Visualización Ilustrativa de Fibras Cerebrales

Pedro Hermosilla Casajús

2 de septiembre de 2011

Agradecimientos

Me gustaría agradecer al grupo de investigación *Moving* por su apoyo y por la formación obtenida durante el tiempo que he estado trabajando con ellos. En especial, me gustaría agradecer a Pere-Pau Vázquez y a Isabel Navazo su constante motivación.

También me gustaría agradecer al departamento de Ingeniería Biomédica de la Universidad de Tecnología de Eindhoven, en especial a Anna Vilanova y Ralph Brecheisen, por su colaboración en este proyecto y por facilitarnos tanto los modelos, como la aplicación sobre la que desarrollar nuestra técnica de pintado.

Índice general

Capítulo 1

Introducción

Diffusion Tensor Imaging (DTI) es una técnica de imagen basada en la resonancia magnética (MR), que ofrece una visión única de la organización estructural de la sustancia blanca del cerebro. Esto se consigue mediante la medición de la difusión de moléculas de agua en el tejido. En el agua, la difusión es libre y tiene la misma magnitud en todas las direcciones. En este escenario obtenemos un perfil de difusión isotrópico. En los tejidos fibrosos sin embargo, como en la materia blanca del cerebro, la difusión se limitará en sentido perpendicular a las fibras. Esto provoca que, en este escenario, obtengamos un perfil de difusión anisotrópico.

En las imágenes DTI, el perfil de difusión se modela como una distribución de probabilidad de Gauss y por lo tanto puede ser descrito por un tensor de segundo orden [BPD94]. En este modelo, el principal *eigenvector* del tensor corresponde a la dirección de mayor difusión, que es la misma que sigue la estructura de fibras (Figura 1.1). Gracias a estos tensores podemos realizar un recorrido sobre ellos e ir reconstruyendo un modelo 3D de estas fibras. Esta reconstrucción recibe el nombre de *fiber tracking* [VAD05] (Figura 1.2).

Pese a su potencial, la salida de la reconstrucción, *fiber tracking*, contiene una cantidad considerable de incertidumbre. Este error o incertidumbre se acumula a lo largo del proceso de obtención del modelo. En la fase de obtención de los datos puede introducir errores por ruido en las imágenes, distorsión de la imagen, parámetros del escáner, etc. En la fase de reconstrucción se introducen errores de aproximación dependiendo del modelo de difusión utilizado.

En la mayor parte de los algoritmos, este error no se muestra, lo cual da una sensación de certidumbre en los datos que no se corresponde con la realidad. Una aplicación para neurocirugía, no puede pasar por alto estos errores, ya que la aplicación se usará para tomar decisiones y evaluar el riesgo quirúrgico. Si no mostramos esta incertidumbre a la hora de estimar

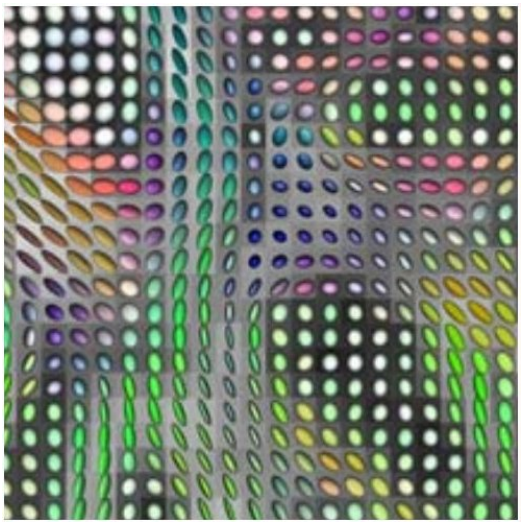


Figura 1.1: Modelo de difusión.

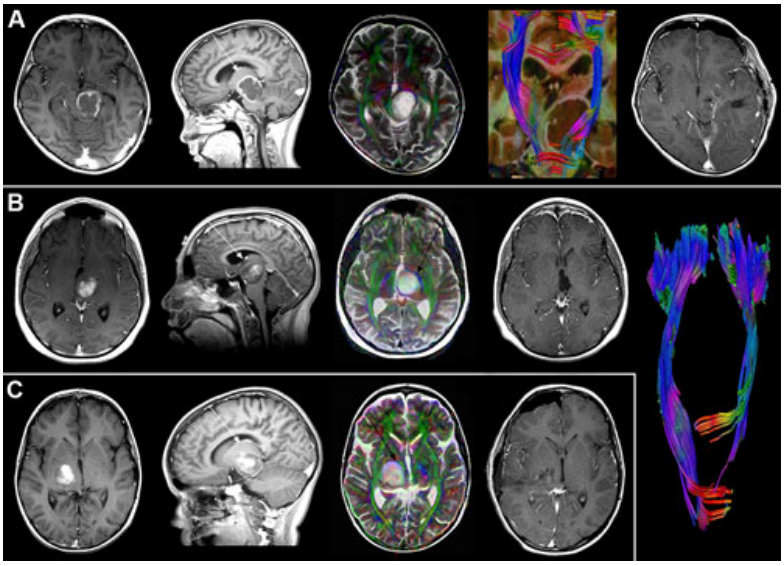


Figura 1.2: Reconstrucción del modelo de fibras.

la longitud de las fibras cerebrales podemos provocar daño en tejido cerebral sano.

Nuestro objetivo en esta tesis es realizar un visualizado de este modelo de fibras mostrando el nivel de incertidumbre que tiene cada fibra o grupo de fibras. El departamento de Ingeniería Biomédica de la Universidad de Tecnología de Eindhoven tiene desarrollada una aplicación para la visualización de fibras cerebrales (http://bmia.bmt.tue.nl/software/dtitool/). Pere-Pau Vázquez tiene contacto con este departamento y están desarrollando métodos de visualización de la incertidumbre de los datos. Este trabajo se hace en colaboración con ellos, en especial con Anna Vilanova y Ralph Brecheisen. Por eso, hemos implementado nuestra técnica de visualización como un plugin para esta aplicación. De esta manera, nos proporcionan muchas facilidades, como el cargado de modelos de fibras o parte del proceso de visualización implementado.

Capítulo 2

Introducción a los Gráficos por Computador

2.1. Proceso de visualización

El proceso de visualización se encarga de transformar información geométrica (vértices, triángulos, etc.) e información sobre la iluminación de la escena (posición, color, intensidad, etc.), en una imagen 2D. Este proceso consta de varios pasos y cada uno de ellos requiere un coste computacional muy alto, por lo que estos pasos deben estar optimizados al máximo. Un libro donde se da una buena introducción al proceso de visualización es el de Allan Watt [Wat00].

Depende del tipo de iluminación que queremos simular en nuestra escena, el algoritmo utilizado será diferente. Los tipos de iluminación que se simulan son los siguientes:

- **Modelo de iluminación global:** En este modelo de iluminación se intenta simular la interacción completa de la luz con la escena. Se simulan los diferentes rebotes de los rayos de luz con las superficies de la escena, por lo tanto la contribución de iluminación sobre un punto de una superficie viene dada por la luz que incide directamente más la iluminación indirecta que viene rebotada desde diferentes superficies vecinas. Estos algoritmos tienen un coste computacional muy alto por lo que es impracticable en tiempo real, aunque con los avances en hardware se han realizado algunas implementaciones en tiempo real (Optix - http://www.nvidia.com/object/optix.html). Existen numerosos algoritmos que simulan este tipo de interacción. Entre ellos podemos destacar:

- **Ray-Tracer:** Este algoritmo intenta simular la *reflexión especular*[1] lanzando rayos desde cada uno de los pixels de pantalla y calculando los diferentes rebotes con la escena (Imagen 2.1 (a)). En el libro [PDB06] se explica este algoritmo con detalle.
- **Radiosity:** Este algoritmo intenta simular la *reflexión difusa*[2] lanzando rayos desde los diferentes puntos de luz de la escena y calculando los diferentes rebotes en la escena (Imagen 2.1 (b)). Este algoritmo se describe en detalle en el libro [PDB06].

(a) RayTracer (b) Radiosity

Figura 2.1: Diferentes imágenes obtenidas con algoritmos de iluminación global.

- **Modelo de iluminación local:** En este modelo de iluminación solo se calcula la primera incidencia de la luz con la superficie. La imagen obtenida es menos realista pero, gracias a su bajo coste computacional, se puede realizar el cálculo en tiempo real. El algoritmo utilizado para implementar este modelo, en lugar de utilizar rayos que intersecan con la escena, se proyecta la geometría en pantalla y para cada pixel sobre el que se proyecta, se calcula la iluminación (Imagen 2.2). Este es el algoritmo que se ha implementado en hardware, ya que es el utilizado en aplicaciones en tiempo real.

[1]Reflexión especular: *Reflexión de la luz en una superficie pulida en la que a cada rayo incidente le corresponde solo uno reflejado que la abandona según el mismo ángulo.*

[2]Reflexión difusa: *Reflexión de la luz en una superficie rugosa en la que a cada rayo incidente le corresponden varios reflejados, que la abandonan en un rango de ángulos.*

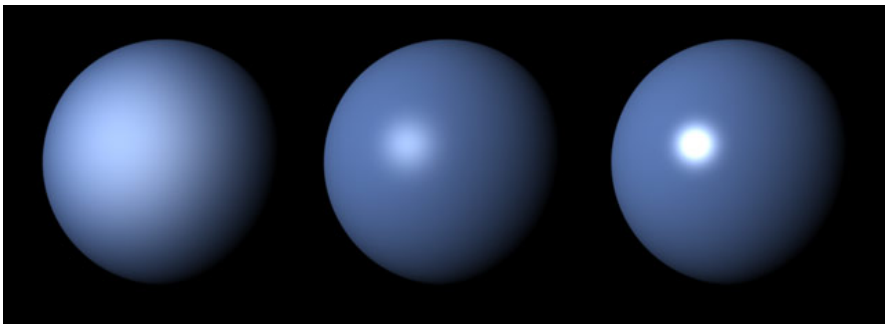


Figura 2.2: Modelo de iluminación local.

2.2. Algoritmo de visualización (Pipeline)

El algoritmo para implementar el modelo de iluminación local consta de varias etapas:

- **Transformación al sistema de coordenadas global:** La geometría de los objetos de la escena está definida en un sistema de coordenadas local, por eso, en un primer paso, hemos de transformar esta geometría al sistema de coordenadas de la escena. Esta transformación se aplica multiplicando los vértices de la geometría por una matriz de transformación. A la salida de esta etapa tenemos toda la geometría definida en un sistema de coordenadas común.

- **Transformación al sistema de coordenadas de cámara:** En esta etapa del algoritmo, transformamos la geometría del sistema de coordenadas global o de mundo, a un sistema de coordenadas definido por la posición y dirección de la cámara. En este sistema de coordenadas la posición de la cámara está en el (0,0,0) y el eje z coincide con la dirección de visión de la cámara.

- **Transformación al sistema de coordenadas de pantalla:** Esta es la última etapa en la que se realiza una transformación de la geometría. En esta etapa se proyecta toda la geometría a un plano definido en la dirección de visión de la cámara. Las dos proyecciones que se suelen aplican en este paso son la *proyección perspectiva* y la *proyección ortogonal*. Al final de esta etapa tenemos la geometría definida en un espacio de rango [-1,1].

- **Eliminación de la geometría no visible:** Una vez tenemos la geometría en espacio de pantalla, eliminamos la lista de primitivas que no son visibles desde ese punto de vista. Las primitivas no visibles son: las

que están orientadas en sentido contrario a la cámara y las primitivas que están fuera o parcialmente fuera del volumen de visión.

- **Rasterizado:** En un último paso se calcula la lista de pixels sobre los que se proyecta cada primitiva y se calcula el color final de cada uno, utilizando un modelo de iluminación local.

Los tres primeros pasos se suelen agrupar en uno solo, concatenando las matrices de transformación. De esta manera solo es necesario realizar una multiplicación por cada uno de los vértices.

Estos pasos del proceso suelen ejecutarse en hardware específico (Tarjetas gráficas), pero, antes de ejecutar todo este proceso, se suele realizar una selección de modelos que son potencialmente visibles para no enviar a procesar geometría que estamos seguros que no serán visibles. Para facilitar el recorrido de la lista de modelos, se suelen almacenar en una estructura de datos que ayude a descartar un buen grupo de ellos con pequeñas comprobaciones (subdivisión del espacio en forma de árbol, subdivisión del espacio en una rejilla, etc.).

En el libro *Real-Time Rendering* [TAMH08] se da una buena descripción de este algoritmo de visualización, junto con un gran número de técnicas de la visualización en tiempo real.

2.3. Hardware Gráfico

El hardware gráfico es el encargado de implementar la mayoría de pasos del algoritmo de visualización. Este hardware tuvo su primera aparición en los años 60 junto con la aparición de los primeros monitores. Estas primeras tarjetas solo eran capaces de visualizar texto a 40x25 u 80x25, pero con la aparición de los primeros chips gráficos, empezaron a dotar a los equipos de capacidades gráficas.

En la década de los 80 aparecieron los primeros chips gráficos para PC personales. Estos chips tenían funcionalidades para pintar líneas, áreas y texturas 2D.

A principio de la década de los 90, aparecieron las primeras tarjetas gráficas aceleradoras 3D, que se basaban en las tarjetas aceleradoras 2D que había hasta el momento. Estas tarjetas no seguían ningún estándar, por lo que se tenía que programar especialmente para cada tipo de tarjeta.

A mediados de los 90, los juegos y aplicaciones 3D se volvieron muy comunes en PC personales y consolas. Por este motivo aparecieron las primeras tarjetas gráficas que seguían un estándar. Las librerías gráficas que implementaban este estándar eran OpenGL y DirectX.

A principios de la década de los 2000 aparecieron las primeras tarjetas gráficas programables. Hasta entonces, las tarjetas gráficas tenían implementadas un algoritmo fijo, pero, con la aparición de estas tarjetas, el programador era capaz de modificar los cálculos que se realizaban en el proceso de visualización. Estos programas que ejecutaba la tarjeta gráfica recibían el nombre de *shaders*, y se programaban en lenguaje ensamblador.

Más adelante aparecieron lenguajes de más alto nivel para hacer más fácil la tarea del programador. DirectX tenía su lenguaje HLSL, OpenGL tenía GLSL y nVidia creó un lenguaje intermedio, Cg, que puede ser compilado para ejecutarse tanto en DirectX como en OpenGL.

Poco a poco, los avances en los procesadores gráficos permitieron incorporar saltos dinámicos en el código, y con ello aparecieron los bucles y condicionales (hasta entonces solo se simulaban). Estos avances hicieron que se empezaran a utilizar los procesadores gráficos para realizar cálculos en paralelo que no tenían nada que ver con los gráficos.

Con la aparición de DirectX 10 y OpenGL 3.0, se añadió una nueva funcionalidad que procesaba datos a nivel de primitiva (antes sólo se podía realizar cálculos a nivel de vértice y de pixel). Esta nueva etapa, hizo que muchos algoritmos se pudieran implementar en hardware.

En la actualidad, la última versión de DirectX es la 11 y de OpenGL es la 4.0. Estas versiones, entre otras cosas, han añadido nuevas etapas para generar la geometría de forma dinámica a partir de superficies paramétricas. En la imagen 2.3 podemos ver los diferentes pasos de esta nueva pipeline.

A continuación se muestra un ejemplo de shaders en GLSL:

- **Vertex Shader:** Este programa se encarga de asignar el color del vértice (*gl_Color*) a la siguiente fase del proceso de visualización (*gl_FrontColor*) y realizar la transformación de espacio de objeto a espacio de pantalla (*ftransform()*).

```
1  void main(void)
2  {
3     gl_FrontColor = gl_Color;
4     gl_Position = ftransform();
5  }
```

- **Pixel Shader:** Este programa se encarga de asignar el color obtenido de las fases anteriores (*gl_Color*) al color final del pixel (*gl_FragColor*).

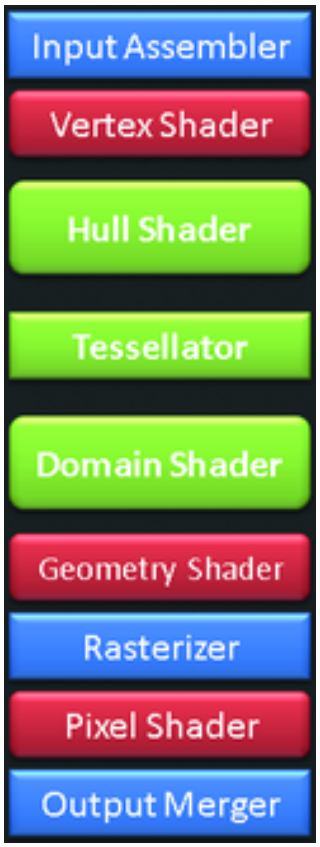


Figura 2.3: Pipeline de DirectX 11.

```
1  void main(void)
2  {
3     gl_FragColor = gl_Color;
4  }
```

Capítulo 3

Visualización Ilustrativa

La visualización ilustrativa, o renderizado no fotorrealista (NPR), se centra, al contrario que el renderizado tradicional, en simular estilos expresivos o de arte digital, como el dibujo, pintura, ilustración técnica, etc. La diferencia entre estas técnicas de pintado y la visualización tradicional, que intenta simular la realidad, es que se centran en la comunicación. Lo importante es comunicar al usuario las características principales del modelo descartando las que carecen de información. Otra característica importante de estas técnicas es que intentan tener un acabado que sea visualmente atractivo. Podemos ver un ejemplo de aplicar alguna de estas técnicas sobre un modelo 3D en la imagen 3.1.

En su libro, Bruce Gooch y Amy Gooch [GG01], clasifican estas técnicas en tres grupos: Simulación de medios artísticos, métodos de asistencia al usuario en el proceso artístico y sistemas automáticos. A continuación daremos algunos ejemplos de técnicas de visualización ilustrativa separadas por grupos. Para ver una lista más amplia de estas técnicas podemos consultar el libro de Bruce y Amy Gooch [GG01].

3.1. Simulación de medios artísticos

Este grupo de técnicas intentan simular diferentes medios artísticos como los pinceles, lápices, tinta o el papel. A continuación daremos algunos ejemplos de algoritmos que están dentro de este grupo.

Un algoritmo para simular el pintado a lápiz es el de Mario Costa Sousa y John Buchanan [SB99]. Ellos realizan un renderizado automático de modelos 3D para obtener una imagen con apariencia de haber sido realizado a mano (Imagen 3.2). En su artículo, descomponen el problema en cuatro partes:

- Simular los medios de pintado (lápiz y papel).

Figura 3.1: Ejemplo de renderizado no fotorrealista.

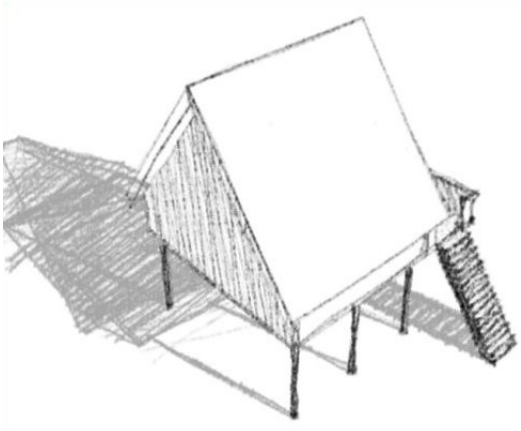
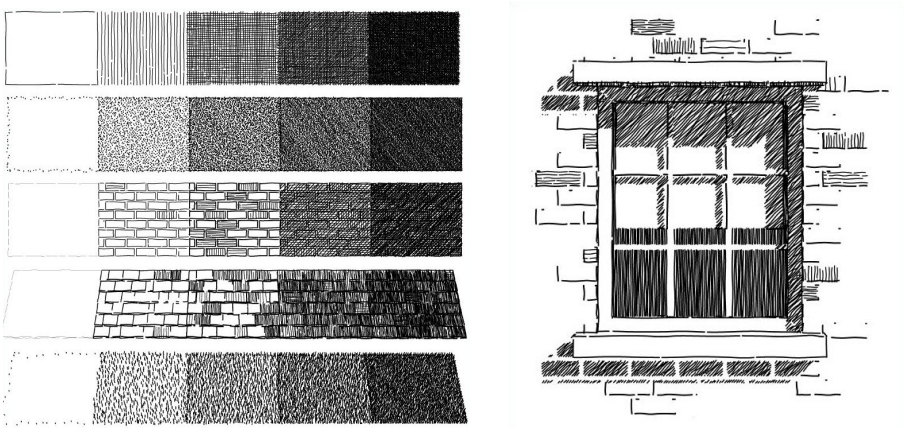


Figura 3.2: *Computer-generated graphite pencil rendering of 3d polygonal models* [SB99].



(a) Texturas (b) Resultado

Figura 3.3: *Computer-generated pen-and-ink illustration* [WS94].

- Modelar las primitivas de pintado. Crear trazos individuales de lápiz y a partir de ellos construir las texturas.
- Simular las técnicas básicas de pintado usadas por los artistas familiarizados con el pintado a lápiz.
- Modelar la composición de pintado final.

Dentro de este grupo también hay que destacar el artículo presentado por Georges Winkenbach y David H. