Comparación de volúmenes: aplicación al análisis del comportamiento de modelos biomecánicos de órganos

Autor: MIGUEL ÁNGEL LAGO ÁNGEL

Director: Carlos Monserrat Aranda (DSIC)

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

UNIVERSIDAD POLITÉCNICA DE VALENCIA

DSIC

3 de febrero de 2011

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

La modelización biomecánica del tejido interno ha sido estudiada ampliamente debido a su gran utilidad en el guiado quirúrgico. Sin embargo, el difícil acceso a los órganos internos hace complicada la creación de un modelo biomecánico que represente de forma fiel su comportamiento. Es decir, un modelo es capaz de calcular el comportamiento físico ante las fuerzas ejercidas sobre el órgano.

El presente trabajo muestra una técnica que permite la validación de modelos biomecánicos de órganos internos, concretamente para el hígado. El modelo biomecánico se obtiene de imágenes tomadas mediante resonancia magnética (MR) o tomografía computerizada (TC). Una vez construido el modelo biomecánico, se necesita validar si su comportamiento se ajusta o no al del órgano real y es ahí donde se lleva a cabo la comparación de volúmenes entre la simulación y la realidad.

Se han diseñado dos experimentos que permiten validar el comportamiento del modelo biomecánico seleccionado. El primero de ellos trata de reproducir las condiciones de una operación laparoscópica donde el órgano se deforma por la presión ejercida por un gas [Martínez-Martínez et al., 2010]. El segundo experimento está basado en el trabajo de [Mazza et al., 2007] y consiste en ejercer una fuerza de tracción sobre la superficie del órgano.

Este trabajo se sitúa dentro del proyecto Naralap del Instituto Interuniversitario de Investigación en Bioingeniería y Tecnología Orientada al Ser Humano (LabHuman) de la Universidad Politécnica de Valencia. Naralap tiene dos partes diferenciadas: en primer lugar, una aplicación de realidad aumentada, cuyo objetivo es proyectar los órganos internos sobre un paciente sometido a una operación por laparoscopia [López-Mir et al., 2011] y en segundo lugar la creación de un modelo biomecánico del hígado, cuyo objetivo será conseguir simular el comportamiento del órgano al someterse a la manipulación por parte del personal sanitario. De esta manera se puede orientar y guiar mejor al cirujano permitiéndole ver una imagen de los órganos internos del paciente.

El proceso global de la validación del modelo biomecánico de los órganos se pue-

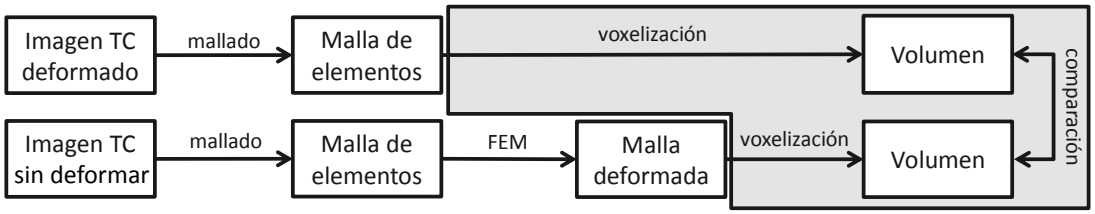

Figura 1.1: *Proceso global del modelado biomecánico. La parte que se realiza en este proyecto está marcada en gris y consiste en la transformación a volumen (voxelización) y la comparación de dos volúmenes.*

de ver en la figura 1.1. Los experimentos parten de un escaneado con tomografía computerizada (TC) del mismo órgano, sin deformar y sometido a una deformación controlada. La idea es averiguar si un modelo biomecánico concreto, y los parámetros biomecánicos que lo caracterizan, es capaz de comportarse de igual manera que el órgano real y obtener así un grado de similaridad entre la deformación real y la simulada.

Para simular la deformación que sufre un objeto, en este caso un órgano, la técnica más utilizada es el método de los elementos finitos (FEM: Finite Element Method) por su robustez y su capacidad de modelar complejas deformaciones y condiciones de contorno.

La necesidad de obtener estos parámetros en un tiempo razonable nos lleva a acelerar estos cálculos. Debido al gran procesamiento de datos que se manejan en los volúmenes y mallas de elementos finitos, los algoritmos implementados en este trabajo se han realizado mediante el cómputo paralelo en unidades de procesado gráfico (GPU), utilizando las herramientas que proporciona NVIDIA con el lenguaje CUDA.

Las tareas que comprenden el proyecto son las siguientes:

1. Generación de una malla regular (voxelización) a partir de un modelo de elementos finitos basado en tetraedros.
2. Comparación de dos volúmenes en forma de voxels, el del órgano deformado real y el de la simulación de la misma deformación.
3. Obtener un grado de similitud o ajuste entre ambos volúmenes comparados, tanto de forma analítica como visual.

El presente documento se organiza de la siguiente manera, en el capítulo 2 se detallan las herramientas y conceptos que van a utilizarse para la realización del proyecto. En el capítulo 3 se analiza el problema de transformar el modelo FEM en un volumen 3D. En el capítulo 4 se describen distintas técnicas de comparación de volúmenes. En el capítulo 5 se comenta la aplicación a los casos reales. Por último en el capítulo 6 se enumeran las conclusiones obtenidas.

Capítulo 2

Contexto científico

En primer lugar, debemos estudiar las áreas en las que se enmarca este trabajo y los conceptos necesarios para el desarrollo del mismo. En este capítulo se hace un estudio de lo que es un modelo biomecánico y los más adecuados para el hígado en concreto. Posteriormente se detallan las herramientas de aceleración en GPU existentes.

En el trabajo se han seleccionado el modelo elástico para simular el comportamiento del hígado frente a pequeñas deformaciones y la programación en shaders y CUDA sobre GPU para acelerar los algoritmos implementados en los siguientes capítulos.

2.1. Modelos biomecánicos

El modelo biomecánico de un órgano define la respuesta física del mismo ante cualquier carga aplicada bajo unas condiciones de contorno. En su definición entran en juego las propiedades de los tejidos blandos que forman el órgano. Determinar dichas propiedades es una tarea complicada al ser órganos internos y de difícil acceso.

Algunos autores han realizado medidas sobre órganos vivos animales y humanos como en [Brouwer et al., 2001], [Carter et al., 2001] y [Nava et al., 2008]. El trabajo de [Shi et al., 2008] realiza experimentos con hígados de cordero y ha sido este animal el elegido por su facilidad de adquisición y manipulación.

2.1.1. Método de elementos finitos

Es muy habitual la utilización del método de elementos finitos (FEM: Finite Element Method) para la resolución del comportamiento físico de un objeto deformable. El FEM obtiene una solución aproximada de las ecuaciones que caracterizan el com-

portamiento físico del objeto.

En primer lugar se debe realizar una discretización del volumen del objeto dividiéndolo en un conjunto de volúmenes más pequeños llamados elementos. Cada elemento está formado por una serie de nodos y el conjunto de los elementos forma la malla. De esta manera, los cálculos se realizan sobre cada uno de los nodos de la malla y, mediante funciones de interpolación, se obtienen los resultados para cualquier punto del objeto. En nuestro caso la malla está compuesta por tetraedros, el elemento 3D más sencillo. El comportamiento y la deformación sufrida se simulará mediante FEM sobre esta malla de tetraedros.

2.1.2. Tipos de modelos

El comportamiento de los tejidos orgánicos puede ser aproximado por tres tipos de comportamiento: elástico, viscoelástico e hiperelástico [Fung, 1993]. En este proyecto nos hemos centrado en el modelo elástico puesto que se puede asumir que el comportamiento ante pequeñas deformaciones del hígado es de este tipo [Shi et al., 2008].

Modelo elástico

Este tipo de modelo tiene dos parámetros principales que controlan el tipo de deformación sufrida. El primero es el índice de Poisson (ν) y hace referencia a la relación entre la deformación longitudinal y la deformación lateral. El segundo es el módulo de Young (E) y representa la relación entre la deformación sufrida y la tensión provocada.

Esta deformación viene definida por la expresión general de la densidad de energía de deformación de la ecuación 2.1, donde σ es el tensor de tensiones y ε el tensor de deformaciones.

$$w = \frac{1}{2}\sigma\varepsilon \tag{2.1}$$

Y teniendo que $\sigma = D\varepsilon$ es lineal ya que D es constante. La matriz $D = D(E, \nu)$ define las propiedades del material con respecto a ν y E [Zienkiewicz, 1994].

Modelo hiperelástico

En algunos materiales, el modelo elástico no tiene la precisión deseable. Un material hiperelástico es un material elástico en el que la relación tensión-deformación deriva de una función de densidad de deformación. El caucho es un buen ejemplo de este comportamiento cuya relación tensión-deformación es elástica no-lineal, isotrópica e incompresible. Los modelos hiperelásticos son capaces de ajustar este tipo de comportamiento.

El modelo hiperelástico más conocido es el de Mooney-Rivlin, cuya expresión general de la densidad de energía se ve en la ecuación 2.2, donde I_1, I_2, I_3 son los invariantes del tensor de Cauchy [Belytschko et al., 1998].

$$w = \sum_{i+j-1}^{N} c_{i,j}(I_1 - 3)^i (I_2 - 3)^j \tag{2.2}$$

Modelo viscoelástico

Se considera un comportamiento viscoelástico cuando un material sufre una fuerza constante y su respuesta varía con el tiempo. La respuesta de este tipo de material se debe a la fluencia lenta, que consiste en el aumento de la deformación progresivamente al estar sometido a una carga constante, y la relajación de tensiones, que es el descenso de tensión necesaria para mantener una deformación constante.

La fórmula 2.3 define la dependencia a través de la función de relajación del módulo cortante $G(t)$, donde g_1, g_2, τ_1 y τ_2 caracterizan el comportamiento de la relajación [Fung, 1993].

$$G(t) = 1 - (g_1(1 - e^{-\frac{t}{\tau_1}}) + g_2(1 - e^{-\frac{t}{\tau_2}})) \tag{2.3}$$

2.2. Aceleración en GPU

Hoy en día la velocidad de procesamiento de las unidades gráficas de los computadores está creciendo muy rápidamente, si bien están diseñadas para la generación de entornos gráficos 3D en tiempo real, últimamente están ampliando su campo de acción dando lugar a la GPGPU (General Purpose computing on Graphics Processing Units). Esto dota a las GPU de capacidad de programar cualquier algoritmo sobre ellas, siendo un dispositivo preparado para ello.

El renderizado de imágenes y entornos 3D en tiempo real tiene un coste computacional muy elevado y debe procesarse cada uno de los píxeles de la pantalla para conocer su color, profundidad, textura, iluminación, sombras y otros efectos en una tasa de fotogramas mayor a 25 por segundo. La GPU (Graphic Processing Unit) debe ser capaz de procesar esta cantidad de datos y para ello dispone de un diseño masivamente paralelo.

Una GPU contiene una alta cantidad de núcleos de procesamiento (del orden de cientos) lo que la dota de gran capacidad para procesar datos en paralelo. Comparando el rendimiento de algoritmos paralelos entre CPU y GPU, los implementados en GPU se procesan mucho más rápidamente. Sin embargo, en cuanto tenemos procesamientos secuenciales, con muchos saltos condicionales y cambios de contexto, la CPU se comporta mucho mejor.

La serie de procesos que realiza la GPU es conocida como tubería gráfica (*graphics pipeline*) y fue definida por Silicon Graphics Inc. en 1992 con la primera versión de OpenGL. Esta tubería divide los procesos en etapas cuya salida es la entrada de la siguiente, un extracto de esta estructura se puede ver en la figura 2.1.

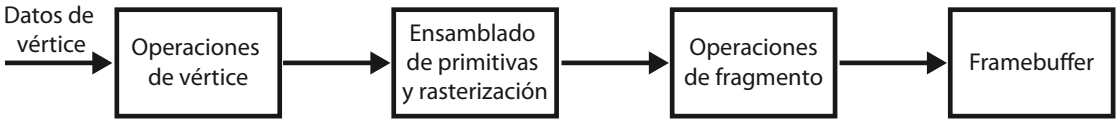

Figura 2.1: *Tubería gráfica de OpenGL simplificada.*

De la tubería gráfica podemos destacar las siguientes etapas:

Operaciones de geometría: se procesan todos los modelos geométricos de la escena.

Operaciones de vértice: de cada modelo se transforma la posición de sus vértices a la pantalla y se calcula su iluminación respecto a las luces de la escena. Este proceso se hace en paralelo para todos los vértices.

Operaciones de fragmento: se procesan los triángulos generando fragmentos (que luego serán píxeles) y se colorean interpolando los colores de los vértices.

Composición: Por último se juntan los fragmentos teniendo en cuenta su transparencia y profundidad, el resultado de esta etapa es una imagen en el `framebuffer`.

A continuación se hará un repaso por las técnicas más comunes de programación en GPU.

2.2.1. Shaders

La forma más básica de darle a la GPU trabajo es mediante los *shaders* o sombreadores. Los shaders se añadieron a la tubería gráfica para permitir generar efectos más realistas. Son pequeños programas que actúan sobre elementos de tres tipos: shaders de vértice, que se encargan de calcular el color, posición y coordenadas de textura de los vértices; shaders de fragmento, que indican el color y profundidad de cada fragmento y shaders de geometría, que añaden y eliminan vértices. La programación de shaders se puede realizar tanto en OpenGL como en Direct3D.

A medida que las GPU evolucionaron, OpenGL y Direct3D empezaron a permitir la programación de shaders propios. Los shaders se aplican de forma paralela a todos y cada uno de los elementos sobre los que actúan y las GPUs facilitan este tipo de computación. En OpenGL se programan en el lenguaje GLSL [GLSL, 2010] y en Direct3D en HLSL [HLSL, 2011].

Sin embargo, aunque los shaders permiten acelerar en gran medida los cálculos, sobretodo en los casos en que se hace el mismo procesamiento para cada dato, su funcionalidad está orientada a entornos gráficos, por lo que si queremos hacer cálculos sobre estructuras o problemas que no tienen como resultado una visualización, será necesario crear una representación falsa de la que podamos extraer los resultados.

Por otra parte, las distintas ejecuciones de un shader sobre los elementos, no son capaces de comunicarse entre sí ni de compartir memoria o sincronizarse en puntos de ejecución, lo cual puede ser un inconveniente para algunos problemas. También encontramos que el número de ejecuciones está limitado al tamaño del `framebuffer` y la resolución que éste tenga.

En definitiva, aunque los shaders son un gran avance en GPGPU, no se pueden considerar como tal ya que están pensados para procesamiento de gráficos y no tanto para utilizar la GPU para resolver otro tipo de problemas [Wong, 2008].

2.2.2. CUDA

Compute Unified Device Architecture (CUDA) es una arquitectura para computación paralela en GPU desarrollada por NVIDIA. CUDA funciona con todas las GPU de NVIDIA a partir de las GeForce 8. La gráficas CUDA permiten a los programadores lanzar instrucciones sobre el hardware gráfico. Debido a la arquitectura multinúcleo de las GPUs, las aplicaciones diseñadas para cómputo paralelo verán acelerada su ejecución en gran medida.

