UNIVERSIDAD DE TALCA
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA CIVIL EN COMPUTACIÓN

Prototipo Web para la Visualización de Imágenes Médicas

DANIEL MELLA HUERTA

Profesor Guía: RODOLFO ALLENDES

Memoria para optar al título de
Ingeniero Civil en Computación

Curicó – Chile
06, 2016

CONSTANCIA

La Dirección del Sistema de Bibliotecas a través de su encargado Biblioteca Campus Curicó certifica que el autor del siguiente trabajo de titulación ha firmado su autorización para la reproducción en forma total o parcial e ilimitada del mismo.

Curicó, 2019

Dedicado a mis padres quienes siempre me dieron su apoyo incondicional.

AGRADECIMIENTOS

Agradesco a mis padres Luis y María , a mi hermana Catalina, mi sobrina Josefina, mi amiga Andrea, mi primo Feny, a mis compañeros de Universidad, Milo, NegsTherionLoki, los Patos, Ichu, Pety, Tavi, a mis amigos de infancia, Koreano, El Jefe, Pipicho, a mi profesor Rodolfo y a todos los que formaron parte de mi proceso universitario.

RESUMEN

A grandes rasgos, es posible señalar que en este documento se explica el proceso necesario para alcanzar una visualización de imágenes en contexto médico, ya sea en 2 dimensiones o 3 dimensiones, partiendo por una breve explicación sobre los orígenes de la visualización de datos, hasta llegar al estándar actual de visualización.

Se destaca algunas técnicas y algoritmos utilizados para la visualización de datos que pueden ser aplicados para el propósito de este proyecto. En el capítulo 2, se explicará el formato DICOM, que es un estándar de comunicación de datos utilizado en medicina y que será utilizado como fuente de datos primaria para las visualizaciones deseadas.

Como propósito del proyecto se explica cómo se implementaron las visualizaciones sobre el navegador utilizando WebGL y qué componentes externos fueron utilizados para su construcción, junto con esto vendrán de la mano simples pruebas realizadas sobre el navegador para medir el rendimiento de éstas.

TABLA DE CONTENIDOS

página

ÍNDICE DE FIGURAS

página

ÍNDICE DE TABLAS

página

1. Introducción

Computación gráfica es una rama de las ciencias de la computación que utiliza los computadores para la generación de imágenes. A través de los años, estas imágenes han sido utilizadas en distintas áreas de la ciencia, en particular en el campo de la visualización científica, que tiene como principal objetivo contribuir al estudio de datos científicos.

Por ejemplo, en medicina se utilizan diferentes herramientas para la inspección de los cuerpos humanos u objetos. Entre estas herramientas podemos destacar la tomografía computarizada (TC) también conocida como escáner, resonancias magnéticas (RM), ultrasonidos (US), entre otros.

Todos estos dispositivos trabajan con un formato de archivos especial denominado DICOM (Digital Imaging and Communication in Medice), que permite un mejor manejo de la información.

Cuando la extracción de datos concluye, podemos dar paso a una visualización de éstos, donde podemos utilizar técnicas de visualización de imágenes en 2 dimensiones o técnicas más avanzadas en 3 dimensiones, como la renderización de volúmenes.

1.1. Trabajo relacionado

En la visualización de datos existen diferentes técnicas que nos ayudan a comprender su contenido de una manera más clara y fácil. En relación a ello, y como se

mencionó anteriormente, existen técnicas en 2 dimensiones y 3 dimensiones.

Dentro de las técnicas en 2 dimensiones, encontramos:

- Mapeo de Colores.
- Definición de Contorno.

Ambos tópicos tienen por objeto ayudar a la visualización de datos; por un lado, el mapeo de colores se centra en el interior de la figura geométrica, mientras que por otro lado, la definición de contorno se enfoca en el exterior de ésta.

Dentro de las técnicas en 3 dimensiones podemos destacar algunos algoritmos que ayudan a la formación de volúmenes, tales como:

- Marching Cubes
- Ray Cast
- Ray Tracing
- Shear-Warp

Por otro lado, existen algunas aplicaciones disponibles para escritorio cuyo principal objetivo es la visualización de imágenes médicas. Dentro de estas aplicaciones podemos mencionar como las populares:

- Osirix imagin Software [1].
- Seg 3D [2].
- ImagenVis3D [3].

[1]Osirix imagin http://www.osirix-viewer.com/
[2]Seg 3D http://www.sci.utah.edu/cibc-software/seg3d.html
[3]Imagen Vis3D http://www.sci.utah.edu/software/imagevis3d.html

Dentro de la investigación realizada, pude encontrar una aplicación web denominada webwiew3d[4],que tiene por función realizar visualizaciones de imágenes médicas, estas pueden ser TC, RM o US. Pero cabe mencionar, que la aplicación web no está disponible para la carga de datos nuevos o propios, debido a que sólo se encuentra en una versión beta.

1.2. Definición del problema

El principal problema es ¿dónde puedo ver mis datos médicos sin realizar instalaciones engorrosas o utilizar programas complejos? La respuesta para ello es que, teniendo presente la fase inicial de investigación, no hay una forma simple de visualizar estos datos médicos y que esté al alcance de toda persona que tenga un computador en su poder.

Para resolver este problema se realizará un prototipo web que ayude a la visualización de datos médicos, intentando velar que sea simple, sin que exista alguna instalación previa de la aplicación o que implique un procedimiento engorroso.

1.3. Hipótesis

Para el desarrollo de este proyecto se consideraron las siguientes hipótesis:

- Es posible implementar una vista en 2 dimensiones y 3 dimensiones sobre navegadores.
- Es posible visualizar datos médicos utilizando el prototipo web.

1.4. Objetivos

Los objetivos a cumplir en este proyecto son los siguientes:

[4]webwiew3d http://webview3d.arivis.com/

Objetivo general

- Desarrollar un prototipo web para la visualización de datos obtenidos en contexto médico.

Objetivos específicos

- Implementar un procesador de datos para el formato DICOM.
- Implementar una visualización de datos en 2 dimensiones.
- Implementar una visualización de datos en 3 dimensiones.
- Implementar una función de transferencias para la visualización en 2 y 3 dimensiones.

1.5. Alcances

En el desarrollo de este proyecto se consideraron las siguientes limitaciones:

- En este proyecto se espera implementar un prototipo web, en donde se pueda realizar la visualización de los datos en formato DICOM.
- En este proyecto no se creará un algoritmo que haga la visualización, sino que se utilizará algunos existentes.
- Las pruebas se harán bajo el navegador google Chrome.

1.6. Resumen de Documento

Este documento se estructura de la siguiente forma:

1.6.1. Visualización y Medicina

Este capítulo se centra en la explicación del proceso de visualización, sus comienzos y el método general que se estableció para la visualización y que se ocupa hasta la actualidad. Dentro del tópico de medicina se expone el formato de los datos describiendo el formato DICOM, cómo es el estándar y la estructura que utiliza para codificar las imágenes, y por último se realiza una revisión a la visualización

de 2 dimensiones, donde se hace una referencia al mapeo de colores y definición de contornos. Además, se explica de manera superficial algunos algoritmos utilizados en medicina para la visualización de imágenes en 3 dimensiones.

1.6.2. Diseño de sistema de visualización

En este capítulo se describen los aspectos más importantes para el desarrollo del prototipo web. Además, se explica brevemente la aplicación y cómo es su funcionamiento, el proceso que realiza el usuario para lograr ver sus imágenes y se describe la división que tiene el prototipo web (3 sub prototipos). El primer prototipo hace referencia al procesador de datos, que permite transformar el formato DICOM a uno más fácil de trabajar; el segundo prototipo está enfocado en la visualización en 2 dimensiones; y por último, el tercer prototipo se refiere a la visualización en 3 dimensiones. En estos 2 últimos prototipos se expone tanto sus funcionalidades básicas como la forma de navegación.

1.6.3. Implementación

En este capítulo se explica en detalle cada uno de los prototipos planteados en el capítulo de diseño de sistemas de visualización y se nombra algunas tecnologías y librerías utilizadas. En el caso de la visualización en 2 dimensiones se utilizó WebGL (tecnología para dibujar gráficos 3D en el navegador) y gl-matrix (librería que nos permite trabajar con matrices y vectores). Por otro lado, en el caso de la visualización en 3 dimensiones se utilizó Three.js (librería 3D Basado en WebGL) y Orbit-Controls (control de cámaras de Three.js). Además, cabe destacar que en ambos prototipos de visualización, se utilizó dat.gui (control de variables a través de una interfaz) y stats.js (monitoreo de rendimiento, memoria y cuadros por segundos).

1.6.4. Pruebas

Este capítulo se centra en 2 tópicos: uno el uso memoria y otro los cuadros por segundos. Se realiza pruebas de carga, para lo cual se utilizan set de datos de diferentes tamaños y medir el rendimiento en términos de cuadros por segundos, para ver cómo se comporta con el navegador, ya que los recursos a través de éste son más restringidos en comparación a una aplicación ejecutada en el escritorio.

1.6.5. Trabajo Futuro y Conclusiones

En este capítulo se destacan algunos aspectos que sirven para mejorar los prototipos propuestos, ya que éstos están en una fase inicial y requieren un mejor desempeño bajo el navegador. Dentro de los aspectos a mejorar se encuentra la unión de estos prototipos en una sola aplicación, que el cambio de una visualización en 2 dimensiones a una en 3 dimensiones sea lo más fluido posible, además de la inclusión de un paseador de datos DICOM que acepte conversiones de datos, por ejemplo JPG 2000,JPG-LS,JPG Extended, entre otros, y por último una mejora en la visualización de datos en 3 dimensiones que consiste en la utilización de un algoritmo renderizado de volúmenes, en donde se sugiere la utilización de Ray Cast.

2. Visualización y Medicina

2.1. Visualización

El proceso de visualización se define como la producción de imágenes a partir de datos o de alguna otra fuente primaria, lo que permite facilitar el entendimiento de éstos.

Un problema muy recurrente que se presenta cuando se trabaja con datos en bruto es que sea bastante difícil comprender en qué consisten y cómo trabajar con ellos. Es por este motivo que se realizan imágenes a partir de los datos disponibles para así facilitar su entendimiento y lograr una adecuada visualización.

El primer concepto de visualización existente se atribuye a Upon[4], quienes lograron darse cuenta que, independiente del campo donde se aplicara este proceso, siempre era necesaria una representación visual de datos que comprendía un patrón de 3 pasos:

- Filtrado: consiste en tomar los datos en bruto y realizar algún tipo de filtro que permita obtener los datos realmente importantes para la tarea que se desea hacer, intentando reducir el espacio que ocupaban los datos.
- Mapeo: permite asignar a cada uno de los datos ya filtrados una primitiva geométrica para su representación.
- Renderizado: es el último paso y consiste en una conversión de las primitivas geométricas previamente asignadas en una imagen para ser mostrada.

Luego, Haber y McNabb[5] continuaron el trabajo llegando a un modelo general de la visualización y formalizando el concepto de visualización. Este modelo define 3 procesos:

- Datos enriquecidos o mejorados: este es el proceso donde se toman los datos en bruto y se realizan las modificaciones pertinentes, con el fin de facilitar la tarea de visualización.
- Mapeo de la visualización: en esta fase se toman los datos obtenidos a partir de la fase anterior y se construyen los datos abstractos, aplicando la geometría necesaria, los colores, transparencias, entre otras propiedades. Es la función que se encarga de dar los valores apropiados a cada dato y que se denomina "función de transferencia".
- Renderizado: constituye el proceso final de una visualización en donde se incluyen los datos generados en las fases anteriores a una escena que comprende todo tipo de operación, tales como rotaciones, traslaciones, iluminación, sombreado, entre otras, y que permite facilitar la visualización.

En la figura 2.1, podemos apreciar el proceso de visualización donde se incluye cada uno de los componentes que conforman la visualización, y cómo interactúa cada una de las funciones descritas precedentemente.

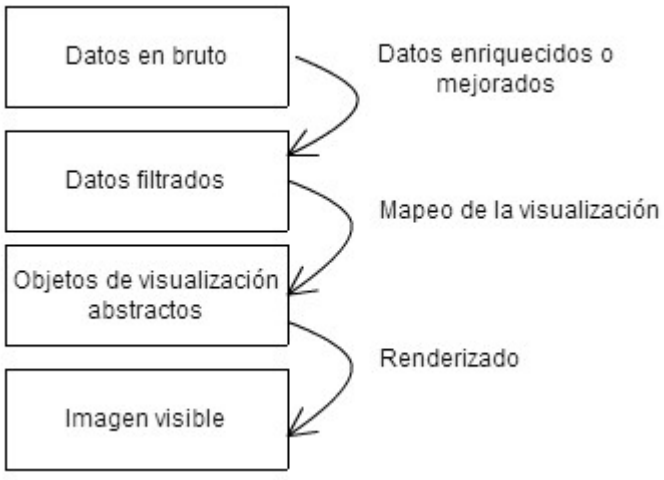

Figura 2.1: *Proceso de visualización descrito por Haber y McNabb[5].*

2.2. Formato de los datos

Si seguimos el proceso de visualización descrito por Haber y Mcnabb[**?**], nuestro primer paso es la obtención de datos en bruto. En este proyecto, estos corresponden a los datos obtenidos por TC, RM, Ultrasonidos o algunos otros dispositivos de extracción de datos. Además, cabe destacar que estos datos vienen en un formato especial de archivo denominado DICOM, el cual posee una estructura definida previamente por un estándar.

2.2.1. ¿Qué es DICOM?

Digital Imaging and Communications in Medicine o más bien conocido como DICOM es un estándar de comunicación y manejo de información de imágenes médicas y datos relevantes al contexto.

Con la introducción de las tomografías computarizadas y otros dispositivos de imágenes médicas de tipo diagnósticas en el año 1970[8], y con el incremento de aplicaciones clínicas que requerían el uso del computador, el Colegio de Radiología Americano (ACR) y la Asociación Nacional de Fabricantes Eléctricos (NEMA), necesitaban de manera urgente un estándar o método para la transferencia de imágenes y toda la información relacionada a ella. La idea era poder transferir la imagen y su información asociada de un dispositivo a otro, sin tener que modificar el tipo de formato pues hasta ese entonces cada dispositivo generaba su propio formato de imagen, dificultando, por ende, la forma de compartir la información.

A raíz de ello, ACR y NEMA conformaron un comité en el año 1983 a fin de desarrollar un estándar, tomando como base para su creación los siguientes puntos:

- Promover la interoperabilidad entre los diferentes dispositivos de imágenes médicas que existen.
- Facilitar el desarrollo, la expansión de archivos de imágenes y sistemas de comunicación (PACS [1]), esto también permitiendo la interacción con otros sistemas presentes en el hospital.

[1] El acrónimo PACS es un sistema computarizado para el archivado digital de imágenes médicas.

- Permitir la creación de bases de datos con información de tipo diagnóstica, con el objetivo de que éstas bases de datos puedan ser consultadas por una amplia variedad de dispositivos y que, a la vez, estén distribuidos geográficamente.

Luego de pasar por una serie de publicaciones, se llegó a la versión del estándar ACR-NEMA, en donde se puede encontrar especificaciones del hardware, conjunto mínimo de comandos de software y el formato de los datos de una manera coherente.

2.2.2. Estándar DICOM

El estándar DICOM tiene la mayor parte de los componentes del estándar ACR-NEMA, y además incluye algunas mejoras con nuevos componentes.

El estándar DICOM es complejo debido a la gran cantidad de contenido que engloba. Por ejemplo, dentro del estándar podemos encontrar la definición de objetos, clases de servicio, estructuras y codificación, comunicación en las redes, formas de almacenamiento y muchos otros tópicos que pueden ser consultados en la página de DICOM[2].

Dada la naturaleza de este proyecto, no es necesario abarcar todo el contenido expuesto por el estándar, por lo tanto, sólo me enfocaré en estructuras y codificación.

A continuación, se describe la estructura e información dentro de un archivo DICOM y cómo se codifica. Esto significa una parte importante, ya que colabora a la lectura y clasificación de los datos.

2.2.3. Estructura y Codificación

La idea o filosofía del estándar DICOM es almacenar la información obtenida del mundo real. Dentro de esta información podemos destacar: datos los pacientes, médicos, estudios, entre otros.

[2]http://dicom.nema.org/standard.html

Por otro lado, este estándar pretende englobar toda la información capturada en uno o varios objetos denominados Information Object Definition (IOD).

Como se puede apreciar en la figura 2.2, se aplica la filosofía antes mencionada. Un paciente llega con alguna complicación o problema al hospital y se captura toda la información concerniente y necesaria de él.

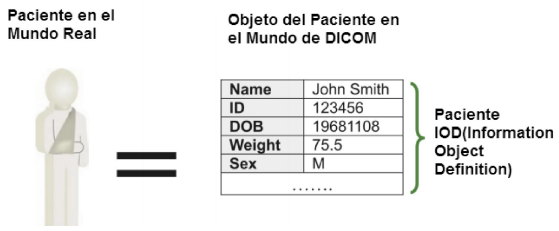

Figura 2.2: *Definición de un IOD. En el ejemplo podemos ver Nombre, Id, F. nacimiento, peso y sexo.*

El estándar DICOM define 4 niveles de objetos, que son: paciente, estudio, series e imágen. Cada uno de estos objetos contiene una cantidad definida, pero a la vez variable de sub-niveles. En este orden de cosas y como se puede observar en la figura 2.3, el paciente se encuentra en la cabeza del diagrama y es quien necesita atención médica.Al paciente se le pueden realizar múltiples estudios, los que a su vez pueden incluir una o más series de imágenes. Cada uno de estos niveles jerárquicos tiene un identificador que permite realizar búsquedas de datos, recuperaciones y transacciones de una manera mucho más sencilla.

2.2.4. Formato del Archivo DICOM

Esta sección es de suma importancia para el desarrollo del prototipo web, ya que es necesario entender cómo se distribuyen los datos en el archivo DICOM[6]. Es por ello que se describirán cada una de las partes que componen el formato de archivo de DICOM.

En primera instancia, los archivos DICOM están codificado en Hexadecimal, formando grupos de 2 bytes cada uno y donde cada fila del archivo contiene de 8 grupos, tal como se observa en la figura 2.4.

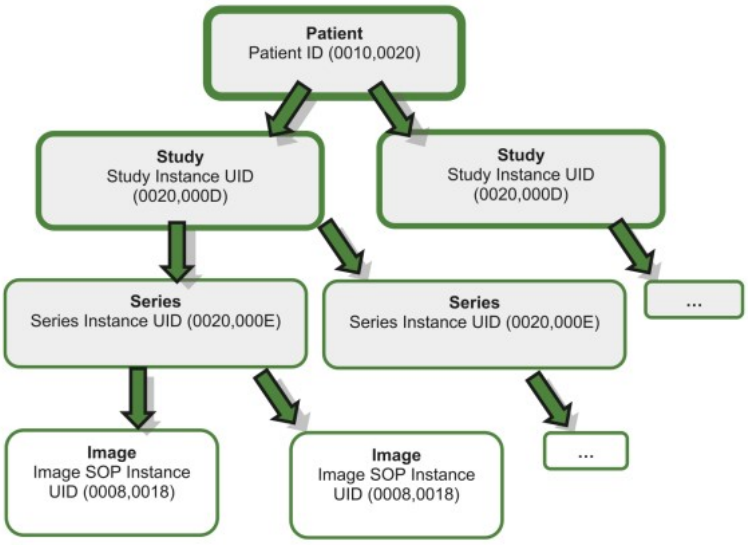

Figura 2.3: *Niveles Jerárquicos de DICOM*

```
4449 434d 0200 0000 554c 0400 c200 0000
0200 0100 4f42 0000 0200 0000 0001 0200
0200 5549 1a00 312e 322e 3834 302e 3130
```

Figura 2.4: *Grupos de Bytes en un archivo DICOM*

Todo archivo DICOM contiene un preámbulo, que se encuentra al comienzo del archivo. Este preámbulo está formado por 128 bytes de ceros. Los siguientes 4 bytes después del preámbulo contienen el código hexadecimal 4449 434d. Si transformamos ese código a codificación ASCII, el resultado corresponde a la palabra DICM. Ello nos indica que es un archivo en formato DICOM, por lo tanto, ésta sería una forma de validar si el archivo que estamos leyendo está en el formato indicado o no. La figura 2.5 muestra el préambulo de un archivo DICOM.

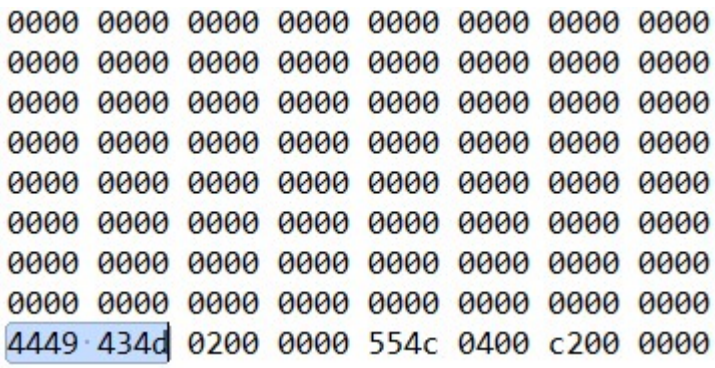

Figura 2.5: *Preambulo de un archivo DICOM*

Los siguientes bytes forman parte del contenido del archivo o más bien denominado data set, donde cada uno de estos data set está compuesto de muchos data element, en el cual cada data element contiene una estructura definida de la siguiente forma:

- Tag Element: Es un valor en hexadecimal, que utiliza 4 bytes, en donde los primeros 2 bytes son el identificador del grupo y los siguientes 2 bytes indican el número de elementos del grupo.
- Value Representation(VR): indica el tipo de datos que está almacenando. La cantidad de bytes que utiliza puede variar de 2 a 4 bytes dependiendo la situación. Además, podemos destacar que el VR puede estar presente de una forma implícita o de una forma explícita.
- Value Lenght(VL): indica la cantidad de bytes que utilizará la información almacenada en el Tag. Este puede variar de 2 a 4 bytes.
- Value: es la información que se almacena el Tag. El largo del value está delimitado por el VL, pero existen ocasiones en donde el VL es indefinido. En tal caso, se utiliza un tag especial de inicio y un tag especial de termino.

Para comprender de mejor manera la forma como se distribuye el data set y el data element, podemos observar la figura 2.6, en donde se muestra la secuencia de data element que existe en un data set y los datos que conforman el data element.

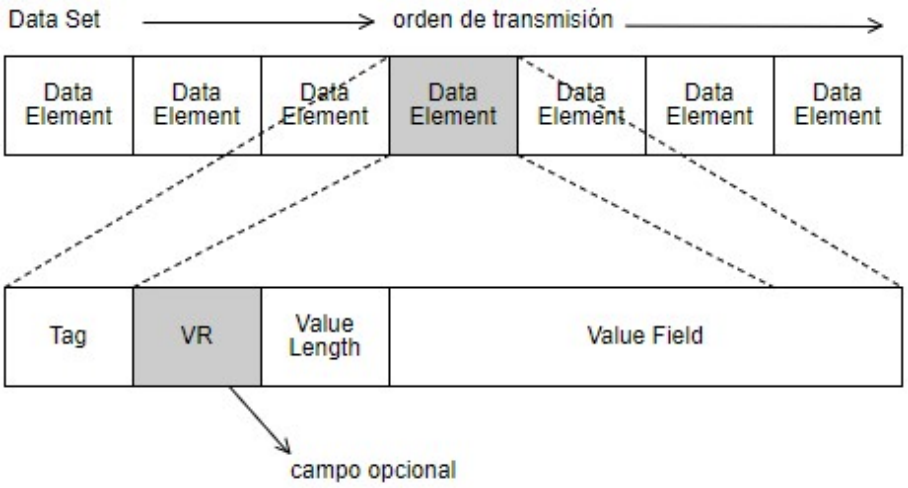

Figura 2.6: *Data set y Data Element*

2.3. Procesamiento de los datos

Cuando hablamos del procesamiento de los datos nos referimos al proceso que utilizamos para transformar nuestros datos en una representación visual que nos ayude a la comprensión de éstos. Dentro de este proceso podemos destacar 2 grupos: las visualizaciones en 2 y en 3 dimensiones.

2.3.1. Visualizaciones en 2 dimensiones

Como se mencionó anteriormente al introducir el tema sobre que trata este proyecto, podemos hablar de dos procesos: el primero es el mapeo de colores y el segundo es la definición de contornos, procesos que serán descritos brevemente a continuación.

Mapeo de Colores

Es un técnica muy básica de visualización de datos, donde la mecánica es muy simple. Se crea un mapa de pixeles, donde cada pixel tiene un color asociado, donde este color puede ser una escala de grises o en formato RGB, como se muestra en la figura[3] 2.7.

[3]Imagen extraída de http://desktop.arcgis.com/es/arcmap/10.3/manage-data/raster-and-images/colormap-function.htm

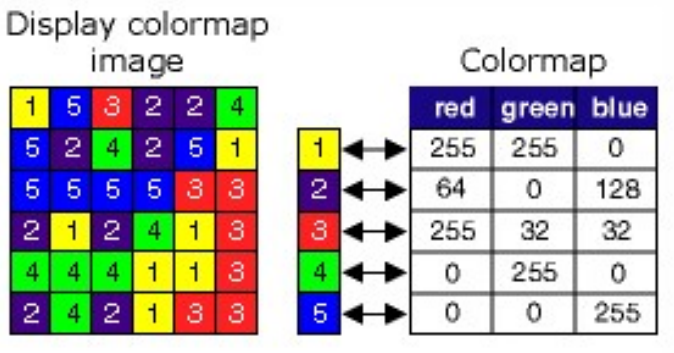

Figura 2.7: *Mapeo de Colores*

Como se observó en la figura anterior, tenemos nuestro mapa de pixeles donde cada pixel tiene un color asociado en formato RGB.

Definición de Contorno

La definición de contorno[14] se puede definir como una extensión de la imagen, dividiendo el contenido de ésta o resaltándola de algún modo. Esta definición de contorno se utiliza generalmente en mapas topológicos, mapas topográficos, mapas de temperatura, entre otros.

En la figura 2.8 podemos ver una imagen de Armenia, donde es posible destacar una escala de colores que define la altitud en cada sector del mapa. Además, podemos observar que en cada región del país Armenia está resaltado su contorno, pudiendo deducir que es el límite que separa cada región.

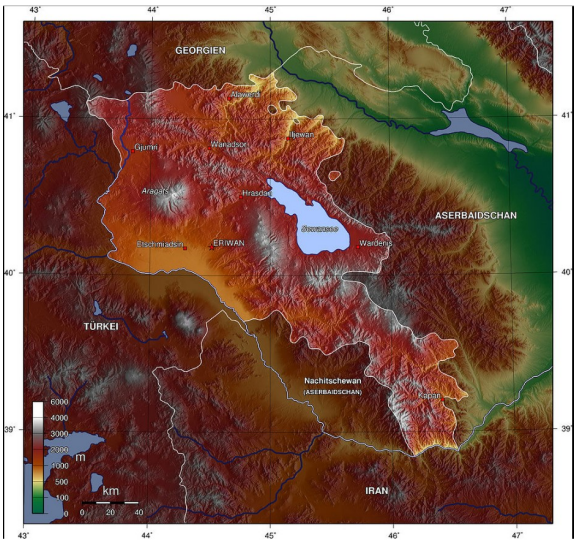

Figura 2.8: *Definición de Contorno en Mapa Topográfico*

Mapeo de Texturas

Es una técnica[3] que se utiliza esencialmente en visualizaciones en 3 dimensiones, como el nombre lo dice utiliza una textura[4] para dar forma a una superficie, y estas son representadas en 2 Dimensiones.

Para esto es necesario contar con una malla que represente al objeto en 3 dimensiones a mapear y la textura que utilizaremos. En la figura 2.9 podemos observar nuestro objeto a mapear y su textura asociada.

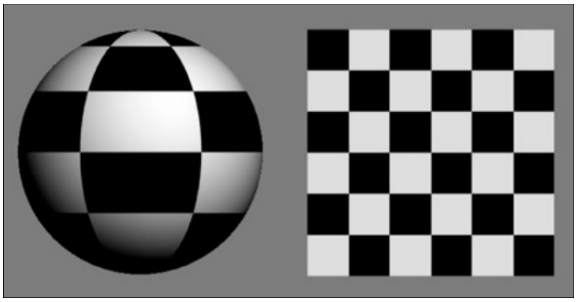

Figura 2.9: *Mapeo de Textura*

[4]Las Texturas son imágenes que representan algo en especial, como por ejemplo podemos utilizar texturas que presentan madera, piedras, piel ,etc.

2.3.2. Visualizaciones en 3 dimensiones

Dentro del conjunto de visualizaciones en 3 dimensiones existen dos grandes grupos.

El primer grupo hace referencia al renderizado de superficie, que básicamente se basa en la reconstrucción de una superficie a parte de un set de datos determinados.

El segundo grupo hace referencia al renderizado de volúmenes, que a diferencia del renderizado de superficie, se encarga de reconstruir un volumen de datos a partir de un set de datos determinados.

Para entender de mejor manera lo señalado precedentemente, observamose el cuadro 2.1 donde se aprecia un cuadro comparativo entre estos dos grupos.

Cuadro 2.1: *Tabla comparativa de Renderizado de superficie y volumen*

Superficie	**Volumen**
Requiere procesamiento, algún tipo de segmentación de los datos	No necesita procesamiento, pero en la práctica es necesario para indicar los valores de opacidad de un objeto
Solo se utiliza la información de la superficie del objeto	Utiliza toda la información
El proceso es más rápido	El proceso es más lento
Entrega menor información	Entrega mayor información
No se puede ver el interior del objeto	Permite ver al interior del objeto

Tal como se observa en la tabla 2.1, es posible percatarse a simple vista que utilizando los mismos datos, el renderizado de superficie es más veloz en comparación al renderizado de volumen, pero si se quiere profundizar en los datos, el renderizado de volumen entrega mayor información debido a que no omite datos.

A continuación, hablaremos de los diferentes algoritmos que se utilizan en el renderizado de superficie y de volumen para la visualización de imágenes.

Marching Cube

Este algoritmo se encuentra en el grupo del renderizado de superficies. Fue propuesto por Lorensen y Cline[15] en 1987 y consiste en que el algoritmo crea una matriz volumétrica que envuelve el contenido, donde cada celda de la matriz representa un cubo que avanza a través de la matriz en busca de alguna intersección con los vértices del cubo. Estas intersecciones están definidas por una configuración de 15 casos, como se muestra en la figura 2.10, en donde cada caso se dibuja una figura geométrica simple sólo si intersecta el dato en alguna de las aristas del cubo. Al final del proceso se forma la superficie final.

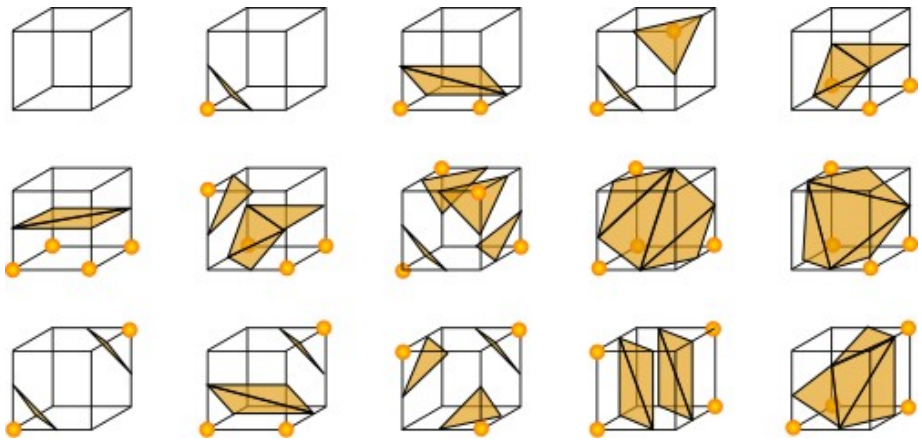

Figura 2.10: *Configuraciones del Algoritmo Marching Cube*

Una vez completado el proceso de Marching Cube podemos ver resultados como los que muestra la figura 2.11.

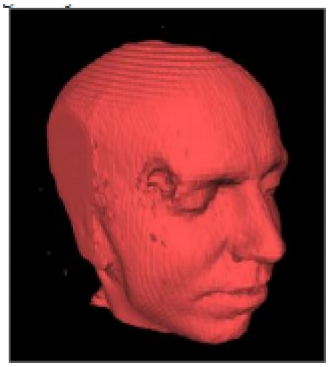

Figura 2.11: *Aplicación del algoritmo Marching Cube*

Ray Tracing

Este algoritmo se encuentra en el grupo del renderizado de superficies. Ray Tracing[11] consiste en la descripción del método de construcción de geometría sólida.Para entender cómo funciona este algoritmo, es necesario entender 3 conceptos fundamentales: el espacio de la imagen, la escena y el punto de vista.

El espacio de la imagen es la ventana de nuestro dispositivo de visión. En términos sencillos, es el campo de visión que tenemos y se describe como un plano de 2 dimensiones. La escena es un espacio en 3 dimensiones donde estará nuestro set de datos a visualizar; y por último, el punto de vista es el lugar donde nos posicionamos en la ventana para ver el contenido de la escena. La siguiente figura 2.12 nos muestra una representación visual de los descrito anteriormente.

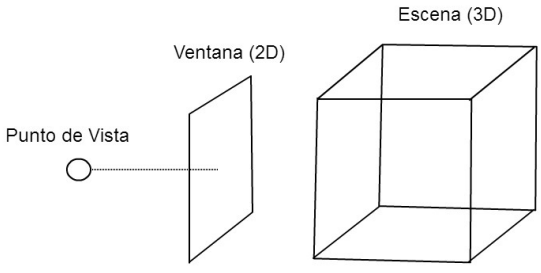

Figura 2.12: *Definición de Conceptos(ventana,escena y punto de vista)*

Ahora que conocemos estos conceptos, podemos describir qué hace este algoritmo.

Ray Tracing traza un rayo por cada pixel de la ventana en busca de una intersección en la escena. Una vez que encuentra una intersección, se calcula el valor del color dependiendo de algunos factores, como por ejemplo si existe algún modelo en la escena tales como la iluminación, reflejos, entre otros. La figura 2.13 muestra cómo se trazan los rayos en busca de intersecciones.

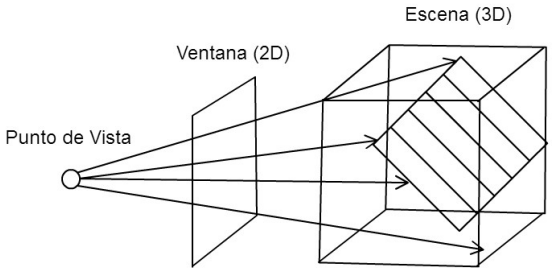

Figura 2.13: *Trazado de Rayos en Ray Tracing*

Una vez que se realiza el trazado de rayos por toda la ventana, damos por concluido el proceso del algoritmo. En la figura 2.14 que se mostrará a continuación, podemos observar el resultado final luego de aplicar Ray Tracing, donde se incluyen modelos de iluminación y reflejos, tal como se mencionó anteriormente.

Figura 2.14: Resultado Final luego de aplicar Ray Tracing

Ray Casting

Este es un algoritmo que se encuentra en el grupo del renderizado de volumen. Ray Casting[10] es muy similar a Ray Tracing, pero lo que lo hace totalmente diferente es que utiliza todo el set de datos, a diferencia de Ray Tracing que sólo utiliza

el contorno.En la figura 2.15 podemos observar que el rayo que traza Ray Casting atraviesa toda la figura y calcula el valor del color por cada intersección. A consecuencia de esto, el algoritmo es más lento.

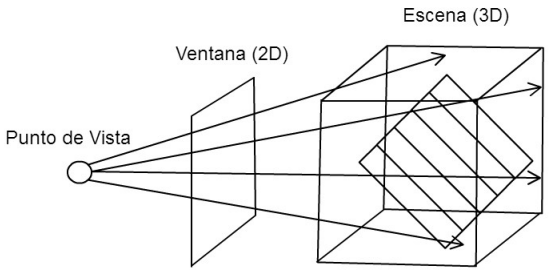

Figura 2.15: *Trazado de rayos de Ray Casting*

Dentro de las ventajas que podemos encontrar en este algoritmo es que al ocupar todo el set de datos obtenemos mayor información de la imagen, y podemos ver el interior de ésta tal como se muestra en la figura 2.16, lo que a su vez permite alcanzar un mayor realismo

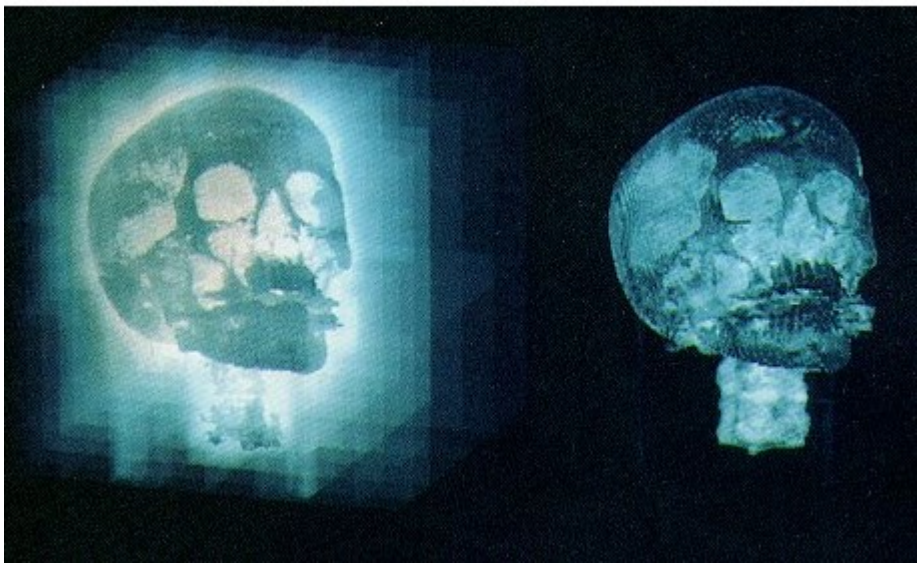

Figura 2.16: *Resultado final luego de aplicar Ray Casting*

También es importante destacar que Ray Casting es uno de los algoritmos más

utilizados en medicina, ya que permite inspeccionar el cuerpo humano en diferentes capas, ya sea tejidos, huesos, órganos, etc.

Shear-Warp

Este algoritmo se encuentra en el grupo de renderizado de volumen. Shear Warp [16] es un algoritmo veloz en comparación con Ray Casting. Este sigue la misma lógica trazando rayos a través de la escena, pero lo que lo hace más veloz radica en el proceso de intersecciones con el set de datos.

Cuando se calcula el valor de la celda, al igual que en Ray Casting, es necesario realizar interpolaciones [5] y dependiendo del tipo de interpolación que se utilice, influye en la calidad y velocidad del generado de la imagen.

En detalle, lo que ocurre tanto en Ray Casting como Shear Warp es la creación de una malla que distribuye uniformemente nuestro set de datos. Lo ideal sería que dicho set de datos fuera intersectado por un vértice o una arista, pero no siempre es así.

Un ejemplo de esto se puede observar en la figura 2.17, donde nuestra muestra de datos se representa con un círculo blanco y nuestro dato atípico con un círculo negro en el centro de la celda, por lo tanto, para calcular ese valor dentro de la celda es necesario realizar una interpolación con las aristas más cercanas.

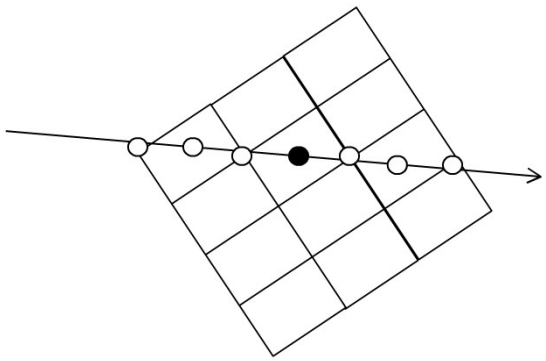

Figura 2.17: *Shear-Warp trazado de rayo.*

[5] creación de un nuevo punto a partir de puntos ya conocidos.

Por lo tanto, dado el problema que presenta ese tipo de interpolación expuesta en la figura 2.17 Sharp Warp presenta una solución a aquello la cual es convertir un pixel, que es el valor de nuestro set de datos en una arista, permitiendo que el proceso de interpolación sea más veloz.

Para lograr esto es necesario seguir los siguientes pasos:

Primero, se debe cambiar el punto de visión dejando alineado nuestro punto de vista con nuestro set de datos, como se muestra en la figura2.18.

Es importante destacar que el punto de visión queda fijo y lo que se mueve es el set de datos.

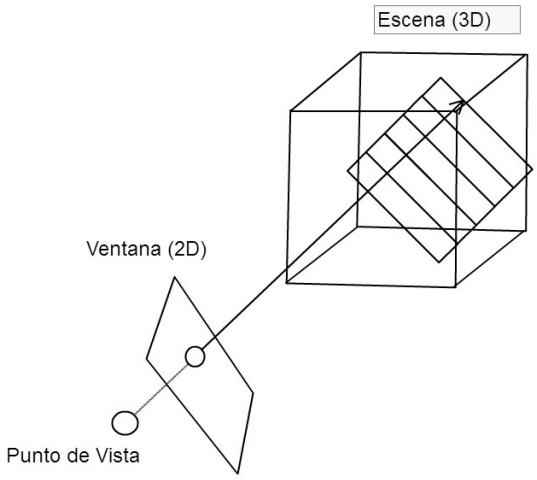

Figura 2.18: *Primer paso de Shear Warp*

Luego, el segundo paso se configura con las denominadas operaciones Shear. Estas operaciones distribuyen nuestro set de datos de la forma como se muestra en la figura 2.19, siendo la idea de esta distribución que nuestro rayo trazado sea el mismo, como si no hubiéramos cambiado nuestro punto de vista.

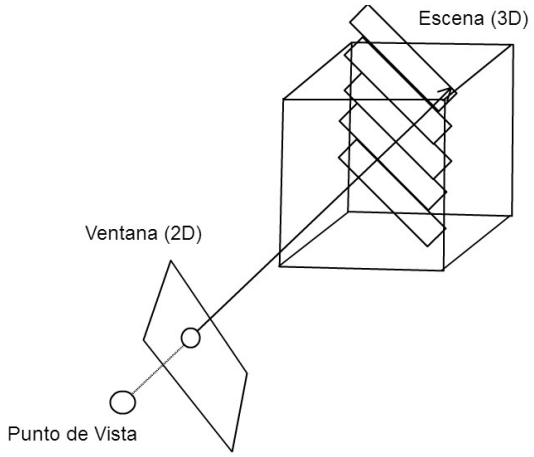

Figura 2.19: *Segundo paso de Shear Warp*

Por último, se encuentra el tercer paso denominado M-Warp. Este proceso de aplica una vez que las operaciones Shear terminan. El resultado del segundo paso genera una imagen distorsionada, por lo tanto, lo que hace este último paso es mostrar la imagen de manera correcta.

En la figura 2.20 se muestra el proceso anteriormente explicado, a nivel de implementación.

Figura 2.20: *Segundo paso de Shear Warp*

2.4. Resumen

Como se pudo apreciar, se obtuvieron conocimientos acerca del estándar DICOM y su complejidad que lo acompaña, además se pudo tener una visión superficial de

como guarda la información este estándar.

Por otro lado se recalcaron algunas técnicas y algoritmos en 2 y 3 dimensiones, describiendo su funcionamiento, que tan complejo y costosos pueden ser a la hora de implementar, además se incluyeron ejemplos para cada uno de esto, pudiendo observarlos en funcionamiento. A raíz de esto nos dio una base e ideas que pueden ser aplicadas en nuestro prototipo web.

3. Diseño de Sistema de Visualización

3.1. Descripción del problema

El problema principal que se presenta es que no existen hasta lo investigado por mí aplicaciones web que realicen una visualización de imágenes médicas, en especial con el formato DICOM. Esto, sin embargo, no quiere decir que no existan aplicaciones que realicen este trabajo, ya que hay varias aplicaciones disponibles hoy en día, como lo son principalmente OsiriX, Seg3D y 3D Slider.

Los puntos importantes de la problemática a destacar son:

- Plataformas de trabajo, la mayoría de las aplicaciones de visualización de imágenes médicas con el formato DICOM son para escritorio, siendo esta su única opción. Un ejemplo de esto es el caso de Seg3D donde solo está disponible su aplicación de escritorio. Por otro lado, encontramos a OsiriX, donde además de su aplicación de escritorio es tiene una aplicación para dispositivos móviles.
- Instalaciones engorrosas, como la mayoría de los programas son para escritorio, estos requieren de una instalación previa donde se debe configurar nuestro programa. Por ejemplo, la generalidad de aplicaciones e instalaciones para Windows no son complicadas, ya que son bastante intuitivas y no requieren mucho proceso. No obstante, ¿qué sucedería si queremos utilizar nuestra aplicación

para Linux o Mac OS X. Si nos basamos en OsiriX, no tiene una versión disponible para Linux; mientras que para Seg3D se requieren algunos componentes extra para su instalación que la dificultan aun más.

- Pagar por utilizar la aplicación: es bastante común escuchar esta situación, independiente de la aplicación de que se trate. Esto no se configura como un gran problema pero vale la pena mencionarlo, ya que para OsiriX se debe pagar; pero existen otras aplicaciones open source como Seg3D y 3D Slider.

- Forma simple de una visualización: como podemos ver, las aplicaciones mencionadas anteriormente son bastante complejas y tienen un sinfín de opciones, cuando donde tal vez lo único que deseamos es ver nuestro set de datos de imágenes DICOM de una manera fácil de entender y explorar.

3.2. Descripción de la solución

La solución para el problema expuesto anteriormente consiste en realizar prototipos web para la visualización de imágenes médicas en formato DICOM. ¿El por qué de la elección de prototipos web? La razón radica principalmente porque, independiente del sistema operativo que estemos utilizando, los navegadores web funcionan de la misma manera en las distintas plataformas existentes.

Cabe mencionar que éste sería uno de los pocos requisitos que va a tener el prototipo web para su correcto funcionamiento, específicamente tener un navegador web instalado, de preferencia Google Chrome en su última version, puesto que es ahí donde se van a realizar todas las pruebas; o en su defecto, algún otro navegador similar que soporte la etiqueta de HTML 5 `Canvas` junto con WebGL, lo cual nos permitirá trabajar con gráficos en el navegador.

Por otro lado, no necesitamos nada más que soporte lo señalado anteriormente aparte de nuestro navegador instalado, debido a que será el navegador el que se lleve todo el trabajo. Es así, que solamente se va a necesitar abrir las imágenes en nuestro navegador y el proceso quedará listo, teniendo presente, además, que la aplicación será de carácter totalmente gratuita.

En otro orden de ideas, la forma en que trabajará la aplicación será muy sencilla. Su primer paso consistirá en cargar las imágenes DICOM a nuestro navegador web para ser procesadas; luego, se podrá elegir el tipo de visualización, ya sea en 2 o 3 dimensiones. En ambas opciones de podrá interactuar con la visualización, cambiando sus colores o moviendo las imágenes.

3.3. Sobre la aplicación web

Como ya se ha señalado, para llevar a cabo la solución del problema se debe crear una aplicación web. Para ello se han definido 3 prototipos.

El primero consiste en la creación de un procesador del formato DICOM. En relación a ello, existe un estándar en donde se especifica todo lo necesario para leer de manera correcta el formato DICOM. Esto lo podemos encontrar en su página web[1], donde se puede observar una lista de archivos en formato PDF que hablan sobre el funcionamiento del estándar DICOM, pero no es necesario saber todo sobre dicho estándar, ya que lo único que nos interesa es la estructura de los archivos DICOM y eso lo podemos encontrar en `Data Structures and Encoding` y `Data Dictionary`. Uno, muestra la composición del formato y el otro, muestra el significado de todas las siglas que pueden estar presentes en un archivo DICOM.

Luego de leer los datos del archivo DICOM, es necesario clasificar el contenido y ver lo que realmente nos sirve de éste. Lo principal radica en el contenido que está presente en el `Tag 7fe0,0010`, ya que aquí se encuentra el `Pixel Data` o el contenido de la imagen médica.

Después de tener los datos o data pixel, podemos trabajar en la siguiente etapa del prototipo (parte 2) en donde se hace necesario utilizar `WebGL`. Este es un `API` desarrollado en `JavaScript` que permite trabajar gráficos en 3 dimensiones sobre navegadores sin necesidad de instalación de plugins o complementos. Esta segunda parte del prototipo consiste, entonces, en una visualización simple de las imágenes en 2 dimensiones, donde podemos ver si las imágenes se cargaron de forma correcta

[1] medical.mena.org/standart.html

o no, y nos permite inspeccionar de manera individual.

El tercer prototipo es un poco más complejo y consiste en una visualización de los datos en 3 dimensiones. La complejidad se debe a que la información necesaria para formar la visualización es mayor a la requerida en 2 dimensiones, por lo que es importante ver el rendimiento que se tendrá a la hora de utilizar esta visualización sobre el navegador.

3.4. Prototipos

En este capítulo se explicará en detalle la funcionalidad que deben tener los prototipos que se plantearon anteriormente. Esto se realiza para tener una idea clara de lo que se quiere lograr con cada prototipo.

3.4.1. La ejecución del código será en el cliente.

Como se trata de una aplicación web, se utilizará el lenguaje `JavaScript` que trabaja en el lado del cliente. Inicialmente se pensó en crear algún servicio que realice determinados trabajos, pero esta opción fue descartada debido a que los set de imágenes son muy grandes. Por ejemplo, el set de imágenes con que se trabajará en este proyecto contiene 300 imágenes, lo que da un total de 115 MB. Poniéndonos en un caso hipotético, si tenemos un internet de 1 MB de subida, nos demoraríamos alrededor de 1.9 minutos sólo en subir la información para luego ser procesada, por ende, si se trabaja directamente sobre el cliente el procesamiento de datos será mucho más veloz.

3.4.2. Prototipo N°1: Procesamiento de datos.

Procesar el archivo DICOM.

Los archivos DICOM tienen una estructura definida por su estándar, por lo tanto, es necesario organizar los datos del archivo DICOM en un formato que sea más fácil de trabajar. Es por este motivo que se pensó en un prototipo enfocado en la lectura de los datos. La idea principal es que tome el archivo DICOM como entrada y retorne una estructura de datos fácil de leer.

Estructura de datos.

Como DICOM trabaja en base a `Tags`, como se aprecia en la figura 3.1, lo ideal sería que al ingresar un `Tag` como entrada, retorne la información asociada a ese `Tag`. Para ello hay que tener en consideración que no siempre estarán todos los `Tags` presente en el archivo.

Tag	Name	Keyword	VR	VM	
(60xx,3000)	Overlay Data	OverlayData	OB or OW	1	
(60xx,4000)	*Overlay Comments*	*OverlayComments*	*LT*	*1*	*RET*
(7FE0,0008)	Float Pixel Data	FloatPixelData	OF	1	
(7FE0,0009)	Double Float Pixel Data	DoubleFloatPixelData	OD	1	
(7FE0,0010)	Pixel Data	PixelData	OB or OW	1	

Figura 3.1: Se muestra un ejemplo del tag y lo que debería almacenar.

Para esto se pensó en un objeto que se creará a medida que se van procesando los datos, por ejemplo podemos tener algo parecido a esto:

head = {
7fe0,0009 : [00 , 00 , 12, 20] ,
7fe0,0010 : [01 , 10 , 13, 24]
}

Para esto se pensó en un `Objeto` que se creará a medida que se van procesando los datos. Por ejemplo, podemos tener algo similar a lo que se mostrará a continuación, donde `head` es nuestro `Objeto` que contiene los `Tags 7fe0,0009` y `7fe0,0010` y cada `Tag` tiene su contenido asociado en decimal.

Contenido de los Tags en Decimal y no Hexadecimal

Esto se origina a partir del capítulo donde se explica el formato del archivo, pues se menciona que los valores del archivo serán en hexadecimal, por eso es extraño que estén en decimal.

Como se mencionó al principio de este capítulo, el lenguaje que se utilizará será el de `JavaScript` debido a que tiene una interfaz que nos ayuda a leer archivos denominada `FileReader`, la que nos permite leer un archivo de manera asíncrona

mediante el control de evento de `Javascript`.

Existe una propiedad muy importante que pertenece a `FileReader` llamada `Result`, que se puede utilizar una vez que se cargó completamente el archivo. Además, `FileReader` tiene 4 métodos de almacenamiento de datos los que son:

- readAsBinaryString(Blob|File): el archivo leído y almacenado en la propiedad `Result` será almacenado en una cadena binaria.

- readAsText(Blob|File, Option Encoding): el archivo leído y almacenado por la propiedad `Result` será almacenada en una cadena de texto, en formato UTF-8 por defecto. Como se puede apreciar, en el segundo parámetro de la función se puede especificar el formato.

- readAsDataURL(Blob|File): el archivo leído y almacenado en la propiedad `Result` será codificado como una URL de datos.

- readAsArrayBuffer(Blob|File): El archivo leído y almacenado en la propiedad `Result` será un objeto ArrayBuffer[2].

El contenido de los archivos debe ser manipulado con un `ArrayBuffer` ya que lo único que haremos es leer datos, por lo tanto, no realizaremos acciones de eliminado o inserción. Entonces, la única opción que nos permite trabajar con ArrayBuffer es `readAsArrayBuffer`.

Funciones

Estas son las funciones que se implementar en el parseador de datos DICOM:

[2]El objeto representado en `JavaScript` como `ArrayBuffer` no se puede manipular directamente como un arreglo común y corriente.

Función SetData

Se le indica al parseador que el set de datos se ha cambiado. Esto se hace ya que se necesitan leer más de una imágenes.

Función GetHead.

Es la encargada de entregar un `Objeto` con el contenido del encabezado, por ejemplo, tal como se había explicado en un capítulo anterior, cada dato está asociado a un `Tag` que se utilizará para identificarlo, por ende, solo con poner al `Objeto` la clave o el `Tag` correspondiente, nos entregará la información que esté asociada a él.

Función PixelData.

Esta es la función que nos devuelve un arreglo con el contenido del `Pixel Data`, el valor de cada pixel en la imagen. El peso del archivo se ve completamente reflejado en el `Pixel Data`, ya que representa alrededor del 95 % del archivo e incluyendo más.

Por ultimo

Por último, sólo se trabajará con estas funciones fuera del procesador de datos, lo que facilitará su uso puesto que sólo debemos entregarle el set de datos, y a través de las funciones `GetHead` y `getPixeldata` obtendremos el encabezado y el contenido de la imagen (Pixel Data), para posteriormente proceder a trabajar con los siguientes prototipos.

3.4.3. Prototipo N°2: Visualización en 2 dimensiones

Una de las formas básicas que se pensó para la visualización de los datos era crear una vista en 2 dimensiones, como se logra apreciar en la figura 3.2.

Figura 3.2: *Imagen sacada del set de datos de prueba, utilizando el programa Agnosco DICOM viewer.*

La idea principal es representar los datos desde el punto de vista original, como fueron capturados. En este caso y como se mostró en la figura 3.2, se puede observar una vista superior, pero también existen otras vistas comunes como la frontal y lateral.

Tipos de vistas a implementar

Como se mencionó anteriormente, es importante tener otros tipos de vista (frontal y lateral), por lo tanto, deben tenerse en consideración estas 3 formas de vistas para el prototipo número 2 y que se denomina vistas en 2 dimensiones. Para poder formar la vista Frontal y Lateral, en caso que nuestra vista principal sea la superior o vista desde arriba, se debe pensar en trazar planos perpendiculares en cada uno de los ejes de coordenadas, como se observa en la figura 3.3.

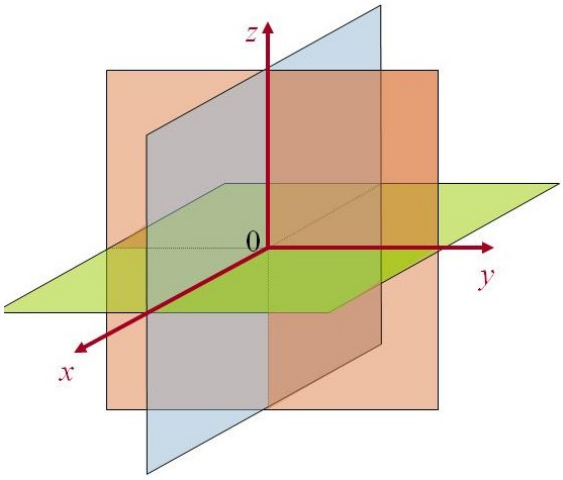

Figura 3.3: *Se muestra los planos perpendiculares que acompañan a cada eje de coordenadas.*

Por lo tanto, si pensamos en la imagen de la 3.2, podemos ver que ella pertenece al plano que está perpendicular al eje `X` que muestra la figura 3.3, por ende, a partir de los datos que nos entregan se puede generar el plano perpendicular al eje `Y` y al `Z`.

Navegación

Una vez que se tengan implementadas las 3 vistas principales, esto es, la vista superior, frontal y lateral, se debe considerar el set de datos que contiene alrededor de 300 imágenes. Por ende, se debe implementar una forma de pasar de una imagen a otra en todas las vistas.

Además, es importante tener una opción que nos permita girar la imagen en un ángulo de 90 grados tantas veces como se quiera, para el caso de que la imagen tenga una dirección un poco confusa de observar. Ejemplo de ello sería lo que muestra la figura 3.4.

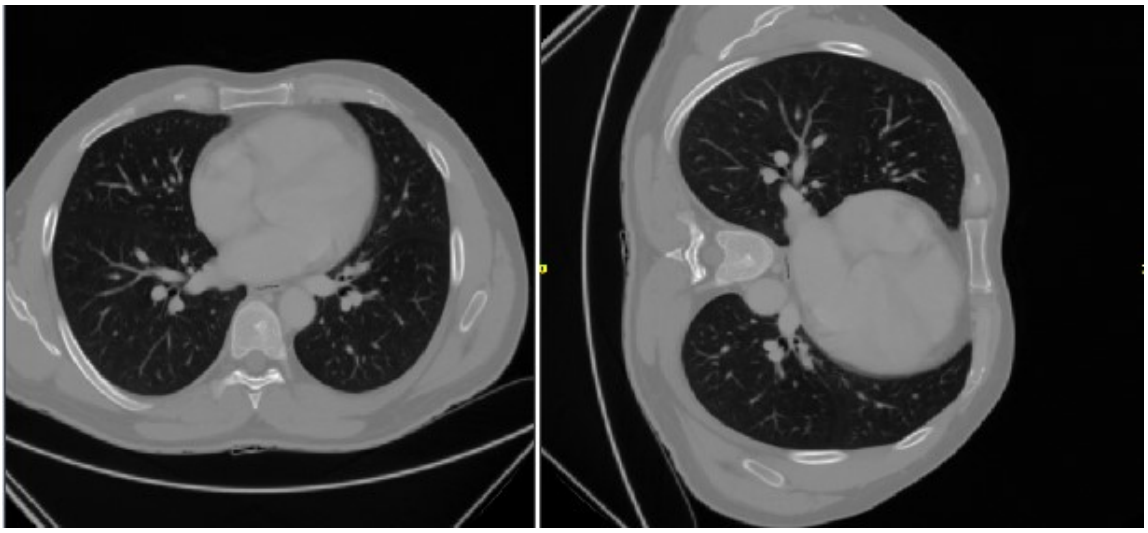

Figura 3.4: *Ejemplo de una imagen con un rotación con un ángulo de 90 grados.*

3.4.4. Prototipo N°3: Visualización en 3 dimensiones

La implementación en 3 dimensiones debe ser simple debido al tamaño del proyecto, ya que no hay tiempo suficiente para cubrir este tópico con un algoritmo complejo. Por lo tanto, lo que se pensó para resolver esta situación es posicionar nuestro set de datos en la escena en 3 dimensiones aplicando filtros simples, ya que esto suele ser muy costoso para el navegador.

Navegación

La forma de navegar es el aspecto fundamental que tiene que tener la visualización en 3 dimensiones. Lo imprescindible es que se puedan realizar rotaciones a nuestro conjunto de datos en todas las direcciones y aplicar algún tipo de `Zoom In` o `Zoom Out`, ya que si nuestra visualización en 3 dimensiones no posee estas características sería prácticamente igual que una visualización en 2 dimensiones y carecería de sentido alguno.

