VNIVERSIDAD NACIONAL
AVTONOMA DE
MEXICO

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Posgrado en Ciencia e Ingeniería de la Computación

IMPLEMENTACIÓN DE TÉCNICAS DE SIMULACIÓN POR COMPUTADORA EN GPU

TESIS
QUE PARA OPTAR POR EL GRADO DE:
MAESTRO EN CIENCIA (COMPUTACIÓN)

PRESENTA:
DAVID ARTURO SORIANO VALDEZ

TUTORES PRINCIPALES

DR. FERNANDO ARÁMBULA COSÍO	**CCADET**
DR. MIGUEL ÁNGEL PADILLA CASTAÑEDA	**CCADET**

MÉXICO, D. F. Enero 2016

Agradecimientos

El trabajo que se presenta a continuación fue realizado gracias al apoyo de varias personas e instituciones, por lo que aprovecho este espacio para agradecer el apoyo que se me ha brindado.

Agradezco enormemente a mis tutores Dr. Fernando Arámbula Cosío, Dr. Miguel Ángel Padilla Castañeda y Dr. Alfonso Gastélum Strozzi, sin su orientación, ideas y paciencia este trabajo no habría sido posible; espero seguir contando con su apoyo en el futuro. Aprovecho para agradecer el conocimiento y experiencia que compartieron conmigo, así como su apoyo para realizar este trabajo y una estancia de investigación.

Agradezco al posgrado en ciencias e ingeniería de la computación por el espacio que me brindo a fin de poder recibir la formación académica que me ha permitido realizar este trabajo, es un honor ser alumno de este posgrado. Aprovecho para agradecer a los profesores del posgrado y personal que labora en dicho posgrado, sin ustedes el posgrado no sería posible. Quiero agradecer también el apoyo económico que me brindo el posgrado a fin de realizar una estancia de investigación.

Agradezco al CCADET-UNAM y Hospital General de México por brindarme un espacio dentro de sus instalaciones en el cual pude trabajar en este proyecto durante este tiempo.

Agradezco al Laboratorio de imagenología biomédica, física y computacional así como a la Unidad de investigación y desarrollo tecnológico por dejarme ser parte del grupo de trabajo que conforman.

Agradezco al CONACYT por el apoyo económico que me brindo durante este tiempo.

Agradezco a la Dra. Kanako Harada y la Universidad de Tokyo por permitirme realizar una estancia de investigación.

Agradezco a mi familia por el apoyo recibido durante toda mi vida.

Resumen

Las simulaciones por computadora son herramientas cada vez más empleadas en la industria e investigación, estas herramientas permiten modelar una gran cantidad de sistemas con los que se provee de medios para analizar fenómenos y/o adquirir habilidades sin la necesidad de realizar el experimento o actividad de manera tradicional. Una de las principales ventajas de emplear simulaciones frente a la reproducción de un experimento, fenómeno o actividad radica en la reducción de costos y riesgos que conlleva la realización del experimento de manera habitual: en el caso del entrenamiento de cirujanos se evita el riesgo que conlleva la práctica de la cirugía *in vivo*.

El desarrollo de la simulación por computadora va de la mano del desarrollo del poder de cómputo, es decir que la complejidad de las simulaciones se ha incrementado conforme a los recursos de cómputo disponibles se han incrementado. Hoy en día contamos con recursos de cómputo masivo como son las unidades de procesamiento gráfico (Graphics Processor Unit o GPU) que brindan una arquitectura enfocada en el cómputo masivo en paralelo, por lo que a fin de aprovecharlos es necesario implementar los métodos de simulación empleando modelos de programación en paralelo. Dicha implementación no es trivial, pero en la actualidad se cuentan con herramientas como CUDA, que es la arquitectura de computo en paralelo desarrollada por la compañía NVIDIA, la cual permite realizar dicha implementación en GPU; sin embargo, la tarea de modificar el método de simulación a fin de que aproveche las capacidades de cómputo masivo en paralelo que ofrecen las GPU sigue recayendo completamente en el programador. Dado lo anterior es que muchas veces se opta por tomar algún motor de simulación que ya aprovecha el poder de cómputo de las GPU y se extiende su funcionalidad a fin de resolver las necesidades que se requieren cubrir para el sistema que se desea simular.

En este trabajo presentamos un breve análisis de los métodos de simulación que permiten simular cuerpos deformables, a partir de ellos se hace la elección de un método libre de mallas para agregar la funcionalidad de realizar cortes en cuerpos deformables, cuya construcción parte de mallas tetraédricas. Se presentan las modificaciones realizadas al método de simulación de dinámica basada en posición a fin de poder modelar volúmenes compuestos de tetraedros con la estructura conocida como *cloth* que ofrece Flex, el cual es una biblioteca de simulación basada en partículas, así como el método para realizar cortes en estos cuerpos.

En este trabajo se empleó OpenGL, el cual es la interfaz de programación de aplicaciones (API por sus siglas en inglés) para realizar despliegue gráfico en 2D y 3D. Dado lo anterior es que se hace mención de algunas técnicas que aprovechan el poder de computo de las GPU, como la interoperabilidad de CUDA y OpenGL, estas técnicas se deben implementar a fin de no afectar el desempeño en la simulación debido a los cuellos de botella que conlleva el trabajar con el cómputo acelerado mediante GPU, sobre todo cuando se trabaja con aplicaciones que requieren despliegue gráfico.

Por último se presenta el análisis del desempeño de la implementación realizada, con lo que se aprecian las capacidades de la implementación del método en GPU y las capacidades de escalabilidad que posee.

Índice

Índice de figuras

Índice de tablas

Capítulo 1. Introducción.

1.1. Simulación por computadora.

Una simulación es la abstracción de un sistema o fenómeno aplicando modelos predeterminados, Shanon [5] define a la simulación como el proceso de diseñar un modelo de un sistema real y llevar a término experiencias con él, con la finalidad de comprender el comportamiento del sistema o evaluar nuevas estrategias para el funcionamiento del sistema.

La simulación por computadora es la aplicación de modelos, los cuales son ejecutados en un programa de computadora con el fin de que éste haga el análisis de los fenómenos o sistemas que se han modelado. Esta área de la computación ha crecido enormemente desde sus inicios, esto no solo se debe a las ventajas que representa para las personas el poder modelar los sistemas que estudian, sino que también la cantidad de recursos de cómputo disponibles para ejecutar las simulaciones han crecido enormemente. En la actualidad los simuladores no solo proporcionan información en forma de resultados, sino que son capaces de evaluar situaciones en tiempo real y proveer una respuesta a los escenarios (acciones o variación en parámetros propios de la simulación) que son introducidas mientras se ejecuta la simulación. Algunos simuladores incluso cuentan con sistemas de realidad virtual o ambientes virtuales sobre los cuales se desarrolla la simulación. Estos últimos tienen la característica de que buscan centrar la simulación con base en la intervención de un usuario.

En la actualidad las simulaciones por computadora forman parte integral de muchas áreas de la ciencia. Existen simuladores para entrenamiento, como son los simuladores de vuelo; existen otros para recrear fenómenos de la naturaleza y analizar los elementos que componen dichos fenómenos, por ejemplo, los simuladores de flujo de agua en poros del suelo como los que se muestra en el trabajo de Gastelum [19]. Un área donde los simuladores han tomado relevancia en los últimos años es en la medicina, un ejemplo de ellos son los simuladores quirúrgicos para el entrenamiento de cirujanos empleando cirugía robótica [20].

Algunas de las razones por las que los simuladores quirúrgicos han crecido en popularidad es porque mediante el uso de dichos simuladores se pueden representar situaciones reales, sin la necesidad de exponer la integridad de algún ser vivo para estudiar algún procedimiento. Aunque en la actualidad existen maniquíes de modelos anatomicos los cuales son empleados con el mismo propósito, estos son bastante costosos y en algunos casos requieren de mantenimiento el cual no siempre es accesible o económico.

Lo anterior es importante considerando que los simuladores se emplean como herramienta de entrenamiento o para la (re)adquisición de habilidades y destrezas. Un ejemplo de ello es el simulador de resección transuretral de próstata (Simulador de RTUP) que se desarrolla en el CCADET-UNAM [2].

Hoy en día contamos con grandes recursos de cómputo, lo que marca una gran diferencia con las primeras simulaciones por computadora, ya que gracias a eso podemos crear simulaciones con mayor resolución, más rápidas, con modelos más complejos y que procesen una cantidad mayor de datos al momento de realizar la simulación. Un ejemplo de estos recursos disponibles son las unidades de procesamiento gráfico o GPU (Graphics Processor Unit).

En el momento que las GPU fueron introducidas al mercado con la capacidad de realizar cómputo masivo de manera paralela a principios del año 2000, mediante el uso de pequeños programas llamados *shaders*, lo cual ha promovido un cambio en el paradigma de cómputo usado para las simulaciones. Años después gracias al desarrollo de lenguajes de programación que trabajan directamente sobre la GPU es que tenemos cálculo acelerado en GPU o aceleración por GPU.

La aceleración por GPU va cobrando mayor importancia en la actualidad ya que proporciona una arquitectura de procesamiento en paralelo la cual es escalable a medida que la tecnología avanza. Esto ha tenido como resultado que hoy en día exista una gran cantidad de áreas de la computación que busquen beneficiarse al aprovechar dichas arquitecturas, además de que se trabaja en el desarrollo de plataformas que aprovechen dichas arquitecturas como lo es CUDA[1] de NVIDIA[2] y OpenCL de Khronos[3].

En este trabajo nos enfocamos en métodos de simulación que permiten realizar la implementación de simuladores quirúrgicos, por lo que los métodos de simulación deben ser capaces de modelar el comportamiento físico de sistemas biológicos que tienen interacción con acciones controladas por un usuario en tiempo real.

1.2. Simuladores quirúrgicos.

Como ya fue mencionando anteriormente, los simuladores quirúrgicos han tomado gran relevancia gracias a que proporcionan un ambiente sintético, generalmente mediante ambientes virtuales, donde es posible que un usuario adquiera habilidades y destrezas sin el riesgo que representa el ser expuesto a una situación de cirugía real, además se reduce el riesgo que representa al paciente ya que en la actualidad la principal manera de entrenamiento para cirujanos se lleva acabo "in vivo" con pacientes reales aunque se procura sea en situaciones controladas. Aunado a lo anterior, los simuladores no limitan la cantidad de tiempo que un usuario puede practicar en ellos a fin de perfeccionar alguna habilidad.

Sin embargo, este tipo de simuladores agrega el factor de interacción humana, lo cual implica el replanteamiento del problema inicial que resuelven los simuladores, pues no solo se debe modelar el sistema sino que se debe tomar en cuenta la retroalimentación que la simulación proporciona al usuario a cada una de las acciones que el usuario realice.

[1] http://www.nvidia.com/object/cuda_home_new.html
[2] http://www.nvidia.com/content/global/global.php
[3] https://www.khronos.org/

Uno de los primeros puntos que debemos considerar es la información que es retroalimentada al usuario durante la simulación, para los simuladores quirúrgicos hay que poner especial atención en la información visual que el usuario recibe, es decir que el ambiente virtual que representa la situación de la vida real debe ser modelada y esta debe comportarse lo más cercana a la realidad. Si nos enfocamos en el aspecto de como percibimos la realidad mediante el sentido de la vista, entonces es indispensable que este tipo de simuladores posean modelos gráficos, texturas y tiempos de respuesta realistas.

A pesar de que los recursos de cómputo disponibles se han incrementado gracias a los avances tecnológicos no es posible alcanzar un realismo visual semejante al obtenido mediante foto realismo, lo que conlleva a un compromiso con los diferentes elementos de la simulación a fin de obtener un producto que pueda ser empleado como herramienta de entrenamiento. Dado lo anterior podemos asumir que los modelos con los que se representa el sistema son acotados de acuerdo a los aspectos que se tenga mayor interés en simular, por lo que nos enfocamos en los aspectos más importantes para el entrenamiento del usuario.

1.3. Estado del arte.

Uno de los desafíos actuales de la simulación aplicada consiste en el desarrollo de simuladores de procesos centrados en humanos [1], como lo son los simuladores quirúrgicos. Estos simuladores representan un reto ya que la simulación por computadora debe ser capaz de reproducir el fenómeno que se está simulando, tomando en cuenta los eventos aleatorios que se producen por parte de agentes externos que interactúan con el modelo de la simulación. Además, este proceso tiene que ser ejecutado en la menor cantidad de tiempo posible a fin de no afectar la experiencia que recibe el usuario. En otras palabras la simulación debe producir datos de manera constante en periodos acotados de tiempo, dichos periodos deben tener la duración adecuada a fin de que la información desplegada al usuario tenga la resolución necesaria para la reconstrucción del fenómeno de manera que el usuario lo aprecie de manera natural: típicamente la simulación debe ejecutarse a una frecuencia de [30-60] Hz para el despliegue gráfico, mientras que para el despliegue háptico de sensaciones táctiles entre [100-300] Hz.

Hoy en día los simuladores quirúrgicos utilizan técnicas de simulación de la dinámica de cuerpos rígidos y cuerpos deformables. La simulación de cuerpos rígidos es un problema que ha sido estudiado ampliamente y para el cual existen diversos métodos para su implementación, sin embargo la simulación de cuerpos suaves es más compleja y la adopción de los diferentes métodos existentes aún no está estandarizada, por lo que la implementación de algoritmos que modelen su comportamiento aún depende en gran medida del sistema que se desea modelar.

Existen dos técnicas bastante aceptadas en el modelado de cuerpos deformables, los métodos de modelado con mallas y lo métodos libres de mallas. Los métodos de modelado con mallas son implementados mediante métodos de masas y resortes [21] o mediante elementos finitos [6], mientras que los métodos libres de mallas emplean modelos de partículas suavizadas o SPH por sus siglas en inglés (Smoothed Particle Hydrodynamics) [17]. Ambos métodos permiten simular cuerpos deformables, pero existen grandes diferencias entre ellos, las cuales hacen que dependiendo del problema a modelar alguno de los dos provea una mejor solución con respecto al otro.

En la actualidad para obtener un mejor rendimiento de las simulaciones se aprovechan las arquitecturas de procesamiento en paralelo como son las GPU, aunque en el caso de los simuladores *que dependen de la interacción del usuario y requieren una respuesta visual en tiempo real* aún existe poco trabajo reportado en la literatura, pues la paralelización de los métodos de simulación, los cuales son complejos, empleando procesamiento en paralelo no es una tarea trivial y si no se realiza de manera adecuada es posible que no se obtenga beneficio de dicha implementación. Lo anterior se debe a que las GPU son habitualmente las encargadas de realizar el despliegue visual mediante el uso de *shaders* y motores de graficación.

A fin de simplificar la tarea de construir un simulador se recurre a motores de simulación física, por ejemplo NVIDIA PhysX[4], Bullet Physics[5] y SOFA[6] entre otros; los cuales por lo general corren en CPU pero aprovechan la aceleración mediante GPU. Esto proporciona herramientas muy útiles para la construcción de simuladores, sin embargo mantiene separados el motor de simulación física del motor de graficación, a pesar de que ambos realizan la mayoría de sus cálculos en la misma unidad de procesamiento. Esto tiene como consecuencia que se tengan diversos cuellos de botella, los cuales afectan el desempeño al momento de la ejecución de los simuladores.

Una de las razones por las cuales se da esta separación es porque los motores están concebidos solo como herramientas de procesamiento y no necesariamente deben tener una salida visual. Dado lo anterior es que una implementación de un simulador el cual implemente un motor de simulación física con comunicación directa con el motor gráfico significa un gran avance y un mejor aprovechamiento de la aceleración por GPU.

Dado lo anterior es que se deben aprovechar las técnicas de graficación más recientes, las cuales emplean la memoria con las que cuentan las GPU y que realizan el despliegue empleando *shaders* los cuales procesan en paralelo todos los elementos que se despliegan gráficamente. Al mismo tiempo se debe elegir una plataforma de cómputo en paralelo que permita la implementación de un motor de simulación física en GPU.

1.4. Objetivo.

El objetivo de esta tesis fue implementar técnicas de simulación que permitan en trabajo futuro la implementación de un simulador de RTUP, es decir una simulación de cuerpos deformables y rígidos, que aproveche el poder de las GPU, no solo para el renderizado del ambiente virtual, sino para la realización de los cálculos que conlleva la simulación física de los tejidos y elementos que interactúan. Para poder aprovechar el poder de cómputo de las GPU fue necesario recurrir a una plataforma de procesamiento en paralelo en GPU con la que se implementó la aceleración por GPU.

[4] https://developer.nvidia.com/gameworks-physx-overview
[5] http://bulletphysics.org/wordpress/
[6] https://www.sofa-framework.org/

En esta tesis se trabajó con tarjetas gráficas de NVIDIA, por lo que se emplearon los ambientes de desarrollo CUDA y Flex para la implementación del procesamiento en paralelo y motor de simulación física en GPU. Para el despliegue de gráficos se empleó OpenGL[7] 4.4 ya que es capaz de trabajar con *shaders* que se ejecutan directamente en la GPU empleando la memoria de la misma.

Dentro de los resultados que se presentan se tiene una implementación de volúmenes generados con tetraedros de cuerpos deformables empelando Flex, a estos modelos se les pueden realizar cortes mediante la implementación de un método de corte que se definió para modelos de partículas. Con dicha implementación se realizó el análisis de desempeño en GPU empleando el IDE Microsoft Visual Studio. Este análisis de desempeño se ejecutó para diversas pruebas a fin de comparar las capacidades que ofrece la plataforma CUDA.

De esta forma, esta tesis está organizada de la siguiente manera: En el capítulo 2 se presenta un método de simulación basado en mallas y un método de simulación libre de mallas, se hace una breve comparación a fin de ilustrar por qué se decidió trabajar con métodos libres de mallas. En el capítulo 3 se describe la arquitectura de las GPU y CUDA con el fin de entrar en detalle en el motor de simulación. En el capítulo 4 se detalla el método propuesto para la simulación junto con la descripción del método para realizar cortes en los cuerpos deformables implementados con el método propuesto. En el capítulo 5 se muestran los experimentos y resultados obtenidos. En el capítulo 6 se abordan las conclusiones y trabajo a futuro.

[7] https://www.opengl.org/

Capítulo 2. Simulación por computadora.

La simulación por computadora se lleva a cabo mediante la definición de modelos de sistemas estáticos o dinámicos expresados en variables de estados, dependiendo del objeto de interés es que se modela el sistema del fenómeno de estudio, en este caso la naturaleza deformable de los tejidos. Para poder realizar la simulación del comportamiento físico del sistema se emplea el modelado numérico para el cálculo de sus respuestas en el tiempo. Así los modos de simulación se pueden dividir en 4 elementos, modelo físico, modelo matemático, modelo numérico y modelo computacional.

En particular, los modelos físicos incluyen:

- Representación geométrica.
- Selección de variables desconocidas que se desean evaluar espacio-temporalmente.
- Leyes de la física que gobiernan el comportamiento del sistema.
- Definición de valores conocidos en el sistema, como lo son posición, densidad, masa, velocidad, etcetera.
- Definición de condiciones iniciales y de frontera.
- Fuerzas externas en forma de cargas y tensiones debidas a la interacción con el usuario y con otros objetos en la simulación.

El modelo matemático se define empleando las leyes físicas que definen el comportamiento de sistema en forma de ecuaciones diferenciales. El modelo matemático (cinemático, estático o dinámico) y el modelo numérico (por ejemplo, integración de Taylor, métodos de Euler, Runge-Kutta, entre otros) están muy ligados pues el modelo numérico es la expresión discreta del modelo matemático. Una vez definido el modelo matemático se puede construir el modelo computacional. Dicho modelo depende de la plataforma en la que se desee trabajar, por lo que mismos modelos numéricos tienen diferente representación en diferentes plataformas.

Como se mencionó anteriormente los modelos empleados para la implementación de simuladores quirúrgicos son principalmente modelos físicos de cuerpos rígidos y modelos físicos de cuerpos suaves. Esto se explica fácilmente si consideramos cuales son los principales elementos que se modelan en una cirugía, tejidos y herramientas. Generalmente los tejidos que conforman los órganos del cuerpo se comportan como objetos deformables, que pueden ser estáticos o dinámicos dependiendo de su implementación, mientas que las herramientas usadas durante la cirugía se comportan como objetos rígidos dinámicos. La convención anteriormente mencionada es válida para el simulador de RTUP.

Dado lo anterior el motor de simulación física del simulador de RTUP requiere modelar tanto cuerpos suaves en este caso dinámicos, como con cuerpos rígidos, así como la interacción entre ellos. Esta interacción es compleja, pues se requiere una respuesta bidireccional a las interacciones de cuerpos rígidos con cuerpos rígidos, cuerpos suaves con cuerpos suaves y cuerpos suaves con cuerpos rígidos.

Desde el punto de vista de la forma de discretizar los modelos, contamos con dos alternativas que nos ofrecen una solución al problema: 1) simulación basada en mallas mediante el uso de métodos de masas-resortes o elementos finitos; 2) simulación libre de mallas mediante el uso de simulación de partículas.

2.1. Técnicas de simulación basadas en mallas.

Las técnicas de simulación basadas en mallas reciben ese nombre debido a que los modelos matemáticos que se emplean para modelar el sistema están relacionados a la geometría mediante la cual se realiza la discretización de la forma de los objetos deformables mediante mallas geométricas. La malla se conforma de elementos geométricos regulares (generalmente triángulos, cuadriláteros para dominios superficiales o tetraedros para dominios volumétricos) que aproximan la forma de un objeto que se desea modelar. Estos elementos se encuentran unidos entre sí mediante nodos, los cuales son puntos elegidos mediante muestreo, pudiendo ser regular o aleatorio y cuya única condición es que deben encontrarse dentro del espacio que define al objeto que representan.

Las simulaciones basadas en mallas realizan cálculos solo sobre los elementos geométricos que definen la malla, se utilizan las relaciones entre cada uno de los nodos con el fin de modelar el comportamiento del sistema y sus relaciones. Típicamente el comportamiento del modelo depende de las relaciones que existen entre cada elemento, si existe un cambio en dicha estructura se afecta en su totalidad el sistema por lo que es complicado modificar la estructura de la malla sin afectar el modelado del sistema.

2.1.1. Simulación con método de elemento finito.

Las ciencias de la ingeniería tales como la termodinámica, mecánica de sólidos y fluidos son ampliamente usadas para describir mediante ecuaciones diferenciales el comportamiento de sistemas físicos mediante la discretización de los modelos [6]. En la actualidad muchas de las simulaciones se llevan a cabo mediante la aplicación de elementos finitos, como las simulaciones quirúrgicas presentadas por Bro-Nielsen [23] y Cotin [22]; la principal razón es que este método permite resolver la gran mayoría de las ecuaciones que modelan los sistemas físicos a partir de modelos analíticos basados en mecánica de medios continuos y por lo tanto con cierto realismo biomecánico (generalmente mediante la linearización de modelos no-lineales bajo el supuesto de ciertas condiciones físicas, por ejemplo, cargas externas dentro de las zonas de esfuerzo-deformación menores a las que causarían ruptura de los tejidos).

Los métodos de elementos finitos se aplican mediante la aproximación de variables desconocidas con lo que las ecuaciones diferenciales parciales se convierten en sistemas de ecuaciones algebraicas matriciales en variables de estado. Para lograr esto es necesario modelar mediante sistemas de ecuaciones diferenciales simultaneas muy complejos, entendiéndose esto como miles de ecuaciones cada una gobernando un elemento de los miles que conforman el sistema de elementos (triángulos, tetraedros, etc.), que modelan el comportamiento físico de los sistemas, así como construir métodos numéricos que permitan solucionar sistemas de ecuaciones algebraicas; por último los métodos deben ser ejecutados con herramientas de cómputo a fin de obtener los cálculos necesarios en tiempo real, lo que es muy complejo dadas las dimensiones de los sistemas. En la Figura 2.1 se muestra un esquema donde se ilustra la formulación del problema hasta llegar a una solución numérica.

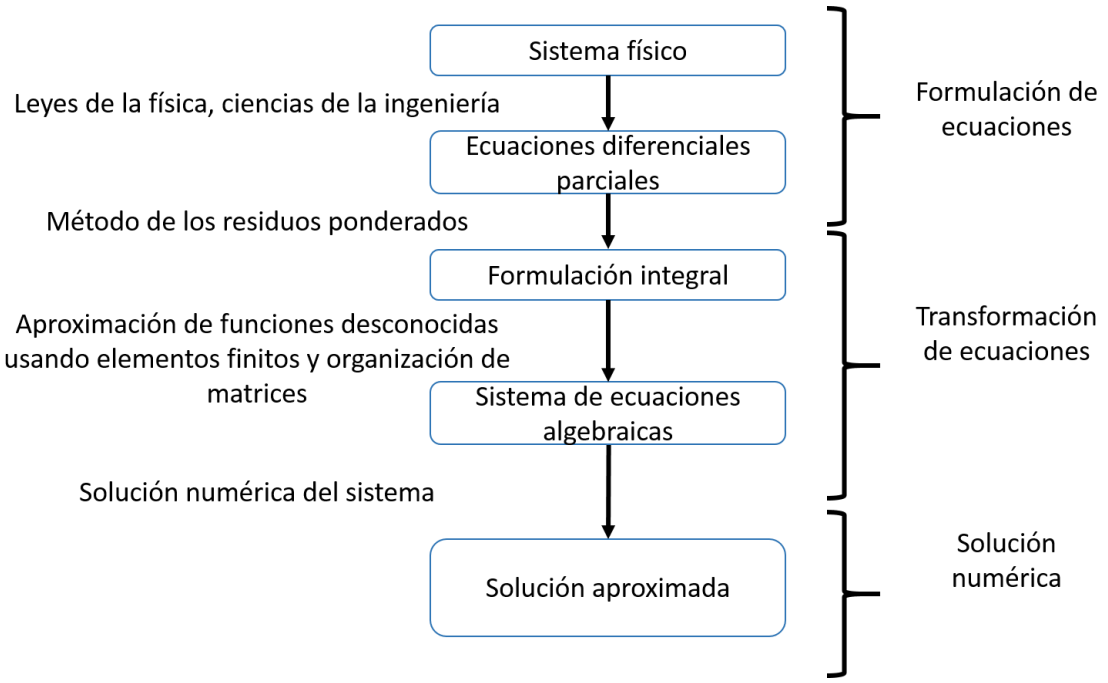


Figura 2.1 Formulación de ecuaciones a partir de sistema físico.

