiones y autocolisiones entre objetos que forman parte de una animación, consiguiendo en pasos posteriores a éste proporcionar a los objetos una serie de efectos, a causa de la colisión, siguiendo una serie de leyes físicas.

Se entiende por colisión aquel instante de tiempo en el que alguna de las partes de varios objetos están ocupando la misma posición en el espacio, produciéndose una interpenetración de ambos. En cambio, una autocolisión ocurre cuando varias partes del mismo objeto están situadas en la misma posición del espacio al mismo tiempo.

La complejidad de una animación depende del número de objetos que la forma y, su vez, del nivel de detalle de estos objetos. Cuanto más compleja sea la escena, más costoso será el proceso de detección de colisiones y autocolisiones, teniendo a veces que llegar a prescindir de un alto nivel de detalle, optando por una mayor fluidez y realismo en los movimientos de los objetos de la animación.

Con el fin de reducir tiempos de ejecución se hará uso de la tarjeta gráfica. Estos dispositivos hardware han evolucionado exponencialmente en los últimos años, aprovechando su cualidad más importante, rompiendo la barrera de realizar los cálculos de forma serializada, dado el gran número de cores del que se dispone. Además de la posibilidad de paralelizar los cálculos, el procesador de la tarjeta gráfica está optimizado para la realización de cómputos en coma flotante de precisión simple, operaciones costosas si hablamos de la cpu.

2.2. Técnicas usadas en la detección de colisiones

Con el fin de reducir el número de cáculos de los tests de triángulos, existen técnicas para poder descartar partes de los objetos que no presentan posibilidad de colisión, reduciendo así la complejidad de $O(n^2)$ en el peor de los casos.

A continuación se expondrán una serie de técnicas y métodos cuya finalidad es la de reducir esta complejidad. En la figura 2.1 se muestra un mapa conceptual de los sistemas de detección de colisiones y autocolisiones.

2.2.1. Volúmenes envolventes

Como se comentó anteriormente, se buscan métodos que eviten el cálculo total de los polígonos que forman las mallas de los objetos, evitando así cálculos mediante fuerza bruta, que se presenta como una tarea inviable si deseamos obtener tiempos para la realización de una animación de manera fluida.

Para evitar ésto, se propone una técnica basada en volúmenes envolventes. Esta técnica consiste en *envolver* al objeto mediante un volumen geométrico, buscando un equilibrio entre el tiempo de cálculo de éste y el tiempo para el cálculo de los test de intersección entre éstos.

Mediante estos volúmenes, estamos añadiendo un paso previo al test de polígonos para evitar el cálculo de los test costosos a cambio de test simples, pues se ha buscado un volumen liviano en estos cómputos como ya se comentó antes.

De este modo, las propiedades que debe de presentar un volumen envolvente para que sea realmente útil son las siguientes[16]:

1. Debe ajustarse al objeto tanto como le sea posible.

2. El cálculo de colisión entre volúmenes envolventes debe de ser poco costoso en comparación con el test de colisión de polígonos.

3. El cálculo del volumen envolvente debe de ser poco costoso.

4. Debe de ser ahorrativo con la memoria.

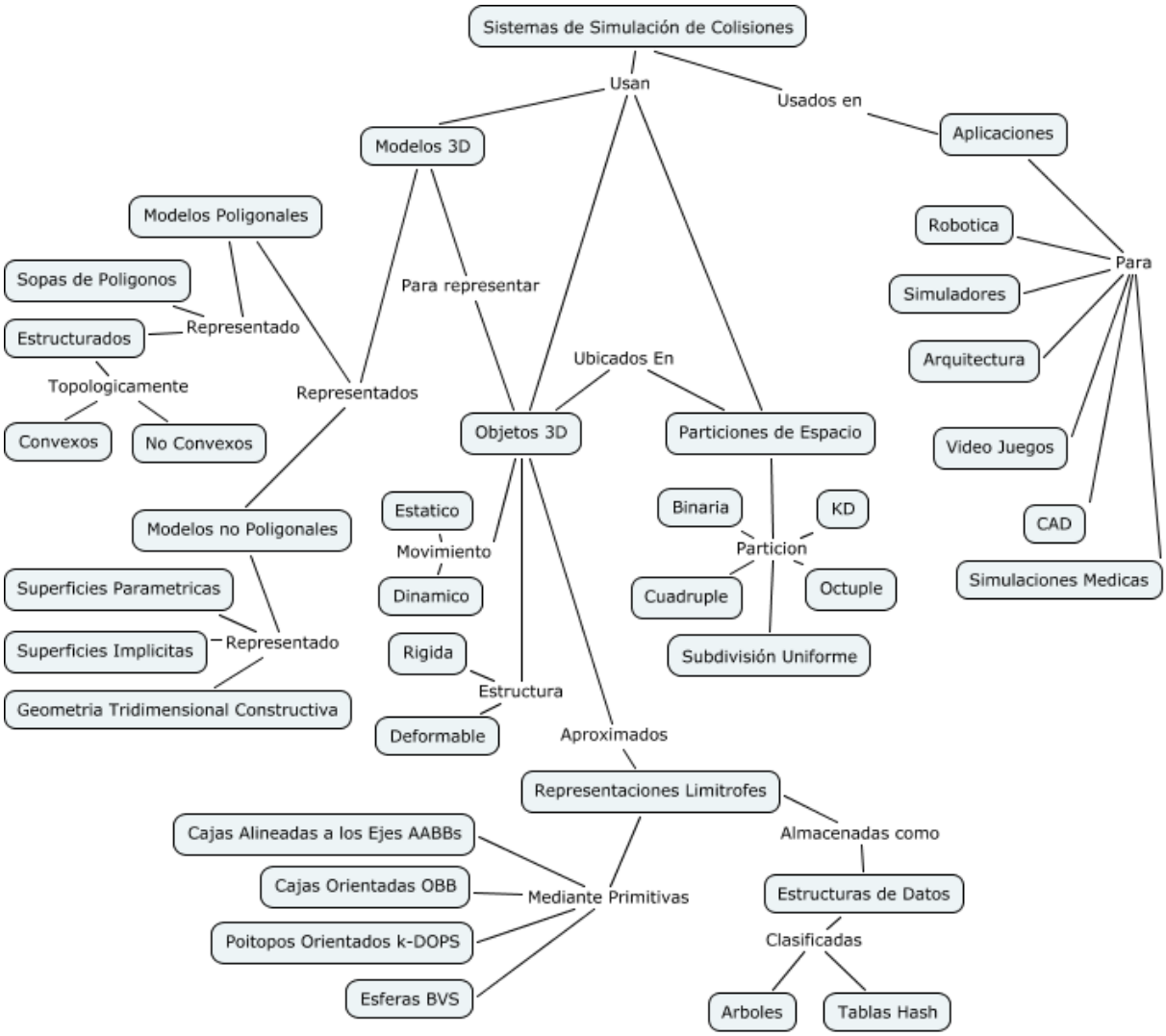

Figura 2.1: Mapa Conceptual de los Sistemas de Detección de Colisiones y Autocolisiones. Figura extraída de [22]

En el caso de que dos volúmenes envolvente intersequen, puede ser suficiente, según el ámbito de la aplicación, para dar positivo en el test de colisiones o será necesario refinar más hasta detectar el punto exacto de colisión en la superficie del objeto (este será nuestro

caso).

Los volúmenes envolventes más representativos y relevantes que aparecen en la bibliografía son las *esferas envolventes (BS)*, las *cajas envolventes alineadas con los ejes (AABB)*, las *cajas envolventes orientadas (OBB)* y los *k-DOPs*. En la figura 2.2 podemos ver estos volúmenes envolventes y cómo se ajustan unos mejor que otros a un objeto determinado.

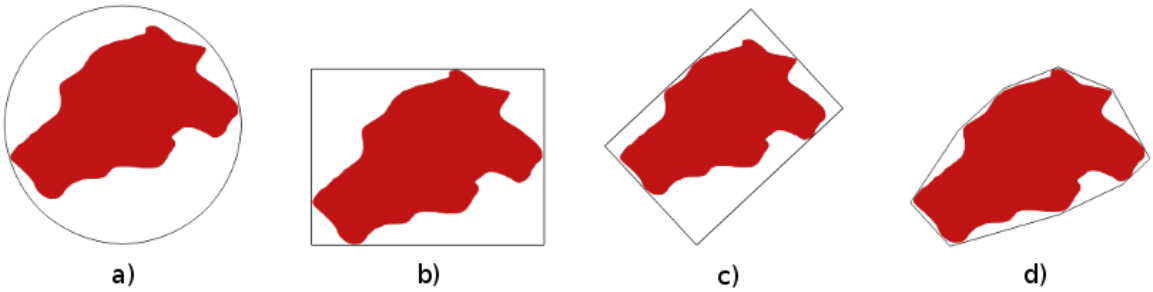

Figura 2.2: Tipos de volúmenes envolventes. a) Esfera. b) Caja envolvente alineada con los ejes. c) Caja envolvente orientada. d) k-DOP

A continuación, se expondrán y explicarán brevemente los volúmenes envolventes citados anteriormente.

2.2.1.1. Esferas envolventes[15]

Este tipo de volumen se basa en el recubrimiento del modelo a través de esferas, consiguiendo obtener un test entre este tipo de volúmenes poco costoso, teniendo que realizar una comprobación entre sus centros y radios, datos necesarios que deberán de ser almacenados junto con este volumen.

2.2.1.2. Caja envolvente alineada con los ejes[7]

Este tipo de volumen recubre al modelo mediante paralelepípedos, cuyas aristas serán paralelas o perpendiculares respecto al eje de coordenadas. El test consistente en las cajas envolventes alineadas con los ejes (comunmente llamado AABB) es poco costoso, sólo será necesario evaluar las proyecciones de las dimensiones sobre los planos coordenados y si intersecan las tres proyecciones. Además, este tipo de volumen envolvente se ajusta mucho mejor a la geometría que las esferas.

2.2.1.3. Cajas envolventes orientadas[12]

La idea de este volumen es similar a la de las cajas envolventes alineadas con los ejes, excepto que éste no presenta la restricción de que las aristas sean paralelas o perpendiculares a los ejes. Las cajas envolventes orientadas (comunmente llamadas OBB) presentan la desventaja de su costoso test basado en el *teorema del eje separador*[11] y una necesidad mayor de almacenamiento. Por contra, este tipo de volumen consigue un mejor ajuste a la geometría de los modelos.

2.2.1.4. Envolventes de orientación fija[20][18]

La principal ventaja de este volumen es el ajuste que se consigue sobre el objeto. Su funcionamiento es similar a la de las cajas envolventes orientadas, incorporando la mejora de que las esquinas de las cajas pueden ser cortadas. El inconveniente de los politopos de orientación discreta (comunmente llamados k-DOP) presentan un test costoso.

2.2.2. Jerarquía de volúmenes envolventes

El uso de volúmenes envolventes reduce de manera rápida la detección de casos negativos de colisiones. Sin embargo, en el caso de que intersequen, habrá que realizar comprobaciones más exhaustivas con test de mayor coste.

Ahora sería óptimo emplear jerarquías que engloben a estos volúmenes envolventes. De este modo, si un volumen envolvente da positivo en un test de colisiones, se continuará evaluando con los volúmenes envolventes hijos de éste. Gracias a que estos test son menos costosos, se reduce notablemente el coste computacional de la detección de colisiones.

El método básico para la construcción de una jerarquía de volúmenes en forma de árbol se realiza de la siguiente manera[16]:

- Obtenemos los volúmenes envolventes del objeto.
- Puesto que estos volúmenes recubren a los triángulos del modelo, se colocan como nodos hoja del árbol de la jerarquía.
- Uniendo los volúmenes envolventes calculados, próximos entre sí, se crearán nuevos volúmenes que engloben a éstos, así hasta obtener uno que recubra a todo el modelo por completo.

Algunas de las descomposiciones mediante jerarquías de volúmenes envolventes más conocidos son: AABB-Trees[7], OBB-Trees[12], Box-Trees[5][25], Sphere-Trees[14][8], k-DOP trees[19], Shell-Trees[25][21], R-Trees[6] y QuOSPO-Trees[13].

2.2.2.1. Construcción de un árbol de jerarquías de volúmenes envolventes

La construcción de un árbol de jerarquías de volúmenes envolventes no es un proceso simple y, además de su complejidad, no siempre se obtiene el árbol óptimo. Para realizar este proceso de construcción del árbol las tres principales estrategias son las que se presentan a continuación.

- *Top-down*. Este método es el más usado para la construcción del árbol y consiste en partir de un gran volumen envolvente que contenga a todos los objetos de la escena, a continuación este volumen se va dividiendo en volúmenes más pequeños, dividiendo también el árbol de manera simultánea. Mediante esta técnica no siempre se obtiene el árbol óptimo, aunque es rápida y fácil de implementar.

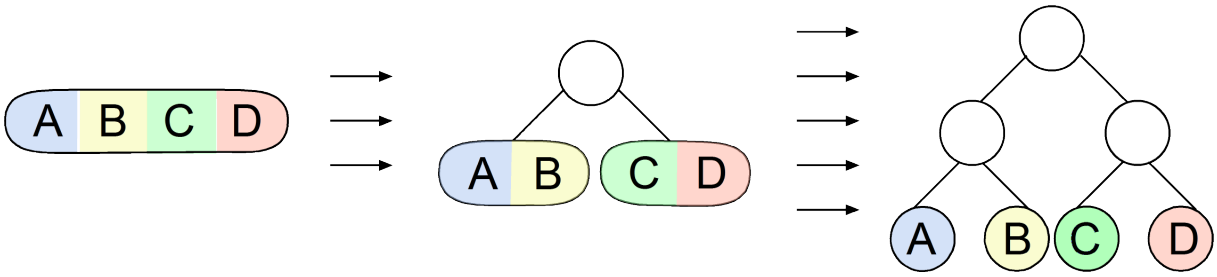

Figura 2.3: Construcción *top-down* del árbol

- *Bottom-up*. En contra de la técnica anterior, en esta se parte de los objetos, es decir, de los polígonos que forman éstos, y se van generando los volúmenes envolventes más grandes uniendo los más cercanos. Con este método se obtiene un árbol óptimo, aunque la complejidad de implemenetanción es alta. Además el tiempo de ejecución aumenta con respecto a la técnica descrita anteriormente.

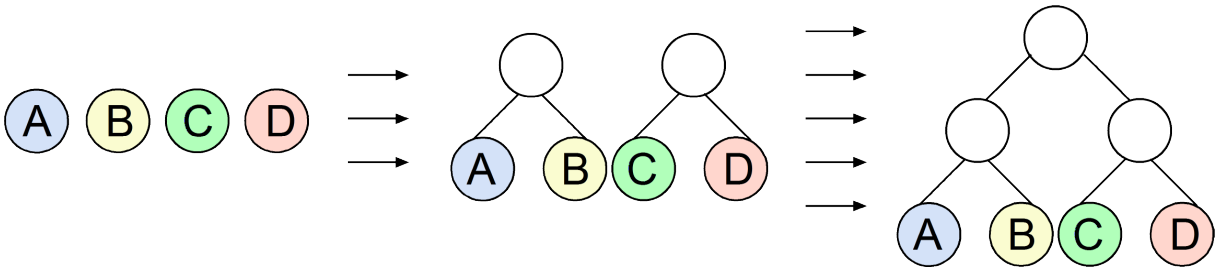

Figura 2.4: Construcción *bottom-up* del árbol

- *Inserción.* En esta técnica se parte del árbol vacío, insertando nodos en éste de manera que se mantenga el equilibrio entre las ramas. Los tiempos de construcción y ejecución son similares a los que ofrece la primera de las técnicas, pero los árboles que se construyen con ésta ofrecen mejores resultados pues, al estar balanceados, los tiempos de búsqueda y manipulación se ven reducidos.

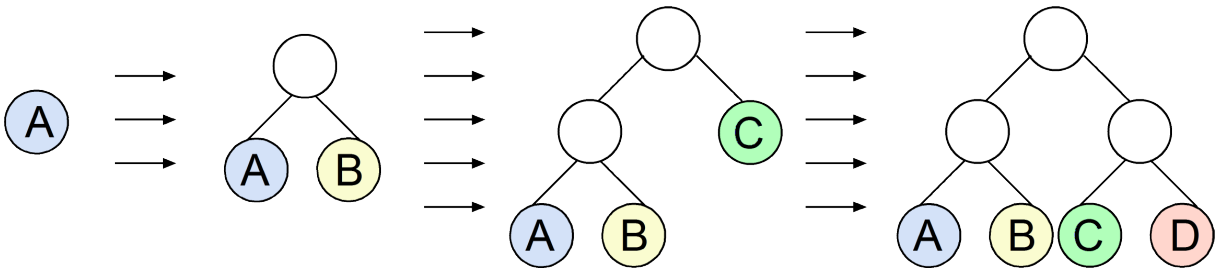

Figura 2.5: Construcción por *inserción* del árbol

2.2.3. Técnicas de descomposición espacial

Como ya se comentó anteriormente, mediante la descomposición espacial en zonas disjuntas, se podrán descartar de forma rápida y con test simples, si los objetos intersecan entre ellos o no, ya que sólo será posible que exista una colisión si ambos volúmenes están ocupando la misma zona del espacio.

De igual modo que para la construcción de los árboles de volúmenes envolventes, para realizar la descomposición espacial existen una serie de técnicas, presentando cada una de ellas unas ventajas respecto a las otras. A continuación se explicará alguna de éstas, consistentes en la división del espacio mediante árboles.

2.2.3.1. Octrees

Los octrees son árboles basados en la división del espacio mediante cajas alineadas con los ejes (ver Apartado 2.2.1.2). Tal y como indica el nombre de esta técnica, los nodos de este árbol serán capaces de almacenar hasta un máximo de ocho nodos hijos (es posible que se tengan menos de ocho nodos hijos en el caso de que alguno de ellos no incluya ninguna parte del objeto dentro del volumen que representa dicho nodo). En la Figura 2.6 se puede apreciar cómo se realiza la construcción de este tipo de árboles a medida que el volumen envolvente va diviéndose.

En este tipo de jerarquías, el nodo raíz es el que representa el volumen global del modelo. A partir de este nodo raíz, se realizan de manera equitativa y alineadas con los

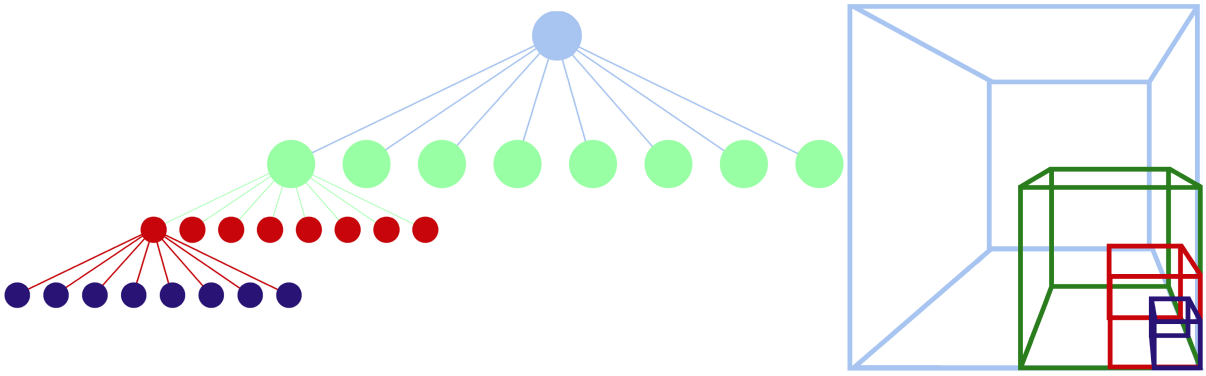

Figura 2.6: Construcción del árbol mediante la división del volumen envolvente (caja alineada con los ejes) en ocho más pequeños

ejes ocho subdivisiones, las cuáles pasarán a ser los nodos hijos. Siguiendo de esta misma manera, se procede con la construcción del árbol, parando de dividir un nodo si se llega a alcanzar una profundidad máxima previamente establecida o, si por el contrario, se alcanza un tamaño mínimo en el volumen que se va a proceder a dividir.

2.2.3.2. BSP trees

La construcción de este tipo de árbol es idéntica a la de los octrees, con la única diferencia que este tipo de jerarquías sólo admite divisiones en dos regiones, teniendo el incoveniente de que éstas no mantienen una orientación y posición constante. En la Figura 2.7 se puede observar la división del modelo y la obtención de un árbol BSP.

Puesto que este tipo de descomposición espacial presenta el incoveniente citado, es compleja la obtención de un árbol óptimo para la división del modelo dado.

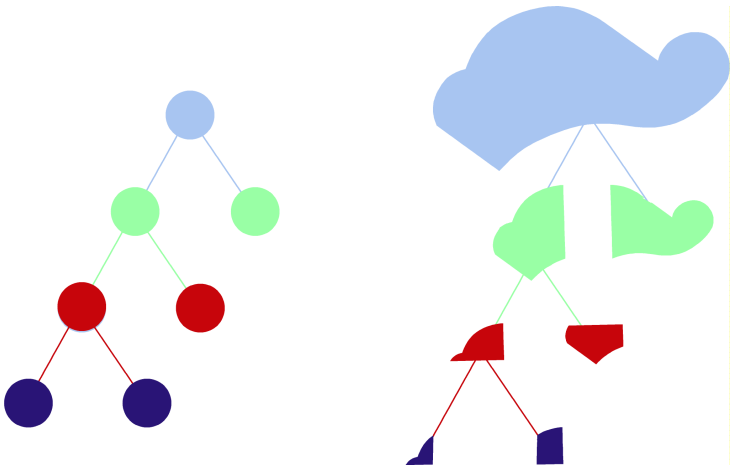

Figura 2.7: Construcción de un BSP tree

2.3. Representación de la animación

Dado que el objetivo del proyecto es la detección de colisiones y autocolisiones entre varios objetos dentro de una animación, en este apartado se explicará qué partes la componen, es decir, se tratará de explicar qué es una *malla 3D* y los *frames* que constituyen a ésta.

2.3.1. Malla 3D

Una malla es una colección de vértices, aristas y caras que definen la forma de un objeto poliédrico en gráficos 3D y en el modelado de sólidos.

Las caras normalmente están constituidas por 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.

Existen varios tipos de mallas, las poligonales, que representan únicamente la superficie, pues el volumen queda implícito mediante la descripción de la superficie. Otro tipo de mallas, distintas a las poligonales, son las volumétricas, que a diferencia de las anteriores, éstas representan explícitamente tanto la superficie como el volumen de una estructura.

Sin embargo, principalmente la representación de las mallas 3D de las que se hará uso, están compuestas por triángulos. A modo de ejemplo en la figura 2.8 se muestra una malla representada mediante triángulos.

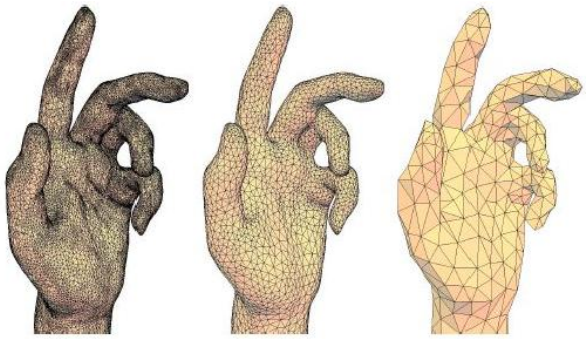

Figura 2.8: Representación de una malla 3D de triángulos. En la figura de la derecha, tenemos una representación con 500 vértices, en el centro con 5000 vértices y en la izquierda con 25000 vértices. Figura extraída de [9]

2.3.1.1. Elementos para la representación de las mallas 3D

Los objetos creados mediante las mallas poligonales pueden ser almacenados mediante distintos tipos de elementos. Éstos incluyen: vértices, aristas, caras, polígonos y superficies (ver Figura 2.9).

En algunas aplicaciones no es necesario almacenar todos estos elementos y pueden ser simplificados.

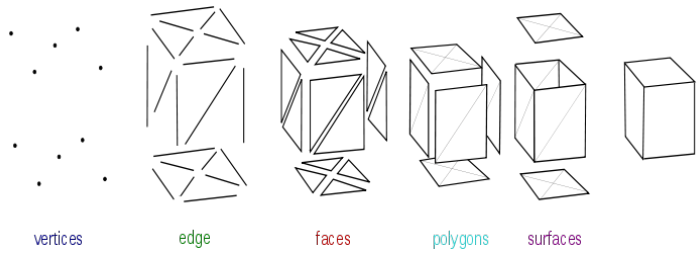

Figura 2.9: Los distintos elementos para la representación de una malla 3D. Figura extraída de [1]

2.3.1.2. Representación

Tal y como se adelantó anteriormente, para la representación de los objetos no es necesario almacenar todos los elementos de éste. De este modo, las mallas 3D de las que se harán uso, quedarán representadas mediante vértices y caras.

En la figura 2.10 se puede ver el modo de almacenamiento de la representación mediante vértices y caras únicamente.

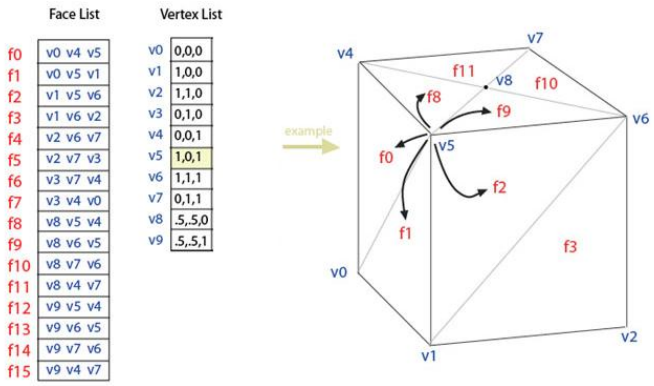

Figura 2.10: Representación de una malla 3D mediante vértices y caras. Figura extraída de [1]

Por un lado se tiene una lista de vértices que se sitúan en nuestro eje de coordenadas de tres dimensiones y, aparte, se dispondrá de otra lista de caras, que relacionan los vértices con el fin de crear una de éstas.

En el caso que nos conlleva, cada una de estas listas están definidas en ficheros planos de texto.

2.3.2. Frame

Una vez introducido el término de malla 3D, se necesita determinar la manera en la que ésta cobra movimiento y se convierte en una animación. Para ello se explicará qué es un frame dentro de un escenario simulado.

2.3.2.1. Definición

Se denomina frame[1] a una imagen particular dentro de una sucesión de imágenes que componen una animación. La continua sucesión de estos fotogramas producen a la vista la sensación de movimiento, fenómeno dado por las pequeñas diferencias que hay entre cada uno de ellos.

La manera de hacer ésto dentro de la malla 3D, es modificar el estado de los vértices que conforman el modelo, es decir, variar las coordenadas de cada vértices que componen los triángulos, con la finalidad de que al cambiar de un frame a otro cree el efecto de movimiento continuo.

2.4. Sistemas con procesadores paralelos masivos

Como se ha comentado en apartados anteriores, para intentar alcanzar el objetivo del cálculo de las colisiones y de las autocolisiones en tiempo real o lo más aproximado a éste, se introduce un nuevo dispositivo de cálculo más optimizado para algunas tareas, como pueden ser los cálculos de operaciones en coma flotante en paralelo.

Además de presentar mejoras en el cálculo de este tipo de operaciones, también introduce el concepto de multiprocesamiento, es decir, el cómputo de un conjunto de instrucciones de manera paralela o concurrente, pudiendo realizar un gran número de

[1] Fotograma o cuadro

cálculos de manera simultánea en poco tiempo.

A continuación se explicará de manera breve la arquitectura de las tarjetas gráficas y por qué de la utilización de éstas para realizar estas operaciones, así como el lenguaje empleado para la programación de estar tareas dentro de la gpu. Además se expondrá el concepto de GPGPU, que no es más que la utilización del procesador de la tarjeta gráfica para realizar aplicaciones de propósito general.

2.4.1. GPU[2]

La GPU no es más que el equivalente a la CPU pero en la tarjeta gráfica. Por tanto, es un procesador dedicado principalmente para procesamiento gráfico, optimizando, por ejemplo, para operaciones de coma flotante en paralelo.

El objetivo de la GPU no es más que el de aligerar el trabajo a la CPU en programas de carácter gráfico, como los videojuegos y/o aplicaciones 3D. De este modo, mientras la GPU se centra en las operaciones relacionadas con los gráficos, la CPU puede dedicarse a cálculos de lógica, como puede ser la inteligencia artificial.

Además de incluir operaciones o primitivas específicamente para gráficos, por ejemplo, el antialiasing[3] u otras encargadas para dotar de mayor realismo las escenas, en los últimos años las tarjetas gráficas se han vuelto cada vez más programables. En éstas se pueden realizar operaciones de propósito general dentro de la GPU, introduciendo de este modo el concepto de *GPGPU*[4] o *GPU Computing*.

2.4.2. GPGPU o GPU Computing

Estos términos hacen referencia al uso de la GPU como procesador para el cálculo de operaciones científicas o técnicas de propósito general.

Esta tecnología se basa en el uso combinado de la CPU y la GPU en un sistema de coprocesamiento heterogéneo. La parte secuencial de la aplicación se realizará en la CPU y la parte de mayor carga computacional se realizará en la GPU, aprovechando de

[2]Graphics Processing Unit o Unidad de Procesamiento Gráfico
[3]Suavizado de bordes de los modelos para simular un aspecto más realista
[4]General Purpose Graphics Processing Unit

este modo la capacidad de ofrece la GPU para la realización de operaciones de manera paralela.

2.4.3. GPU vs CPU

Una de las ventajas del uso de la GPU es la cantidad de procesadores de la que ésta dispone.

A modo de ejemplo, una tarjeta gráfica NVIDIA®GeForce®GTX 590[2] dispone de 1024 procesadores (1 hilo por cada procesador), mientras que un microprocesador Intel®Core™i7 Extreme Processor[3] dispone de tan sólo 6 procesadores (2 hilos por cada procesador). Como se puede observar, la cantidad de hilos de ejecución que dispone una GPU es cerca de 100 veces superior que los de la CPU, lo que hace que para problemas donde la repetitibilidad de un mismo cálculo es alto, la introducción de la GPU puede colaborar en la disminución del tiempo total.

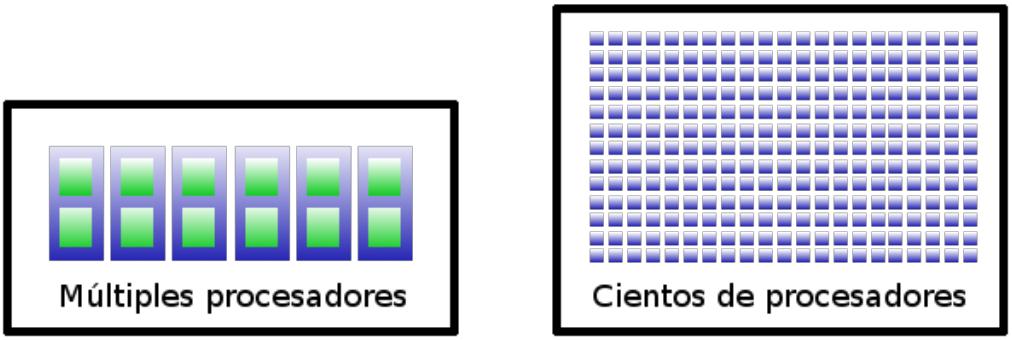

Figura 2.11: Diferencia de cantidad de cores entre la CPU y la GPU

La ventaja de la GPU no sólo se queda en la diferencia en el número de procesadores entre ambos, sino que además, ésta también dispone de un tipo de memoria mucho más evolucionada que la usada en la CPU. En el caso de las GPU actualmente se emplea memoria GDDR5 mientras que la CPU DDR3. La ventaja de esta memoria se traduce en un aumento del ancho de banda, es decir, la cantidad de datos que puede mover por unidad de tiempo.

Como se puede ver en la Tabla 2.1, en la segunda tarjeta gráfica, aun teniendo el mismo interfaz de memoria, el ancho de banda es casi del doble con la tecnología de memoria GDDR5. En el caso de la primera de las tarjetas gráficas, incluso con un interfaz de memoria peor, el ancho de banda sigue siendo superior, casi un 50 %.

Tipo de memoria	Interfaz de memoria	Ancho de banda (GB/s)
NVIDIA®GeForce®GT 545		
DDR3	192-bit	43
GDDR5	128-bit	64
NVIDIA®GeForce®GT 440		
DDR3	128-bit	28,8
GDDR5	128-bit	51,2

Tabla 2.1: Diferencia de ancho de banda entre tarjetas gráficas del mismo modelo con distinta tecnología de memoria

2.5. Lenguajes de programación

A continuación se expondrán los lenguajes que se pueden usar para realizar ejecuciones de un programa creado dentro del procesador de la tarjeta gráfica.

2.5.1. CUDA

CUDA[5], creado por nVidia, está compuesto tanto por un compilador como por un conjunto de herramientas de desarrollo. La programación en CUDA se realiza mediante un lenguaje similar a C/C++.

Una de las principales diferencias con otros lenguajes es que CUDA es propietario, es decir, nVidia ha perfeccionado este lenguaje para sus tarjetas gráficas y no es posible exportar estos programas a tarjetas que no sean de la propia marca.

2.5.1.1. Modelo de memoria

CUDA define una jerarquía de memoria, de más privada a más global en cuanto al acceso. Las zonas de memoria definidas por CUDA son la siguientes:

- Registros y memoria local: memoria sólo visible para un thread concreto. La única diferencia es la velocidad de acceso a éstas.
- Memoria compartida: memoria visible entre todos los threads del mismo bloque, utilizado para la compartición de datos entre estos threads.
- Memoria global, memoria de constantes y memoria de texturas: memoria empleada para el paso de datos entre el host y todos los bloques de la GPU.

[5]Compute Unified Device Architecture (Arquitectura de Dispositivos de Cómputo Unificado)

- Memoria del host: memoria emplazada en la CPU y, ya que ésta será encargada de crear los flujos de ejecución dentro de la tarjeta gráfica, será la que copie los datos necesarios en la memoria con más visibilidad de la GPU.

En la Figura 2.12 se puede ver el modelo definido por CUDA.

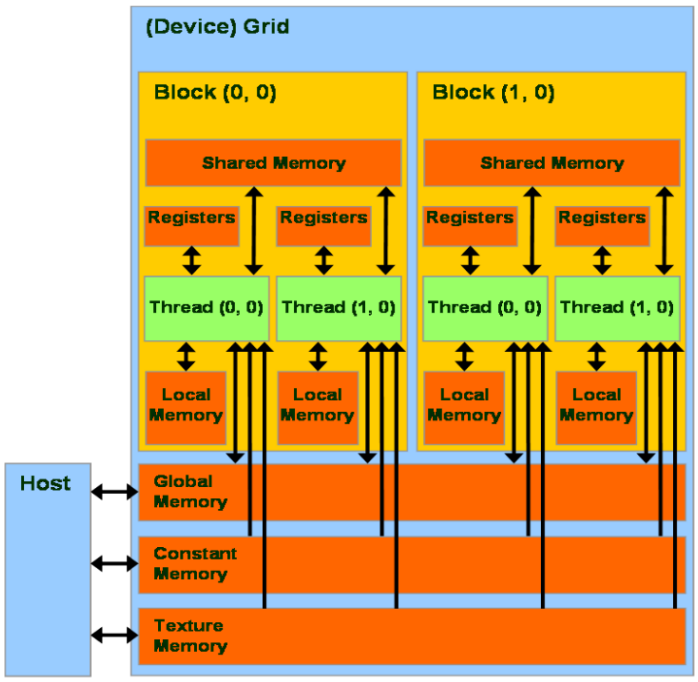

Figura 2.12: Modelo de memoria de CUDA (Figura extraída de [17])

2.5.2. OpenCL

OpenCL[6] está formado por una API y un lenguaje de programación similar a C. OpenCL es un estándar abierto y libre con el fin de que no dependa de una arquitectura determinada de un fabricante, como puede ser CUDA en nVidia o Stream en AMD/ATI.

Gracias a que es un estándar abierto y libre facilita trabajar en un entorno multiplataforma, sin dependencias del sistema operativo o de la arquitectura. Gracias a esto y sin tener que modificar el código de la aplicación, se puede portar a distintos medios, algo que facilita en gran medida el trabajo de los programadores, al no tener que realizar modificaciones dependientes del lugar de ejecución.

[6]Open Computing Language

2.5.2.1. Modelo de memoria

El modelo de memoria definido por openCL es multinivel, desde zonas privadas que sólo verán las unidades de cómputo específicas hasta una zona global con visibilidad para todos los grupos de unidades de cómputo. En la Figura 2.13 se puede ver este modelo de memoria.

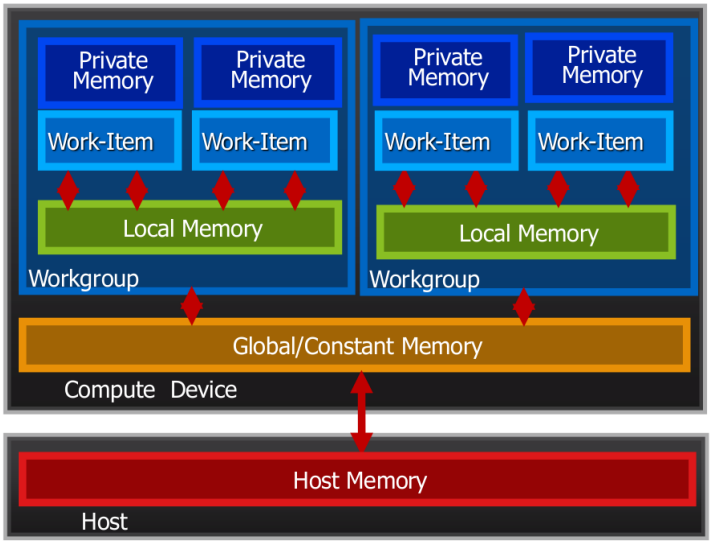

Figura 2.13: Modelo de memoria de openCL (Figura extraída de [4])

El modelo de memoria de openCL define cuatro tipos de memorias:

- Memoria privada: memoria sólo visible para una unidad de cómputo concreta.
- Memoria local: memoria visible para un grupo de unidades de cómputo.
- Memoria global/constante: memoria visible por todos los grupos de unidades de cómputo.
- Memoria del host: memoria ubicada en la CPU, la encargada de copiar los datos necesarios en el resto de disposivitos y recoger los datos después de la ejecución del programa.

Capítulo 3

Descripción Informática

Una vez explicados algunos de los métodos más conocidos de resolución de detección de colisiones, se pasará a explicar con más detalle cual de estas técnicas vamos a emplear para determinar las posiciones exactas de las colisiones en la escena.

3.1. Volumen envolvente

El volumen envolvente seleccionado ha sido el de las *cajas alineadas con los ejes de coordenadas* o *AABB* (ver Apartado 2.2.1.2), dada la facilidad de cálculo de éstos, coordenada máxima en cada uno de los ejes, así como por la simplicidad del test de colisión que presenta.

En este método, dada una caja A y una caja B, se puede afirmar que hay colisión entre ambas según las comprobaciones que hace el Algoritmo 3.1.

```
function Collision(A, B) : return boolean
begin
if A_xMax < B_xMin || A_xMin > B_xMax then
  return 0
end if
if A_yMax < B_yMin || A_yMin > B_yMax then
  return 0
end if
if A_zMax < B_zMin || A_zMin > B_zMax then
  return 0
end if
return 1
end function
```

Algoritmo 3.1: Determinación de colisión entre una caja A y una caja B

Tal y como se ve en el algoritmo, se evalúa que un nodo A y un nodo B no intersecan

mediante la comprobación de las coordenadas de las cajas. La estrategia a seguir será comprobar que tanto las coordenadas x, y y z del nodo A sean mayores o menores que las del nodo B, pudiendo determinar así que ambos nodos no ocupan el mismo espacio, concluyendo que ambas cajas están disyuntas.

3.2. Jerarquía de volúmenes envolventes

Para la implementación de las jerarquías de volúmenes se ha optado por un *BSP tree*. Este tipo de jerarquía consiste en la división espacial de manera binaria, es decir, cada región recubierta por un volumen envolvente será dividida en dos.

Tal y como se comentó en el Apartado 2.2.3.2, los BSP trees presentan el incoveniente de saber por dónde dividir el volumen con el fin de obtener la mejor partición posible. Para no hacer este proceso extremadamente complejo, se realizará una división exacta por la mitad del volumen sobre el eje x de éste. De este modo se obtendrán dos divisiones, exactas en tamaño, que pasarán a ser los nodos hijos.

Con el fin de evitar cálculos futuros innecesarios y, además, reducir la memoria que consumirá esta jerarquía, se realizará una comprobación antes de insertar las divisiones como nodos hijos. Esta comprobación consiste en comprobar si tras la división, alguno de estos nuevos volúmenes no incluye ningun fragmento del modelo o, lo que es lo mismo, uno incluye exactamente el mismo fragmento que el del volumen original. Si ocurre ésto, estos nodos serán descartados, no pudiendo llegar a formar parte de la jerarquía total, terminando aquí el proceso de subdivisión sobre esa rama de árbol.

Así pues, el proceso de construcción de la jerarquía de volúmenes envolventes es de la siguiente manera:

1. Se recubre el objeto mediante una caja.
2. Se calcula el centro del eje x de la caja anteriormente hallada.
3. A partir del centro del eje x, se subdivide la caja en dos más pequeñas del mismo tamaño.
4. Si cada una de estas nuevas cajas incluye algún fragmento del modelo, será insertado como nodo hijo.

5. Si estas nuevas cajas aun se puede subdividir (ambos hijos incluyen algun fragmento del modelo) y no se ha llegado a un límite preestablecido de profundidad del árbol, se procederá de la misma manera.

3.3. Jerarquía de detección de autocolisiones

Como se comentó en apartados anteriores, las autocolisiones son aquellas colisiones que se producen entre partes o zonas de un mismo objeto. Para detectarlas, se necesitan añadir nuevos elementos y nuevas tácticas para su localización.

En este caso se usarán *jerarquías de detección de autocolisiones*[1] explicadas en [23], que define varios métodos para la detección de este tipo de colisiones, enfocados principalmente en las normales de los polígonos y los contornos de los modelos.

Según estos enfoques, una superficie tridimensional S es libre de autocolisiones sí y sólo sí se cumplen dos condiciones. Éstas son las siguientes:

1. Existe un vector V que satisface $V * N \geq 0$ para un vector normal N en la superficie S (ver figura 3.1). De ahora en adelante, llamaremos *normal común* a V.

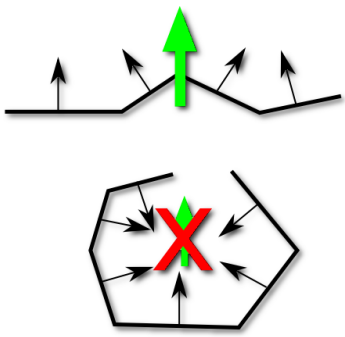

Figura 3.1: En esta figura hay dos ejemplos de dos superficies en 2D: en la primera figura existe un vector V (en verde) que tiene un producto cartesiano positivo frente a cada normal de cada cara. En el segundo ejemplo no hay ningún vector V que tenga el producto cartesiano positivo con cada normal de todas las caras, por tanto, la primera condición no se cumple. Figura extraída de [10]

2. Dado un plano ortogonal P de V, la proyección del contorno de S sobre P no debe de autocolisionar (ver figura 3.2).

[1] Del inglés Hierarchical Self-Collision Detection (HSC)

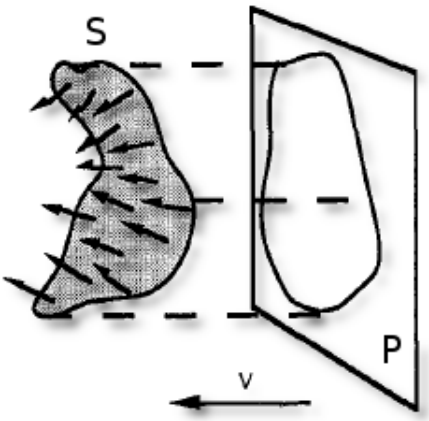

Figura 3.2: Dado un vector V con un producto cartesiano positivo con cada normal de la superficie, construimos un plano ortogonal P de V, y cada borde de S es proyectado sobre P. Hay riesgo de autocolisión si dos o más bordes proyectados están colisionando. Figura extraída de [10]

En la práctica, la segunda condición es raramente evaluada, porque sólo se usa para ejemplos concretos. Generalmente, para la mayoría de los casos, con la primera condición es suficiente. En la figura 3.3 se presenta un ejemplo de una malla con un caso patológico que necesitará de la comprobación de la segunda condición.

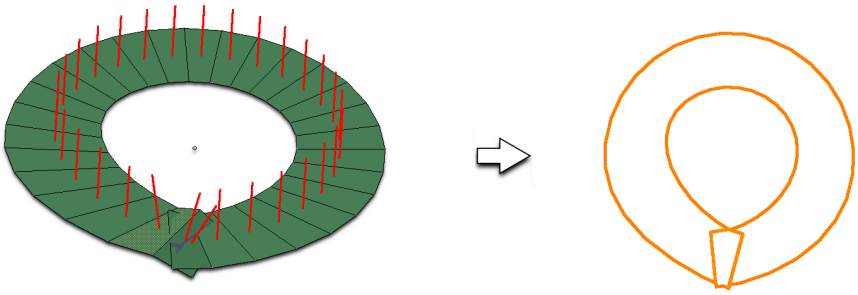

Figura 3.3: En la figura de la izquierda se ve un ejemplo patológico que satisface la primera condición de esta técnica, pero presenta un claro problema de autocolisión. Esta situación hace necesario el uso de la segunda condición. En la derecha vemos la proyección del contorno y la zona donde está autocolisionando. Con ayuda de esta proyección, se puede concluir que la segunda condición no se cumple. Figura extraída de [10]

Para llevar a cabo este algoritmo se necesita usar una estructura de datos jerárquica, en este caso un árbol, para hacer uso del que ya se tiene para el cálculo de las colisiones, así pues, cada nodo de la estructura representará una parte de la superficie del modelo y los nodos hijos contendrán aquellos triángulos que se usarán para determinar finalmente si existe colisión o autocolisión.

De este modo, se incluirá una nueva estructura en cada nodo del árbol llamada *Cono de Normales* que indicará si la superficie está autocolisionando. Cuando un cono de normales indica que no existe posibilidad de autocolisión, se puede concluir aquí la búsqueda por esa rama del árbol. En cambio, si por el contrario, indica que existe posibilidad de autocolisión, será necesario descender hacia los nodos hijos, pues de manera distinta al caso anterior, esto no determina de manera concluyente que sí que hay una superposición de la superficie del objeto.

3.3.1. Cono de normales

Un cono de normales (de ahora en adelante DNC[2]) es una estructura de datos que almacena las regiones angulares de los vectores normales que están dentro. Además no será necesario conocer las coordenadas exactas de este vector normal, ya que sólo se necesita saber si existe o no.

El cono de normales se calcula primeramente desde los nodos hoja de nuestra jerarquía, donde están los triángulos con los que calculamos los vectores normales. Desde éstos construimos y/o actualizamos nuestros DNCs.

Ya que será necesario el uso del vector normal de los triángulos, en la figura 3.4 se explica la manera de obtenerlo, que es simplemente realizando un producto vectorial de dos aristas.

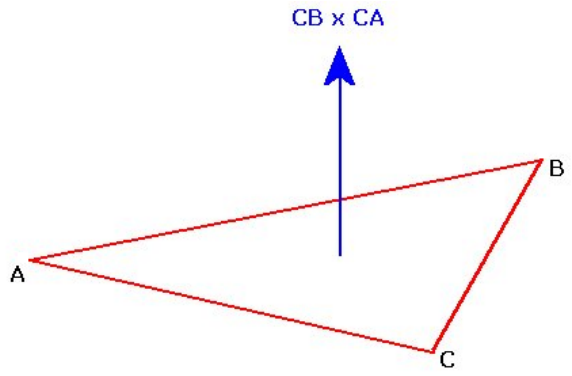

Figura 3.4: Cálculo del vector normal de un triángulo

[2]Del inglés Discrete Normal Cones

3.3.1.1. Construcción y actualización

La implementación del DNC consiste en un array de 14 bits. Cada bit representa el producto cartesiano entre un vector fijo y el vector normal del triángulo.

Los vectores fijos son 14 (ver figura 3.5), correspondiendo cada uno de ellos al producto cartesiano que se tendrá que hacer para el cálculo de los bits que componen el DNC. Estos 14 vectores son: 6 vectores que proyectan sobre las 6 direcciones positivas y negativas del eje cartesiano (x, y, z). Los 8 vectores adicionales apuntan del centro a las esquinas del cubo.

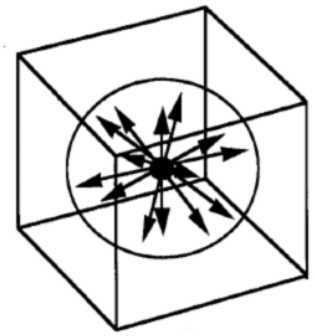

Figura 3.5: En esta figura se muestran los 14 vectores representados geométricamente. Figura extraída de [10]

Si este producto cartesiano ofrece como resultado un número positivo, dentro del DNC se representará con un '1', en cambio, si el resultado dado es negativo, se representará con un '0'.

Por tanto, el array de 14 bits que conforma nuestro DNC almacenará 0s y 1s dependiendo del valor dado de realizar los productos cartesianos de las normales.

En la figura 3.6 podemos ver de forma gráfica el proceso de construcción de nuestros DNCs tal y como ha sido explicado anteriormente. Con nuestro vector normal de los triángulos y tras realizar el producto cartesiano con los 14 vectores fijos, se obtiene el DNC característico al triángulo.

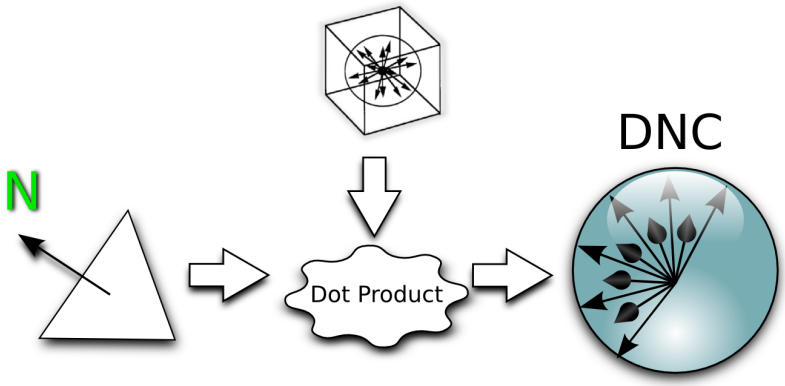

Figura 3.6: En la figura se muestra el proceso de obtención del DNC a partir del vector normal del triángulo y realizar el producto cartesiano con cada uno de los 14 vectores. Se puede considerar al DNC como un conjunto de vectores, aunque después en la implementación se usan representaciones más simples. Figura extraída de [10]

Una vez explicado cómo es el mecanismo de creación de los conos de normales, se presenta el algoritmo que se encargará de su construcción (ver Algoritmo 3.2).

```
procedure updateTree(node)
if is_leaf(node) then
  mconetriangleCone[ntriangles]
  int i = 0
  for all triangle_i in {node.vtriangles} do
    triangle_i.update()
    triangleCone[i].createCone(triangle_i.normal)
    i++
  end for
  node.DNC = triangleCone[0]
  for j = 1 to i do
    node.DNC.include(triangleCone[j])
  end for
else
  boolean first = true
  for i = 0 to node.nsons do
    if first then
      node.DNC = node.sons[i].get_cone()
      first = false
    else
      node.DNC.include(node.sons[i].get_cone())
    end if
  end for
end if
end procedure
```

Algoritmo 3.2: Creación y actualización de los conos de normales

Aparte de la creación de los conos, este algoritmo se encargará de la actualización con los nuevos valores de los triángulos cuando el nuevo frame es cargado y el cono de normales ha sido creado con anterioridad.

Para el proceso de actualización, los DNC implementan la operación intersección $\cap$ (*and* lógico), ya que al estar los conos representados mediante ceros y unos, gracias a esta operación se obtendrá el nuevo DNC a partir de otros dados.

El proceso de actualización de los DNC se realiza siguiendo un método bottom-up (desde los nodos hoja hasta el node raíz). Así, y procediendo mediante la operación citada anteriormente, se obtendrá el cono de normales de un nodo realizando la intersección de todos los conos de los nodos hijos. En la Figura 3.7 se puede ver este proceso.

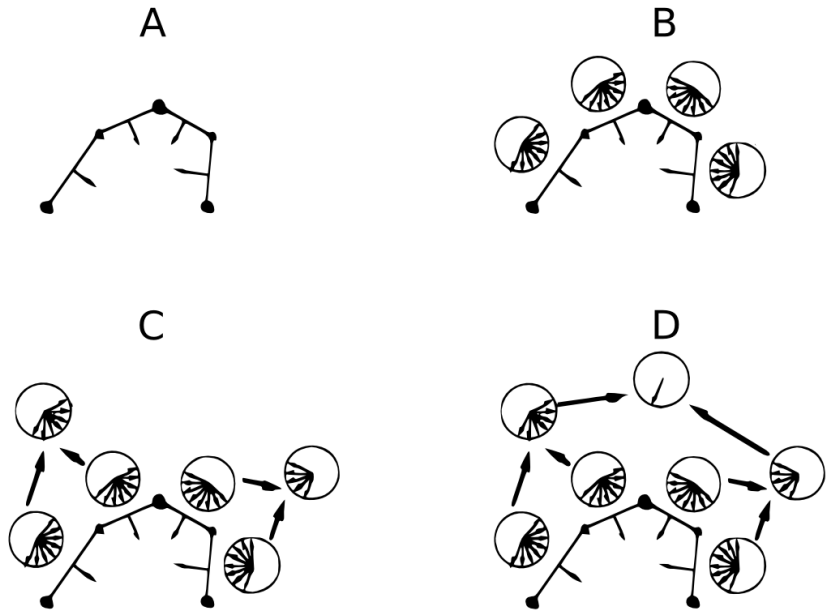

Figura 3.7: En estas cuatro figuras se muestra el proceso de actualización de la jerarquía de los DNCs. En *A* tenemos una vista interior de la superficie, con sus caras y normales. En *B* se calcula el DNC de cada normal. En *C* cada DNC de los nodos, que no son hojas, son actualizados mediante el *and* lógico de los DNCs de sus hijos. Finalmente, en *D* el nodo raíz es actualizado del mismo modo que en el paso *C*. Figura extraída de [10]

3.3.1.2. Detección de autocolisión

Como se comentó anteriormente, mediante los conos de normales se puede determinar si existe o no posibilidad de colisión. Gracias a que el test de conos de normales es de poco costo computacional, se reduce el tiempo de total en la detección de las autocolisiones.