3.4.5. Función de Transferencia

Una función de transparencia se encarga de tomar valores de la visualización original y transformarlas en otro, formando una nueva imagen. Un proceso simple de cómo sería una función de transferencia es como se muestra en la imagen 3.5, donde

toma un valor `X` y lo transforma en un valor `Y`.

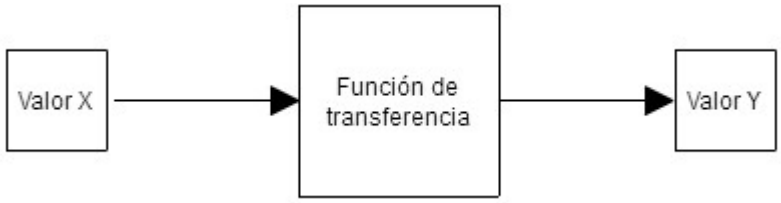

Figura 3.5: *Ejemplo simple de una función de transferencia.*

Como se vio en la imagen 3.5, no sabemos lo que realiza la función para llegar al valor `Y`, pero ésta nos permitirá realizar acciones como cambiar de color los valores de la visualización o simplemente eliminar valores que no queremos ver.

Un ejemplo de una función de transferencia aplicada a una visualización es la que se muestra en la imagen 3.6. Como se puede observar, al lado izquierdo está la imagen original en una escala de grises y al lado derecho podemos observar la misma imagen pero con otras tonalidades de colores, donde se destacan algunas secciones de la imagen original.

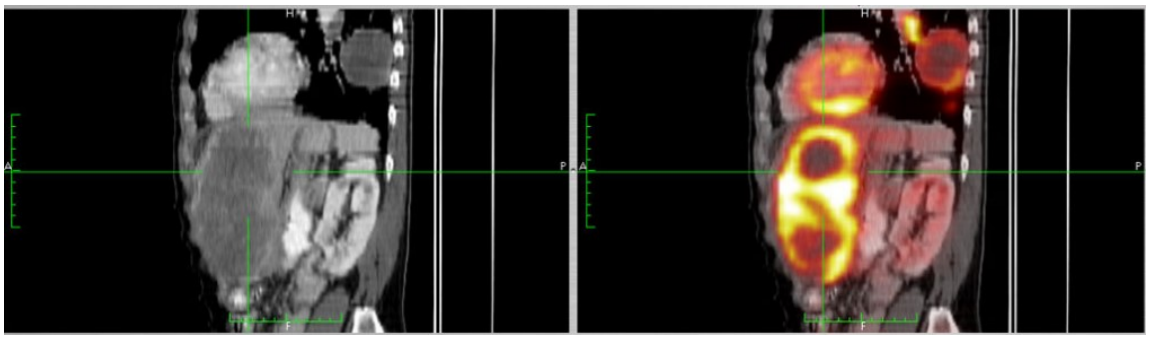

Figura 3.6: *Ejemplo simple de una función de transferencia*

Esta función de transferencia se aplicará una vez que las visualizaciones del prototipo 2 y del prototipo 3 estén completas.

3.5. Resumen

En este capítulo se pudo apreciar la motivación habida para realizar este proyecto, destacando los puntos fuertes que tendrán estos prototipos en comparación a las aplicaciones ya existentes.

Por un lado, se encuentra la gratuidad ya que permite al usuario utilizar la aplicación libremente sin restricción alguna.

Otro punto a destacar es la compatibilidad del proyecto con diversos sistemas operativos, como Windows, Linux, Mac OS X, entre otros, siendo limitado únicamente por la versión del navegador.

Siguiendo en contexto, pero no menos importante, se debe mencionar que esta será una aplicación simple, sencilla y sin procesos engorrosos.

Por otro lado, se realizó una mirada a los prototipos que se implementarán en este proyecto, destacando como el más complejo el procesador de datos DICOM, ya que sin este prototipo no se podrían realizar los siguientes 2, que son la visualización en 2 y 3 dimensiones.

4. Implementación

4.1. Introducción

Siguiendo el modelo que describió Haber y McNabb, debemos realizar una serie de procesos para llegar a una visualización de datos. Para eso, como se mencionó anteriormente, se trabajará con un set de datos de 300 imágenes en formato DICOM y se implementarán en cada componente del proceso de visualización descrito en la figura 4.12.

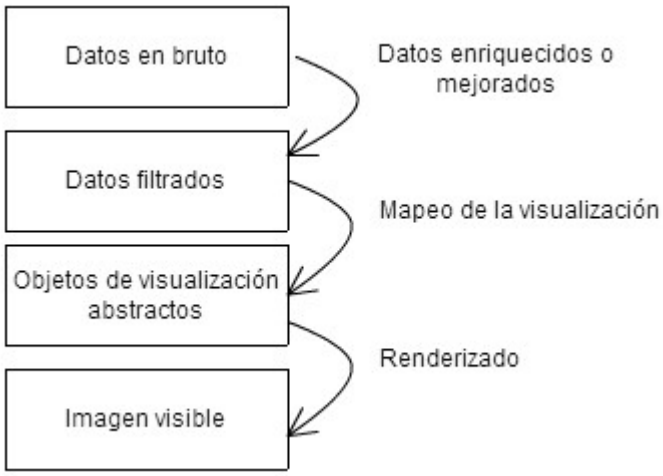

Figura 4.1: *Proceso de visualización descrito por Haber y McNabb.*

Estructuras de la Implementación

En la figura 4.2 se definió la estructura de la implementación, donde se puede apreciar una serie de pasos que tiene como fin una visualización de los datos, ya sea

en 2 dimensiones o en 3 dimensiones.

El proceso comienza en `index HTML` donde se cargan los datos desde el navegador y se comunica con el lector de archivos. Luego, se da paso al procesador de datos. Es aquí donde hacemos referencia al proceso de datos enriquecidos o mejoras descrito por Haber y McNabb.

Una vez que los datos fueron modificados, volvemos al lector de archivos y elegimos el tipo de visualización que queremos alcanzar, ya sea 2 o 3 diemensiones.

Cuando se elige el tipo de visualización pasamos al siguiente paso del proceso de visualización denominado `Mapeo de la Visualización`, donde se crea toda la geometría y colores necesarios para alcanzar la imagen deseada.

Por último, una vez que tenemos todo lo necesario, pasamos al proceso de renderizado donde se muestra la imagen en sí. Esto ocurre cuando volvemos a nuestro `index HTML`, pero esta vez cargamos todo nuestro contenido en la etiqueta `Canvas` de nuestro `HTML` que se encarga de hacer la visualización final.

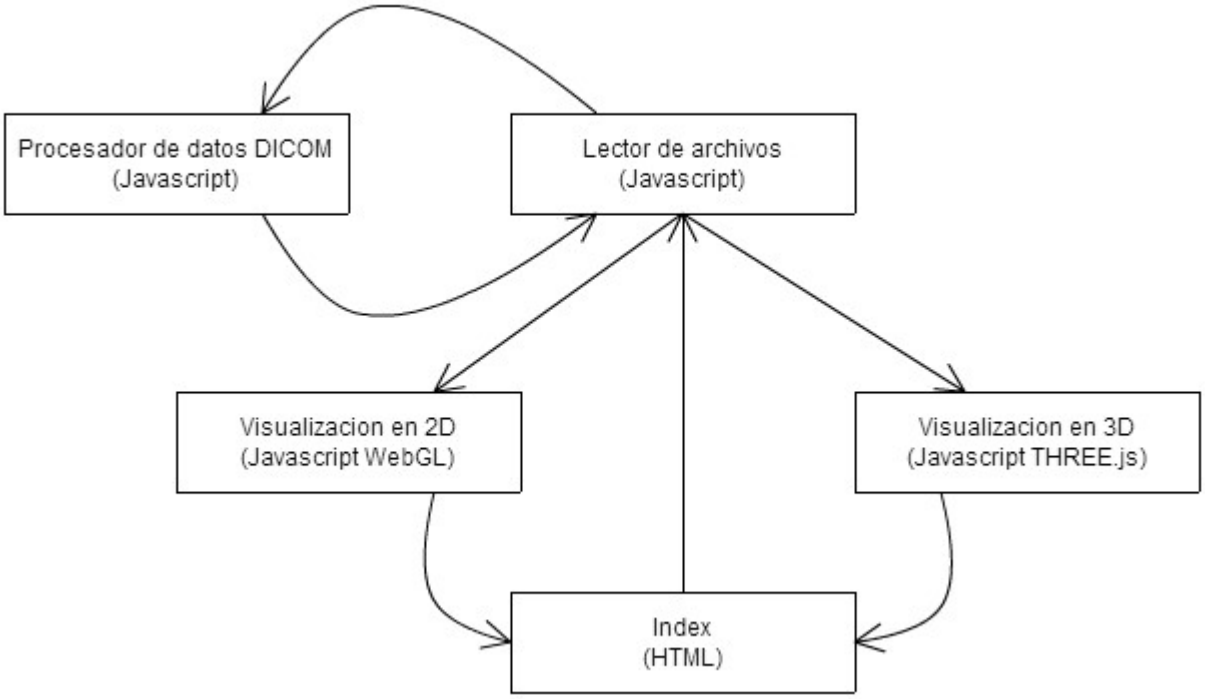

Figura 4.2: *Estructura de la implementación.*

4.1.1. Index y Lector de archivos

Ya sabemos de antemano cómo `JavaScript` trabaja con archivos y las funciones que se pueden utilizar, ya que fue descrita en el capítulo de prototipos cuando se hablaba acerca del procesador de datos, por lo tanto, nos concentraremos en explicar el proceso paso a paso; cómo fue la carga de estos datos, para así dar paso a la búsqueda de información.

4.1.2. Tags HTML necesarios.

Los `Tags` importantes de `HTML` a destacar son: el `Tag input` que tiene como `type='File'` y el `Tag Canvas` que será explicado más adelante.

Al `Tag input` le indicamos qué entrada va a recibir. En este caso será de tipo archivo, como se muestra en el código 4.1, Además, es importante destacar el atributo `multiple` que permite seleccionar múltiples archivos, tal como se observa en la

figura 4.3.

```
<input type="file" id="archivos" multiple />
```

Código 4.1: *Entrada HTML utilizada para la lectura de archivos desde el navegador.*

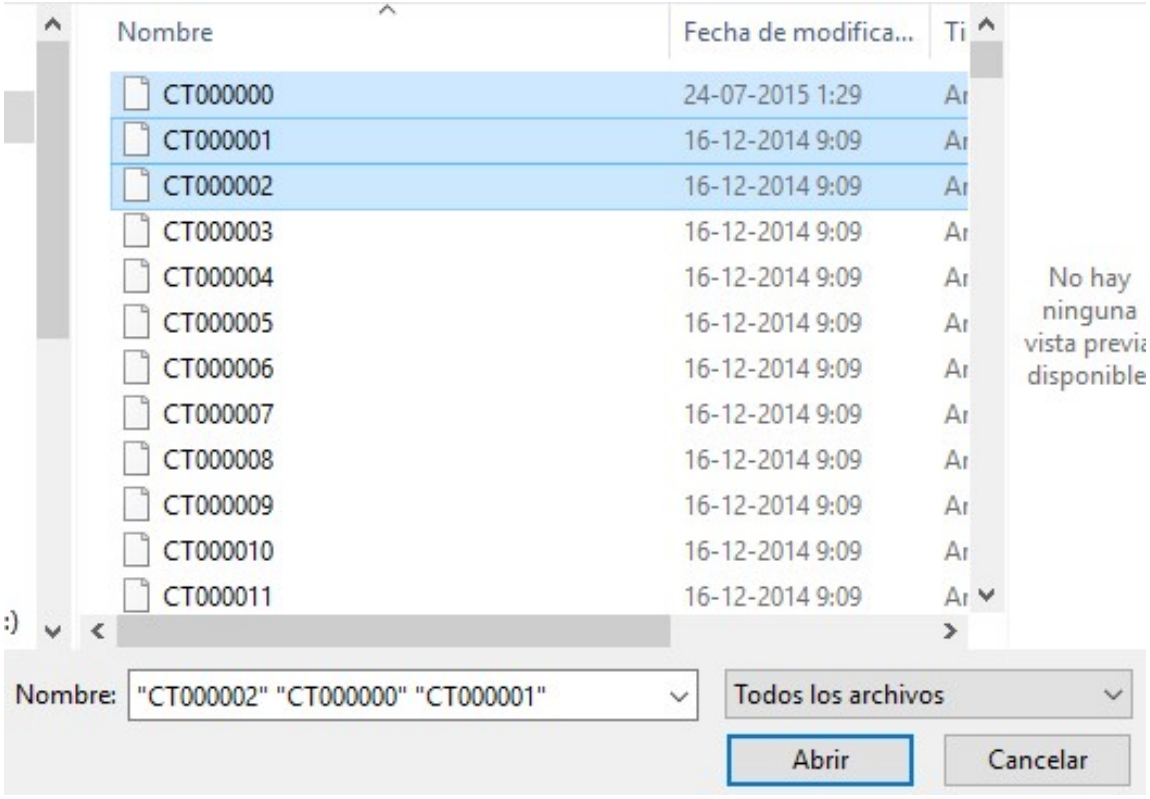

Figura 4.3: *Selección de varios archivos desde el navegador.*

4.1.3. Funciones implementadas en JavaScript.

Luego de tener nuestro elemento `HTML` capturado en una variable desde nuestro `Javascript`, es necesario agregar un `Listener` al elemento, como se ve en el código 4.2. Básicamente, lo que hace es revisar nuestro `Tag HTML` para ver si hemos realizado algún cambio, ya que en este caso nos interesa saber cuándo se subieron los archivos lo cual se indica con la función `change`. Además, se debe mencionar qué hacer luego de que ocurre esta acción.

```
elemento.addEventListener('change', change, false);
```

Código 4.2: *Se agrega un listener a un elemento HTML.*

Función Change

Cuando el contenido del `input` es cambiado, se recurre a la función `change` que se encarga de ver cuántos archivos trae el `input`, ya que como se había definido en `HTML` la variable `multiple`, se podían ingresar más de uno.

Por otro lado, es importante destacar que cada archivo se cargará por separado, por lo tanto, cada archivo se llamará a la nueva función definida como `loadBuffer`. Además, hasta este punto, los archivos todavía no son cargados pero ya se tiene información acerca de ellos.

Función loadBuffer

Es importante ir al detalle de esta función, porque se trata de cómo se implementó la carga de todos los archivos seleccionados anteriormente.

Comencemos con la descripción de los parámetros de entrada de la función:

- File: archivo entregado por el `input HTML`.
- CallBack: una vez que el archivo finalice su carga, se llamará a la función `callback`. En nuestro `loadBuffer` está definida como `callback`, pero en nuestra `JavaScript` la función que se llama es a la `AddImage`.
- Value: variable utilizada como contador para identificar en qué archivo vamos.
- Limit: hace referencia al total de archivos cargados en el input.

En esta parte entra en funcionamiento el `FileReader` descrito anteriormente en el capítulo de prototipos, por lo tanto, una vez que nuestro archivo es cargado por el `FileReader` lo transformo a un `Uint8Array`. Por último, llamamos a la función

`callback` que sería el paso siguiente luego de que el archivo ya está cargado y almacenado en un `Uint8Array`. En el código 4.3 se muestra la implementación de la función `loadBuffer`.

```
1  function loadBuffer(file, callBack,value,limit) {
2    var data;
3    reader = new FileReader();
4    reader.onload = function () {
5      data = new Uint8Array(this.result);
6      reader = null;
7      callBack(data,value,limit);
8    };
9    reader.readAsArrayBuffer(file);
10 };
```

Código 4.3: *Implementación de la función loadBuffer.*

Función callback o addImage

En la función `addImage` hacemos el vínculo entre nuestro procesador de datos y nuestro lector de archivos descritos anteriormente, ya que le enviamos la información capturada para ser organizada de una forma que nos permita trabajar más fácilmente.

Parámetros que recibe la función `addImage`:

- Data: arreglo `Uint8`.
- Value: varible utilizada como contador para identificar el fin de la lectura.
- Limit: hace referencia al total de archivos cargados en el `input`.

En el código 4.4 se muestra la implementación de la función `addImage`. Como se puede observar, tenemos un objeto denominado `DCM`; este es nuestro procesador de datos.

Cabe mencionar que el procesador de datos será descrito en detalle más adelante, pero sí es necesario saber su funcionamiento. La idea es entregar un arreglo de datos `Unit8` a través de la función `setData`, básicamente le decimos que los datos

fueron cambiados. Luego, con la función `getHead` y `getPixel` obtendremos nuestro encabezado y nuestro Pixel Data; estos son almacenados en unas variables globales denominadas `head`, que guardan todos los encabezados de las imágenes, y content que guarda todos los Pixel Data de las imágenes.

Por último, si nuestro `Limit` es igual al `Value` estamos diciendo que llegamos al último archivo, y llamamos a la función `finishLoad` que es el proceso posterior a la carga y manipulación de los datos.

```
1 function addImage(data,value,limit) {
2   DCM.setData(data);
3   head[value] = DCM.getHead();
4   content[value] = DCM.getPixelData();
5   if(value===limit) {
6     finishLoad();
7   }
8 };
```

Código 4.4: *Implementación de la función addImage.*

Función finishLoad

Cuando llegamos a esta función quiere decir que el proceso de carga de datos ha concluido, por lo tanto, podemos comenzar con nuestro mapeo de los datos, ya sea para la visualización en 2 o en 3 dimensiones.

4.2. Procesador de datos (prototipo número 1)

4.2.1. Inicializando contenido

Para dar comienzo al procesador de datos se necesita cambiar el set de datos, ya que cuando se carga por primera vez el procesador de datos está vacío (de ahí el cambio a través de la función `setData`). Una vez que realizamos este paso podemos dar comienzo a la búsqueda del encabezado y el Pixel Data. Es importante destacar que se debe llamar en orden a las funciones: primero se llama a `getHead` y luego a `getPixelData`.

4.2.2. Validando archivo

Esta es una función bastante útil que nos permite saber si el archivo cargado pertenece al formato DICOM o no. Esto se mencionó anteriormente en el capítulo de contexto cuando se hablaba del formato DICOM y donde se señalaba que éste cuenta con un preámbulo de 128 bytes. Luego, los siguientes 4 bytes nos indican el siguiente código Hexadecimal `4499 434d`, que si lo transformamos a `ASCII` nos dirá DICM, que quiere decir que pertenece al formato DICOM.

El código 4.5[1] muestra cómo se validó el archivo de datos.

```
1  validate: function () {
2    return (data[128] === 68 && data[129] === 73 &&
3        data[130] === 67 && data[131] === 77)
4        ?  true :  false;
5  }
```

Código 4.5: *Validación de un archivo DICOM.*

Una vez que nuestros datos fueron cambiados se puede consultar directamente en las posiciones sus valores, por lo tanto, los valores 68,73, 67 y 77 son los mismos valores hexadecimales de los que se habló anteriormente, pero esta vez convertidos a decimal, ya que nuestro arreglo de datos es un `Unit8`.

4.2.3. Función readTag

La función `redTag` es la encargada de leer un `Tag` del formato DICOM y retornarla en forma correcta. Es importante destacar que el largo de un `Tag` es un valor fijo de 4 bytes. Esta función recibe como parámetro el punto de partida y los datos donde se van a leer, por lo tanto, lo que hace esta función es transformar los valores a hexadecimal en el orden correcto. Esto se hace para que más adelante sea fácil la búsqueda de un `Tag` en especial, ya que la lógica sería que solo ingresando al `Tag` obtengamos el contenido de éste. En el código 4.6 se muestra cómo se implementó esta función.

```
1  function readTag(start, datos) {
```

[1]La estructura de la función de JavaScript es diferente, esto se debe a que ahora nuestro archivo JavaScript lo estamos trabajando como objeto

```
2   return (intToHex(datos[start+2])+intToHex(datos[start+1]))+","+
3       (intToHex(datos[start+4])+intToHex(datos[start+3]));
4 }
```

Código 4.6: *Implementación de la lectura de tag.*

4.2.4. Función readVR

Esta función se encarga de leer el tipo de dato que contiene el `Tag`. Para ver todos los tipos de datos que existen y su significado podemos ir al estándar DICOM[7].

El `Value Representation` o `VR` no siempre aparecerá después de un `Tag`, por lo tanto hay ocasiones que puede aparecer implícitamente.

En el código 4.7 se muestra la manera en que se implementó la lectura del `VR`. Como nuestros valores están en decimal es necesario transformarlos a ASCII a través de la función `String.fromCharCode` definida en `JavaScript`.

```
1 function readVr(start, data ) {
2   return String.fromCharCode(data[start+1]) +
3       String.fromCharCode(data[start+2]);
4 }
```

Código 4.7: Implementación de la lectura de tag

4.2.5. Función readVL

Esta función es la encargada de obtener el largo del contenido que trae un `Tag` en específico. El largo del contenido puede variar 2 o 4 bytes dependiendo del `VR`. La función que se encarga de determinar qué tipo de `VR` es será explicada más adelante, pero lo importante de la función `value length` o `VL` es que nos entrega el largo de nuestro `Tag`.

Existe 3 casos para el `VL`:

El primero es que nuestro `VL` no esté definido. Para saber esto se debe consultar 4 bytes. Si encontramos un `tag = ffff, ffff` quiere decir que no está definido el `VL`, por lo tanto, se retornará -1.

El segundo y tercer caso es que nuestro `VL` sea de 2 byte o 4 bytes dependiendo del `VR`, y se retornará el valor correspondiente al `VL`.

En el código 4.8 se muestra la implementación de cómo se obtuvo el largo.

Como sabemos que nuestros valores están en decimal, lo ideal sería convertirlo a hexadecimal y luego ver el número que presentar. Sin embargo, existe otra forma que es un poco más rápida y que es multiplicando el valor de inmediato por la potencia de 256. No se escribió el 256 elevado a la potencia de 3 debido a que es número extremadamente largo y no es probable que sea un archivo tan grande. Es un byte extra que trae el estándar pero que nunca es utilizado.

```
1  function readVl(start, data, value) {
2
3    if( data[start+1] === 255 &&
4      data[start+2] === 255 &&
5      data[start+3] === 255 &&
6      data[start+4] === 255) {
7      return -1;
8
9    } else if(value === 1) {
10     return (data[start+2]*256)+data[start+1];
11   }
12   else{
13     return (data[start+4])+(data[start+3]*65536)+
14         (data[start+2]*256)+data[start+1];
15   }
16 };
```

Código 4.8: *Implementación de la función VL.*

4.2.6. Tag Especiales

Existe una serie de Tags especiales que trae el estándar DICOM que nos ayudan a identificar cuándo el contenido de un Tag está anidado. También existen otros Tags que sirven para saber el comienzo y el fin de un contenido cuando el VL no está definido.

A continuación, se nombrarán los Tag especiales que se deben conocer para realizar una lectura correcta de los datos.

Tag 7fe00,0010

Este es el Tag más importante de todos debido a que tiene el contenido del Pixel Data, ya que este es el data que buscamos para hacer una visualización de los datos. La forma de validar si el Tag que estamos leyendo es el Pixel data se realizó de la manera como se muestra en el código 4.9

```
1 function is7fe00010(start, data) {
2   return (data[start+1]===224 &&
3       data[start+2]===127 &&
4       data[start+3]===16 &&
5       data[start+4]===0) ? true : false;
6 };
```

Código 4.9: *Implementación de la validación del Tag 7fe00,0010.*

Tag fffee,000d

Este Tag nos permite saber cuándo llegamos al final de un contenido no definido en particular. El código4.10 nos muestra cómo se valida cuando tenemos este Tag. Es importante saber, por otro lado, que este Tag hace referencia al fin de un solo contenido.

```
1 function isfffee00d(start, data) {
2   return (data[start+1]===254 &&
3       data[start+2]===255 &&
4       data[start+3]===13 &&
```

```
5       data[start+4]===224) ? true : false;
6 };
```

Código 4.10: *Implementación de la validación del tag ffee,000d.*

Tag fffe,00dd

Este `Tag` nos permite saber cuándo llegamos al final de una secuencia de contenidos no definidos. Por ejemplo, comenzamos a leer nuestros datos y tenemos una secuencia de datos respecto del cual no sabemos su fin. Cuando nos encontramos con este `Tag`, nos dice que ya se terminó de leer toda la secuencia de datos. En el código 4.11 se muestra cómo se valida cuando se tiene este `Tag`.

```
1 function isfffe0dd(start, data){
2   return  (data[start+1] === 254 &&
3     data[start+2] === 255 &&
4     data[start+3] === 221 &&
5     data[start+4] === 224) ? true : false;
6 }
```

Código 4.11: *Implementación de la validación del tag ffee,000d.*

Tag fffe,e000

Como explicamos anteriormente, ya sabemos cuándo termina el contenido de los datos pero no sabemos cuándo empieza. Es por eso que se utiliza este `Tag`, ya que nos permite saber cuándo comienza una secuencia de datos no definidos o solo un contenido no definido, toda vez que este `Tag` sirve para indicar el comienzo de los dos `Tag` anteriormente descritos. En el código 4.12 se muestra cómo se implementó esta modalidad para identificar el `Tag` de comienzo.

```
1 function isfffee000(start, data){
2   return  (data[start+1] === 254 &&
3       data[start+2] === 255 &&
4       data[start+3] === 0 &&
5       data[start+4] === 224) ? true : false;
6 };
```

Código 4.12: *Implementación de la validación del tag ffee,000d.*

4.2.7. Obteniendo encabezado

Una vez que se tiene claro cuáles son los `Tags` especiales que pertenecen al formato DICOM, es necesario iterar el archivo en busca de los `Tags` normales, los que contienen la información real del paciente.

En el pseudocódigo 4.13 se explicará la lógica que se utilizó para la búsqueda de los `Tags` del encabezado del archivo, ya que la explicación del código en detalle se tornará muy engorrosa y difícil de explicar.

Para seguir el proceso que se describirá a continuación, se puede ir a la carpeta del proyecto `lib/dcm-parser.js` en la línea del código número 94. La función encargada de realizar esta tarea se denomina `findTags`, así es como se definió el código. Además, esta función recibe como parámetros la entrada, los datos a leer, el punto de partida y el punto de término.

```
1  Mientras no llegue al Fin del Archivo Iterar
2    tag <- Leer un tag con la función readTag
3
4    Si tag es 7fe0010 (llegamos al pixel Data)
5      Terminamos la búsqueda
6
7    Si tag es fffee00d (Terminamos la secuencia de tag no definidos)
8      Terminamos la búsqueda
9
10   tipo <- Leer con la función typeVr (Verificamos el largo del
       contenido)
11     Si tipo es 1 (tiene esta estructura 4bits=tag/4bits=vr/4bits=vl)
12       vr <- Leemos con la función  readVr (tipo de dato)
13       vl <- Leemos con la función readVl(largo del contenido)
14       Si vr es "SQ"
15         Llamamos recursivamente a la función findtags con la
             posición actual más el largo del contenido
16     Si tipo es 2 (tiene esta estructura 4bits=tag/2bits=vr/2bits=vl)
```

```
17        vr <- Leemos con la función readVr (tipo de dato)
18        vl <- Leemos con la función readVl(largo del contenido)
19      Si tipo es 3 (tiene esta estructura 4bits=tag/4bits=vl)
20        vl <- Leemos con la función readVl(largo del contenido)
21
22    Leímos el contenido que almacena el tag
23  Retornamos la posición actual
```

Código 4.13: *Pseudocódigo de la función de búsqueda de tags.*

Como se pudo observar en el pseudocódigo 4.13, existen dos casos de retorno en la función. El primer caso es que encontremos el pixel data y el otro caso es que encontremos el final de una secuencia de datos no definida.

4.2.8. Obteniendo pixel data

Una vez que se encuentra el `Tag 7fe0010` perteneciente al Pixel Data debemos dar comienzo a su lectura, pero antes es necesario explicar los siguientes `Tags` que nos ayudarán a leer nuestro Pixel Data.

Tag 0028,0100

Este hace referencia a `Bits Allocated`, permitiendo saber la cantidad de bit que se usarán para el almacenamiento de cada pixel.

Tag 0028,0101

Este `Tag` hace referencia a `Bits Stored`, permitiendo saber la cantidad de bits que se utilizarán dentro de los `Bits Allocated`. Por ejemplo, si tenemos `Bits Allocated` igual a 16 bits y nuestro `Bits Stored` es de 8 bits, esto nos dice que de los 16 bits sólo utilizaremos 8 bits.

Tag 0028,0102

Este `Tag` hace referencia a `Hight Bit`, permitiendo saber la posición de nuestro bit más lejano, por lo tanto, si tenemos `Bits Allocated` igual a 16 bits y `Bits Stored` 8, nuestro `High Bit` podría ser el 7, ya que estamos considerando la posición 0.

Tags 0028,0010 y 0028,0011

Estos `Tag` hacen referencia a las filas y las columnas que tendrá nuestra imagen, por lo tanto, si nos dice que tenemos filas igual a 512 y columnas igual a 512, tenemos un total de 262144 pixeles, si no existe la compresión en los datos.

Tag 0028,0002

Este `Tag` hace referencia a la cantidad de muestras que hay por `Bits Allocated`. En este proyecto sólo se contemplará cuando la cantidad de muestras sea igual a 1 y no considerará el formato RGB, que implica que nuestra cantidad de muestras sea igual a 3.

Tag 0002,0010

Este `Tag` es de suma importancia y hace referencia al `Transfer Syntax`. Este nos dice cómo está codificado nuestro Pixel Data.

En la siguiente referencia[9] podemos ver todas las `Transfer Syntax` que existen en el estándar DICOM, pero primero cabe destacar que en este proyecto se trabajará con las siguientes `Transfer Syntax`:

Transfer Syntax UID	**Transfer Syntax name**
1.2.840.10008.1.2s	Implicit VR Endian: Default Transfer Syntax for DICOM
1.2.840.10008.1.2.1	Explicit VR Little Endian
1.2.840.10008.1.2.1.99	Deflated Explicit VR Little Endian
1.2.840.10008.1.2.2	Explicit VR Big Endian

Cuadro 4.1: *Transfer Syntax utilizadas.*

Por lo tanto, esto será a nivel de bits sin compresión, ya que para los otros tipos de `Transfer Syntax` es necesario aplicar algoritmos de descompresión donde toma mucho más tiempo su implementación, y quedará fuera del alcance del proyecto.

Ahora que tenemos identificados los `Tag`, estos nos servirán para la lectura del Pixel Data que se explicará a continuación.

Cuando estamos frente al `Tag` y leemos el tipo de variable, existen dos posibilidades: que sea de tipo `OB` (al inglés Other Byte String) o de tipo `OW` (al inglés Other Word String), teniendo una lectura diferente para cada uno.

Como habíamos mencionado anteriormente, sólo se trabajará con muestras igual a 1, ya que `OW` trabaja con 3 muestras. Además, es importante destacar que el tamaño del Pixel Data puede estar definido o no. En caso de que no esté definido, el tamaño solo se busca en el `Tag fffe,e00d` que indica el final de una secuencia.

Datos de prueba

Implica los pasos realizados para la lectura del Pixel Data. Para hacer seguimiento a estos pasos se puede ir al código fuente de la implementación en `lib/dmc-parses.js` línea 281.

Este ejemplo se realizará con los valores extraídos del set de datos de prueba, por lo tanto, cuando se hizo la lectura del encabezado, los `Tags 0028,0100, 0028,0101, 0028,0102, 0028,0002` tuvieron los siguientes valores:

- BitsAllocated = 16 bits
- BitsStored = 12 bits;
- HightBit =11;
- Samples = 1;

Esto nos dice que debemos leer cada muestra del Pixel Data cada 16 bits, donde en los primeros 8 bits se encuentra un valor de la muestra y como posición final el 11. Como se puede observar en la figura 4.4, se aprecia lo descrito anteriormente resaltando el largo de cada muestra.

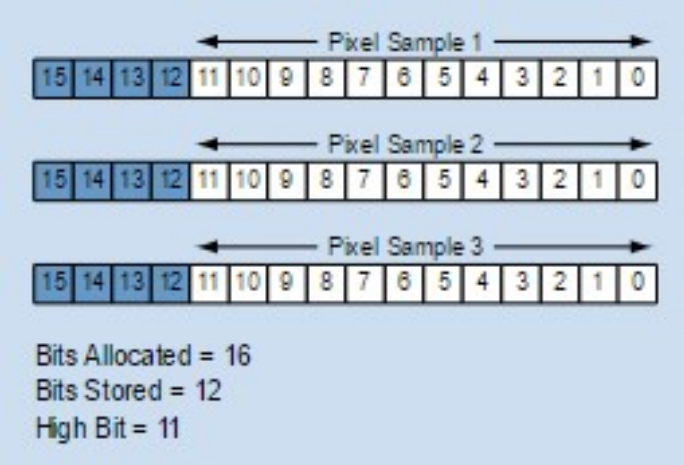

Figura 4.4: *Muestreo del pixel data.*

Manipulación de bits

`Javascript` permite realizar operaciones binarias que es lo que necesitamos para saber el valor real de la muestra. Si se quieren ver todas las funciones que `Javascript` maneja a nivel de bits se puede ir al siguiente enlace[13] en la sección `Bitwise Operator`.

La función encargada de realizar esta tarea se encuentra en el código fuente `/lib/dcm-parser.js` función `bitsManipulate` número de línea 311.

Para entender el funcionamiento de la función implementada seguiremos usando el mismo ejemplo anterior, donde tenemos 16 `Bits Allocated`, 12 `Bits Stored`, `High Bit` de 11 y `Samples` igual a 1.

La función recibe como entrada 3 parámetros. Como primer parámetro recibimos la diferencia entre los `Bits Allocated` 16 y los `Bits Stored` 12, por lo tanto, la diferencia será 4. En el segundo parámetro recibimos 8 bits pertenecientes a la primera muestra de los 16 bits. Por último, como tercer parámetro recibimos los siguientes 8 bits pertenecientes al resto de la muestra, por lo tanto, nuestra muestra de 16 bits se divide en 2 partes de 8 bits, toda vez que nuestro arreglo donde está el contenido del archivo se inicializó como un `Array Unit8`. Cada posición en el arreglo será de 8 bits, por ende, nuestras variables quedan de la siguiente forma:

```
1 var mov = 4,
2     bits8a = 10,
3     bits8b = 5;
```

Código 4.14: *Definición de variables en la función bitmanipulate*

Como primer paso debemos tomar nuestra muestra de 8 bits y convertirla en una muestra de 16 bits. Esto se hace rellenando nuestra muestra de 8 bits con ceros.

En nuestra función lo definimos de la forma como se muestra en el código 4.15, donde nuestra variable `bits8a` representa a los primeros 8 bits y `0xff` representa a 8 bits de ceros, por lo tanto, el resultado de la operación (&) se puede ver en el comentario después del `doble slash`.

```
1 var aux = (bist8a & 0xff)   //  0000000 000001010
```

Código 4.15: *Conversión de variable a 16 bits.*

Luego, el resultado obtenido lo desplazamos 8 bits hacia la izquierda con el operador binario `<<`, tal como se muestra en el código 4.16. La razón del por qué se desplazó 8 bits radica en que necesitamos concadenar nuestro primer valor del arreglo con el siguiente y así formar una sola muestra de 16 bits.

```
1 aux = aux << 8  // 000001010 00000000
```

Código 4.16: *Desplazamiento de bits para la variable aux.*

Luego, realizamos el mismo procedimiento que en el código 4.15, pero ahora para nuestra variable `bits8b`, que es nuestra siguiente muestra de 8 bits. Eso se puede observar en el código 4.17.

```
1 var aux2 = (bist8b & 0xff) // 00000000 00000101
```

Código 4.17: *Desplazamiento de bits para la variable aux2.*

Por lo tanto, a estas alturas del proceso, tenemos nuestro valor `aux` que tiene un largo de 16 bits y nuestro valor `aux2` que también tiene un largo de 16 bits,

por lo tanto, ahora lo que necesitamos es combinar estos dos resultados a través del operador (|), como se observa en el código 4.18.

```
1       var final_value = (aux | aux2); //000001010 00000101
```

Código 4.18: *Desplazamiento de bits para la variable aux2.*

Ya que tenemos nuestros valores de las muestras en una sola variable de 16 bits, debemos eliminar los primeros valores de esta muestra, ya que como se mencionó anteriormente sólo utilizaremos los primeros 12 bits. Es por eso que necesitamos la diferencia entre 16 y 12 bits (4), porque esta es la cantidad que debemos tener presente para desplazar nuestros valores, por ende, desplazamos 4 valores a la izquierda << y 4 valores a la derecha >>, como se aprecia en el siguiente código:

```
1           final_value = (final_value & 0xffff) << mov; //01010
               000001010000
2           final_value = (final_value & 0xffff) >> mov; //000001010
               00000101
```

Código 4.19: *Desplazamiento de bits para la variable aux2.*

Como nuestros valores no eran tan elevados no hubo diferencia al desplazarlos, por lo tanto, este proceso se debe aplicar cada 2 muestras de 8 bits. Esto quiere decir por cada dos posiciones del arreglo hasta finalizar la lectura del Pixel Data.

4.3. Info.js

4.3.1. Introducción

`Info.js` es la encargada de hacer transformaciones adecuadas al set de datos. Esto se aplica una vez que se termina de procesar las archivos DICOM.

4.3.2. Filas y Columnas

Se debe consultar al objeto obtenido por el procesador de datos DICOM y buscar el `Tag 0028,0010` y el `Tag 0028,0011`, que pertenecen al `Width` y `Height`. Una vez que se obtiene estos valores es necesario se hace la transformación correspondiente de a un número entero.

4.3.3. Generando Estructura

Para generar la estructura es necesario constar con nuestro `Width` y `Height`. La función encargada se denomina `generateVertex` y recibe como parámetros recibe el ancho y alto de la imagen.

Normalización

Este proceso de normalización se lleva a cabo con 2 funciones, una se denomina `normalize` y otra `getColor`. Con `normalize` lo único que haces recorrer el contenido del Pixel Data y con `getColor` se hace el calculo de la normalización. Para el calculo de la normalización es necesario obtener el valor máximo y mínimo de nuestro Pixel Data.

t

```
1  getColor : function (value) {
2    return ((value - this.min_value)/(this.max_value - this.min_value)
       );
3  },
4  normalize : function () {
5    var i = 0 , files = this.size(),
6        j ,
7        end = this.data[0].length;
8    for(; i < files ; i++) {
9      for(j = 0 ; j < end ; j++ ) {
10       this.data[i][j] = this.getColor(this.data[i][j]);
11     }
12   }
13 },
```

Código 4.20: *Función de normalización.*

Por lo tanto una vez terminado el proceso de normalización de nuestro set de datos se tendrán una rango entre [0,1]. Esto es necesario debido a que `WebGL` (Será explicado más adelante) trabaja con el formato `RGB`[2] con valores entre [0,1],

[2]Formato de colores donde se utiliza cambios de intensidad de colores primarios(Red,Green,Blue) para la obtención de otros colores.

Planos Ortogonales

A través de las funciones `changeDataY` y `changeDataZ` reorganizamos nuestro set para generar los planos ortogonales, simplemente se recorren de forma diferente, no lineal.

4.4. Prototipo 2

4.4.1. Introducción

El prototipo número 2 consta de la implementación de una visualización en 2 dimensiones. Para esto será necesario utilizar `WebGL`, una tecnología introducida en el navegador para dibujar sobre éste.

4.4.2. WebGL

En realidad, `WebGL` es `OpenGL`[3]. Esta es una implementación web de `OpenGL` que se basa en la versión `OpenGL ES 2.0` y que permite trabajar con funciones `Shaders`, las cuales serán explicadas más adelante. Por lo tanto, si comparamos la arquitectura de una página web normal con una página que utiliza `WebGL` quedaría de la siguiente manera:

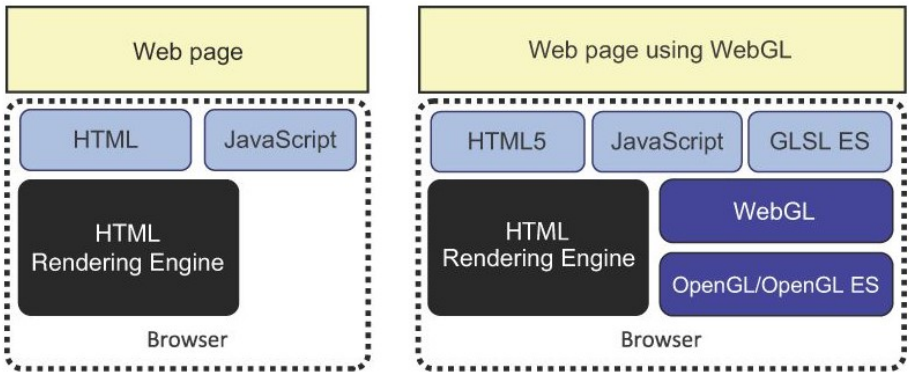

Figura 4.5: *Página web vs Página web utilizando WebGL.*

Como demuestra la figura 4.5, en la parte superior de la arquitectura se introduce un nuevo lenguaje denominado `GLSL` que es utilizado por `OpenGL` para manejar los `Shaders` y está basado en el lenguaje `C`.

[3]Librería grafica de código abierto.

Shaders

Ya se hizo mención acerca de los `Shaders`, pero todavía no tenemos una definición clara y precisa sobre qué son estos.

De esta manera, podríamos señalar que los `Shaders` son pequeños programas que se escriben en lenguaje `GSLS` y que se comunican con `WebGL`. Cabe destacar que `WebGL` trabaja con dos tipos de `Shaders` y estos son:

- Vertex Shader: tiene relación con la posición y el color de un vértice. Este vértice puede ser en 2 o 3 dimensiones.
- Fragment Shader: es un proceso que se realiza por cada vértice declarado en el `Vertex Shader`, por ejemplo un proceso se podría considerarse el color que va a tomar por cada vértice.

En la figura 4.6 que se muestra a continuación, se puede apreciar cómo es el proceso de pintado que utiliza `WebGL`.

Primero, definimos todo lo necesario en nuestro archivo de `Javascript` para inicializar nuestro `WebGL`; luego, a través de nuestro `Javascript` le entregamos la información de los vértices al `Vertex Shader`; posteriormente declaramos en nuestro `Fragment Shader` que cada vértice será de un color (en este caso será de color rojo) y por último lo pintamos en nuestro `Canvas`.

Esta sería la descripción básica sobre el funcionamiento de `WebGL`.

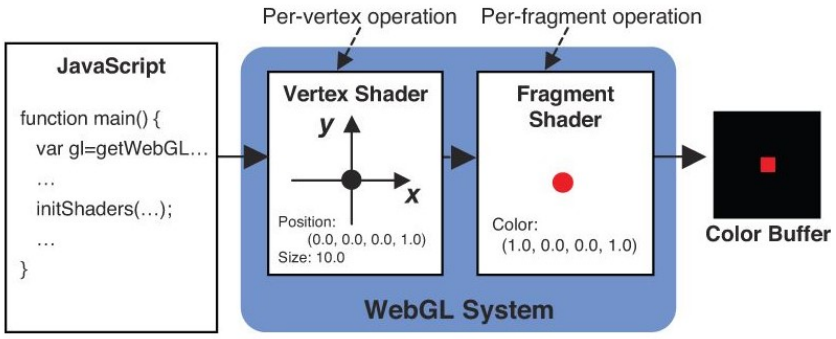

Figura 4.6: *Proceso de WebGL*

4.4.3. Librerias

Dentro de las librerías que se utilizaron para la implementación de la visualización en 2 dimensiones podemos mencionar a tres:

Dat.gui

`Dat.gui`[12] es una librería de `Javascript` que permite cambiar el valor de una variable definida en un `objeto` a través de una interfaz gráfica que proporciona. La idea central es que cada variable que definamos en nuestro objeto se pueda modificar fácilmente, por ejemplo, en la figura 4.7 se puede observar la variable `message` que es de tipo `string`, `speed` de tipo float, `displayOutline` de tipo boolean, y por último `explode` que es la llamada de una función. Por lo tanto, si queremos modificar estas variables sólo debemos hacerlo a través de la interfaz señalada anteriormente.

Figura 4.7: *Librería Dat.gui*

Stats

`Stats`[1] es una librería de `Javascript` que permite monitorear en tiempo real nuestra aplicación, permitiendo saber los cuadros por segundo, los milisegundos que necesita para renderizar la aplicación, y por último la memoria utilizada por el programa. En la figura 4.8 que se observa a continuación se puede apreciar la forma como se muestra la información del monitoreo.

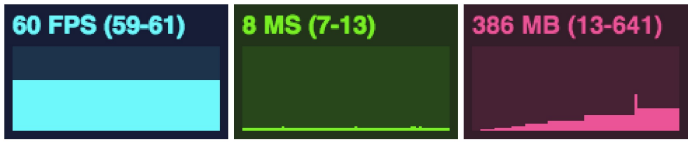

Figura 4.8: *Librería Stats*

Gl-matrix

`GL-matrix`[1] es una librería de `Javascript` que permite trabajar con matrices y vectores, teniendo como su principal característica su buen desempeño en el desarrollo de aplicaciones que utiliza `WebGL`.

4.4.4. Visualización en 2 dimensiones

Una vez que nuestros archivos pasaron por el paseador de datos DICOM y nuestro `info.js`, debemos inicializar el contenido de `WebGL`, tal como se observa en la figura 4.9.

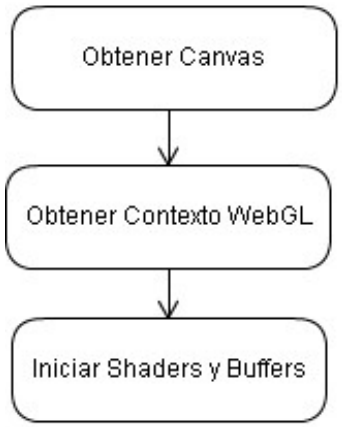

Figura 4.9: *Proceso de iniciación de WebGL*

Nuestro primer paso es obtener nuestro `Canvas` que se encuentra en el cuerpo del HTML; luego se obtiene el contexto de `WebGL`; se inicializan los `Shaders` que consisten en un proceso de validación para ver si nuestro programa escrito en `GLSL` compiló correctamente, y los `Buffers`, que se refieren al espacio donde se almacena la información del Pixel Data y vertices.

A continuación en la figura 4.10, tenemos los pasos posteriores a los de la figura 4.9, que son; `Update`, `Clear` y `Draw` que se transforman en un ciclo sin fin, ya que forman parte del renderizado de nuestro programa.

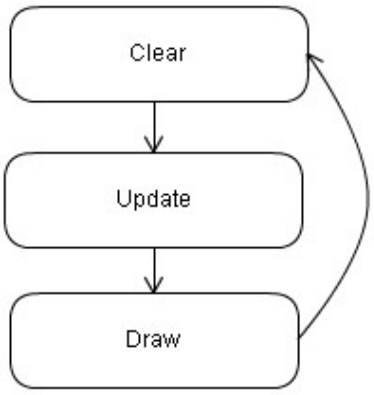

Figura 4.10: *Proceso de Renderizado.*

Lo descrito anteriormente constituye a grandes rasgos el procedimiento de cómo se realizaría el proceso de visualización de datos. Ahora, se explicarán en detalle los procesos que fueron marcados.

Inicializar Shaders y Buffer

Como se señaló anteriormente, nuestro `Shaders` son programas pequeños que se comunican con `WebGL` para dibujar sobre el navegador, por lo tanto, nuestro `Vertex Shader` lo definimos de la siguiente manera:

Vertex Shader

```
1  attribute vec4 a_position;
```

```
2   attribute float a_color;
3   uniform mat4 u_projMatrix;
4   uniform mat4 u_viewMatrix;
5   uniform float u_pointSize;
6   varying float v_color;
7
8   void main() {
9     gl_Position = u_projMatrix * u_viewMatrix * a_position;
10    gl_PointSize = 1.0;
11    v_color = a_color;
12  }
```

Código 4.21: *Definición de Vertex Shader*

Como se puede observar en la imagen, tenemos 2 variables de tipo matriz (mat4): una pertenece a la matriz de proyección y la otra matriz pertenece a la de vista.

Nuestra matriz de proyección la definimos con la librería `GL-matrix`, donde tenemos dos opciones para caracterizarla: una, como una matriz ortogonal, y otra, como una matriz de perspectiva. Se decidió optar por la proyección ortogonal debido a que las dimensiones que se presentan no se alteran, por lo tanto, no nos dará la sensación de que el objeto proyectado sea más grande o más pequeño, dependiendo del punto de vista.

Luego tenemos tres variables de tipo float, donde dos de ellas representan un color y la otra el tamaño del pixel. Es aquí donde entra en juego la figura de los `Buffers`.

Los `Buffers` son arreglos de tamaño definido que manejan grandes cantidades de información. En nuestro caso, se necesita manejar nuestro Pixel Data y la posición que va a tener en la pantalla. Los `Buffers` se inicializan y se cargan a través de Javascript y una vez que se estos se cargan, se pueden ejecutar los programas `Shaders`.

Por último, tenemos un vector (vec4) donde nos indica la posición que va a tener en la pantalla cada valor del pixel data.

En otro orden de ideas, nuestra función `main` es la encargada de arrancar nuestro programa. Este programa se ejecuta por cada valor del Pixel Data, por lo tanto, si tenemos 250.000 mil pixeles, esa será la cantidad de veces que se ejecutará.

`gl_position` y `gl_PointSize` son variables definidas por `WebGL` y son utiliza-

das para asignar el valor que deseamos. Como se puede observar, para nuestro `gl_position` realizamos una multiplicación entre dos matrices y un vector, lo que nos da como resultado el vector.

La idea de trabajar con una matriz de proyección y una matriz de vista, es que en caso de que queramos realizar una rotación a nuestro vector posición, sólo cambiaríamos nuestra matriz de vista por una matriz de rotación.

Fragment Shader

En el código 4.22 definimos el `Fragment Shader`. Cabe destacar que es en nuestro `Fragment Shader` es donde definimos también la función de transferencia que nos ayudará a modificar los valores de la visualización.

```
1  #ifdef GL_ES
2  precision highp float;
3  #endif
4
5
6  uniform float u_densidad;
7  uniform float u_transparencia;
8  uniform sampler2D u_textura_transf;
9
10 varying float v_color;
11
12
13 vec4 getColor(float color) {
14   vec4 color_final;
15   if(color >= u_densidad) {
16     color_final = texture2D(u_textura_transf,vec2(color,0.0));
17     color_final.a = u_transparencia;
18   }
19   else{
20     color_final = vec4(color,color,color,0.0);
21   }
22     return color_final;
23 }
24
25
26 void main() {
```

```
27    vec4 color_texture = getColor(v_color);
28    gl_FragColor = color_texture;
29  }
```

Código 4.22: *Definición de Frament Shader*

En el código 4.22 definimos el `Fragment Shader`. Cabe destacar que es en nuestro `Fragment Shader` es donde definimos también la función de transferencia que nos ayudará a modificar los valores de la visualización.

Como se puede observar en el código 4.22, en la parte superior se define la presión que tendrán los valores en decimales, entre mayor sea su presión, la interpretación de los datos será más cercana a la realidad. Luego, tenemos dos valores float: uno hace referencia a la densidad y el otro a la transparencia.

Como ya sabemos, nuestros valores se encuentran normalizados, por lo tanto, toman un valor del 0.0 a 1.0; entonces, cuando hacemos un filtro por densidad mostramos todos los valores superior al valor de la densidad y en el caso de la transparencia se reemplaza el valor por el valor actual de transferencia.

Como se puede apreciar, hay un tipo de atributo que no habíamos mencionado anteriormente que es el `sampler2D`. Este tipo de atributo nos permite trabajar con una textura determinada, permitiendo rescatar colores de ésta y utilizarlos en nuestra visualización.

También se definió una función para obtener un color determinado denominada `getColor`. Esta función tiene por objeto aplicar todo lo anteriormente descrito, ya sea cambiar la densidad, transparencia y/o utilizar los colores de la textura.

Por último, tenemos nuestro `gl_fragColor` que es nuestra variable de `WebGL` para indicar el color.

4.4.5. Renderización

Es el proceso final para dar paso a la visualización de una imagen. Dentro de la Renderización debemos considerar dos puntos importantes: uno, es la actualización de los datos, y otro, el dibujado de ellos.

Dentro de la actualización de los datos, podemos considerar los mencionados anteriormente tales como cambios en la densidad, transparencia y/o de la función de transferencia, además, debemos agregar el cambio de una imagen a otra.

Como se había mencionado en un principio, nuestro set de datos consta de 300 imágenes, por lo tanto, se necesita una manera que nos permita pasar de una imagen a otra, que se permitan rotaciones de la imagen, etc., entonces, cuando se tienen definidos todos los cambios procedemos al dibujado.

En el dibujado de los datos se utilizan `Buffers` definidos que son llenados con la información actual de la imagen para luego pasar por nuestros `Shaders`, que realizan el trabajo de dibujado. Si se desea ir al detalle con la implementación del prototipo 2 se puede ir al código fuente `js/views.js`.

4.4.6. Resultados

Interfaz

A continuación, se mostrarán los resultados obtenidos luego de la implementación de la visualización en 2 dimensiones.

En la figura 4.11 podemos observar cómo sería la primera mirada de los datos una vez que cargamos el set de datos.

Figura 4.11: *Visualización sin filtros.*

En la parte izquierda del programa podemos observar un panel de información que entrega la información necesaria para la visualización de los archivos, tal como se observa en la figura 4.12.

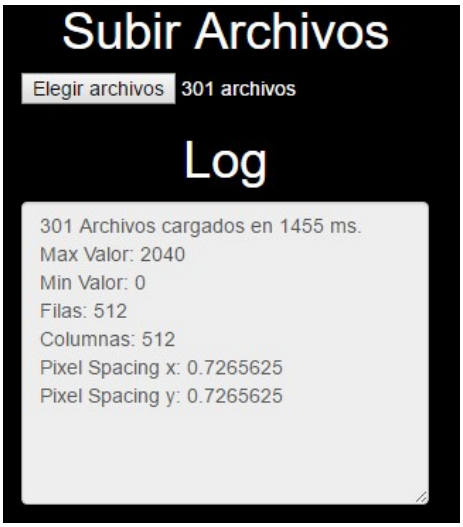

Figura 4.12: *Información de archivo DICOM.*

En la parte derecha del programa podemos observar un panel de configuraciones que fue construido con la librería `dat.gui`.

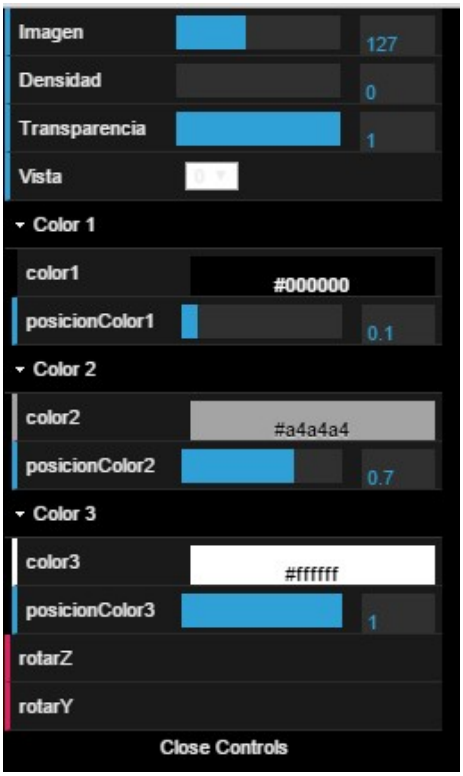

Figura 4.13: *Información de archivo DICOM.*

Funcionalidades

Dentro de las funcionalidades que se implementaron en la visualización 1 encontramos:

Cambio de imágenes

Como se puede observar en la figura 4.14, a medida que la barra de imagen varía, también cambia el contenido de ésta, teniendo como largo máximo la cantidad de imágenes utilizadas.

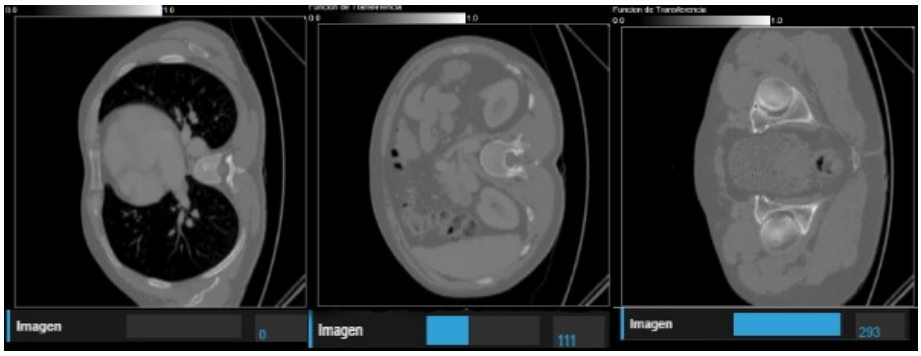

Figura 4.14: *Cambio de Imágenes.*

Filtro por densidad

Como se observa en la figura 4.15, a medida que nuestra barra de densidad varía, también cambia el contenido de la imagen, pero esta vez modificando la imagen en sí.

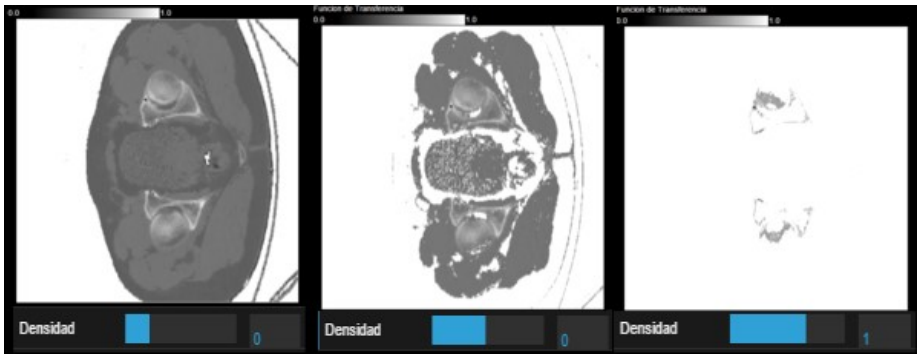

Figura 4.15: *Aplicación del filtro Densidad*

Filtro de transparencia

En la figura 4.16, se puede ver la aplicación del valor de transparencia. Este valor se aplica a toda la imagen; es por eso que a medida que disminuimos el valor llegamos al punto en que casi es transparente.

Figura 4.16: *Aplicación de transparencia.*

Vista ortogonales

En la figura 4.17 se pueden observar las vistas ortogonales que tiene la visualización en 2 dimensiones. Para generar correctamente estas vistas, es necesario tener un set de datos grandes como el que se utilizó en este proyecto, ya que si nuestro set de datos está compuesto por dos o tres imágenes sólo podremos ver correctamente la vista por defecto y respecto de las otras dos sólo se verá una línea, ya que la cantidad de imágenes nos da la profundidad del cuerpo.

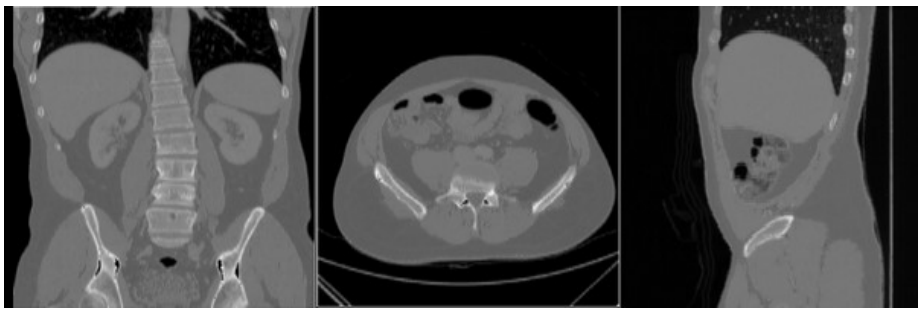

Figura 4.17: *Vista ortogonales.*

Función de transferencia

En las figuras 4.18, 4.19, 4.20 se pueden apreciar diferentes configuraciones que podemos utilizar con nuestra función de transparencias.

Cabe destacar que estas configuraciones son totalmente personalizables, por lo

tanto, se puede definir el color en cada intervalo de la función. Los colores que fueron utilizados como pruebas fueron sacados de otros Software de visualizaciones de imágenes médicas, ya que existen diferentes configuraciones que ayudan a comprender mejor la imagen.

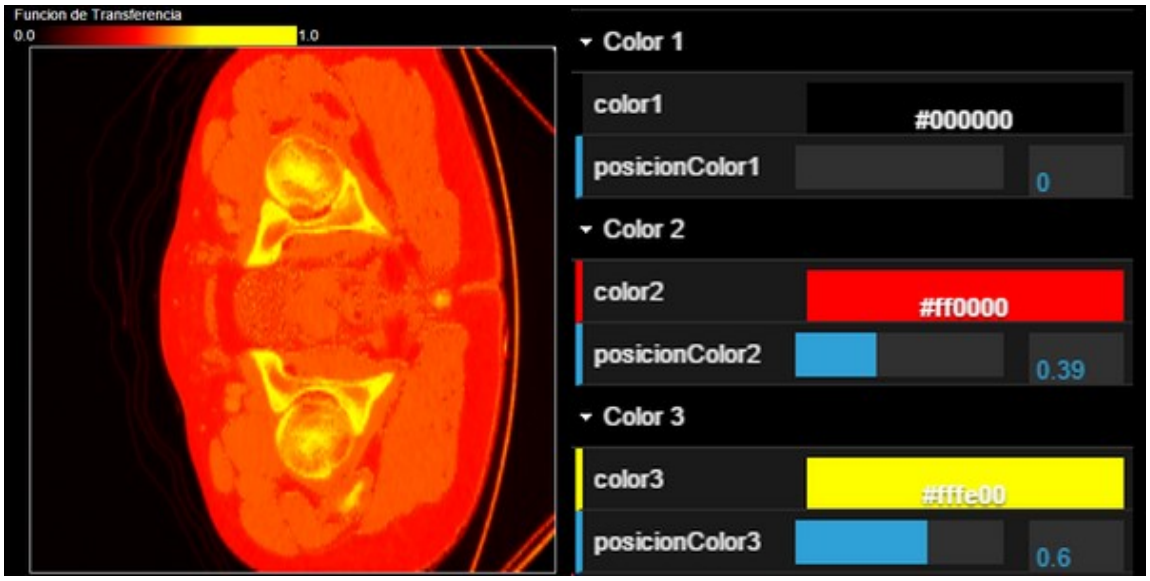

Figura 4.18: *Función de transferencia combinación de colores.*

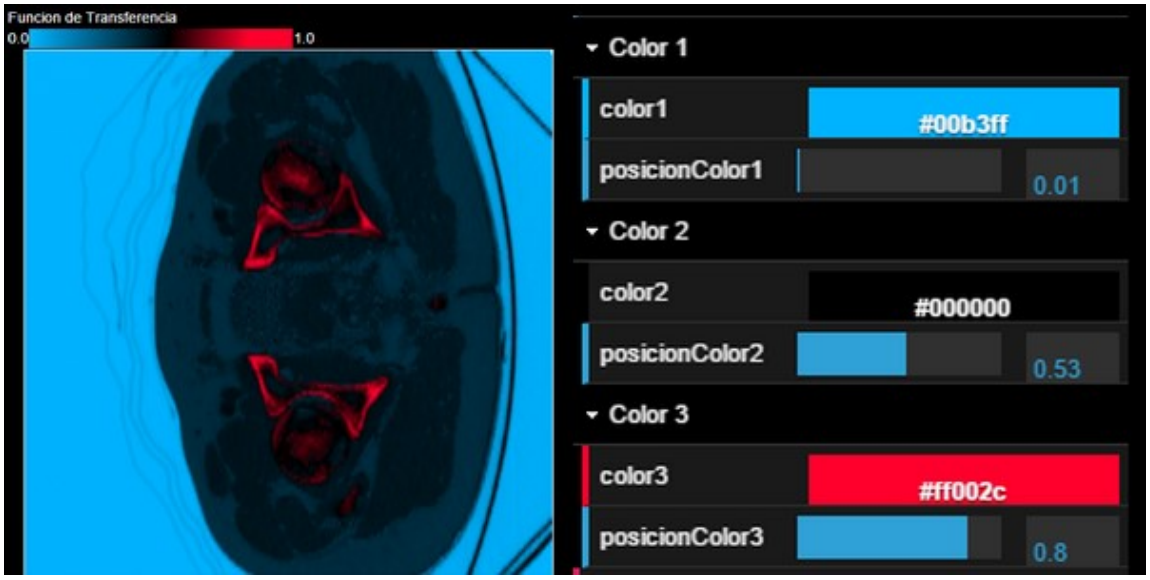

Figura 4.19: *Función de transferencia combinación de colores.*

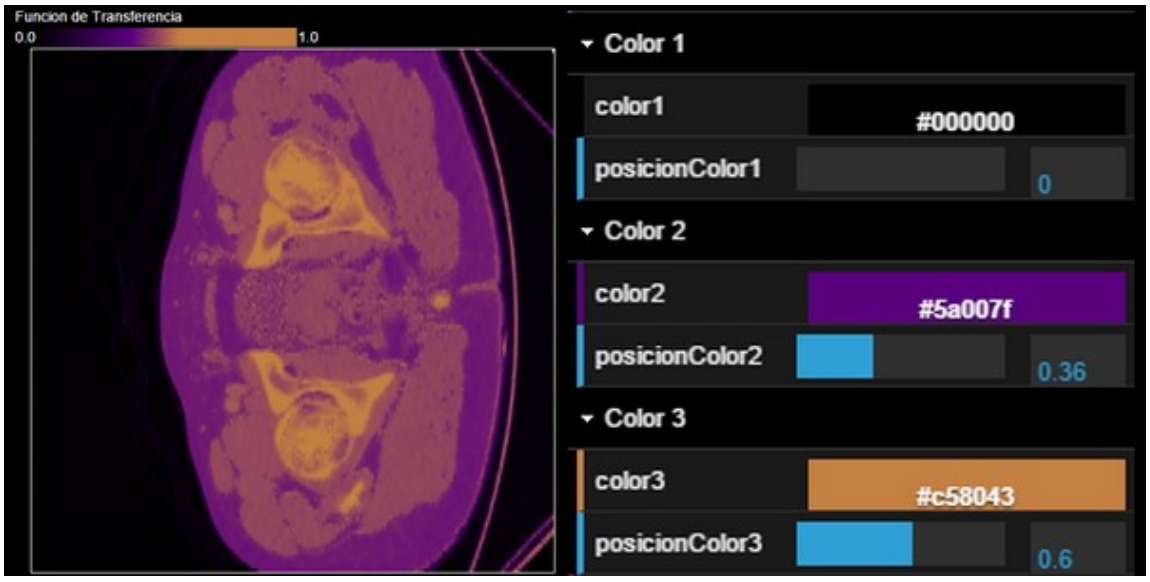

Figura 4.20: *Función de transferencia combinación de colores.*

Rotación en Y y Z

En la figura 4.21 y 4.22 se puede observar las diferentes rotaciones que se permiten hacer a una imagen. Se pueden realizar rotaciones en torno al eje `Z` y al eje `Y`, permitiendo acomodar la imagen a gusto.

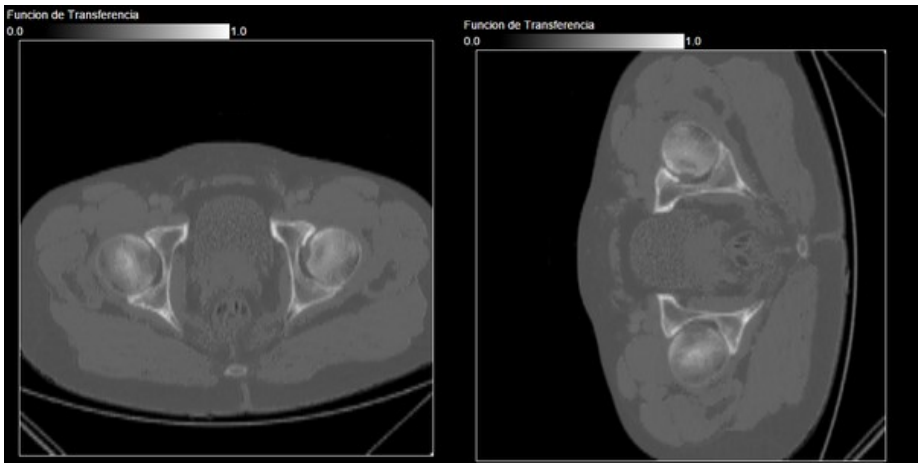

Figura 4.21: *Rotación en torno al eje Z.*

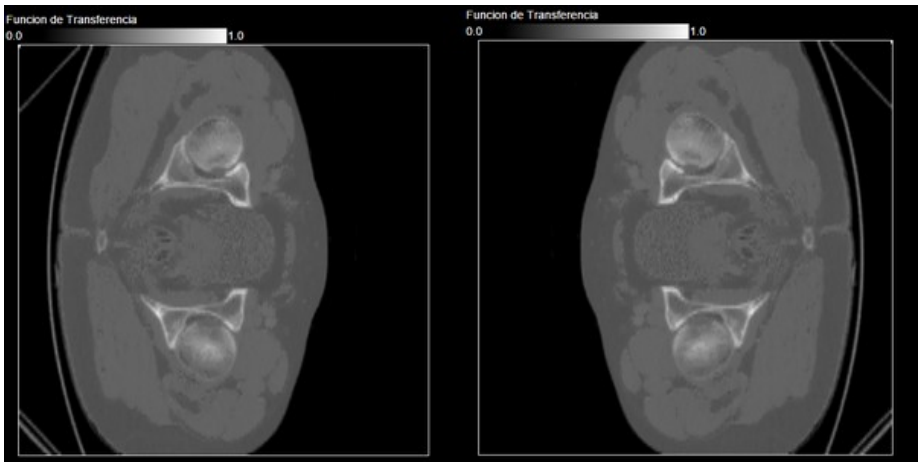

Figura 4.22: *Rotación en torno al eje Y.*

4.5. Prototipo 3

4.5.1. Introducción

El prototipo número 3 consta de la implementación de una visualización en 3 dimensiones. Para esto será necesario utilizar una librería de Javascript llamada `Three.js`[2]. Esta librería está basada en `WebGL` y tiene como principal ventaja que está diseñada optimizada para la implementación de escenas en 3 dimensiones.

4.5.2. Librerias

Además de las librerías utilizadas anteriormente en el prototipo número 2, se utilizó la recién aludida `Three.js` y junto con ella una sub-librería desarrollada para `Three.js` denominada `Orbit Control`. Esta sub-librería nos ayudará a manejar una cámara en la escena en 3 dimensiones donde nos permitirá realizar rotaciones, `Zoom-in` o `Zoom-out` cuando los estimemos pertinente.

4.5.3. Visualización en 3 dimensiones

La visualización en 3 dimensiones tiene la misma lógica que la visualización en 2 dimensiones, pero basándose en cómo trabaja la librería de gráficos `Three.js`.

Inicializar contenido

Se inician los componentes básicos de `Three.js`: escena, cámara y el render. Además, como estamos trabajando con `Orbit control`, es necesario iniciarlo de igual forma. A continuación, procederé a explicar cómo inicializar cada componente.

Escena

Ahora se mostrará cómo inicializar una escena en `Three.js`.

```
1 this.scene = new THREE.Scene();
```

Código 4.23: *Inicializando una escena en Three.js.*

Como se muestra en el código anterior, crear una escena en `Three.js` es bastante sencillo. Cabe hacer presente que es en esta escena donde agregaremos todos los componentes que forman parte de ésta, ya sea cámara, objetos, luz, entre otros.

Camara

A continuación se mostrará cómo inicializar una cámara en `Three.js`:

```
1 this.camera = new THREE.OrthographicCamera(width / - 2, width / 2,
    height / 2, height / - 2, near, far );
```

Código 4.24: *Inicializando una escena en Three.js.*

En el código 4.25 podemos observar cómo definir una vista ortogonal en `Three.js`, donde se recibe como parámetros porciones de nuestro ancho y alto. Por ejemplo, si nuestra ventana es de 500 pixeles por 500 pixeles, entonces recibirá como entrada (500/-2, 500/2 , 500/2 ,500/-2).

Por último, nuestro `near` y `far` nos indica qué tan cerca estará de la pantalla y cuál será el rango de visión máxima.

Renderización

A continuación se mostrará como inicializar la renderización en `Three.js`:

```
1 this.renderer = new THREE.WebGLRenderer();
```

Código 4.25: *Inicializando el render en Three.js.*

Como se puede observar, tenemos la función de crear el render. Por ello es importante destacar que `Three.js` puede manejar dos tipos de render: uno lo realiza con `WebGL`, que tiene un mejor desempeño con visualizaciones en 3 dimensiones pero a mayor costo de recurso, y el otro tipo que provee es utilizando `Canvas`, que tiene un mejor desempeño pero con visualizaciones en 2 dimensiones.

Orbit Control

A continuación se mostrará cómo inicializar la librería `Orbit control` para nuestro `Three.js`:

```
1 this.controls = new THREE.OrbitControls(camara,contexto);
```

Código 4.26: *Inicializando orbit control para Three.js.*

Como se mencionó anteriormente, `Orbit control` es la librería que nos ayudará a que nuestra visualización en 3 dimensiones sea más interactiva, permitiendo realizar acciones sobre la escena. Para ello, `Orbit control` trabaja con la cámara antes definida y el contexto que obtuvimos a través del render anteriormente definido.

4.5.4. Geometría

En resumen, creamos la geometría necesaria. En este punto utilizaremos posicionamiento en la escena en 3 dimensiones. Para esto será necesario utilizar un `objeto` que nos permita crear una geometría en `Three.js` denominada THREE.Geometry, donde le indicamos nuestros vértices y el color que va a ir en cada vértice.

La definición anterior se ve reflejada en el código 4.27:

```
1 var geom =new THREE.Geometry();
2 geom.vertices = this.info.vertex[i]
3 geom.colors = this.info.colors[i];
```

Código 4.27: *Inicializando geometría en Three.js.*

4.5.5. Material

Es necesario crear el material para cada vértice. Es aquí donde entra en contexto nuestro `vertex` y `fragment shaders`.

Como `Three.js` es una librería que se encuentra a más alto nivel, trae definiciones de `vertex` y `fragment shader` por defecto. Sin embargo, no es lo que se busca puesto que necesitamos modificar nuestros `shaders` para aplicar nuestra función de transferencia.

Por fortuna, `Three.js` provee un `objeto` que permite aplicar nuestros propios `shaders` denominado `THREE.ShaderMaterial` y cuya inicialización se puede ver en el código 4.28.

```
1 var material = new THREE.ShaderMaterial({
2     uniforms: this.uniforms,
3     vertexShader: document.getElementById('vertexShader').
          textContent,
4     fragmentShader: document.getElementById('fragmentShader').
          textContent,
5     vertexColors: THREE.VertexColors,
6     size: 1,
7     transparent: false
8 });
```

Código 4.28: *Inicializando un ShaderMaterial en Three.js.*

Como podemos observar, tenemos nuestros `uniforms`[4] que son variables que utilizaremos para la función de transferencia, ya sea la textura a utilizar, el valor de las transparencias o la densidad. Luego, tenemos nuestro `Vertex Shader` y `Fragment Shader`. A este respecto cabe destacar que ambos son los mismos que se utilizaron para la visualización en 2 dimensiones, debido a que se trata de la misma función de transferencia.

A través de `vertexColors` le indicamos el formato que tendrá nuestro color, este

[4]Son variables que cambian constantemente, a deferencia de los que son fijas como es en el caso de los vértices.

caso como estamos trabajando con `Three.js`, utilizaremos el formato de color que se ocupa en la librería.

Por útimo, indicamos el tamaño de cada pixel y la transparencia.

4.5.6. Malla

Una vez que se define la geometría y el material, es necesario realizar una mezcla de estos a través de una malla (Mesh al inglés). `Three.js` provee otro objeto denominado `THREE.PointCloud` el cual nos permite trabajar de manera eficiente grandes cantidades de puntos.

En el código 4.29 se muestra cómo se inicializa `point cloud`.

```
1 var pointCloud = new THREE.PointCloud( geom, material );
2 this.scene.add(pointCloud )
```

Código 4.29: *Inicializando point cloud en THREE.js.*

Como se puede observar, nuestro `PointCloud` recibe como parámetros nuestra geometría y el material que utilizaremos. Una vez que esté definido, podemos agregar el nuevo componente a la escena a través de la función `add`.

4.5.7. Modelo de Luz

No entraremos a analizar en profundidad este tema, ya que por el momento no tiene mayor relevancia y sólo utilizaremos los modelos de luz que nos provee `Three.js` sin mayores detalles.

En el código 4.30 se mostrará cómo se agrega un modelo de luz.

```
1 var light = new THREE.AmbientLight(0xffffff);
2 this.scene.add(light);
```

Código 4.30: *Inicializando un modelo de luz en THREE.js.*

Como se puede ver en el código 4.30, se eligió un modelo de luz ambiental, por tanto, esta función recibe como parámetro el color que tomará ese modelo. Luego de definirlo, se agrega a la escena correspondiente.

4.5.8. Render

Este es el último paso y funcionará como un ciclo sin fin, donde se revisará permanentemente si ha habido algún cambio en las variables de densidad, transparencia, cámara, o si se ha realizado algún cambio en la textura utilizada para la función de transferencia.

4.5.9. Resultados

Interfaz

Al igual que en el prototipo número 2, se utilizó `dat.gui` y `stats`, por lo tanto, no será necesario mostrar lo mismo. Ahora bien, sí hay cambios en la visualización ya que ahora es en 3 dimensiones; además, ahora nuestra visualización es interactiva.

A continuación se mostrará el mismo set de datos utilizado para la visualización en 2 dimensiones. En la figura 4.23 se puede observar cómo sería la imagen que muestra nuestro prototipo 3, sin aplicar ningún filtro ni funcionalidad.

Figura 4.23: *Primera mira de la visualización en 3 dimensiones.*

Funcionalidades

Filtro de Transparecia

En la figura 4.24 se puede observar cómo se utiliza el valor de transparencia. Este valor se aplica a toda la imagen, por lo tanto, a medida que disminuimos el valor se vuelve más transparente.

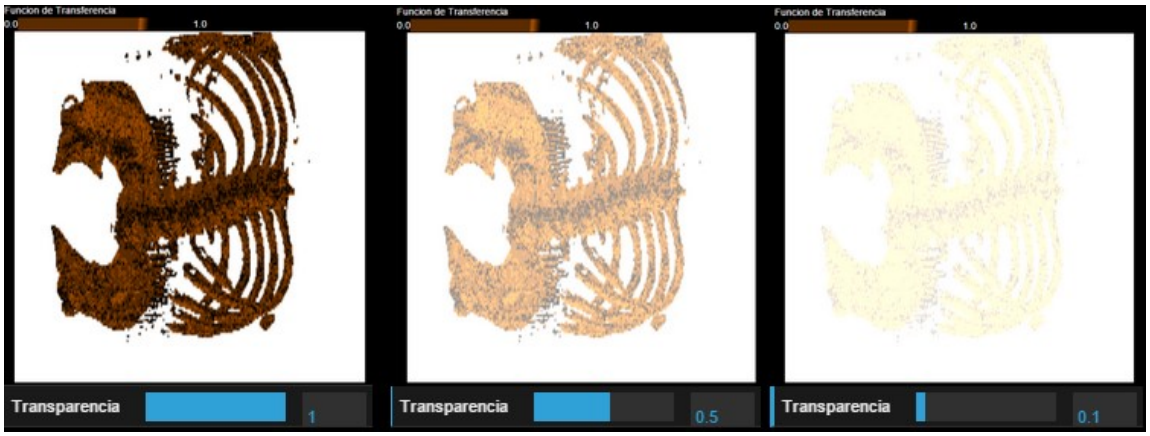

Figura 4.24: *Filtro de transparencia en visualización de 3 dimensiones.*

Filtro de densidad

Como se puede observar en la figura 4.25, a medida que nuestra barra de densidad varía, cambia el contenido de la imagen.

Cabe destacar que solo podremos filtrar valores superiores a 0.6, ya que no era posible carga el set de datos completos.

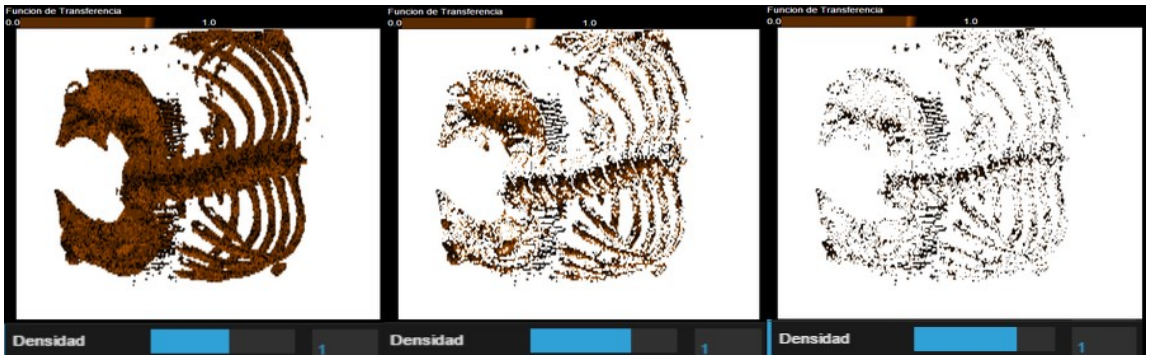

Figura 4.25: *Filtro de densidad en visualización de 3 dimensiones.*

Zoom

En la figura 4.26 y 4.27, se puede observar la funcionalidad de `zoom-in` y `zoom-out`. Es necesario mencionar que esta funcionalidad sólo puede ser ejecutada con el botón intermedio del mouse, ya que no provee un botón en la interfaz que permita realizar

esta acción.

Figura 4.26: *Zoom-in en visualización de 3 dimensiones.*

Figura 4.27: *Zoom-out en visualización de 3 dimensiones.*

Rotación

En la figura 4.28 podemos observar cómo se realiza una rotación de la imagen. Este tipo de rotación no es igual a la implementada en la visualización en 2 dimensiones debido a que no está limitada a la rotación de un solo eje, por lo tanto, se puede rotar en cualquier eje.

Al igual como sucede en el tópico `zoom`, es necesario utilizar el mouse para aplicar esta funcionalidad. Esta se utiliza con el click izquierdo sobre la imagen y se arrastra hacia la dirección deseada.

Figura 4.28: *Rotación en visualización de 3 dimensiones.*

Funcion de Transferencia

En la figura 4.29 y 4.19 se pueden apreciar diferentes configuraciones que se pueden utilizar con nuestra función de transferencias. Cabe destacar que al igual que en la visualización en 2 dimensiones, las configuraciones son totalmente personalizables.

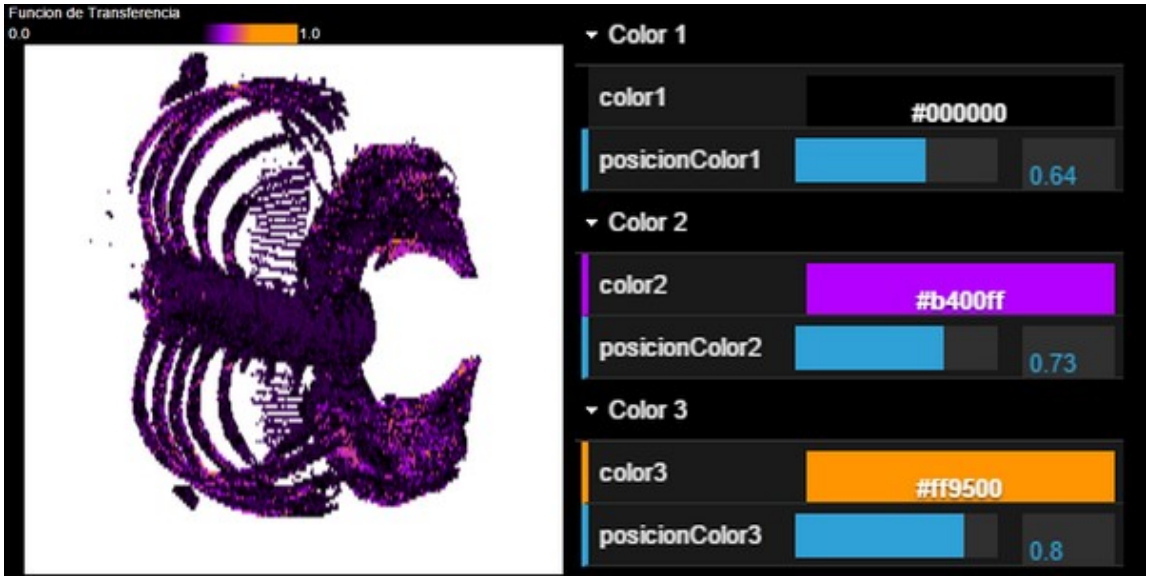

Figura 4.29: *Función de transferencia visualización en 3 dimensiones.*

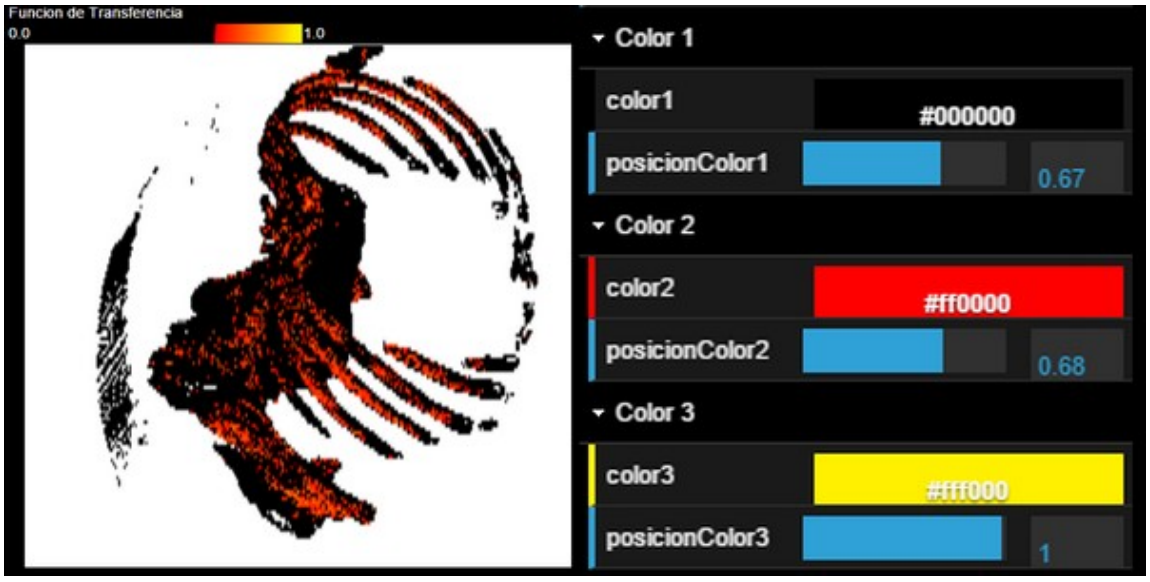

Figura 4.30: *Función de transferencia visualización en 3 dimensiones.*

5. Pruebas

Las pruebas que se consideraron para evaluar la implementación de los prototipos sobre el navegador fueron: cuadros por segundo, memoria utilizada, tiempo de renderizado y tiempo de carga de datos.

Para realizar las pruebas se crearon diferentes set de datos para los prototipos. Se comenzó inicialmente con un set de datos de 300 imágenes, aumentando progresivamente de 50 imágenes hasta llegar a las 750 imágenes, que fue el tope de la implementación puesto que a ese nivel el navegador tardaba demasiado en realizar las acciones y se reiniciaba.

Para poder evaluar estos tópicos y tal como se mencionó anteriormente, realizaremos una tarea a través de la librería `stats.js` que nos entrega información en tiempo real de la memoria, cuadros por segundos y el tiempo de renderizado.

Hay que tener en consideración que los test fueron realizados bajo la siguiente configuración de Hardware y Software:

- Procesador: Intel(R) Core(TM) i5-4670 CPU @ 3.40GHz (4 CPUs), 3.4GHz
- Ram: 6144MB
- Tarjeta de video: NVIDIA GeForce GTX 960
- Navegador: Google Chrome
- Sistema Operativo: Windows 10 Pro 64-bit

5.1. Tiempo de Carga

El tiempo considerado en este tópico es la carga de los archivos sin procesar, implica leer el archivo DICOM y generar la visualización.

En la figura 5.1 se puede observar un gráfico comparativo con los resultados obtenidos tanto en la visualización en 2 dimensiones como en 3 dimensiones, cuyo tiempo de medición fue en milisegundos.

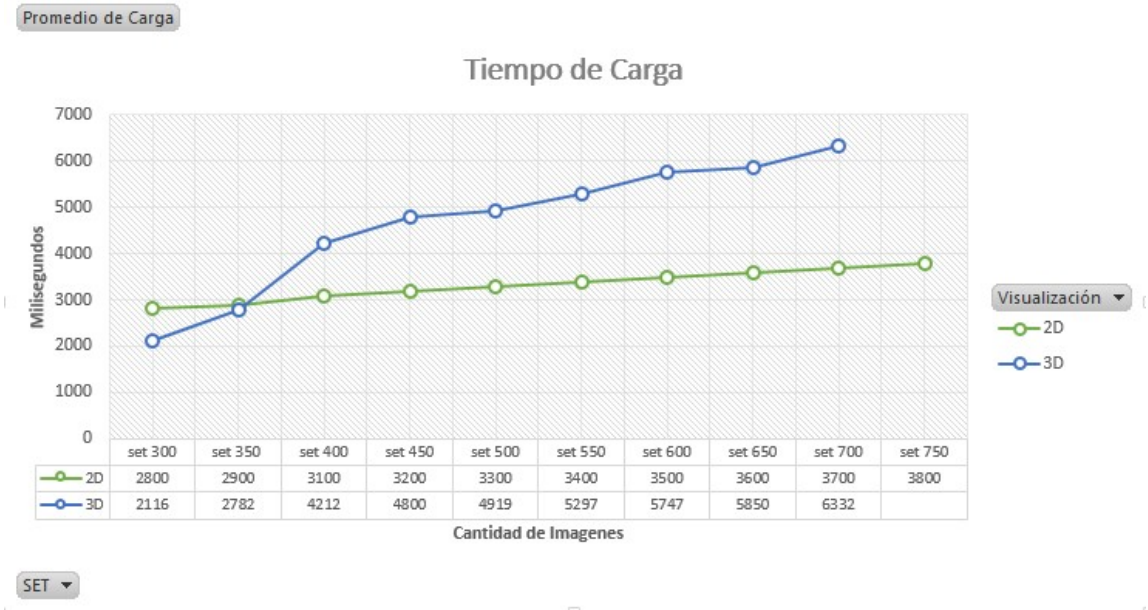

Figura 5.1: *Gráfico comparativo para el tiempo de carga.*

A partir de los resultados obtenidos en el tiempo de carga, podemos concluir que a medida que aumentamos el set de imágenes, el tiempo de carga también aumenta y es algo totalmente esperado, ya que aumentamos la carga de datos en el navegador.

Otro aspecto importante a destacar es el set de datos compuesto por 750 imágenes para la visualización en 3 dimensiones no se encuentra. Esto, debido a que no fue posible generar la visualización para 750 imágenes y ello se debe a que la visualización en 3 dimensiones posee una mayor cantidad de componentes en comparación a la visualización en 2 dimensiones, como por ejemplo la adición de una tercera coordenada, generar mallas, etc.

5.2. Cuadros por Segundos

Los cuadros por segundo nos indican la velocidad en que nuestro programa muestra las imágenes y donde éstas son denominadas `cuadros`, por lo tanto, a mayor cantidad de cuadros por segundos, con mayor fluidez se verán las imágenes; y a menos cuadros por segundos, las imágenes se verán más cortas, dando una sensación de saltos en los intervalos de tiempo.

En la figura 5.2 se muestra un gráfico comparativo con los resultados obtenidos para los cuadros por segundo aplicadas para las visualizaciones en 2 y 3 dimensiones.

Figura 5.2: *Gráfico comparativo para los cuadros por segundo.*

Como se logra apreciar en los resultados de la figura 5.2, los cuadros por segundo en ambas visualizaciones no varían mucho de un set de datos a otro. Esta situación se debe a que siempre se están ejecutando las mismas funciones para dibujar y no constituyen una gran carga para el programa.

Los casos que pueden darse al existir una baja de cuadros por segundos ocurren cuando aplicamos algunas funciones, por ejemplo, en la visualización en 2 dimensiones al cambiar de imagen o modificar la densidad, y en la visualización en 3 dimensiones al cambiar la densidad o realizar rotaciones.

5.3. Tiempo de Renderizado

renderizado menor va a ser el rendimiento de nuestro programa; mientras que a menor tiempo, mayor será el rendimiento de éste.

Lo anterior se define por la cantidad de acciones que tiene que realizar el programa; un ejemplo de esto puede ser cuando se procesan los `shaders`, que, como ya vimos, son programas que aplicamos por cada vértice en nuestra aplicación.

En la figura 5.3 podemos observar un gráfico comparativo del tiempo de renderizado entre la visualización en 2 y 3 dimensiones.

Figura 5.3: *Gráfico comparativo de tiempode renderizado.*

Como se puede observar en los resultados, el tiempo de renderizado se mantuvo constante a través de los set de datos. Fue algo bastante similar a lo que ocurrió con los cuadros por segundo, ya que siempre aplicábamos las mismas funciones cuando dibujábamos las imágenes, es decir, las variaciones del tiempo varían en base a las acciones que se estén aplicando, como por ejemplo cuando se juega con la función de transferencia o se cambia de imagen.

5.4. Memoria utilizada

El último tópico a analizar es el uso de la memoria que utiliza el programa sobre el navegador.

Es importante destacar que este fue el tópico que nos ayudó a obtener el máximo punto de rendimiento de nuestro programa, debido a que no soportó más de 750 imágenes sobre el navegador.

En la figura 5.4 podemos observar un gráfico comparativo de memoria que hay entre la visualización en 2 y en 3 dimensiones.

Figura 5.4: *Gráfico comparativo de memoria utilizada.*

Como se pudo observar en los resultados anteriores, el punto máximo de nuestro programa en 3 dimensiones es de 2.2 gigabytes.

Cuando se estaba realizando este monitoreo y la memoria superaba los 2.2 gigabytes, el programa se volvía inestable. Por ejemplo, si nuestro tiempo de carga para 700 imágenes era de alrededor 6 segundos y probábamos para nuestro set de datos de 750 imágenes, lo lógico sería que se demorara entre 7 a 8 segundos, sin embargo, superaba los 15 segundos y el navegador lanzaba la excepción de que el proceso tomaba mucho tiempo.

6. Trabajo a futuro

Dentro del desarrollo de los prototipos quedaron algunas cosas inconclusas que pueden ser implementadas en un futuro y dar paso de unos simples prototipos a una aplicación web más formación para la lectura de datos en formata DICOM.

6.1. Integración los prototipos

Como lo fue mencionado anteriormente el prototipo de la visualización en 2 y 3 dimensiones se ejecutaron totalmente por separado, por lo tanto una de las mejoras es integrar estas donde funcionalidades en una sola aplicación, donde se incluya las opciones de ver cada una de estas aplicaciones y que solamente se carguen una vez los datos, además decidir si trabajar todo el proyecto con una librería como THREE.js o trabajarla directamente con WEBGL como fue en el caso de prototipo número.

6.2. Carga de datos

Este es uno de los puntos que se puede mejorar, ya que se debe buscar una solución para manejar de una manera más eficiente los archivos sobre el navegador, ya que nuestro límite es de 750 imágenes. Para el caso de la visualización en 2 dimensiones no es necesario tener todo el set de datos sobre el navegador, ya que solo vemos 1 imágenes a la vez, pero si estamos en la visualización en 3 dimensiones se necesita tener todo el set de datos para generar la imágenes, una de las opciones para solucionar este problemas es cargar el set de datos por sectores que nos permitirá

alivianar la carga de memoria sobre el navegador.

6.3. Procesador de datos actualizado

Como sabemos nuestro procesador de datos DICOM funciona sin problema a la hora de realizar la lectura de datos, hasta que llegamos al pixel data, ya que esto solo permite leer datos con tag OB y que tengan una sola muestra por color, además debemos recordar que no leer archivos de imágenes comprimidas, por lo tanto implementación de lo faltante, se podría tener un procesador de datos DICOM completo.

6.4. Implementar un algoritmo de volumen rendering

Si se llegara a implementar un algoritmo de volumen rendering sería el punto donde nuestro prototipo dejaría de ser como tal y se transformaría en una aplicación más formal, debido a que la visualización en 3 dimensiones tendría una imagen más definida y más cercana a la realidad. Dentro de las opciones que se tiene para la implementación es a través de Ray Cast o Shear Warp, donde me inclinaría más por Ray Cast, ya que tendríamos una calidad de imagen superior.

7. Conclusiones

Este proceso de visualización lo realicé sin previo conocimiento del estándar DICOM,por ende al ser tan grande me tomó más del tiempo necesario comprenderlo y llevar a cabo su implementación. Es por este motivo que no fue posible implementar de forma completa el procesador de datos DICOM, ya que si no continuaba con los siguientes pasos no iba a tener tiempo suficiente para continuar con las visualizaciones en 2 y 3 dimensiones.

Por otro lado, como no se utilizó un algoritmo de renderizado de volúmenes, no se pudo cargar de manera óptima todo el volumen de datos, por lo tanto, fue necesario reducir la cantidad de datos a visualizar y sólo se mostraron valores mayores a 0.6, ello en razón de que si tomábamos todo el set de datos, el navegador lanzaba una excepción de error.

El trabajar bajo una plataforma que en el caso concreto es nuestro navegador, tiene un costo adicional en cuanto procesamiento y memoria requerida se trata. Además, el trabajar con éste puede llegar a generar errores o mal funcionamiento, como ocurre cuando un proceso toma más de lo debido y nuestro navegador lanza una excepción de error, lo cual no se vería reflejado en una aplicación nativa para Windows, Linux o Mac, puesto que solo utilizaríamos los recursos necesarios.

Por último, quedé bastante satisfecho con los resultados obtenidos con las implementaciones realizadas de los prototipos webs, dejando como falencia sólo el manejo de la memoria sobre el navegador, ya que cuando se excedía su uso para set de datos superior a 750 imágenes se volvía algo inestable luego de superar los 2 gigas de memoria.

Bibliografía

[1] Ricardo Cabello. stats.js, 2016.

[2] Ricardo Cabello. Three.js, 2016.

[3] Edwin Catmull. A subdivision algorithm for computer display of curved surfaces. 1974.

[4] David Kamins David Laidlaw David Schlegel Jeffrey Vroom Robert Gurwitz Craig Upson, Thomas Faulhaber Jr. and Andries van Dam. The application visualization system: A computational environment for scientific visualization. 1989.

[5] Robert B. Haber and David A. McNabb. A conceptual model for scientific visualization systems. 1990.

[6] NEMA. Dicom ps3.5 2015b - data structures and encoding. 2015.

[7] NEMA. Dicom ps3.6 2016c - data dictionary. 2015.

[8] NEMA. Introduction and overview. pages 11–13, 2015.

[9] NEMA. Transfer syntax, 2016.

[10] John Pawasauskas. Volume visualization with ray casting. 1997.

[11] Peter-Pike Sloan y Charles Hansen Steven Parker, Yarden Livnat. *Interactive Ray Tracing for Volume Visualization*. IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS, 1999.

[12] Google Data Arts Team. dat.gui, 2016.

[13] w3schools. Javascript bitwise operators, 2016.

[14] Ken Martin Will Schroeder and Bill Lorensen. The visualization toolkit. 1998.

[15] William E. Lorensen y Harvey E. Cline. *MARCHING CUBES: A HIGH RESOLUTION 3D SURFACE CONSTRUCTION ALGORITHM.* General Electric Company Corporate Research and Development Schenectady, New York 12301, Julio 1987.

[16] Philippe Lacroute y Marc Levoy. *Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing.* COMPUTER GRAPHICS PROCEEDINgS,Annual Conference Series, 1994.