Como se muestra en el diagrama anterior uno de los primeros puntos en la construcción de un método de elementos finitos es la definición del modelo matemático para el sistema F, en el caso de los métodos de elementos finitos el modelo matemático se formula con base a cierto número de variables que representan características del sistema como se ilustra en la ecuación (2.1).

$$e(x) = u(x) - u_{ex}(x)\,, \qquad (2.1)$$

donde $u(x)$ es la función que modela de manera aproximada el sistema *F*, el cual es caracterizado por $u_{ex}(x)$ y $e(x)$ es el error que existe entre los *valores medidos* y *los valores obtenidos* mediante el modelo. La función $u(x)$ se construye considerando n parámetros y m nodos, los cuales se conocen en el sistema, esto nos da una formulación del tipo $u(x_i, a_0, a_1, \dots, a_n)$.

La definición de dicha función se complica conforme se incrementa el número de nodos y parámetros que conforman el modelo.

Ya que se tiene definido el modelo matemático con base en variables se divide el dominio espacial en subdominios. La colección de subdominios en los que se divide el dominio debe ser tal que la unión de los subdominios debe conformar el dominio espacial. A fin de que la discretización de a lugar elementos finitos, se siguen las siguientes reglas para la partición del dominio, esto se muestra en la Figura 2.2 y Figura 2.3:

a) Dos elementos diferentes pueden tener puntos en común solo en los bordes.

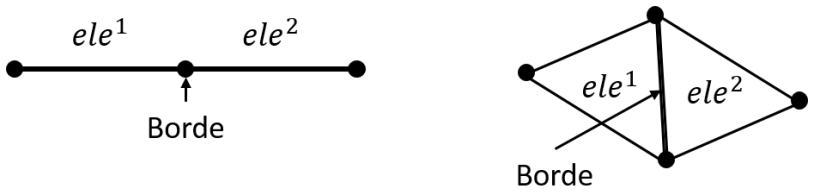


Figura 2.2 Definición geométrica de borde para elementos finitos.

b) El conjunto de subdominios debe ser tan cercano al dominio como sea posible. No debe haber "hoyos" entre elementos.

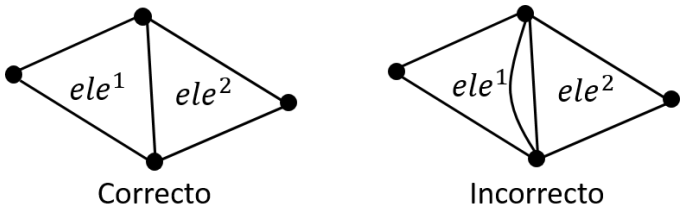


Figura 2.3. Definición de conjuntos de elementos finitos que conforman el dominio de manera correcta e incorrecta.

Al realizar lo anterior se incorpora un error conocido como error de discretización geométrica, este error puede ser reducido incrementado el número de elementos finitos y por consiguiente aumentando la resolución de la malla, sin embargo, el realizar esto incrementa la complejidad computacional del problema.

A fin de simplificar la definición analítica para los elementos se emplean elementos de referencia, dichos elementos tienen la característica de que son formas simples y se pueden transformar en cualquier elemento del subconjunto discretizado mediante una transformación geométrica. Usualmente todos los elementos finitos que componen el dominio son construidos con un mismo elemento de referencia, usualmente triángulos para mallas de superficie y tetraedros para volúmenes.

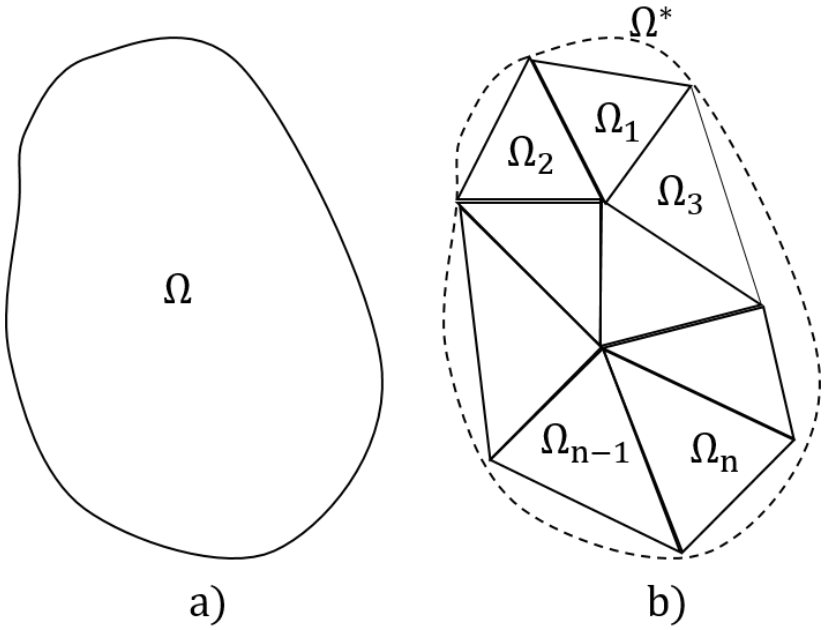


Figura 2.4. Discretización de un dominio mediante elementos finitos triangulares.

a) Dominio continuo, b) Dominio discretizado.

En la Figura 2.4 se ilustra la discretización de un dominio Ω a un dominio Ω* empleando elementos triangulares Ω_i donde $i \in [1, \ldots, n]$, se observa el error producido por la aproximación de la forma mediante el uso de este tipo de elementos. Para que la discretización sea correcta debe cumplir las condiciones arribas mencionadas y además $\Omega^* = \cup_{i=1}^{n} \Omega_i$ donde $\Omega^* \approx \Omega$.

Los sistemas continuos y discretos pueden ser subdivididos en tres categorías: problemas de equilibrio o valor frontera, eigenvalor y propagación. Dada la naturaleza de las simulaciones que intentamos construir en este trabajo caemos en la categoría de problemas de propagación. Estos problemas consisten en la evaluación de $u(x,t)$ para $t \geq t_0$ en un sistema no estático, cuando $u(x,t_0)$ es conocido.

La definición de las ecuaciones diferenciales que modelan el sistema que cae en la categoría de problemas de propagación se formula de la siguiente manera.

$$\mathrm{m}\frac{\delta^2 \mathrm{u}}{\delta \mathrm{t}^2} + \mathrm{c}\frac{\delta \mathrm{u}}{\delta \mathrm{t}} + \mathfrak{L}(\mathrm{u}) + f_\Omega = 0 \; sobre\; el\; dominio\; \Omega\,, \qquad (2.2)$$

$$\mathcal{C}(\mathrm{u}) = f_S \; sobre \text{ la frontera } S\; del\; dominio\; \Omega\,, \qquad (2.3)$$

donde $\mathfrak{L}$ y $\mathcal{C}$ son operadores diferenciales que caracterizan el sistema, u son las funciones desconocidas, f_Ω son las funciones asociadas con las cargas externas aplicadas en forma de fuerza o torque y f_s son las funciones asociadas con las condiciones de frontera. Además se toman en cuenta las siguientes condiciones iniciales.

$$u = u_0 \text{ y } \frac{\delta \mathrm{u}}{\delta \mathrm{t}} = ui_0 \; para\; t = t_0.$$

Ya que están formuladas las ecuaciones diferenciales que modelan el sistema se aplica el método de los residuos ponderados a fin de obtener la formulación integral. Para lograr lo anterior es necesario encontrar la función *u* que nulifica la siguiente forma integral.

$$\mathrm{W}(\mathrm{u}) = \int_v \langle\psi\rangle\{R(u)\}dV = \int_v \langle\psi\rangle\{\mathfrak{L}(u) + f_v\}dV = 0\,, \qquad (2.4)$$

donde $\langle\psi\rangle$ es un vector fila de función de peso y $R(u)$ es el término de residuo el cual cuando tiende a 0 bajo las mismas condiciones que $e(x)$ en la ecuación (2.1), define a la función u que modela al sistema.

Concluido lo anterior se procede a realizar la formulación matricial en variables de estado de las ecuaciones lineales que modelan el sistema, se trabaja con matrices dada la facilidad para transformar dicha representación en código de computadora. Se inicia realizando la discretización del modelo matemático con base en los elementos finitos, es decir planteamos la relación ilustrada en la ecuación (2.5) que al expandirla con respecto a las ecuaciones (2.2) y (2.4) resulta en la ecuación (2.6).

$$W^c \Rightarrow \mathrm{W}_\mathrm{h}^\mathrm{c}\,, \qquad (2.5)$$

$$\mathrm{W}_\mathrm{h}^\mathrm{c} = \langle\delta u\rangle\left([m]\left\{\frac{d^2u_i}{dt^2}\right\} + [c]\left\{\frac{du_i}{dt}\right\} + [k]\{u_i\} - \{f\}\right) = \langle\delta u\rangle(r)\,, \quad (2.6)$$

donde [m], [c] y [k] son matrices de masas, viscosidad y rigidez para cada elemento; {f} es el vector de cargas y {r} el vector de residuos.

A continuación se realiza la unión de la contribución de cada elemento al sistema de modo que $W_k = \sum_c W_h^c$, la solución a este sistema involucra encontrar una función $\{U(t)\}$ tal que se satisfaga la ecuación (2.7),

$$\{R\} = [M]\left\{\frac{d^2U}{dt^2}\right\}+[C]\left\{\frac{dU}{dt}\right\} + [K]\{U\} - \{F\} = 0 \ , \quad (2.7)$$

donde [M], [C] y [K] son las matrices globales de masa, viscosidad y rigidez, {f} el vector global de cargas y reacciones, {U} el vector global de funciones nodales desconocidas y {R} el vector de residuos globales.

Hasta este punto se ha descrito el método de elementos finitos, el cual es un método con gran precisión y con el cual se puede llevar a cabo la simulación de sistemas físicos dinámicos con gran precisión la cual está en función de la cantidad de elementos en los que se discretice el sistema. Esto es de vital importancia para los simuladores en tiempo real como el que se planea construir, sin embargo, el método tiene un problema fundamental, que es que cuando se retiran elementos de la simulación por lo que es necesario modificar las matrices [M], [C] y [K] y sus inversas en tiempo real. Un ejemplo de lo anterior se presenta en la simulación de fluidos o simulación de cortes y rupturas de tejido.

En los casos descritos anteriormente las simulaciones basadas en mallas como las de elementos finitos pueden emplear una técnica conocida como método de los elementos discretos, la cual considera a cada uno de los elementos finitos como un cuerpo independiente, sin embargo para simular cortes es posible aprovechar técnicas de simulación libres de mallas.

2.2. Técnicas de simulación libres de mallas.

Los métodos de simulación libres de malla han tomado gran popularidad debido a las ventajas que ofrecen sobre los métodos de simulación basados en mallas en cuanto a que su formulación no depende de la estructura geométrica del objeto discreto. Los métodos de simulación basados en partículas son un ejemplo de dichos métodos.

A diferencia de las técnicas de simulación basadas en mallas, las técnicas de simulación libres de mallas no modelan el sistema en relación directa con la malla, aunque puede existir una malla esta no juega un papel fundamental en el modelado del sistema. En estos sistemas las relaciones entre los elementos son definidas de manera independiente y no depende necesariamente de la malla que aproxima la forma de un objeto. Lo anterior tiene como implicación el definir un modelo con el cual los elementos que conforman el dominio del sistema interactúan entre ellos, es decir modelar la interacción entre los elementos que conforman el sistema.

Al emplear estas técnicas de simulación es posible modificar de manera independiente cada elemento e incluso eliminarlos sin la necesidad de replantear las ecuaciones que modelan el sistema, esto presenta una gran ventaja sobre las simulaciones basadas en mallas cuando se realiza la simulación de sistemas cuyo dominio se modifica en el transcurso de la simulación.

Entre los métodos de simulación libres de mallas se encuentran SPH (*smoothed particle hydrodynamics*) [17] el cual fue desarrollado por R. A. Gingold y J. J. Monaghan en 1977 o los sistemas dinámicos basados en posición (*Position Based Dynamics*) [7], por nombrar algunos ejemplos.

A continuación se aborda la simulación basada en partículas empleando métodos de sistemas dinámicos basados en posición.

2.2.1. Simulación de partículas.

Los métodos basados en partículas tienen la ventaja de que se puede realizar la simulación de la dinámica de sistemas que no necesariamente emplean métodos basados en fuerzas, sino que se pueden aprovechar métodos basados en posición. El emplear métodos basados en posición tiene la ventaja de que proporciona más control sobre la simulación que su contraparte basada en fuerzas, esto se debe a que se evitan los problemas generados por los métodos de integración numérica para obtener posiciones a partir de fuerzas, como lo es el cálculo de fuerzas debido a una colisión con penetración de cuerpos; esto es de vital importancia en aplicaciones de gráficos por computadora, videojuegos y simulaciones donde existen elementos cinemáticos de interacción controlados por un usuario.

Las simulaciones basadas en partículas se emplean tanto en simulaciones de cuerpos rígidos, cuerpos deformables (ya sea plásticos o visco elásticos) y fluidos como son líquidos o gases. Aunque estos métodos gozan de gran popularidad en la industria de los videojuegos por su eficiencia y facilidad para trabajar con objetos cinemáticos, no gozan de tanta aceptación en aplicaciones que buscan comportamientos más realistas físicamente. Sin embargo hay que hacer un compromiso entre desempeño, realismo y funcionalidad, por lo que son una buena alternativa para construir simulaciones enfocadas en el entrenamiento de usuario, donde lo que más interesa es la interacción con el usuario y no el realismo físico.

Al igual que en los métodos basados en mallas los métodos basados en partículas requieren dividir el dominio en un conjunto de elementos a partir de los cuales se modelaran las leyes que rigen el sistema. En este caso el elemento que se emplea para discretizar el dominio son puntos que se encuentren contenidos en el dominio. Dependiendo de la cantidad de puntos que conformen el dominio se mejora la precisión con la que se modela el sistema, es decir a mayor cantidad de puntos mejor aproximación al modelado real del sistema, sin embargo lo anterior lleva consigo el incremento en la cantidad de cálculos que se deben realizar a fin de integrar la información de todas las partículas que modelan el sistema.

Para este trabajo se tomó como referencia lo reportado por M. Müller et al [7] en cuanto a dinámica basada en posición, el cual es un método de simulación basado en partículas que posteriormente fue implementado en la biblioteca de simulación de NVIDIA, la cual lleva el nombre de *Flex*[8], sobre esta biblioteca se hablara más en detalle en el capítulo 4.

[8] https://developer.nvidia.com/flex

El algoritmo de simulación de partículas basado en posición tiene como primer paso el representar el objeto dinámico mediante un conjunto de N vértices y M restricciones. A cada vértice $i \in [1, \dots, N]$ se le asignan las propiedades de masa m_i, velocidad v_i y posición x_i. Por otro lado las restricciones $j \in [1, \dots, M]$ tienen las propiedades de cardinalidad n_j, las restricciones C_j son definidas con la función $C_j: \mathbb{R}^{3n_j} \to \mathbb{R}$, empleando los índices de los vértices donde $\{i_1, \dots i_{n_j}\}, i_k \in [1, \dots, N]$, un parámetro de rigidez $k_j \in [0 \dots 1]$ que es un valor escalar y un tipo de igualdad para la restricción j del tipo $C_j\left(x_{i_1}, \dots, x_{i_{n_j}}\right) = 0$ o desigualdad del tipo $C_j\left(x_{i_1}, \dots, x_{i_{n_j}}\right) \geq 0$.

El siguiente algoritmo indica los pasos a seguir a fin de llevar a cabo la simulación del objeto dinámico, los cálculos se realizan con base en la definición de un lapso de tiempo Δ_t, el cual se define usualmente como 0.16 segundos ($\frac{1}{60}$). El algoritmo recibe como entrada el conjunto de vértices y f_{ext} que inicialmente solo es la gravedad.

(1) Para todos los vértices *i*
(2) Inicializa $x_i = x_i^0, v_i = v_i^0, w_i = 1/m_i$
(3) Fin de ciclo
(4) Ciclo
(5) Para todos los vértices *i* haz $v_i \leftarrow v_i + \Delta_t w_i f_{ext}(x_i)$
(6) Rutina de amortiguamiento de velocidades $(v_1, \dots, v_N)$
(7) Para todos los vértices *i* haz $p_i \leftarrow x_i + \Delta_t v_i$
(8) Para todos los vértices *i* crea restricciones de colisión $(x_i \to p_i)$
(9) Ciclo de iteraciones para Solucionador de posición
(10) Rutina de proyección de restricciones $(C_1, \dots, C_{M+M_{coll}}, p_1, \dots, p_N)$
(11) Fin de ciclo
(12) Para todos los vértices *i*
(13) $v_i \leftarrow (p_i - x_i)/\Delta_t$
(14) $x_i \leftarrow p_i$
(15) Fin ciclo
(16) Actualizar velocidad de vértices $(v_1, \dots, v_N)$
(17) Fin ciclo

En las líneas 1-3 se realiza la inicialización de las variables de posición, velocidad e inverso de la masa de cada vértice. En la línea 5 se realiza el cálculo de la velocidad a partir de las fuerzas externas, en el caso de que no haya fuerzas debido a la interacción de partículas solo se considera la fuerza de gravedad. Dichas velocidades son amortiguadas a fin de mejorar la estabilidad. En la línea 7 se genera una estimación inicial de la posición de los vértices con base a la velocidad de cada uno. En la línea 8 se generan nuevas restricciones a causa de colisiones derivadas de la estimación de las nuevas posiciones. En las líneas 9-11 se realiza el cálculo de posición al final del intervalo de tiempo Δt, empleando un método iterativo que encuentra la posición válida para los vértices empleando las restricciones que tiene cada vértice, esto es la proyección de restricciones. En las líneas 12-16 se realiza el cálculo de la velocidad, y se actualiza la posición de los vértices.

Un punto importante del sistema es que las restricciones $C_1, \dots, C_M$ permanecen fijas a lo largo de la simulación, además de que a cada intervalo de tiempo se calculan las colisiones que existen entre partículas p, estas reciben el nombre de restricciones de colisión M_{coll}. Este esquema es estable ya que no extrapola los valores de la velocidad y posición que se calculan en los incisos 13 y 14. Sin embargo, es posible encontrar inestabilidades para Δ_t grandes debido a que emplea el método de Newton-Raphson para producir las posiciones válidas.

La entrada para el solucionador son las restricciones iniciales, las restricciones debidas a las colisiones es decir $M + M_{coll}$, así como las estimaciones de las nuevas posiciones $p_1, \dots, p_N$ de los puntos. El modificar las estimaciones tal que se satisfagan las restricciones resulta en un sistema de ecuaciones no lineales, para cuya solución se emplea la iteración del tipo Gauss-Seidel.

Lo anterior nos lleva a realizar la proyección de un conjunto de puntos de acuerdo con las restricciones, es decir proyectar las restricciones, para determinar las posiciones válidas de los vértices. El problema más importante en cuanto a la conectividad de puntos en movimiento es la conservación de los momentos lineal y angular. La conservación del momento lineal y momento angular está definida por las ecuaciones (2.8) y (2.9), respectivamente, en dichas ecuaciones Δp_i corresponde al desplazamiento del *vértice i* causado por la proyección anteriormente mencionada. Cabe mencionar que r_i es la distancia de p_i al centro de rotación común fijado de manera arbitraria.

$$\sum_i m_i \Delta p_i = 0, \qquad (2.8)$$

$$\sum_i r_i \times \mathrm{m_i} \Delta p_i = 0\ . \qquad (2.9)$$

En la Figura 2.5 se muestra un ejemplo de la proyección de restricciones de 2 puntos.

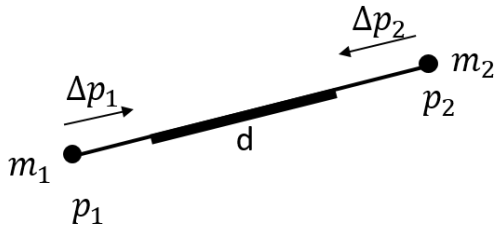


Figura 2.5. Proyección de restricción $C(p_1, p_2) = |p_1 - p_2| - d$.

En el método que propone Müller [7] es necesario calcular los cambios en las restricciones de las partículas, para ellos se define el cambio en la restricción en función de la variación de la posición Δp y el gradiente de la restricción $\nabla_p C$. Lo anterior se muestra en la ecuación (2.10). En el caso de cuerpos rígidos $\nabla_p C$ es perpendicular a los nodos del cuerpo ya que es la dirección del máximo cambio.

$$C(p + \Delta p) \approx C_p + \nabla_p C(p) \cdot \Delta p = 0\ . \qquad (2.10)$$

Al restringir Δp para que siga la misma dirección de $\nabla_p C$ implica elegir un valor escalar λ tal que,

$$\Delta p = \lambda \nabla_p C(p)\ . \qquad (2.11)$$

Al sustituir la ecuación (2.11) en la ecuación (2.10), despejando λ y sustituyendo nuevamente en la ecuación (2.11) obtenemos la ecuación (2.12) para determinar el valor de cambio en posición de una partícula en la forma de una solución iterativa con un método de Newton-Raphson.

$$\Delta p = -\frac{C(p)}{|\nabla_p C(p)|^2} \nabla_p C(p). \qquad (2.12)$$

Para realizar la corrección de un punto individual $\mathrm{p_i}$ tenemos la ecuación (2.13),

$$\Delta p_i = -s\nabla_p C(p_1, \dots, p_n), \qquad (2.13)$$

donde el factor de escala s se define en la ecuación (2.14) y es el mismo para todos los puntos.

$$\mathrm{s} = \frac{C(p_1,\dots,p_n)}{\sum_j \left|\nabla_{p_j} C(p_1,\dots,p_n)\right|^2}. \qquad (2.14)$$

En el caso de que los puntos posean diferentes masas las correcciones Δp_i son ponderadas aplicando $w_i = 1/m_i$.

$$\Delta p_i = \lambda w_i \nabla_{p_i} C(p_i)\ , \qquad (2.15)$$

donde
$$\mathrm{s} = \frac{C(p_1,\dots,p_n)}{\sum_j w_j \left|\nabla_{p_j} C(p_1,\dots,p_n)\right|^2}, \qquad (2.16)$$

y
$$\Delta p_i = -s\mathrm{w_j}\nabla_p C(p_1, \dots, p_n). \qquad (2.17)$$

Considerando el ejemplo ilustrado en la Figura 2.5 obtenemos las ecuaciones de corrección (2.18) y (2.19).

$$\Delta p_1 = -\frac{w_1}{w_1+w_2}(|p_1 - p_2| - d)\frac{p_1-p_2}{|p_1-p_2|}, \qquad (2.18)$$

$$\Delta p_2 = -\frac{w_2}{w_1+w_2}(|p_1 - p_2| - d)\frac{p_1-p_2}{|p_1-p_2|}. \qquad (2.19)$$

Lo anterior no considera la rigidez *k* con la que se definen las restricciones *C*, lo ideal es considerar la rigidez como un factor escalar pero dada la naturaleza iterativa del solucionador el efecto de k no es lineal, para volverlo lineal se emplea un factor $k' = 1 - (1-k)^{1/\mathrm{n_s}}$. Sin embargo, el efecto de la rigidez depende en mayor medida del intervalo de tiempo usado en la simulación. El efecto restrictivo de rigidez se obtiene de multiplicar la variación en posición por el factor que *k'* con lo que tenemos la aproximación $\Delta p(1-k)^{1/\mathrm{n_s}} = \Delta p(1-k)$.

Una ventaja de las implementaciones basadas en posición es la facilidad en la que la respuesta a colisiones se lleva a cabo. Las restricciones de colisión M_{coll} son dinámicas y se generan a cada paso de la simulación, estos dependen del número de vértices que colisionen. En el caso de colisiones continuas se prueba para cada vértice i el rayo $x_i \rightarrow p_i$. Si el rayo $x_i \rightarrow p_i$ penetra un objeto se calcula el punto en el que penetra q_c y la normal a la superficie de dicho punto n_c. Con lo anterior se genera y agrega la restricción $C(p) = (p - q_c) \cdot n_c$ con una rigidez $k = 1$. En este caso pueden existir fallas cuando el rayo $x_i \rightarrow p_i$ se encuentre completamente dentro del objeto, es decir los objetos se penetren y sobrepongan entre sí mismos.

Cuando lo anterior ocurre se maneja como una colisión estática. Se calcula el punto de superficie q_s que es el más cercano a p_i así como la normal n_s. La restricción se genera empleando la función $C(p) = (p - q_s) \cdot n_s$ considerando una rigidez $k = 1$.

Para acelerar el ciclo del solucionador, el proceso de generación de colisiones se realiza fuera de ese ciclo. Hay que considerar que el método para detección de colisiones no es infalible y puede fallar cuando las partículas se mueven a velocidades altas; sin embargo es posible mantener la simulación pese a esas fallas ya que la penetración ente partículas no vuelve inestable la simulación.

En el caso de la respuesta debida a la colisión de objetos dinámicos ésta se consigue simulando ambos objetos.

2.3. Simulación de cuerpo rígido.

Un cuerpo rígido es aquel que no es susceptible a alteraciones de su forma o estructura por efecto de fuerzas externas, este concepto es una idealización del comportamiento de un objeto, pues en la naturaleza no hay objetos que cumplan esa característica, sin embargo, este modelo es muy útil en simulaciones en las que se desea modelar el comportamiento de cuerpos que interactúan en un espacio determinado.

Al realizar la simulación de un cuerpo rígido unas de las características que se busca simular son sus *propiedades cinemáticas y dinámicas* debido a las fuerzas que interactúan con el objeto, no se requiere hacer análisis de fueras internas ya que el objeto no experimenta cambios de forma. Los puntos más importantes que se requiere modelar de un cuerpo rígido son posición, velocidad y orientación, para lograr lo anterior es necesario modelar las reglas con las que se realizan los movimientos de traslación y rotación, así como también el efecto de las colisiones de un cuerpo rígido sobre otros cuerpos ya sean rígidos o deformables.