Kernels en GPU

La SDK de CUDA provee una serie de APIs, originalmente en C/C++, que permiten crear los llamados *kernels*. Cada uno de los kernels corresponde a un hilo del dispositivo.

CUDA organiza los hilos en una jerarquía, un kernel se lanza sobre un dispositivo en cada uno de los hilos y, a su vez, estos hilos se agrupan en bloques. El programador es capaz de especificar el tamaño de los bloques y el número de bloques. Se puede ver un diagrama de esta estructura en la figura 2.2. El grid hace referencia a todos los hilos de ejecución, un grid corresponde a una unidad gráfica.

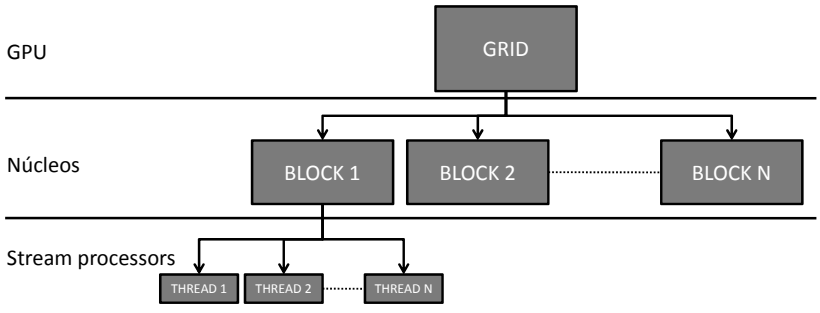

Figura 2.2: *Organización en bloques e hilos.*

Los hilos y los bloques tienen asociados un identificador que les permite saber sobre qué elemento deben actuar. Los hilos pueden saber qué posición ocupan dentro de un bloque, a qué bloque pertenecen, el número de bloques y el número de hilos por bloque.

Es importante tratar a los bloques de forma totalmente independiente, ya que son lanzados a ejecución sin garantizar ningún orden concreto, por lo que no se deberá depender del resultado de otros bloques.

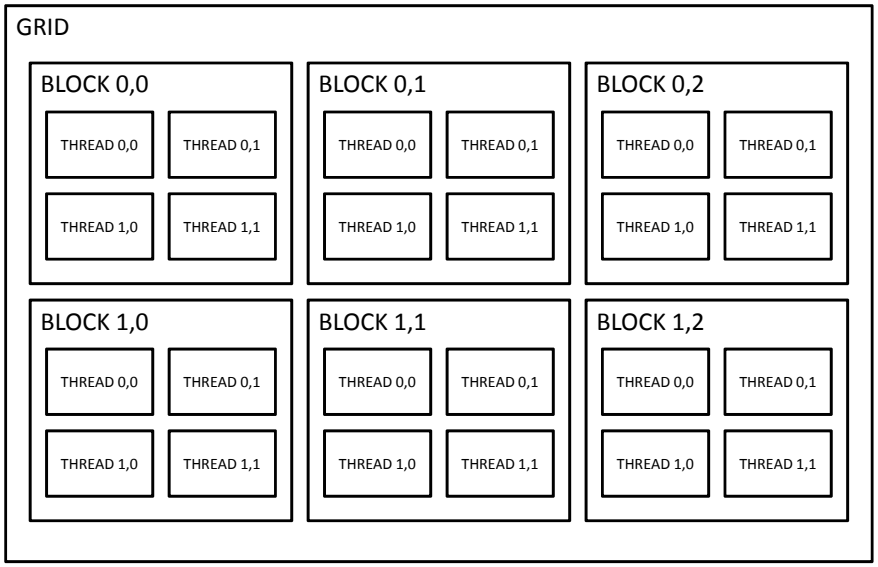

Figura 2.3: *Organización de hilos en 2D y bloques en 2D sobre un grid*

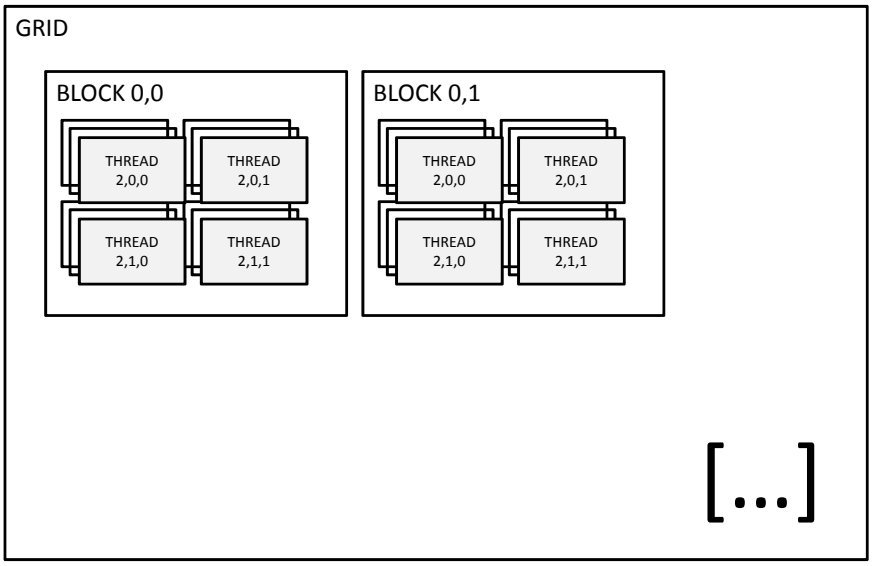

Figura 2.4: *Organización de hilos en 3D y bloques en 2D sobre un grid*

Administración de la memoria

La GPU tiene direcciones de memoria distintas a la CPU, por lo tanto los accesos a la memoria reservada deberán hacerse de manera explícita sobre la GPU. Afortunadamente, el lenguaje nos ofrece funciones muy similares a las utilizadas en C para este propósito.

Por ejemplo, en el código del cuadro 2.1 se puede ver cómo se reserva un vector de 1024 posiciones de tipo entero en la GPU que posteriormente se libera.

Código fuente 2.1: *Código para reservar un vector de enteros en GPU*

```
1 int n = 1024;
2 int nbytes = 1024*sizeof(int);
3 int * d_a = 0;
4 cudaMalloc( (void**)&d_a,  nbytes );
5 cudaMemset( d_a, 0, nbytes);
6 cudaFree(d_a);
```

CUDA mantiene una jerarquía de memoria de la siguiente manera:

Memoria por hilo: Cada hilo tiene su propio espacio de memoria independiente.

Memoria por bloque: Los hilos de un bloque pueden acceder a un espacio de memoria del bloque común a todos ellos.

Memoria global: Existe también una memoria global a la que cualquier hilo puede acceder.

En la figura 2.5 se puede ver un diagrama que ilustra la jerarquía de memoria presente en la GPU. Los hilos del mismo bloque pueden compartir información e incluso sincronizar su ejecución.

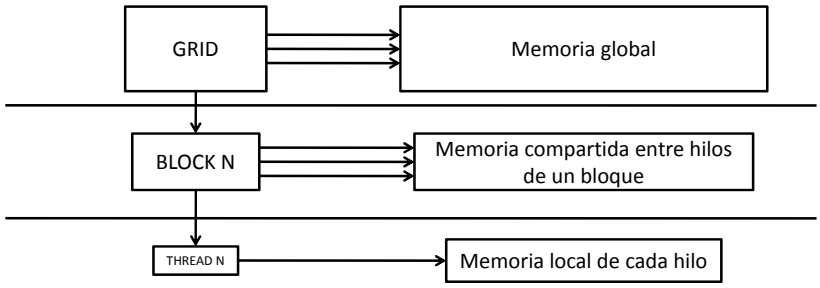

Figura 2.5: *Estructura jerárquica de la memoria en GPU*

Optimización de la memoria

Debido a la gran paralelización que sufren los kernels ejecutados, es importante optimizar los accesos a memoria que realizan los hilos, puesto que, si queremos conseguir una alta eficiencia y rapidez, deberemos evitar los accesos concurrentes que puedan producir fallos de memoria y caché.

Es importante saber que las GPUs dan prioridad a las unidades de procesamiento frente a los bancos y registros de memoria, por lo que deberemos reducir en la medida de lo posible la interacción de la memoria en GPU.

La memoria compartida es cientos de veces más rápida que la memoria global, así que la idea es que los hilos se comuniquen entre sí a través de esta memoria. De esta manera, los hilos pueden almacenar resultados compartidos de forma ordenada en la memoria compartida para que otros hilos (o ellos mismos) puedan acceder posteriormente evitando así los fallos de memoria.

Para conocer mejor cómo funciona la memoria de las tarjetas gráficas podemos observar el diagrama de la figura 2.6. Los hilos se ejecutan en un procesador escalar, el bloque se ejecuta en un multiprocesador. Cada multiprocesador contiene 8 procesadores escalares, dos SFU (Special Function Unit) y una unidad de doble precisión, además de un banco de memoria compartida.

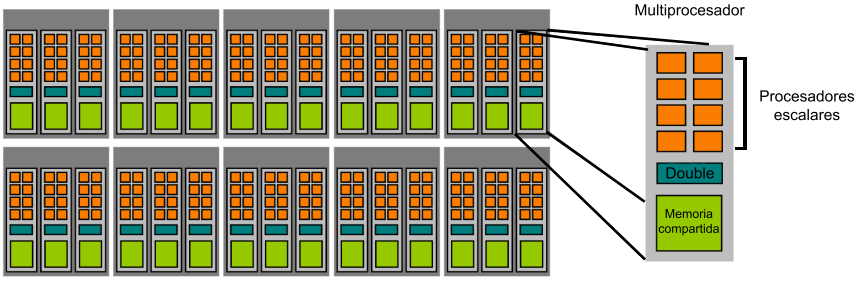

Figura 2.6: *Arquitectura de memoria de las series 10 de NVIDIA*

Cada bloque se organiza en *warps* de 32 hilos. Cada uno de los warps ejecuta una instrucción en paralelo en el multiprocesador: Single Instruction, Multiple Data (SIMD). La mitad de un warp (half-warp) de 16 hilos, puede coordinar los accesos a memoria en una única transacción (lectura y escritura).

El lenguaje CUDA

El lenguaje con el que se programan los kernels CUDA es un subconjunto de C con las siguientes características [NVIDIA, 2009]:

- Acceso a memoria sólo de GPU
- Las funciones no pueden tener un número variable de argumentos
- No hay variables estáticas
- No existe la recursión

Además, se pueden declarar los kernels con los siguientes cualificadores:

__global__: se lanza sólo en la CPU y debe devolver `void`.

__device__: se ejecuta sólo en la GPU y puede ser llamado por otras funciones en GPU.

__host__: se permite que lo ejecute la CPU.

El lanzamiento del kernel se realiza con la siguiente llamada:

```
nombre_kernel <<<TAM_GRID, TAM_BLOCK, SHARED_MEMORY>>> (PARÁMETROS);
```

TAM_GRID: es el número de bloques, su tipo es `dim3` (`dim3 grid(16,16);`)

TAM_BLOCK: es el número de hilos por bloque, su tipo es `dim3` (`dim3 block(16,16);`)

SHARED_MEMORY: permite controlar el número de bytes por bloque, por defecto es 0

Cuando se lanza un kernel CUDA sobre la unidad gráfica, la ejecución del programa principal continúa de forma asíncrona, por lo que el programa necesita saber cuándo ha terminado de ejecutarse el kernel lanzado, para esto tenemos la posibilidad de esperar a su terminación por medio de una llamada a `cudaThreadSynchronize()`.

También es importante para depurar saber qué errores puede haber provocado un kernel en su ejecución en GPU, puesto que no es posible acceder a posiciones de memoria de la GPU ni utilizar puntos de ruptura, podemos hacer uso de la función `cudaGetLastError()` que devolverá los errores producidos por el kernel.

2.3. Conclusión

En este capítulo hemos repasado aquellas tecnologías sobre las que se va a apoyar el trabajo, en primer lugar se han explicado los modelos biomecánicos que utilizaremos para simular los comportamientos del hígado frente a las deformaciones. La elección del modelo elástico nos permitirá realizar cálculos más rápidamente.

En segundo lugar hemos repasado la tecnología de aceleración de algoritmos paralelos en GPU, estas técnicas nos permitirán acelerar en gran medida los cálculos que permitirán validar si el comportamiento del modelo biomecánico seleccionado se ajusta a la realidad.

Capítulo 3

Remuestreo de una malla

El problema de remuestrear una malla ha sido estudiado desde hace tiempo con diversos objetivos: desde conseguir un renderizado y visualización en tiempo real de este tipo de mallas, hasta la voxelización o creación del volumen incluido en la malla.

En este capítulo se detallan las distintas aplicaciones en las que es necesario un remuestreo de la malla para reducir la complejidad del modelo inicial, entre ellas la comparación de volúmenes, que necesitará de un remuestreo previo.

El trabajo desarrollado en este capítulo es la transformación de cualquier malla de tetraedros a un volumen en forma de voxels, siendo ésta la mejor manera de llevar a cabo la posterior comparación de volúmenes.

3.1. La malla desestructurada

Según [Prakash and Manohar, 1995], una malla desestructurada está formada por un conjunto de elementos o celdas que cumplen algunas de las siguientes propiedades:

- Tienen un tamaño distinto e irregular
- La información de conectividad y vecindad puede no ser conocida
- Pueden no estar alineadas a los ejes o entre sí
- Pueden formar regiones cóncavas o huecos en el volumen

Esta serie de características hace que las mallas desestructuradas sean realmente difíciles de tratar, tanto para su visualización como para su procesado como volúmenes o en su caso, para realizar análisis físicos mediante el cálculo de elementos finitos.

Debido a eso, resulta necesario hacer una transformación a otro tipo de estructura más regular de la que resulte más sencillo obtener sus propiedades geométricas y topológicas.

3.2. Aplicaciones del remallado

El conjunto de aplicaciones en que podemos utilizar las técnicas de remallado se pueden agrupar en: renderizado de mallas desestructuradas, deformación de volúmenes y comparación de volúmenes.

3.2.1. Obtención del volumen asociado a una malla

Uno de los objetivos de la reestructuración es obtener un volumen en forma de voxels [Kaufman and Shimony, 1986]. La voxelización consiste en generar una representación volumétrica de un objeto geométrico, generando en un espacio 3D discreto una serie de celdas (voxels) lo más cercanas posibles al volumen original, como se aprecia en la figura 3.1. El requisito de este proceso es tener el menor error posible al discretizar en celdas cúbicas el volumen contenido por la malla.

Figura 3.1: *Voxelización de un modelo tridimensional*

Para este proceso podemos encontrar dos técnicas principales. Las que se basan en el orden de la imagen y las basadas en el orden de objeto [Reed et al., 1996].

Las primeras realizan un cálculo por cada voxel de la malla destino buscando aquel elemento de la malla original que caiga en ese lugar, posteriormente se interpola el color entre los elementos colindantes para finalmente asignarlo a ese voxel. Estas técnicas son costosas puesto que en el fondo es un proceso de trazado de rayos con ligeras variaciones. La otra aproximación se basa en el proceso contrario, por cada elemento de la malla original se determinan qué voxels caen en el interior del elemento y se colorean en consecuencia.

