



**UNIVERSIDAD
DE GRANADA**

**TRABAJO FIN DE GRADO
INGENIERÍA EN INFORMÁTICA**

Visualización Inmersiva de Datos Médicos

Tractogramas en VTK

Autor

Federico Rafael García García

Director

Francisco Javier Melero Rus



**ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN**

—
Granada, noviembre de 2019



Visualización Inmersiva de Datos Médicos

Tractogramas en VTK

Autor

Federico Rafael García García

Director

Francisco Javier Melero Rus

Visualización Inmersiva de Datos Médicos

Federico Rafael García García

Palabras clave: tractogramas, VTK, VTK.js, datos médicos, MRI, DWI, resonancia magnética, realidad virutal, polilínea

Resumen

El objetivo de este trabajo consiste en desarrollar una aplicación web de visualización de datos médicos inmersiva, permitiendo al usuario interactuar con estos en un entorno 3D de realidad virtual. La visualización de este tipo de datos presenta una serie de retos dado el enorme volumen de información que representan. Este trabajo se ha enfocado en un tipo concreto de datos: tractos nerviosos del cerebro representados como polilíneas. El creciente interés en la realidad virutal y uso de aplicaciones web ha llevado a la creación de este proyecto, donde se aplican técnicas de preprocesamiento y generación de tubos dado un conjunto de polilíneas para facilitar su visualización y rendimiento en un entorno de realidad virtual, todo ello ejecutable desde un navegador web compatible. Una aplicación web tiene alcance global y funciona en múltiples dispositivos, ahorrando además al usuario la necesidad de instalar módulos o programas específicos para su uso. Se ha utilizado para todo ello la librería de gráficos VTK en su versión para web, VTK.js, junto con HTML y JavaScript para crear una interfaz y entorno ejecutable en un navegador web.

Immersive Visualization of Medical Data

Federico Rafael García García

Keywords: tractograms, VTK, VTK.js, medical data, MRI, DWI, magnetic resonance, virtual reality, polyline

Abstract

The objective of this work is to develop a web application for viewing medical data immersively, allowing the user to interact with said data in a 3D virtual reality environment. The visualization of this type of data presents a series of challenges given the enormous volume of information which they represent. This work has focused on a specific type of data: brain nerve tracts, which are represented as polylines. The growing interest in virtual reality and the use of web applications has led to the creation of this project, where preprocessing and tube generation techniques are applied to a given polyline set in order to improve visualization and performance in a virtual reality environment, all runnable from a compatible web browser. A web application has global reach and works on multiple devices, and saves the user the need to install specific modules or programs for its use. The VTK graphics library, in its web version, VTK.js, has been used for this project, along with HTML and JavaScript to create an interface and executable environment in a web browser.

Yo, **Federico Rafael García García**, alumno de la titulación Grado en Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 45898434W, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Federico Rafael García García

Granada a 15 de noviembre de 2019.

D. **Francisco Javier Melero Rus**, Profesor del Área de Lenguajes y Sistemas Informáticos del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado ***Visualización Inmersiva de Datos Médicos***, ha sido realizado bajo su supervisión por **Federico Rafael García García**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 15 de noviembre de 2019.

El director:

Francisco Javier Melero Rus

Agradecimientos

Quisiera agradecer a mis padres, a mi hermano, a mi novia y a mis amigos de Granada por haberme apoyado emocionalmente durante la realización de este trabajo.

Quisiera agradecer a mi amiga Mar, por toda la gran ayuda que me ha dado durante mis años en la Escuela y los buenos momentos compartidos.

También quisiera agradecer a Anna Vilanova, por su involucración y aportaciones al proyecto.

En último lugar, pero no menos importante, quisiera agradecer a mi tutor, Javier Melero, por haberme dado la oportunidad de trabajar en el apasionante campo de la computación gráfica, dejando a mi disposición cualquier cosa que se necesitase para el desarrollo del proyecto.

Índice general

1. Introducción	1
1.1. Método de investigación	1
1.2. Definición del problema	2
1.2.1. Difusión molecular	2
1.2.2. DWI (Diffusion Weight Imaging)	2
1.2.3. Vóxeles	3
1.2.4. Tractografía	3
1.2.5. Interés	4
1.3. Objetivo	5
1.4. Estado del arte	6
1.4.1. Agrupamiento de tractos nerviosos	6
1.4.2. Visualización de tractos nerviosos	8
1.4.3. Interacción con tractos nerviosos	9
2. Especificación de requisitos	11
2.1. Introducción	11
2.1.1. Propósito	11
2.1.2. Ámbito del sistema	11
2.1.3. Definiciones, acrónimos y abreviaturas	12
2.1.4. Visión general del documento	13
2.2. Descripción general	13
2.2.1. Perspectiva del producto	13
2.2.2. Funciones del producto	13
2.2.3. Características de los usuarios	14
2.2.4. Restricciones	14
2.2.5. Suposiciones y dependencias	14
2.2.6. Requisitos Futuros	15
2.3. Requisitos específicos	15
2.3.1. Páginas	15
2.3.2. Funciones	17
2.3.3. Requisitos de rendimiento	18
2.3.4. Restricciones de diseño	19
2.3.5. Atributos del sistema	19

3. Diseño	21
3.1. Módulos	21
3.1.1. Módulo principal	21
3.1.2. Módulo de manipulación de líneas	22
3.1.3. Módulo de selector	22
3.1.4. Módulo de matemáticas	23
3.1.5. Módulo misceláneo	23
3.1.6. Diagrama	23
4. Implementación	25
4.1. Introducción	25
4.2. Requisitos previos	25
4.2.1. Librerías	25
4.3. Datos	27
4.3.1. Formato VTK	28
4.4. Aplicación	29
4.4.1. Entorno	29
4.4.2. Página interfaz principal	30
4.4.3. Página aplicación	31
4.5. VTK.js	32
4.6. Código fuente	33
4.6.1. Módulo misceláneo: misc.js	33
4.6.2. Módulo matemático: js	34
4.6.3. Módulo selector: selector.js	35
4.6.4. Módulo de líneas: lineFunctions.js	37
4.6.5. Módulo principal: index.js	40
4.7. Modo VR	42
5. Pruebas	43
5.1. Introducción	43
5.2. Dispositivos	43
5.3. Conjuntos de datos	45
5.3.1. Corpus callosum	45
5.3.2. Fibres	46
5.4. Funcionamiento de la aplicación	47
5.4.1. Interfaz	47
5.4.2. Widget de interacción	48
5.4.3. Selección	49
5.4.4. Opacidad	50
5.4.5. Radio	51
5.4.6. Nivel de detalle	52
5.4.7. Grosor de tubos	53
5.4.8. Decimación	54
5.4.9. Renderización en el tiempo	55

5.4.10. Lados de tubo	56
5.4.11. Densidad de vértices	57
5.4.12. Color	58
5.4.13. Showcase	59
6. Conclusiones, problemas y trabajo futuro	61
6.1. Conclusiones	61
6.2. Objetivos alcanzados	62
6.3. Problemas	62
6.3.1. Resueltos	62
6.3.2. No resueltos	63
6.4. Trabajo futuro	67
Bibliografía	70
A. Instalación	71

Índice de figuras

1.1.	Resonancia magnética cerebral, donde se muestran vóxeles. Imagen tomada de “MRI Segmentation of the Human Brain: Challenges, Methods, and Applications” (Bart Goossens, 2015).	3
1.2.	Polilínea de 6 vértices. Imagen tomada de Wikipedia.	4
1.3.	Diferentes técnicas para obtener tractos nerviosos de una re- sonancia magnética. Imagen tomada de “Introduction to Dif- fusion Tensor Imaging” (Susumu Mori, 2009)	4
1.4.	Izquierda: polilínea. Centro: tubo generado. Derecha: <i>wireframe</i> del tubo.	5
1.5.	Oclusión de tubos. Los más al frente ocultan los más al fondo.	6
1.6.	Representación de tractos nerviosos como “Prototipos”. N in- dica el número original de tractos; K indica el número de prototipos, con el porcentaje de reducción de tractos entre paréntesis. Imagen tomada de “Parsimonious Approximation of Streamline Trajectories in White Matter Fiber Bundles” (2016).	7
1.7.	(A) Centroide de un cluster. (B) 245 tractos cercanos al cen- troide (A). (C) 23 centroides de (B) obtenidos mediante Quick- Bundles. (D) 3241 tractos cercanos al centroide (A). Imagen tomada de “QuickBundles, a method for tractography sim- plification” (2012).	7
1.8.	(A) Conjunto de tractos originales. (B) Centroides obtenidos con QuickBundles. (C), (D) y (E) Distintos niveles de detalle aplicando triangulación. Imagen tomada de “Progressive and Efficient Multi-Resolution Representations for Brain Tracto- grams” (2018).	8
2.1.	Boceto de la interfaz principal.	16
2.2.	Boceto de la visualización de datos.	17
2.3.	Uso del selector (círculo gris) para interacción. Izquierda: se- lección de un tubo (rojo), donde se aprecia su polilínea (azul). Derecha: punto de intersección con polilínea.	18

3.1. Estructura del software en módulos.	24
4.1. Logo de VTK (Visualization Toolkit).	26
4.2. Terminal Git Bash durante compilación.	27
4.3. Página principal de la aplicación.	30
4.4. Simulación de VR mediante la aplicación.	42
5.1. Renderización del cuerpo calloso utilizando la aplicación.	45
5.2. Renderización del conjunto de datos Fibres utilizando la aplicación.	46
5.3. Izquierda: interfaz de la aplicación. Derecha: mensaje de espera.	47
5.4. Izquierda: opciones del <i>widget</i> expandido. Derecha: opciones del <i>widget</i> ocultas.	48
5.5. Izquierda: selección de un conjunto de fibras combinadas. Derecha: expansión.	49
5.6. Izquierda: opacidad igual a 0. Centro: opacidad igual a 0.068. Derecha: opacidad igual a 1; ocurre oclusión.	50
5.7. De izquierda a derecha: radio igual a 0.05; radio igual a 0.1; radio igual a 0.2; selector visto de cerca.	51
5.8. De izquierda a derecha, arriba a abajo: nivel de detalle igual a 3; nivel de detalle igual a 2; nivel de detalle igual a 1; nivel de detalle igual a 0.	52
5.9. Izquierda: grosor de tubo igual a 0.001. Derecha: grosor de tubo igual a 0.003.	53
5.10. Izquierda: tubo original, sin decimación. Derecha: tubo con decimacion igual a 0.05 (se ha reducido el 95 % de los vértices).	54
5.11. De izquierda a derecha, arriba a abajo: evolución en el tiempo de renderización de tubos tras cambiar la opacidad de 0 a 1.	55
5.12. Arriba: número de lados igual a 5. Abajo: número de lados igual a 50.	56
5.13. Izquierda: densidad igual a 0. Derecha: densidad igual a 0.001.	57
5.14. Gama de colores disponibles en la aplicación.	58
5.15. Funcionamiento de la aplicación para el conjunto de datos <i>Fibres</i>	59
6.1. Error en la generación de un tubo.	65
6.2. Izquierda: renderización sin transparencias. Centro: técnica simple de transparencia; el dragón azul parece estar detrás de los demás. Derecha: <i>depth peeling</i> ; los dragones están renderizados en el orden correcto.	66
6.3. El tubo que se encuentra entre la cámara y el selector, a pesar de ser transparente, oculta parte del selector.	66

Capítulo 1

Introducción

En los últimos años, la tecnología de RV (Realidad Virtual) ha seguido un curso ininterrumpido de avances, con la aparición de nuevos y mejores dispositivos: *HTC Vive Pro*, *Oculus Rift* y *Playstation VR*, entre otros, y a precios más asequibles para el público general [1].

Estos avances tecnológicos ya se han abierto paso en el campo de la medicina, facilitando la visualización e interacción con todo tipo de datos médicos [2] e incluso ayudando a educar y entrenar a cirujanos [3]. Surge por tanto la necesidad de seguir desarrollando aplicaciones capaces de soportar los elevados requisitos de estos dispositivos tan exigentes [4].

Las ventajas de las aplicaciones web incluyen una implementación rápida (no es necesario descargar una aplicación o un complemento de navegador), una mayor compatibilidad multiplataforma (por ejemplo, dispositivos móviles), flexibilidad para compartirlas, y un fácil desarrollo. Recientemente ha habido una gran cantidad de mejoras en cuanto al rendimiento, además de nuevas bibliotecas que han hecho que las aplicaciones web sean comparables a las de escritorio [5].

Con todo ello en mente, se ha diseñado una aplicación web de visualización de datos médicos, en concreto tractos cerebrales obtenidos a través de resonancias magnéticas, atendiendo a estas necesidades; se ha diseñado una técnica de preprocesado con el fin de mejorar el rendimiento durante la ejecución.

1.1. Método de investigación

Se han estudiado los *papers* publicados en el evento *Eurographics Workshop on Visual Computing for Biology and Medicine 2018*, en la que se trataron temas relacionados con el amplio campo de *Computer Graphics* rela-

cionados con el campo de la medicina [6].

Se ha utilizado además el motor de búsqueda *Google Scholar* y las plataformas *Research Gate*, y *Science Direct*, donde se realizaron búsquedas con las palabras clave *web application*, *VR*, *Virtual Reality*, *medical data visualization*, *tractograms*, *MRI* y *voxel*.

El número de artículos encontrados sobre aplicaciones web ha sido amplio, por lo que se ha limitado a estudiar únicamente aquellos que analizan sus beneficios en cuanto a su uso por usuarios.

1.2. Definición del problema

Antes de comenzar a definir el problema que se trata de solucionar en este trabajo, se explicarán algunas definiciones necesarias para su mejor comprensión.

1.2.1. Difusión molecular

La *difusión molecular*, o simplemente *difusión*, es el movimiento de partículas. Esta explica el flujo neto de moléculas que se desplazan de una región de mayor concentración a una de menor concentración [7].

1.2.2. DWI (Diffusion Weight Imaging)

El agua forma un gran porcentaje de la composición del cuerpo humano como fluidos intra y extracelulares. En los tejidos, la difusión de las moléculas de agua sigue un patrón de acuerdo con la estructura y las propiedades del tejido. En algunas condiciones patológicas, como puede ser un infarto de miocardio, este patrón de difusión se ve alterado y la cantidad de difusión cambia en el área afectada. Mediante el estudio de estos cambios en la difusión, se pueden detectar anomalías. Esto se puede lograr utilizando una técnica especializada de resonancia magnética llamada DWI (*Diffusion Weight Imaging*), en la que se hace uso de la difusión de las moléculas de agua para visualizar la fisiología interna. El contraste de la imagen en DWI refleja la diferencia en la velocidad de difusión entre los tejidos [7].

En lo que concierne a este trabajo, nos interesa exclusivamente las resonancias magnéticas de la materia blanca y la materia gris del cerebro.

1.2.3. Vóxeles

Dada la dificultad de estimar la difusión de moléculas individuales, estas se agrupan en regiones denominadas *vóxeles*. Un voxel es la unidad mínima de volumen en forma de cubo que comprende una imagen 3D más grande. Se puede considerar como una versión 3D de un píxel. Por lo tanto, los véxeles de materia gris son las unidades volumétricas medibles más pequeñas que forman el volumen total de materia gris en una imagen cerebral [8].

Para su visualización se suelte utilizar una escala de grises, como se muestra en la figura 1.1.

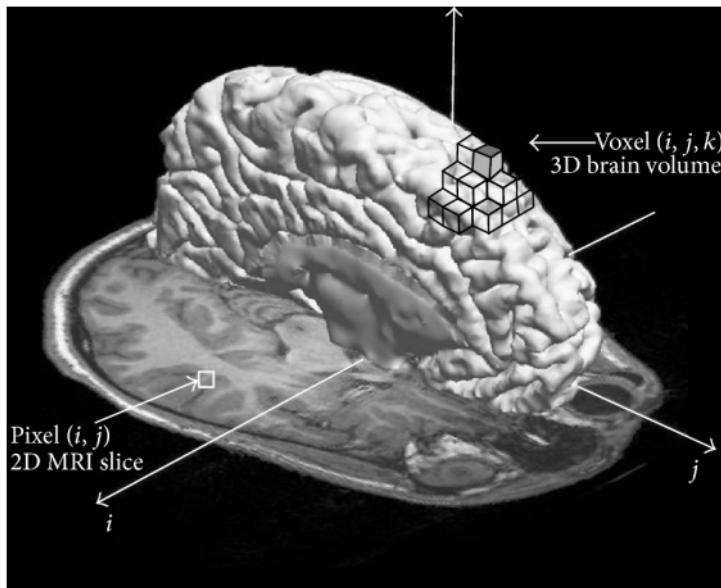


Figura 1.1: Resonancia magnética cerebral, donde se muestran véxeles. Imagen tomada de “MRI Segmentation of the Human Brain: Challenges, Methods, and Applications” (Bart Goossens, 2015).

1.2.4. Tractografía

De los véxeles se puede extraer información adicional mediante la técnica de *tractografía*. Esta consiste en buscar la relación existente entre véxeles adyacentes para determinar los tractos nerviosos del cerebro [9], [10]. Estas tractos nerviosos se pueden representar como polilíneas, es decir, una sucesión de vértices 3D conectados.

En la imagen 1.2 se puede ver una representación de un polilínea, y en

la figura 1.3 la obtención de estas desde un conjunto de véxeles.