La traslación es definida como el movimiento del centro de masa c_m mientras que la rotación se define como el movimiento del cuerpo rígido alrededor del centro de masa. En la Figura 2.6 se muestra el concepto de traslación y rotación con base al momento lineal *P* y el momento angular *L*.

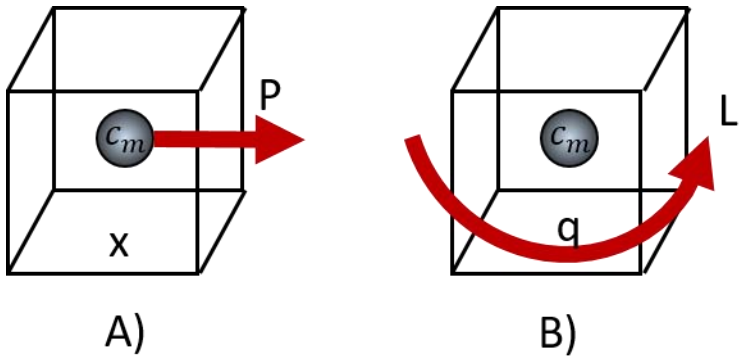


Figura 2.6. Momento lineal P y Momento angular L.

A) Traslación de cuerpo rígido con centro de masa c_m en posición x debido al momento lineal P.

B) Rotación de cuerpo rígido con centro de masa c_m con orientación q debido a momento angular L.

Considerando la definición de rigidez propuesta por el método de simulación dinámica basado en posición que se revisó en el apartado anterior donde la variación de las posiciones entre partículas de un objeto se ponderan mediante el factor de rigidez $k \in [0 \dots 1]$ cuando un cuerpo rígido tiene *k=1* obtenemos que la variación en la posición relativa entre partículas que componen al cuerpo rígido es $\Delta p(1-k)^{1/n_s} = 0$. Con lo que la simulación del cuerpo rígido cumple la primera condición que estipula que no sufre deformaciones debido a la aplicación de fuerzas externas.

La definición de las fuerzas con las que se determina el *momento lineal P* y *momento angular L* del cuerpo rígido quedan expresadas de la siguiente forma. El aplicar una fuerza *F* sobre el cuerpo rígido es igual a aplicar una fuerza *F* sobre un punto del objeto, el centro de masa, lo que produce un cambio en el momento lineal *P* del cuerpo. Si esa fuerza se aplica sobre otro punto del objeto que no sea el centro de masa existe un cambio en el momento angular *L* del cuerpo, este es dependiente de una posición relativa *r*.

$$f = \frac{dP}{dt}, \qquad (2.20)$$

$$P = v\,m, \qquad (2.21)$$

$$v = \frac{dx}{dt}. \qquad (2.22)$$

donde *P* es el momento lineal, *v* es la velocidad del centro de masa, *M* la masa del objeto y *x* la posición del centro de masa.

$$f \times r = \frac{dL}{dt}, \qquad (2.23)$$

$$w = I(t)^{-1}L, \qquad (2.24)$$

$$I(t)^{-1} = R\,I(0)^{-1}\,R^{T}, \qquad (2.25)$$

Donde *L* es el momento angular, *r* es la posición relativa al centro de masa donde actúa la fuerza *f*, *w* es la velocidad angular el cuerpo y *I(t)* es el tensor de inercia en el tiempo *t*, *R* es la matriz de rotación que describe la orientación del objeto.

Con lo anterior es posible modelar los movimientos de rotación y traslación de un cuerpo rígido sobre el que actúan fuerzas externas.

2.4. Simulación de cuerpo deformable.

En la sección anterior se abordó la simulación de un cuerpo rígido que se idealiza indeformable e indestructible, lo cual simplifica su modelado en términos de traslación y rotación. En el caso de un cuerpo deformable o cuerpo suave no solo se debe modelar a fin de determinar su traslación y su rotación, sino que se deben tomar en cuenta las variaciones en forma que sufre el objeto debido a la aplicación de fuerzas externas.

Uno de los métodos más populares para el modelado de un cuerpo deformable o suave es el considerar al objeto como un cuerpo con propiedades de viscosidad y elasticidad. Se considera al cuerpo como un sólido visco-elástico. El representar mediante un modelo que considera la elasticidad y viscosidad para calcular la reacción debido a fuerzas simplifica la tarea de modelar los cuerpos deformables, en especial cuando se enfoca en modelar tejidos biológicos. Sin embargo en la naturaleza no hay cuerpos que únicamente tengan estas propiedades por lo que esta técnica solo proporcionan una aproximación a la realidad. Afortunadamente en el caso de simulaciones enfocadas a la interacción con el usuario como lo es la cirugía asistida por computadora o entrenamiento para adquisición de habilidades y destrezas, la aproximación que ofrecen dichos modelos es suficiente.

La ley de Hooke proporciona un modelo para la elasticidad lineal de un cuerpo, donde se considera que la aplicación de una fuerza de entrada F guarda una relación lineal con un coeficiente de elasticidad propio del material μ y la deformación del cuerpo X.

$$F = \mu X. \qquad (2.26)$$

Por otro lado un modelo simple de viscosidad se obtiene de considerar al cuerpo como un fluido viscoso, bajo tal consideración tenemos que la fuerza aplicada F está relacionada con la velocidad de deformación X_v y el coeficiente de viscosidad η.

$$F = \eta X_v. \qquad (2.27)$$

Como ya se mencionó en general los cuerpos deformables o suaves se modelan como cuerpos visco-elásticos, los cuales están caracterizados por funciones de relajación y *creep* (variación lenta de forma), dos de los modelos constitutivos con mayor popularidad son el modelo de Maxwell y el de Kelvin-Voigt [24, 25], cuyas representaciones esquemáticas se ilustran en la Figura 2.7.

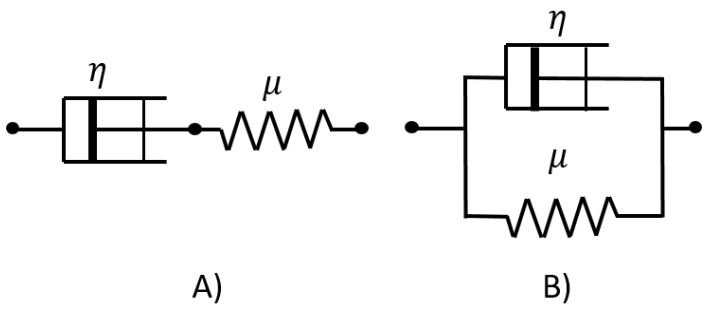


Figura 2.7. Modelos constitutivos de visco elasticidad lineal. A)Modelo de Maxwell B)Modelo de Kelvin-Voigt. (Coeficiente de viscosidad η y coeficiente de elasticidad μ).

En este trabajo no se entra en detalle en cuanto a las ecuaciones de cada uno de los modelos mencionados arriba.

Otra alternativa para la simulación de cuerpos deformables es la simulación del objeto mediante nubes de puntos las cuales se encuentran ligadas mediante restricciones que asemejan resortes. Este método es aplicado en simulaciones basadas en partículas como el que se mencionó en los capítulos anteriores. En el caso de la simulación dinámica basada en posición el objeto deformable se modela usando propiedades similares a las de una tela, de ahí que el modelo reciba el nombre de *cloth* o de ropa.

La implementación de la representación de un cuerpo mediante *cloth* en la simulación dinámica basada en posición se realiza generando una malla de triángulos que represente la superficie del objeto que se desea representar, donde cada vértice de los triángulos es una partícula del sistema. A partir de dicha representación se hacen las siguientes consideraciones, a lo más una arista puede tener dos triángulos en común, dada una densidad ρ expresada en masa sobre área $[kg/m^2]$, la masa de cada partícula es una tercera parte de la suma de la masa de todos los triángulos que contengan dicha partícula.

Para cada arista se generan una restricción de elasticidad junto con un valor restricción de rigidez global $k_{stretch}$, la restricción de estiramiento que se muestra en la ecuación (2.28) requiere que se defina el valor de la longitud inicial de cada arista.

$$C_{stretch}(p_1, p_2) = |p_1 - p_2| - l_0. \qquad (2.28)$$

Para cada par de triángulos se genera una restricción de flexión, dicha restricción se describe en la ecuación (2.29) y toma en cuenta el ángulo inicial entre los triángulos que comparten una arista φ_0. Del mismo modo se define un valor de rigidez de flexión global k_{bend}.

$$C_{bend}(p_1, p_2, p_3, p_4) = \operatorname{acos}\left(\frac{(p_2-p_1)\times(p_3-p_1)}{|(p_2-p_1)\times(p_3-p_1)|}.\frac{(p_2-p_1)\times(p_4-p_1)}{|(p_2-p_1)\times(p_4-p_1)|}\right) - \varphi_0. \qquad (2.29)$$

Las restricciones $C_{stretch}$ y C_{bend} componen el conjunto de restricciones C, estos valores son usados en la ecuación (2.17) con el fin de modular la variación en la posición que puede sufrir una partícula, con lo anterior se consigue simular un cuerpo deformable con propiedades elásticas ya que la restricción $C_{stretch}$ busca mantener la distancia inicial entre partículas.

Lo anterior funciona bien para representar objetos volumétricos mediante una superficie cerrada, sin embargo, no es capaz de representar volúmenes mediante mallas tetraédricas debido a que la geometría que impone dicha representación no cumple las *consideraciones iniciales* con las que se plantea el método, por ejemplo, la restricción de que una arista es compartida a lo más por dos triángulos.

Capítulo 3. Implementación de técnicas de simulación en GPU.

A finales del siglo pasado la velocidad de los procesadores (frecuencia de reloj de CPU) se venía incrementando rápidamente, por lo que el aprovechamiento de los nuevos recursos de cómputo disponibles se lograba con el simple hecho de adquirir e instalar un nuevo CPU con mayor velocidad, es decir el aprovechamiento de los recursos de computo no requería de modificaciones al código de los programas, solo era necesario adquirir nuevo hardware. El fenómeno que observábamos era que cada nueva generación de procesadores brincaba de decenas de megahertz a centenas y posteriormente a un gigahertz y así hasta que hemos llegado a velocidades que se mantienen debajo de los 10 gigahertz.

Hoy en día observamos que esa velocidad se mantiene casi constante entre las nuevas generaciones, la principal razón por la que los fabricantes han limitado el incremento de la velocidad es la temperatura de los procesadores. El incrementar la frecuencia del procesador eleva su temperatura tanto que puede llevarlo a un punto donde se derrita. Usualmente se maneja una relación en cuanto a la disipación de energía de un procesador[9]:

$$P \sim C_{dyn} * Volt^2 * freq,$$

donde P es la energía disipada, C_{dyn} es la capacitancia dinámica, *Volt* es el voltaje y *freq* la frecuencia. Con lo anterior es posible observar que el incrementar la frecuencia tiene como consecuencia un incremento en la temperatura.

Debido a esta limitación es que se cambió el paradigma para el desarrollo de las nuevas generaciones de procesadores, hoy en día la tendencia es tener cada vez más núcleos embebidos en un solo procesador, con lo que ya no es imperante que se incremente la velocidad de los procesadores. Podemos decir que hoy en día los procesadores son multiprocesadores.

Lo anterior conlleva la desventaja de que ahora a fin de aprovechar mejor esos recursos de cómputo es necesario modificar el código de los programas con la finalidad de aprovechar las nuevas características. Dado lo anterior es que hoy en día la programación en paralelo aparece como la mejor alternativa para aprovechar estos recursos de cómputo.

[9] https://software.intel.com/en-us/blogs/2014/02/19/why-has-cpu-frequency-ceased-to-grow

Como se ha venido mencionando en la actualidad la cantidad de recursos de cómputo disponibles se ha incrementado, sin embargo la manera en que podemos aprovecharlos ha cambiado. Uno de estos nuevos recursos de cómputo son las *unidades de procesamiento gráfico* o *Graphics Processing Unit* (GPU) en inglés, que permiten cómputo extensivo de manera paralela.

Las primeras GPU fueron diseñadas para el aceleramiento de gráficos, por lo que solo soportaban ciertas funciones. Sin embargo para el año de 1999 NVIDIA lanzo su primer GPU la cual poseía un gran desempeño en operaciones de punto flotante, por lo que fue tomada por investigadores para diseñar General Purpose Graphics Processing Unit (GPGPU Unidades de procesamiento gráfico de propósito general). En un inicio implementar computo en GPGPU no era una tarea sencilla, esto es debido a que se tenía que tener conocimiento de OpenGL y abstraer los problemas que se deseaban resolver a modelos que usaran triángulos y polígonos.

En 2003 Ian Buck y su equipo de trabajo revelo Brook [26], el cual fue el primer modelo de programa que extendía el lenguaje de programación C para construir paralelismo de datos. Brook no solo era más sencillo de usar, sino que lograba mejoras en el desempeño 7 veces mejores a las implementaciones "artesanales" tradicionales. Debido a lo anterior fue que NVIDIA junto con Ian Buck trabajaron para implementar cómputo de propósito general en GPU usando el lenguaje de programación C, con lo que en 2006 revelaron CUDA [27].

Hoy en día las GPU poseen un mayor poder de computo que los CPU en termino de paralelismo de datos, además la tendencia que marca su desarrollo mantiene un crecimiento mayor en termino de paralelismo de datos que los CPU. En la Figura 3.1 se muestra una gráfica en la que se compara el desempeño de GPU contra el desempeño de CPU en términos de GFLOPS. Para definir que es un GFLOPS (GigaFLOPS 10^9 FLOPS) se requiere definir primero que es FLOPS, FLOPS es el acrónimo en inglés para floating-point operations per second. Los FLOPS son calculados con la ecuación (3.1).

$$FLOPS = sockets * \frac{cores}{socket} * clock * \frac{FLOPs}{cycle}. \quad (3.1)$$

En la ecuación *sockets* se refiere al número de procesadores que se usan, $\frac{cores}{socket}$ se refiere a los núcleos de cada procesador y $\frac{FLOPs}{cycle}$ es el número de operaciones de punto flotante que puede realizar un microprocesador por ciclo.

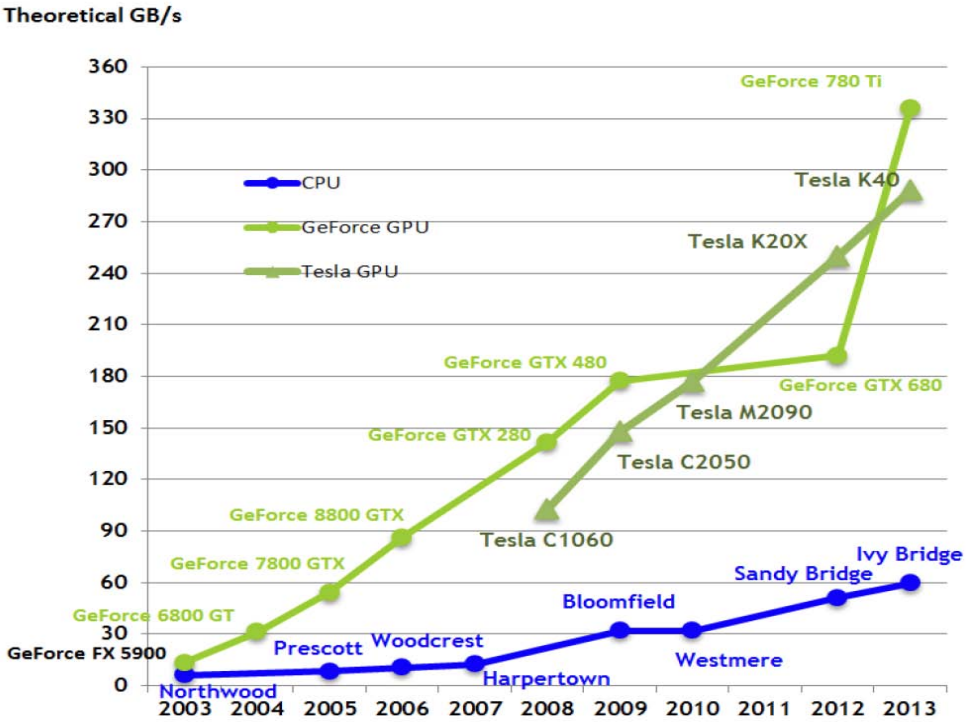


Figura 3.1. Comparación de desempeño entre GPU y CPU con respecto a GFLOPS.[10]

La manera más popular de aprovechar el poder de las GPU hoy en día es mediante el cómputo acelerado en GPU. El cómputo acelerado en GPU hace uso de la GPU en conjunto con el CPU a fin de acelerar la ejecución de un programa. El cómputo acelerado en GPU ofrece un cambio en el desempeño al ejecutar en GPU las porciones del programa con cómputo intensivo, mientras que el resto del programa sigue corriendo en CPU. Lo anterior implica que no todo el cómputo se tiene que realizar en GPU, solo aquellas partes que requieren de cómputo intensivo en paralelo. En la Figura 3.2 se muestra un diagrama de NVIDIA donde se ilustra la aceleración por GPU comparándola contra diferentes generaciones de CPU iniciando con NorthWood, Prescott hasta llegar a Ivy Bridge. Tanto Tesla como GeForce son GPU que pueden ser usadas para computo científico pero Tesla no posee las capacidades de despliegue grafico que posee GeForce, sin embargo, que Tesla proporciona mejores capacidades de cómputo.

[10] NVIDIA CUDA C Programming Guide, versión 7.5, Mayo 2015

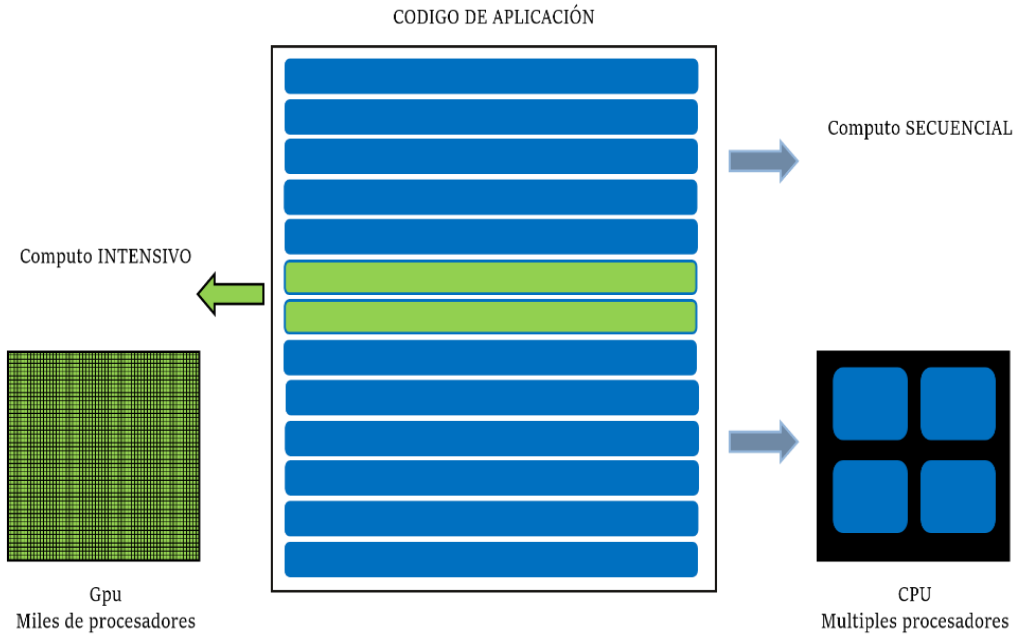


Figura 3.2. Aceleración por GPU de NVIDIA. Imagen de NVIDIA.

Hay que tomar en cuenta que la porción de código que se acelera en GPU tiene una mejora en el desempeño siempre y cuando tome ventaja del cómputo masivo en paralelo que ofrece la GPU. Es decir se debe aprovechar la arquitectura que ofrece la GPU y que no ofrece un CPU. En la Figura 3.3 se muestra un diagrama en el que se ilustra la arquitectura de una GPU NVIDIA Maxwell SMM. Una GPU tiene muchos más *cores*, o núcleos, que un CPU, sin embargo cada uno de los *cores* de la GPU está más limitado en cuanto a capacidades de cómputo y velocidad, debido a lo anterior es que la GPU y CPU tienen ventaja sobre diferentes formas de paralelización. El CPU tiene superioridad al emplear paralelismo de tareas mientras que el GPU tiene mejor desempeño al usar paralelismo de datos.

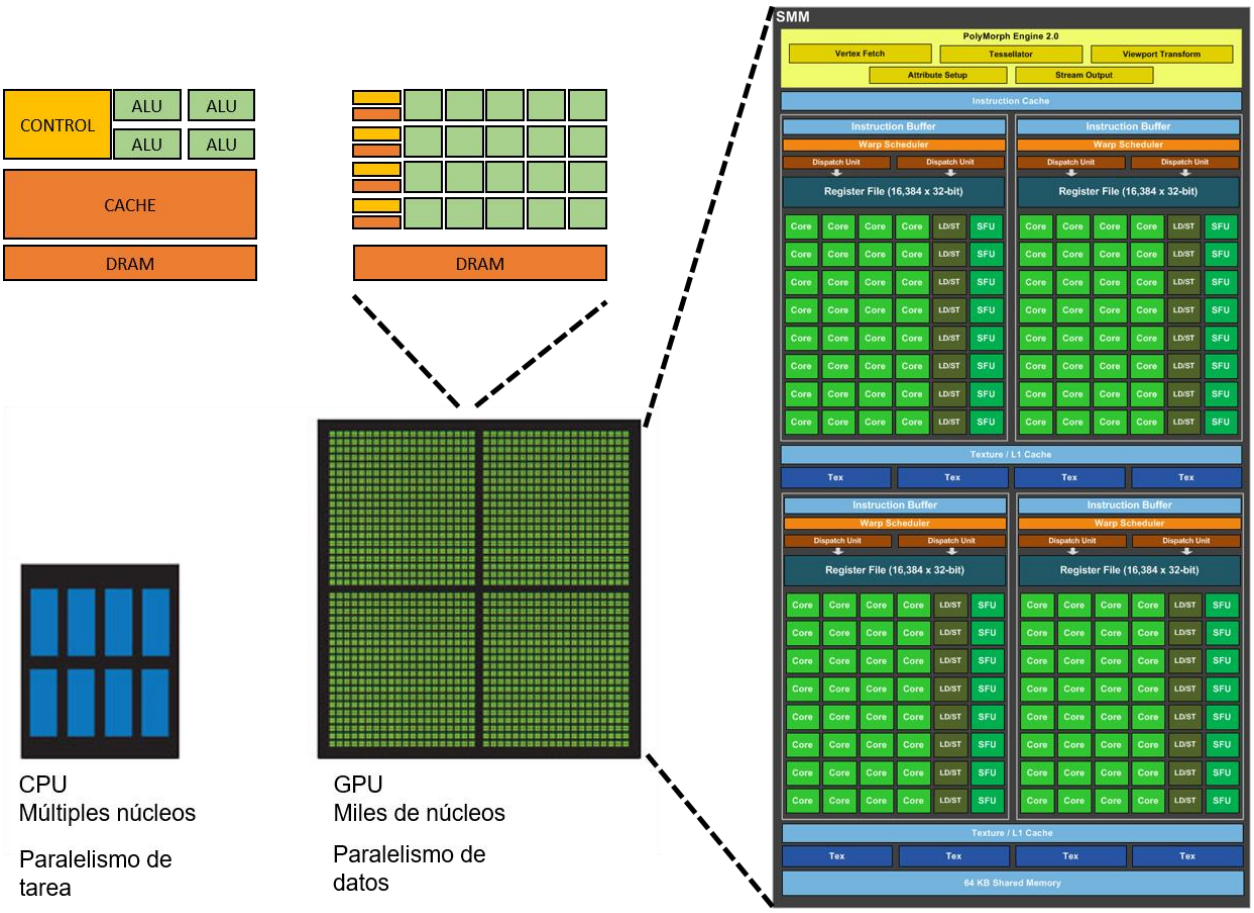


Figura 3.3. Comparación de CPU y GPU con énfasis en Diagrama de bloques de Maxwell SM.

Imagen de NVIDIA[11] .

En este trabajo se decidió trabajar con GPU de la compañía NVIDIA, por lo que a fin de implementar el aceleramiento por GPU se emplea la plataforma de cómputo en paralelo CUDA, la cual se implementa en conjunto con el lenguaje de programación C.

3.1. CUDA

En los últimos años las implementaciones de programas con ayuda de GPGPU han cobrado gran popularidad, principalmente porque científicos y desarrolladores se encuentran en búsqueda de nuevas formas para mejorar el desempeño de sus programas. Debido a que cada vez se busca que más programas exploten las capacidades de la aceleración por GPU es que se han desarrollado herramientas que permiten implementar programas que puedan ser ejecutados en GPGPU.

[11] NVIDIA WhitePaper, NVIDIA GeForce GTX Ti.

Dado lo anterior es que una de las compañías más grandes en cuanto a desarrollo y fabricación de GPU, NVIDIA, busca posicionar sus productos como herramientas capaces de acelerar el cómputo de cualquier tipo de aplicación o programa, para lograr eso es que ha desarrollado un lenguaje de programación: CUDA.

CUDA es el acrónimo en inglés para *Compute Unified Device Architechture*. CUDA es una arquitectura de cálculo en paralelo de NVIDIA la cual aprovecha el poder de procesamiento de la GPU. A diferencia de sus predecesores al emplear CUDA no es necesario que el programador posea conocimiento de graficación o primitivas de graficación, esto simplifica la labor de programación pues ya no es necesario modificar el planteamiento de los problemas que se desean resolver para funcionar con rutinas e instrucciones definidas en un API (*application programming interface*) de graficación. De hecho cualquier persona que sepa programar en el lenguaje de programación C puede programar en CUDA.

Aunque no es necesario tener conocimiento de graficación para usar CUDA es necesario conocer el paradigma de programación en paralelo, pues a fin de obtener un mejor desempeño en las aplicaciones es ideal hacer un planteamiento del problema que se beneficie del procesamiento en paralelo, una de las razones es porque las GPU están diseñadas para el cálculo masivo en paralelo y la otra es porque el poder de una GPU para la ejecución de programas secuenciales es menor a la que posee un CPU.

3.1.1. Flujo de proceso en CUDA.

En esta sección se describe de manera sencilla el flujo que debe tener un proceso cuando trabaja con CUDA. Hay que mencionar que una GPU no solo es un dispositivo que cuenta con procesadores únicamente, a fin de optimizar el trabajo de los procesadores una GPU cuenta con memoria sobre la cual operan los procesadores que contiene.