Cuando un test de cono de normales o DNC tiene un valor de 0 indica que existe riesgo de autocolisión, pues ninguna de sus normales sigue la misma dirección y orientación. Así pues, habría que continuar profundizando en el árbol de jerarquías para determinar si realmente se está produciendo una autocolisión y, si es así, determinar el punto exacto de ésta.

Mediante el siguiente algoritmo, se comprueba si todas las componentes del DNC tienen un valor de 0 mediante la máscara 0x0000.

function $normal_indicates_autocollision(cone)$ **: return boolean**
begin
return $cone \equiv 0x0000$
end function

Algoritmo 3.3: boolean normal_indicates_autocollision(cone)

A través del siguiente algoritmo, dados los conos de normales, se sabrá si existe riesgo de autocolisión realizando la intersección de ambos, obteniendo de este modo un nuevo DNC. De igual modo que en el caso anterior, se realizará una comprobación para saber si todas las componentes del cono tienen un valor de 0.

function $normal_indicates_autocollision(coneA, coneB)$ **: return boolean**
begin
return $(coneA \ \& \ coneB) \equiv 0x0000$
end function

Algoritmo 3.4: boolean normal_indicates_autocollision(coneA, coneB)

3.3.2. Cálculo de adyacencias

Una vez explicado el concepto de cono de normal o DNC, se pasará a explicar en qué momentos son usados. Para esto será necesario conocer si los nodos son adyacentes entre ellos o no. Si lo son, se procederá a detectar la posibilidad de autocolisión mediante los conos de normales. En cambio, si no son adyacentes, se procederá a realizar los test de los volúmenes envolventes que recubren a cada nodo.

3.3.2.1. Construcción de las listas de adyacencias

Con el fin de determinar las adyacencias de los nodos, se procede de la siguiente manera:

1. El primer paso es recuperar todos los nodos hoja del árbol de jerarquía de volúmenes, pues éstos son los que almacenan los triángulos del modelo y, a través de los vértices de éstos, se podrá realizar el cálculo para determinar si ambos nodos son adyacentes.

2. A continuación, con los nodos hoja recuperados, se realiza la comprobación de si tienen algún vértice en común los triángulos que encierra un nodo. Si comparten alguno, se podrá determinar que esos nodos son adyacentes.

3. Por último, con las adyacencias de los nodos hoja calculadas, se comenzará con su propagación hacia los niveles superiores hasta llegar hasta el nodo raíz.

3.3.2.2. Comprobación de adyacencia entre nodos

La comprobación de la adyacencia de los nodos es un proceso simple: sólo será necesario comprobar que uno de los nodos está contenido en la lista de adyacencias del otro nodo.

Este método es simple y efectivo, consiguiendo determinar si un nodo es adyacente al otro con una complejidad de O(N) en el peor de los casos, donde N es el número de vecinos que el nodo puede tener.

3.4. Determinación de autocolisión

A continuación se explicarán los dos algoritmos que se emplearán para determinar la presencia de autocolisión en uno de los modelos de la escena.

El primero de ellos será el que inicie el proceso de detección de autocolisiones, es decir, a partir del nodo raíz del árbol de jerarquías. En el caso de que este nodo presente posibilidad de autocolisión mediante los conos de normales (ver Algoritmo 3.3) se continuará con una exploración total de los nodos hijos, tanto del propio nodo como de todas las permutaciones posible de éstos, consiguiendo de este modo una búsqueda completa de todo el árbol. En caso de que no exista posibilidad de autocolisión, se detendrá la exploración por esa rama del árbol, evitando realizar comprobaciones en vano (ver Algoritmo 3.5)

```
function selfCollide(node) : return boolean
begin
if (is_leaf(node)) then
  return false
else
  boolean is_autocollision = false
  if normal_indicates_autocollision(node.get_cone()) then
    for i = 0 to node.nsons do
      for j = i + 1 to node.nsons do
        is_autocollision |= selfCollide(node.sons[i], node.sons[j])
      end for
      is_autocollision |= selfCollide(node.sons[i])
    end for
  end if
  return false
end if
end function
```

Algoritmo 3.5: selfCollide(node)

Como ya se comentó anteriormente, también será necesaria la evaluación de pares de nodos. El proceso de evaluación de pares de nodos es un proceso más complejo, pues incluye todas las funcionalidades citadas en los apartados anteriores.

El primero de los pasos es la comprobación de si ambos nodos son adyacentes, dependiendo de si lo son o no se procederá de la siguiente manera:

- En el caso de que estos nodos no sean adyacentes, la manera de saber si puede existir autocolisión es realizando la comprobación de los vólumenes envolventes de estos nodos, ya que no se pueden resolver las autocolisiones mediante los conos de normales.

- En el caso de que estos nodos sean adyacentes, se pasará a ver si la intersección (ver Algoritmo 3.4) de los conos de normales indican la posibilidad de autocolisión.

  Si la intersección no indica riesgo, se descartará la exploración de ambos nodos, pues se puede estar seguro que sobre estos nodos no hay posibilidad de autocolisión.

  En el caso de que la intersección de los DNC indique posibilidad, se deberá de proceder de la misma manera que con el algoritmo anterior, es decir, se crearán nuevas evaluaciones entre pares de nodos hijos, garantizando la total exploración del árbol de jerarquías.

A continuación se expone el algoritmo que desarrolla el comportamiento explicado.

```
function selfCollide(nodeA, nodeB) : return boolean
begin
boolean is_autocollision = false
if (is_adjacent(nodeA, nodeB)) then
  if (normal_indicates_autocollision(nodeA.get_cone(), nodeB.get_cone())) then
    if (is_leaf(nodeA)) then
      if (is_leaf(nodeB)) then
        addCollisionTests(collisionList, nodeA, nodeB)
      else
        for i = 0 to nodeB.nsons do
          is_autocollision |= selfCollide(nodeA, nodeB.sons[i])
        end for
        return is_autocollision
      end if
    else if (is_leaf(nodeB)) then
      for i = 0 to nodeA.nsons do
        is_autocollision |= selfCollide(nodeA.sons[i], nodeB)
      end for
      return is_autocollision
    else
      for i = 0 to nodeA.nsons do
        for j = 0 to nodeB.nsons do
          is_autocollision |= selfCollide(nodeA.sons[i], nodeB.sons[j])
        end for
      end for
      return is_autocollision
    end if
  else
    return false
  end if
else
  return collide(nodeA, nodeB)
end if
end function
```

Algoritmo 3.6: selfCollide(nodeA, nodeB)

Cuando se llega al momento de realizar comprobaciones entre nodos hojas, se insertarán en una lista los tests de los triángulos englobados dentro de ambos nodos, consiguiendo de este modo almacenar en una misma lista todos los test resultantes de la exploración total del árbol del objeto.

3.5. Determinación de colisión

Con el fin de determinar si existe una colisión entre dos objetos, será necesaria la incorporación de un algoritmo para su detección. En este caso, no se hará uso de ninguna de las técnicas específicas explicadas con anterioridad para el caso de las autocolisiones, es decir, sólo se empleará el método de los volúmenes envolventes.

Este algoritmo parte de ambos nodos raíz de los objetos y mediante evaluaciones de intersección entre los volúmenes envolventes (ver Algoritmo 3.1) se realizará un recorrido de aquellas partes que presentan posibilidad de colisión.

En el siguiente algoritmo se puede ver la manera de proceder de este algoritmo.

```
function collide(nodeA, nodeB) : return boolean
begin
boolean is_autocollision = false
if (collision(nodeA, nodeB)) then
  if (is_leaf(nodeA)) then
    if (is_leaf(nodeB)) then
      addCollisionTests(collisionList, nodeA, nodeB)
    else
      for i = 0 to nodeB.nsons do
        is_autocollision |= Collide(nodeA, nodeB.sons[i])
      end for
      return is_autocollision
    end if
  else if (is_leaf(nodeB)) then
    for i = 0 to nodeA.nsons do
      is_autocollision |= collide(nodeB, nodeA.sons[i])
    end for
    return is_autocollision
  else
    for i = 0 to nodeA.nsons do
      for j = 0 to nodeB.nsons do
        is_autocollision |= collide(nodeA.sons[i], nodeB.sons[j])
      end for
    end for
    return is_autocollision
  end if
end if
return false
end function
```

Algoritmo 3.7: collide(nodeA, nodeB)

Primeramente se realizará la comprobación de si ambos volúmenes está ocupando una misma parte del espacio. En caso negativo, se puede estar completamente seguro de que no va a existir una colisión entre esas partes de los objetos. En cambio, si ambos recubrimientos intersecan en alguna de sus partes, será necesario proceder de la manera siguiente:

- Si ambos nodos evaluados son nodos hoja de sus respectivos árboles se deberán de realizar las comprobaciones de todos los triángulos contenidos en la porción del espacio de ambos nodos. Como ya se comentó, este es el último de los test que se realizan en el proceso de detección de colisiones y, además, el más caro de todos. Estos triángulos involucrados en dichos tests, se almacenarán en una lista para su futura evaluación.

- Si uno de los nodos es hoja y el otro no, se creará un nuevo test de manera recursiva entre el nodo hoja y todos los nodos hijos del nodo que no lo era. De esta manera se busca exactamente cuál es el fragmento exacto de los objetos que presentan la colisión.

- En el caso de que ninguno de los nodos sea hoja, se constituirán test con las permutaciones entre todos los nodos hijos de ambos nodos.

Este algoritmo, además de ser el encargado de encontrar las colisiones en varios objetos, será el que se encargará de evaluar las autocolisiones en aquellos nodos que no son adyacentes entre sí.

3.6. OpenCL

Entre los lenguajes de programación para la tarjeta gráfica enumerados en el estado del arte (ver Apartado 2.5), se ha optado por openCL. A continuación se explicarán los pasos necesarios para crear una ejecución en la GPU, así como la estructura que se ha usado para almacenar todos los test de triángulos que presentan un riesgo de colisión o autocolisión y serán evaluados dentro de este procesador.

3.6.1. Definición del problema

Al hacer uso del procesador de la tarjeta gráfica estamos intentando realizar una tarea que aproveche la gran capacidad de paralelismo/cómputo paralelo que presenta este dispositivo hardware. Por tanto, se pretende ejecutar un mismo problema que presente

una gran repetibilidad y que sea bastante específicos, pues una de las cosas, por ejemplo, que no está permitida dentro de la tarjeta gráfica es la recursividad.

Así pues, la tarea seleccionada para realizar en la GPU es la de comprobar si dos triángulos presentar colisión o autocolisión. Este test consiste básicamente en evaluar si los planos que forman cada triángulo intersecan entre ellos. Principalmente, estos cálculos están basados en valores en coma flotante y test independientes que aprovecharán el paralelismo. De este modo, se están aprovechando dos de las ventajas predominantes de este procesador.

Dado el funcionamiento básico de los algoritmos de detección de colisiones y autocolisiones (ver Algoritmo 3.5, Algoritmo 3.6 y Algoritmo 3.7), primero se realizará la agrupación de los triángulos que presentan riesgo de colisión o autocolisión, pero sin llegar a realizar la evaluación final de éstos. En lugar de la evaluación final, estos triángulos serán almacenados en una estructura creada. Se dispone de dos de estas estructuras, una será usada para el proceso de las colisiones y la otra será usada para el proceso de las autolisiones.

Una vez que todos los árboles son explorados al completo y la estructura anteriormente citada está compuesta por todos los triángulos problemáticos y, antes de cambiar de frame de la escena, se prepara todo para crear una ejecución con esta estructura dentro de la GPU.

Dependiendo del número de tarjetas gráficas y, por ende, del número de GPUs, la creación de las ejecuciones de los programas dentro de éstas variará, pues habrá que adaptar los datos de entrada de manera acorde. Cada *work item* de cada GPU acceda a los test de los triángulos una vez y que todos los test sean evaluados.

3.6.2. Evaluación de los test de triángulos

Puesto que los tests contenidos en la estructura son independientes uno de otros, se podrán realizar de manera simúltanea, aprovechando de este modo la paralelización que presenta la GPU.

Aunque este proceso puede parecer complejo, cada *work item* tiene un identificador

único que se hará corresponder de manera única con la posición exacta del test dentro de la estructura de tests. En la siguiente figura se puede observar cómo se realiza.

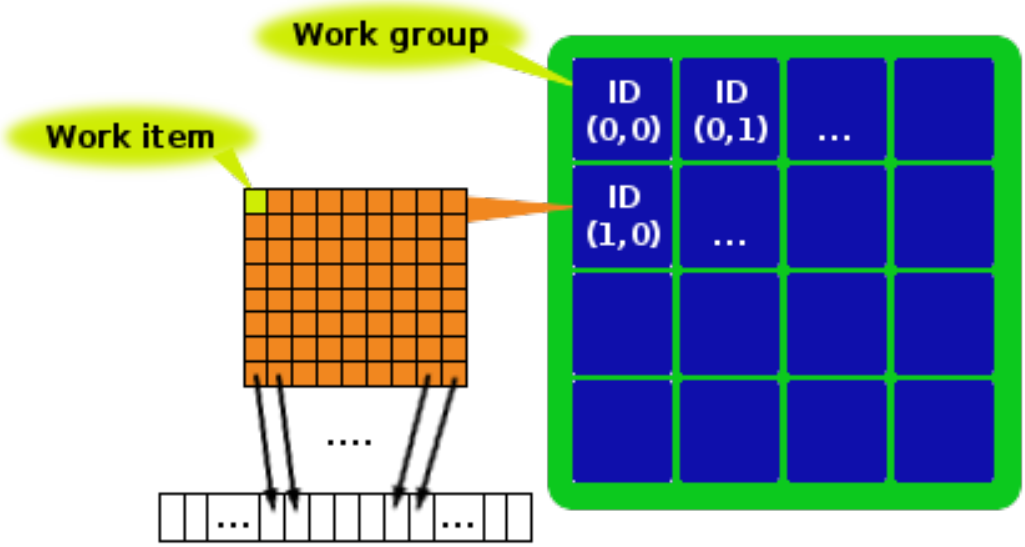

Figura 3.8: Correspondecia entre un *work item* y una posición de la estructura de test

Cada *work item* viene identificado de manera única, consiguiendo de este modo la exclusividad en el acceso de la posición de la estructura de test. Este identificador se sabe mediante la llamada a la función *get_global_id()*.

3.6.3. Ejecución en un entorno con una sola GPU

En el caso en el que sólo se dispone de una GPU, se entiende que todo el trabajo recaerá en ésta únicamente. De este modo, será necesario realizar la copia de toda la estructura con los triángulos candidatos de presentar colisión en la memoria de la tarjeta gráfica, en la zona de la memoria global (ver Figura 2.13), teniendo, de este modo, todos los *work item* acceso directo a los datos necesarios para realizar los test de colisiones y autocolisiones.

Cuando el proceso de ejecución en la tarjeta de gráfica finalice, se devolverá la estructura con las colisiones calculadas a la original que se pasó. Así, de este modo, será actualizado el valor que hace referencia a la presencia de un problema en el test de ambos triángulos.

3.6.4. Ejecución en un entorno con varias GPUs

La manera de actuar en este caso es realizar divisiones de esta estructura en tantos fragmentos como número de GPUs de las que se dispone, además del tamaño de la porción proporcionada. Así en el momento de realizar los cálculos, cada procesador gráfico evaluará cada una de estas porciones de manera totalmente paralela. Siempre se intentarán crear trozos del mismo tamaño, realizando de este modo la misma carga de trabajo en cada una de las GPUs, pues al estar balanceada, se consiguen tiempos de ejecución más óptimos en cuanto a tiempo.

Una vez que el proceso de ejecución de la GPU ha finalizado, hay que reconstruir la estructura enviada con los nuevos resultados. En este paso hay que ser cuidadoso, introduciendo los resultados en la estructura original en el lugar correcto. Si no se reconstruye esta estructura bien, podrían darse casos de pérdida de resultados positivos o ubicarlos en triángulos que no presentan ninguna colisión.

3.6.5. Proceso de ejecución en la tarjeta gráfica

En las secciones anteriores se ha explicado el proceso que se llevará a cabo cuando se dispone de un entorno con una única GPU o en entornos con varias. De este modo, el proceso típico para una ejecución de un algoritmo/programa dentro del procesador de la tarjeta gráfica es el siguiente.

1. Creación de un contexto para un tipo de plataforma determinado.
2. Selección de un dispositivo para la ejecución.
3. Creación de una *command queue* para la ejecución y petición de memoria.
4. Creación de los buffers de memoria de entrada/salida con los datos necesarios.
5. Compilación y construcción del código que se ejecutará en el kernel. Este código deberá de estar en un fichero por separado.
6. Creación del kernel con el programa que se ejecutará (Punto 5)
7. Inserción de los argumentos para el kernel (buffers creados en el Punto 4)
8. Ejecución del kernel
9. Extracción de los resultados.

Mediante estos pasos, se creará la ejecución de la detección de colisiones y autocolisiones en la tarjeta gráfica. Este proceso es idéntico en ambas versiones del algoritmo, tanto como en multiprocesador como en monoprocesador, salvo algunos detalles que se comentarán a continuación:

- Se deberá de crear un *command queue* por cada procesador gráfico sobre el que se ejecutará el algoritmo, dado que a partir de este componente se pueden copiar los elementos en la memoria de la tarjeta gráfica, así como recuperarla cuando finaliza la ejecución.
- Como ya se ha comentado, la estructura al completo será dividida en fragmetos, los cuales serán evaluados en cada GPU, es por esto, que se deberá de crear un buffer específico para cada uno de éstos.
- Puesto que la entrada del algoritmo variará de una GPU a otra, será necesario crear tantos kernels como procesadores se disponga, pues cada uno de ellos tendrá unos parámetros distintos. Tanto la estructura de los triángulos, como del tamaño del fragmento, aunque se intenten crear fragmentos del mismo tamaño.
- Una vez que cada kernel es ejecutado en su correspondiente GPU, se procederá a la reconstrucción de la estructura de tests con los resultados obtenidos de la ejecución del algoritmo. Como ya se dijo, este proceso es de los más importantes, pues si se realiza mal, los resultados no serán consistentes y se podrá perder información de las colisiones o autocolisiones.

3.7. Estructura usada para el cálculo de colisiones

Para la selección de la estructura de los triángulos candidatos de presentar autocolisión o colisión se ha realizado un estudio para saber cuál es la que más se ajusta al comportamiento requerido. La necesidad primordial es un acceso rápido a los datos de los triángulos, puesto que dentro de la tarjeta gráfica, cada *work item* accederá una posición determinada por su número de identificador. Esta deberá de ser la principal característica de la estructura a elegir.

Para la comprobación de la eficiencia de cada una de las estructuras, se realizarán comprobaciones de la ocupación en memoria, del tiempo de respuesta en una serie de inserciones y el tiempo de respuesta en una serie de consultas.

3.7.1. Array

Un array es una matriz o vector de tamaño fijo, no puede redimesionarse en tiempo de ejecución, y cuyos elementos son siempre del mismo tipo. Al tener un tamaño fijo, hay que saber de antemano cuál es éste o se derrochará memoria en reservar más espacio del necesario o se podrá presentar la posibilidad de que el array se llene y no se puedan almacenar más elementos.

Esta estructura es adecuada cuando el acceso a los datos es de forma aleatoria e impredecible, pues a través de un índice, el acceso será de manera directa e inmediata.

En cuanto a lo que se refiere a la inserción, si se sabe cuál es la ocupación del array, será de manera inmediata, pues se sabrá el índice o posición del siguiente elemento vacío y candidato a almacenar los datos. Así pues, la inserción al final será directa, mientras que si la inserción es en el medio, será un proceso costoso.

3.7.2. Lista

Una lista es una estructura de datos compuesta por nodos, con posibilidad de ser de distinto tipo. Cada nodo mantiene un puntero que hacen referencia al nodo siguiente, o puede almacenar dos punteros haciendo referencia al nodo anterior y al siguiente. Gracias a que está compuesto por nodos, la ocupación de memoria es idónea, a medida que se necesita añadir un nuevo elemento, un nuevo nodo es creado con los datos, actualizando los punteros de los nodos que se vean afectados por la inserción.

Para recuperar un elemento a partir del índice o posición, habrá que recorrer la lista desde el principio hasta llegar al elemento requerido, ya que no se dispone de una manera directa de acceso a posiciones de los nodos de la lista.

Si se desea insertar un nuevo elemento, se pueden distinguir distintos comportamientos:

- Se puede realizar insertar por el principio de la lista, lo que supone una inserción inmediata.
- Se puede insertar por el final de la lista, lo que supone recorrer toda la lista hasta el final para realizar la inserción o mantener un puntero al final de la lista para conseguir inserciones inmediatas.

- Se puede insertar por el medio, lo que supone recolocar el puntero del nodo anterior para que apunte a éste. Este tipo de inserción será costoso.

3.7.3. Tabla Hash

La principal característica de las tablas hash es que relaciona claves con valores. Esta estructura suele está compuesta por un array, donde cada elemento de éste es una lista. Al estar compuesto por ambos, el uso de memoria es intermedio al que usa cada uno de ellos. Si se quiere un mejor acceso a los datos, el array deberá de ser más grande, perdiendo eficiencia en la memoria. Si en cambio, se quiere mejor eficiencia en memoria, se reducirá el tamaño del array y, por ende, la eficiencia en los accesos a los datos.

Para realizar un inserción y saber dónde estará ubicado el elemento, se somete la clave a una función hash, que su principal función es conseguir una dispersión de los elementos uniforme a través de todo el array. Una vez que se sabe la posición del array donde se ubicará, se añadirá a la lista de dicha posición. Si la función hash no es adecuada, existirían posiciones del array con muchos elementos, lo que perjudicaría la búsqueda y recuperación de los datos.

En el caso del acceso a los datos, a partir de la clave y la función hash se sabrá la posición exacta del array donde estará dicho elemento. Este acceso será inmediato, pues el acceso a las posiciones del array así lo son. Una vez que se sabe dónde estará, se tendrá que recorrer la lista hasta encontrarlo.

Como se puede ver, esta estructura está en la mitad en cuanto a redimiento respecto a las otras dos estructuras, auna las mejoras de ambas, reduciendo los inconvenientes de éstas.

3.7.4. Comparativa entre las distintas estructuras

Para determinar cuál de las estructuras citadas y explicadas anteriormente se ajusta más a lo que se busca, se van a realizar una serie de pruebas sobre cada una de ellas. Estas pruebas serán las siguientes:

1. Se realizarán un número aleatorio de inserciones en cada una de ellas, siempre de los mismos datos.

2. Se realizarán una serie de extracciones de cada una de las estructuras dependiendo de la cantidad de datos insertados, en este caso, se extraerán un tercio. Se consultarán las mismas posiciones en cada de las estructuras, para tener en todos los casos el mismo entorno de pruebas. Además, las posiciones a consultar serán de manera aleatoria, pudiendo realizar accesos totalmente inesperados y no controlados.

En la Figura 3.9 se pueden observar los resultados obtenidos después de realizar una serie de inserciones en cada una de las estructuras.

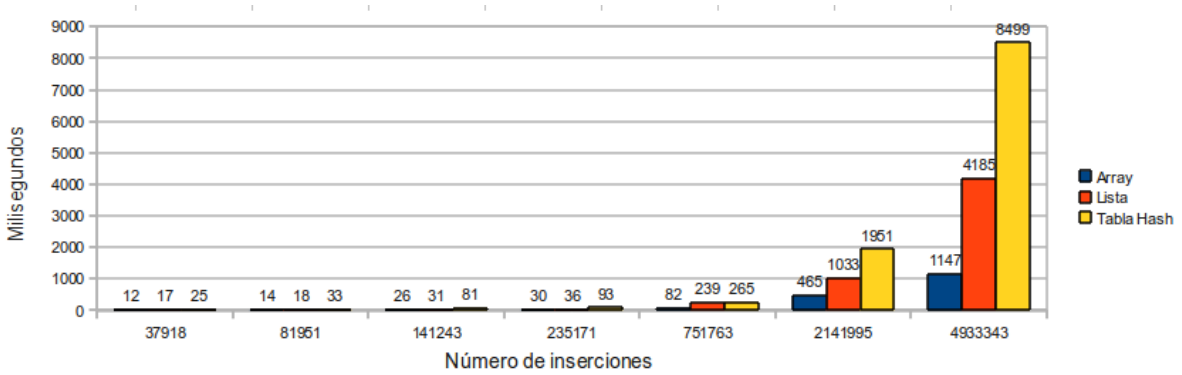

Figura 3.9: Inserciones en cada una de las estructuras

En inserciones de un tamaño reducido, hasta los 250000 elementos, los tiempos parecen mantenerse entre el array y la lista, mientras la tabla hash se desmarca con una pequeña diferencia respecto a éstas. A partir de un mayor número de inserciones, mientras el array se mantiene en un tiempo más o menos estable, sin mucha diferencia, la lista y la tabla hash comienzan a dar un resultados bastante elevados respecto a la primera.

En cuanto a las inserciones, las mejores elecciones posibles son los arrays y las listas. Para decantarse por una de éstas, se evaluará la otra de las propiedades necesarias para el problema que nos atañe, que es la extracción de los elementos de las estructuras a partir de su posición. En la Figura 3.10 se pueden ver los resultados obtenidos en esta prueba.

Ahora al contrario que en las insercciones, los que mantienen unos tiempos similares a consultas bajas son los arrays y las tablas hash, mientras los tiempos de las listas son extremadamente altos incluso en estos números tan bajos. Sólo con ver el tiempo que arroja cuando se realizan alrededor de 250000 extracciones las listas, se puede llegar a la conclusión de que son inviables para ser usadas, pues si la intención es conseguir tiempo bajos en las consultas para conseguir ejecutar paralelamente gran cantidad de test, no

se puede perder más tiempo en conseguir los datos que en la realización de los propios cálculos.

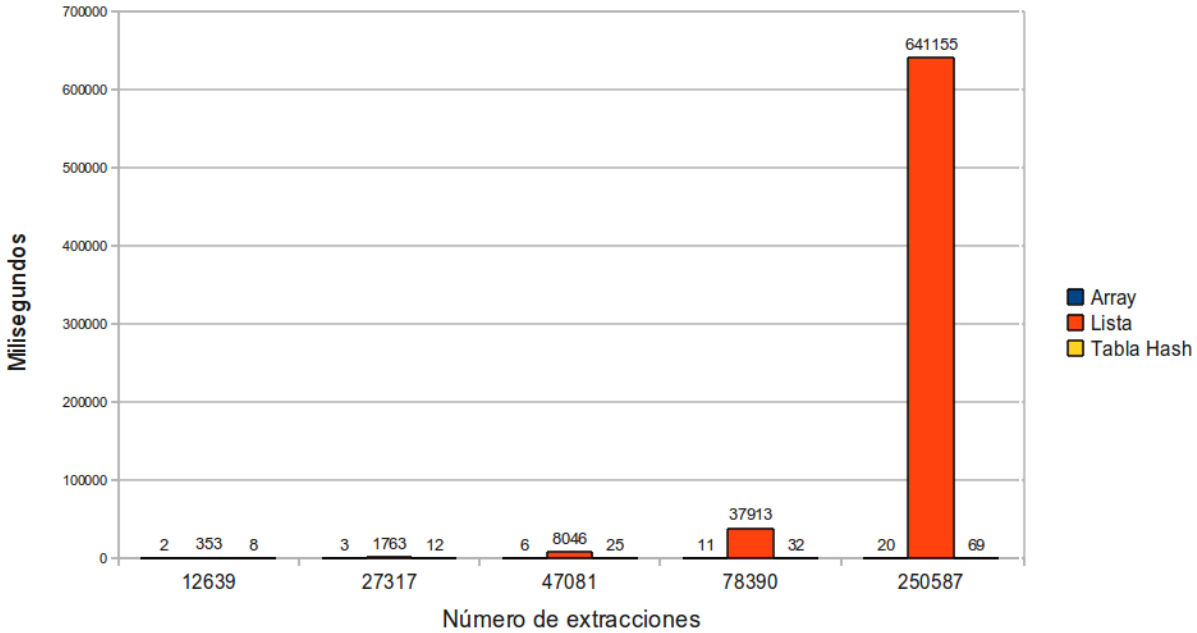

Figura 3.10: Extracciones en cada una de las estructuras

Así pues, tras observar todos los datos obtenidos mediante estas simples pruebas, se puede llegar a la conclusión de que la mejor elección posible son los arrays, pues tanto los tiempos de inserción como los de extracción son los más razonables de todos, aunque se deje de lado el aprovechamiento de la memoria.

Capítulo 4

Resultados Experimentales

En este capítulo se expondrán los resultados obtenidos de todas las pruebas realizadas sobre la detección de colisiones y autocolisiones.

Los resultados serán mostrados de manera visual, mediante gráficas, mostrando así cómo mejora un método respecto a otro.

4.1. Entorno de pruebas

Todas las pruebas han sido realizadas sobre un entorno de pruebas con dos máquinas. Una de ellas es de última generación, mientras que la otra ya cuenta con 5 años. Se han elegido máquinas de tal diferencia, para comprobar si el hardware influye, en gran medida, sobre los resultados que se van a obtener.

4.1.1. Máquina 1

A continuación se mostrará una descripción de los componentes de la primera de las máquinas.

- Procesador Intel® Core™2 Quad Q6600 2.4 GHz (4 cores)
- 4 GB de memoria RAM DDR2
- Tarjeta gráfica NVIDIA GeForce 8800 GT.
  - CUDA Cores: 112
  - Frecuencia del procesador: 600 MHz
  - Memoria: 512 MB GDDR3

  - Frecuencia de la memoria: 900 MHz
  - Interfaz de memoria: 256-bit
  - Ancho de banda de la memoria: 57.6 GB/s

- Sistema Operativo Ubuntu 11.04 Natty Narwhal.

4.1.2. Máquina 2

Tal y como se hizo con la otra máquina, a continuación se mostrará la descripción de la segunda máquina sobre la que se han realizado pruebas.

- 2 procesadores Intel® Xeon® E5645 2.4 GHz (6 cores cada procesador)
- 48 GB de memoria RAM DDR3
- 2 tarjetas gráficas NVIDIA GeForce 580 GTX.
  - CUDA Cores: 512
  - Frecuencia del procesador: 772 MHz
  - Memoria: 1536 MB GDDR5
  - Frecuencia de la memoria: 2004 MHz
  - Interfaz de memoria: 384-bit
  - Ancho de banda de la memoria: 192.4 GB/s
- Sistema Operativo Ubuntu 11.04 Natty Narwhal.

4.1.3. Animación

Las pruebas principales se realizarán sobre una animación preestablecida. Esta animación está compuesta por una bola y una tela, desarrollada a través de 183 frames. En primera instancia la tela está por encima de la bola y, a medida que esta avanza, la tela cae y comienza a enrollarse sobre la bola. Este es el punto crítico de la animación, donde ambos objetos están en contacto y, además, existen gran cantidad de pliegues sobre la tela.

En esta animación tenemos las dos clases de objetos descritos en apartados anteriores (objeto rígido y objeto deformable). La tela será el objeto deformable, por lo que sobre éste se realizará el estudio de las autocolisiones. Con ambos objetos que forman la animación

se estudiarán las colisiones.

A continuación se describirá el modelo de cada uno de estos objetos.

- Tela: objeto compuesto por 91470 triángulos.
- Bola: objeto compuesto por 760 triángulos.

En la siguiente imagen se pueden apreciar un conjunto de autocolisiones en la manta (zonas verdes).

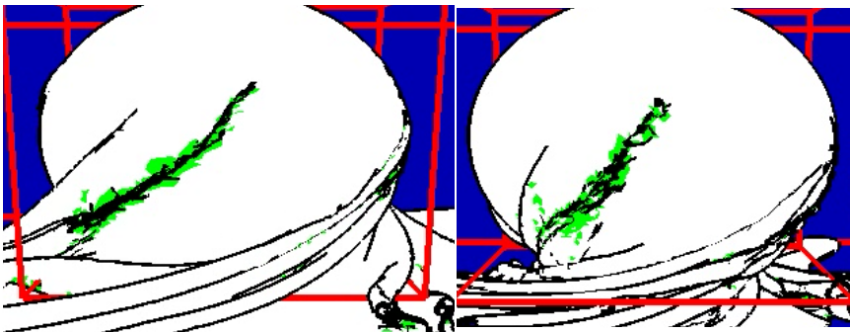

Figura 4.1: Autocolisiones en la animación

En la Figura 4.2 se pueden apreciar tanto colisiones (zonas rojas) como autocolisiones (zonas verdes).

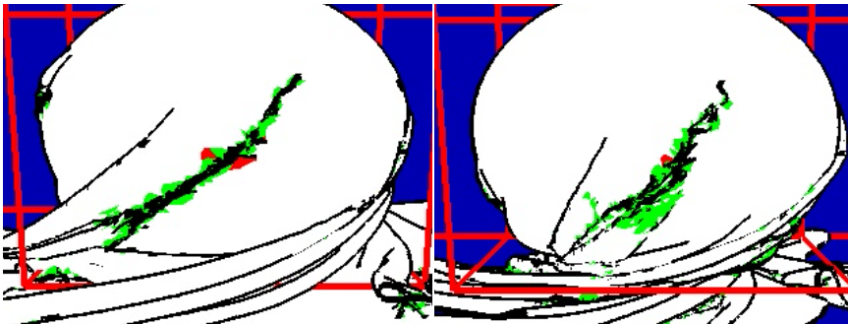

Figura 4.2: Colisiones y autocolisiones en la animación

4.1.4. Leyendas de las gráficas

En este apartado se explicará qué significa cada uno de los elementos de las leyendas.

Gráficas Single GPU y Dual GPU:

- Execution Time: tiempo de ejecución total.

- GPU Execution Time: tiempo en el que la GPU ha estado ejecutando.
- Memcpy DtoH Time: tiempo empleado para traerse la estructura desde la tarjeta gráfica.

Gráficas comparativas GPU vs CPU:

- CPU Execution Time: tiempo de ejecución total usando únicamente la CPU.
- GPU Execution Time: tiempo de ejecución total incorporando a la GPU en el proceso.

4.2. Resultados de tests de triángulos usando únicamente una GPU

Como ya se explicó, el trabajo que se enviará a la GPU es la comprobación de si dos triángulos intersecan entre ellos. Para que la GPU tenga constancia sobre qué triángulos deberá de evaluar, se enviará un array con la descripción de éstos (ver Apartado 3.7).

En primera instancia, la descripción de los triángulos venía dada mediante la definición de los vértices de ambos triángulos, así como de la definición de las aristas de éstos. Más adelante y con el fin de realizar comprobaciones sobre la mejora en tiempo, se redujo la descripción de los triángulos a sólo sus vértices, hallando internamente (en la GPU) las aristas. Mediante este nuevo método, se reduce el tamaño del array a enviar a la GPU, consiguiendo mejores tiempos en la transferencia de esta estructura.

Las siguientes pruebas mostrarán los resultados de la evaluación de los tests de triángulos, sin el paso previo de evaluación que determinará qué triángulos son los que presentan posibilidad de colisión. Para conseguir resultados similares a los reales en el proceso completo, se simularán las mismas condiciones que en éste.

Para obtener tiempos que sean deterministas, es decir, que no se vean influenciado por un pico de CPU o GPU por alguna tarea propia del sistema operativo, cada evaluación se ha realizado 10 veces, eliminando el tiempo más alto y el más bajo, y haciendo la media de los valores resultantes.

4.2.1. Resultados enviando vértices y aristas

En este caso, los tests estarán compuestos de la definición de los vértices y aristas de ambos triángulos a evaluar. Para ver el crecimiento del tiempo, se lanzarán ejecuciones con distintos tamaños del array. Además, para ver en qué se está empleando el tiempo, se mostrará también el tiempo de ejecución de la GPU, así como del tiempo destinado a la copia de los resultados desde la tarjeta gráfica. Los tiempos destinados para la copia del array hacia la tarjeta gráfica no se puede obtener, pues este es un proceso asíncrono y OpenCL no lo facilita.

A continuación se mostrarán los resultados obtenidos en ambas máquinas.

4.2.1.1. Máquina 1

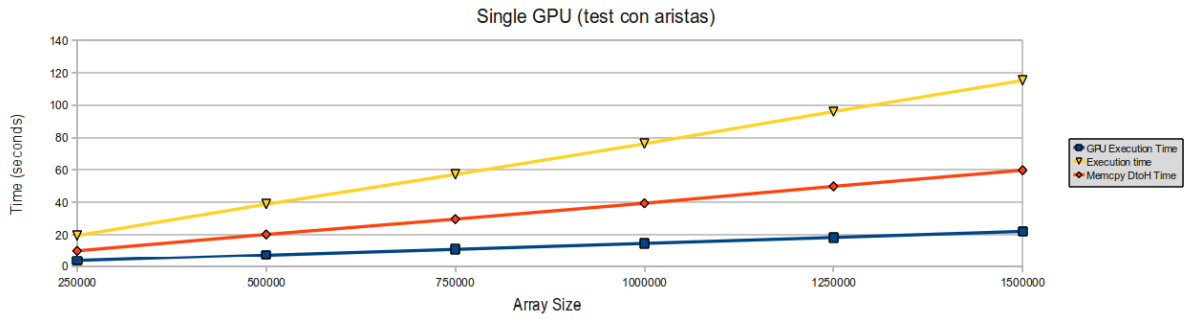

Figura 4.3: Resultados de la ejecución de tests de triángulos sobre una GPU en la máquina 1 (enviando el array con vértices y aristas)

Así pues, los tiempos destinados a cada tarea son:

Tamaño del array	Tiempo de ejecución total (s)	Tiempo de ejecución en GPU (s)	Copia del array (s)	% ejecución GPU	% copia del array
250000	19.53	3.67	10.11	18.79 %	51.76 %
500000	38.89	7.35	20.24	18.90 %	52.04 %
750000	57.26	11.03	29.70	19.26 %	51.87 %
1000000	76.19	14.71	39.48	19.30 %	51.81 %
1250000	96.07	18.39	49.91	19.14 %	51.95 %
1500000	115.32	22.14	59.89	19.20 %	51.93 %

Tabla 4.1: Tiempos destinados a la ejecución de la GPU y de la copia del array desde la tarjeta gráfica, para el caso del envío del array con aristas y vértices, usando únicamente una GPU en la máquina 1

En la siguiente gráfica se puede observar una comparativa entre los tiempos de ejecución de la CPU y los tiempos de ejecución de la GPU.

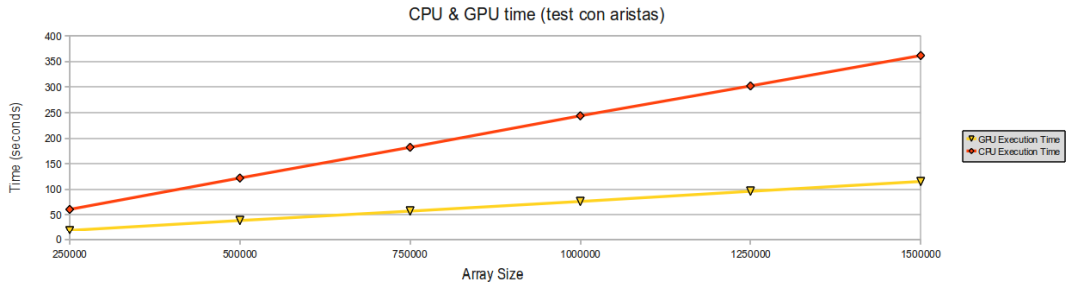

Figura 4.4: Comparativa de tiempos de ejecución entre, sólamente, la CPU e incorporando la GPU (enviando aristas y vértices)

La mejora obtenida con la incorporación del uso de la tarjeta gráfica es de $\left(\frac{361{,}78}{115{,}32} - 1\right) * 100 = 213{,}72\,\%$.

4.2.1.2. Máquina 2

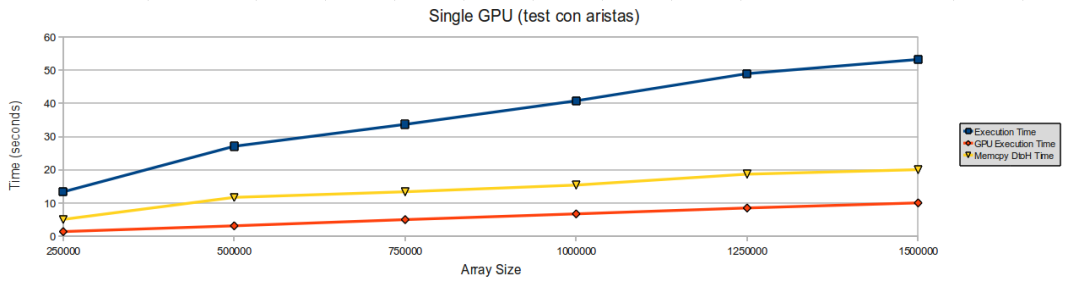

Figura 4.5: Resultados de la ejecución de tests de triángulos sobre una GPU en la máquina 2 (enviando el array con vértices y aristas)

Así pues, los tiempos destinados a cada tarea son:

Tamaño del array	Tiempo de ejecución total (s)	Tiempo de ejecución en GPU (s)	Copia del array (s)	% ejecución GPU	% copia del array
250000	13.39	1.43	5.10	10.68 %	38.09 %
500000	27.10	3.17	11.72	11.70 %	43.25 %
750000	33.69	5.03	13.41	14.93 %	39.80 %
1000000	40.79	6.75	15.42	16.55 %	37.80 %
1250000	48.96	8.53	18.72	17.42 %	38.23 %
1500000	53.23	10.06	20.07	18.9 %	37.70 %

Tabla 4.2: Tiempos destinados a la ejecución de la GPU y de la copia del array desde la tarjeta gráfica, para el caso del envío del array con aristas y vértices, usando únicamente una GPU en la máquina 2

El resto del tiempo es consumido por la CPU, ya sea para la construcción del array de tests, como para la copia del array hacia la gpu o las iteraciones que lanzarán las ejecuciones de las evaluaciones.

Además de los tiempos obtenidos tras la ejecución de los tests en el procesador de la tarjeta gráfica, en la siguiente gráfica se muestra una comparativa entre los tiempos de ejecución de los tests con ayuda de la tarjeta gráfica y sin ella. De este modo, se podrá observar la mejora obtenida al introducir este nuevo dispositivo hardware para las ejecuciones.

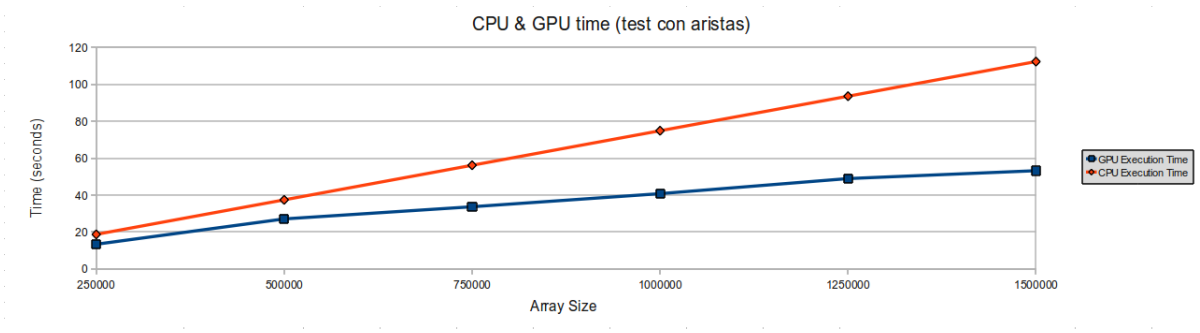

Figura 4.6: Comparativa de tiempos de ejecución entre, sólamente, la CPU e incorporando la GPU (enviando aristas y vértices)

En esta gráfica se puede ver la mejora al incoporar la tarjeta gráfica para la detección de las colisiones, obteniendo a tamaño máximo del array un $\left(\frac{112{,}31}{53{,}23} - 1\right) * 100 = 110{,}99\,\%$ de mejora.

Se puede ver que la mejora en porcentaje es menor, evidente ya que tiene un margen menor de mejora (diferencias de CPUs). Si se comparan las dos máquinas, se puede ver como el porcentaje de mejora en este caso son un $\left(\frac{213{,}72}{110{,}99}\right) * 100 = 192{,}55\,\%$ mejores.

4.2.2. Resultados enviando vértices

Para estas pruebas sólo serán enviados los vértices de ambos triángulos para la realización de los test. A continuación se pueden ver los resultados obtenidos con este método.

4.2.2.1. Máquina 1

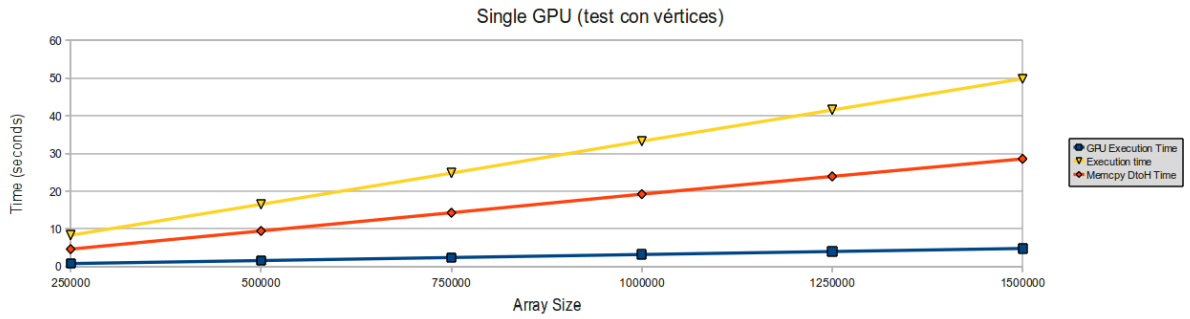

Figura 4.7: Resultados de la ejecución de tests de triángulos sobre una GPU en la máquina 2 (enviando el array con vértices únicamente)

Tamaño del array	Tiempo de ejecución total (s)	Tiempo de ejecución en GPU (s)	Copia del array (s)	% ejecución GPU	% copia del array
250000	8.33	0.82	4.65	9.84 %	55.82 %
500000	16.53	1.62	9.46	9.80 %	57.19 %
750000	24.80	2.43	14.29	9.80 %	57.62 %
1000000	33.28	3.22	19.22	9.67 %	57.75 %
1250000	41.53	4.02	23.93	9.68 %	57.62 %
1500000	49.84	4.83	28.58	9.69 %	57.34 %

Tabla 4.3: Tiempos destinados a la ejecución de la GPU y de la copia del array desde la tarjeta gráfica, para el caso del envío del array con vértices, usando únicamente una GPU en la máquina 1

A continuación se puede ver la mejora obtenida al introducir a la tarjeta gráfica como elemento de cálculo de las colisiones.

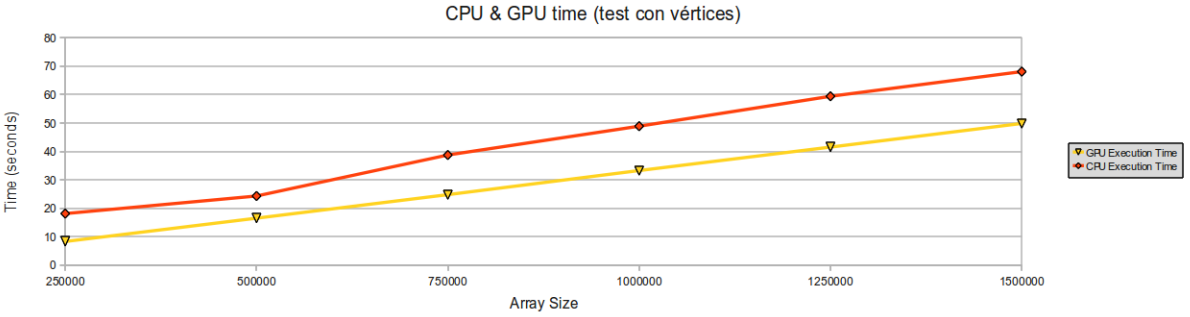

Figura 4.8: Comparativa de tiempos de ejecución entre, sólamente, la CPU e incorporando la GPU en la máquina 1(enviando vértices solamente)

El speedup obtenido, en el máximo tamaño del array, es $\left(\frac{68{,}08}{49{,}84} - 1\right) * 100 = 36{,}60\,\%$.

4.2.2.2. Máquina 2

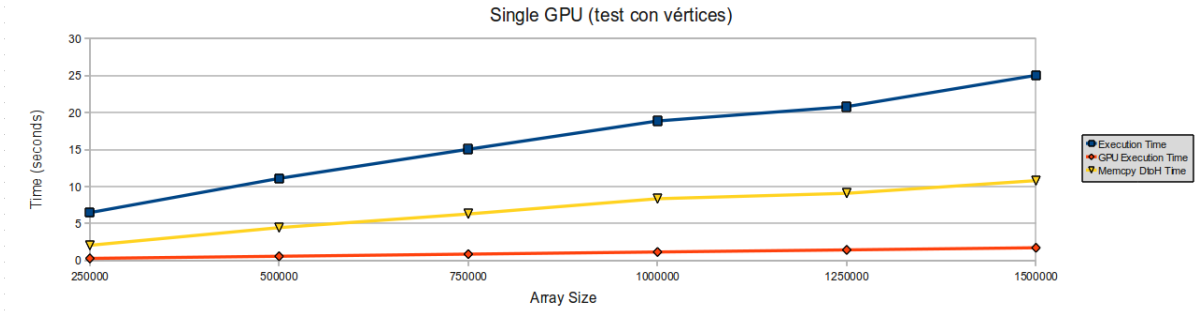

Figura 4.9: Resultados de la ejecución de tests de triángulos sobre una GPU en la máquina 2 (enviando el array con vértices únicamente)

Tamaño del array	Tiempo de ejecución total (s)	Tiempo de ejecución en GPU (s)	Copia del array (s)	% ejecución GPU	% copia del array
250000	6.48	0.29	2.06	4.47 %	31.79 %
500000	11.10	0.58	4.45	5.22 %	40.09 %
750000	15.04	0.87	6.23	5.78 %	41.42 %
1000000	18.86	1.16	8.35	6.15 %	44.27 %
1250000	20.80	1.44	9.09	6.92 %	43.70 %
1500000	25.03	1.73	10.80	6.91 %	43.15 %

Tabla 4.4: Tiempos destinados a la ejecución de la GPU y de la copia del array desde la tarjeta gráfica, para el caso del envío del array con vértices, usando únicamente una GPU en la máquina 2

En la siguiente gráfica se puede observar la relación de tiempos entre ejecutar, únicamente, con la CPU los tests y ejecutar incoporando la GPU para resolver las colisiones.

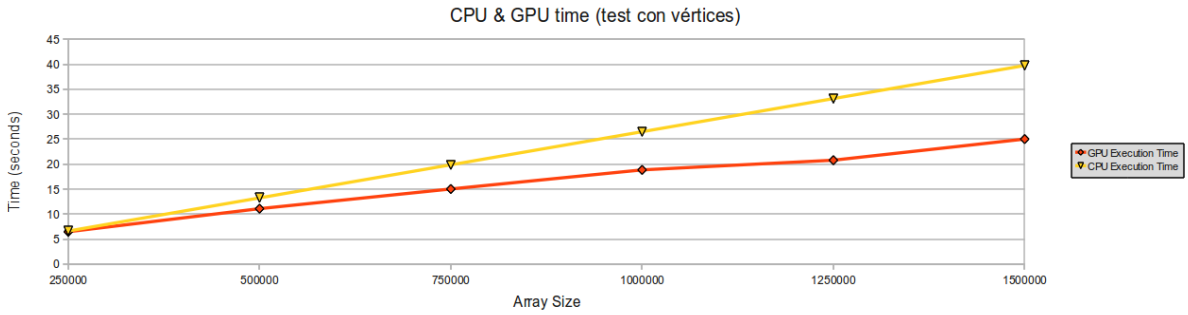

Figura 4.10: Comparativa de tiempos de ejecución entre, sólamente, la CPU e incorporando la GPU en la máquina 2(enviando vértices solamente)

La mejora obtenida con la incorporación de la tarjeta gráfica, en el mayor tamaño del array, es de un $\left(\frac{39{,}75}{25{,}03} - 1\right) * 100 = 58{,}81\,\%$

4.3. Resultados de tests de triángulos usando dos GPUs

En este caso, se realizará una ejecución en un entorno de varias GPUs, más concretamente dos. Como sólo se disponen de dos gráficas en la máquina 2, se mostrarán los resultados sobre esta máquina.

4.3.1. Resultados enviando vértices y aristas

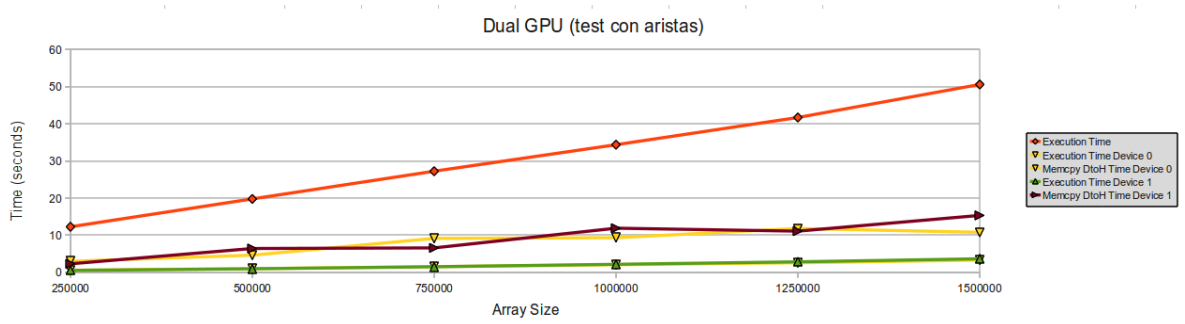

Figura 4.11: Resultados de la ejecución de tests de triángulos sobre dos GPUs (enviando el array con vértices y aristas)

Se puede observar como los tiempos son similares en ambas GPUs, tanto el tiempo de ejecución como el tiempo de copiado. Este comportamiento es el esperado, pues se manda a cada GPU exactamente el mismo número de tests a realizar.

4.3.2. Resultados enviando vértices

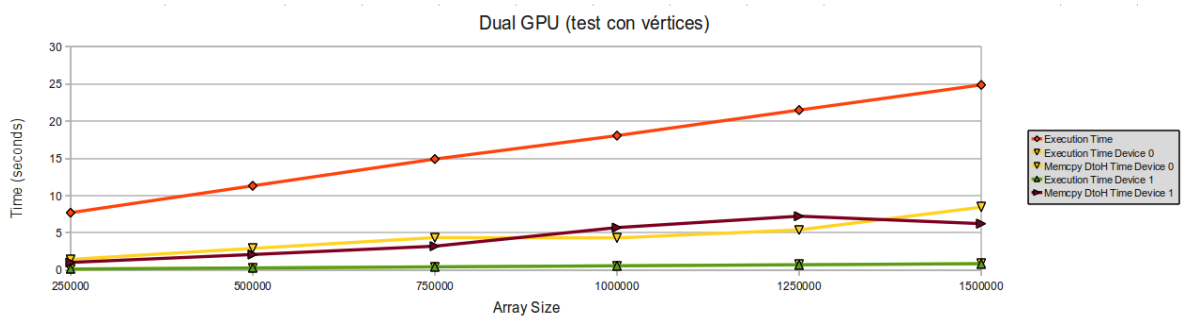

Figura 4.12: Resultados de la ejecución de tests de triángulos sobre dos GPUs (enviando el array con vértices)

De igual manera que en la gráfica del apartado anterior, se puede observar como los tiempos en ambas GPUs son similares en todos los pasos de la gráfica.

4.4. Resultados de la ejecución completa usando únicamente una GPU

A continuación se expondrán los resultados obtenidos tras la ejecución completa de la detección de las colisiones y autocolisiones. De igual modo que anteriormente, se realizarán ejecuciones con los dos métodos de detección, enviando sólo vértices o añadiendo también las aristas.

Las gráficas serán mostradas de manera separadas, pues los tiempos de ejecución en la GPU son demasiados pequeños como para poder diferenciar bien los resultados.

4.4.1. Resultados enviando vértices y aristas

4.4.1.1. Máquina 1

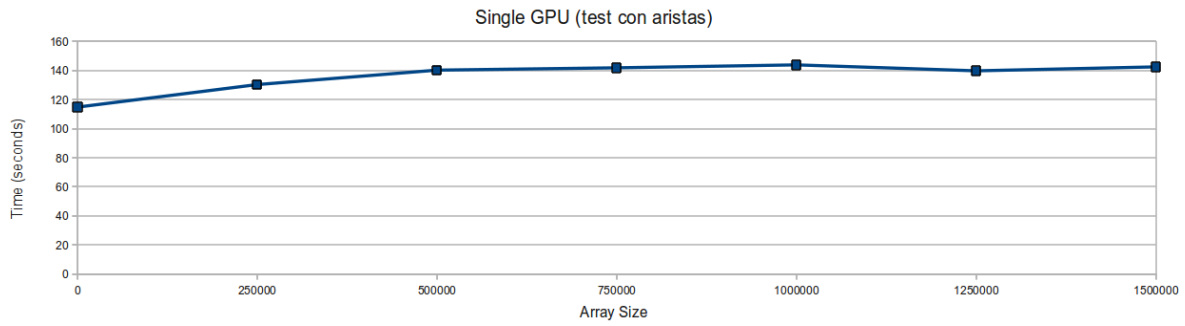

Figura 4.13: Resultados de la ejecución completa sobre una GPU en la máquina 1 (enviando el array con vértices y aristas)

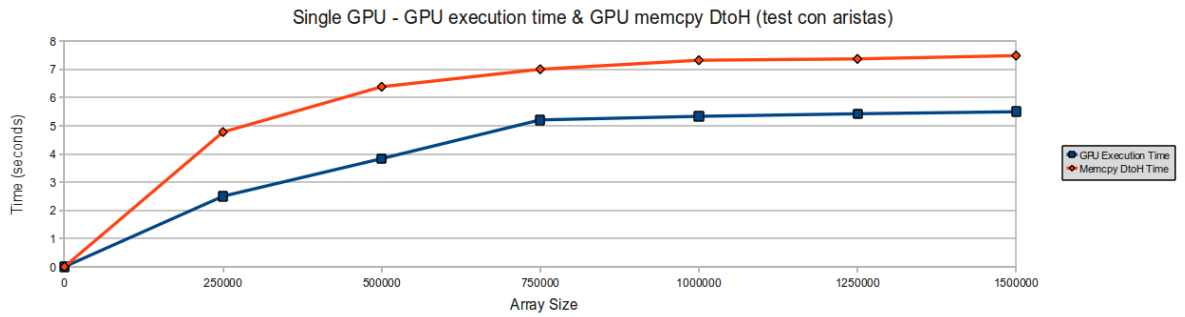

Figura 4.14: Resultados de la ejecución completa sobre una GPU en la máquina 1 (enviando el array con vértices y aristas)

Tamaño del array	Tiempo de ejecución total (s)	Tiempo de ejecución en GPU (s)	Copia del array (s)	% ejecución GPU	% copia del array
0	114.78625	0	0	0 %	0 %
250000	130.28	2.50	4.78	1.92 %	3.67 %
500000	140.22	3.83	6.38	2.73 %	4.55 %
750000	141.82	5.21	7.00	3.67 %	4.93 %
1000000	143.82	5.33	7.32	3.70 %	5.09 %
1250000	145.73	5.43	7.37	3.72 %	5.06 %
1500000	147.47	5.50	7.49	3.73 %	5.08 %

Tabla 4.5: Relación de tiempos en la ejecución completa de la animación en la máquina 1, usando únicamente una gráfica

4.4.1.2. Máquina 2

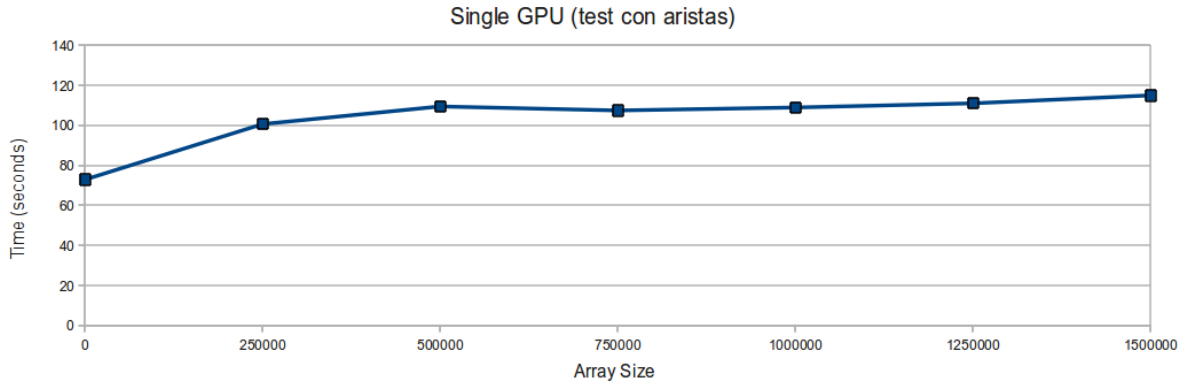

Figura 4.15: Resultados de la ejecución completa sobre una GPU en la máquina 2 (enviando el array con vértices y aristas)

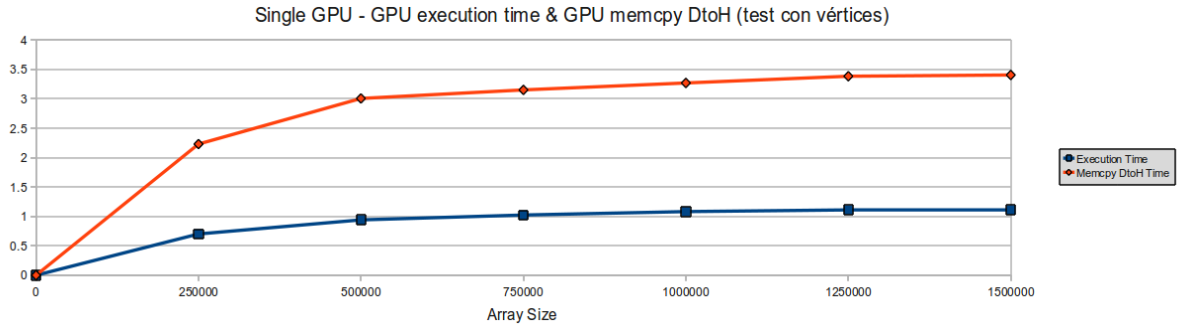

Figura 4.16: Resultados de la ejecución completa sobre una GPU en la máquina 2 (enviando el array con vértices y aristas)

Tamaño del array	Tiempo de ejecución total (s)	Tiempo de ejecución en GPU (s)	Copia del array (s)	% ejecución GPU	% copia del array
0	72.86	0	0	0 %	0 %
250000	100.61	0.70	2.23	0.69 %	2.21 %
500000	109.43	0.94	3.01	0.86 %	2.75 %
750000	107.43	1.02	3.15	0.95 %	2.93 %
1000000	108.92	1.08	3.27	0.99 %	3.00 %
1250000	111.01	1.11	3.38	1.00 %	3.04 %
1500000	114.98	1.13	3.40	0.98 %	2.95 %

Tabla 4.6: Relación de tiempos en la ejecución completa de la animación en la máquina 2, usando únicamente una gráfica

Como se puede apreciar, los tiempos de ejecución de la GPU solamente son, como máximo, de un 1 % del total del tiempo.

4.4.2. Resultados enviando vértices

4.4.2.1. Máquina 1

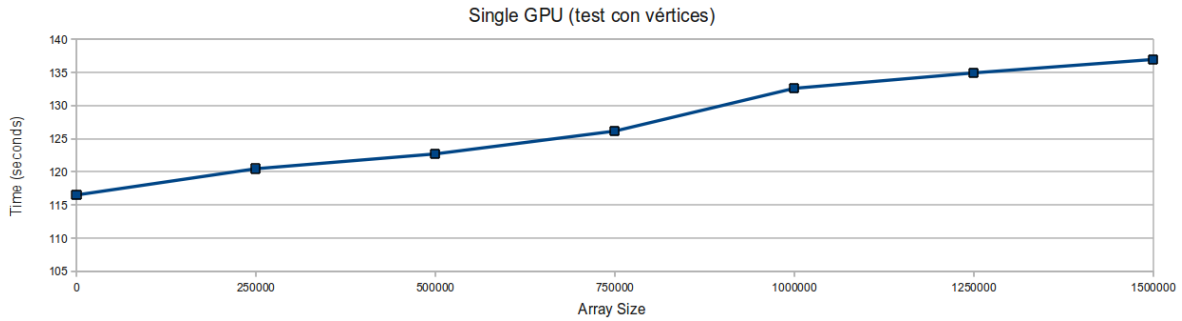

Figura 4.17: Resultados de la ejecución completa sobre una GPU en la máquina 1 (enviando el array con vértices)

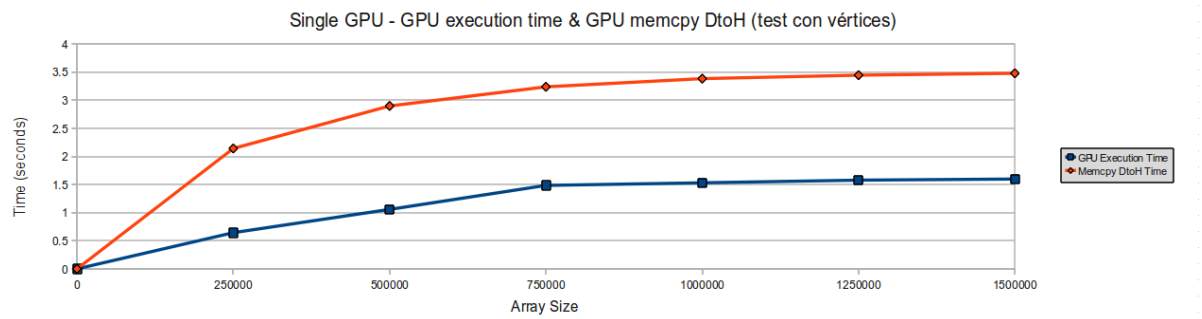

Figura 4.18: Resultados de la ejecución completa sobre una GPU en la máquina 1 (enviando el array con vértices)

Tamaño del array	Tiempo de ejecución total (s)	Tiempo de ejecución en GPU (s)	Copia del array (s)	% ejecución GPU	% copia del array
0	116.50	0	0	0 %	0 %
250000	120.46	0.64	2.14	0.53 %	1.77 %
500000	122.70	1.05	2.90	0.85 %	2.36 %
750000	126.14	1.48	3.24	1.17 %	2.57 %
1000000	132.60	1.53	3.38	1.15 %	2.55 %
1250000	134.93	1.58	3.45	1.17 %	2.55 %
1500000	136.96	1.60	3.48	1.17 %	2.54 %

Tabla 4.7: Relación de tiempos en la ejecución completa de la animación en la máquina 1, usando únicamente una GPU

Con este tipo de test se puede observar como los tiempos de ejecución son mínimos respecto al tiempo total.

4.4.2.2. Máquina 2

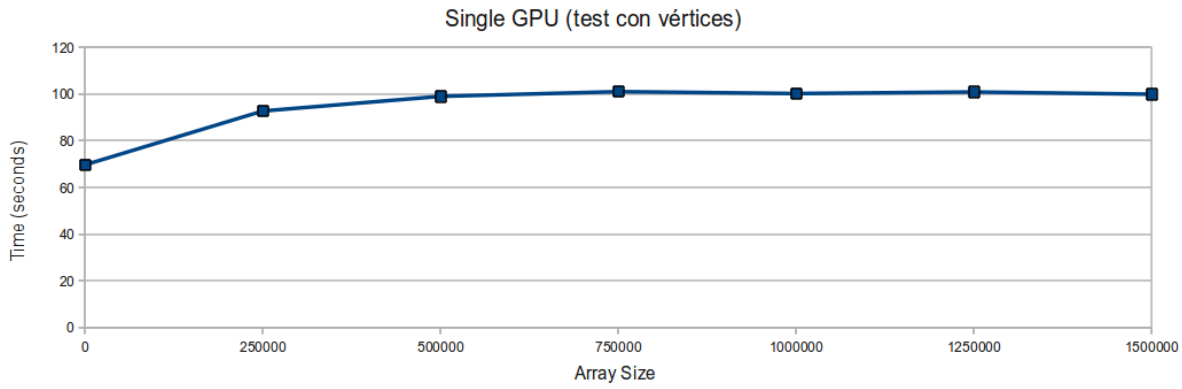

Figura 4.19: Resultados de la ejecución completa sobre una GPU en la máquina 2 (enviando el array con vértices)

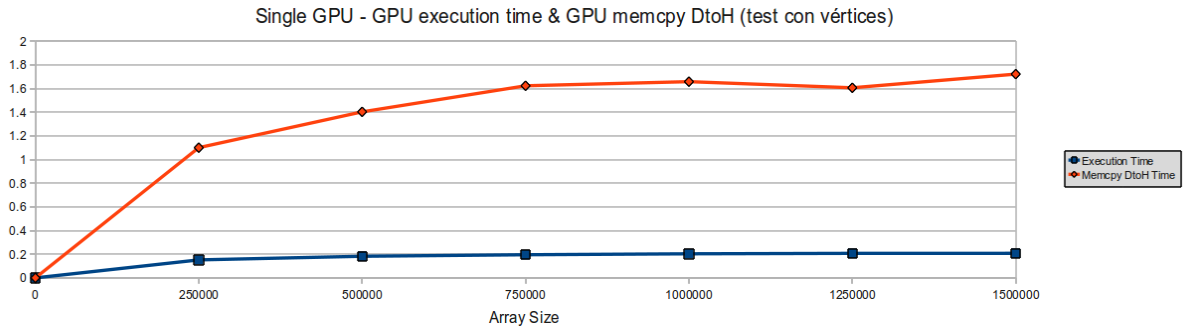

Figura 4.20: Resultados de la ejecución completa sobre una GPU en la máquina 2 (enviando el array con vértices)

Tamaño del array	Tiempo de ejecución total (s)	Tiempo de ejecución en GPU (s)	Copia del array (s)	% ejecución GPU	% copia del array
0	69.65	0	0	0 %	0 %
250000	92.75	0.15	1.10	0.16 %	1.18 %
500000	99.01	0.18	1.40	0.18 %	1.41 %
750000	101.04	0.19	1.62	0.19 %	1.60 %
1000000	100.23	0.20	1.65	0.20 %	1.64 %
1250000	100.89	0.21	1.60	0.21 %	1.58 %
1500000	99.92	0.22	1.72	0.22 %	1.72 %

Tabla 4.8: Relación de tiempos en la ejecución completa de la animación en la máquina 2, usando únicamente una GPU

En este caso, enviando únicamente los vértices como parte del test, los tiempos se reducen, llegando a tener menor porcentaje de ejecución que en el caso anterior.

4.5. Resultados de la ejecución completa usando dos GPUs

En este apartado se expondrán los resultados tras la ejecución sobre dos GPUs. Estas pruebas serán ejecutadas únicamente sobre la máquina 2, que es la que dispone de dos GPUs.

Puesto que los tiempos de ejecución total y el destinado únicamente para la ejecución dentro de la GPU son muy distintos, estos se mostrarán en dos gráficas.

4.5.1. Resultados enviando vértices y aristas

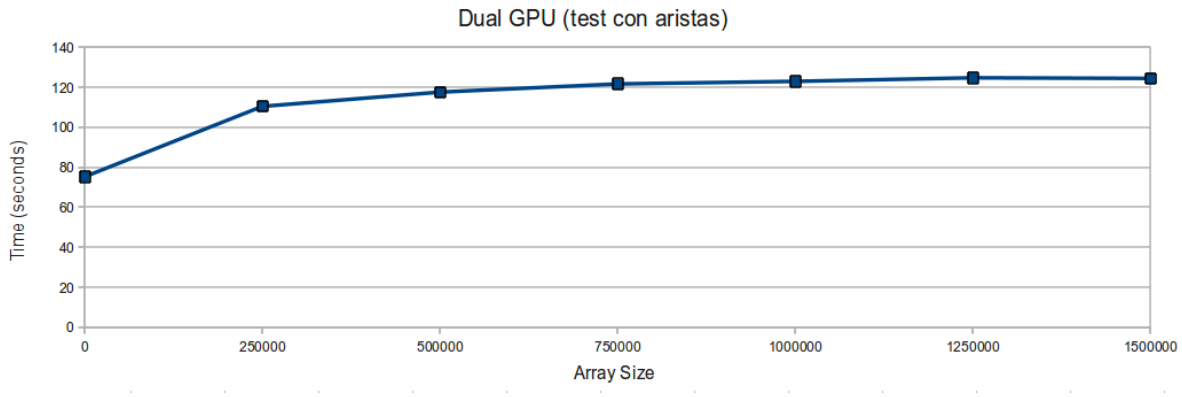

Figura 4.21: Resultados de la ejecución completa sobre dos GPUs (enviando el array con vértices)

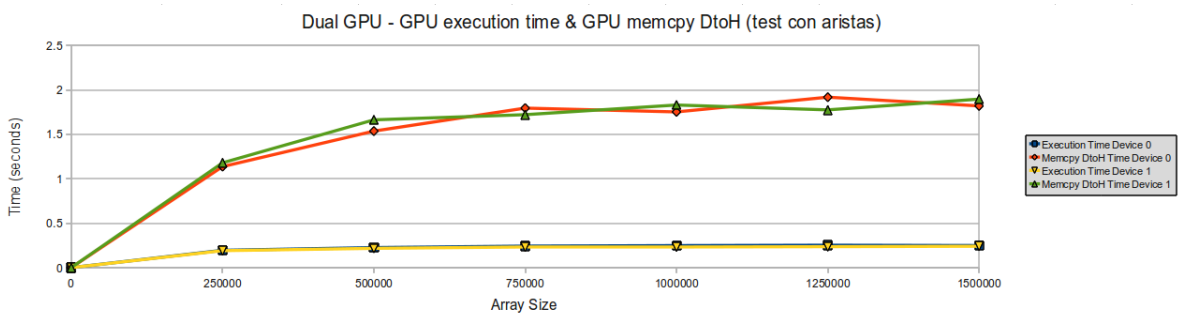

Figura 4.22: Tiempos de ejecución de las GPUs y tiempos de copiado del array desde la tarjeta gráfica (enviando el array con vértices)

Se puede observar como las ejecuciones en ambas GPUs están bien distribuidos, dado que a ambas tarjetas gráficas se las está enviando el mismo número de tests a evaluar.

4.5.2. Resultados enviando vértices

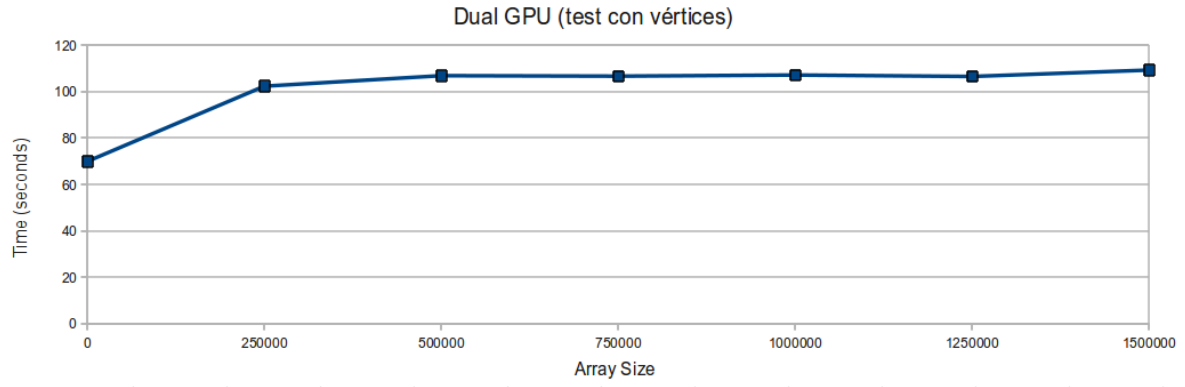

Figura 4.23: Resultados de la ejecución completa sobre dos GPUs (enviando el array con vértices)

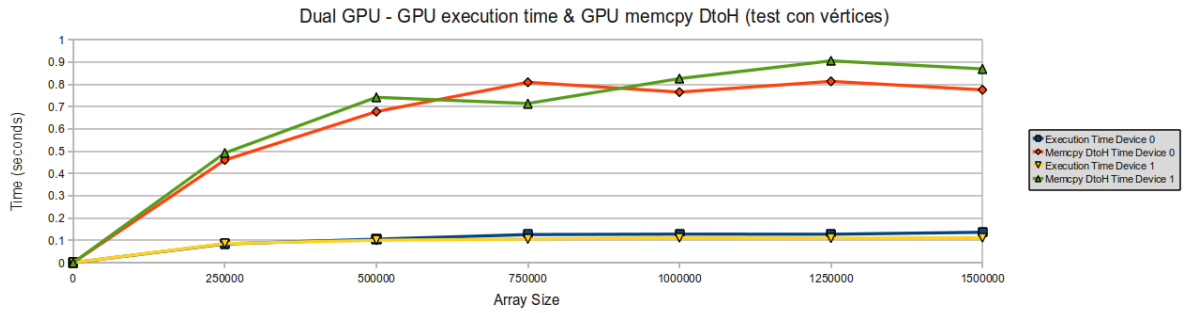

Figura 4.24: Tiempos de ejecución de las GPUs y tiempos de copiado del array desde la tarjeta gráfica (enviando el array con vértices)

4.6. Comparativa de resultados

En este apartado se mostrará la comparativa entre todos los tiempos recogidos, comparándolos también con los de la CPU.

4.6.1. Máquina 1

A continuación se puede ver en la gráfica la comparativa de todos los tiempos obtenidos en esta máquina.

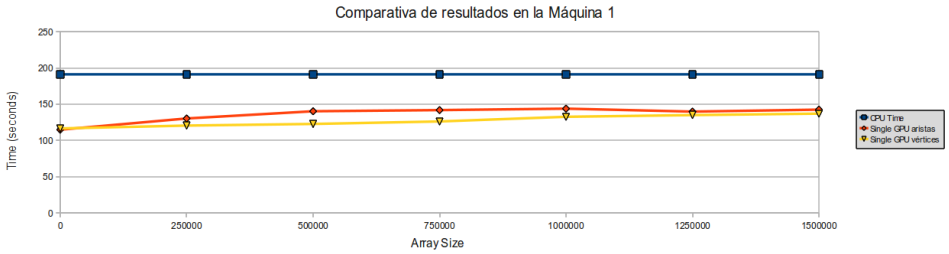

Figura 4.25: Comparativa entre los tiempos obtenidos en la máquina 1

Los speedup obtenidos, con un tiempo de CPU de 191.26 segundos, son:

	Tiempo	Speedup
Single GPU enviando aristas y vértices	142.47	$\left(\frac{191{,}26}{142{,}47} - 1\right) * 100 = 34{,}24\,\%$
Single GPU enviando vértices	136.96	$\left(\frac{191{,}26}{136{,}96} - 1\right) * 100 = 39{,}64\,\%$

Tabla 4.9: Speedups obtenidos en la máquina 2

Los speedups obtenidos son razonables, se ha conseguido reducir el tiempo de ejecución, obteniendo mejoras de un 35 %.

4.6.2. Máquina 2

En la siguiente gráfica se pueden ver todos los tiempos obtenidos.

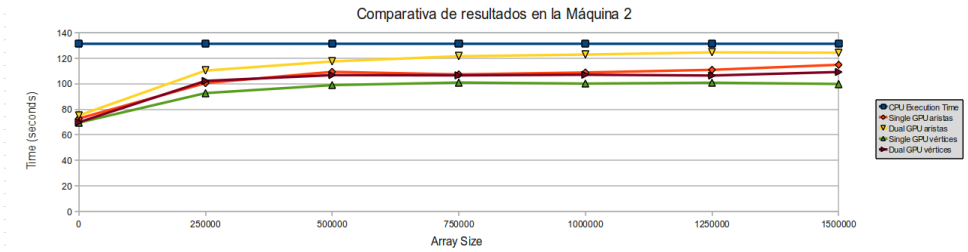

Figura 4.26: Comparativa entre los tiempos obtenidos en la máquina 2

Los speedup obtenidos son:

	Tiempo	Speedup
CPU	131.42	
Single GPU enviando aristas y vértices	114.98	$\left(\frac{131{,}42}{114{,}98} - 1\right) * 100 = 14{,}30\,\%$
Single GPU enviando vértices	99.92	$\left(\frac{131{,}42}{99{,}92} - 1\right) * 100 = 31{,}52\,\%$
Dual GPU enviando aristas y vértices	124.43	$\left(\frac{131{,}42}{124{,}43} - 1\right) * 100 = 5{,}61\,\%$
Dual GPU enviando vértices	109.29	$\left(\frac{131{,}42}{109{,}29} - 1\right) * 100 = 20{,}25\,\%$

Tabla 4.10: Speedups obtenidos en la máquina 2

Se puede observar como en todas las ejecuciones en las que se usa la GPU se obtienen mejores tiempos. Las mejoras no son tan altas como las esperadas, pues aun se sigue haciendo un alto uso de la CPU, lo que habría que remediar en trabajos futuros.

Además, se puede apreciar que los tiempos con varias GPUs son mayores que sus versiones con sólo una. Esto es debido a que el tiempo de uso de la GPU, para ejecutar, es pequeño respecto al tiempo que se tarda en copiar el array correspondiente a esa ejecución. En la siguiente gráfica, se puede observar cómo se ha obtenido una reducción en los tiempos, tanto en el de ejecución como en el del copiado de la estructura desde la tarjeta gráfica.

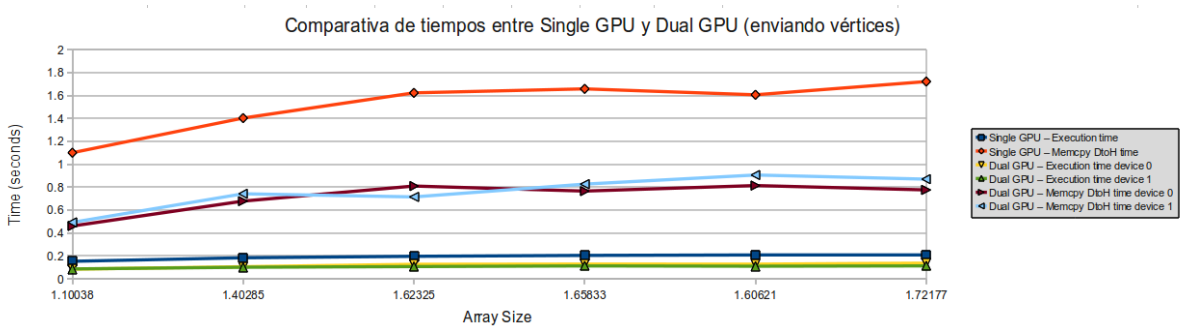

Figura 4.27: Comparativa entre los tiempos single GPU y dual GPU (enviando aristas y vértices)

En el caso del tiempo de ejecución en la GPU, se está obteniendo un speedup de $\left(\frac{0{,}21}{0{,}13} - 1\right) * 100 = 61{,}54\,\%$. Para el caso de los copiados de la estructura, el speedup es de $\left(\frac{1{,}72}{0{,}87} - 1\right) * 100 = 97{,}70\,\%$

Capítulo 5

Conclusiones y trabajos futuros

En capítulos anteriores se ha descrito el problema abordado en el proyecto, los objetivos y la solución propuesta. En este capítulo se expone una conclusión global del trabajo realizado y se proponen un conjunto de trabajos futuros a partir de este trabajo.

5.1. Conclusiones

El objetivo de este proyecto era la optimización del tiempo empleado en la detección de colisiones y autocolisiones sobre modelos de objetos 3D, incorporando un nuevo elemento hardware para el cálculo de éstas. Este dispositivo hardware es la tarjeta gráfica, caracterizada por su gran nivel de paralelización. Gracias a esta característica, se ha conseguido una mejora en el tiempo global de ejecución de este problema.

Los resultados obtenidos empleando una sola GPU y empleando varias, los tiempos resultantes no están siendo los óptimos. Si se hace uso de varias GPUs, lo esperado es que el tiempo bajara, pero en este caso, al emplear la tarjeta gráfica para la evaluación de un test tan poco costoso, no se está aprovechando al máximo su capacidad. Así pues, se está perdiendo tiempo en las copias de los datos hacia/desde la tarjeta gráfica. En la mayoría de los casos de prueba realizados, el tiempo de ejecución en la GPU es de menos del 50 % del tiempo de copiado, lo que supone un desaprovechamiento de este tiempo invertido.

5.2. Trabajos futuros

Como trabajo futuro se podría seguir reduciendo el uso de la CPU. Puesto que la tarjeta gráfica nos proporciona una de una gran paralelización, habría que seguir aprovechándolo, buscando tareas que fueran fácilmente paralelizables. Por ejemplo, una

tarea que podría ser realizada en la tarjeta gráfica, sería el cálculo de los volúmenes envolventes, que consume un 8.77 % del tiempo total. Además, esto se realiza cada vez que se carga un nuevo frame, por lo que es algo repetitible en el tiempo.

Un trabajo futuro a corto plazo podría ser probar esta implementación con animaciones más pesadas, comprobando si así se consigue un carga considerable sobre la GPU. Además de esto, sería buena opción probar si es mejor enviar todo el array entero a la GPU o, por el contrario, enviar porciones de éste.

Otra posible mejora que podría ser incorporada, sería la paralelización, mediante otro tipo de arquitecturas, de la parte que se está realizando en la CPU.

Bibliografía

[1] http://es.wikipedia.com.

[2] http://www.nvidia.es/object/product-geforce-gtx-590-es.html.

[3] http://ark.intel.com/products/63696/Intel-Core-i7-3960X-Processor-Extreme-Edition-(15M-Cache-3_30-GHz).

[4] http://www.khronos.org/assets/uploads/developers/library/overview/opencl-overview.pdf.

[5] G. Barequet, B. Chazelle, L. Guibas, J. Mitchell, and A. Tal. Boxtree: A hierarchical representation of surfaces in 3d. *Proceedings of Eurographics'96*, 1996.

[6] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. *In Proceedings of ACM SIGMOD Conf. On Management of Data*, pages 322–331, 1990.

[7] Van Der Bengen. Efficient collision detection of complex deformable models using aabb trees. *Journal of Graphics Tools*, 1997.

[8] G. Bradshaw and C. O'sullivan. Adaptative medial-axis approximation for sphere-tree construction. *ACM Transaction on Graphics, Vol. 23, num. 1*, pages 1–26, 2004.

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

[10] J. Gascón Perez. Self-collision detection for reduced deformation models. *Proyecto fin de máster*, 2009.

[11] S. Gottschalk. Separating axis theorem. In *Technical Report TR96-024*, Dept. of Computer Science, University of North Carolina Chapel Hill, 1996.

[12] S. Gottschalk, M. Lin, and D. Manocha. Obb-tree: A hierarchical structure for rapid interference detection. *Proceedings SIGGRAPH'96*, 1996.

[13] F. He. Fast collision detection using quospo trees. *Proceedings of then 1999 Symposium on Interactive 3D Graphics*, pages 55–62, 1999.

[14] P. M. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Transaction on Graphics, Vol. 15, num. 3*, pages 179–210, 1996.

[15] Philip M. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Trans. Graph.*, 1996.

[16] J. J Jiménez Delgado. Detección de colisiones mediante recubrimientos simpliciales. *Tesis doctoral*, 2006.

[17] D. Kanter. Nvidia's gt200: Inside a parallel processor. 2008.

[18] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE transactions on Visualization and Computer Graphics*, 1998.

[19] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics, Vol. 4, num. 1*, pages 21–36, 1998.

[20] Petr. Konecný and Karel Zikan. Lower bound of distance in 3d. *In Proceedings of WSCG*, volume 3, pages 640–649, 1997.

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

[22] A. Pinzón Fernández. Sistema de simulación de colisiones deformables con interacción mediante dispositivos hápticos en ambientes de realidad aumentada: estado del arte. *Universidad Nacional de Colombia.*

[23] P. Volino and N. Magnenat-Thalmann. Efficient self-collision detection on smoothly discretized surface animation using geometrical shape regularity. *Eurographics*, 1994.

[24] F. Wang, E. Burdet, A Dhanik, T Poston, and C. L. Teo. Dynamic thread for real-time knot-tying.

[25] G. Zachmann. Rapid collision detection by dinamically aligned dop-trees. *Proceedings of IEEE Virtual Reality Annual International Symposium VRAIS'98*, pages 90–97, 1998.