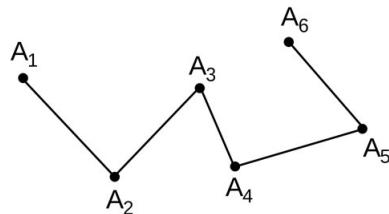


Figura 1.2: Polilínea de 6 vértices. Imagen tomada de Wikipedia.

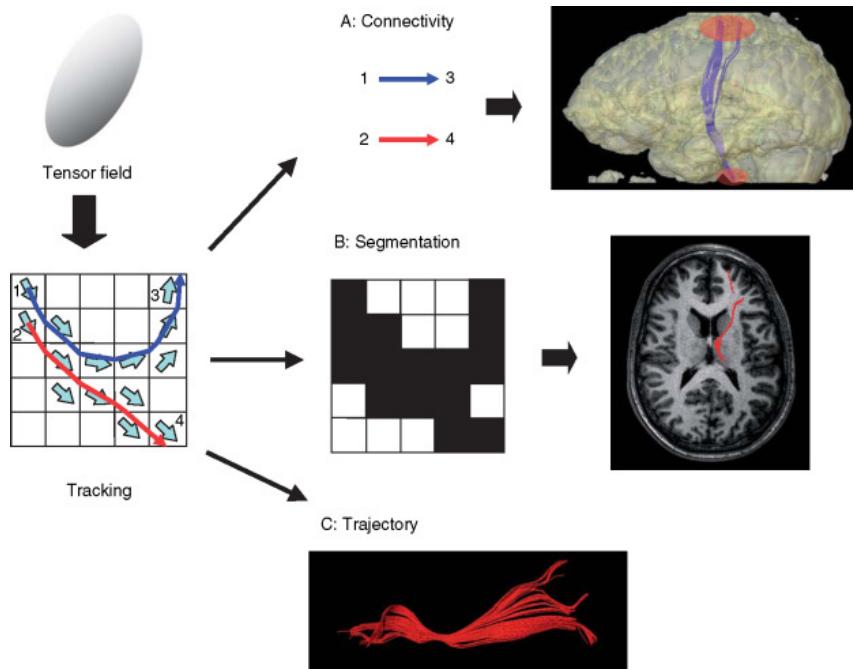


Figura 1.3: Diferentes técnicas para obtener tractos nerviosos de una resonancia magnética. Imagen tomada de “Introduction to Diffusion Tensor Imaging” (Susumu Mori, 2009)

1.2.5. Interés

Son diversos los motivos por los que las técnicas de DWI y tractografía son de gran interés:

- Ayudan a avanzar nuestro conocimiento sobre la anatomía del cerebro.

- Permiten el diagnóstico de patologías como lesiones axonales, desmielinización, esclerosis múltiple, trastornos psiquiátricos como la esquizofrenia, el TDAH y trastornos neurológicos y neurodegenerativos como el deterioro cognitivo, la enfermedad de Alzheimer, etc.
- Actualmente, la tractografía también se utiliza en el mapeo de la conectividad en el cerebro, en el *Human Connectome Project* [7].

1.3. Objetivo

En este trabajo se intentan solucionar diversos problemas relacionados con la visualización e interacción de tractogramas.

El volumen de datos obtenidos de una resonancia magnética de difusión, y los tractos nerviosos obtenidos posteriormente mediante tractografía, puede ser considerable; se pueden obtener cientos de miles de véxoles [11].

Para facilitar su visualización, las polilíneas suelen convertirse en tubos, como se muestra en la figura 1.4.

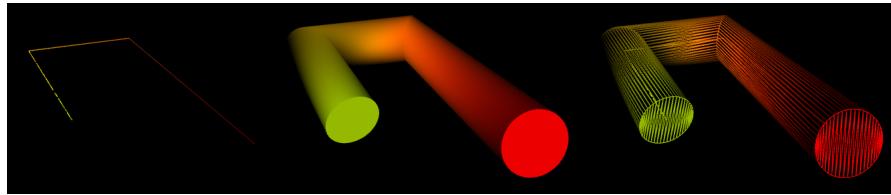


Figura 1.4: Izquierda: polilínea. Centro: tubo generado. Derecha: *wireframe* del tubo.

Navegar sobre este gran conjunto de datos supone varios retos:

- La renderización gráfica requiere de mucha potencia, lo que lleva a una baja tasa de fotogramas y, por lo tanto, dificulta la interacción por parte del usuario.
- La interacción clásica, utilizando ratón y teclado y visualizando los datos en un monitor, presenta varias desventajas al tratarse de datos en tres dimensiones. El ratón solo se puede mover en dos dimensiones; la carencia de profundidad dificulta la interacción. Se dificulta la percepción de los datos al visualizarse en dos dimensiones.
- Existe también el problema de *occlusión*. Al haber tantas polilíneas, o tubos, se ocultan entre ellas, dificultando ver los datos en su totalidad

y ocultando información que puede resultar relevante. En la figura 1.5 se puede ver un caso de oclusión.

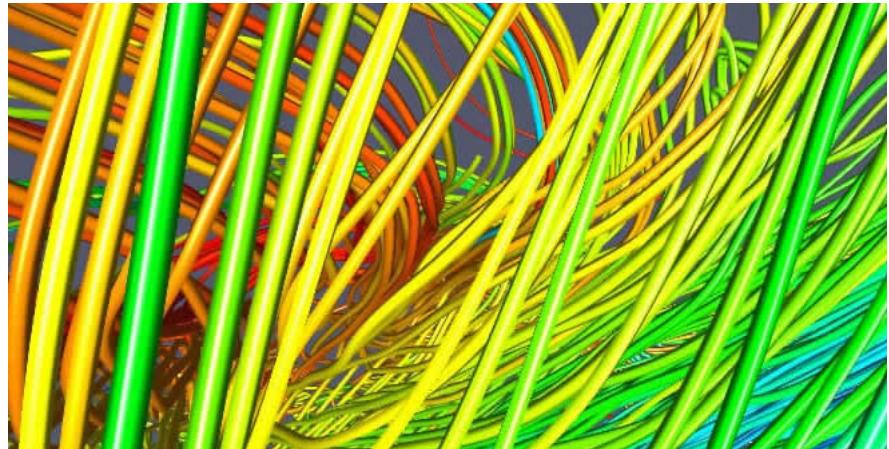


Figura 1.5: Oclusión de tubos. Los más al frente ocultan los más al fondo.

1.4. Estado del arte

Se comentará en esta sección técnicas empleadas en el procesamiento, visualización e interacción de tractos nerviosos.

1.4.1. Agrupamiento de tractos nerviosos

Los tractos nerviosos obtenidos de algoritmos de tractografía contienen muchas polilíneas. Se requiere por lo tanto de una gran cantidad de memoria y recursos computacionales para almacenarlos, visualizarlos y procesarlos. Una reducción de los mismos ayudará al rendimiento. Una buena aproximación consiste en elegir, de un grupo de polilíneas similares, un “prototipo” de polilínea, consistente en la suma de pesos ponderados. Dos polilíneas se considerarán similares si sus puntos extremos son cercanos [12].

En la figura 1.6 se ve la reducción de polilíneas según esta técnica.

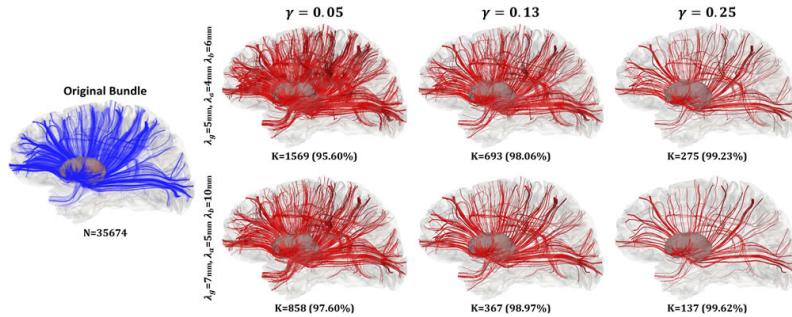


Figura 1.6: Representación de tractos nerviosos como “Prototipos”. N indica el número original de tractos; K indica el número de prototipos, con el porcentaje de reducción de tractos entre paréntesis. Imagen tomada de “Parsimonious Approximation of Streamline Trajectories in White Matter Fiber Bundles” (2016).

Otra técnica, denominada *QuickBundles*, está pensada para combinar grupos de tractos nerviosos formando centroides. Estos representan fielmente el conjunto original de datos. *QuickBundles* es capaz de generar resultados en segundos y utilizando poca memoria. Para ello agrupa tractos en *clusters*; una vez que un tracto es asignado no es reasignado ni tampoco modificado. Un centroide del grupo es el utilizado para representarlo. [13].

En la figura 1.7 se aprecia la reducción realizada por *QuickBundles*.

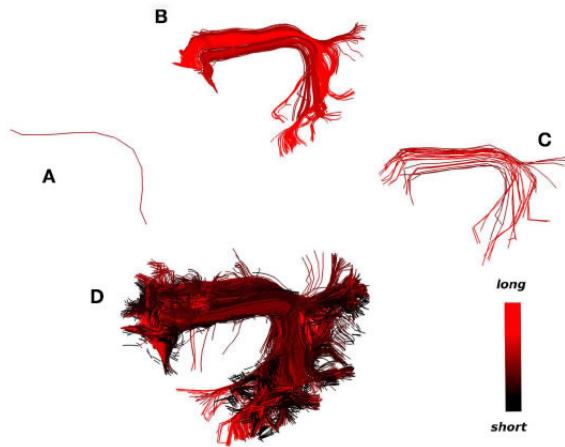


Figura 1.7: (A) Centroide de un cluster. (B) 245 tractos cercanos al centroide (A). (C) 23 centroides de (B) obtenidos mediante QuickBundles. (D) 3241 tractos cercanos al centroide (A). Imagen tomada de “QuickBundles, a method for tractography simplification” (2012).

1.4.2. Visualización de tractos nerviosos

Una polilínea es comúnmente renderizada como un tubo, cuyo color y grosor son elegidos según la dirección y atributos de sus vértices. Para el renderizado se pueden utilizar APIs de gráficos como OpenGL. Se han propuesto diversas técnicas que incrementan la calidad visual (shaders, iluminación, ...) pero afectan al rendimiento. Una solución consiste en aplicar diferentes “niveles de detalle”, donde se define el grado de simplificación de la geometría utilizando exclusivamente la tarjeta gráfica; esto evita la necesidad de preprocesamiento y almacenamiento de nueva información [14].

Basándose en *QuickBundles* y en los “niveles de detalle” comentados anteriormente, otra técnica basada en *Triangulación de Delaunay* agrupa tractos en diferentes niveles, ofreciendo una representación multi-resolución en donde cada en nivel se reduce la precisión de la geometría [15].

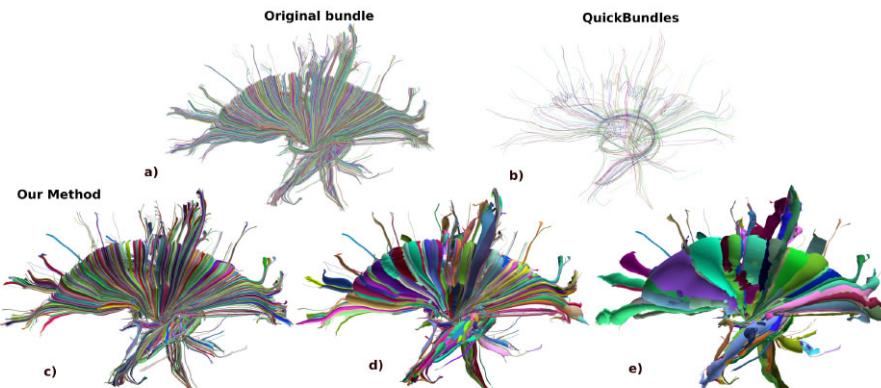


Figura 1.8: (A) Conjunto de tractos originales. (B) Centroides obtenidos con QuickBundles. (C), (D) y (E) Distintos niveles de detalle aplicando triangulación. Imagen tomada de “Progressive and Efficient Multi-Resolution Representations for Brain Tractograms” (2018).

En otra técnica, se analizan similitudes de tractos *a diferentes escalas*, acercándolos entre sí en un proceso iterativo. Esta técnica genera una abstracción de la estructura global del cerebro como así también *vacíos volumétricos*; con ello se reduce la oclusión [16].

1.4.3. Interacción con tractos nerviosos

Interactuar con tractogramas con un gran número de puntos hace que las técnicas clásicas de interacción tengan dificultades en tener tiempos de respuesta bajos. Dado que estas técnicas necesitan iterar rápidamente sobre todos los puntos, aumentar el número de puntos disminuye el rendimiento y la capacidad de respuesta de la aplicación. Por lo tanto, se puede considerar eliminar puntos de los tractos. Otro enfoque es el de agrupar tractos rápida y eficientemente en grupos de diferentes resoluciones [17], como se ha comentado en el apartado anterior.

Capítulo 2

Especificación de requisitos

A pesar de no ser estrictamente necesario, es conveniente cernirse a algún estándar para especificar los requisitos del software para facilitar la tarea. En este capítulo se seguirá lo especificado en el estándar IEEE830. A pesar de estar desfasado (año 2008), sigue siendo una buena guía.

2.1. Introducción

2.1.1. Propósito

En este capítulo se definirán los requisitos funcionales y características del sistema para el desarrollo de software de visualización e interacción de datos médicos, en concreto tractogramas representados como polilíneas.

2.1.2. Ámbito del sistema

- Se denominará a la aplicación como *VTK TRACT DATA VR VISUALIZATION*.
- La aplicación será capaz de, dado una URL de un fichero en formato VTK de polilíneas y algunos parámetros, combinar dichas polilíneas siguiendo un algoritmo prefijado para posteriormente renderizarlas, tanto las polilíneas combinadas como el grupo que la conforma, según el usuario lo deseé. La aplicación permitirá detectar diversos sistemas de realidad virtual compatibles para comunicarse con ellos, enviando imagen y recibiendo *input*. Es deseable que los datos estén normalizados; es decir, que cada posición de cada eje de cada vértice esté comprendido en el rango [-1, 1]. La aplicación será además capaz de ejecutarse en un navegador web (siempre y cuando cumpla ciertos requisitos que se especificarán más adelante). No será capaz de generar o

guardar datos, ni de manipularlos de otra forma a la establecida, como puede ser aplicando otros algoritmos de combinación de polilíneas.

- Con este proyecto se pretende conseguir unificar diversas áreas (preprocesado de datos, renderización de los mismos, interacción con realidad virtual y sensores, compatibilidad multidispositivo, y uso desde un entorno web).

2.1.3. Definiciones, acrónimos y abreviaturas

- **HTML:** *HyperText Markup Language*; lenguaje de formato de páginas web.
- **CSS:** *Cascading Style Sheet*; lenguaje de estilos para páginas web.
- **ERS:** Especificación de Requisitos Software.
- **Polilínea:** conjunto de vértices 3D ordenados, cada uno conectado con el siguiente.
- **VR:** *Virtual Reality* (Realidad Virtual).
- **HMD:** *Head-mounted Display*, dispositivo de visualización colocado en la cabeza.
- **VTK:** *The Visualization ToolKit*; la librería gráfica base de la aplicación.
- **VTK.js:** versión JavaScript de VTK.
- **Widget:** elemento de interacción de una interfaz gráfica.
- **FPS:** *Frames Per Second* (fotogramas por segundo).
- **WebGL:** versión web de la librería de gráficos OpenGL.
- **Selector:** elemento de interacción, como puede ser un disco o plano, que permite interaccionar con objetos 3D al interseccionar o colisionar con estos.
- **Canal de transmisión:** medio físico por el cual viajan los datos como señales.
- **Ancho de banda:** cantidad de datos posibles a transmitir por segundo.
- **Paquete:** conjunto de bits que contiene parte de los datos a enviar por el canal de transmisión.

- **Retardo de transmisión:** tiempo que se tarda en pasar todos los bits de un paquete por el canal de transmisión. Depende del ancho de banda.
- **Retardo de propagación:** tiempo que tarda un paquete en su destino.
- **Caché:** copia local de datos para un acceso más rápido.
- **CPU:** *Central Processing Unit*: procesador.
- **Cuello de botella:** situación en la que la capacidad de procesamiento de un dispositivo es mayor que la capacidad del bus al que se encuentra conectado el dispositivo.
- **HTTP:** *Hypertext Transfer Protocol*: protocolo para la transferencia de información en Internet.
- **HTTPS:** *Hypertext Transfer Protocol Secure*: versión segura del protocolo HTTP, donde la información viaja encriptada.

2.1.4. Visión general del documento

Este capítulo se divide en dos grandes partes.

En la primera se da una visión general de la aplicación, especificando qué puede hacer, tipos de usuarios, limitaciones y posibles mejoras futuras.

En la segunda, se especifican todos los requisitos necesarios del sistema: diseño de interfaces, estructura, mayor explicación de las funcionalidades y limitaciones, necesidades software y hardware, y características del sistema.

2.2. Descripción general

2.2.1. Perspectiva del producto

La aplicación puede utilizarse meramente para la visualización de datos o como base para la creación de un software de mayores capacidades.

2.2.2. Funciones del producto

La aplicación será capaz de:

- Elegir parámetros y modos de visualización.
- Cargar datos de polilíneas en formato VTK.