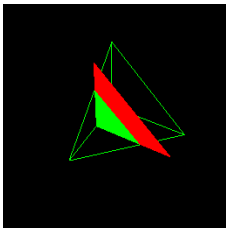

Figura 3.2: *Corte de un plano sobre un tetraedro*

También es posible extender el segundo método a técnicas con planos de recorte. De esta manera, tendremos un número definido de cortes en los que se proyectará la silueta de los elementos que corta, utilizando el algoritmo de proyección de tetraedros propuesto en [Shirley and Tuchman, 1990] con el que evitaremos calcular las intersecciones de forma explícita. Como se puede ver en la figura 3.2, en cada plano de recorte se obtendrá una discretización del corte del plano con la malla, finalmente podremos apilar cada uno de los planos procesados de forma que tengamos un conjunto de voxels que definan el volumen.

Otra forma de resolver este problema es la que proponen en [Dong et al., 2004], donde realizan un triple renderizado en tres texturas que posteriormente se combinan para conseguir el volumen resultante, consiguiendo una tasa de frames interactiva.

3.2.2. Renderizado de mallas desestructuradas

El otro ámbito de estudio de las mallas desestructuradas tiene como objetivo su visualización y renderizado en tiempo real. Puesto que carecen de propiedades que permitirían un visualizado interactivo, muchos trabajos de investigación han intentado resolver este problema. Si bien algunas técnicas lo que hacen es poligonizar los tetraedros que definen la malla, ya sea por superficies independientes o por la generación de abanicos y tiras de tetraedros [King et al., 2001], las que han demostrado ser más rápidas son las basadas en proyección de celdas para obtener el contorno [Max et al., 1990].

Basándose también en el algoritmo de proyección de tetraedros, el algoritmo de proyección propuesto en [Shirley and Tuchman, 1990] consigue convertir cada tetraedro en triángulos que posteriormente se visualizarán en la pantalla con un proceso en GPU [Weiler et al., 2002], de esta manera se puede controlar la densidad del volumen y las funciones de transferencia para resaltar algunas características del volumen. También se ha estudiado cómo visualizar el volumen a partir de una textura 3D [Westermann and Ertl, 1998], sin embargo este proceso es complicado al realizarse mediante trazado de rayos y actualmente se presta a ser acelerado con algoritmos paralelos.

3.2.3. Deformación de volúmenes

La deformación de volúmenes complejos ha sido ampliamente estudiada para conseguir que sea fácilmente aplicable y eficiente en el manejo de gran cantidad de datos. Sin embargo pocos autores consideran los modelos como sólidos, con contenido también en su interior, sólamente como representaciones poliédricas. Un ejemplo claro de aplicaciones en las que es fundamental conservar el volumen completo es el campo de la imagen médica, concretamente en técnicas como el registro de tejido blando o procesado de imágenes volumétricas. En las resonancias magnéticas (figura 3.3) se obtiene una imagen volumétrica o textura 3D. Este resultado permite acceder al interior de la imagen y ver los distintos tejidos que la forman. En el caso de que queramos deformar volúmenes de este tipo es indispensable deformar tanto la superficie como el interior. Para ello, se han de utilizar aquellas técnicas de deformación de volúmenes que tienen en cuenta todo el espacio tridimensional que forma dicho volumen.

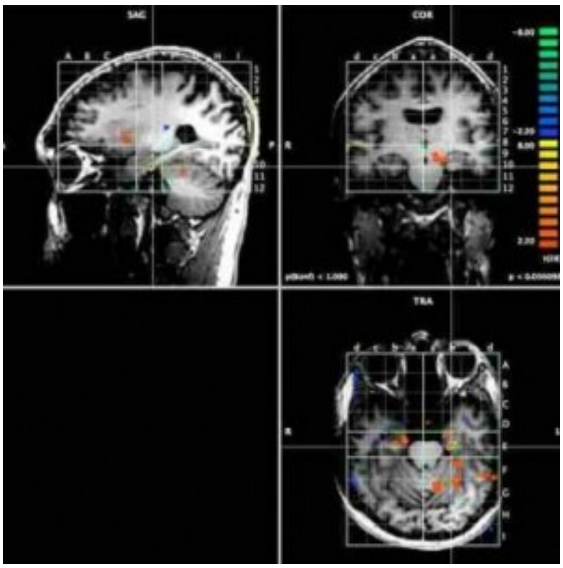

Figura 3.3: *Distintos cortes de una resonancia magnética*

En [Kurzion and Yagel, 1997] se proponen distintas técnicas de deformación de volúmenes basados en texturas 3D mediante la utilización de rayos para la visualización deformada. Estos rayos ven modificada su trayectoria por medio de deflectores. Los deflectores son transformaciones en el espacio que afectan a los rayos que pasan a través de su zona de influencia. Estos mecanismos pueden servir para modelizar deformaciones de traslación, rotación y escalado.

Otra técnica de deformación de volúmenes es la que propusieron [Fang et al., 1996] hace tiempo. Esta aproximación tesela el volumen en octrees cuyos vértices finalmente son los que se deforman, interpolando el color en su interior. También se han aplicado texturas 3D [Rezk-Salama et al., 2001] que permiten, de forma similar, aplicar deformaciones al volumen. El algoritmo ChainMail, presentado por primera vez en [Gibson, 1996], es también utilizado para la deformación de mallas y también se

ha implementado en GPU [Schulze et al., 2008]. En este caso actúa sobre los voxels de un volumen y es capaz de deformar estructuras en tiempo real, sin embargo no garantiza que la deformación sea físicamente correcta.

Encontramos otras aproximaciones que forman una malla de tetraedros a partir del volumen que luego son convertidos a superficies poligonales para su visualización, técnica que aplican [Bhaniramka and Demange, 2002]. La implementación de este método conlleva una reordenación de los polígonos, con la consecuente carga computacional para la CPU (aunque acelerable en GPU). Además, por cada voxel de volumen que tenemos, debe dividirse en al menos cinco tetraedros, siendo muy grande el número de polígonos que finalmente habrá que procesar.

Por otra parte, en [Chua and Neumann, 2000] se realiza una deformación teniendo en cuenta sólamente las coordenadas de texturas. Se divide el volumen en una serie de cubos y la deformación se realiza trasladando las coordenadas de textura para cada vértice de este modelo. En términos interactivos, en cada fotograma sólo se tendrán que recalcular las coordenadas de textura, no siendo necesario modificar la geometría. Seguidamente se computan una serie de planos de recorte para obtener el volumen deformado.

Finalmente, en [Rhee et al., 2007] consideran también la deformación de un volumen respecto a la obtenida de los sensores, teniendo en cuenta los distintos tejidos y grados de deformación que deben tener.

3.3. Remuestreo de volúmenes implementado

Como se ha comentado anteriormente, el objetivo es transformar un modelo formado por tetraedros a un volumen en forma de voxels. De esta manera se podrá realizar una comparación de volúmenes a partir de la malla de elementos finitos.

El método propuesto sigue las indicaciones que describen [Weiler and Ertl, 2001] para la visualización de mallas desestructuradas pero orientado a la transformación del modelo a un volumen. Este método se ha implementado utilizando shaders y CUDA para una mayor aceleración de los cálculos.

3.3.1. Ajuste del espacio al modelo

En primer lugar, se define el espacio en el que se incluirá el modelo como $m \times n \times l$ el número de voxels. En el caso que nos ocupa, al guardarse el resultado sobre una textura 3D, el límite de la tarjeta gráfica sobre la que se ejecuta el algoritmo (GeForce 9800GT) está limitado a $256 \times 256 \times 256$. Por lo tanto, hay que ajustar una caja sobre el modelo que queremos transformar en volumen. Esta caja será la caja de inclusión del modelo. De esta manera, el modelo quedará enmarcado dentro la caja y posteriormente será transformado en un volumen de máximo $256 \times 256 \times 256$. Se aprovecha así todo el espacio de la textura 3D que permite la tarjeta gráfica, a pesar

de que el modelo quedará deformado en los ejes que ocupe menos espacio. De este modo se ganará precisión en ellos.

3.3.2. Procesamiento de los tetraedros

El algoritmo desarrollado hace varias pasadas sobre la malla de tetraedros, en cada pasada avanza en la dirección Z, haciendo un total de 256 pasos. De esta manera, en cada pasada se obtendrá una rasterización 2D de la proyección de los tetraedros sobre el plano.

En la figura 3.4 se puede ver una muestra del resultado obtenido. Este es un caso simple con un modelo y un único tetraedro. En la figura el plano Z se aleja y se proyectan distintos cortes del tetraedro. Posteriormente todos ellos se apilan formando una textura 3D que almacenará el volumen del tetraedro.

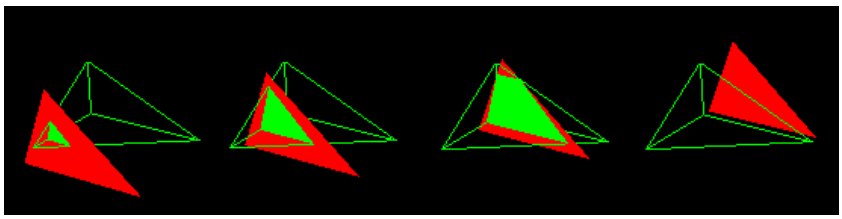

Figura 3.4: *Cuatro proyecciones de un tetraedro sobre un plano, con Z cada vez mayor. La parte roja es la proyección de las partes que no caen dentro del plano, la parte verde es la parte interior del tetraedro que corta el plano.*

Para conseguir estas proyecciones es necesario calcular qué partes del tetraedro quedan dentro (verde) y qué partes quedan fuera (rojo) del plano de proyección, para ello utilizaremos los shaders que, mediante interpolación de coordenadas, determinarán si el punto de corte está dentro o fuera del tetraedro. Serán las coordenadas baricéntricas de un tetraedro las que permitan conocer la posición de un punto respecto al tetraedro (dentro o fuera). Las coordenadas baricéntricas indican la distancia a cada uno de los vértices de un tetraedro de cualquier punto del espacio.

Sea un tetraedro de vértices $\overrightarrow{v_1}$, $\overrightarrow{v_2}$ y $\overrightarrow{v_3}$ y $\overrightarrow{v_4}$ y un punto P perteneciente a dicho tetraedro. Si las coordenadas baricéntricas de P son $(\lambda_1, \lambda_2, \lambda_3, \lambda_4)$, P puede escribirse como:

$$\overrightarrow{v_P} = \lambda_1\overrightarrow{v_1} + \lambda_2\overrightarrow{v_2} + \lambda_3\overrightarrow{v_3} + \lambda_4\overrightarrow{v_4} \tag{3.1}$$

donde

$$\lambda_1 + \lambda_2 + \lambda_3 + \lambda_4 = 1 \tag{3.2}$$

$$\lambda_4 = 1 - (\lambda_1 + \lambda_2 + \lambda_3) \tag{3.3}$$

y donde

$$\begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{pmatrix} = \mathbf{T}^{-1}(\overrightarrow{v_P} - \overrightarrow{v_4}) \tag{3.4}$$

con

$$\mathbf{T} = \begin{pmatrix} x_1 - x_4 & x_2 - x_4 & x_3 - x_4 \\ y_1 - y_4 & y_2 - y_4 & y_1 - y_4 \\ z_1 - z_4 & z_2 - z_4 & z_1 - z_4 \end{pmatrix} \tag{3.5}$$

y

$$\overrightarrow{v_i} = (x_i, y_i, z_i) \tag{3.6}$$

El problema se reduce así a la inversión de la matriz $\mathbf{T}$, que al ser 3×3 es un proceso sencillo. Propiedades a tener en cuenta sobre las coordenadas baricéntricas:

- P está dentro del triángulo $\leftrightarrow 0 < \lambda_i < 1, \forall i \in [1, 2, 3, 4]$
- Se puede realizar una interpolación lineal entre los vértices y sus coordenadas baricéntricas

Con estos cálculos procedemos de la siguiente manera: dado un tetraedro cuyos vértices son $\overrightarrow{v_1}$, $\overrightarrow{v_2}$, $\overrightarrow{v_3}$ y $\overrightarrow{v_4}$ y el plano de corte en z_{proy} donde realizaremos la proyección, se calculan las coordenadas baricéntricas de los cuatro vértices proyectados en ese plano: $\lambda_1 = [x_1, y_1, z_{proy}]$, $\lambda_2 = [x_2, y_2, z_{proy}]$, $\lambda_3 = [x_3, y_3, z_{proy}]$ y $\lambda_4 = [x_4, y_4, z_{proy}]$. Nótese que la coordenada z de los puntos siempre es z_{proy}, con esto llevamos cada uno de los vértices al plano de recorte de la iteración actual.

En este momento surge el problema de conocer aquellas caras que forman la silueta del tetraedro, cuyos vértices serán los que hay que proyectar. La silueta puede estar formada por uno o dos triángulos, como se ve en la figura 3.6, dependiendo de la orientación del tetraedro respecto al plano de recorte Z. En (a) el tetraedro tiene tres caras frontales, la silueta la forman los vértices de la cara trasera. En (b) se tienen dos caras frontales, por lo que la silueta la forman esas dos caras. En (c) sólo una cara es frontal al plano de recorte por lo que la silueta la forma esa misma cara.

En la fila central de la figura se puede ver la silueta que se extrae en los tres casos, además del orden en que se deben dibujar esas caras triangulares: sentido antihorario. En la última fila se muestra cómo se ha dibujado la silueta en los tres casos, la zona roja son los píxeles del plano de recorte que están fuera del triángulo y la verde los que han caído dentro.

Para averiguar el número de caras frontales se toma un punto D cuyas coordenadas baricéntricas son $\lambda_i = 0{,}25$, figura 3.5. Esto garantiza que el punto se encuentra justo en el centro del tetraedro. De cada una de las cuatro caras, se calcula la normal externa $\overrightarrow{n}$ y se proyecta el punto D sobre el plano que define $\overrightarrow{n}$, dando lugar a P. Seguidamente se obtiene el vector $\overrightarrow{v} = \overrightarrow{v_P} - \overrightarrow{v_D}$ y se calcula el ángulo α entre $\overrightarrow{v}$ y el vector normal al plano de proyección $\overrightarrow{n_{z_{proy}}} = [0, 0, -1]$. Si $\cos(\alpha) < 0$ la cara no es delantera, si $\cos(\alpha) > 0$ la cara es delantera.

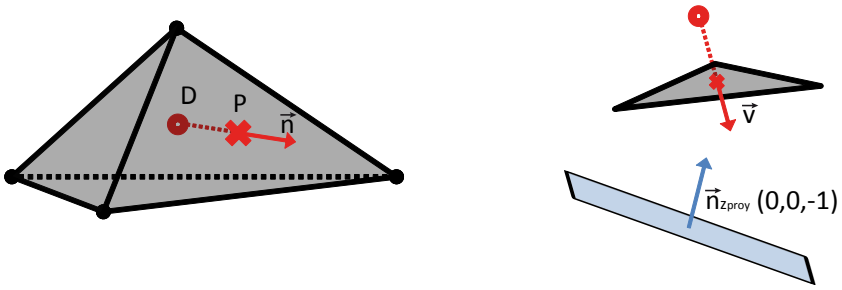

Figura 3.5: *Proceso de proyección del punto interior sobre la cara para averiguar el ángulo que forma con el vector* z_{proy}.

Una vez identificadas las caras delanteras, almacenamos las coordenadas baricéntricas de los triángulos que forman la silueta sobre el plano de proyección. Un triángulo en los casos (a) y (c) y dos triángulos en el caso (b). Con esto termina el preprocesado de los tetraedros, cuyo resultado son las coordenadas baricéntricas de la silueta que se proyecta sobre el plano de proyección de la iteración actual. Todo este proceso, al ser idéntico para cada tetraedro, se ha implementado sobre CUDA, lo cual ha acelerado en gran medida el cálculo, sobretodo cuando el número de tetraedros es muy elevado.

Si bien en casos en los que el número de tetraedros no es demasiado elevado el algoritmo en GPU no resulta ser más rápido que en CPU, cuando tenemos un número de tetraedros del orden de cientos de miles, la aceleración sufrida es importante. En la tabla 3.1 se puede ver el tiempo medio medido para el procesamiento en CPU y GPU del algoritmo de transformación en volúmenes: procesado de tetraedros (CPU y GPU) + cálculo de intersección (shaders).

Nº tetraedros	Tiempo CPU (seg)	Tiempo GPU (seg)	Aceleración
179885	64.88	45.34	31.18 %
204816	73.55	53.49	27.27 %

Tabla 3.1: *Diferencia entre el algoritmo de voxelización lanzado sobre CPU y GPU y aceleración sufrida.*

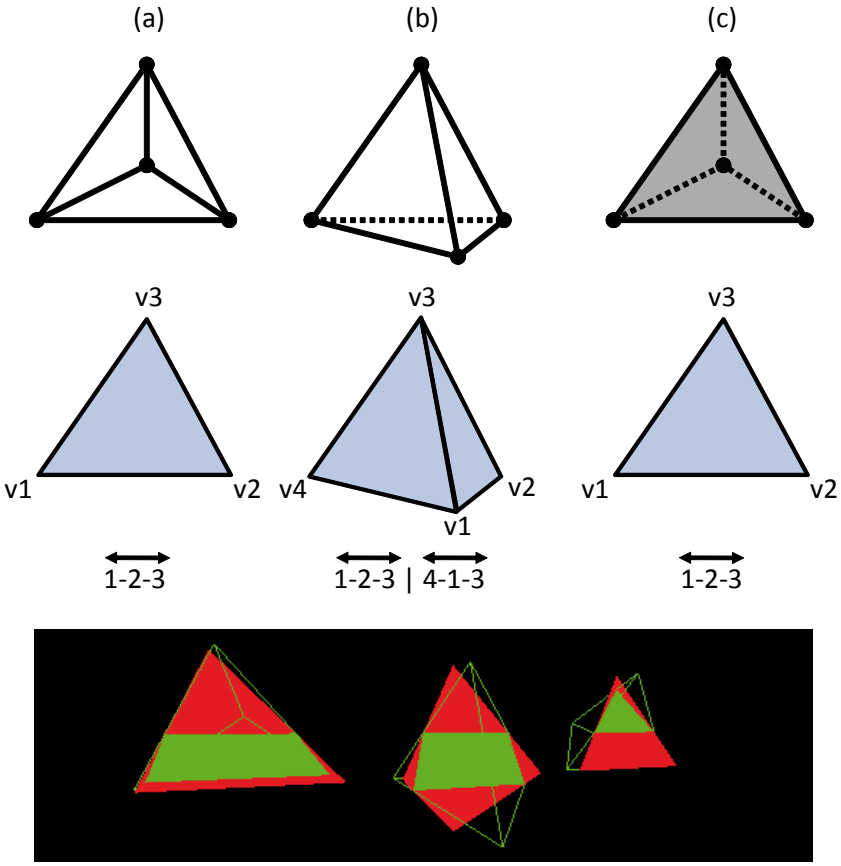

Figura 3.6: *Diferencia de la silueta y orden de dibujo según el número de caras delanteras de los tetraedros.*

3.3.3. Cálculo de la intersección

Como hemos comentado, el cálculo de la intersección del plano de corte z_{proy} con los tetraedros, se hará de forma indirecta gracias a una programación en shaders.

De cada tetraedro y sobre cada plano se almacenan las siluetas. Activamos un `framebuffer` para almacenar los resultados gráficos y donde se pintan las caras que forman la silueta (una o dos según el caso). En cada vértice de esta silueta se asocia un atributo correspondiente a su coordenada baricéntrica respecto al tetraedro del que viene esa silueta (calculada en el preproceso). Con esto hemos dibujado tres vértices con un atributo cada uno, como se puede ver en el cuadro 3.1.

Código fuente 3.1: *Código que pinta un triángulo y añade como atributos las coordenadas baricéntricas de cada vértice*

```
1 //asociar la variable con el shader
2 indice_bari = lGetAttribLocation(pid, "bari");
```

```
3
4  void dibujaTriangulo(Punto v1, Punto v2, Punto  v3,
5       GLfloat landa1[4], GLfloat  landa2[4], GLfloat  landa3[4]){
6    glBegin(GL_TRIANGLES);
7      glColor3f(1,1,1);
8      glVertexAttrib4f(indice_bari,landa1[0],landa1[1],landa1[2],landa1[3]);
9      glVertex3f(v1.x,v1.y,planoRecorte);
10
11     glColor3f(1,1,1);
12     glVertexAttrib4f(indice_bari,landa2[0],landa2[1],landa2[2],landa2[3]);
13     glVertex3f(v2.x,v2.y,planoRecorte);
14
15     glColor3f(1,1,1);
16     glVertexAttrib4f(indice_bari,landa3[0],landa3[1],landa3[2],landa3[3]);
17     glVertex3f(v3.x,v3.y,planoRecorte);
18   glEnd();
19 }
```

El shader de vértice traspasa la información al de fragmento, cuadro 3.2. Gracias a la utilización de los shaders, las coordenadas baricéntricas pasadas como atributos se interpolan en cada uno de los fragmentos. De esta manera, en cada fragmento se comprueba si $0 \leq \lambda \leq 1$, en este caso el fragmento se encuentra dentro del tetraedro y lo pintamos de verde, en caso contrario se pinta de rojo (cuadro 3.3).

Código fuente 3.2: *Shader de vértice*

```
1  attribute vec4 bari;
2  varying vec4 bariF;
3
4  void main(){
5          gl_Position = ftransform();
6          gl_FrontColor=gl_Color;
7          gl_BackColor=gl_Color;
8          gl_FrontSecondaryColor=gl_Color;
9          gl_BackSecondaryColor=gl_Color;
10         bariF=bari;
11         gl_TexCoord[0]=gl_MultiTexCoord0;
12 }
```

Código fuente 3.3: *Shader de fragmento*

```
1  varying vec4 bariF;
2  uniform sampler3D tex;
3
4  void main(){
5          if(bariF.x>=0 && bariF.y>=0 && bariF.z>=0 && bariF.w>=0
6                  && bariF.x<=1 && bariF.y<=1 && bariF.z<=1 && bariF.w<=1){
7                  gl_FragColor = vec4(0,1,0,1); //lo pintamos verde
8          } else {
9                  gl_FragColor = vec4(1,0,0,1); //lo pintamos rojo
10                 //discard; //también podemos descartar el fragmento
```

```
11              }
12      }
```

De esta manera tendremos en verde aquellos elementos del plano de proyección que estén dentro del tetraedro que estamos procesando. Se repite este proceso para todos y cada uno de los tetraedros del modelo y para los 256 planos de recorte. Con esto conseguimos almacenar en un `framebufferTexture3D` las distintas capas. Finalmente se pasa el contenido del `framebuffer` a memoria de CPU en una matriz de $256 \times 256 \times 256$ elementos de tipo `unsigned char`. Esta matriz contendrá el volumen asociado al modelo de elementos finitos inicial, sin importar su forma, tamaño o posición.

3.4. Conclusión

En los puntos anteriores se han descrito las diferentes aplicaciones en las que la estructuración de mallas resulta fundamental y la que se ha llevado a cabo en este trabajo. Esta transformación a volumen permitirá realizar la comparación entre dos conjuntos de forma más eficiente. El alto coste computacional de los algoritmos de remallado y voxelización existentes se ha afrontado con un desarrollo en tarjetas multiprocesador de tipo GPU (GPGPU) con la utilización de CUDA para el preproceso y cálculo de las coordenadas baricéntricas de la proyección y con shaders para el cálculo indirecto del corte de cada plano.

Capítulo 4

Comparación de volúmenes

Uno de los objetivos del proyecto es evaluar la calidad de la deformación simulada respecto a la deformación real que se habrá escaneado. La voxelización de las mallas de elementos finitos es necesaria para poder realizar esta comparación puesto que es muy costoso comparar automáticamente mallas formadas por tetraedros. Por ello dispondremos de dos volúmenes en forma de vóxels que habrá que comparar para saber lo parecidos o distintos que son y actuar en consecuencia.

La comparación de volúmenes es un problema poco estudiado hasta este momento. Las técnicas presentadas en los siguientes apartados corresponden a métricas o coeficientes inicialmente pensados para comparar segmentaciones 2D. Estos coeficientes son fácilmente aplicables a comparación de volúmenes 3D tratándolos como segmentaciones, es decir, imágenes binarias en las que sólo aparece fondo y objeto.

A continuación se presentan las métricas clásicas de comparación de segmentaciones, posteriormente se amplían con algunas propuestas más recientes y por último se propone una combinación de ellas que nos pueda dar más información sobre ambos volúmenes.

4.1. Métodos clásicos de diferencia

En la evaluación de segmentaciones podemos encontrar varios coeficientes clásicos de comparación entre dos imágenes binarias. La extensión de estos coeficientes a la comparación de volúmenes es trivial puesto que sólo habría que ampliar una dimensión el recorrido que hacen estos algoritmos.

Sean X e Y los dos conjuntos que queremos comparar. Definiremos las siguientes variables: $a = |X \cap Y|$; $b = |X - Y|$; $c = |Y - X|$; $d = \overline{|X \cup Y|}$. Basados en estas variables se presentan un conjunto de coeficientes. Todas estas medidas son independientes de la forma del volumen (no tienen en cuenta la distancia que puede haber entre elementos)

sólo se basan en el número de elementos que solapan (o no) para dar una evaluación de la similitud de los conjuntos. Sin embargo, son las distancias más comunes para evaluar la calidad de las segmentaciones y son utilizadas en multitud de trabajos como se verá en la sección 4.3.

4.1.1. Jaccard

El coeficiente de similaridad de Jaccard [Jaccard, 1901] permite comparar la similitud de dos conjuntos de datos. Su aplicación es muy sencilla y viene dado por la ecuación 4.1.

$$JC = \frac{|X \cap Y|}{|X \cup Y|} = \frac{a}{a+b+c} \tag{4.1}$$

Este coeficiente provee un número entre 0 y 1, donde 1 se dará cuando los conjuntos tienen todos sus elementos solapados y 0 cuando no tengan ninguno.

4.1.2. Dice

El coeficiente de Dice [Dice, 1945] es similar a Jaccard y se obtiene como muestra la ecuación 4.2.

$$DS = \frac{2|X \cap Y|}{|X| + |Y|} = \frac{2a}{2a+b+c} = \frac{2JC}{JC+1} \tag{4.2}$$

Debido a la equivalencia con Jaccard, sus valores definen de igual forma la diferencia entre ambos conjuntos. El coeficiente de Dice toma valores desde 0 hasta 1 según incremente el nivel de solape. Este coeficiente beneficia más los elementos solapados y penaliza menos las diferencias que el coeficiente de Jaccard.

4.1.3. Tanimoto

El coeficiente de Tanimoto (o coeficiente de Jaccard extendido) propuesto en el trabajo de [Tanimoto, 1957] da una medida en la que participa por primera vez el número de elementos que son fondo. Es el resultado de la operación de la ecuación 4.3

$$TN = \frac{|X \cap Y| + \overline{|X \cup Y|}}{|X \cup Y| + \overline{|X \cap Y|}} = \frac{a+d}{a+2b+2c+d} \tag{4.3}$$

Este coeficiente da un valor 1 si los datos son iguales y 0 si sus regiones son totalmente disjuntas y ocupan todo el espacio. Con el coeficiente de Tanimoto la posición de los conjuntos es significativa ya que variarán los píxeles de fondo.

4.1.4. Volume Similarity

El último coeficiente clásico de comparación de segmentaciones se cita por los autores [Cárdenes et al., 2009] y se define en la ecuación 4.4.

$$VS = 1 - \frac{||X| - |Y||}{|X| + |Y|} = 1 - \frac{|b - c|}{2a + b + c} \tag{4.4}$$

Tomará valor 1 si el número de elementos son iguales en los dos conjuntos y 0 si alguno de ellos no tiene elementos. Sin embargo, no será útil para este proyecto puesto que su valor es independiente de la posición de los conjuntos para su comparación, cosa que es significativa para este trabajo.

4.2. Métodos basados en distancias

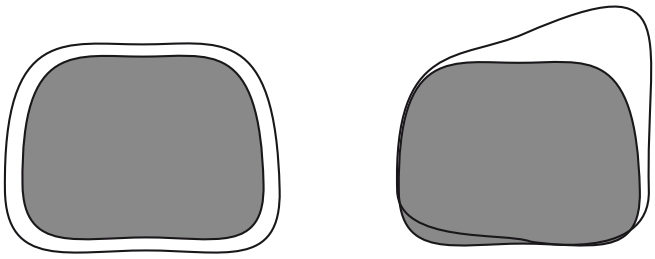

Figura 4.1: *Comparación de dos segmentaciones distintas de un objeto. Tanto (a) como (b) tienen el mismo número de elementos mal segmentados.*

Para mejorar los resultados de los coeficientes anteriores, es necesario añadir información sobre la forma del objeto. Pongamos de ejemplo el caso de la figura 4.1. En este caso, los coeficientes anteriores darían un resultado idéntico para ambos casos de la comparación (a) o (b). Sin embargo, en (b) se estaría realizando una deformación incorrecta del volumen mientras que la aproximación de (a) sería aceptable.

Los coeficientes que aparecen en este apartado tienen en cuenta también las distancias de los elementos mal clasificados a donde deberían estar (o a los más cercanos) para mejorar el proceso de evaluación con información sobre la forma y contorno de los conjuntos. El valor de $d_V(i)$ representa la distancia mínima del elemento i (pixel en

2D, voxel en 3D) al conjunto V como se puede ver en la ecuación 4.5. Donde $||.||$ define la distancia euclídea.

$$d(i) = \begin{cases} 0, & i \in V \\ \text{mín}_{v \in V} ||i - v|| & i \notin V \end{cases} \tag{4.5}$$

4.2.1. Yasnoff

La distancia propuesta por Yasnoff [Yasnoff et al., 1977] se define en la ecuación 4.6.

$$YASNOFF = \frac{1}{N} \sum_{i=1}^{N} d(i)^2 \tag{4.6}$$

Para los N elementos del conjunto, se suman los cuadrados de las distancias $d(i)$. La normalización de esta medida se hace dividiendo su valor por el número de elementos del conjunto.

4.2.2. Factor of Merit

Otra distancia habitualmente utilizada es la del Factor de mérito (FOM: Factor of Merit) definida en [Pratt, 1978] y se muestra en la ecuación 4.7.

$$FOM = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{1 + d(i)^2} \tag{4.7}$$

Esta distancia dará un valor normalizado entre 0 y 1 cuyo significado es el mismo que la distancia Yasnoff. Ambas distancias se calculan únicamente para los elementos mal clasificados.

4.2.3. Hausdorff

Otra medida clásica de comparación con distancias es el coeficiente de Hausdorff que se definió en [Huttenlocher et al., 1993] y [Baudrier et al., 2008]. Esta medida se define según la ecuación 4.8.

$$HAUSDORFF(X,Y) = \text{máx}\,(h(X,Y), h(Y,X)) \tag{4.8}$$

$$h(X,Y) = \underset{x\in X}{\text{máx}}\ \underset{y\in Y}{\text{mín}}\,||x-y|| \tag{4.9}$$

La función $h(X,Y)$ es la distancia Hausdorff directa y da como resultado el máximo de las distancias mínimas de cada punto del conjunto X al conjunto Y utilizando, por ejemplo, la distancia euclídea.

La distancia Hausdorff simétrica $H(X,Y)$ devolverá el máximo de las distancias $h(X,Y)$ y $h(Y,X)$, lo que mide el grado de disparidad de los dos conjuntos, es decir: la del punto más alejado de un conjunto respecto al otro.

$$H(X,Y) = \text{máx}\,(\underset{x\in X}{\text{máx}}\ \underset{y\in Y}{\text{mín}}\,||x-y||, \underset{y\in Y}{\text{máx}}\ \underset{x\in X}{\text{mín}}\,||x-y||) \tag{4.10}$$

4.2.4. Clásicas con distancias

En el trabajo de [Cárdenes et al., 2009] se modifica el coeficiente de Jaccard añadiendo las distancias (ecuación 4.5), lo que permite obtener un valor más preciso de similaridad entre los conjuntos, ecuación 4.11.

$$JCd = \frac{a}{a + \sum_{x\in X} d(x) + \sum_{y\in Y} d(y)} \in [0,1] \tag{4.11}$$

Esta nueva distancia da más información sobre ambos conjuntos ya que no se basa únicamente en el número de elementos mal clasificados sino que se combina con las distancias al más próximo del otro conjunto.

Según los experimentos realizados en el trabajo de [Cárdenes et al., 2009], la diferencia se vuelve mayor con este nuevo coeficiente que con el Jaccard original, lo que permitirá dar con más precisión la calidad de la segmentación.

4.3. Trabajos anteriores

Los coeficientes descritos han sido aplicados a la evaluación de segmentaciones. Para esta evaluación, normalmente se cuenta con una segmentación de referencia llamada *ground truth* o *gold standard* que se considera perfecta. Existen trabajos que comparan la calidad de las segmentaciones respecto a los niveles de gris de las imágenes, como las basadas en entropía [Zhang et al., 2004], pero este tipo de medidas

no nos sirven para la comparación de volúmenes, donde tenemos representaciones binarias (objeto y fondo). Otros trabajos utilizan las medidas de comparación clásicas descritas anteriormente, como en [Strasters and Gerbrands, 1991] que, entre otras, utilizan el Factor of Merit al que le añaden un factor de escalado γ que le permite comparar segmentaciones dando el resultado de la ecuación 4.12. El factor γ se utiliza para cambiar la contribución de los errores a la medida. De esta manera se eliminan las dependencias con el tamaño de las imagenes para comparar el Factor of Merit de varias segmentaciones.

$$FOM = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{1+\gamma d(i)^2} \tag{4.12}$$

La medida utilizada en [Huttenlocher et al., 1993] es el coeficiente de Hausdorff, que extienden añadiendo transformaciones de posición para cuadrar mejor los conjuntos (Hausdorff original es para posiciones fijas) dando como resultado la distancia Hausdorff real entre ambos conjuntos sin importar la zona de la imagen donde se encuentren. También se utiliza este coeficiente en el algoritmo que proponen [Aspert et al., 2002]. En este caso, el problema que intentan resolver es la diferencia entre dos mallas de triángulos y la distancia utilizada es punto-punto y punto-triángulo.

En el artículo de [He et al., 2008] donde utilizan para segmentar distintas variaciones del algoritmo de *snakes*, las comparaciones las realizan con los coeficientes de Jaccard y Hausdorff. Variables que le sirven para determinar si las formas del contorno de la segmentación han sido extraídas correctamente. Por otra parte, el trabajo de [Pichon et al., 2004] introduce otra manera de tratar las medidas de distancias. Utiliza las medias y las desviaciones típicas para decidir con ellas la discrepancia entre volúmenes. Con estos datos calcula la probabilidad de que un voxel haya sido mal clasificado, además del error medio de los n voxels peor clasificados.

4.4. Herramientas para calcular distancias

El cálculo de la distancia definida en la ecuación 4.5 es un cómputo costoso, sobretodo al tener que iterar por cada elemento del primer conjunto sobre todos los del segundo, buscando de esta manera la distancia mínima. Si lo extendemos a tres dimensiones, se vuelve inmanejable. Para evitar este costoso cómputo podemos utilizar algoritmos de Transformada de distancia (Distance Transform: DT) que permitan calcular todas las distancias de forma rápida y eficiente. Los algoritmos clásicos de DT calculan, en imágenes binarias, la distancia de cada píxel objeto al fondo. En este trabajo se ha implementado un algoritmo de cálculo de distancias aproximadas en paralelo sobre GPU.

4.4.1. Cálculo paralelo de la DT

El primer caso en que se presenta un cálculo paralelo de la DT Euclídea (EDT) es en [Yamada, 1984]. El funcionamiento de este algoritmo se basa en lanzar un kernel sobre cada uno de los píxeles de la imagen de forma paralela e independiente, actualizando así su valor. Se continúa haciendo varias pasadas de esta manera y se termina cuando entre una iteración y la siguiente no cambie ningún valor. Por lo tanto, el coste computacional es proporcional a la máxima distancia que haya en la DT. Este algoritmo es candidato a ser acelerado en GPU, con una buena planificación de bloques CUDA es posible conseguir una aceleración importante para el cálculo rápido de la EDT. Teniendo en cuenta que habrá muchos accesos a memoria (27 en el caso de tres dimensiones), habrá que organizar bien y sincronizar las lecturas para evitar fallos de caché y minimizar los retrasos que puedan causar.

Otros autores han demostrado que cálculos en GPU pueden servir para obtener de forma rápida y eficiente el diagrama de Voronoi. El ejemplo de [Cao et al., 2010] sirve para ilustrar esta afirmación, mediante CUDA son capaces de calcular el diagrama de Voronoi de una imagen, obteniendo tiempos muy inferiores a los de los algoritmos clásicos para esta tarea. Sin embargo, no está preparado para cálculo directo de la DT sino que requiere un proceso posterior para esta obtención, que añade un coste secuencial.

4.4.2. Cálculo aproximado de la DT

Muchos autores han apostado por el cálculo de la DT no exacto, lo que les permite reducir el coste temporal de los algoritmos a costa de la precisión alcanzada. Los algoritmos presentados en [Borgefors, 1986] actúan de forma secuencial sobre las imágenes, el autor define también una cota máxima del error cometido respecto a la DT exacta.

En [Saito and Toriwaki, 1994] se explica otra forma de realizar la EDT con un algoritmo en tres pasadas sobre la imagen. Este algoritmo ve incrementado su coste temporal según la forma de la figura, cuanto más grande sea, más le cuesta calcular las distancias. Respecto al algoritmo original, es más rápido para figuras pequeñas. Su pararelización es inviable puesto que cada una de las pasadas se hace respecto al resultado de la anterior.

El trabajo de [Baudrier et al., 2008] compara dos imágenes binarias X a Y y crea dos imágenes que serán las restas $X - Y$ y $Y - X$ respectivamente, se considera como fondo Y y X respectivamente en estas imágenes restadas. Se calcula entonces la DT con esta nueva definición de fondo y como resultado se obtiene la distancia de cada elemento de un conjunto al más cercano del otro conjunto (considerado como fondo). Es una solución similar a la que se propone en este trabajo. Consiguiendo una paralelización masiva en GPU, el cálculo de la DT se verá acelerado en gran medida.

4.5. Algoritmo de comparación

La comparación de los volúmenes utiliza la transformada de distancia para hallar una serie de medidas (Hausdorff, medias, desviaciones típicas...) que permitirán dar un grado de ajuste entre dos volúmenes. A continuación se describe el algoritmo de DT realizado enteramente sobre GPU y cómo se han utilizado los coeficientes aplicados para comparar volúmenes.

4.5.1. Cálculo de la distancia en GPU

El algoritmo seguido para el cálculo de la DT aproximada se basa en el Vector Distance Transform [Satherley and Jones, 2001], uno de los algoritmos aproximados de DT más precisos y rápidos. El algoritmo implementado en este trabajo calcula sobre GPU, para cada volumen, la distancia mínima de cada voxel al voxel objeto más cercano. De esta manera, se obtendrá una matriz de distancias para cada volumen. Posteriormente, de esta matriz se podrán extraer todas las distancias $d(i)$ de forma muy eficiente.

La implementación paralela en GPU aplica un *kernel* como el de la figura 4.2 sobre cada elemento del volumen (voxel). Cada pasada de este kernel transferirá las distancias desde el objeto hasta el límite del volumen. Cada voxel almacena las coordenadas relativas al voxel objeto más cercano. Se realizan varias pasadas hasta que, entre una iteración y la siguiente, no haya cambios en ninguna coordenada. Así, la mínima distancia de cualquier voxel al objeto puede ser obtenida aplicando simplemente la fórmula de la distancia euclídea sobre las coordenadas anotadas. La figura 4.4 muestra un ejemplo de la aplicación del kernel de la figura 4.2 para un caso bidimensional.

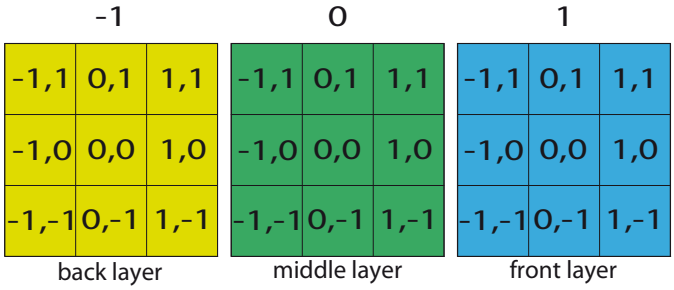

Figura 4.2: *Kernel aplicado a un cubo de 3x3x3 voxels.*

En la iteración dos, la distancia del pixel seleccionado al objeto es: $\sqrt{2^2 + (-1)^2} = 2{,}236$. Sin embargo, en puntos más lejanos con distancias idénticas donde existe ambigüedad se puede llegar a perder precisión. En el trabajo de [Satherley and Jones, 2001] se demuestran y cuantifican los posibles errores para dos y tres dimensiones.

Para validar el algoritmo y averiguar cuánto error se comete, se ha implementado un algoritmo iterativo. Este algoritmo lanza una búsqueda por cada voxel, almace-

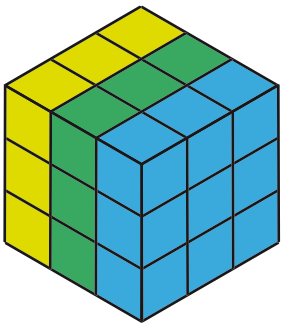

Figura 4.3: *Cubo de 3x3x3 voxels sobre el que se aplica el kernel.*

nando la distancia mínima al objeto. En la fase de validación, para un objeto con una distancia media de 75.5856 voxels se ha obtenido un error medio de 0.4605 voxels entre el algoritmo en GPU aproximado y el algoritmo de distancias exacto sobre CPU. Esto implica un error de menos del 0.6 %, demostrando que es una solución muy precisa.

La implementación ha sido llevada a cabo en el lenguaje CUDA de NVIDIA sobre una tarjeta gráfica GeForce 9800GT [NVIDIA, 2009]. Debido a restricciones de memoria, el tamaño del volumen se fijó a un máximo de $256 \times 256 \times 256$ voxels. Los tamaños seleccionados para lanzar el kernel CUDA implementado han sido: hilos de 256×1 elementos en bloques organizados en una rejilla (grid) de 256×256. Con esta configuración, cada bloque escanea una capa del volumen. El resultado de cada iteración se almacena en una segunda matriz, para evitar la sobreescritura de la matriz original. Esto se hace porque CUDA no garantiza la ejecución en orden entre hilos; en el caso de no utilizar dos matrices (una para lectura y otra para escritura) podría degenerar en una situación inestable debido a la concurrencia. Posteriormente, ambas matrices son intercambiadas y se lanzan las siguientes iteraciones hasta que no se observen cambios entre las matrices. Cuando el algoritmo ha terminado, la matriz obtenida representa la transformada de distancia (ecuación 4.5).

El coste temporal de este algoritmo es proporcional al tamaño del objeto respecto al tamaño máximo del volumen que lo contiene. En otras palabras, el algoritmo iterará tantas veces como sea la distancia máxima presente en el espacio seleccionado para el estudio. El coste temporal del algoritmo sobre GPU es $O(max(m,n,l))$, en un volumen de tamaño $m \times n \times l$. La distancia máxima en un espacio 3D de dimensiones $m \times n \times l$ puede ser determinada como el camino entre los vértices opuestos de la caja que contiene el volumen. Estos valores serán almacenados como tres coordenadas que constituirán la tupla $[m,n,l]$, por lo tanto, para el tamaño seleccionado $256 \times 256 \times 256$ las iteraciones máximas serán 256 y la distancia máxima posible será $d_{max} = \sqrt{256^2 + 256^2 + 256^2} = 443{,}4$.

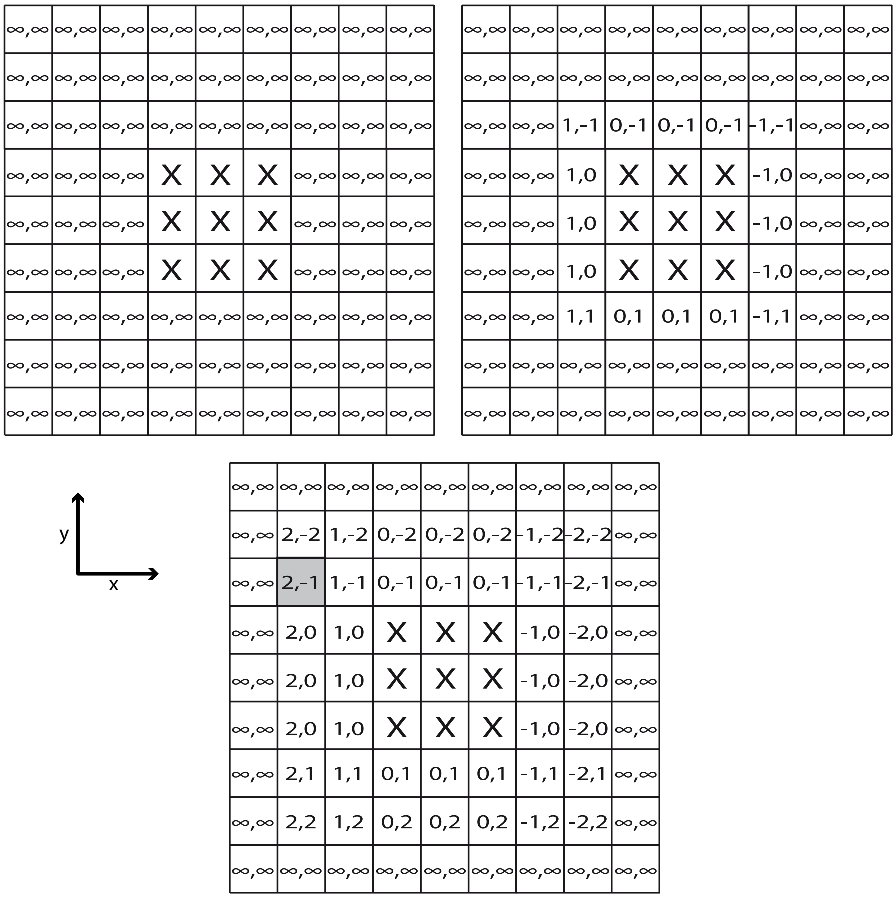

Figura 4.4: *Iteraciones 0, 1 y 2 sobre un objeto 2D de 3x3 pixels.*

4.5.2. Cálculo de las distancias exactas

Para completar el trabajo, se ha desarrollado también un algoritmo que genera la DT de un volumen 3D con las distancias exactas. Aunque el proceso resulta más lento que el algoritmo en GPU de la sección 4.5.1, es interesante comprobar la variabilidad de los resultados al aplicar distancias exactas frente a las aproximadas.

El algoritmo desarrollado sigue los pasos del trabajo de [Maurer et al., 2003] ampliado a tres dimensiones. El algoritmo funciona de la siguiente manera:

- Para cada corte en la dirección z del volumen 3D, se recorren las líneas del corte 2D en la dirección x como se puede ver en la figura 4.5.

Fase 1 Se busca aquél voxel objeto más cercano en la dirección y. El resultado es una lista de voxeles candidatos a ser los más cercanos en la línea recorrida

Fase 2 Sobre cada elemento de la lista de candidatos se actua de la siguiente

manera:

- Inicio: Se extraen tres puntos de la lista p_1, p_2 y p_3
- Se traza un línea recta que pasa por el punto medio de $p_1 - p_2$ y es perpendicular a ellos y otra línea recta que pasa por el punto medio de $p_2 - p_3$ y es perpendicular a ellos
- Si las rectas trazadas se cruzan antes de la línea del plano que se está recorriendo, el punto p_2 deja de ser candidato
- En caso contrario se extrae un nuevo punto de la lista de candidatos y se itera hasta que no queden puntos por procesar

Fase 3 Por último, para cada elemento de la línea recorrida se calcula la distancia mínima a todos los elementos presentes en la lista de candidatos reducida

- Se recorre el volumen en dirección transversal (en dirección z)

Fase 1 Se almacena en una lista aquellos voxeles anotados en cada elemento de la línea recorrida en z de las fases anteriores

Fase 2 Sobre cada elemento de la lista de candidatos se actua de la siguiente manera:

- Inicio: Se extraen tres puntos de la lista p_1, p_2 y p_3
- Se traza un plano que pasa por el punto medio de $p_1 - p_2$ y es perpendicular a ellos y otro plano que pasa por el punto medio de $p_2 - p_3$ y es perpendicular a ellos
- Si los planos trazados se cruzan antes de la línea recorrida, el punto p_2 deja de ser candidato
- En caso contrario se extrae un nuevo punto de la lista de candidatos y se itera hasta que no queden puntos por procesar

Fase 3 Por último, para cada elemento de la línea recorrida se calcula la distancia mínima a todos los elementos presentes en la lista de candidatos reducida

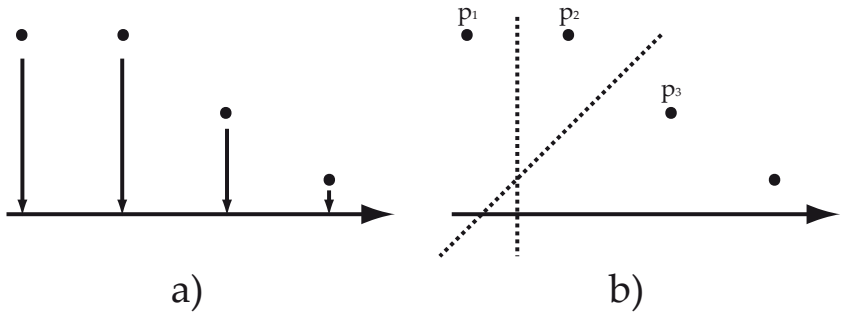

Figura 4.5: *En a) se muestra la Fase 1 del algoritmo donde se almacenan los puntos más cercanos. En b) se muestra la Fase 2 donde se descartará el punto p_2 al cruzarse las rectas trazadas antes de la línea de escaneado actual.*

La propiedades descritas en [Maurer et al., 2003] demuestran que este proceso da como resultado la DT exacta con un coste computacional lineal con el número de elementos. Sin embargo, el tiempo empleado por este algoritmo es mayor que el desarrollado en GPU y los resultados no difieren en exceso como se puede ver en los experimentos de la sección 5.3.

4.5.3. Comparación

La comparación de volúmenes utilizada para este proyecto deberá ser capaz de dar información sobre las deformaciones localizadas y decidir si la deformación calculada es acorde o no con la que se ha escaneado en la realidad. De esta manera se podrá decidir si el modelo biomecánico aplicado al tejido ha sido correcto.

Utilizando las herramientas que se han descrito (coeficientes clásicos, medidas con distancias y transformada de distancias) se podrá conseguir un valor de similaridad entre dos volúmenes que indique el tipo de error cometido por el modelo biomecánico en cuestión.

Para calcular la distancia $d(i)$, se lanza el algoritmo de transformada de distancias en GPU (sección 4.5.1) sobre los dos volúmenes que queremos comparar y, seguidamente, se analiza cada elemento de ambas matrices de distancias. Además del estudio de los coeficientes de Jaccard, Dice y Hausdorff, que tienen la misma forma para 3D, se han definido dos nuevas distancias para cada elemento escaneado i de cada conjunto X e Y: la distancia sin signo ($d_{unsigned}$) que se muestra en la ecuación 4.13 y la distancia con signo (d_{signed}) de la ecuación 4.14.

$$d_{unsigned}(i) = \begin{cases} 0, & i \in X \cap Y \\ +d(x), & i \in Y \backslash X \\ +d(y), & i \in X \backslash Y \\ 0, & i \notin X \cup Y \end{cases} \tag{4.13}$$

$$d_{signed}(i) = \begin{cases} 0, & i \in X \cap Y \\ +d(x), & i \in Y \backslash X \\ -d(y), & i \in X \backslash Y \\ 0, & i \notin X \cup Y \end{cases} \tag{4.14}$$

La media de estas distancias y su desviación típica ofrece información adicional muy útil para la comparación de volúmenes. Por ello, son añadidas a la lista de coeficientes utilizados para la comparación.

Este procedimiento lleva a la media de distancias (con y sin signo) y las desviaciones típicas de X e Y. A partir de los valores de las desviaciones típicas se podrá obtener la siguiente información:

- Si el valor de Hausdorff es cercano a la media sin signo: la figura ha sido deformada coherentemente, sin deformaciones localizadas en sitios concretos.
- Si Hausdorff es lejano a la media sin signo: ha habido una deformación alta en alguna zona que ha hecho que las distancias ahí hayan sido mayores. Lo mismo pasa si hay grandes distancias respecto a la media.

En caso de tener una deformación coherente (desviación típica baja), el siguiente dato a considerar será la media de las distancias. Si el valor de la media es bajo, la deformación ha sido muy buena, en caso contrario habrá sido deformado por encima o por debajo de lo esperado.

En última instancia, podemos añadir información sobre la comparación con las medidas clásicas como Jaccard, Dice y Hausdorff que complementarán las medidas anteriores. La inclusión de estas medidas permitirán que también se considere la información global respecto al número de elementos mal clasificados.

4.6. Experimentos y resultados

Para validar el comparador de volúmenes descrito se han hecho distintos experimentos que permiten estudiar el comportamiento de los coeficientes estudiados.

El primer experimento compara dos cilindros, del mismo tamaño y forma, pero rotados sobre su propio eje. La unión de ambos volúmenes en dos rotaciones diferentes se puede ver en la figura 4.6.

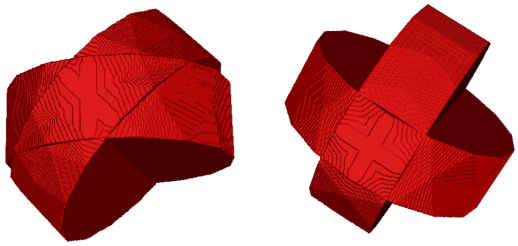

Figura 4.6: *Comparando un cilindro con otro rotado 45° y 90°.*

Los valores obtenidos en este experimento se muestran en la figura 4.7. Esta figura muestra como, cuando la discrepancia entre ambos volúmenes es mayor (90°), los valores de los coeficientes incrementan o decrementan, según el coeficiente mida similaridad o diferencia. Los coeficientes de Jaccard y Dice representan el nivel de solape entre los dos volúmenes, dando los valores más bajos en el centro de la gráfica (90°). Esta curva también revela que el coeficiente de Dice discrimina peor que el coeficiente de Jaccard ya que la disminución del coeficiente de Dice es más lenta que la que muestra Jaccard. Esto significa que el coeficiente de Jaccard indica antes la diferencia entre los volúmenes de los cilindros.

Por otra parte, la distancia Hausdorff discrimina muy rápidamente, pero luego se estabiliza; esto sucede porque, en el experimento de rotación, las distancias máximas incrementan muy rápidamente en el intervalo [0°,45°] y más lentamente en el intervalo [45°,90°].

Un tercer análisis puede ser realizado utilizando las medias y desviaciones típicas de la distancia $d(i)$ calculada con el algoritmo propuesto en la sección 4.5.1. Analizando las medias de distancias y desviaciones típicas, se puede observar un incremento más o menos constante hasta el punto de máxima divergencia.

El último análisis realizado implica el estudio de las medias de distancia con y sin signo; sin embargo, en el experimento de rotación, estas variables no añaden información ya que las medias de $d(i)$ son prácticamente idénticas para ambos volúmenes (figura 4.7 abajo izquierda).

Todas las medidas proveen información de la similaridad entre ambos volúmenes, pero cada una contribuye de forma diferente. Los valores idénticos de las dos medias implican que los conjuntos tienen formas similares y los altos valores que toma el coeficiente de Hausdorff indica que sus posiciones no son las mismas.

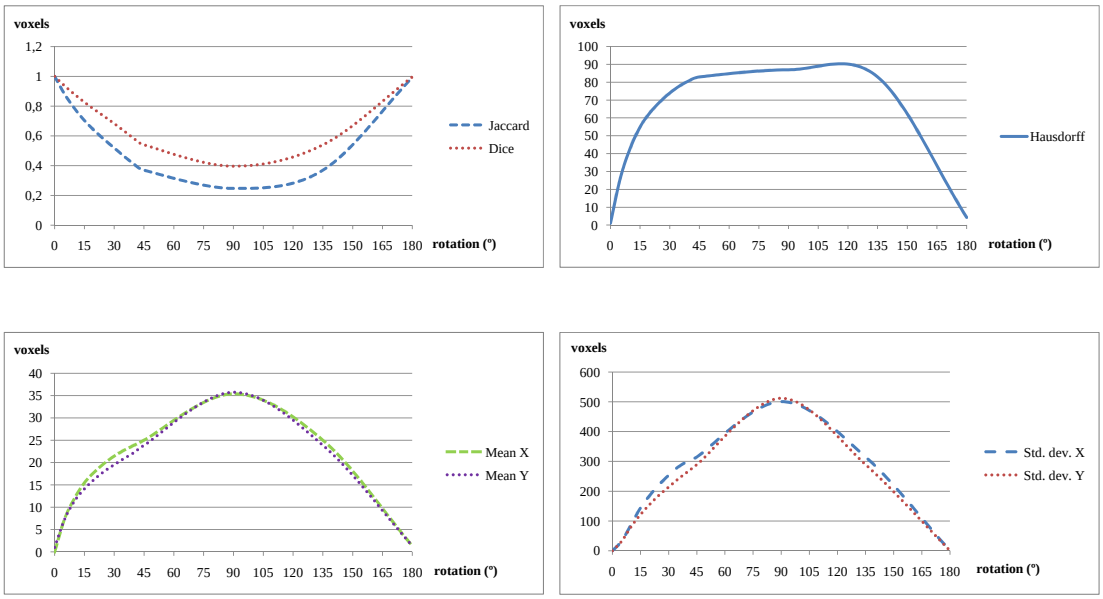

Figura 4.7: *Comparación de volumen de un cilindro rotado respecto a otro, desde 0° hasta 180°. Los coeficientes de Jaccard, Dice y Hausdorff se muestran en la parte superior. La evolución de la media de distancias y desviaciones típicas de X e Y se muestran en la parte inferior.*

El mismo cilindro se utilizó en un segundo experimento, pero esta vez variando su tamaño. La figura 4.8 confirma la diferencia entre los coeficientes de Jaccard y Dice, siendo el coeficiente de Jaccard el que más rápido crece/decrece y discrimina mejor. En cuanto a los valores de Hausdorff, se detecta un cambio brusco en los primeros escalados, con lo que es una buena medida para la discrepancia entre los dos vo-

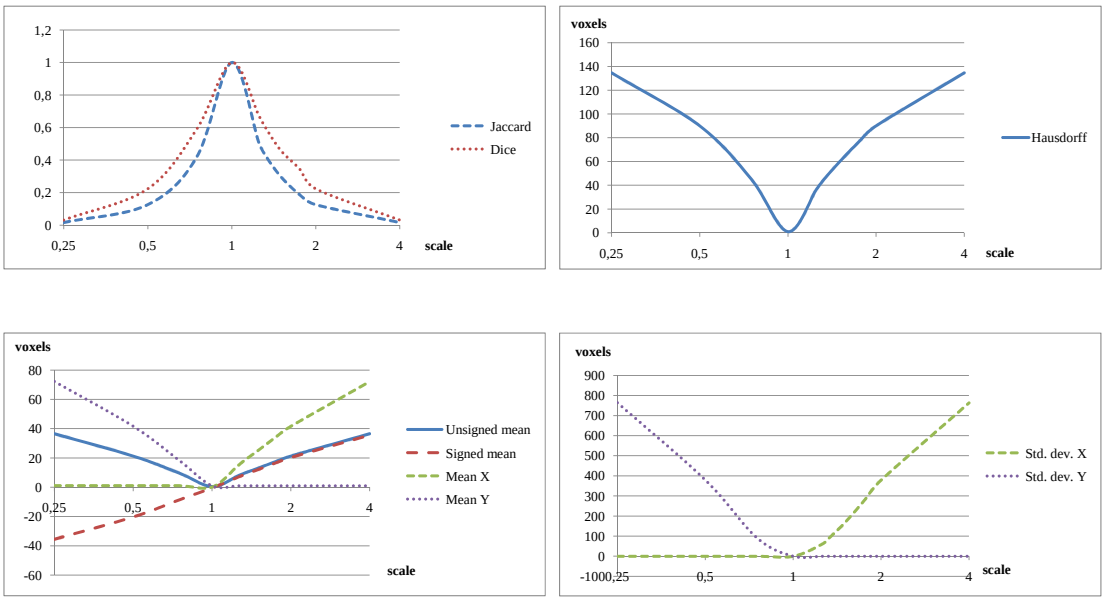

Figura 4.8: *Datos obtenidos de un cilindro escalado sobre otro, desde 0.25 a 4. Los coeficientes de Jaccard, Dice y Hausdorff se muestran en la parte superior. La comparción entre medias con signo y sin signo y desviaciones típicas de X e Y se muestran en la parte inferior.*

lúmenes. Mirando las gráficas inferiores, se puede ver cómo la media de distancias con signos es capaz de discriminar cuando $Y \subset X$ (dando valores negativos) y cuando $X \subset Y$ (dando valores positivos). Las mismas conclusiones se pueden extraer analizando el comportamiento de medias y desviaciones típicas de $d(i)$ para X e Y. En este caso, el comportamiento de las desviaciones típicas y medias no contribuye con nueva información con respecto a la media con signo. Estas gráficas llevan a la conclusión de que la media de distancias con signo nos permite diferenciar cuándo un volumen está incluído dentro de otro.

4.7. Conclusión

En esta sección se ha demostrado que la comparación de volúmenes no es un método trivial y requiere un estudio profundo de diversas técnicas para la obtención de una respuesta adecuada. Además, la implementación en GPU es capaz de calcular la transformada de distancia en un tiempo muy inferior que el algoritmo en CPU.

Del conjunto de parámetros obtenidos, se ha demostrado que los coeficientes de Jaccard y Dice dan información genérica del solape de los volúmenes, sin importar su forma y posición, siendo el coeficiente de Jaccard el que mejor diferencia. Sin embargo, para dar una mejor información del ajuste entre los volúmenes, debemos añadir las medidas basadas en distancias, como el coeficiente de Hausdorff. Además, los com-

portamientos de medias y desviaciones típicas nos permiten extender la información para determinar si un volumen está contenido dentro de otro. Ambos experimentos validan el algoritmo implementado y muestran la evolución de las medidas respecto a la rotación y el escalado.

Capítulo 5

Aplicación a un caso real

Para comprobar el correcto funcionamiento del comparador de volúmenes implementado, se han realizado dos experimentos con muestras de hígado de cordero. El primero de ellos trata de evaluar la deformación sufrida cuando se aplica la técnica del pneumoperitoneo. Esta técnica se utiliza en cirugía laparoscópica abdominal y sirve para crear un espacio en el abdomen para que el cirujano pueda operar con mayor facilidad, esto se hace inyectando CO_2 en la cavidad abdominal, con lo que los órganos internos reciben cierta presión. En el segundo experimento se realiza una deformación controlada del hígado y se comprueba si los parámetros del modelo que se simula coinciden con la deformación sufrida en la realidad.

En ambos casos se ha utilizado hígado de cordero por ser fácil de adquirir y de manipular. Además sus propiedades biomecánicas son fácilmente extraíbles de la literatura [Shi et al., 2008].

5.1. Introducción

La modelación biomecánica del hígado permitirá simular las deformaciones sufridas por parte del órgano durante las operaciones de laparoscopia. Estas deformaciones pueden ser causadas por la respiración del paciente, por el pneumoperitoneo y, en general, por cualquier manipulación que haga el cirujano sobre el hígado.

Cuando se trata de pequeñas deformaciones se puede asumir que el comportamiento del hígado de cordero es elástico lineal y las propiedades de este modelo las definen los parámetros $\nu \approx 0{,}45$ y $E = 11050$ Pa [Shi et al., 2008].

5.2. Simulación del pneumoperitoneo

En primer lugar se han aplicado las herramientas desarrolladas para comparar la deformación sufrida por el hígado cuando se aplica la técnica del pneumoperitoneo. Esta técnica se utiliza en cirujía de mínima invasión como la laparoscopia y consiste en insuflar un gas en la cavidad abdominal para crear un espacio donde el cirujano pueda trabajar sin dañar los tejidos internos. Este gas comprime los órganos internos ligeramente.

Se obtuvieron dos muestras cilíndricas de 80mm $\varnothing$ de un hígado de cordero. Estas muestras se colocaron en un bote de cristal herméticamente sellado y conectado a un tubo por el que se le insufló gas CO_2. El dispositivo utilizado para inyectar el CO_2 fue un Wolf IP20 (figura 5.1 izquierda). La presión se mantuvo constante durante todo el procedimiento. Los valores de presión probados se encontraban en el rango 10-13 mmHg, el más común en las intervenciones abdominales.

Figura 5.1: *Experimento para la simulación de la técnica del pneumoperitoneo. A la izquierda, el dispositivo que insufla gas de forma controlada. A la derecha, el bote con la muestra de hígado de cordero dentro, el cual es escaneado en una TC y en el que se inyecta el gas.*

El bote se introdujo en un multi-detector CT GE LightSpeed VCT - 5124069 de la sala de TC del Hospital Clínica Benidorm, en Alicante. Las imágenes de tomografía computerizada fueron adquiridas para cada muestra con un intervalo axial entre cortes de 0.625 mm antes y después de aplicar el CO_2.

Las imágenes fueron procesadas utilizando el software ScanIP v4.0 de Simpleware para segmentar el hígado del bote y filtrar. Una vez obtenido el volumen reconstruído en forma de voxels se procedió a la comparación de la muestra original y la muestra obtenida bajo la presión del gas.

De esta manera se pudo comprobar que si el hígado sufrió una ligera compresión cuando se le insufló el gas [Martínez-Martínez et al., 2010].

A este experimento se le aplicaron las técnicas de comparación definidas en el capítulo anterior. Los resultados obtenidos se pueden observar en la tabla 5.1. Estos valores prueban que los dos volúmenes tienen una forma similar, dando los valores de la media de distancias con signo ≈ 0. Sin embargo, aunque sus valores son similares, los coeficientes de Jaccard y Dice muestran una diferencia al tener valores diferentes a 1. Los voxels de la resta de ambos volúmenes pueden verse en la figura 5.2. Los voxels rojos representan los voxels que pertenecen al volumen X y no al Y. Los voxels verdes son aquellos que pertenecen a Y y no a X. Esta figura muestra cómo el hígado ha sido ligeramente deformado, por ello los coeficientes de Jaccard y Dice son diferentes de 1 y la distancia máxima que establece el coeficiente de Hausdorf es ≈ 4 voxels para la muestra 1 y ≈ 2 para la muestra 2.

Pieza	Jaccard Coefficient	Dice Coefficient	Hausdorff Distance	Media sin signo	Media con signo
Liver 1	0.97989	0.98985	4.4721	1.1208	0.11085
Liver 2	0.97739	0.98857	2.2360	1.0037	0.00031

Tabla 5.1: *Resultados numéricos de las medidas para el pneumoperitoneo.*

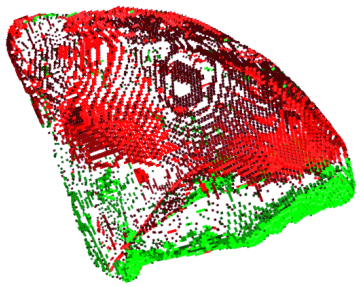

Figura 5.2: *Diferencia entre la muestra original (Liver 1: 80mm ⌀) y la deformada con 10-13 mmHg.*

5.3. Validación del modelo biomecánico elástico lineal

Una vez comprobado el correcto funcionamiento de la herramienta con los experimentos sintéticos del capítulo anterior, esta herramienta también se utilizó para llevar a cabo la validación del modelo biomecánico (elástico lineal) del hígado para pequeñas deformaciones. Para esto se diseñó un proceso experimental basado en el llevado a cabo por [Mazza et al., 2007] donde captan las propiedades de un hígado humano deformado por aspiración.

Se situó una muestra de hígado pegada sobre una base de cartón, dejándola así

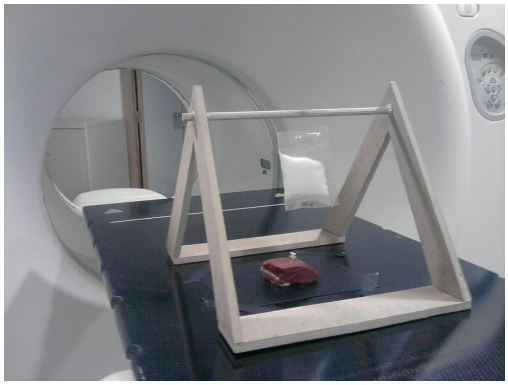

Figura 5.3: *Dispositivo que deforma controladamente el hígado. En la foto se ejerce una presión de 20g sobre la superficie del órgano.*

sujeta a la base del dispositivo de TC. Posteriormente se pegó una pieza de plástico en el centro de la muestra de hígado, midiendo su posición y tamaño. Dicha muestra se colocó en el centro de una polea de madera tal y como muestra la figura 5.3. La polea se colocó sobre la muestra y un hilo de nylon de peso despreciable se ató a la pieza de plástico y a una bolsa que hacía de contrapeso con un peso controlado de 20g y 40g. El dispositivo se colocó en la mesa de la TC descrita anteriormente en el Hospital Clínica Benidorm.

Se escanearon en tomografía computerizada los hígados deformados y sin deformar. Posteriormente se utilizó el software de FEM ANSYS para simular la misma deformación aplicada en el experimento sobre el hígado sin deformar. Los parámetros utilizados y condiciones del modelo simulado son las siguientes:

- Módulo de Young: 11055 [Shi et al., 2008]
- Módulo de Poisson: 0.4 [Shi et al., 2008]
- Tipo de elemento: Tetraedro (SOLID45)
- Número de nodos: de 16.000 a 46.000
- Número de elementos: de 73.000 a 207.000
- Condiciones de contorno de ANSYS: los nodos de la base están restingidos en las tres direcciones puesto que la base del hígado se pegó a la base de cartón
- Tipo de carga: fuerza vertical y hacia arriba de valores 20g y 40g

Una vez obtenidos ambos modelos geométricos, con la superficie mallada en triángulos, se procedió a su transformación a volumen 3D con un proceso de voxelización. Este proceso siguió los pasos descritos en el capítulo 3. De esta manera se obtuvieron dos matrices 3D que contenían voxels de tipo objeto o fondo. Estas matrices son la entrada al comparador de volúmenes descrito en el capítulo 4.

En la comparación llevada a cabo entre el órgano con la deformación simulada y el órgano con la deformación real se obtuvieron claras diferencias (tabla 5.2). Se puede apreciar como en todos los casos la deformación estaba por debajo de la real ($d_{signed} < 0$) y siempre la diferencia es mayor en las muestras de 40g respecto a las de 20g.

Distancias aproximadas					
Pieza	Jaccard Coefficient	Dice Coefficient	Hausdorff Distance	Media sin signo	Media con signo
Liver 1 20g	0.967305	0.983381	19.6214	2.22755	-1.07055
Liver 1 40g	0.952139	0.975483	22.9129	2.98701	-1.56316
Liver 2 20g	0.980384	0.990095	15.0665	1.54044	-0.51817
Liver 2 40g	0.956887	0.977968	25.7876	2.14127	-0.83978
Liver 3 20g	0.941750	0.970001	18.9737	2.05159	-0.52942
Liver 3 40g	0.925187	0.961140	28.3019	2.98872	-1.08244
Distancias exactas					
Liver 1 20g	0.967305	0.983381	21.1187	2.01346	-0.35740
Liver 1 40g	0.952139	0.975483	24.0832	2.63530	-0.46496
Liver 2 20g	0.980384	0.990095	16.6132	1.48700	-0.01870
Liver 2 40g	0.956887	0.977968	27.0924	2.35437	-0.40348
Liver 3 20g	0.941750	0.970001	20.3224	2.61484	-0.37186
Liver 3 40g	0.925187	0.961140	29.4279	3.06777	-0.68018

Tabla 5.2: *Resultados numéricos de las medidas tomadas con la deformación aplicada con* $E = 11055$

Los datos de la tabla demuestran que los parámetros del modelo no han sido correctos. La deformación simulada es menor que la real, pudiéndose apreciar bien esta diferencia en la figura 5.4. Además, al utilizar distancias exactas varían ligeramente los valores numéricos, sin embargo, esto no hace cambiar el significado de las medidas tomadas por lo que en este caso para mejorar la velocidad del proceso es más interesante utilizar distancias aproximadas.

Tanto en la visualización de las diferencias como en las medidas obtenidas se aprecia que la deformación simulada no ha sido suficiente, esto puede deberse a que la clasificación como modelo elástico del hígado para este tipo de deformaciones no sea correcta.

5.4. Conclusión

La aplicación del comparador de volúmenes desarrollado en este trabajo ha demostrado ser capaz de comparar volúmenes en casos reales. De esta manera se ha comprobado que la técnica del pneumoperitoneo ejerce una fuerza apreciable sobre el órgano aunque no llega a comprimirlo (a perder volumen). Sin embargo, en el segundo experimento, la modelización elástico-lineal no se ajusta a la realidad con los parámetros propuestos en la literatura [Shi et al., 2008]. Se puede ver que la deformación simulada resulta mucho menor a la que se produce en la realidad, por ello, si queremos obtener el comportamiento real debemos movernos a modelos viscoelásticos o hiperelásticos para el hígado, o bien considerar un módulo de Young muy inferior al obtenido por otros autores.

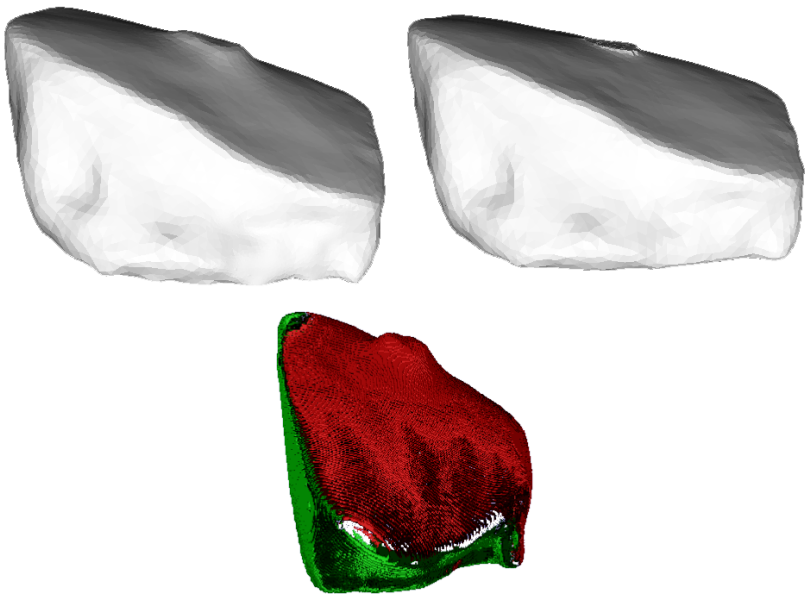

Figura 5.4: *Comparación de la muestra deformada real (Liver 1) con la deformación simulada aplicando 20 gramos. Arriba a la izquierda la muestra real escaneada, arriba a la derecha la simulación de 20 gramos sobre la muestra real escaneada sin deformar, abajo la comparación de volúmenes: voxels en rojo pertenecen a la real y no a la simulada, voxels en verde pertenecen a la simulada y no a la real.*

Capítulo 6

Conclusiones finales

En este trabajo se han explicado los pasos necesarios para transformar un modelo de elementos finitos de un objeto (en este caso, un órgano) en un volumen (voxelización) y posteriormente compararlo con otro volumen. El paso de la estructuración o transformación en volumen se ha realizado mediante programación de shaders y resulta un proceso rápido y eficiente que permite obtener en un espacio cúbico el volumen de cualquier modelo de elementos finitos, independientemente de su tamaño o complejidad.

Se han estudiado y reseñado diferentes algoritmos de comparación de segmentaciones, y seleccionado finalmente algunos de ellos para su ampliación a tres dimensiones. Además, con los coeficientes basados en distancias y la ayuda de la transformada de distancia se han complementado las primeras medidas para dar un resultado más adecuado sobre el ajuste que tienen los dos volúmenes.

Los experimentos sintéticos mostrados en la sección 4.6 demuestran el correcto funcionamiento del comparador implementado y el comportamiento de los parámetros obtenidos durante el proceso. Además se ha probado en un entorno real con el objetivo de comprobar la deformación causada al utilizar la técnica del pneumoperitoneo y en un segundo experimento para comprobar si el modelo biomecánico utilizado al simular ciertas deformaciones que sufre el hígado es el correcto.

Por otra parte, la herramienta desarrollada ha sido acelerada por la aplicación de algoritmos sobre GPU, el proceso de los tetraedros sobre CUDA, el cálculo de la intersección para voxelizar en shaders y la transformada de distancia sobre CUDA.

Esta herramienta pretende ser una ayuda en la validación de modelos biomecánicos de órganos que posteriormente serán utilizados en aplicaciones de guiado quirúrgico, al ser órganos internos su manipulación in-vivo es muy complicada y este tipo de comparación permite la validación con exploraciones no invasivas (RM o TC).

En el caso del segundo experimento, los resultados obtenidos muestran que, o bien el modelo no es el adecuado, o bien los parámetros elásticos de nuestro hígado no son

los mismos que los obtenidos por [Shi et al., 2008].

Siguiendo en la línea de este trabajo, sería posible realizar una búsqueda de aquellos parámetros del modelo biomecánico que consigan un mejor comportamiento del órgano. Para ello se podría realizar una exploración de los coeficientes definidos variando los parámetros de búsqueda y lanzando distintas simulaciones hasta conseguir que los coeficientes de la comparación de volúmenes tomen los valores apropiados ($Jaccard \approx 1$, $Dice \approx 1$, $medias \approx 0$, etc.).

Por último, un análisis de componentes principales (PCA) de las métricas aquí presentadas y de nuevas métricas interesantes, nos llevaría a quedarnos con aquellas que mejor discriminan y permiten realizar la adaptación del modelo con más información.

Bibliografía

[Aspert et al., 2002] Aspert, N., Santa-Cruz, D., and Ebrahimi, T. (2002). Mesh: Measuring errors between surfaces using the hausdorff distance. In *Proceedings of the IEEE International Conference on Multimedia and Expo*, volume I, pages 705 – 708. `http://mesh.epfl.ch`.

[Baudrier et al., 2008] Baudrier, 1., Nicolier, F., Millon, G., and Ruan, S. (2008). Binary-image comparison with local-dissimilarity quantification. *Pattern Recogn.*, 41(5):1461–1478.

[Belytschko et al., 1998] Belytschko, T., Liu, W. K., and Moran, B. (1998). *Nonlinear finite elements for continua and structures*. Wiley.

[Bhaniramka and Demange, 2002] Bhaniramka, P. and Demange, Y. (2002). OpenGL volumizer: a toolkit for high quality volume rendering of large data sets. In *VVS '02: Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*, pages 45–54, Piscataway, NJ, USA. IEEE Press.

[Borgefors, 1986] Borgefors, G. (1986). Distance transformations in digital images. *Comput. Vision Graph. Image Process.*, 34(3):344–371.

[Brouwer et al., 2001] Brouwer, I., Ustin, J., Bentley, L., Sherman, A., Dhruv, N., and Tendick, F. (2001). Measuring in vivo animal soft tissue properties for haptic modeling in surgical simulation. In *Stud Health Technol Inform*, pages 69–74. IOS Press.

[Cao et al., 2010] Cao, T.-T., Tang, K., Mohamed, A., and Tan, T.-S. (2010). Parallel banding algorithm to compute exact distance transform with the gpu. In *I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 83–90, New York, NY, USA. ACM.

[Carter et al., 2001] Carter, F. J., Frank, T. G., Davies, P. J., McLean, D., and Cuschieri, A. (2001). Measurements and modelling of the compliance of human and porcine organs. *Medical Image Analysis*, 5(4):231 – 236.

[Chua and Neumann, 2000] Chua, C. and Neumann, U. (2000). Hardware-accelerated free-form deformation. In *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 33–39, New York, NY, USA. ACM.

[Cárdenes et al., 2009] Cárdenes, R., de Luis-García, R., and Bach-Cuadra, M. (2009). A multidimensional segmentation evaluation for medical image data. *Computer Methods and Programs in Biomedicine*, 96(2):108 – 124.

[Dice, 1945] Dice, L. R. (1945). Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302.

[Dong et al., 2004] Dong, Z., Chen, W., Bao, H., Zhang, H., and Peng, Q. (2004). Real-time voxelization for complex polygonal models. In *PG '04: Proceedings of the Computer Graphics and Applications, 12th Pacific Conference*, pages 43–50, Washington, DC, USA. IEEE Computer Society.

[Fang et al., 1996] Fang, S., Huang, S., Srinivasan, R., and Raghavan, R. (1996). Deformable volume rendering by 3D texture mapping and octree encoding. In *VIS '96: Proceedings of the 7th conference on Visualization '96*, pages 73–ff., Los Alamitos, CA, USA. IEEE Computer Society Press.

[Fung, 1993] Fung, Y. C. (1993). *Biomechanics: Mechanical Properties of Living Tissues*. Hardcover, Springer Us.

[Gibson, 1996] Gibson, S. F. F. (1996). 3d chainmail: a fast algorithm for deforming volumetric objects. pages 149–154.

[GLSL, 2010] GLSL (2010). The OpenGL shading language v4.1. `http://www.opengl.org/registry/doc/GLSLangSpec.4.10.6.clean.pdf`.

[He et al., 2008] He, L., Peng, Z., Everding, B., Wang, X., Han, C. Y., Weiss, K. L., and Wee, W. G. (2008). A comparative study of deformable contour methods on medical image segmentation. *Image and Vision Computing*, 26(2):141 – 163.

[HLSL, 2011] HLSL (2011). Programming guide for HLSL. `http://msdn.microsoft.com/en-us/library/bb509635(v=vs.85).aspx`.

[Huttenlocher et al., 1993] Huttenlocher, D., Klanderman, G., and Rucklidge, W. (1993). Comparing images using the hausdorff distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15:850–863.

[Jaccard, 1901] Jaccard, P. (1901). Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bulletin del la Société Vaudoise des Sciences Naturelles*, 37:547–579.

[Kaufman and Shimony, 1986] Kaufman, A. and Shimony, E. (1986). 3D scan-conversion algorithms for voxel-based graphics. In *SI3D '86: Proceedings of the 1986 workshop on Interactive 3D graphics*, pages 45–75, New York, NY, USA. ACM.

[King et al., 2001] King, D., Wittenbrink, C. M., and Wolters, H. J. (2001). An architecture for interactive tetrahedral volume rendering. In *In Proc. IEEE/EG Workshop on Volume Graphics '01*, pages 163–180. Springer Verlag.

[Kurzion and Yagel, 1997] Kurzion, Y. and Yagel, R. (1997). Interactive space deformation with hardware assisted rendering. *IEEE COMPUTER GRAPHICS AND APPLICATIONS*, 17:66–77.

[López-Mir et al., 2011] López-Mir, F., Martínez-Martínez, F., Fuertes, J., Lago, M., Rupérez, M., Naranjo, V., and Monserrat, C. (2011). Naralap: Augmented reality system for navigation in laparoscopic surgery. Under revision.

[Martínez-Martínez et al., 2010] Martínez-Martínez, F., Rupérez, M. J., Lago, M. A., López-Mir, F., Monserrat, C., and Alcañíz, M. (2010). Pneumoperitoneum technique simulation in laparoscopic surgery on lamb liver samples and 3d reconstruction. *Medicine meets Virtual Reality (MMVR18) (Accepted)*.

[Maurer et al., 2003] Maurer, Jr., C. R., Qi, R., and Raghavan, V. (2003). A linear time algorithm for computing exact euclidean distance transforms of binary images in arbitrary dimensions. *IEEE Trans. Pattern Anal. Mach. Intell.*, 25:265–270.

[Max et al., 1990] Max, N., Hanrahan, P., and Crawfis, R. (1990). Area and volume coherence for efficient visualization of 3D scalar functions. *SIGGRAPH Comput. Graph.*, 24(5):27–33.

[Mazza et al., 2007] Mazza, E., Nava, A., Hahnloser, D., Jochum, W., and Bajka, M. (2007). The mechanical response of human liver and its relation to histology: An in vivo study. *Medical Image Analysis*, 11(6):663 – 672.

[Nava et al., 2008] Nava, A., Mazza, E., Furrer, M., Villiger, P., and Reinhart, W. (2008). In vivo mechanical characterization of human liver. *Medical Image Analysis*, 12(2):203 – 216.

[NVIDIA, 2009] NVIDIA (2009). Cuda basics. `http://developer.download.nvidia.com/CUDA/training/NVIDIA_GPU_Computing_Webinars_Introduction_to_CUDA.pdf`.

[Pichon et al., 2004] Pichon, E., Tannenbaum, A., and Kikinis, R. (2004). A statistically based flow for image segmentation. *Medical Image Analysis*, 8(3):267 – 274. Medical Image Computing and Computer-Assisted Intervention - MICCAI 2003.

[Prakash and Manohar, 1995] Prakash, C. E. and Manohar, S. (1995). Volume rendering of unstructured grids – a voxelization approach. *Computers & Graphics*, 19(5):711 – 726.

[Pratt, 1978] Pratt, W. K. (1978). *Digital image processing*. John Wiley & Sons, Inc., New York, NY, USA.

[Reed et al., 1996] Reed, D. M., Yagel, R., Law, A., Shin, P.-W., and Shareef, N. (1996). Hardware assisted volume rendering of unstructured grids by incremental slicing. In *VVS '96: Proceedings of the 1996 symposium on Volume visualization*, pages 55–ff., Piscataway, NJ, USA. IEEE Press.

[Rezk-Salama et al., 2001] Rezk-Salama, C., Scheuering, M., Soza, G., and Greiner, G. (2001). Fast volumetric deformation on general purpose hardware. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 17–24, New York, NY, USA. ACM.

[Rhee et al., 2007] Rhee, T., Lewis, J. P., Neumann, U., and Nayak, K. (2007). Soft-tissue deformation for in vivo volume animation. In *PG '07: Proceedings of the 15th Pacific Conference on Computer Graphics and Applications*, pages 435–438, Washington, DC, USA. IEEE Computer Society.

[Saito and Toriwaki, 1994] Saito, T. and Toriwaki, J.-I. (1994). New algorithms for euclidean distance transformation of an n-dimensional digitized picture with applications. *Pattern Recognition*, 27(11):1551 – 1565.

[Satherley and Jones, 2001] Satherley, R. and Jones, M. W. (2001). Vector-city vector distance transform. *Computer Vision and Image Understanding*, 82(3):238 – 254.

[Schulze et al., 2008] Schulze, F., Bühler, K., and Hadwiger, M. (2008). Interactive deformation and visualization of large volume datasets.

[Shi et al., 2008] Shi, H., Farag, A., Fahmi, R., and Chen, D. (2008). Validation of finite element models of liver tissue using micro-ct. *Biomedical Engineering, IEEE Transactions on*, 55(3):978 –984.

[Shirley and Tuchman, 1990] Shirley, P. and Tuchman, A. (1990). A polygonal approximation to direct scalar volume rendering. *SIGGRAPH Comput. Graph.*, 24(5):63–70.

[Strasters and Gerbrands, 1991] Strasters, K. C. and Gerbrands, J. J. (1991). Three-dimensional image segmentation using a split, merge and group approach. *Pattern Recogn. Lett.*, 12(5):307–325.

[Tanimoto, 1957] Tanimoto, T. (1957). Ibm internal report 17th nov. Technical report, IBM.

[Weiler and Ertl, 2001] Weiler, M. and Ertl, T. (2001). Hardware-software-balanced resampling for the interactive visualization of unstructured grids. In *Visualization, 2001. VIS '01. Proceedings*, pages 199 –558.

[Weiler et al., 2002] Weiler, M., Kraus, M., and Ertl, T. (2002). Hardware-based view-independent cell projection. In *VVS '02: Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*, pages 13–22, Piscataway, NJ, USA. IEEE Press.

[Westermann and Ertl, 1998] Westermann, R. and Ertl, T. (1998). Efficiently using graphics hardware in volume rendering applications. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 169–177, New York, NY, USA. ACM.

[Wong, 2008] Wong, T.-T. (2008). Shader programming vs cuda. `http://www.cs.ucl.ac.uk/staff/W.Langdon/cigpu2008/slides/shader-vs-cuda.pdf`.

[Yamada, 1984] Yamada, H. (1984). Complete euclidean distance transformation by parallel operation. *Proc. of 7th ICPR*, pages 69–71.

[Yasnoff et al., 1977] Yasnoff, W. A., Mui, J. K., and Bacus, J. W. (1977). Error measures for scene segmentation. *Pattern Recognition*, 9(4):217 – 231.

[Zhang et al., 2004] Zhang, H., Fritts, J. E., and Goldman, S. A. (2004). An entropy-based objective evaluation method for image segmentation. In *Proc. SPIE- Storage and Retrieval Methods and Applications for Multimedia*, pages 38–49.

[Zienkiewicz, 1994] Zienkiewicz, O. (1994). *Metodo De Elementos Finitos, volumen 1*.