Dado que la memoria que emplean las GPU tiene una mayor velocidad a la memoria del CPU es que existen restricciones en cuanto a la cantidad de memoria que una GPU puede tener. Estas limitaciones se deben a costos, espacio y disipación de energía principalmente. A esta memoria que reside en la GPU se le nombra memoria de dispositivo (*device memory*), la nomenclatura *device* aplica para todo lo que resida en GPU, mientras que toda la memoria externa o de CPU se denomina memoria de huésped (*host memory*), del mismo modo se denomina *host* a todo lo que reside fuera del GPU.

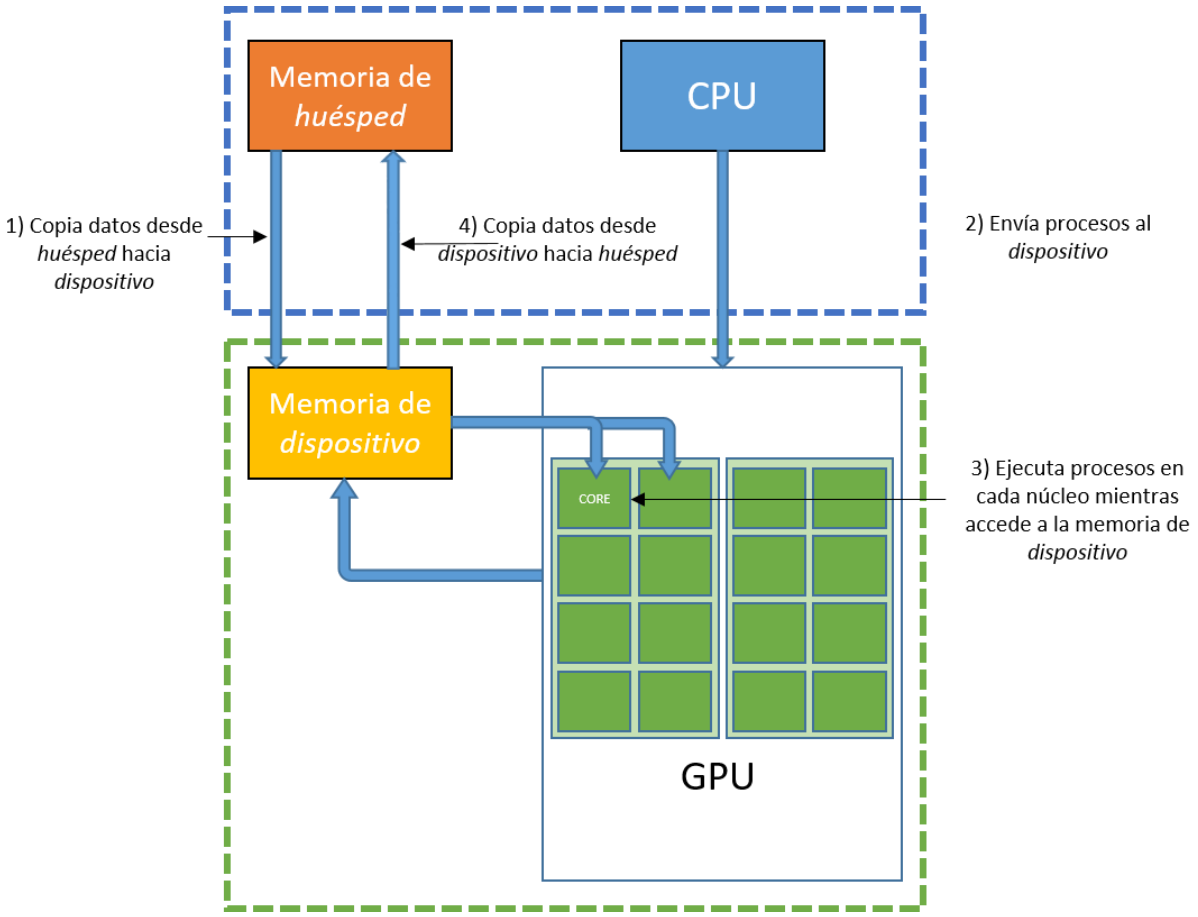


Figura 3.4. Flujo de proceso en CUDA.

En la Figura 3.4 se muestran los cuatro pasos principales que se siguen durante la ejecución de un proceso el cual es acelerado por GPU mediante CUDA. Al inciar el proceso la ejecución es completamente en CPU, por lo que la memoria con la que trabaja el programa en CPU o *host memory* debe ser copiada al *device memory* para posteriormente poder ser usada por la GPU. Ya que se tiene la memoria asignada en *device memory* el CPU debe enviar las instrucciones al *device* para que sean ejecutadas por la GPU. Las instrucciones son ejecutadas de manera paralela por los *cores* de la GPU. Los *cores* únicamente pueden trabajar con la memoria de *device*, así como el CPU solo puede trabajar con el *host memory*, por lo que al finalizar el proceso de los datos de la GPU el contenido del *device memory* debe actualizar al *host memory* correspondiente a fin de continuar con el flujo del proceso.

Debido a la manera en que está construido el flujo de proceso en CUDA es que surge un cuello de botella que afecta el desempeño de una aplicación, el proceso de copiado de memoria entre *device* y *host*. La trasferencia de datos entre *host* y *device* es un paso ineludible y que siempre estará limitado por la velocidad de los canales de datos que permiten la transmisión de datos, por lo que a fin de no afectar el desempeño que se busca ganar con la aceleración por GPU, es necesario transferir solo los datos indispensables la menor cantidad de veces que sea posible y de preferencia hacerlo de manera que se eficiente el uso del canal de datos.

3.1.2. Modelo de programación.

Como se mencionó anteriormente el modelo de programación de CUDA requiere que el programador modifique su código con un enfoque de un solo hilo a un enfoque de multi-hilos en el cual cada hilo puede interactuar con los demás. Es obvio que el programador puede ya tener un enfoque en el que su programa se ejecuta en múltiples hilos, sin embargo, al implementarlo en CUDA se debe implementar en forma de paralelismo de datos (*data parallelism*) y no bajo un esquema de paralelismo de tarea (*task parallelism*).

Al implementar los programas mediante paralelismo de datos se obtiene un beneficio en el desempeño en GPU, pues el modelo de SIMD (*Single Instruction Multiple Data*) es el modelo para el que fue creada la arquitectura de la GPU, ya que el paralelismo de datos es un paralelismo en el que se ejecuta una misma instrucción en un gran conjunto de datos mientras que el paralelismo de tarea se enfoca en tener diferentes procesos en paralelo los cuales ejecutan diferentes instrucciones.

En el caso de un CPU se obtiene un mayor beneficio al implementar paralelismo de tarea debido a que los núcleos de los procesadores tienen mucho mayor poder de cómputo y pueden realizar procesos más complejos, esto no ocurre en los núcleos de las GPU, pues estos núcleos están enfocados principalmente al cálculo extensivo de datos por lo que en términos de capacidad de cómputo son mucho más limitados que los núcleos de un CPU.

En este trabajo solo mencionaremos los *kernels* de CUDA[12] ya que son la herramienta que proporciona mayor versatilidad al emplear CUDA y sobre la cual se realiza la mayor parte de la programación.

Los *kernels* de CUDA son funciones que define el programador mediante la extensión del lenguaje de programación C usando CUDA C. Recordemos que el flujo de proceso de CUDA considera la comunicación entre dos entidades denominadas *host* y *device*, los *kernels* son mandados llamar desde el *host*, pero su ejecución se realiza en *device*. Dado que los *kernels* se ejecutan en *device* es posible ejecutar un número N_h de estas funciones de manera paralela mediante la utilización de un número N_h de hilos de CUDA.

Los *kernels* de CUDA son ejecutados como una rejilla o *grid* en inglés, la cual contiene bloques (*blocks*) y estos a su vez tienen hilos (*threads*) en los cuales se ejecutan las funciones definidas previamente. En las GPU actuales hay un límite para la cantidad de hilos por bloque que se pueden mandar a ejecutar, en la Tabla 3.1 se muestran las especificaciones técnicas en cuanto a dimensión de bloques e hilos de las GPU que tiene una capacidad de computo superior a 3.0 (*Compute Capability* definida por NVIDIA), este valor está definido por NVIDIA en base a los avances en cuanto a la arquitectura de sus GPU.

[12] https://docs.nvidia.com/cuda/cuda-c-programming-guide/

Tabla 3.1. Especificaciones técnicas para GPU con Compute Capability 3.0 o superior.

Dimensión máxima del bloque de hilos de una grid	3
Tamaño máximo en dimensión X de bloque de hilos de una grid	$2^{31} - 1$
Tamaño máximo en dimensión Y o Z de bloque de hilos de una grid	65,535
Máxima dimensión de bloque de hilos	3
Tamaño máximo en dimensión X o Y de bloque de hilos	1,024
Tamaño máximo en dimensión Z de bloque de hilos	64
Número máximo de hilos por bloque	1,024
Número máximo de instrucciones por kernel	512 millones

Como se muestra en la tabla la dimensión de cómo se manejan los bloques de una *grid* se puede expresar con un vector de 3 dimensiones, existe la misma condición para los hilos de cada bloque, en la Figura 3.5 se muestra un ejemplo de la organización de los bloques e hilos en una GRID de los *kernels* de CUDA.

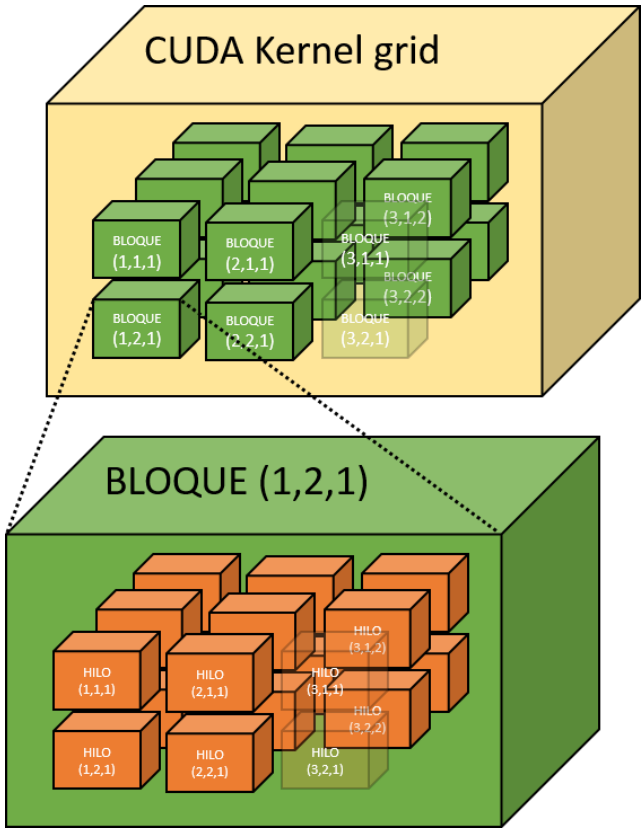


Figura 3.5. GRID de bloques de hilos.

Cuando se invoca a un *kernel* de CUDA es necesario especificar la cantidad de hilos y bloques con los que se construirá la *grid* que ejecutará el *kernel* en la GPU. Para esto se multiplica la cantidad de Bloques por la cantidad de hilos por bloque (* *producto de escalares*), dependiendo de la cantidad de dimensiones con las que se defina los bloques en la *grid* y los hilos en los bloques se multiplica cada uno de los tamaños en cada una de las tres dimensiones con la premisa de que cualquier valor que no es especificado es igual a 1.

$$Número\ de\ Bloques = Bloque\_X * Bloque\_Y * Bloque\_Z$$
$$Número\ de\ Hilos\ por\ Bloque = Hilo\_X * Hilo\_Y * Hilo\_Z$$
$$Número\ de\ Hilos\ en\ grid = Número\ de\ Hilos\ por\ Bloques * Número\ de\ Bloques$$

Se debe tener en cuenta que el número máximo de hilos por cada bloque es de 1024 mientras que el máximo de bloques en una *grid* es de $2^{31} - 1$, esto significa que al momento de invocar el *kernel* se debe especificar el número de hilos que se desean ejecutar de manera paralela siguiendo las especificaciones mostradas en la Tabla 3.1. Suponiendo que queremos ejecutar 1 millón de hilos de manera paralela podemos indicar a CUDA que la *grid* tenga las siguientes dimensiones:

Tabla 3.2. Ejemplos de dimensión de hilos y bloques para conformación de grid.

Bloques (x,y,z)	**Hilos por bloque (x,y,z)**	**Total de hilos en *grid***
(1000 , 1 , 1)	(1000 , 1 , 1)	1,000,000
(10 , 10 , 10)	(10 , 10 , 10)	1,000,000
(1000000 , 1 , 1)	(1 , 1 , 1)	1,000,000
(100 , 100 , 1)	(100 , 1 , 1)	1,000,000

En los ejemplos mostrados en la Tabla 3.2 se muestra que la dimensión de los bloques y los hilos puede ser variada a fin de buscar obtener el mejor desempeño posible, este es un tema muy complicado, ya que a fin de obtener el mejor desempeño es necesario conocer de manera muy detallada la manera en que funciona la arquitectura de la GPU, así como las capacidades de los *cores* con los que trabaja la misma.

Debajo se muestra una porción de código en CUDA donde se invoca un *kernel* de CUDA llamado *MatAdd*, usando $N * N$ hilos. La palabra reservada __global__ define las funciones que son ejecutadas en GPU al usar CUDA C.

```
//Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
   int i = threadIdx.x;
   int j = threadIdx.y;
   C[i][j] = A[i][j] + B[i][j];
}
int main() {
   ...
   // Kernel invocation with one block of N * N * 1 threads
   int numBlocks = 1;
   dim3 threadsPerBlock(N, N);
   MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
   ...
}
```

En la porción que se muestra debajo se muestran las funciones cudaMemcpy y cudaMalloc, las cuales son funciones de CUDA para realizar copia y reserva de memoria respectivamente.

```
// Codigo de Device
__global__ void SumaVector(float* A, float* B, float* C, int N)
{
int i = blockDim.x * blockIdx.x + threadIdx.x;
if (i < N) C[i] = A[i] + B[i];
}
// Codigo de Host
int main()
{
int N = ...;
size_t size = N * sizeof(float);
// Reserva memoria a vectores de entrada h_A and h_B y vector h_C en memoria de Host
float* h_A = (float*)malloc(size);
float* h_B = (float*)malloc(size);
float* h_C = (float*)malloc(size);
// Inicializa vectores de entrada ...
// Reserva memoria de vectores en memoria de Device
float* d_A;
cudaMalloc(&d_A, size);
float* d_B;
cudaMalloc(&d_B, size);
float* d_C;
cudaMalloc(&d_C, size);
// Copia vectores de Host a Device
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
// Invoca kernel
int threadsPerBlock = 256;
int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
VecAdd <<<blocksPerGrid, threadsPerBlock >>>(d_A, d_B, d_C, N);
// Copia el resultado de memoria de Device a memoria de Host
// d_C contiene el resultado y se copia a h_C
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
// Libera memoria de Device
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
// Libera memoria de Host
...
}
```

No se entra en detalle en este trabajo en cuanto a instrucciones del API de CUDA por lo que se recomienda consultar la bibliografía y documentación oficial de NVIDIA[13].

3.2. Flex

Flex es una biblioteca de simulación basada en partículas, esta biblioteca toma como base la dinámica basada en posición de los métodos libres de malla que se abordó en el Capítulo 2 y el cual forma parte del trabajo de Müller et al [7] y Miles Macklin et al [8]. En la Figura 3.6 se observa un ejemplo de cuerpos rígidos, suaves y resortes implementado con Flex.

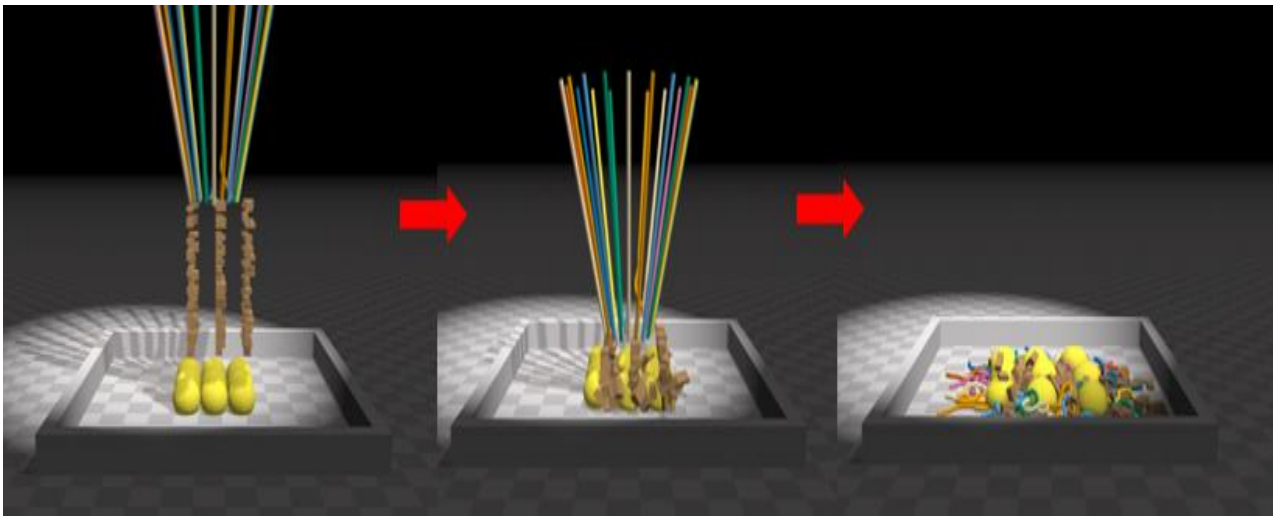


Figura 3.6. Implementación de resortes, cuerpos rígidos y cuerpos deformables usando Flex.

Al usar Flex se asume que todo se puede modelar mediante un sistema de partículas conectadas por restricciones, desde objetos rígidos o suaves, hasta fluidos y gases por dar algunos ejemplos. Una ventaja de tener una representación unificada es que permite el modelado eficiente de diferentes materiales y al mismo tiempo permite interacción entre los distintos tipos de cuerpos, es decir que es posible la interacción entre fluidos y cuerpos rígidos de manera bidireccional. En la Figura 3.7 se muestra un ejemplo de interacción bidireccional entre cuerpos rígidos y cuerpos deformables, así como entre fluidos y cuerpos rígidos.

[13] https://docs.nvidia.com/

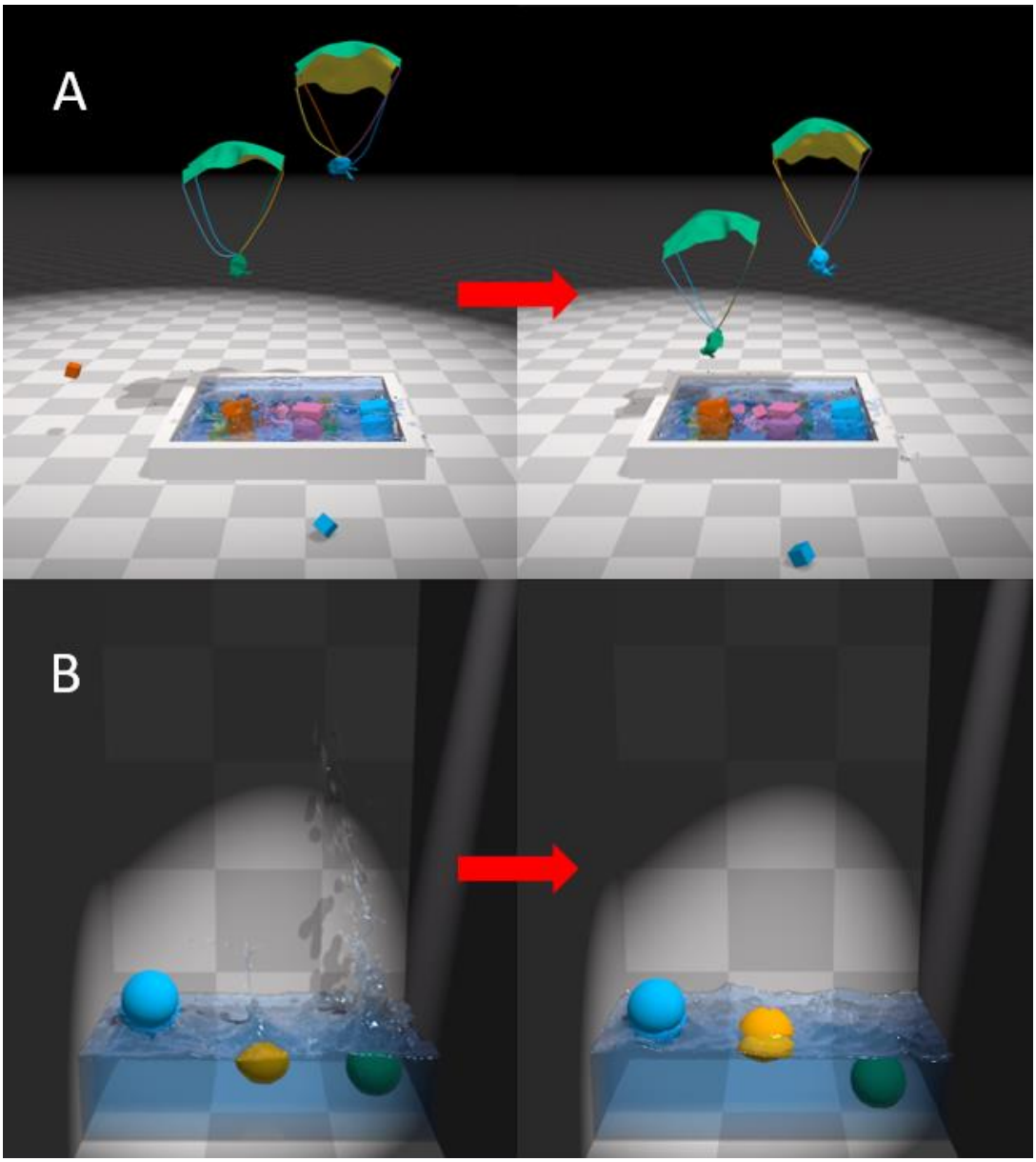


Figura 3.7. A) Interacción bidireccional entre cloth y cuerpo rígido. B) Interacción bidireccional entre fluido y cuerpo rígido.

Flex emplea como bloque fundamental de construcción a las partículas. Éstas tienen propiedades como posición, velocidad y el inverso de la masa. Adicional a lo anterior Flex permite incluir un valor denominado *Phase*, el cual se emplea para *controlar* el comportamiento de diferentes conjuntos de partículas. Dependiendo de los configuración del parámetro *Phase* se pueden construir fluidos que colisionen consigo mismo y generen restricciones de densidad, también se pueden crear cuerpos rígidos que no colisionen consigo mismo y solo colisionen con otros grupos.

Una característica que se debe mencionar es que todas las partículas que se usan en Flex poseen el mismo radio, además dicho valor tiene un impacto significativo en el comportamiento y desempeño de la simulación.

Como se mencionó anteriormente Flex permite simular mediante partículas y restricciones diferentes tipos de cuerpos, dependiendo de las restricciones que tengan las partículas es que se obtienen diversos comportamientos. Lo anterior hace posible el modelado de cuerpos rígidos, cuerpos suaves, fluidos, resortes y *cloth* (*tela elástica*). En la Figura 3.8 se muestra la implementación de *cloth* usando Flex, como se observa en dicha figura, el objeto presenta interacción dinámica, además de que se puede modificar su topología al ser *desgarrada*. Éste evento de *desgarre* es un ejemplo de las ventajas de las métodos libres de mallas y cuya implementación en métodos basados en mallas, como FEM, es muy complicado.

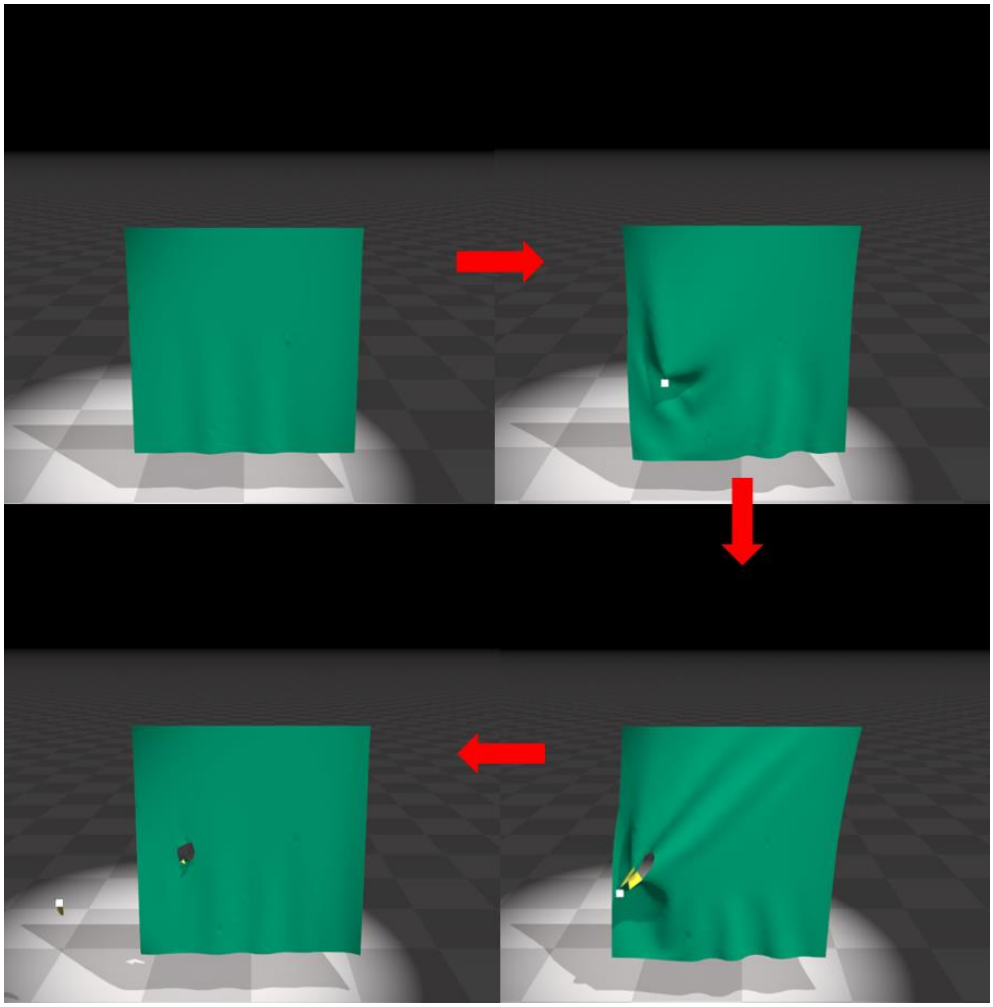


Figura 3.8. Desgarramiento de cloth (tela elástica) implementado usando Flex.

La implementación de fluidos en Flex está basada en el método de Fluidos basados en posición (*Position Bases Fluids*) de Miles [9]. Éste método emplea restricciones basadas en densidad y posición a fin de dar mayor estabilidad en comparación con los métodos de SPH tradicionales.

El comportamiento del fluido depende de la relación entre el radio de la partícula y la distancia de reposo del fluido, usualmente dicho valor es 2:1. Flex se encarga de calcular la densidad de reposo del fluido empleando la distancia de reposo y el radio de las partículas. Otros comportamientos que simula Flex son cohesión, tensión superficial y adhesión. La cohesión es generada con eventos de atracción y repulsión calculados con la distancia de reposo y la restricción de densidad.

En Flex es posible modelar resortes mediante pares de partículas a las que se les asigna una longitud de reposo y coeficiente de rigidez. Los resortes son modelados como restricciones de distancia con una rigidez definida por el usuario, es posible usarlos para modelar diferentes efectos como es el desgarramiento o unión.

Los cuerpos rígidos en Flex se modelan mediante la posiciones de reposos que tiene las partículas con respecto al centro de masa del cuerpo que modelan. Por otro lado Flex permite emplear partículas para representar mallas triangulares.

Algunas de las limitaciones de los métodos basados en partículas provienen del intervalo de tiempo elegido para la simulación, pues en algunos casos cuando el intervalo es muy grande las partículas pueden penetrarse y quedar atrapadas entre ellas mismas.

Tabla 3.3. Fortalezas y debilidades de cuerpos rígidos implementados en Flex.

Fortaleza	***Debilidades***
Escombro ambiental	Objetos delgados o afilados
Apilado no estructurado	Apilado estructurado
Forma no convexa	Tamaños relativamente grandes
Interacción bidireccional	Colisión filtrada

En la Tabla 3.3 se muestran algunas de las debilidades y fortalezas con que se cuentan los cuerpos rígidos implementados con Flex.

En el capítulo 2 se hizo referencia a la estructura *cloth* dado que a partir de ésta es posible implementar cuerpos deformarles o suaves en Flex, aunque hay que tomar en cuenta que es posible modelar cuerpos deformables al reducir la rigidez de los cuerpos rígidos, pero esto no es lo suficientemente acertado para deformaciones de gran escala. En Flex, los cuerpos deformables se simulan mediante mallas superficiales cerradas, las cuales se les asigna una presión interna de forma similar a un cuerpo inflable. Es posible modelar de manera limitada tetraedros de esta manera, pero no es algo que soporte Flex, uno de los motivos principales es debido a que el generar tetraedros no es posible mantener la regla de que una arista sea compartida a lo más por 2 triángulos, lo cual tiene implicaciones en la resolución de colisiones con los triangulos.

En la Figura 3.9 se muestra la implementación de cuerpos deformables mediante *cloth* cerrada a la que se le asigna una presión interna que infla el cuerpo como un globo.

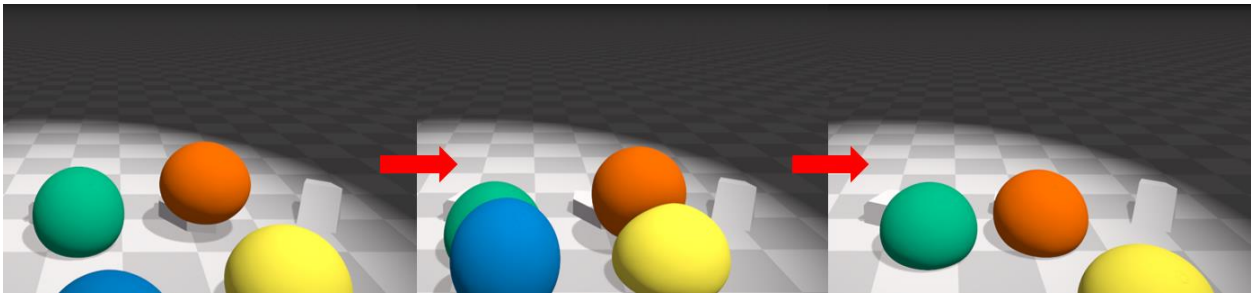


Figura 3.9. Cuerpos deformables implementados en Flex empleando cloth.

Flex emplea el API de ejecución de CUDA de manera interna, por lo que es posible usar cualquier contexto de CUDA, ya sea uno creado por el programador o el que genera Flex por omisión. Como se ha mencionado anteriormente es importante que se minimice la sincronización entre CPU y GPU (transferencia de memoria entre *host* y *device*).

3.3. OpenGL

Cuando se trabaja con aplicaciones de despliegue gráfico las dos opciones más populares son OpenGL y DirectX. La elección de cualquiera de estas bibliotecas debe ser analizada por el programador dado que hacer migración entre éstas consume tiempo de desarrollo, pero dado que ambas satisfacen las necesidades de la mayoría de las aplicaciones con despliegue grafico es que la selección usualmente depende más de opinión personal y familiaridad con cualquiera de los APIs.

OpenGL fue creado en 1990 a partir de la necesidad que observo la compañía Silicon Graphics de un API de graficación en 3D en la industria de software y la investigación; en la actualidad OpenGL es administrado por el grupo Kronos. OpenGL provee una interfaz de programador con la tarjeta de despliegue de gráficos, es decir opera con la mayoría de las GPU que tienen capacidades de despliegue gráfico.

El API de OpenGL especifica primitivas de representación, propiedades que se asignan a dichas primitivas, una vista a fin de desplegar la información representada por las primitivas y sus propiedades mediante la aplicación de ciertos procesos de cómputo ya establecidos.

Dado que buscamos implementar técnicas de simulación que aprovechen el poder de cómputo de la GPU, es igualmente necesario implementar el despliegue gráfico con APIs que aprovechen del mismo modo la GPU. Dado que en este trabajo se ha decidido trabajar con OpenGL es que se ha optado por usar OpenGL 4.4.

Para entender por qué se eligió OpenGL 4.4 es necesario revisar el pipeline de graficación a fin de identificar los puntos que explotan las capacidades de cómputo de la GPU. En la Figura 3.10 se muestra un diagrama con el pipeline de graficación de OpenGL.

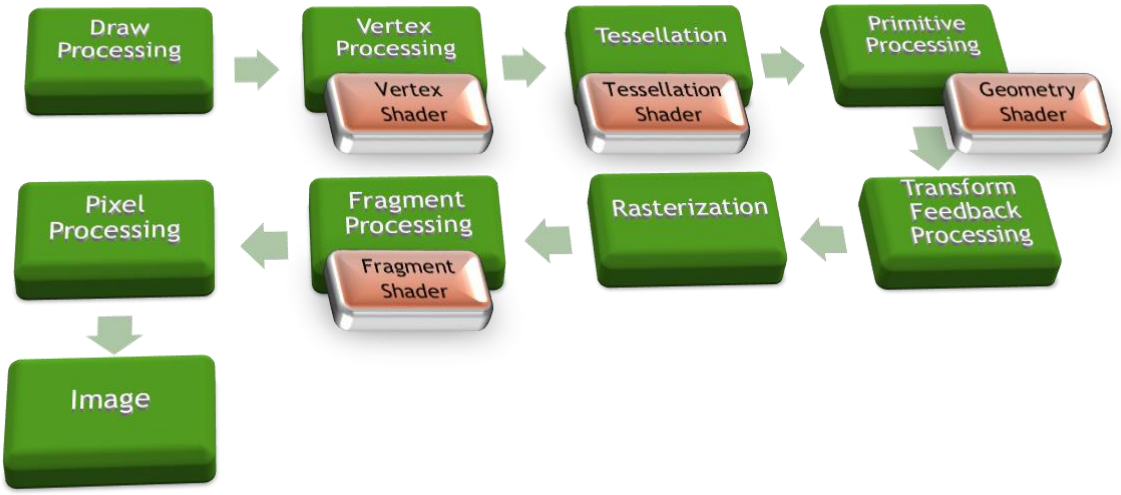


Figura 3.10. Pipeline de graficación de OpenGL.

Las operaciones de mayor importancia de las que se deben realizar en una aplicación OpenGL a fin de producir el renderizado de una imagen son:

- Especificar datos para la construcción de formas por medio de las primitivas geométricas de OpenGL.
- Ejecución de *shaders* para realizar los cálculos en las primitivas de entrada para determinar sus posiciones, color y otros atributos de renderizado (*vertex shader*).
- Convertir la descripción matemática de las primitivas de entrada en elementos discretos a los cuales se les asocia una posición en pantalla (*rasterization*), estos segmentos reciben el nombre de fragmentos y contienen los datos de posición en 3D y un valor de acuerdo a una plantilla.
- Ejecución de un *fragment shader* para cada uno de los fragmentos generados por la rasterización, este paso genera el color final de cada fragmento y su posición.
- Realizar operaciones adicionales por fragmento según sean requeridas, por ejemplo determinar la visibilidad de fragmentos, mezcla de colores, etcétera.

El *pipeline* de renderizado de OpenGL 4.3 (el cual no emplea Fixed Function Pipeline) está basado en el uso de *shaders* a fin de procesar los datos que se le ingresan. A decir verdad el único proceso de renderizado que se puede realizar sin *shaders* es el limpiado de pantalla. Los *shaders* son escritos usualmente en un lenguaje de programación especializado, en el caso de OpenGL este lenguaje es GLSL. Una de las ventajas de los *shaders* es que ocupan el poder de cómputo de las GPU, por lo que tiene un mejor desempeño al momento de realizar cálculos relacionados con el renderizado.

El *pipeline gráfico* tiene 4 etapas de procesamiento, cada una de estas etapas es controlada usando *shaders* que provee el programador. Con base en la Figura 3.10 ubicamos cuatro *shaders* de la siguiente manera:

- *Vertex Shader* (*Vertex Processing*). Recibe los datos que se especifican en los objetos de almacenamiento de vértices (*vertex-buffer objects* o VBO) y procesa de manera separada cada vértice.

- *Tessellation Shader* (*Tessellation*). Se emplea para generar geometría adicional dentro del *pipeline* de OpenGL. Este *shader* recibe como entrada la salida del *vertex shader*.

- *Geometry Shader* (*Primitive Processing*). Opera sobre las primitivas geométricas individuales permitiendo la modificación de cada una de ellas, por ejemplo cambiar triángulos en líneas.

- *Fragment Shader* (*Fragment Processing*). Este *shader* recibe los fragmentos individuales generados por el rasterizado de OpenGL, con lo que calcula color valores de color y profundidad.

Algunos de los procesos que efectúan dichos *shaders* son opcionales, sin embargo el proceso realizado por el *vertex shader* no es opcional.

Los VBO son característicos de OpenGL, los cuales proveen métodos para ingresar datos de vértices al dispositivo de video (GPU) para renderizado no inmediato. Esta memoria además de residir en la GPU provee de un gran desempeño ya que puede ser renderizada directamente por la tarjeta gráfica, cosa que no ocurre cuando se emplea la memoria del sistema (CPU).

En este trabajo no se entra en detalle en cuanto a cómo usar el API de OpenGL 4.4, solo se hace referencia a ella dado que a fin de desplegar los resultados obtenidos con los métodos de simulación que son implementados en GPU es necesario aprovechar las GPU para realizar el despliegue gráfico sin afectar negativamente el rendimiento de la aplicación.

3.4. Interoperabilidad entre CUDA y OpenGL

Una de las ventajas de trabajar con CUDA (Flex) es la interoperabilidad que existe entre OpenGL y CUDA. La interoperabilidad se basa en concepto muy simple, el mapeo de memoria.

El mapeo al que se hace referencia ocurre dentro de la memoria de GPU únicamente, es decir la interoperabilidad trabaja en el espacio conocido como *device*, gracias a esto es que la interoperabilidad evita el cuello de botella que significa hacer copia de memoria entre *host* y *device*. La interoperabilidad no afecta el desempeño pues comparte datos a través de memoria común.

La interoperabilidad aprovecha la capacidad de CUDA para el cálculo, generación de datos, procesamiento de imagen y demás procesos implementados en GPU. Por otro lado se aprovecha OpenGL para dibujar píxeles en pantalla. En el diagrama 3.11 se muestra un diagrama en el que se relaciona el *pipeline* de OpenGL junto con los procesos implementados en CUDA.

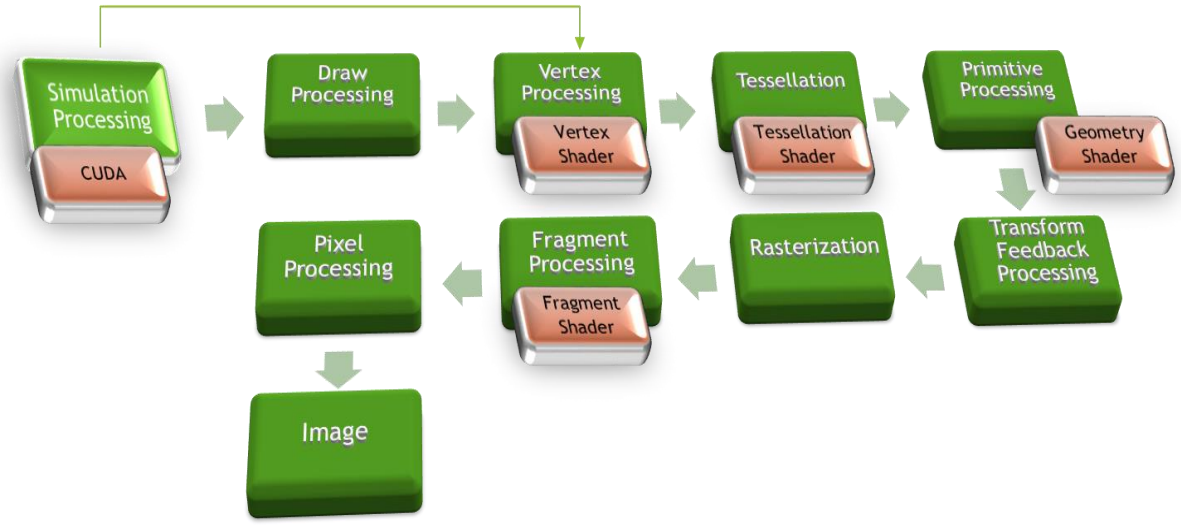


Figura 3.11. Pipeline de graficación de OpenGL con interoperabilidad con CUDA mapeando memoria de vértices.

En el diagrama anterior se muestra como se emplea CUDA para realizar cálculos que modifican vértices los cuales son usados posteriormente para renderizar una imagen. Si esto lo aplicamos a un ambiente virtual que despliega los resultados de una simulación tenemos que CUDA contiene la implementación de los métodos de simulación. Ésta simulación tiene como principal objetivo calcular las posiciones de los vértices que conforman las mallas de los objetos que se encuentran en el ambiente virtual. Tanto OpenGL como CUDA tienen acceso a la misma memoria mediante el mapeo de la región común, por lo que al momento que se tienen los resultados calculados por CUDA es posible graficarlos sin la necesidad de hacer transferencia de memoria entre *host* y *device*. Es necesario permitirle a OpenGL mapear la memoria que modificó CUDA para poder iniciar el *pipeline* de graficación, del mismo modo es necesario que cuando OpenGL termine el proceso de renderizado permita a CUDA mapear la memoria para realizar los cálculos necesarios en los vértices.

La interoperabilidad tiene implementados varios mecanismos que permiten mantener la integridad de datos, por lo que se debe poner atención especial al mapeo de la memoria, es decir OpenGL y CUDA tienen acceso a la memoria mapeada de manera *excluyente*, mientras uno de los procesos la esté usando el otro proceso debe esperar hasta que la libere. CUDA tiene mecanismos que permiten hacer el mapeo de memoria mediante instrucciones de CUDA, sin embargo, es labor del programador conocer el flujo a fin de mapear y liberar dicha memoria para que CUDA y OpenGL puedan emplearla.

En este trabajo no se entra en detalle en cuanto a las instrucciones necesarias para hacer el mapeo de la memoria, solo se hace una descripción general dado que su uso es necesario para no afectar el rendimiento de una aplicación grafica acelerada por GPU.

Capítulo 4. Simulación de cortes en un cuerpo deformable.

Los simuladores quirúrgicos son cada día más empleados para el entrenamiento de cirujanos [11], principalmente porque que se reduce el riesgo que corren los pacientes con los que se realiza el entrenamiento. Uno de los desafíos que enfrentan los simuladores quirúrgicos que emplean simulaciones con mallas es la simulación de cortes en tejido suave. Aunque existen métodos que permiten eliminar tetraedros o fracturar la malla separando los elementos que conforman la malla, dichos métodos son muy complejos y permiten cortes relativamente simples, o bien generalmente aplican solo para cuerpos rígidos como es el caso de los métodos implementados en Bullet Physics y Nvidia PhysX.

La implementación de simulación de cortes en cuerpos suaves es un problema complejo para el cual existen diversas soluciones para los métodos que emplean mallas. Los métodos más estudiados consisten en la subdivisión de tetraedros [14], dichos métodos consisten en identificar las diferentes configuraciones sobre las cuales se produce el corte, esto está en función de un plano de corte. Uno de los problemas con este método es la alta combinatoria de los casos en que se puede subdividir la malla dependiendo de la trayectoria de corte, la dificultad geométrica de mantener la malla de manera conformal después de los cortes, además de que mientras más cortes se realicen más se incrementa el número de elementos que conforma la malla, debido al refinamiento de la misma.

Existen métodos más recientes en los que se reduce el número de elementos agregados a cada corte mediante la eliminación y conservación de nodos y aristas mediante un criterio de distancia al plano de corte [15], sin embargo dichos métodos no aplican para simulaciones en las que se acumulen cortes.

Otro problema crítico al usar las técnicas de subdivisión de tetraedros en conjunto con los métodos de elemento finito (FEM) es que al retirar o agregar nuevos elementos a la simulación es necesario recalcular la matriz fundamental que modela el sistema (matrices de rigidez en el caso estático o matrices de rigidez, masa y viscosidad para el caso dinámico), dicho cálculo es costoso computacionalmente, lo cual limita su uso durante la simulación. Más aún una vez recalculadas las matrices del sistema es necesario invertirlas en tiempo real a fin de computar los desplazamientos de los vértices en función de las fuerzas externas al sistema

Para evitar el problema de subdivisión de tetraedros una alternativa para simular cortes es aplicar un efecto plástico, mediante la compresión los tetraedros que conlleva a la reducción de la masa de los elementos que conforman la malla, con lo anterior se evita el cálculo de la matriz fundamental, sin embargo la reducción de la masa es un proceso computacional costoso y poco realista. Existe otra alternativa la cual es predefinir los cortes, pero se afecta la estructura de la malla lo cual regresa al problema inicial que es la afectación de la matriz fundamental que define el modelo físico de deformación de la malla.

Debido a las complicaciones que existen para lograr una simulación de cortes en cuerpos deformables con métodos basados en mallas es que en este trabajo se presenta una alternativa que emplea métodos libres de mallas la cual fue implementada con el método de dinámica basada en posición revisado en el capítulo 2.

4.1. Método propuesto.

El método propuesto parte de la definición de cuerpos mediante el uso de partículas. Dichas partículas están unidas mediante restricciones, estas restricciones permanecen constantes durante la simulación siempre y cuando el objeto solo se deforme. Lo anterior implica que las partículas tienen definido un conjunto de vecinos los cuales no cambian durante todo el proceso.

A partir de las partículas, definimos el corte como la modificación de las restricciones iniciales de una partícula, estas restricciones son solo aquellas que definen la vecindad de la partícula y que pertenecen al mismo objeto, como se explica a continuación.

Sea V el conjunto de partículas que define un cuerpo, el evento de corte definido sobre una partícula $p \in V$, con respecto a su vecindad $p_i' \in V$, se realiza conforme a los planos de corte π_i que son definidos con los vectores $\bar{n}_i = \frac{p_i' - p}{|p_i' - p|}$ y los puntos $q_i = p + \frac{p_i' - p}{2}$ que son calculados a partir de la partícula p y su vecindad p_i'.

El primer paso en el evento de corte es identificar las restricciones que ligan a la partícula p, la cual se va a separar por efecto del corte de sus vecinos p_i', una vez que se identifican dichas relaciones se generan los planos de corte π_i. Con los planos de corte es evalúan las partículas S que es el conjunto formado por las partículas que interaccionan debido a las restricciones con la partícula p, es decir $S = \{s_j \in V \,|s_j \in p \cup p_i\}$ que se separan del conjunto V, para eso se emplea la relación 4.1.

$$corte(S) = \begin{cases} (s_j + q_i) \cdot \overline{n_\iota} > 0, separa\ partícula\ s_j \\ (s_j + q_i) \cdot \overline{n_\iota} < 0, duplica\ partícula\ s_j \end{cases} \quad \ldots\ldots\ldots\ldots (4.1)$$

Sean a, b los índices de las partículas en S, se definen las restricciones $C_{a,b}$, donde la partícula que se separa es s_a y las partículas que se mantienen unidas son s_b. Por cada partícula que se mantiene ligada y pertenece a la vecindad de s_a se crea una partícula s_b', la cual es idéntica en propiedades de posición, velocidad y masa. A esta nueva partícula se le asignan las restricciones $C_{a,b}$. Dado que las partículas s_b' no están conectadas a las partículas de V, es que es posible separar la partícula s_a dando lugar a un conjunto $V' = \{s_a \cup s_b'\}$ el cual define un nuevo objeto que representa a la porción que ha sido cortada.

En la Figura 4.1 se muestra un diagrama donde se ilustra el corte con respecto a una partícula p, la cual debe ser separada de sus vecinos por la acción de corte.

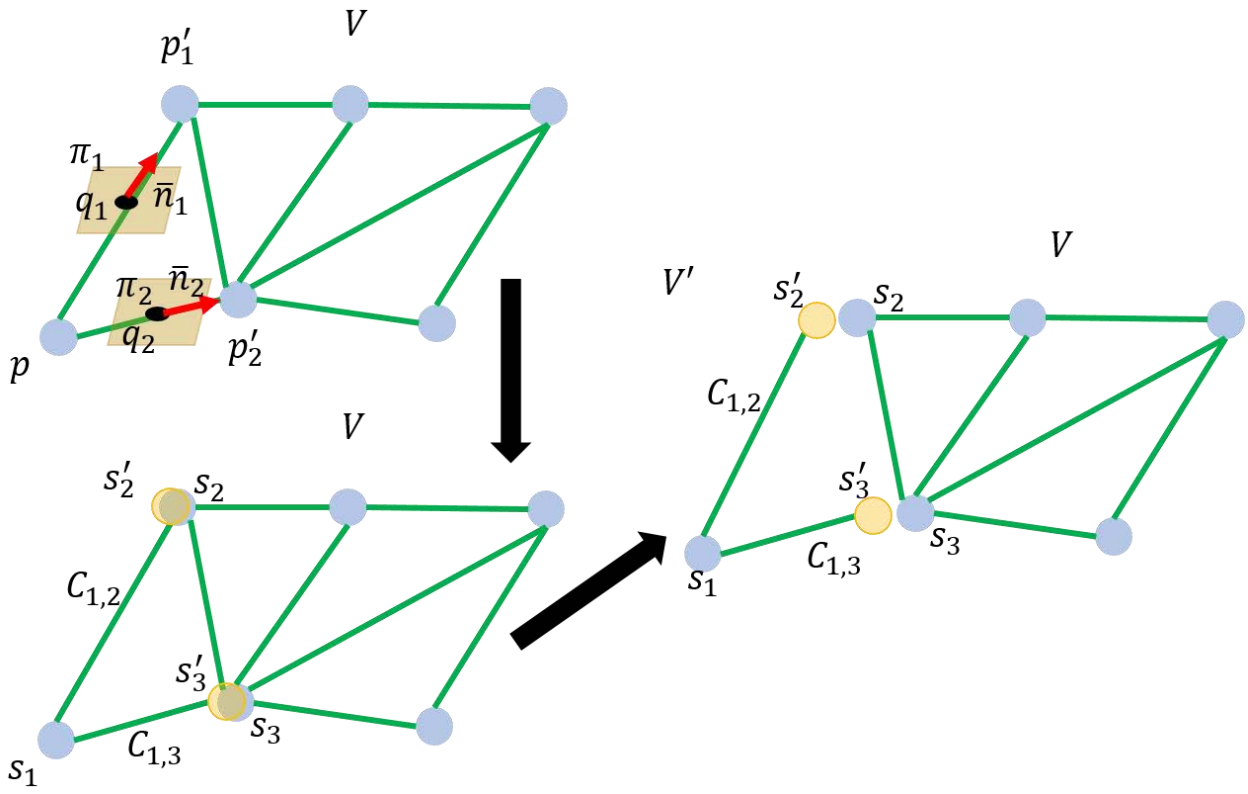


Figura 4.1. Diagrama de corte de una partícula p.

El método propuesto para generar los cortes en cuerpos suaves crea nuevas partículas que no están unidas al objeto original, dado que el proceso de remoción de partículas genera un nuevo objeto que representa el trozo de tejido debido al corte. El método de corte genera esas nuevas partículas debido a que no se desean eliminar restricciones, pues se busca mantener constante el número de restricciones originales para mantener la estabilidad del método de dinámica basada en posición. Lo anterior obedece a que el cálculo y modificación de restricciones durante el proceso de simulación demanda mucho poder de cómputo y es preferible emplearlo para generar las restricciones originadas por las colisiones entre partículas.

4.1.1. Implementación de cortes empelando Flex.

La implementación de cortes se realizó con Flex, el principal motivo es que éste emplea un método de simulación libre de mallas consistente en dinámica basada en posición. Con este método se pueden modelar cuerpos deformables que tiene un comportamiento elástico, y como ya se mencionó, eso se logra empelando la estructura de simulación de *cloth*. Sin embargo, la implementación de *cloth* en Flex solo soporta superficies cerradas, por lo que fue necesario modificar la construcción de mallas de *cloth* a fin de poder representar volúmenes mediante tetraedros.

Las propiedades que se tienen que modificar a fin de generar volúmenes basados en *cloth* son la restricciones de las aristas, pues originalmente estas solo permiten que una arista sea compartida como máximo por 2 triángulos. Otro aspecto importante es que en la implementación original de *cloth* se definen restricciones de estiramiento $C_{stretch}$ y restricciones de flexión C_{bend}, estas últimas brindan restricciones que le brindan resistencia al modelo en cuanto a la modificación del ángulo inicial entre los triángulos que conforman la superficie que modela el *cloth*, pero al implementar una estructura volumétrica se vuelven redundantes las restricciones de flexión por lo que solo se trabaja con restricciones de estiramiento.

En la Figura 4.2 se muestra por qué ya no se puede cumplir la propiedad de que una arista se encuentra en a los más 2 triángulos. La arista $e_{4,5}$ es compartida por los triángulos $t_{1,5,4}$, $t_{2,5,4}$, $t_{4,5,8}$ y $t_{4,5,7}$.

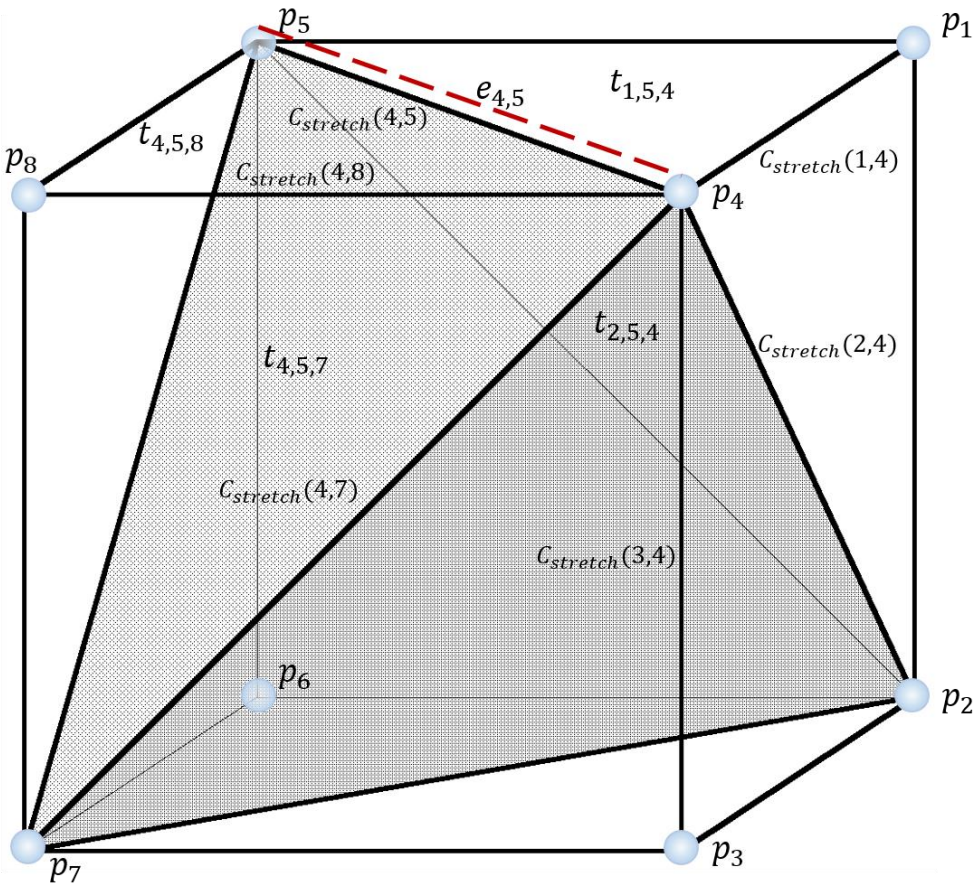


Figura 4.2. Descomposición de volumen en caras triangulares a partir de tetraedros.

Dado que las partículas en un volumen están ligadas no solo en un plano es que la restricción de flexión se vuelve innecesaria, pues la flexión es restringida por las restricciones de elasticidad $C_{stretch}$ que forman un volumen cerrado. En la Figura 4.2 se observa que la partícula p_4 tiene restricciones de movimiento en 3 ejes debido a las restricciones de elasticidad, por lo que si se agregaran restricciones de flexión C_{bend} serian redundantes para los propósitos de la simulación del volúmen.

Debemos recordar que la estructura *cloth* determina la masa de las partículas empleando una densidad de superficie $[\frac{kg}{m^2}]$ en conjunto con la cantidad de triángulos adyacentes, esto mismo se puede hacer al representar el volumen. Con esta implementación volumétrica, la densidad deja de ser superficial $[\frac{kg}{m^3}]$ y la masa de la partícula viene dada por un cuarto de la suma de las masas de todos los tetraedros que comparten esa partícula. Dependiendo de qué tan uniforme se encuentren distribuidas las partículas en el objeto se puede hacer una aproximación considerando que la masa de todas las partículas es igual entre sí.

En Flex se emplean triángulos como primitiva de representación de los cuerpos, por lo que la representación volumétrica también tiene que estar expresada mediante triángulos, esto significa que cada uno de los tetraedros se debe transformar a triángulos. Aunque el proceso es trivial requiere realizar el filtrado de triángulos duplicados, esto se debe a que una misma cara triangular se puede encontrar en a lo más 2 tetraedros. Si observamos la Figura 4.2 podemos observar que el triangulo $t_{4,5,7}$ es común a los tetraedros $tet_{4,5,7,8}$ y al tetraedro $tet_{2,4,5,7}$.

El proceso que se empleó para construir un objeto deformable usando la versión modificada de *cloth* en Flex se ilustra en la Figura 4.3.

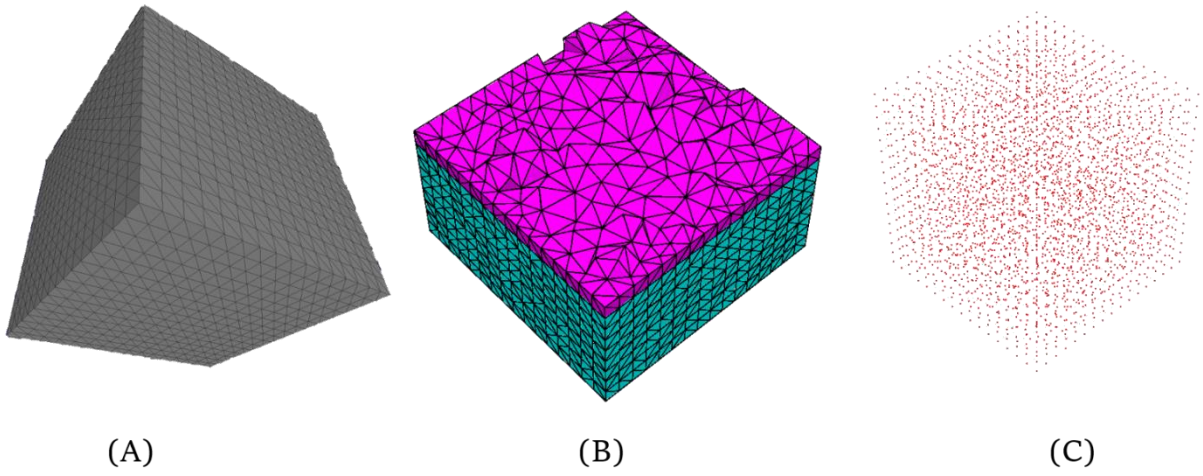


Figura 4.3. Proceso para construcción de modelo volumétrico de partículas a partir de una superficie cerrada. A) Superficie cerrada B) Volumen de tetraedros C) Nube de puntos

El primer paso es obtener una representación de la superficie que modela el objeto, dado que se trata de un volumen, la superficie debe ser cerrada y no tener problemas de auto intersección. A partir de esta superficie cerrada se genera una malla triangular que servirá de entrada para el programa de computadora Tetgen. Tetgen[18] es un programa que nos permite generar mallas de tetraedros y triangulaciones de Delaunay en 3D. Además, permite crear una malla de tetraedros a partir de la superficie cerrada triangular que define al volumen. Es posible variar la calidad de la malla así como ingresar diferentes parámetros a fin de ajustar el tamaño, uniformidad y creación de tetraedros.

Una vez que se ha creado la malla de tetraedros se puede generar un modelo de partículas. A fin de simplificar este proceso y poder aprovechar al máximo la malla volumétrica se decidió crear una nube de puntos con cada uno de los vértices de la malla; la posición de cada una de las partículas que conforman el objeto volumétrico se obtiene a partir de esta nube de puntos. El traducir los vértices de la malla a partículas conlleva un error de discretización en la simulación pues los vértices son el centro de cada partícula y entre mayor sea el radio que se asigna a las partículas mayor es el error que se tendrá en la representación del objeto.

Dado que los vértices de cada tetraedro son trasformados en partículas, es que las partículas que se encuentran en la superficie del volumen no se encuentran totalmente contenidas dentro del volumen original. Una alternativa para solucionar esto sería generar partículas por cada tetraedro, asignando la posición de la partícula con base en el centro geométrico del tetraedro y definiendo las restricciones con base en los vecinos de dicho tetraedro. La idea anterior tiene la limitante de que se requerirían tetraedros de volumen uniforme y muy regulares, por lo que un modelo que requiriera una resolución diferente en diferentes regiones no se podría implementar, es por eso que se decidió basarse en la malla a pesar de que las partículas de la superficie del volumen no están completamente contenidas dentro del volumen del objeto.

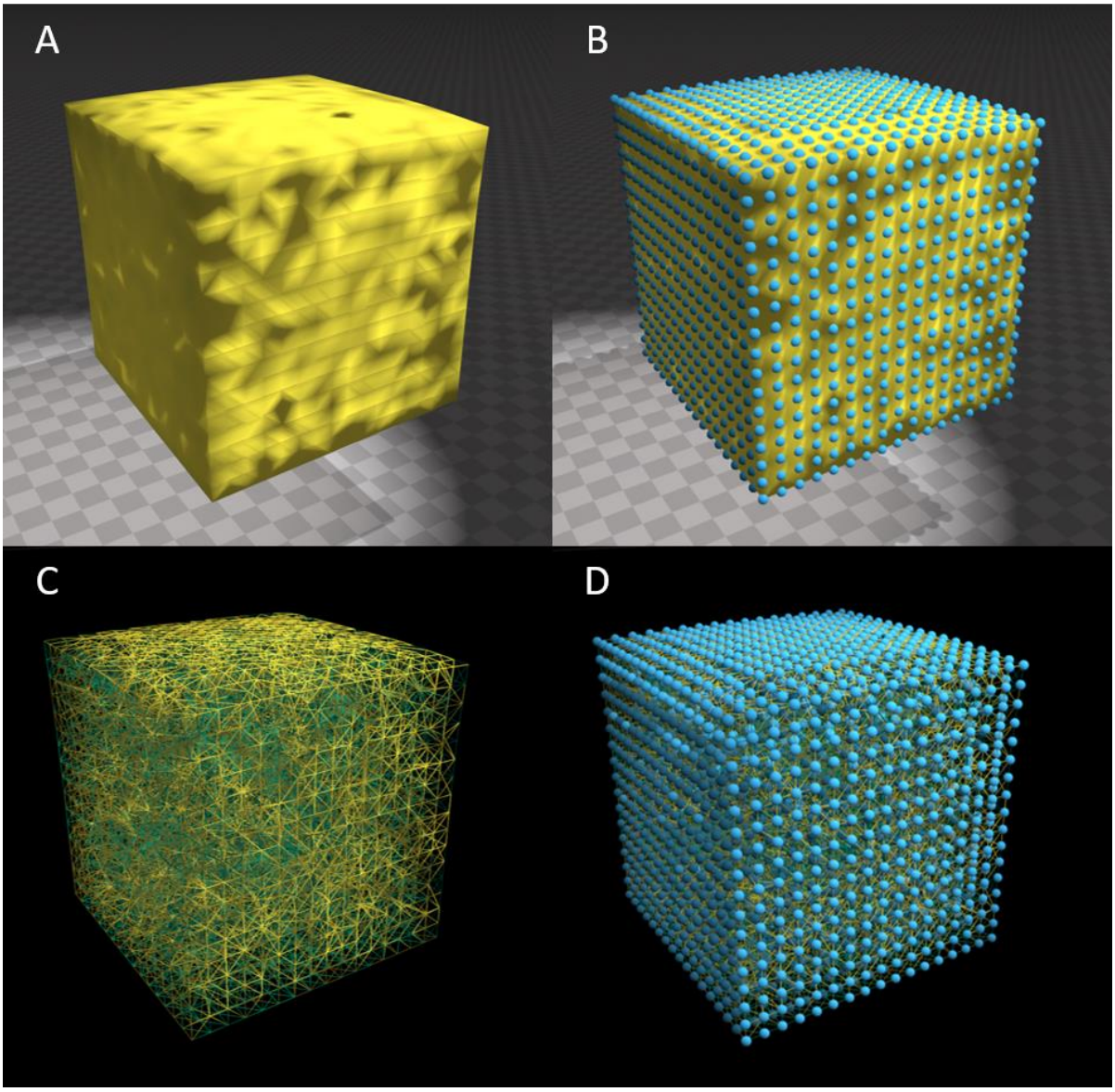


Figura 4.4. Modelo de partículas de un volumen usando versión modificada de cloth en Flex.
A) Modelo deformable generado con tetraedros. B) Modelo de partículas a partir de los vértices de la malla.
C) Restricciones generadas a partir de aristas de la malla. D) Partículas relacionadas por medio de restricciones.

Dado que el modelo de partículas requiere asignar restricciones es que por cada arista que se tiene en la malla se genera una restricción de estiramiento $C_{stretch}$. Estas restricciones conforman la red de resortes que unen a las partículas. Se les asigna una longitud de reposo y valor de rigidez tal como se hace con el *cloth*.

En la Figura 4.4 se muestra el modelo de partículas generado para Flex, dicho modelo está construido a partir de la malla de tetraedros. Para la construcción se emplea la versión modificada de la estructura de *cloth* definida en Flex. En esta versión modificada es posible realizar cortes al objeto deformable como se muestra en la Figura 4.5.

Es importante mencionar que dependiendo de donde se aplique el corte se tienen diferentes resultados, esto es debido a que los vecinos de cada partícula dependen de las aristas de la malla de tetraedros. Por suerte es posible mejorar la calidad del corte incrementando la cantidad de partículas en el modelo y dado que la implementación no requiere procesamiento adicional a fin de continuar con la simulación después de realizar algún corte, se pueden realizar tantos cortes como se requieran. Solo se debe considerar que existe un límite definido en Flex para el número de partículas permitidas en la simulación, por lo que se debe tomar en cuenta que cada corte agrega algunas partículas al sistema a fin de representar el material que se retira del objeto. Sin embargo, empíricamente probamos este mecanismo de corte y no encontramos ningún problema de desborde de partículas durante nuestras pruebas, por lo que deducimos que el método propuesto es adecuado para nuestros fines.

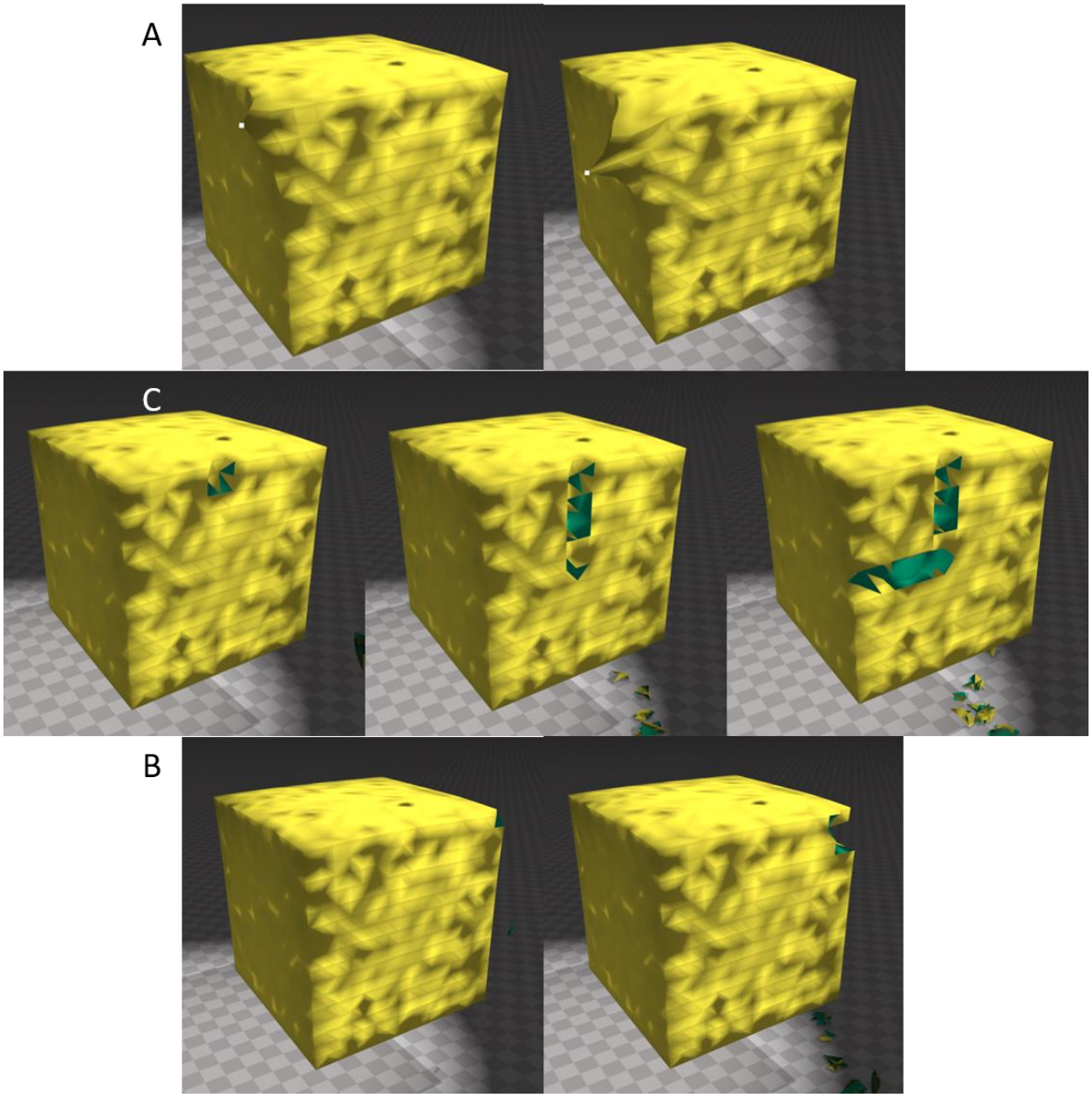


Figura 4.5. Implementación de volumen deformable con Cloth en Flex.
A) Deformación elástica. B) Corte transversal. C) Corte en borde.

En este trabajo se ha hecho mención de dos métodos de simulación para cuerpos rígidos y cuerpos deformables, método de elementos finitos el cual es un método basado en mallas y dinámica basada en posición que es un método de simulación de partículas. De entre estos métodos la simulación por partículas es el método que se decidió usar dado que la aplicación que se busca dar a estos métodos de simulación es su implementación en simuladores quirúrgicos.

Anteriormente se mencionó que uno de los problemas más importantes de los cuerpos deformables es la simulación de eventos que modifiquen el dominio (vértices, aristas, triángulos) que define al objeto, como ejemplo de este evento se tiene la realización de cortes que experimenta un tejido u órgano durante la realización de un procedimiento quirúrgico. Con el método expuesto anteriormente se logra una implementación que permite realizar cortes usando un método de simulación que aprovecha las capacidades de procesamiento en paralelo de las GPU.

Capítulo 5. Experimentos y Resultados.

Uno de los puntos más importantes por lo que se realizó la implementación de las técnicas de simulación que permiten cortes en GPU es mejorar el desempeño de las simulaciones y poder realizar cortes en tejido más eficientemente.

Uno de los elementos más relevantes que debe cumplir una simulación en un ambiente virtual con interacción del usuario es la velocidad de respuesta de cómputo numérico. Para el despliegue gráfico, en términos de cuadros por segundo (fps o *frames per second*). En el caso del cine se maneja un estándar de 24fps, dicha velocidad de despliegue grafico es suficiente para crear la ilusión de movimiento continuo, sin embargo, en la actualidad ese estándar no es el idóneo para las aplicaciones con ambientes virtuales. En la industria de los videojuegos, por ejemplo se busca tener entre 30fps y 60fps. Para el caso del monitoreo de movimientos de interacción se requieren velocidades aún mayores, por ejemplo, para transmisiones deportivas es preferible 300fps debido a la gran cantidad de acción que se debe captar. Para los fines que perseguimos nos basaremos en el estándar de 30fps a 60fps, esto obedece a la base que sientan los videojuegos.

Otro punto a considerar es la cantidad de partículas que son simuladas al mismo tiempo, para ello consideraremos un objeto que concentra la totalidad de las partículas de la simulación y sobre el cual se interactúa a fin de medir el desempeño con base en la cantidad de partículas que lo componen. La cantidad de partículas es de vital importancia ya que conforme se incrementen la cantidad de partículas, mayor es la calidad visual que se puede alcanzar. Además de que, debido a las limitaciones del método en cuanto a cortes, el tener una mayor cantidad de partículas conlleva a tener cortes más precisos.

A fin de demostrar la capacidad de escalabilidad que proporcionan las GPU al emplear CUDA se decidió comparar el desempeño de la implementación de cortes usando diferentes GPU. En la Tabla 5.1 se muestran las características que nos interesan de los equipos empelados para los experimentos.

Tabla 5.1. Especificaciones de equipos empleados en experimentos.

ID de Equipo	***Modelo CPU***	***CPU Cores***	***Velocidad CPU***	***Modelo GPU (NVIDIA)***	***CUDA Cores***	***Velocidad GPU***
A	Intel Core i7-4710HQ	4	2.5 GHz	GeForce GTX 860M	640	797 MHz
B	Intel Core i7-4700HQ	4	2.4 GHz	GeForce GTX 770M	960	811 MHz
C	Intel Core i7-3820	4	3.6 GHz	GeForce GTX Titan	2688	837 MHz
D	AMD Opteron 1222	2	3.0 GHz	GeForce GTX Titan Black	2880	889 MHz

Tabla 5.2. Modelos empleados en experimentos.

Id de Modelo	***Número de partículas***	***Número de restricciones (Resortes)***	***Número de triángulos***	***Radio de partícula [unidad]***
1	27	196	120	1.0
2	128	1324	974	0.7
3	686	8168	6414	0.4
4	3400	42166	33832	0.2
5	4513	59518	48958	0.2
6	7863	83786	61918	0.1
7	10417	123396	96420	0.1
8	22353	303528	252680	0.1

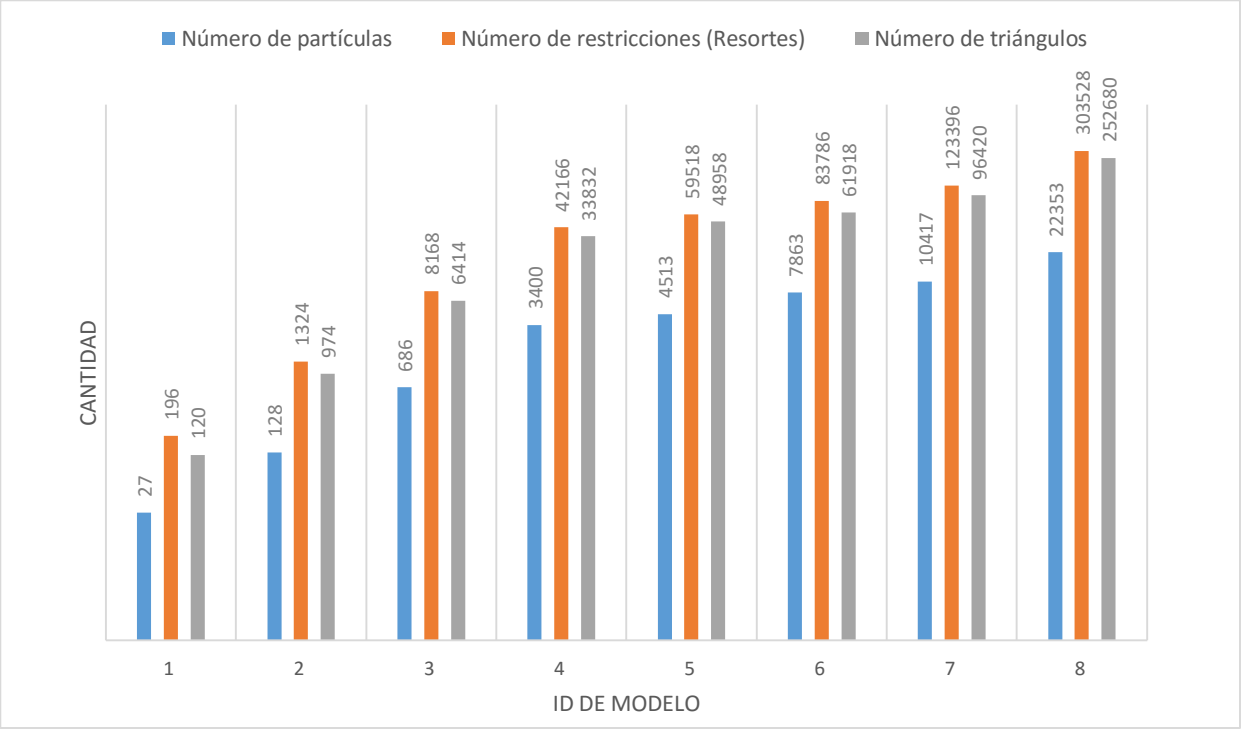


Figura 5.1. Comparación de elementos de los modelos empleados.

5.1. Desempeño visual (fps).

La primera prueba consistió en medir el desempeño gráfico de la aplicación empleando los modelos mostrados en la Tabla 5.2. Los resultados del desempeño gráfico se miden en fps, dado que tenemos como límite 60fps, por lo que la aplicación parte de una constante de tiempo de integración (Δ_t) que le permita alcanzar 60fps haciendo pausas si es que la generación del cuadro de despliegue toma menos del tiempo destinado. En el caso en el que la generación del cuadro para el Δ_t se mayor que este, la aplicación no ajusta el Δ_t por lo que se observa un movimiento más lento en la animación generada. En la Figura 5.1 se muestra la gráfica de comparación de elementos que conforman los diferentes modelos empelados en las pruebas.

La Figura 5.2 muestra los resultados obtenidos. Los fps fueron obtenidos empleando el software FRAPS[14] el cual es una herramienta de captura para Windows. La grafica muestra el desempeño del despliegue gráfico de la aplicación, en ella se observa como el incremento en partículas y restricciones tiene un impacto negativo en el desempeño, sin embargo entre mayor sea el número de partículas el modelo tiene un comportamiento más libre, permitiendo crear deformaciones más finas en el modelo.

Es importante mencionar que aunque el equipo D tiene una GPU con más *cores* que el equipo C, esta no presenta mejor desempeño debido principalmente al cuello de botella entre el *host* y *device* pues el equipo D tiene un canal de datos de menor capacidad que el equipo C.

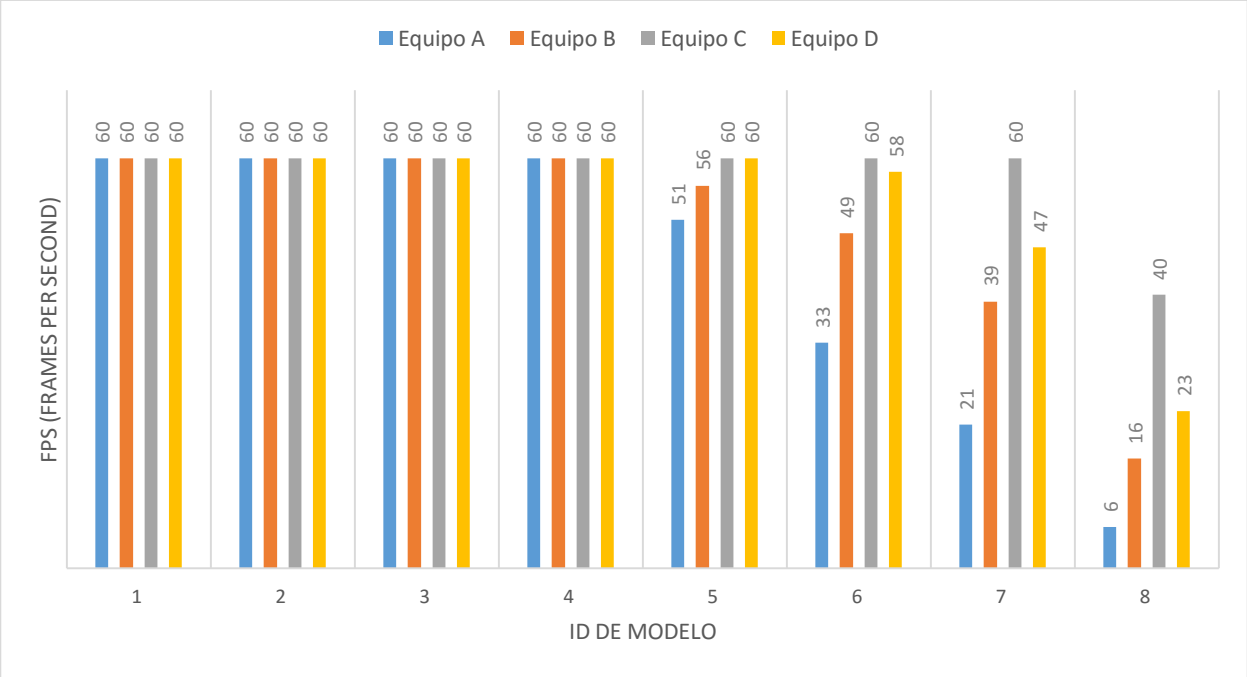


Figura 5.2. Desempeño gráfico de modelos en diferentes equipos.

En la Figura 5.3 se muestran los modelos a fin de poder apreciar sus diferencias visuales, así como sus capacidades de deformación.

[14] www.fraps.com

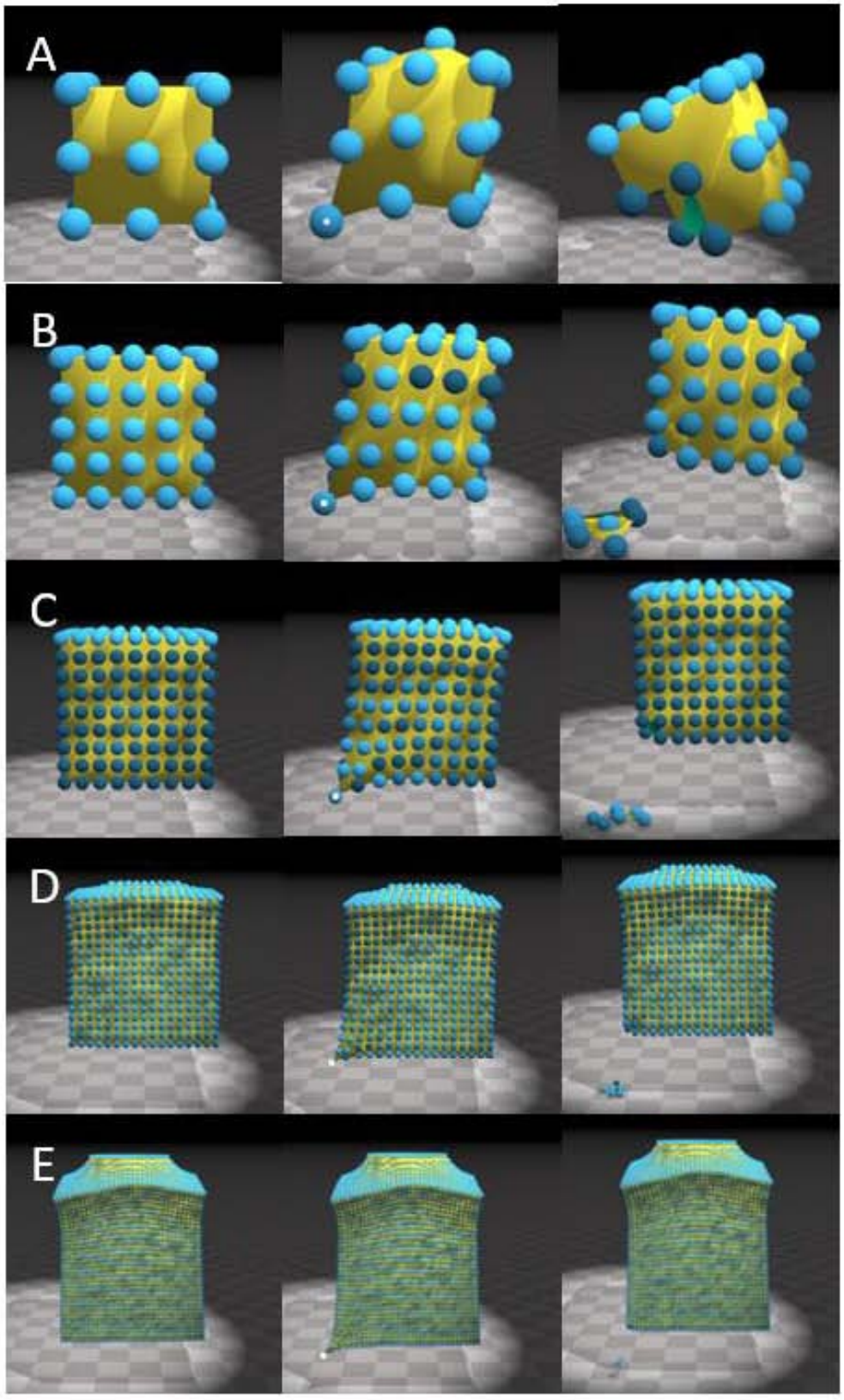


Figura 5.3. Deformación y corte de modelos usados en experimentos.
Modelos basados en Tabla 5.2.
A) Modelo 1. B) Modelo 2. C) Modelo 3. D) Modelo 5. E) Modelo 8.

5.2. Desempeño en GPU.

Para medir el desempeño en CUDA se utilizó la herramienta NSIGHT Performance Analysis[15] que se encuentra en Microsoft Visual Studio, la cual nos permite generar reportes de rendimiento. Empleando esta herramienta se hizo el trazado de la aplicación y se decidió medir el rendimiento de CUDA, los *kernels* de CUDA en los que se concentraba la mayor parte del procesamiento y el despliegue gráfico con OpenGL. Los valores medidos y reportados en este trabajo son:

- Desemepeño de CUDA (*CUDA Overview*). Se muestran el porcentaje y tiempo de uso de CPU y GPU de CUDA con respecto al tiempo total de captura. El valor de CPU corresponde al uso de CPU para la ejecución de las llamadas al API de CUDA mientras que el valor de GPU corresponde al uso de GPU para la ejecución de los *kernels* de CUDA.

- Desempeño de Kernles de CUDA (*Kernel Summary*). Se muestran el tiempo que consume la ejecución de los *kernels* de CUDA *SolveSpring* y *UpdateTriangles*, los cuales se decidieron tomar dado que son los que tienen mayor carga de trabajo o presentan mayor cambio a lo largo de las diferentes pruebas con los diferentes modelos.

- Desempeño de OpenGL (*OpenGL*). Se muestra el tiempo promedio que emplea la GPU en generar un cuadro de despliegue, así como el porcentaje de ese tiempo en el que se realiza la carga de trabajo.

Se debe tomar en cuenta que los tiempos se obtienen durante la ejecución de la captura empleando una sola GPU, es decir esta última ejecuta el proceso y se encarga de ejecutar el análisis, por lo que es de esperar que el desempeño durante una ejecución normal sea superior al de los reportes.

[15] https://developer.nvidia.com/performance-analysis-tools

Un punto que salta a la vista es el hecho de que los equipos de cómputo no tienen el mismo CPU, sin embargo son de capacidades similares y además la mayor parte del procesamiento se realiza en la GPU, por lo que podemos considerar ese factor como no determinante en el cambio en cuanto al desempeño de la simulación.

En la Tablas 5.3, 5.4, 5.5 se muestran los tiempos de ejecución para diferentes CPU y GPU medidos mediante la herramienta *Performance Analysis* para una ejecución de 30 cuadros de despliegue empleando los 8 modelos que se mencionan en la Tabla 5.2. Las tablas corresponden al desempeño medido en los equipos B, C y D respectivamente. Hay que mencionar que en algunos casos al generar la prueba se tuvieron problemas de memoria al generar los reportes por lo que alguno de los resultados no se reportan en las tablas, del mismo modo no se agregan los resultados de las pruebas en equipo A dado que al tratarse de un equipo de menor poder se experimentaron en mayor medida los problemas mencionados anteriormente al generar los reportes.

En la Figura 5.4 se muestra la comparativa del tiempo de procesamiento en CPU y GPU de los 3 equipos usados para realizar las pruebas con los diversos modelos. Como se mencionó anteriormente los valores corresponden al tiempo que se requiere para las llevar a cabo la ejecución de llamadas al API de CUDA y ejecución de *kernels* de CUDA de la aplicación. En la gráfica se observa claramente una tendencia en el incremento del tiempo requerido por la aplicación en cuanto a tiempo de ejecución por parte de la GPU, mientras que el tiempo en CPU mantiene un incremento menor con respecto al GPU, pero aun así empleándose más tiempo en realizar las llamadas al API de CUDA que en realizar la ejecución de los *kernels* de CUDA.

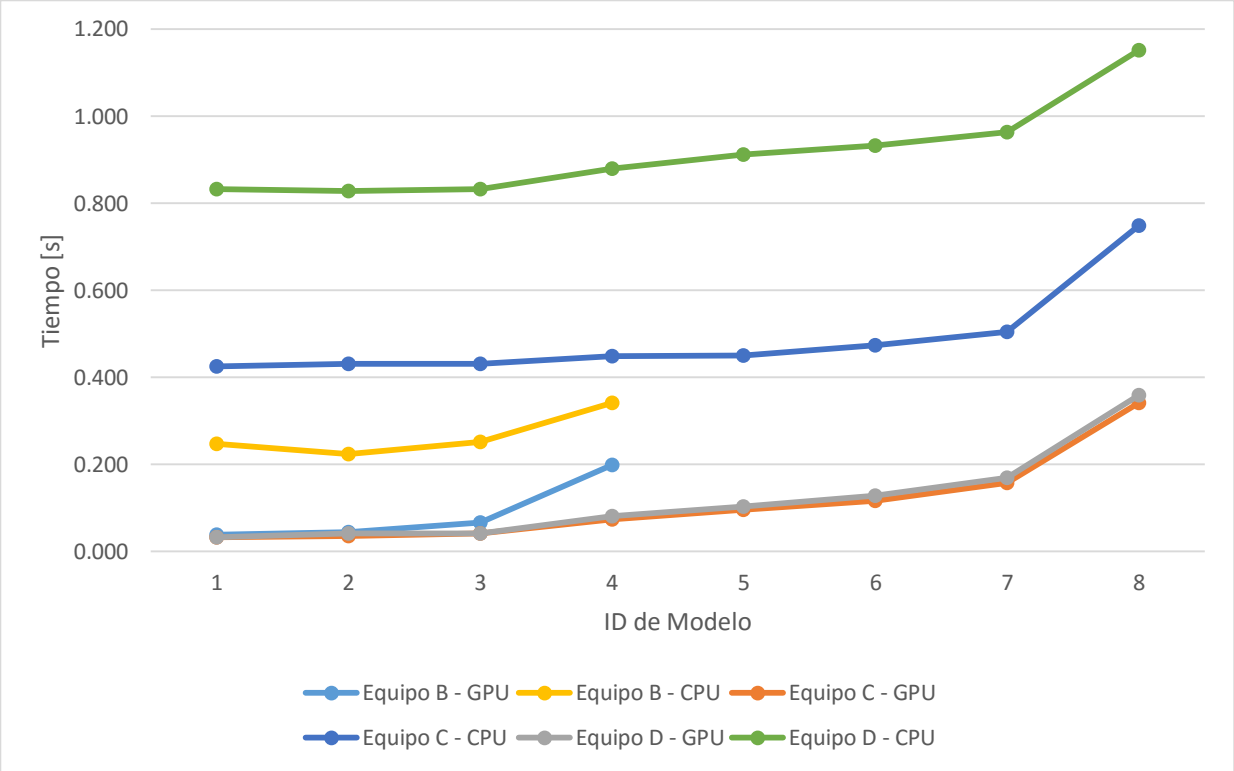


Figura 5.4. Análisis de desempeño de API de CUDA.

En la Figura 5.5 se muestran los tiempos promedio de ejecución del *kernel* de CUDA *SolveSprings* que es requerido por la aplicación que fue ejecutada en diferentes GPU, esto se compara con respecto a los diferentes modelos. En la gráfica se observa que el tiempo promedio varía conforme la cantidad de resortes se incremente, sin embargo las GPU con mayor cantidad de *cores* muestran un incremento menos marcado por lo que mantienen un tiempo promedio de ejecución más uniforme.

En la Figura 5.6 se muestra el tiempo total de ejecución del *kernel* de CUDA *SolveSprings* tal como en la Figura 5.5. En esta gráfica se observa que existe la misma tendencia como es de esperar, sin embargo al tomar en cuenta ambas gráficas podemos observar como la existencia de más *cores* en las GPU mejora el rendimiento en la aplicación. Otro punto al que se debe prestar atención es que aunque el número de restricciones (resortes) no es proporcional al decremento en el desempeño de la GPU, se mantiene una variación lineal hasta cierto punto.

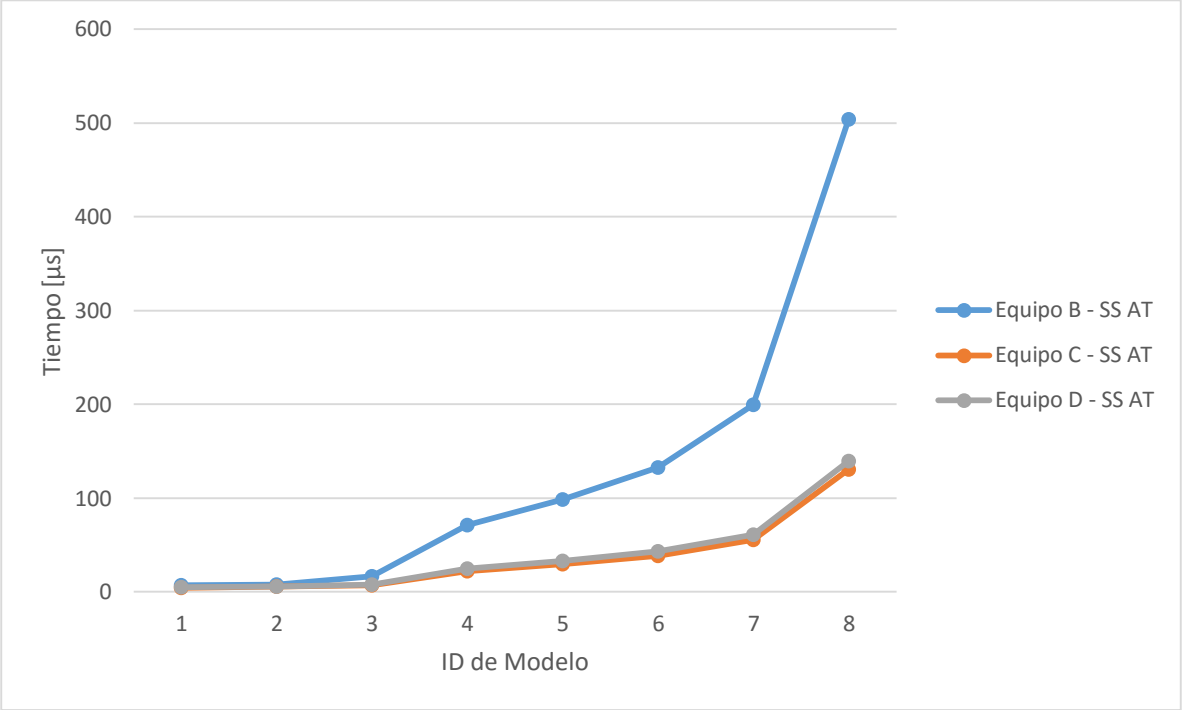


Figura 5.5. Resumen de kernel de CUDA SolveSprings (Tiempo Promedio).

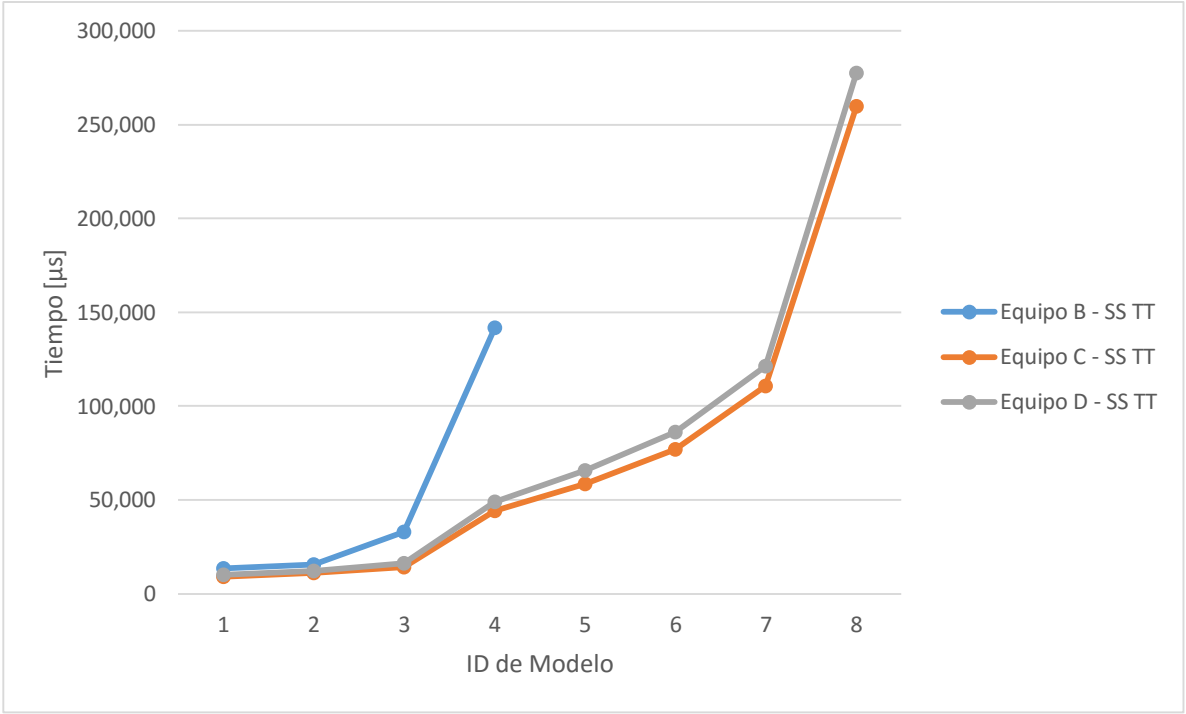


Figura 5.6. Resumen de kernel de CUDA SolveSprings (Tiempo Total).

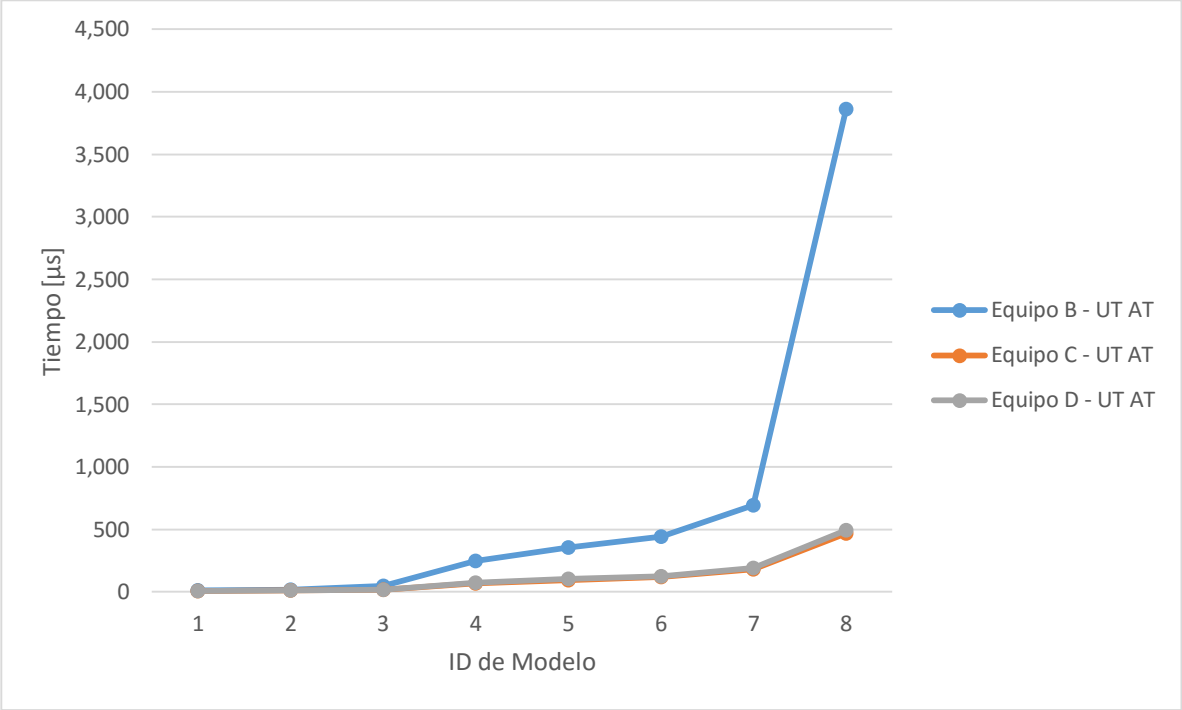


Figura 5.7. Resumen de kernel de CUDA UpdateTriangles (Tiempo Promedio).

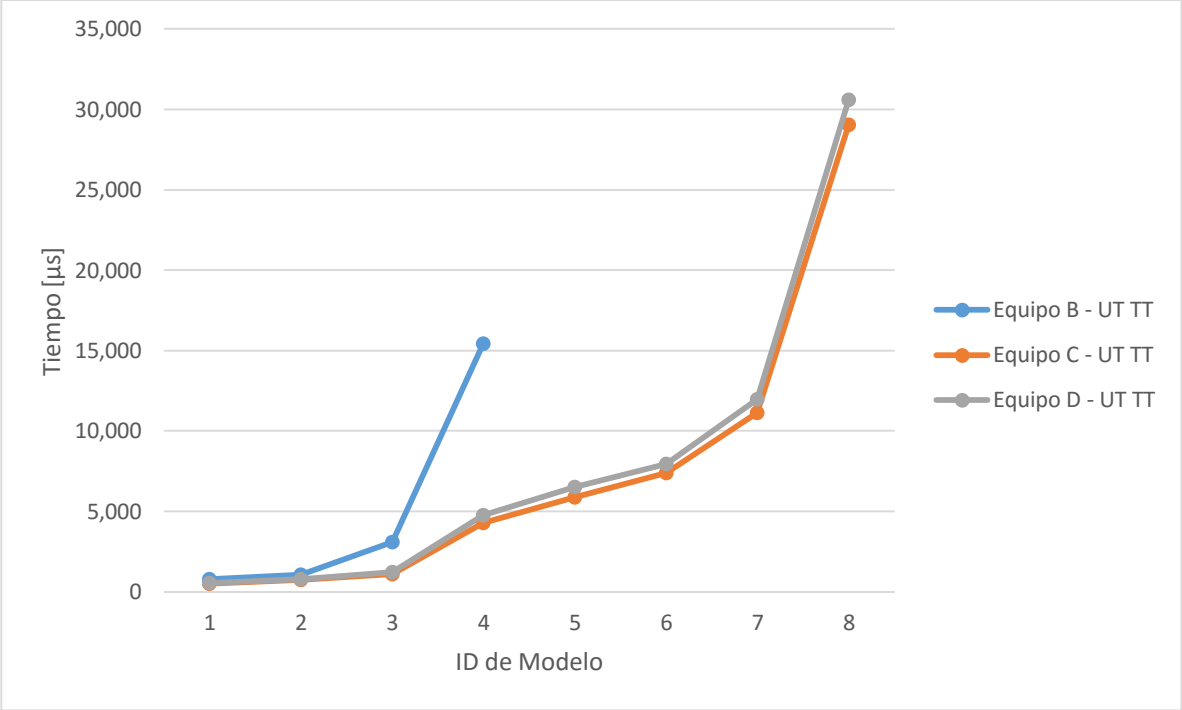
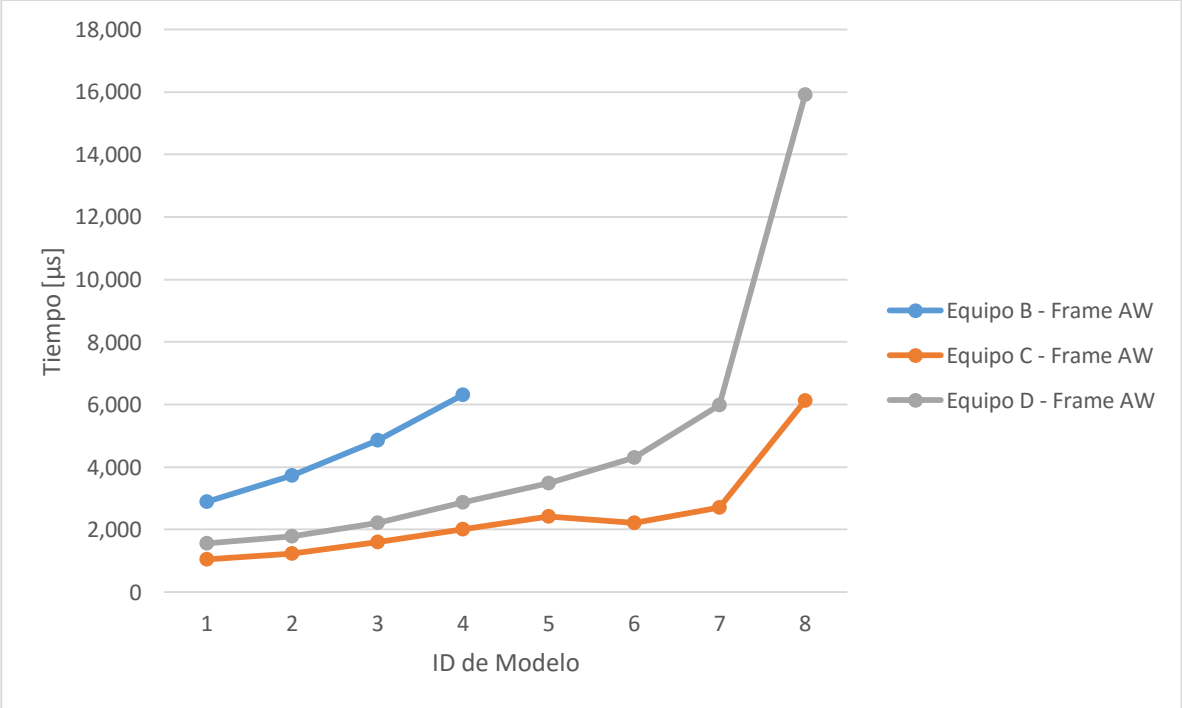


Figura 5.8. Resumen de kernel de CUDA UpdateTriangles (Tiempo Total).

En las gráficas de la Figura 5.7 y Figura 5.8 se muestra el mismo análisis pero para el *kernel* de CUDA *UpdateTriangles*. Este presenta un mayor incremento en el tiempo de ejecución conforme se incrementa la calidad del modelo, sin embargo no es un factor tan determinante dado que la cantidad de tiempo de procesamiento de GPU que consume es mucho menor que el *kernel* de CUDA *SolveSprings*. Es posible entender el comportamiento si se considera que los *kernels* ejecutan diferentes tareas, algunas más complejas que otras.

Por ultimo en la Figura 5.9 se muestra el tiempo promedio que toma generar los cuadros de despliegue de OpenGL usando la GPU. En esta parte es notorio que los tiempos son muy similares, pero sí existe una tendencia a incrementar el tiempo de trabajo, sin embargo en comparación al tiempo que se requiere para ejecutar los *kernels* este es muy inferior.



Gráfica 5.1. Carga de trabajo en GPU para generación de frame con OpenGL.

Tabla 5.3. Análisis de desempeño en equipo B.

Equipo B											
		CUDA Overview				Kernel Summary				OpenGL	
	Captura	API calls		Launches		SolveSprings		UpdateTriangles		Frames	
ID de Modelo	Time [s]	CPU [%]	Time [s]	GPU [%]	Time [s]	Avg. Time [µs]	Total Time [µs]	Avg. Time [µs]	Total Time [µs]	Avg. GPU Duration [µs]	Avg. Workload Duration [%]
1	1.50	16.5	0.248	2.6	0.039	6.877	13,643.60	12.215	757.33	33,964.67	8.52
2	1.53	16.8	0.223	2.9	0.044	7.801	15,477.51	17.340	1,075.05	32,964.64	11.28
3	1.70	14.8	0.252	3.9	0.066	16.573	32,880.82	49.735	3,083.58	35,365.72	13.71
4	2.31	14.8	0.342	8.6	0.199	17.466	141,787.93	248.513	15,407.82	41,650.31	15.13
5						98.665		356.955			
6						133.072		442.848			
7						199.442		691.102			
8						504.066		3,859.92			

Tabla 5.4. Análisis de desempeño en equipo C.

Equipo C											
		CUDA Overview				Kernel Summary				OpenGL	
	Captura	API calls		Launches		SolveSprings		UpdateTriangles		Frames	
ID de Modelo	Time [s]	CPU [%]	Time [s]	GPU [%]	Time [s]	Avg. Time [µs]	Total Time [µs]	Avg. Time [µs]	Total Time [µs]	Avg. GPU Duration [µs]	Avg. Workload Duration [%]
1	2.75	15.5	0.426	1.2	0.033	4.608	9,143.14	8.081	501.00	27,402.08	3.84
2	2.71	15.9	0.431	1.3	0.035	5.546	11,003.42	11.799	731.53	26,968.51	4.56
3	2.78	15.5	0.431	1.5	0.042	7.139	14,164.37	17.837	1,105.88	29,949.81	5.36
4	3.23	13.9	0.449	2.3	0.074	22.228	44,099.51	69.199	4,290.36		
5	3.44	13.1	0.451	2.8	0.096	29.541	58,608.46	94.763	5,875.32	32,379.55	7.44
6	3.65	13.0	0.475	3.2	0.117	38.807	76,992.16	118.930	7,373.65	33,015.32	6.72
7	4.04	12.5	0.505	3.9	0.158	55.789	110,685.34	179.217	11,111.48	33,411.35	8.13
8	5.99	12.5	0.749	5.7	0.341	130.894	259,692.91	468.333	29,036.64	40,186.71	15.24

Tabla 5.5. Análisis de desempeño en equipo D.

Equipo D											
		CUDA Overview				Kernel Summary				OpenGL	
	Captura	API calls		Launches		SolveSprings		UpdateTriangles		Frames	
ID de Modelo	Time [s]	CPU [%]	Time [s]	GPU [%]	Time [s]	Avg. Time [µs]	Total Time [µs]	Avg. Time [µs]	Total Time [µs]	Avg. GPU Duration [µs]	Avg. Workload Duration [%]
1	8.25	10.1	0.833	0.4	0.033	5.021	9,961.55	8.585	532.27	248,029.86	0.63
2	8.28	10.0	0.828	0.5	0.041	6.120	12,141.63	12.791	793.06	246,648.90	0.72
3	8.33	10.0	0.833	0.5	0.042	8.089	16,048.68	19.641	1,217.76	245,386.92	0.91
4	9.07	9.7	0.880	0.9	0.082	24.600	48,806.14	76.550	4,746.10	251,846.75	1.14
5	9.41	9.7	0.913	1.1	0.104	33.139	65,748.16	105.297	6,528.44	252,939.04	1.38
6	9.92	9.4	0.932	1.3	0.129	43.385	86,074.85	127.860	7,927.34	253,178.74	1.70
7	10.59	9.1	0.964	1.6	0.169	61.172	121,365.17	192.689	11,946.71	255,159.34	2.35
8	14.40	8.0	1.152	2.5	0.360	139.835	277,433.36	493.263	30,582.33	264,831.09	6.01

5.3. Demostración de cortes.

Por último se presentan ejemplos de cuerpos deformables a los que se aplican cortes, cada uno de los cuerpos tiene diferente cantidad de vértices por lo cual el resultado es diferente, pero el objetivo es apreciar como la finesa del corte depende de la cantidad de partículas, mientras mayor sea la cantidad de partículas se pueden generar cortes más finos.

Tabla 5.6. Modelos para demostración de cortes.

Modelo	***Número de partículas***
X (Cubo)	3400
Y (Cubo)	10417
Z (Conejo)	8651

En la Figura 5.9 se muestra una comparación de cortes realizados en los modelos mencionados en la Tabla 5.6. Los modelos X y Y tienen la misma forma pero contienen diferente número de partículas, por lo que al realizar cortes similares se aprecia un corte más fino en el cuerpo que tiene mayor número de partículas. Por otro lado el modelo Z corresponde a un modelo que tiene una forma completamente diferente, pero que tiene más partículas que el modelo X y menos partículas que el modelo Y, en ese ejemplo se observa como la distribución de partículas no es uniforme y se generan cortes diferentes lo largo del objeto, pero también se observa cómo es posible atravesar un objeto por completo al realizar cortes, es decir que se pueden realizar cortes que generan un cambio en la topología del objeto.

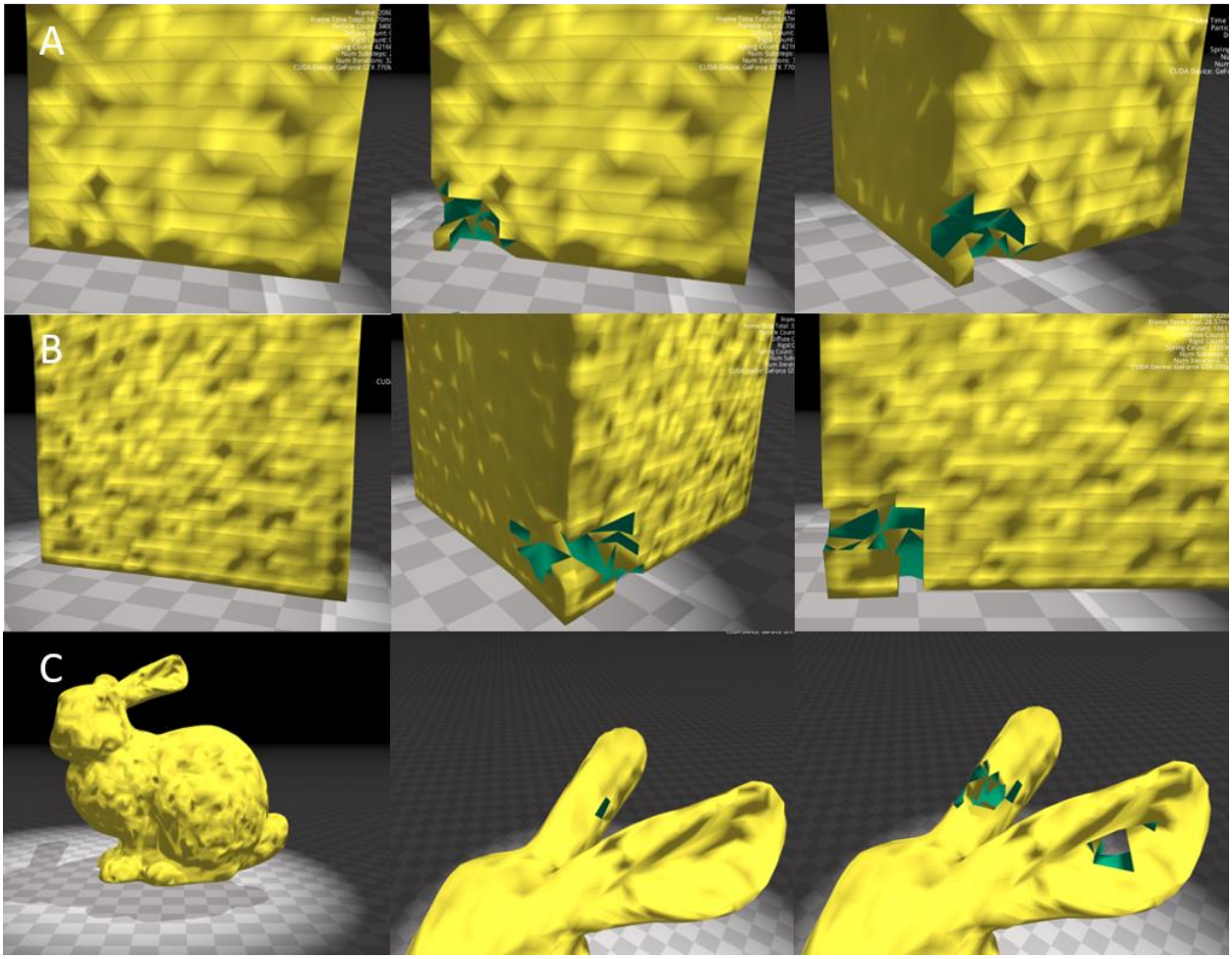


Figura 5.9. Demostración de cortes en cuerpos deformables.

A) Cubo con 3400 partículas. B) Cubo con 10417 partículas. C) Conejo con 8651 partículas.

Capítulo 6. Conclusiones y Trabajo Futuro.

En la actualidad la simulación por computadora es un área de la computación cada vez más aceptada y usada. En el caso de las simulaciones interactivas ha tomado gran importancia debido a que permite exponer a personas en ambientes controlados en los cuales experimentan eventos del mundo real, mediante la simulación de ambientes virtuales, que son usados para obtener información del evento o para llevar a cabo la adquisición de destrezas y habilidades de un individuo sin ser expuesto a los riesgos de una situación del mundo real.

Dependiendo del enfoque que se le desee dar a la simulación es que se han desarrollado diversos métodos, en un inicio la tendencia se marcaba hacia la representación fidedigna de la realidad, sin embargo conforme han tomado popularidad las simulaciones interactivas es que se ha prestado más atención a lograr representaciones en tiempo real (30 – 60 fps), con efectos visuales físicos, por lo que algunos de los métodos más recientes sacrifican realismo a fin de mejorar el desempeño para lograr una interacción que se sienta natural al usuario.

En este trabajo se utilizó el método basado en partículas conocido como dinámica de posición de base para la simulación de deformaciones y cortes de objetos virtuales, el cual tiene como principal motivación la implementación de simulaciones interactivas sin alcanzar una representación fidedigna de los fenómenos simulados, pero que mantiene el realismo necesario de ciertos eventos como lo es la elasticidad de cuerpos, rigidez, así como la interacción entre ellos mediante colisiones, así como cortes en los objetos de manera simple y eficiente.

Dado que la motivación para la implementación de estos métodos es la posibilidad de ser usados en simuladores quirúrgicos, el método estudiado se extendió a fin de poder simular volúmenes deformables con estructuras volumétricas como lo son los tetraedros. Adicional a eso se propuso un método mediante el cual es posible implementar eventos de corte en los objetos simulados. El poder simular cortes en volúmenes cubre gran parte de las necesidades que un simulador quirúrgico debe cubrir en cuanto a simulación.

Como trabajo a futuro se encuentra la implementación de este método para un simulador de RTUP, dada la naturaleza de dicha cirugía se requiere una simulación que permita aplicar cortes a un tejido suave, además de que al realizar el corte los elementos cortados del tejido suave se mantienen en el espacio de trabajo del cirujano. Es importante para la simulación el representar dicho evento, pues forma parte de las complicaciones en el proceso quirúrgico, pues obstruyen la visibilidad por lo que el cirujano debe realizar algunos movimientos a fin de recuperar la visibilidad y continuar con la cirugía.

Como se mencionó en el trabajo es importante que las simulaciones interactivas tengan un tiempo de respuesta lo suficientemente pequeño, lo cual permita hacer un despliegue grafico entre 30 y 60 Hz, con este método es posible alcanzar dichas velocidades al implementar la simulación de RTUP.

En este trabajo también se hace mención de uno de los recursos de cómputo masivo que se tienen disponibles en la actualidad, las GPU. Sin embargo aunque es posible adquirir dichos recursos de cómputo, la implementación de los métodos de simulación sigue siendo un proceso que debe ser realizado con cuidado, pues aunque existen herramientas como CUDA que simplifican el desarrollo, aún es necesario conocer la arquitectura y estar familiarizado con conceptos de computo en paralelo, como lo es el paralelismo de datos y paralelismo de tarea. Si no se manejan los conceptos anteriores no se puede asegurar que el acceder a estos nuevos recursos de cómputo masivo sea benéfico para las metas que se desean alcanzar al realizar la implementación de algún método de simulación.

A fin de mejorar el desempeño de la simulación es necesario que el ambiente gráfico que se implementa en las simulaciones interactivas pueda emplear de manera extensiva el uso de la GPU, lo anterior se logra mediante el uso de *vertex-buffer objects* en conjunto con *shaders* de graficación como lo son el *vertex shader* y *fragment shader*. Es por ello que el empleo de OpenGL 4.4 para generar el ambiente virtual es ampliamente recomendado, pues al combinarse con CUDA se pueden reducir cuellos de botella entre *host* y *device* mediante la interoperabilidad de CUDA y OpenGL.

Aunque en este trabajo no se hizo una comparación entre una implementación en CPU y una implementación en GPU para demostrar las capacidades del aceleramiento por GPU, hemos demostrado que la arquitectura CUDA permite conseguir una escalabilidad como la que se observaba cuando la velocidad de los procesadores se incrementaba da manera sostenida. Es por eso que aprovechar los recursos de cómputo disponibles en la GPU tiene el beneficio de poner a disposición del programador cada vez más poder de cómputo y si se realiza una implementación que aproveche las ventajas del paralelismo de datos se puede esperar una mejora en el desempeño.

En cuanto al método de simulación hemos mostrado las capacidades para simular cuerpos deformables a los que se les pueden aplicar cortes, los modelos usados en los experimentos alcanzan el desempeño deseado de 30 a 60fps con una resolución que consideramos suficiente para la representación de modelos anatómicos en un simulador quirúrgicos.

Se debe mencionar que la implementación de simulación de cortes no se pudo implementar totalmente en GPU, por lo que el método de simulación depende de datos calculados en CPU a fin de poder realizar la simulación del corte en GPU. Esto constituye un problema que será trabajado a futuro.

En cuanto a los problemas que observamos en la implementación de cómputo acelerado por GPU tenemos el gran cuello de botella que existe entre *host* y *device*, sobre todo en cuanto a copiado de memoria entre ambos dispositivos. Un escenario ideal es lograr una implementación de un método de simulación que se ejecute totalmente en GPU, pero si eso no es posible como en este caso, conviene reducir el paso por este cuello de botella mediante el uso de interoperabilidad. Dado que existe este cuello de botella se pudo observar que el solo depender del poder de computo de la GPU no es suficiente, es necesario tener un CPU que no frene el desempeño de la GPU. Esto se observó en los experimentos que se realizaron usando el equipo D, pues aunque cuenta con la mejor GPU del conjunto de prueba, ésta no mostro el mejor desempeño en debido al cuello de botella que existe en la comunicación de *host* y *device*.

Como observamos la implementación de métodos de simulación de GPU no es una solución simple, pero si se implementan aprovechando las ventajas de la arquitectura se obtienen mejoras en el desempeño que no son posibles empleando solo procesamiento en CPU.

Una de las ventajas de trabajar con el aceleramiento por GPU es que no se requiere de un equipo muy especializado ni tampoco muy costoso, por lo que observamos en los experimentos, una computadora con un procesador con capacidades similares o superiores a un Intel i7 de sexta generación (skylake), así como una GPU con características similares o superiores a una GeForce GTX Titan son los requisitos mínimos que se deben cubrir a fin de tener un equipo con el poder de computo suficiente. El costo de los componentes que se requieren son relativamente bajos y pueden ser adquiridos fácilmente. En el caso del simulador de RTUP que se planea implementar a futuro es necesario construir un equipo con características un poco superiores a las que se mostraron en estos equipos, los cual incrementa el costo pero sin llegar a duplicarlo.

Cabe mencionar que al construir los modelos no se modificó la rigidez de los resortes, por lo que aunque se modifica la masa y radio de las partículas para cada modelo, la rigidez se mantuvo constante, pero conforme se incrementan las restricciones dicha rigidez debe incrementarse a fin de mantener la misma rigidez inicial en todos los modelos. En este trabajo no se abordó ese aspecto dado que es un tema que requiere involucrar mucho más trabajo de investigación que permita la caracterización de cuerpos biomecánicos.

Referencias.

[1] Aleksander Byrski, Zuzana Oplatková, Marco Carvalho, and Marek Kisiel-Dorohinicki (Eds.), "Advances in Intelligent Modelling and Simulation: Simulation Tools and Applications", *Springer* 2012.

[2] F. Arámbula Cosío, M.A. Padilla Castañeda, P.R. Sevilla Martínez (2006), "Computer assisted prostate surgery training", *International Journal Humanoid Robotics*, vol.3, 4, pp.485-498.

[3] Wen-mei W. Hwu, "GPU computing gems", *Elsevier*, 2011.

[4] Mario A. Gutiérrez A., Frédéric Vexo, Daniel Thalmann, "Stepping into virtual reality", *Springer Verlag*, c2008.

[5] Shannon Robert, Johannes James, "Systems simulation: the art and science", *IEEE Transactions on Systems*, Man and Cybernetics, 6(10), pp. 723-724, 1976.

[6] Dhatt G, Touzot G, Lefrancois E, "Finite element method", *Numerical methods series*, 2012.

[7] Matthias Müller, Bruno Heidelberger, Marcus Hennix, John Ratcliff, "Position Based Dynamics", *3rd Workshop in Virtual Reality Interactions and Physical Simulation*, 2006.

[8] Miles Macklin, Matthias Müller, Nuttapong Chentanez, Tae-Yong Kim, "Unified Particle Physics for Real-Time Applications", *ACM Transactions on Graphics (SIGGRAPH 2014)*, 33(4), 2014.

[9] Miles Macklin, Matthias Müller, "Position Based Fluids", *ACM Transactions on Graphics (SIGGRAPH 2013)*, 32(4), 2013.

[10] Jihun Yu and Greg Turk, "Reconstructing surfaces of particle-based fluids using anisotropic kernels", *ACM Transactions on Graphics,* 32, 1, Article 5, February 2013.

[11] Tuan Nguyen Trung, Hoeryong Jung, Myeongjin Kim and Doo Yong Lee, "A Method for Generating Cut Surface in Surgery Simulation", *13th International Conference on Control, Automation and Systems (ICCAS 2013)*, 2013.

[12] M. Akay, A. Marsh et al, "Information Technologies in Medicine, Volume 1: Medical Simulation and Education", *John Wiley & Sons, Inc*., 2001.

[13] J. M. Rosen, M. K. Simpson, C. Lucey, "Virtual Reality and Surgery. Surgical Research", *Academic Press*, 2001.

[14] D. Bielser, V. A. Maiwald, and M. H. Gross, "Interactive cuts through 3-dimensional soft tissue," *scalpel*, vol. 1, p. 1, 1999.

[15] Denis Steinemann, Matthias Harders, Markus Gross, Gabor Szekely, "Hybrid Cutting of Deformable Solids", *Proceedings of the IEEE Virtual Reality Conference (VR'06)*, 2006.

[16] Chenyang Zhu, Yueshan Xiong, Kai Xu, Peng Shi. "Fast Cutting Simulations with Underlying Lattices", *6th International Conference on Biomedical Engineering and Informatics (BMEI 2013)*, 2013.

[17] J. J. Monaghan, "Smoothed Particle Hydrodynamics", *Annual Review of Astronomy and Astrophysics*, Vol. 30: 543-574. September 1992.

[18] Hang Si, "TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator", *ACM Transactions On Mathematical Software*, 41(2), Article 11, February 2015.

[19] A. Gastelum, J. Marquez, F. Trujillo, C. Duwig, B. Prado, P. Gamage, P. Delmas. "3D porus media liquid-solid interaction simulation using SPH modeling and Tomographic images", *MVA2009 IAPR Conference on Machine Vision Application*, May 20-22, 2009.

[20] Nicolas C. Buchs , François Pugin, Francesco Volonté, Philippe Morel, "Learning Tools and Simulation in Robotic Surgery: State of the Art", *World Journal of Surgery*. Volume 37, Issue12, pp 2812-2819, December 2013.

[21] T. Liu, A. W. Bargteil, J. F. O'Brien, L. Kavan, "Fast Simulation of Mass-Spring Systems", *ACM Transactions on Graphics*, Vol. 32, No. 6, Article 209, November 2013.

[22] S. Cotin, H. Delingette, "Real-time surgery simulation with haptic feedback using finite elements. Robotics and Automation", *Proceedings, 1998 IEEE International Conference*, Volume 4, May 1998.

[23] M. Bro-Nielsen, "Finite element modeling in surgery simulation", *Proceedings of IEEE*, Volume 86, Issue 3, March 1998.

[24] Y. C. Fung, "Biomechanics-Mechanical Properties of Living Tissues", *Berlin: Springer-Verlag*, 1993.

[25] E. Levi, "Elementos de Mecánica del Medio Continuo", *Ed. Limusa*, México, 1973.

[26] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, P. Hanrahan, "Brook for GPUs: stream computing on graphics hardware", *ACM Transactions on Graphics – Proceedings of ACM SIGGRAPH 2004*, Volume 23 Issue 3, August 2004.

[27] I. Buck, "GPU Computing: Programming a Massively Parallel Processor", *CGO 07 Proceedings of the International Symposium on Code Generation and Optimization*, March 2007.