- Combinar polilíneas cercanas entre sí.
- Generar tubos a partir de las polilíneas.
- Visualizar los tubos.
- Moverse por el mundo.
- Seleccionar tubos y visualizar diferentes grupos.

2.2.3. Características de los usuarios

El usuario puede ser de cualquier ámbito; profesionales de la medicina o investigadores que desean estudiar el cerebro con el fin de detectar anomalías relacionadas con patologías, estudiantes que desean tener una mejor compresión del cerebro, o interesados por el cuerpo humano que desean una experiencia más cercana a la visualización de un cerebro.

2.2.4. Restricciones

La aplicación se desarrollará en JavaScript, con la librería VTK.js. Se hará un desarrollo evolutivo, en el que se intentará implementar las funciones especificadas anteriormente según se avance en el proyecto, pudiendo añadir otras posteriores o eliminar aquellas que no beneficien al proyecto o resulten imposibles de implementar.

2.2.5. Suposiciones y dependencias

Se asume que el usuario dispone del software y hardware necesario para ejecutar la aplicación:

- Navegador web actualizado capaz de ejecutar aplicaciones WebGL, como es el caso de Chrome 78 o Firefox 70 en adelante.
- Drivers y demás software necesario para utilizar el dispositivo que disponga, como puede ser el caso de un casco de realidad virtual.
- Tarjeta gráfica capaz de soportar los datos que desea visualizar.
- Drivers actualizados de la tarjeta gráfica.

2.2.6. Requisitos Futuros

Con la creciente expansión de las redes 5G, se podría modificar el sistema para ser ejecutado desde un servidor remoto de altas prestaciones, y ser accedido por diferentes usuarios con cualquier tipo de dispositivo. Estos enviarían órdenes de interacción al servidor, que renderizaría un nuevo fotograma tras aplicar la acción recibida y devolvería dicho fotograma al usuario. Gracias al bajo retardo de las redes 5G, el usuario no percibiría que la aplicación se está ejecutando de forma remota.

Al ser web, se podría considerar comprimir los datos para reducir su almacenamiento en la nube y descargarlos con mayor rapidez. Una vez estos son recibidos por la aplicación, esta procedería a descomprimirlos y cargarlos.

Otras mejoras podrían estar relacionadas con la calidad del algoritmo de combinación de polilíneas, más acciones de interacción como cortado de tubos, y mejoras visuales como texturado de tubos o *shaders*.

2.3. Requisitos específicos

2.3.1. Páginas

La aplicación se dividirá en dos páginas web; estas consistirán en ficheros HTML con CSS, para definir un estilo de formato, y con código JavaScript, para la ejecución de funciones, que deberán ser alojadas en un servidor.

Main.html

Interfaz principal, con el nombre de la aplicación y autor, desde la que el usuario deberá introducir una URL con un fichero VTK y seleccionar parámetros.

Los parámetros principales serán:

- *Modo de visualización*: para VR, móvil, ordenador, etc.
- *Color de fondo*.
- *Nivel de combinación*: cuántas polilíneas deberán ser combinadas.
- *Simplificación*: eliminar parte de los datos, manteniendo coherencia, para incrementar el rendimiento.
- *Radio 1*: radio de tubos para las polilíneas originales.
- *Radio 2*: radio de tubos para las polilíneas combinadas.

Deberá de haber además parámetros específicos para VR, como:

- *Espaciado de ojos*: distancia entre ambas cámaras.
- *Distorsión*: nivel de distorsión de curvatura de la cámara.

Se podrá considerar añadir nuevos parámetros futuros según se considere.

Cada parámetro podrá ser introducido en una caja (o seleccionado de una lista de valores prefijados), la cual tendrá al lado el nombre del parámetro y un valor por defecto asignado. Se podrá poner un valor por defecto de URL de datos, de modo que el usuario pueda probar la aplicación inmediatamente.

Sería deseable colocar al lado de cada caja un ícono de ayuda, que al ser presionado muestre en una ventana emergente informando al usuario sobre el significado de cada parámetro, y a ser posible con imágenes del efecto del parámetro.

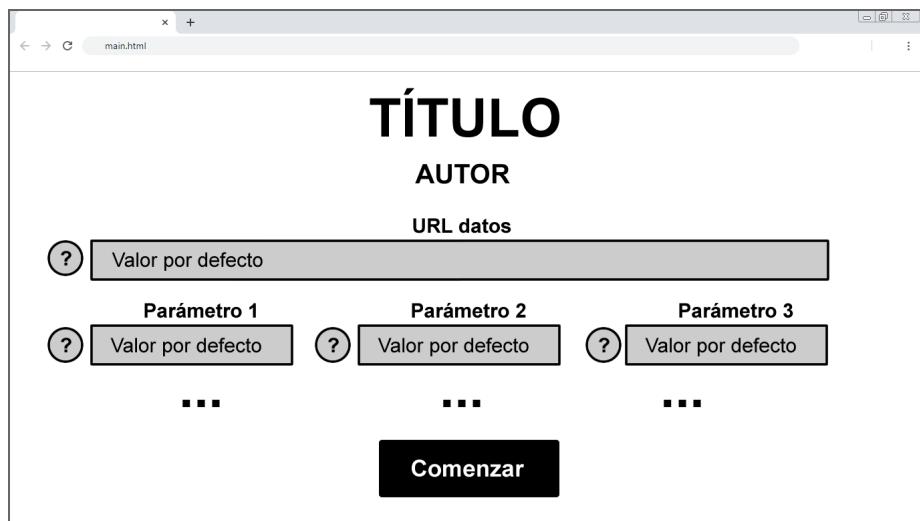


Figura 2.1: Boceto de la interfaz principal.

Index.html

Ejecución del código de aplicación para la visualización e interacción de datos en un *container* incrustado en el HTML.

Mientras se realiza el cargado de datos, preprocessamiento e inicialización de la visualización, se mostrará un mensaje de espera. Este se mostrará en otro contenedor encima del contenedor principal de visualización.

Una vez los datos estén preparados para ser mostrados, se ocultará el contenedor con el mensaje de espera.

El contenedor principal dispondrá de un *widget* con opciones para la interacción y modificación de la visualización. Este podrá ser ocultado en la visualización para VR, puesto que resultará un obstáculo visual considerable y se podrá optar por usar botones de controles físicos.

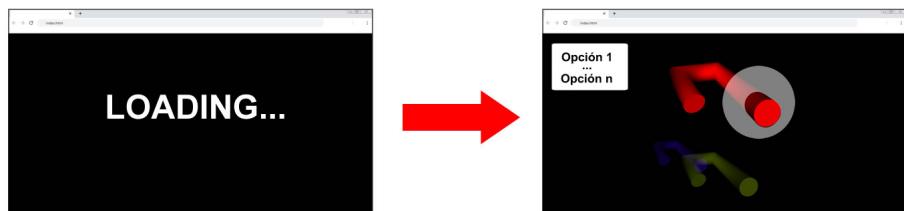


Figura 2.2: Boceto de la visualización de datos.

2.3.2. Funciones

Se discutirá a continuación las funciones que deberá implementar la aplicación con más detalle. Estas podrán ser ejecutadas de forma transparente al usuario o no.

- *Elegir parámetros y modos de visualización:* Una vez el usuario haya elegido los diferentes parámetros comentados anteriormente, será *main.html* responsable de cargar la página *index.html*, pasándole cada parámetro con su valor.
- *Cargar datos de polilíneas en formato VTK:* Una vez en *index.html*, desde la URL del fichero se extraerá la información de cada polilínea y se guardará en una variable para su posterior uso.
- *Combinar polilíneas cercanas entre sí:* Con la información de cada polilínea disponible, se aplicará un algoritmo de combinación de polilíneas según el nivel especificado. Se tendrán por tanto dos conjuntos de datos: el de polilíneas originales y el de combinadas.
- *Generar tubos a partir de las polilíneas:* Por cada conjunto de datos, se aplicará un filtro de generador de tubos para cada polilínea con los radios correspondientes, generando otros dos conjuntos de datos listos para ser visualizados.
- *Visualizar los tubos:* Se mostrará inicialmente los grupos combinados, ocultando los originales. Cada grupo tendrá un color diferente para ayudar a diferenciarlo.

- *Moverse por el mundo:* se deberá considerar posibles formas de movimiento para cada tipo de dispositivo. En el caso de ordenadores personales se hará uso del teclado y ratón; en el caso de móviles y tablets, gestos en la pantalla táctil para moverse y posiblemente sensores como el giroscopio para orientarse; en el caso de realidad virtual, el HMD para orientación y controles para moverse.
- *Seleccionar tubos y visualizar diferentes grupos:* Mediante el uso de un selector, como se muestra en la figura 2.3, se seleccionarán los grupos deseados de tubos de polilíneas combinadas. Una vez seleccionados, se podrá elegir ocultar los no seleccionados o expandir los seleccionados para mostrar el grupo de tubos de las polilíneas originales.

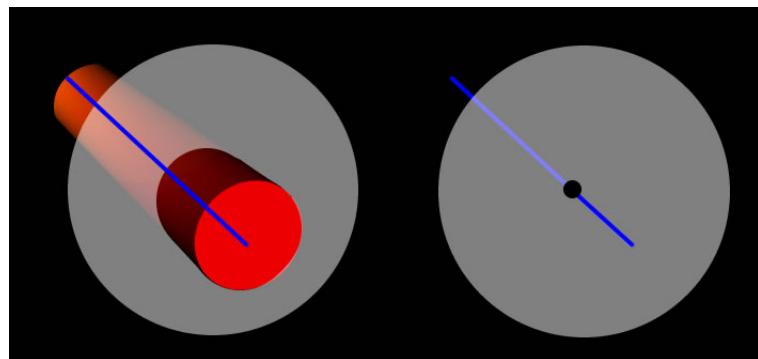


Figura 2.3: Uso del selector (círculo gris) para interacción. Izquierda: selección de un tubo (rojo), donde se aprecia su polilínea (azul). Derecha: punto de intersección con polilínea.

2.3.3. Requisitos de rendimiento

Se deberán tener en cuenta los siguientes aspectos:

Web

- *Velocidad y retardos:* Los retardos afectarán a la velocidad con que se descargan los datos: descargar datos por web será siempre más lento que accederlos directamente desde disco. En este punto son muy importantes las caché y el ancho de banda; cuanto mayor sea este último, menor será el tiempo de transmisión. En cuanto al retardo de propagación, cuanto más distante sea la localización física del servidor, mayor será el retardo. Esto no presenta un inconveniente, puesto que la aplicación se ejecuta desde el terminal del usuario, y no del servidor.

- *Almacenamiento:* Los datos deberán ser almacenados en el mismo servidor donde se encuentra la aplicación o en uno distinto. Se debe considerar cuánto espacio será necesario.
- *Disponibilidad:* La aplicación solo puede ser accedida por el usuario si el servidor se encuentra en funcionamiento. Se podría considerar disponer de un servidor secundario en caso de no estar disponible el primero. Se debe pensar además en un horario de mantenimiento de servidor.

Hardware del usuario

- *CPU:* Los datos deben ser preprocesados y enviados a la tarjeta gráfica ante cualquier cambio. Cuanto más potente sea el CPU menor será el tiempo de carga.
- *Tarjeta gráfica:* Cuanto más moderna y potente sea esta, mayor será la cantidad de datos a visualizar y con una mayor cantidad de FPS. Se ha de evitar posibles cuellos de botella por culpa del CPU.
- *Dispositivo:* Será necesario un dispositivo capaz de ejecutar la aplicación web (móvil, ordenador con casco de realidad virtual, ...).

Software del usuario

- *Navegador web:* Un navegador web actualizado será necesario; permitirá mayor compatibilidad con las últimas versiones de HTML y WebGL, ofreciendo una mejor experiencia.

2.3.4. Restricciones de diseño

El diseño estará mayormente restringido a las capacidades ofrecidas por VTK.js. Sin embargo, se pueden utilizar librerías adicionales de JavaScript que permitan extender sus capacidades.

Al disponerse del código fuente de VTK.js, se pueden añadir algunas funcionalidades en la medida de lo posible.

Se sigue el formato *Legacy VTK*, por lo que los tipos de datos estarán restringidos a este formato en concreto.

2.3.5. Atributos del sistema

- *Fiabilidad:* La fiabilidad del sistema dependerá de la instalación del servidor.

- *Mantenibilidad:* No se requiere de mantenibilidad.
- *Portabilidad:* Al ser una aplicación en JavaScript, ya se dispone en cierto modo de un sistema portátil. La ejecución del mismo en un dispositivo u otro dependerá de si disponen de un navegador web compatible.
- *Seguridad:* No serán necesarios mecanismos de seguridad. Se trata principalmente de una aplicación de visualización de datos; no se almacena ni comparte información privada. En caso de considerarse los datos de una resonancia magnética como privados, aunque no se pueda reconocer al sujeto, los datos pueden viajar encriptados de extremo a extremo mediante HTTPS.

Capítulo 3

Diseño

La aplicación se diseñará de manera modular. El módulo principal será el encargado de ejecutar la aplicación e invocar el código de otros módulos. El código de cada uno de ellos estará encapsulado en un fichero de JavaScript.

Los módulos son *funcionales*; cada uno dispone de un conjunto de funciones relacionadas.

3.1. Módulos

3.1.1. Módulo principal

Módulo principal de aplicación.

Sus tareas serán:

- Mostrar, y posteriormente ocultar, el mensaje de espera mientras se cargan y procesan los datos.
- Obtener parámetros de URL.
- Cargar los datos.
- Invocar funciones para la combinación de polilíneas y generación de tubos.
- Almacenar en distintos arrays las polilíneas, las polilíneas combinadas, y los actores asociados.
- Definir propiedades de ventana y cámara.
- Comunicación con dispositivos de interacción.

- Detectar en un bucle intersecciones del selector con tubos.

Requerirá del uso de todos los otros módulos.

3.1.2. Módulo de manipulación de líneas

Se encargará de la manipulación de polilíneas y generación de tubos. Dispondrá de funciones a ser llamadas por el módulo principal o por otras funciones del propio módulo.

Estas serán:

- Extracción de polilíneas de los datos, aplicando la decimación indicada por el usuario.
- Combinación de polilíneas según el nivel indicado por el usuario.
- Reducir densidad de puntos en polilíneas.
- Suavizar polilíneas.
- Generar los tubos y asociarlos a distintos actores.

Este módulo utilizará el de matemáticas, explicado más adelante, que permite pasar de radianes a grados y calcular distancias entre puntos.

3.1.3. Módulo de selector

Contendrá variables de estado del selector y funciones para manipularlo.

Estas serán:

Variables

- Actor asociado.
- Radio.
- Posición (tres parámetros, uno por cada eje XYZ).
- Rotación (tres parámetros, uno por cada eje XYZ).
- Orientación (tres parámetros, uno por cada eje XYZ).

Las variables de rotación se aplicarán al actor, mientras que las de orientación, deducibles de las primeras, se utilizarán por otros módulos en el cálculo de intersecciones con polilíneas.

Funciones

- Inicializar el selector, creando el actor asociado e inicializando sus variables.
- Cambiar la posición.
- Cambiar la rotación y calcular la nueva orientación.
- Cambiar el radio.

Este módulo utilizará el de matemáticas, al trabajar con vectores.

3.1.4. Módulo de matemáticas

Funciones matemáticas no existentes en la librería *Math* (nótese la primera letra en mayúsculas) de JavaScript.

Estas serán:

- Conversión de grados a radianes.
- Distancia entre dos puntos.
- Operaciones de vectores: suma, resta, producto vectorial, producto escalar, normalización, etc.
- Determinar intersección entre plano circular y línea.

3.1.5. Módulo misceláneo

Todas las demás funciones que no se relacionen con los módulos anteriores se implementarán en este.

Este módulo podrá ser usado por cualquiera de los módulos.

3.1.6. Diagrama

En la figura 3.1 se puede apreciar un diagrama del diseño de la aplicación en módulos, cada uno con sus funciones, y la dependencia de un módulo con otro mediante flechas.

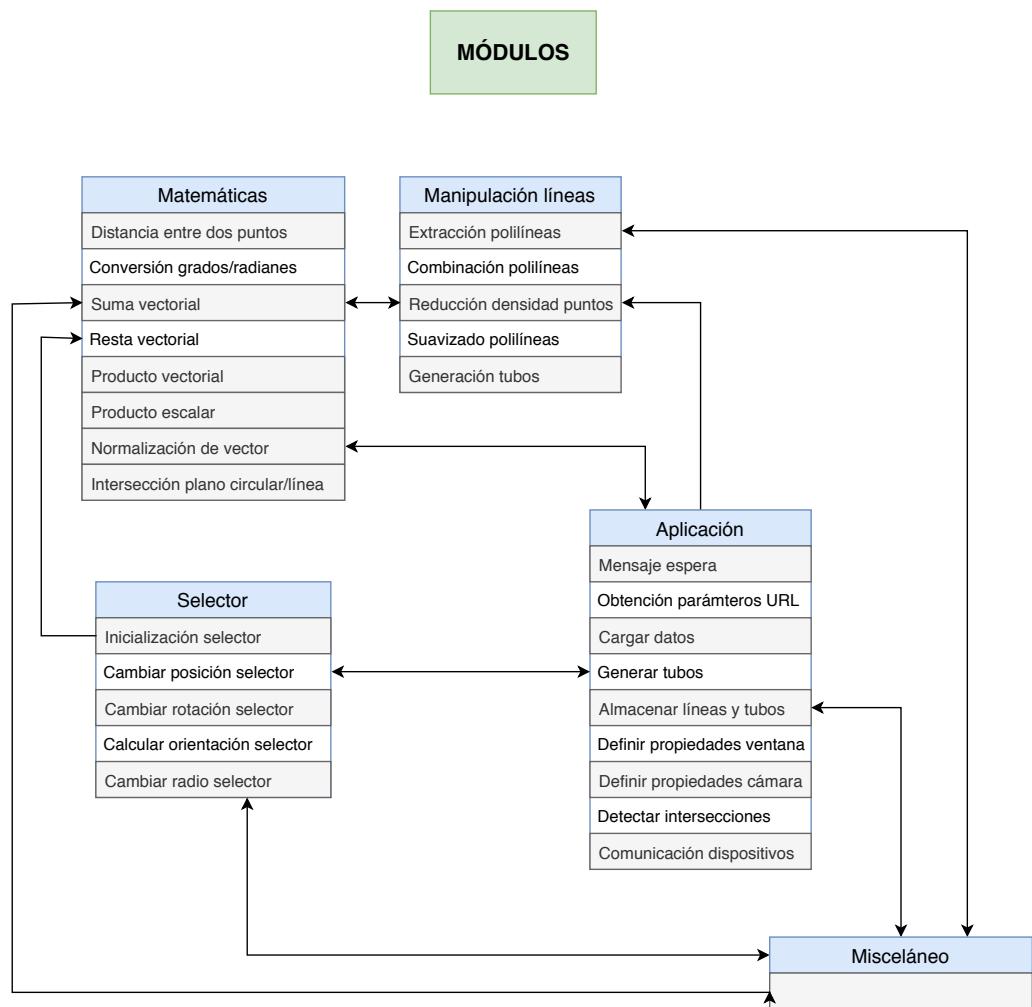


Figura 3.1: Estructura del software en módulos.

Capítulo 4

Implementación

4.1. Introducción

Con los requisitos del sistema vistos en el capítulo 2 y el diseño del capítulo 3, se explicará a continuación la implementación llevada a cabo para la construcción de la aplicación.

La aplicación se desarrolló en un portátil con Windows 7 (32-bit).

4.2. Requisitos previos

Se han de instalar varias librerías y establecer un entorno antes de comenzar a programar.

4.2.1. Librerías

VTK

La aplicación se ha implementado para su uso en web, utilizando la versión de JavaScript de la librería VTK (Visualization Toolkit).

VTK consiste en software de código abierto bajo licencia BSD que permite desarrollar con facilidad en múltiples lenguajes (C++, JavaScript, Python...) creación, procesamiento y renderización de datos visuales, además de interacción. Para ello se apoya en OpenGL.



Figura 4.1: Logo de VTK (Visualization Toolkit).

VTK.js, su versión en JavaScript, permite desarrollar aplicaciones que pueden ser ejecutadas desde un navegador web, ahorrando así al usuario la necesidad de instalar módulos o programas.

Se puede encontrar más información en su página oficial:

<https://kitware.github.io/vtk-js/api/index.html>.

Node.js

Node.js permite la creación de servidores web y herramientas de red usando JavaScript y módulos que manejan varias funcionalidades, como manipulación de datos binarios, uso y control de redes, etc.

JavaScript es el único lenguaje que admite *Node.js*, y su instalación es requerida por VTK.js.

Git Bash

Con el fin de probar la aplicación en diferentes dispositivos, se creó un repositorio *Git* a través del servicio *BitBucket*, que permite subir los contenidos del repositorio a su servidor con la aplicación *Git Bash*.

Se configuró el repositorio para tratar los ficheros HTML como páginas web. De esta manera el repositorio se convierte en un servidor web.

Aunque cada cuenta de *BitBucket* ofrece a cada usuario un límite de 2GB de espacio, no existe un límite en el tamaño de cada fichero individual, lo cual resulta ideal para subir ficheros de conjuntos de datos.

Durante la realización del trabajo, se utilizó *Git Bash* para compilar la aplicación mediante el comando

```
npm run build
```

y subir al repositorio con los comandos

```
git add .
git commit -m 'comentario'
git push
```

En la figura 4.2 se muestra una captura del terminal *Git Bash* durante una compilación de la aplicación.

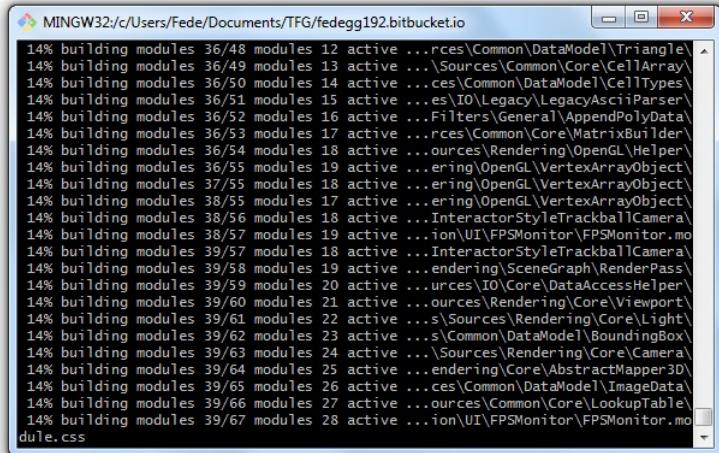


Figura 4.2: Terminal Git Bash durante compilación.

4.3. Datos

Como se ha comentado en la introducción, los tractos cerebrales se representan como un conjunto de *polilíneas*, es decir, una sucesión de vértices 3D conectados.

Los datos utilizados por el programa siguen el formato *Legacy VTK*, explicado en el siguiente apartado.

Durante la implementación se han utilizado los conjuntos de datos explicados en el apartado 5.3 para probar la correcta ejecución del programa.

4.3.1. Formato VTK

Una fichero de tipo VTK consiste en un fichero en formato de texto plano con los siguientes apartados:

1. Tipo de fichero.
2. Formato de texto.
3. Tipo de datos.
4. Número de vértices y tipo.
5. Por cada vértice, tres valores del tipo indicado anteriormente, representando la posición XYZ del vértice.
6. Número de líneas y número total de vértices.
7. Por cada línea, número de vértices de la línea y los índices de los vértices.

Se permiten comentarios, que deben de comenzar con el carácter #.

Se muestra a continuación un ejemplo del formato VTK, consistente en dos polilíneas de dos vértices cada uno.

```
# vtk DataFile Version 3.0 vtk output
ASCII
DATASET POLYDATA
POINTS 4 float
0.5 0.5 -1.5
0.5 0.5 1.5
1.5 0.5 -1.5
1.5 0.5 1.5
LINES 2 6
2 0 1
2 2 3
```

Los datos han de ser normalizados entre el rango [-1, 1]; es decir, cualquier vértice v_i debe cumplir:

$$v_i \in [(-1, -1, -1), (1, 1, 1)]$$

Esto es así puesto que el selector de interacción, explicado más adelante, tiene un tamaño variable comprendido en [0, 1]. Si los datos no se encuentran normalizados, puede ocurrir que el selector no consiga seleccionarlos por ser demasiado pequeño.

4.4. Aplicación

La aplicación consiste principalmente de dos páginas web, que no son nada más que ficheros HTML con diferentes elementos incrustados para definir funcionalidades y estilos de visualización.

En la primera se introducen parámetros y una dirección URL con la ubicación de los datos a visualizar, mientras que en la segunda se visualizan e interactúa con ellos.

4.4.1. Entorno

Dentro de la carpeta principal del proyecto se encuentran los siguientes archivos y carpetas:

- **data** carpeta con ficheros de datos VTK subidos a la web.
 - **corpuscallosum.vtk**
 - **fibres.vtk**
- **dist** carpeta con páginas web y JavaScript compilado.
 - **back.jpg**
 - **index.html**
 - **loader.css**
 - **main.html**
 - **MyWebApp.js**
- **node_modules** carpeta con módulos de *Node.js*.
- **src** carpeta con código fuente.
 - **controlPanel.html**
 - **index.js**
 - **lineFunctions.js**
 - **math.js**
 - **misc.js**
 - **selector.js**
- **package.json**
- **package-lock.json**
- **webpack.config.js**

4.4.2. Página interfaz principal

main.html, mediante código HTML y CSS, define la interfaz principal. En la parte superior muestra el título de la aplicación, el nombre del autor y fecha de creación, mientras que en la inferior diversas cajas donde introducir los valores de los parámetros; en caso de que el usuario no introduzca alguno, se utilizan valores por defecto. Cada caja muestra sus valores por defecto en color gris.

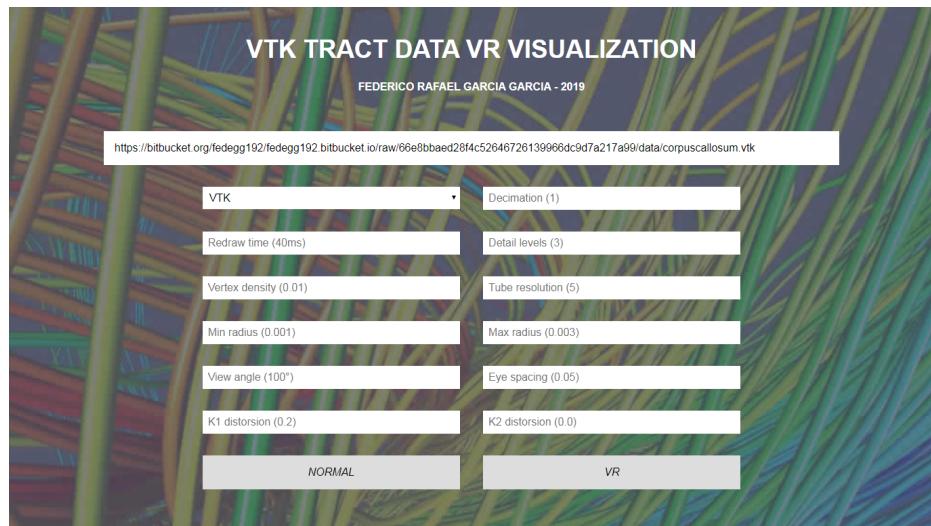


Figura 4.3: Página principal de la aplicación.

Los parámetros son los siguientes:

1. *URL*: dirección del fichero de polilíneas. Por defecto contiene una URL del fichero *corpus callosum*, subido al repositorio de *BitBucket*.
2. *Detail levels*: nivel de reducción de detalle. Indica cuántas veces se aplicará el algoritmo de combinación de polilíneas. Por defecto 3.
3. *Color*: selector para establecer el color de fondo de la aplicación.
4. *Decimation*: la inversa del porcentaje de decimación aplicado a las polilíneas originales. 1.0 por defecto (no hay decimación).
5. *Min radius*: el radio de los tubos de las polilíneas originales. 0.001 por defecto.
6. *Max radius*: el radio de los tubos de las polilíneas combinadas. 0.003 por defecto.

7. *View angle*: ángulo de visión de la cámara, en grados. Por defecto 100.
8. *K1 distorsion*: usado para calcular la distorsión radial de la cámara. Por defecto 0.2.
9. *K2 distorsion*: usado para calcular la distorsión radial de la cámara. Por defecto 0.0.
10. *Eye spacing*: en modo VR, separación lateral entre las dos cámaras. Por defecto (0.05).
11. *Modo*: tiene dos posibles valores para indicar el tipo de visualización e interacción: *normal* y *vr*. *normal* permite interacción tecladoretón en ordenador y táctilsensores en móvil; *vr* permite interacción en realidad virtual.

Se dispone de dos botones para comenzar la visualización e interacción. Presionar cualquiera de ellos cargará la página *index.html*, pasando los parámetros como variables de URL. Los nombres de las variables se colocan después de la URL tras el símbolo “?”; cada variable se separa de las demás con el símbolo “&”, y se le asigna el valor introducido por el usuario con “=”:

```
index.html?variable1=valor1&...&variableN=valorN
```

El botón etiquetado como *Normal* pondrá el valor de la variable *Modo* a *normal*, mientras que el etiquetado como *VR* la pondrá a *vr*.

El código del fichero *main.html* consiste de las siguientes partes:

1. *head*: define un meta-valor para detectar si el móvil, en caso de usarse, puede utilizar pantalla completa.
2. *style*: código CSS especificando la visualización los elementos HTML y la imagen de fondo, *back.jpg*.
3. *body*: conjunto de elementos *div* para establecer la organización de los demás elementos y un *form* compuesto de las cajas de introducción de parámetros, *input*, con la acción de abrir la página especificada.

4.4.3. Página aplicación

index.html, mediante código HTML, incrusta el código en Javascript compilado, *MyWebApp.js*. Crea un elemento de tipo *canvas* donde se ejecuta dicho código.

4.5. VTK.js

VTK.js consta de múltiples clases que pueden ser importadas con la palabra clave `import`. El nombre de cada uno de estas clases comienza con la palabra “`vtk`”. Algunas se pueden instanciar, mientras que otras disponen de funciones estáticas.

En el siguiente enlace sobre la API de VTK.js se puede encontrar información de cada clase junto con su código fuente:

<https://kitware.github.io/vtk-js/api/index.html>

A continuación se explican las clases utilizadas por la aplicación.

- *vtkPolyData*: conjunto de datos poligonales de diverso tipo: vértices, polilíneas o triángulos. Define su topología, color y normales.
- *vtkActor*: objeto en el espacio 3D al que se le asocia un objeto *vtkMapper*. Un actor se puede mover, rotar y cambiar de escala. También se pueden cambiar sus propiedades visuales, como brillo y opacidad. Representa los datos gráficos del *vtkMapper* asociado relativos a su posición, rotación y escala.
- *vtkMapper*: consiste en una interfaz entre datos de un objeto *vtkPolyData* con primitivas gráficas, que son transferidas a un objeto *vtkActor* para su visualización.
- *vtkDeviceOrientationToCamera*: permite detectar dispositivos con giroscopio para establecer la orientación de la cámara.
- *vtkForwardPass*: funciones relacionadas con la tubería de renderizado.
- *vtkRadialDistortionPass*: implementa un algoritmo de *warping*: deforma la perspectiva de imágenes dándoles una apariencia esférica.
- *vtkFullScreenRenderWindow*: define un contenedor con la vista completa de la aplicación, incluyendo *widgets*, canvas OpenGL donde se renderizan los gráficos, y *listeners* para determinar cambios o interacción con la pantalla, pulsación de teclas y movimiento de ratón.
- *vtkRenderer*: mediante una cámara define un *viewport*, es decir, una región de la escena a visualizar. Define además qué actores deben ser dibujados.
- *vtkRenderWindow*: define una ventana donde añadir uno o varios *vtkRenderer*.
- *vtkURLExtract*: permite extraer las variables URL del navegador.

- *vtkPolyDataReader*: clase que permite leer la información de un fichero VTK y pasárselos a un objeto *vtkPolyData*.
- *vtkTubeFilter*: generador de tubos. Dado un objeto *vtkPolyData* representando polilíneas, genera vértices alrededor del trayecto de cada polilínea según sus normales. Si no dispone de normales, las genera automáticamente; la generación de tubos no es perfecta, como se verá en la sección de Problemas del capítulo 6. Se puede especificar el número de lados y radio. Los datos poligonales de los diversos tubos generados pueden ser enviados a un *vtkMapper*.
- *vtkdataArray*: define un tipo de objeto array usado por VTK. Se le puede asociar un array de JavaScript, un nombre, y el número de componentes que compone cada elemento.
- *VtkDataTypes*: define tipos primitivos de datos, como enteros y números en coma flotante, usados por VTK.
- *vtkPoints*: define un vértice 3D usado por *vtkPolyData*.
- *vtkCylinderSource*: define la información geométrica de un cilindro, pudiendo especificar su altura, radio, y número de lados. Los datos poligonales pueden ser enviados a un *vtkMapper*.

4.6. Código fuente

Se explicará a continuación la implementación en código JavaScript de los módulos definidos en el capítulo 3. Cada uno de ellos se encuentra encapsulado en un fichero.

Todos, salvo el módulo principal, definen funciones y variables globales. Otros módulos hacen uso de ellas de forma estática. Para ello es necesario incluir la palabra clave `export` antes de la declaración de cada función y variable.

4.6.1. Módulo misceláneo: misc.js

Contiene dos funciones:

setShiny(actor)

Dado un actor de VTK, accede a sus propiedades visuales, modificando los parámetros de iluminación ambiente a 0.3, difusa a 1, poder especular a 40, nivel especular 1 y color especular 1 (blanco). Con ello se consigue que el actor tenga cierto efecto de reflejo.

setTransparent(bool, actor, opacity)

Dado un actor de VTK, accede a sus propiedades visuales, modificando el parámetro de opacidad al deseado.

Puede establecer su opacidad directamente a 1 con el parámetro *bool*.

4.6.2. Módulo matemático: js

Contiene las siguientes funciones:

deg2rad(x)

Dado un *x* número en grados lo convierte a radianes.

vdot(a, b)

Dados dos arrays representando vectores 3D, devuelve el producto vectorial.

vadd(a, b)

Dados dos arrays representando vectores 3D, devuelve el array suma.

vsub(a, b)

Dados dos arrays representando vectores 3D, devuelve el array resta.

vtimes(a, b)

Dados un array representando un vector 3D y un escalar, devuelve el vector multiplicado por el escalar.

vdistance(a, b)

Dados dos arrays representando puntos 3D, devuelve la distancia euclídea entre ellos.

vnorm(a)

Dado un array representando un vector 3D, lo devuelve normalizado: calcula la magnitud del vector y divide cada uno de sus tres componentes

ente la magnitud.

distance(x1, y1, z1, x2, y2, z2)

Dados 6 valores que representan dos puntos 3D, devuelve la distancia euclídea entre ellos.

intersectionLine(v0, n, p0, p1, r

Permite detectar la intersección de una línea con un plano circular.

Recibe como entrada los siguientes arrays de tres componentes: v_0 (posición del plano), n (normal del plano), p_0 (punto de inicio de la línea), p_1 (punto de final de la línea). Recibe además un *float*, r (radio del plano).

Si el producto vectorial de n con la diferencia entre p_1 y p_0 es cero, el plano es paralelo a la línea y no interseccionan. Por problemas de aproximación, en lugar de comprobar la igualdad a cero, se comprueba si es menor a 0.0001.

En caso contrario, mediante cálculos vectoriales se determina si hay intersección con el plano infinito. En caso positivo, se calcula el punto de intersección y se determina si su distancia a v_0 es menor o igual al radio. En caso positivo existe intersección y se devuelve **true**; en caso contrario se devuelve **false**.

intersection(line, v0, n, r)

Permite detectar la intersección de una polilínea con un plano circular.

Recibe como entrada *line* (representa la polilínea como un array de objetos de tipo punto XYZ), v_0 (posición del plano), n y r (radio del plano).

Ejecuta la función **intersectionLine** para cada par de puntos consecutivos del array *line* en un bucle **for**.

Si **intersectionLine** es verdadero para alguna pareja, devuelve **true**; en caso contrario se devuelve **false**.

4.6.3. Módulo selector: selector.js

Define las variables y funciones necesarias para implementar el selector de interacción en la aplicación. Las variables son, como se explicó anteriormente, globales. De esta manera, cualquier módulo puede comprobar el estado del selector.

Estas son:

- *selectorRadius*: radio del cilindro.
- *cylinderSource*: objeto de tipo *vtkCylinderSource*.
- *actorCylinder*: objeto de tipo *vtkActor*.
- *mapperCylinder*: objeto de tipo *vtkMapper*.
- *nx, ny, nz*: normal del selector, es decir, su vector de orientación. Inicialmente $nx = 0, ny = 0, nz = 0$.
- *x, y, z*: posición del selector. Inicialmente $nx = 0, ny = 1, nz = 0$; el selector está orientado hacia arriba.
- *rx, ry, rz*: rotación del selector, en grados. Inicialmente $rx = 0, ry = 0, rz = 0$.

Sus funciones son:

createSelector(radius)

Recibe como parámetro un radio; este se guarda en la variable *selectorRadius*. Se crea un objeto de tipo cilindro de altura 0.001, para que parezca un plano, número de lados igual a 200 y el radio indicado.

Posteriormente se definen las propiedades visuales del actor (color blanco y transparencia igual a 0.5).

rotateSelector()

Se llama a la función *setRotation* de *actorCylinder*, pasándole los valores de *rx, ry, rz*, para rotar al actor.

Luego se convierten *rx, ry, rz* a radianes y se guardan en variables locales.

Finalmente, usando las variables locales anteriores, se actualizan los valores de *nx, ny, nz* aplicando rotaciones según el orden establecido en VTK: primero eje Z, luego X y finalmente Y. Estas transformaciones se aplican al vector inicial de orientación (0, 1, 0).

$$\begin{bmatrix} \cos(rry) & 0 & \sin(rry) \\ 0 & 1 & 0 \\ -\sin(rry) & 0 & \cos(rry) \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(rrx) & -\sin(rrx) \\ 0 & \sin(rrx) & \cos(rrx) \end{bmatrix} \begin{bmatrix} \cos(rrz) & -\sin(rrz) & 0 \\ \sin(rrz) & \cos(rrz) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Las variables quedan como:

```

nx = -cos(rry)*sin(rrz)+sin(rrx)*sin(rry)*cos(rrz);

ny =  cos(rrx)*cos(rrz);

nz =  sin(rry)*sin(rrz)+cos(rry)*sin(rrx)*cos(rrz);

selectorUI(renderWindow)

```

Recibe como parámetro un objeto *vtkRenderWindow* y añade un *listener* a cada elemento del *widget* interacción. Si uno es modificado, el selector guarda en sus variables el nuevo valor y ejecuta funciones de rotación o cambio de posición.

Finalmente, llama a la función `renderWindow` de *vtkRenderWindow* para refrescar la pantalla con los cambios.

4.6.4. Módulo de líneas: `lineFunctions.js`

Las funciones de este módulo devuelven valores por referencia. Dado que deben devolverse arrays, pueden producirse errores de referencias. Los arrays creados dentro de la función serán eliminados al terminar esta; si son devueltos, devolverán una referencia nula. Por ello, uno de los parámetros de las funciones que se mencionarán a continuación es el array de salida, modificado dentro de la propia función.

`getLines(polydata, decimation, lines)`

Extrae de *polydata*, objeto de tipo *vtkPolyData*, sus *cells* (polilíneas). En un bucle *for*, se recorre cada polilínea. Se determina el número de vértices a conservar dada *decimation*, multiplicando este parámetro con la longitud de la polilínea. A partir de ahí se determina *pointsPerLine*: cada cuántos vértices recorridos se debe conservar el actual. Se crea un array vacío llamado *linea*, en el que se añaden vértices de la siguiente manera: en un bucle *for* se añaden los vértices avanzando saltándose el número indicado por *pointsPerLine*. El array *linea* es añadido al array de salida *lines*.

`combineLines(lines, linesCombined, level)`

El algoritmo de combinación de polilíneas. Se basa en una búsqueda de parejas *greedy* en base a la distancia de tres de sus vértices.

Dado el array de *lines* de la función anterior, se recorre este obteniendo de cada línea sus vértices del principio, medio y final. Estos son almacenados

en un array local llamado *linesMainPoints*; por lo tanto, este array está formado por líneas de tres vértices.

Luego se crean dos arrays de índices, *availableIds* y *pairs*. El primero está formado por enteros que van desde 0 hasta el número de polilíneas. El segundo, inicialmente vacío, contendrá varios arrays; cada array almacenará los índices de una pareja de líneas a combinar.

En un bucle *while*, mientras se cumpla la condición de que *availableIds* no es vacío, se extrae el primer índice y almacena en dos variables, *id₁* e *id₂*. Otra variable, *diff*, inicialmente igual a un valor muy alto (100000), indica el grado de diferencia entre dos líneas de tres vértices; el grado de diferencia se calcula como la media de las distancias euclídeas del vértice *v_i* de la primera línea con el vértice *u_i* de la segunda línea.

Cada línea se compara con el resto de líneas no emparejadas todavía, y se guarda en cada iteración el índice de la línea con la que el grado de diferencia fue el mínimo. Con esto se pretende combinar líneas similares, puesto que comienzan y terminan en posiciones similares (se ha decidió añadir el vértice del medio para determinar mejor las similitudes).

Puede ocurrir que dos líneas con el menor grado de diferencia no sean similares: para ello se comprueba si para el nivel de detalle actual, *level*, el grado de diferencia es menor que *level* multiplicado por 0.1. Al ser un valor bajo, se consideran similares y se pueden combinar, ya que las distancias entre sus respectivos vértices son mínimas. Se multiplica por el valor *level* dado que al realizar sucesivas combinaciones, las polilíneas comenzarán a distanciarse unas de otras, y será necesario permitir que polilíneas más distantes puedan combinarse.

Si el grado de diferencia es muy alto, se iguala *id₁* a *id₂*. Si estos dos índices son iguales, la polilínea se combinará consigo misma, manteniéndose la original. Finalmente se añade la pareja *id₁* a *id₂* a *pairs*.

Una vez cada línea está emparejada con otra, se recorre cada pareja del array *pairs*. Se obtiene cada línea de cada pareja, y se recorre en un doble bucle *for* cada vértice de ambas líneas. Se determina, para cada vértice de la primera línea, el más cercano en la segunda; puede ocurrir que varios vértices compartan el mismo más cercano. Se promedian ambos vértices y se guardan en un array local llamado *newLine*. *newLine* es la polilínea combinada, y cada *newline* es añadido al array de salida *linesCombined*.

undenseLines(lines, density, linesUndensed)

Esta función recorre cada línea *l* del array *lines* en un bucle *for*, creando un array vacío llamado *newLine*, en el que no se añaden vértices de *l* si la distancia euclídea de un vértice con el siguiente es menor a *density*.

Cada *newLine* es añadido al array de salida *linesUndensed*.

smoothLines

Esta función no se usa en la aplicación, aunque sí se ha probado.

Recorre cada vértice de un array de líneas suavizándolo con un kernel gaussiano de tamaño 3. Es decir, cada eje de posición de un vértice v_i se promedia con el vértice anterior v_{i-1} y el siguiente v_{i+1} , siendo el nuevo vértice $v = 0.107973v_{i-1} + 0.468592v_i + 0.107973v_{i+1}$

Con ello se pretendía reducir un problema producido por el algoritmo de combinación: se producen regiones con una alta densidad de puntos ruidosos. Este problema al final fue solucionado por la función **undenseLines**. Se prefirió **undenseLines** ya que reduce el número de vértices (lo cual incrementa el rendimiento), es más rápido, y da mejores resultados.

createTubes(vertexArray, cellArray, g, t, radius, tubeResolution, actorTubes)

Esta función genera actores de tubos dados arrays de VTK de vértices y polilíneas, *vertexArray* y *cellArray*. El parámetro *g* es un entero que determina el índice de una polilínea dentro del conjunto completo, de tamaño *t*. Los parámetros *g* y *t* se usan para añadir a un array, *scalarsData*, el valor $g*(1.0/t)$ una cantidad de veces igual al número de vértices de *vertexArray*. De esta manera, cada polilínea del conjunto tiene asignado a sus vértices un valor único comprendido entre 0 y 1.

Se crea un objeto de tipo *vtkTubeFilter*, y se le da la orden de generar tubos, pasándole para ello el array *scalarsData* para definir los colores de vértices de los tubos, el radio según *radius*, y el número de lados según *tubeResolution*. Se extraen los datos poligonales, se pasan a un objeto *vtkMapper*, y a este se le asocia el actor de salida *actorTubes*.

getLinesAndTubes(polydataFull, lines, linesCombined, actorsTubesCombined, actorsTubesOriginal, actorsLines, detailLevels, vertexDensity, decimation, minRadius, maxRadius, tubeResolution)

Esta función hace uso de todas las funciones del módulo de líneas mencionadas anteriormente.

Llama a **getLines** para obtener el conjunto de líneas de *polydataFull* y almacenarlas en el array de salida *lines*.

Luego llama en un bucle a **combineLines** un número de veces igual a *detailLevels*, combinando así sucesivamente polilíneas; el último conjunto

de polilíneas combinadas se almacena en el array de salida *linesCombined*.

Después elimina radio de las líneas de *linesCombined* llamando a `undenseLines`.

Finalmente, crea un actor tubo por cada polilínea original y combinada. Se crean también actores de líneas de las polilíneas originales. Todos estos actores son almacenados en los arrays de salida *actorsTubesCombined*, *actorsTubesOriginal*, y *actorsLines*.

4.6.5. Módulo principal: index.js

Este fichero es el responsable de crear todo lo necesario para la ejecución de la aplicación.

Se obtienen los parámetros de usuario de la URL llamando a `vtkURLExtract.extractURLParameters()`. Estos se analizan, determinando si son correctos, puesto que algunos deben ser números enteros y otros en coma flotante. Para ello se utilizan dos funciones propias, `isNumber` (devuelve `true` si se trata de un número entero o coma flotante) e `isInteger` (devuelve `true` si se trata de un número entero). En caso de negativo, se utilizan los parámetros por defecto.

Luego se definen dos variables, *opacity*, igual a 0.1, y el array de valores booleanos inicialmente vacío *intersections*. Su uso se verá más adelante.

A través de *queries* al documento, se obtiene el elemento HTML *body*, para poder injectar al mismo código HTML con el mensaje y animación de espera.

Mientras se muestra el mensaje de espera, se sigue ejecutando código de fondo. Se crea un objeto *vtkPolyDataReader* y se ejecuta su función estática `setUrl`, pasándole como argumento la dirección URL del fichero VTK. Con el comando `then` se especifica que se ejecute el código siguiente una vez haya terminado de cargar los datos.

Se ejecuta la función `getLinesAndTubes`, guardando las líneas y actores de tubos en diferentes arrays.

Después se eliminan todos los elementos HTML hijo del contenedor principal, consiguiendo así quitar el mensaje de espera.

Se configuran propiedades de cámara para que esta se ubique en el origen, mirando en dirección negativa del eje Z con *viewUp* igual (0, 1, 0).

A continuación se crea el selector con un radio igual a 0.1, llamando para ello a `createSelector`.

En varios bucles se establecen los actores tubos de polilíneas originales y combinados con opacidad igual a *opacity*, llamando a `setTransparent`, y con efecto de reflejo, llamando a `setShiny`.

Se analiza el valor del parámetro URL *mode*. Si este es igual a *vr*, se crean dos cámaras y dos *vtkRenderer*, uno para cada mitad del *viewport* y separadas según el parámetro URL *eyeSpacing*. Se establece el color de fondo para ambos *vtkRenderer* y en un bucle se añaden los actores invocando a *addActor* para cada *vtkRenderer*. De *vtkRenderWindow* se elimina el *vtkRenderer* por defecto llamando a *removeRenderer*, y se añaden en su lugar los otros dos definidos anteriormente con *addRenderer*. Se crea un objeto de tipo *vtkRadialDistortionPass* que recibe los parámetros URL *distk1*, *distk2* y *eyeSpacing*. Este se añade al *pipeline* de OpenGL.

En caso de que *mode* no sea igual a *vr*, se añade al *vtkRenderer* por defecto los actores y se establece su color de fondo.

Se añade a *vtkFullScreenRenderWindow* el *widget* especificado en el fichero *controlPanel.html*; este define en HTML *sliders* para controlar la posición, orientación y radio del selector, la opacidad de los tubos no seleccionados, y los modos de visualización entre actores combinados u originales; para ello dispone del *checkbox* llamado *SimplifiedView*. El *widget* se puede ocultar con el *checkbox* llamado *Show*. En modo VR el *widget* se oculta por defecto. Dado que el *widget* es opaco por construcción, se modificó código fuente de VTK.js, en concreto la clase *vtkFullScreenRenderWindow*, para que el *widget* sea transparente.

Finalmente, en dos *queries* se comprueba el estado de los *checkbox*, para determinar si ocultar o no el *widget* y si mostrar los tubos combinados u originales. Se comprueba el array *intersections* para determinar qué actores añadir o quitar de los *vtkRenderer* según el modo de visualización, VR o normal. Los actores se añaden o quitan con un *timeout*, que permite especificar cuándo ejecutar código transcurridos los milisegundos indicados, en este caso tras *actorRedrawTime* milisegundos.

En otro *timeout* se comprueba tras cada milisegundo si han habido intersecciones entre el selector y tubos, cambiando sus niveles de transparencia a opacos si ha habido intersección y a transparente, según el nivel de *opacity*, en caso contrario.

Tras cada modificación visual, se ejecuta *render()*, para refrescar la pantalla.

4.7. Modo VR

El modo de VR, consistente en utilizar dos cámaras, ha permitido emular mediante software el uso de gafas de realidad virtual, al no haberse podido usar hardware real, como se explica en el capítulo 6.

En la figura 4.4 se puede ver cómo la escena se renderiza dos veces, con una distorsión de perspectiva esférica, para simular con mayor realidad la visión humana.

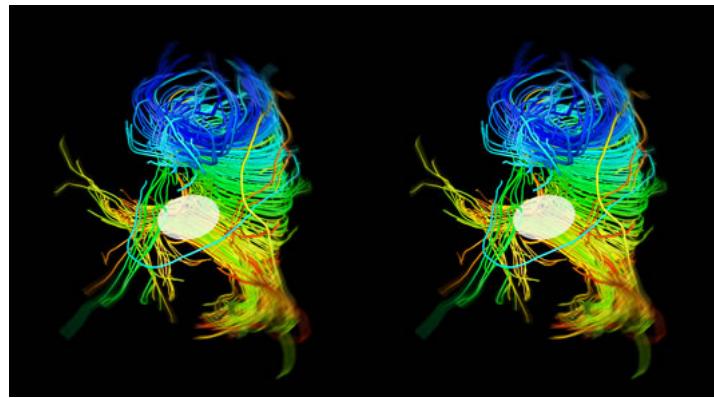


Figura 4.4: Simulación de VR mediante la aplicación.

Capítulo 5

Pruebas

5.1. Introducción

Este capítulo se dividirá en dos partes. En la primera se mostrarán capturas de la aplicación, mostrando las diferentes funcionalidades. En la segunda, se hará una evaluación del rendimiento bajo diferentes circunstancias.

5.2. Dispositivos

Para las capturas se ha utilizado un móvil de alta gama, *LG G7 One*, y un portátil *gaming*, *ASUS F555L*. Para la valoración del rendimiento se ha utilizado únicamente el portátil.

Las especificaciones de los dispositivos son las siguientes:

LG G7 One

- Sistema Operativo: Android 9.0.
- CPU: Qualcomm MSM8998 Snapdragon 835 (10 nm) Octa-core (4x2.45 GHz Kryo & 4x1.9 GHz Kryo).
- Memoria: 4GB.
- Tarjeta gráfica: Adreno 540
- Resolución de pantalla: 1440x3120 píxeles.
- Navegador web: Chrome 78.0.3904.96

ASUS

- Sistema Operativo: Windows 10 64-Bit.
- CPU: Intel i5 (2x2.4 GHz).
- Memoria: 4GB.
- Tarjeta gráfica: Nvidia GT920M.
- Resolución de pantalla: 1366x768 píxels.
- Navegador web: Chrome 78.0.3904.97

5.3. Conjuntos de datos

Los datos utilizados para las capturas son los siguientes:

5.3.1. Corpus callosum

El *corpus callosum* (*cuerpo calloso* en español) es el haz de fibras nerviosas más extenso del cerebro humano. Se encuentra ubicado en la *cisura interhemisférica*, la hendidura que divide longitudinalmente al cerebro en dos hemisferios cerebrales.

Este conjunto de datos cuenta con 554.060 vértices y 753 polilíneas.

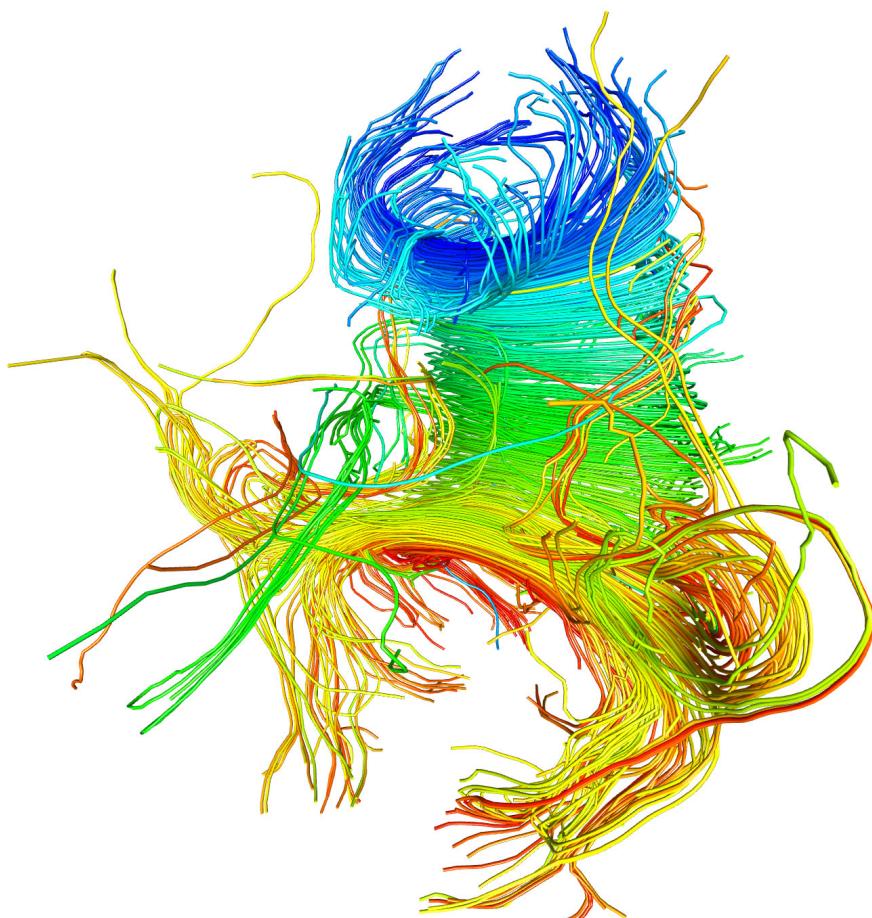


Figura 5.1: Renderización del cuerpo calloso utilizando la aplicación.

5.3.2. Fibres

Conjunto de datos ofrecido por la aplicación vISTe como muestra para probar la aplicación. Cuenta con 249.561 vértices y 364 polilíneas.

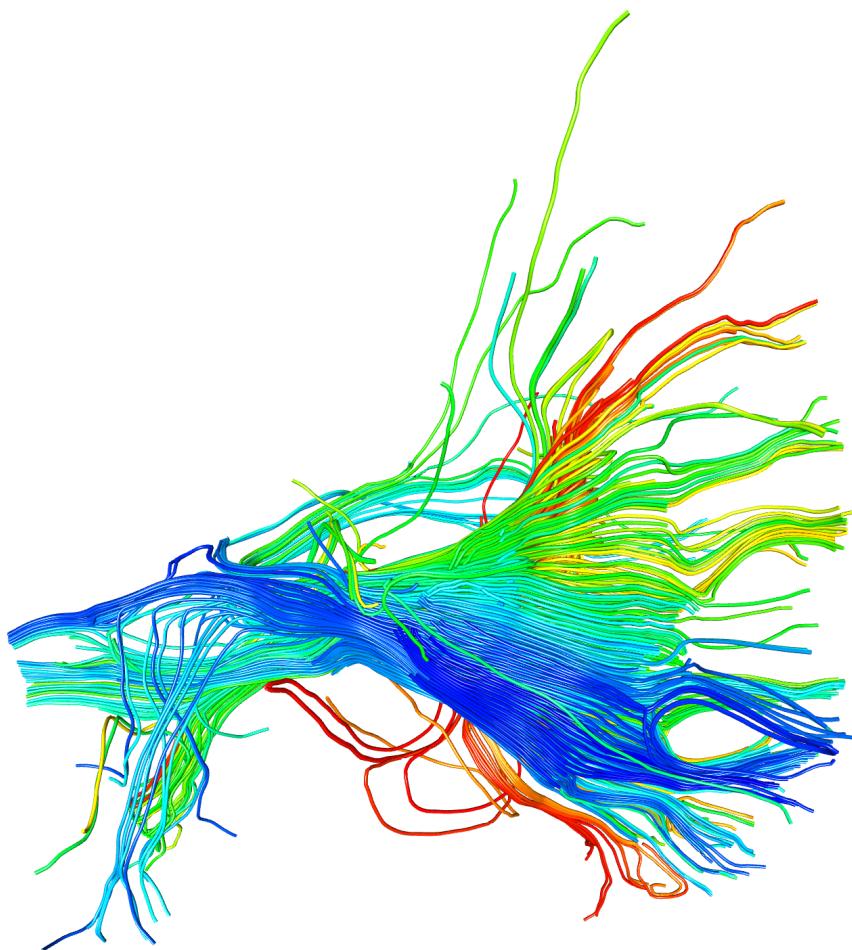


Figura 5.2: Renderización del conjunto de datos Fibres utilizando la aplicación.

5.4. Funcionamiento de la aplicación

En muchas de las capturas se ha ocultado el *widget* de interacción para apreciar mejor los resultados, deseleccionando la opción *Show*.

5.4.1. Interfaz

En la figura 5.3, se muestra la interfaz de la aplicación donde se seleccionan los parámetros y el mensaje de espera.



Figura 5.3: Izquierda: interfaz de la aplicación. Derecha: mensaje de espera.

5.4.2. Widget de interacción

En la figura 5.4, en la imagen izquierda, se puede observar el *widget* con las distintas opciones de interacción. Gracias a ser semitransparente, no se ocultan los datos. Con el fin de obtener una mejor visualización, se pueden esconder las opciones de interacción, como se muestra en la imagen derecha.

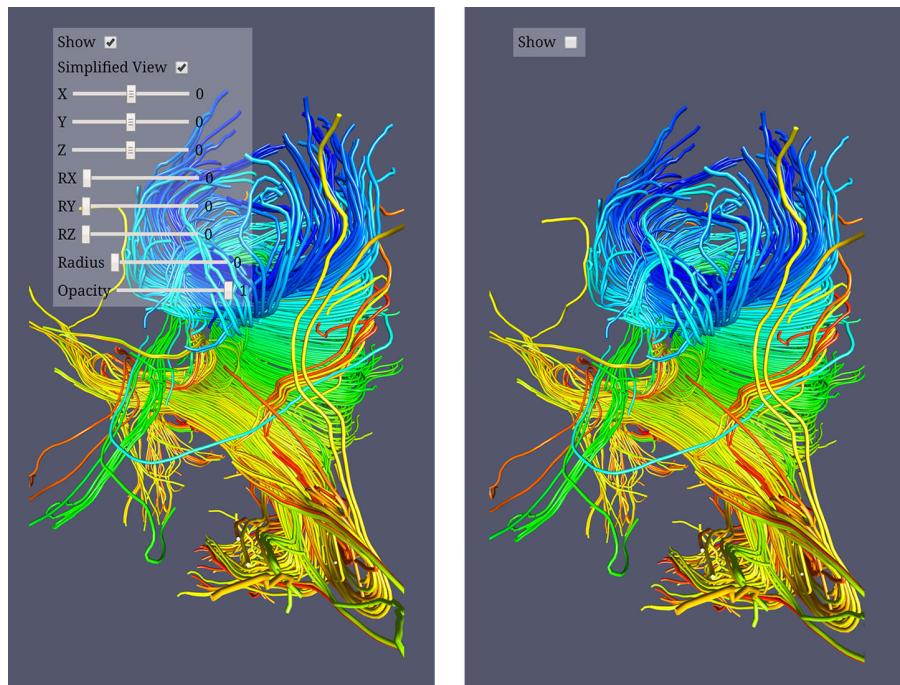


Figura 5.4: Izquierda: opciones del *widget* expandido. Derecha: opciones del *widget* ocultas.

5.4.3. Selección

En la figura 5.5 se puede ver, en la imagen de la izquierda, la selección mediante el selector de tres fibras combinadas de colores azul, verde y verde claro, con un nivel de detalle igual a 3, un grosor de tubo de 0.003 para los tubos combinados y de 0.001 para los originales, un número de lados igual a 5, y densidad igual a 0.001. Se ha ocultado el resto de fibras no seleccionadas estableciendo la opacidad a cero, a fin de apreciar mejor la selección.

A la derecha se pueden ver las fibras originales tras la desección de la opción *Simplified View*.

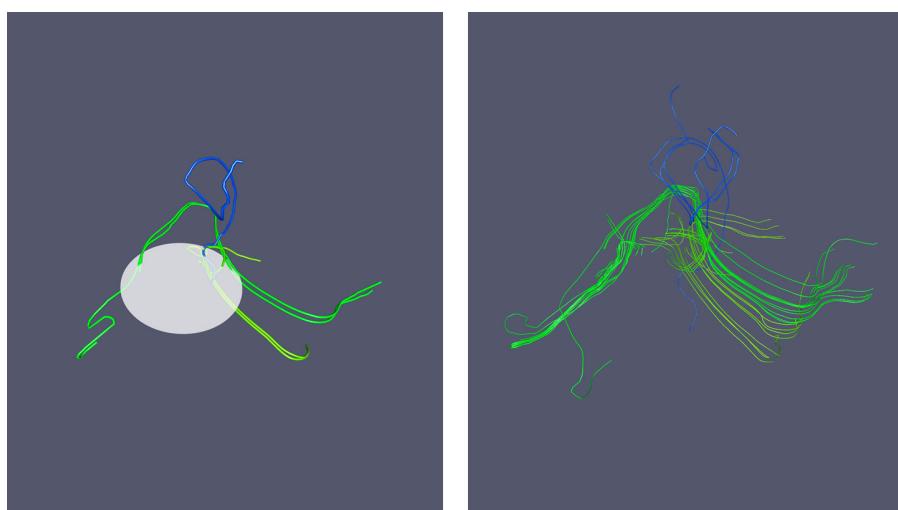


Figura 5.5: Izquierda: selección de un conjunto de fibras combinadas. Derecha: expansión.

5.4.4. Opacidad

En la figura 5.6 se muestran diferentes niveles de opacidad para los tubos no seleccionados.

En todos los casos se ha empleado el mismo nivel de detalle, 3, grosor de tubo, 0.003, número de lados, 5, y densidad, 0.001.

En la figura 5.6 se puede apreciar, en la imagen de la derecha, la existencia de oclusión: los tubos más cercanos a la cámara ocultan a los demás y a gran parte del selector. En las imágenes izquierda y central se ha reducido el nivel de opacidad a 0 y 0.068, respectivamente. Esto permite revelar completamente el selector y los tubos seleccionados: dos fibras de colores verde y verde claro.

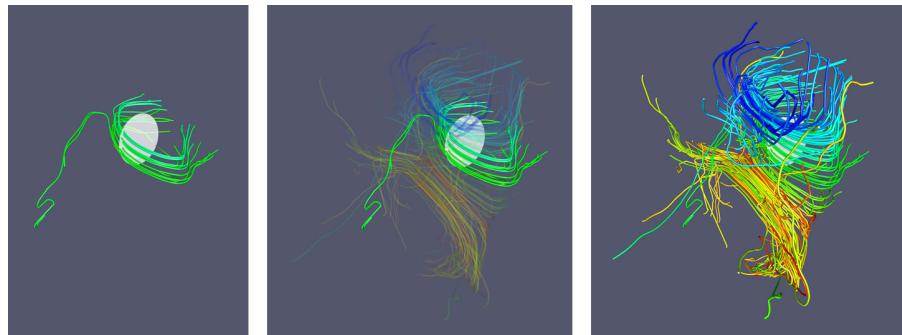


Figura 5.6: Izquierda: opacidad igual a 0. Centro: opacidad igual a 0.068. Derecha: opacidad igual a 1; ocurre oclusión.

5.4.5. Radio

En la figura 5.7 se muestran diferentes tamaños de radio para el selector.

En todos los casos se ha empleado el mismo nivel de detalle, 3, grosor de tubo, 0.003, n mero de lados, 5, opacidad 0.01, y densidad, 0.001.

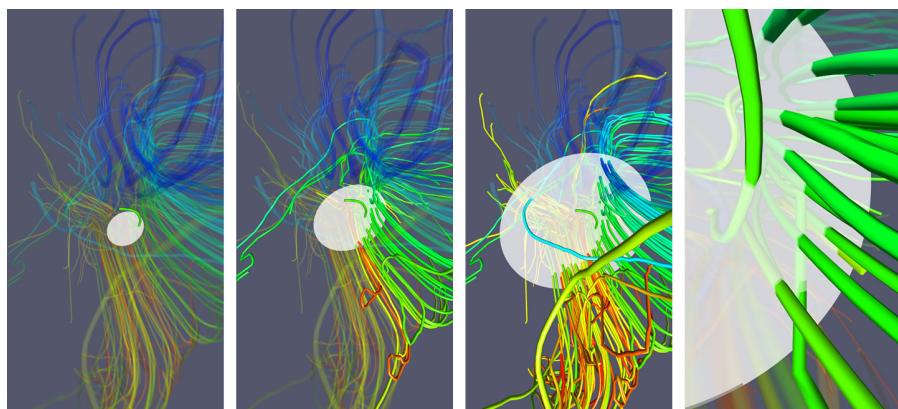


Figura 5.7: De izquierda a derecha: radio igual a 0.05; radio igual a 0.1; radio igual a 0.2; selector visto de cerca.

5.4.6. Nivel de detalle

En la figura 5.8 se muestran los resultados de aplicar diferentes niveles de detalle. Cuanto mayor sea este, más combinaciones de polilíneas se realizarán, resultando en un modelo más simple con menos tubos.

En caso de aplicar un nivel igual a 0, no se ejecuta el algoritmo de combinación; se muestran los datos originales sin ninguna alteración.

En todos los casos se ha empleado el mismo grosor de tubo, 0.003, número de lados, 5, y densidad, 0.001. Se ha establecido opacidad igual a 1 y el radio del selector a 0 para apreciar correctamente todos los tubos.

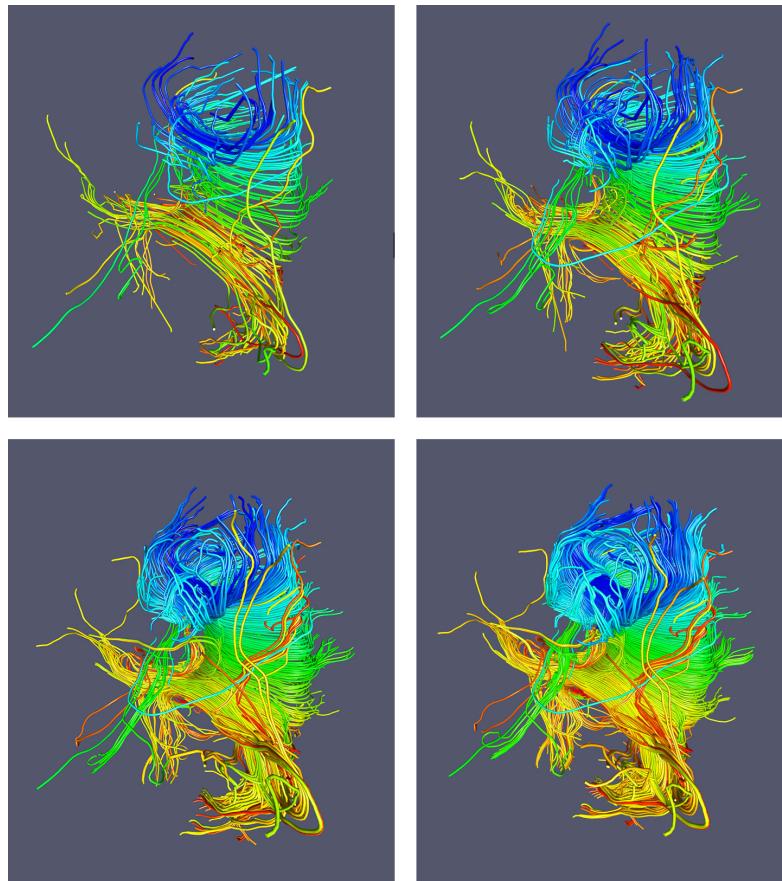


Figura 5.8: De izquierda a derecha, arriba a abajo: nivel de detalle igual a 3; nivel de detalle igual a 2; nivel de detalle igual a 1; nivel de detalle igual a 0.

5.4.7. Grosor de tubos

En la figura 5.9 se muestran los resultados de aplicar dos tamaños de grosor de tubo.

En ambos casos se ha empleado el mismo nivel de detalle, 0, número de lados, 5, y densidad, 0.001.

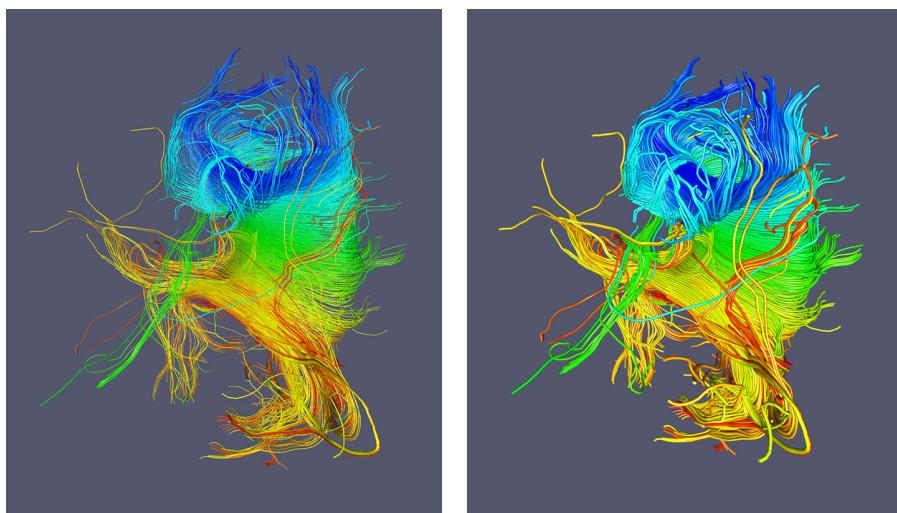


Figura 5.9: Izquierda: grosor de tubo igual a 0.001. Derecha: grosor de tubo igual a 0.003.

5.4.8. Decimación

En la figura 5.10 se muestran los resultados de aplicar decimación al conjunto original de polilíneas. Se aprecia la reducción en el número de vértices.

En ambos casos se ha empleado el mismo número de lados, 5.

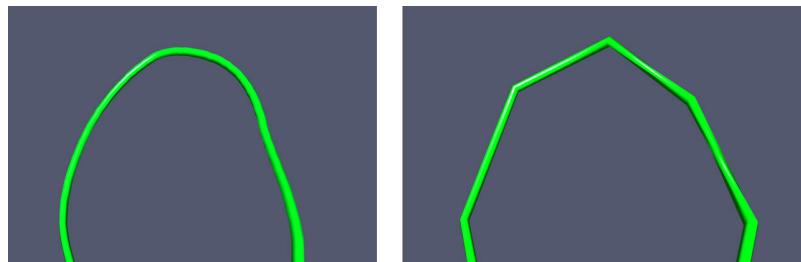


Figura 5.10: Izquierda: tubo original, sin decimación. Derecha: tubo con decimación igual a 0.05 (se ha reducido el 95 % de los vértices).

5.4.9. Renderización en el tiempo

Ante cualquier cambio en la renderización, como puede ser la opacidad o expansión de tubos, se actualiza cada tubo combinado, o grupo de tubos originales, uno tras otro en orden con un tiempo de intervalo especificado por el usuario.

En la figura 5.11 se muestra la evolución en el tiempo de renderización de tubos tras modificar la opacidad de los tubos no seleccionados (en este caso todos) desde 0 hasta 1; se ha puesto el radio del selector a 0 para apreciar mejor la evolución.

Se ha empleado el mismo nivel de detalle, 0, grosor de tubos, 0.003, y densidad, 0.001.

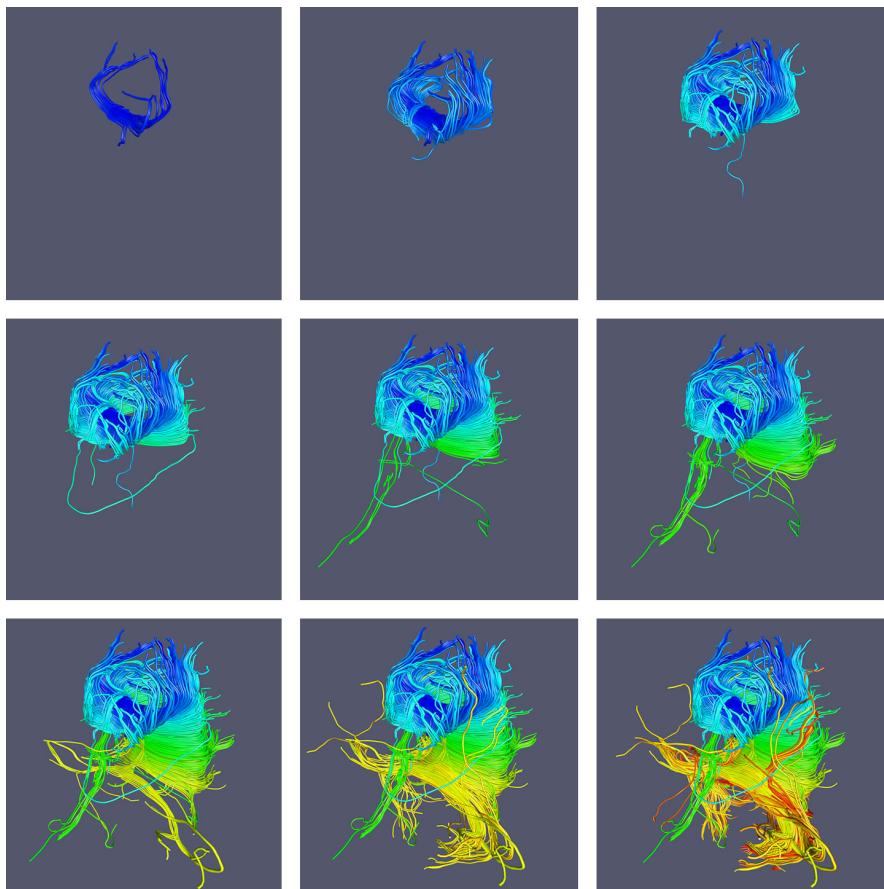


Figura 5.11: De izquierda a derecha, arriba a abajo: evolución en el tiempo de renderización de tubos tras cambiar la opacidad de 0 a 1.

5.4.10. Lados de tubo

Cuánto más lados tenga un tubo, mejor será su aspecto visual, aunque tendrá repercusión en el rendimiento al haber más vértices.

En la figura 5.12 se muestra la diferencia entre tubos de 5 lados y de 50. De cerca la diferencia es notable, pero de lejos es imperceptible.

En ambos casos se ha empleado el mismo nivel de detalle, 3, grosor de tubos, 0.003, y densidad, 0.001.

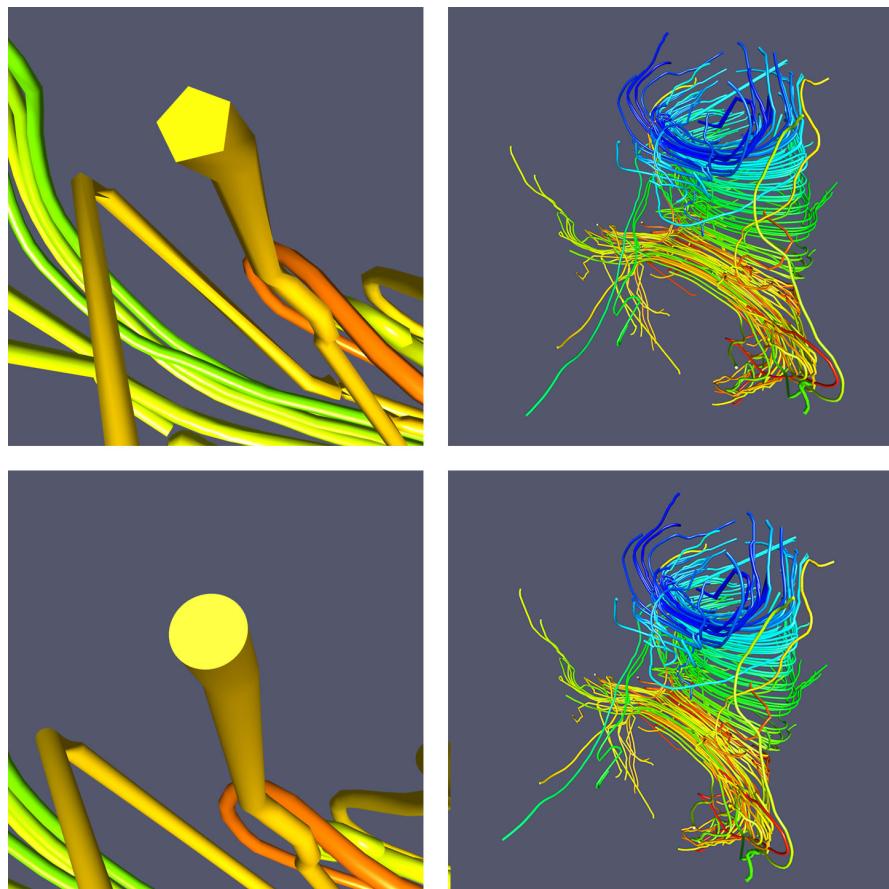


Figura 5.12: Arriba: número de lados igual a 5. Abajo: número de lados igual a 50.

5.4.11. Densidad de vértices

La combinación de polilíneas puede generar un exceso de vértices en algunas regiones; visualmente no es atractivo.

En la figura 5.13 se muestra la diferencia entre una densidad igual a 0 e igual a 0.001. Con una densidad superior a 0 se eliminan irregularidades a cambio de obtener un modelo poligonal más simple.

En ambos casos se ha empleado el mismo nivel de detalle, 3, grosor de tubos, 0.003 y número de lados, 5.

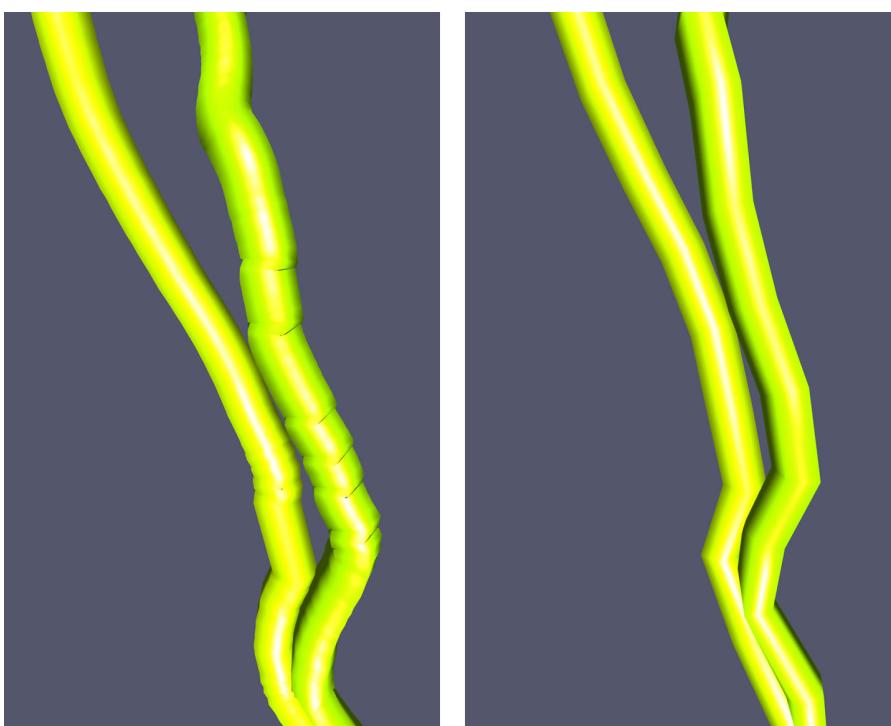


Figura 5.13: Izquierda: densidad igual a 0. Derecha: densidad igual a 0.001.

5.4.12. Color

La aplicación permite seleccionar diferentes colores de fondo.

En la figura 5.14 se muestran los disponibles: tema de VTK (#52576E), gris, negro y blanco.

En todos los casos se ha empleado el mismo nivel de detalle, 2, grosor de tubos, 0.003, número de lados, 5, y densidad, 0.001.

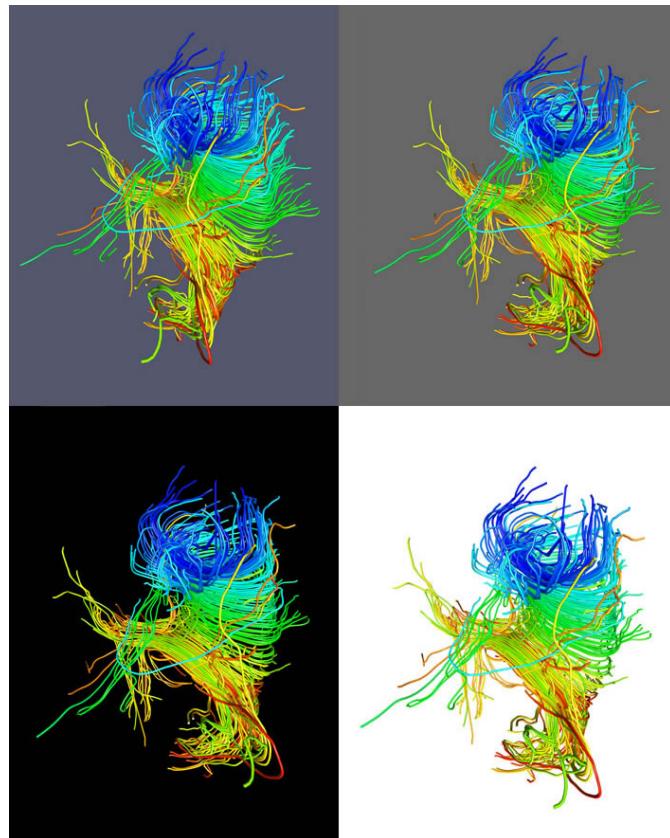


Figura 5.14: Gama de colores disponibles en la aplicación.

5.4.13. Showcase

Se muestra a continuación diferentes capturas de funcionamiento de la aplicación para el conjunto de datos *Fibres*.

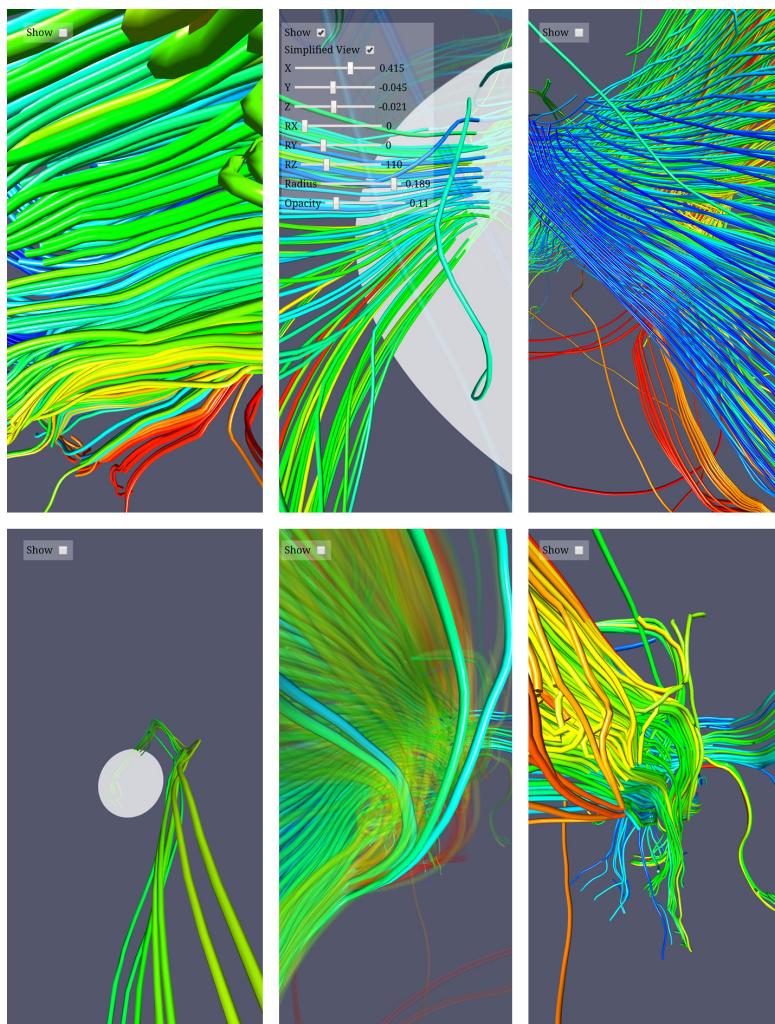


Figura 5.15: Funcionamiento de la aplicación para el conjunto de datos *Fibres*.

Capítulo 6

Conclusiones, problemas y trabajo futuro

En este capítulo se explicarán las conclusiones ha las que se ha llegado tras realizar el proyecto, los objetivos alcanzados, las dificultades para llegar hasta ellos, problemas que se han conseguido resolver e ideas para mejoras futuras.

6.1. Conclusiones

Con la visualización de datos interactiva en 3D realizada, se ha podido entender la facilidad que tienen este tipo de aplicaciones para comprender mejor los datos. Estos se han podido ver desde diferentes ángulos, niveles de detalle y con distintos colores para identificar fácilmente grupos y subconjuntos.

Al tratarse de una aplicación web, esta se ha podido probar en distintos dispositivos (móviles, portátiles y sobremesas) sin la necesidad de instalarla en cada uno de ellos, y al utilizar una URL con la dirección de la ubicación de los datos, se ha conseguido otro nivel de comodidad; no ha sido necesario la compartición de los ficheros entre los distintos dispositivos; en una aplicación clásica de escritorio, una modificación de un fichero implicaría la actualización en todos los dispositivos. Esto resulta de gran utilidad en el mundo de la investigación. Pensemos por ejemplo en un investigador que ha de viajar constantemente o que debe trabajar con otros investigadores en ubicaciones distantes.

Dada la variedad de dispositivos existentes en la actualidad, cada uno cuenta con unas características específicas que determinan su rendimiento: un PC último modelo destinado para videojuegos será más potente que un

móvil de gama media. Gracias a la selección de parámetros, el usuario puede adaptar la visualización según las capacidades de su dispositivo.

Han habido, sin embargo, experiencias negativas en el desarrollo del proyecto, todas ellas relacionadas con las gafas de realidad virtual *HTC Vive Pro*. No se han conseguido instalar, y a pesar de haber realizado un extenso *troubleshooting*, no se consiguió que funcionasen. Esto da a entender la necesidad de seguir desarrollando y mejorando estos sistemas VR, facilitando y simplificando su instalación y uso y aumentando su compatibilidad con más dispositivos.

6.2. Objetivos alcanzados

Se han conseguido casi todos los objetivos planteados en este proyecto: el diseño y creación de la aplicación en formato web, aplicar de técnicas de pre-procesamiento para facilitar y mejorar el rendimiento de la visualización de datos, conseguir interactuar con dichos datos (mediante el selector), elegir parámetros que afectan a la visualización, y ejecución en distintos dispositivos.

Cabe destacar que las sesiones de Skype realizadas con la doctora Anna Vilanova, profesora de *Delft University of Technology*, en la que comentó la aplicación de visualización de datos médicos vISTe, fueron de gran utilidad y ayudaron al desarrollo del proyecto. Se puede leer más sobre la aplicación en el siguiente enlace:

<https://sourceforge.net/projects/viste/>

El único objetivo no alcanzado ha sido utilizar las gafas de realidad virtual, aunque en su lugar se ha podido emular su uso mediante software.

6.3. Problemas

En esta sección se comentarán todas las dificultades y problemas, resueltos y no resueltos, que han ocurrido a lo largo del desarrollo del proyecto.

6.3.1. Resueltos

Número de actores Vs. Polydata

En VTK, cada actor tiene asociado un conjunto de datos geométricos, o *Polydata*. Por cada actor generado, el CPU envía una orden a la tarjeta gráfica, indicando que se ha de renderizar dicho objeto.

Cuanto mayor sea el número de actores, peor será el rendimiento (menor tasa de fotogramas), puesto que se producirá un cuello de botella.

Si se dispone de un conjunto de datos formado por X subconjuntos, se aprecia un mejor rendimiento si se utiliza un único actor en lugar de X actores cada uno con un subconjunto asociado.

Esta mejora de rendimiento supone un problema; no se facilita la interacción. En caso de desear modificar alguna parte del conjunto, como puede ser cambiar colores, transparencias o la topología, se ha de realizar una llamada a la tarjeta gráfica enviando todo el conjunto de datos modificado. Esto resulta en una enorme caída de la tasa de fotogramas durante cada interacción, lo que resulta en una peor experiencia e inviable para aplicaciones VR.

La solución ha sido asociar a cada actor un grupo de subconjunto de datos, consiguiendo un balance entre rendimiento e interacción.

Rotación de actores

Las matrices de transformación, rotación y escala de actores en VTK provienen de la superclase *vtkProp3D*. Esta clase no permite restablecer las distintas matrices de transformación a la identidad; únicamente permite aplicar transformaciones. Esto dificulta la labor de establecer una rotación determinada. Por ello se ha modificado el código fuente de *vtkProp3D* para añadir una nueva función, `setRotation(x, y, z)`, que permite especificar la rotación.

Hubo problemas también determinando el orden de aplicación de las matrices de rotación. La documentación de VTK indica el orden en el que se ejecutan las funciones de rotación para cada eje, pero resulta ser el contrario.

6.3.2. No resueltos

Instalación y uso HTC Vive Pro

DisplayPort

El primer problema encontrado fue el *DisplayPort*; *HTC Vive Pro* no utiliza HDMI, ya que con el *DisplayPort* se puede transmitir una mayor cantidad de datos geométricos, necesarios en las aplicaciones VR. Fue necesario por tanto adquirir una nueva tarjeta gráfica que contase con *DisplayPort*, en concreto una *Nvidia GTX 1070 Ti*.

Instalación

Previo a su uso, las gafas requieren de la instalación de software ofrecido por *HTC*, que incluye los drivers necesarios para comunicarse con el dispositivo y *Steam*, el famoso servicio de librería de videojuegos online. El instalador incluye además *SteamVR*, la librería para juegos de VR.

La instalación se realizó tanto en Windows 10 64-bit como Windows 7 64-bit. Durante el proceso de instalación se detectan las bases y los mandos, y funcionan los auriculares del *headset*, pero no se consigue instalar uno de los drivers, resultando en un “*Unknown Device*” con código de error 43.

Posteriormente se intentó la instalación en Linux, en concreto en Ubuntu 18.04, pero tampoco se ha conseguido hacerlas funcionar.

Parece ser que los ordenadores utilizados no disponían de una tarjeta PCI USB 3.1 xHCI de última generación, requeridas por el dispositivo.

No se consiguió ejecutar ninguna aplicación VR, ni de *Steam* ni propia; *Steam* informa que el dispositivo se encuentra desconectado, a pesar de estar encendido y correctamente conectado.

Espacio y cableado

El dispositivo utiliza una gran cantidad de cables, lo cual resulta incómodo en ciertos espacios, como puede ser un despacho. Un extenso cable conecta el *headset* a un controlador, del que salen otros tres cables, dos que se conectan al PC y otro a una fuente de alimentación.

Además, se deben instalar “bases”, encargadas de detectar los mandos. Estas se deben atornillar a paredes y conectar cada una a fuente de alimentación.

Finalmente, los mandos utilizan baterías, por lo cual requieren ser cargados previo a su uso, necesitando otras dos fuentes adicionales de alimentación.

Generación de tubos y normales

La generación de tubos realizada por la clase *vtkTubeFilter* requiere de normales de línea. En caso de que la polilínea no disponga de normales, estas son generadas automáticamente. Puede ocurrir que en algunas regiones del tubo, sobretodo en curvas muy afiladas, no se realice correctamente, como se ve en la figura 6.1.

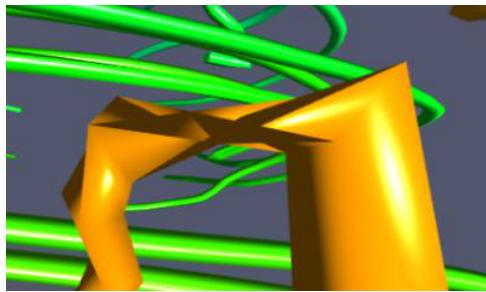


Figura 6.1: Error en la generación de un tubo.

Se intentó solucionar el problema utilizando normales propias, pasándolas a *vtkTubeFilter* junto con la polilínea. Sin embargo, la aplicación deja de funcionar. Según miembros de equipo de soporte de VTK, podría tratarse de un error en el código fuente de *vtkTubeFilter*. Cabe recordar, a fecha de publicación de este documento, VTK está siendo todavía convertido a su formato JavaScript.

Depth Peeling

Depth Peeling es una técnica de renderización de transparencias. Esta técnica, presente en VTK, no está presente todavía en su versión JavaScript. *Depth Peeling* renderiza la escena varias veces por capas y luego las combina. En VTK.js se utiliza una técnica de transparencia más simple: se acumulan transparencias de triángulos, lo cual genera efectos no deseados; objetos cercanos a la cámara pueden parecer estar más distantes. Es de vital importancia corregir estos errores, ya que la visualización es la práctica de obtener significado de los datos [18]. Si estos no se muestran correctamente, obtendremos un significado equivocado.

En la figura 6.2 se compara una renderización correcta y una incorrecta.



Figura 6.2: Izquierda: renderización sin transparencias. Centro: técnica simple de transparencia; el dragón azul parece estar detrás de los demás. Derecha: *depth peeling*; los dragones están renderizados en el orden correcto.

En la figura 6.3 se muestran errores de transparencias existentes en la aplicación.

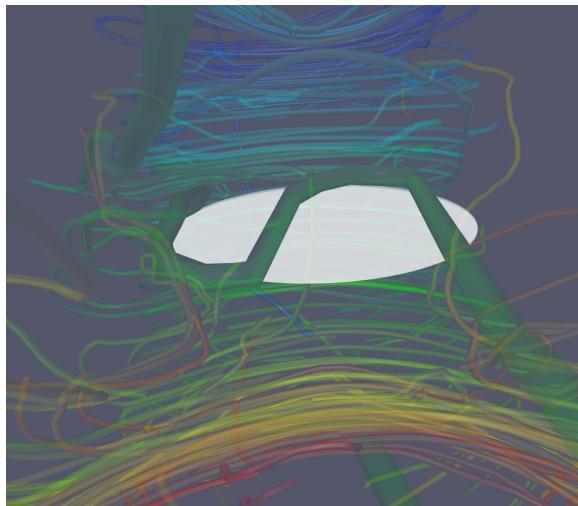


Figura 6.3: El tubo que se encuentra entre la cámara y el selector, a pesar de ser transparente, oculta parte del selector.

Paralelización

Se intentó paralelizar el procesamiento, permitiendo modificar la geometría de los tubos en tiempo real. Para ello se utilizaron *WebWorkers*, hebras que ejecutan su propio código JavaScript desde un fichero aparte. Sin embargo, cada modificación de geometría requería ser notificada al GPU,

producíendose cuellos de botella y no consiguiendo mejorar el rendimiento.

VTK, y por extensión VTK.js, está diseñado para utilizar una única hebra principal.

Documentación VTK.js

VTK dispone de una amplia y detallada documentación. No es el caso con VTK.js; en la página de la API algunas clases no disponen de explicaciones, únicamente código fuente. Se ha de estudiar el código fuente para poder determinar qué funciones están implementadas y cómo funcionan.

Algunas clases nuevas no presentes en VTK también carecen de documentación.

6.4. Trabajo futuro

Aparte de las ideas comentadas en la sección “*Requisitos Futuros*”, todos los problemas sin resolver comentados anteriormente se podrían solucionar:

- **Instalación y uso HTC Vive Pro:** se podría adquirir un sistema diferente, como puede ser *Occulus Rift*, y sustituir la emulación por el sistema real.
- **Generación de tubos y normales:** mejorar el algoritmo de combinación de polilíneas para evitar curvas afiladas o esperar al equipo de VTK.js a corregir errores en la clase *vtkTubeFilter*.
- **Depth Peeling:** esperar al equipo de VTK.js a implementar esta técnica o utilizar shaders propios.

Bibliografía

- [1] Tuukka M. Takala. Ruis – a toolkit for developing virtual reality applications with spatial interaction. *Hybrid Interaction Spaces*, page 94, October 2014.
- [2] Micha Pfeiffer, Hannes Kenngott, Anas Preukschas, Matthias Huber, Lisa Bettscheider, Beat Müller-Stich, and Stefanie Speidel. Imhotep: virtual reality framework for surgical applications. page 1, March 2018.
- [3] Hardvard Business Review. Research: How virtual reality can help train surgeons. <https://hbr.org/2019/10/research-how-virtual-reality-can-help-train-surgeons>.
- [4] FORBES. Nvidia says vr is too demanding for most pcs. <https://www.forbes.com/sites/antonyleather/2016/01/04/nvidia-says-vr-is-too-demanding-for-most-pcs-bad-news-for-consoles/>.
- [5] Zachary A. King, Andreas Dräger, Ali Ebrahim, Nikolaus Sonnenschein, Nathan E. Lewis, and Bernhard O. Palsson. Escher: A web application for building, sharing, and embedding data-rich visualizations of biological pathways. August 2015.
- [6] Eurographics. <https://www.eurographics2018.nl/>.
- [7] Geetha Soujanya Chilla, Cher Heng Tan, Chenjie Xu, and Chueh Loo Poh. Diffusion weighted magnetic resonance imaging and its recent trend—a survey. pages 407, 411, January 2015.
- [8] D.M. Gazzola and R.I. Kuzniecky. Encyclopedia of basic epilepsy research. pages 1580–1585, 2009.
- [9] Encyclopedia of basic epilepsy research, 2009.
- [10] Susumu Mori. Introduction to diffusion tensor imaging. *Elsevier*, pages 93–123, 2007.

- [11] Raymond K. W. Wong, Thomas C. M. Lee, Debashis Paul, Jie Peng, and Alzheimer's Disease Neuroimaging Initiative. Fiber direction estimation, smoothing and tracking in diffusion mri. page 9, September 2015.
- [12] Pietro Gori, Olivier Colliot, Linda Marrakchi-Kacem, Yulia Worbe, Fabrizio De Vico Fallani, Mario Chavez, Cyril Poupon, Andreas Hartmann, Nicholas Ayache, , and Stanley Durrleman. Parsimonious approximation of streamline trajectories in white matter fiber bundles. December 2016.
- [13] Eleftherios Garyfallidis, Matthew Brett, Marta Morgado Correia, Guy B. Williams, and Ian Nimmo-Smith. Quickbundles, a method for tractography simplification. December 2012.
- [14] Vid Petrovic, James Fallon, and Falko Kuester. Visualizing whole-brain dti tractography with gpu-based tuboids and lod management. December 2007.
- [15] C. Mercier, P. Gori, D. Rohmer, M-P. Cani, J-M. Thiery T. Boubekeur and, and I. Bloch. Progressive and efficient multi-resolution representations for brain tractograms. 2018.
- [16] Maarten H. Everts, Eric Begue, Henk Bekker, Jos B. T. M. Roerdink, and Tobias Isenberg. Exploration of the brain's white matter structure through visual abstraction and multi-scale local fiber tract contraction. 2015.
- [17] Francois Rheault, Jean-Christophe Houde, and Maxime Descoteaux. Visualization, interaction and tractometry: Dealing with millions of streamlines from diffusion mri tractography. 2017.
- [18] Vtk technical highlight: Dual depth peeling. <https://blog.kitware.com/vtk-technical-highlight-dual-depth-peeling/>.

Apéndice A

Instalación

En este apéndice se explicarán los pasos necesarios para la instalación de la aplicación.

1. Crear una carpeta en la ubicación y con el nombre deseados. Para la explicación de la instalación, llamaremos a la carpeta *app*.
2. Abrir un terminal y ubicarse en la carpeta *app* con el comando `cd`.
3. Instalar *Node.js*. Para ello se debe descargar el instalador para el sistema operativo utilizado desde la siguiente URL y seguir las instrucciones: <https://nodejs.org/en/>
4. Instalar *VTK.js* ejecutando en el terminal el comando `npm install vtk.js --save`.
5. Descomprimir el fichero *proyecto.zip* en una ubicación distinta a la de la carpeta *app* y copiar todos sus archivos en la carpeta *app*. Se avisará de la existencia de dos ficheros, ambos llamados *index.js*; reemplazar los originales.
6. Todavía ubicados en el terminal en la carpeta *app*, ejecutar el comando `npm run build`. Deberá aparecer en la carpeta *dist* los ficheros compilados.
7. Abrir el fichero *distmain.html* con un editor de texto y en la línea 132 modificar el campo `action` con la URL de la ubicación del fichero *distindex.html*:

```
<form name="myform" action=".../index.html">
```

Por defecto el campo `action` contiene la URL del fichero *index.html* del repositorio utilizado en el desarrollo del proyecto.

Para ejecutar la aplicación se ha de abrir el fichero *distmain.html* en un navegador web.

NOTA

El termino “compilar” está erróneamente usado, puesto que técnicamente se “trans-compila”: se convierte código JavaScript de alto nivel a código JavaScript de más bajo nivel, nunca a código máquina.

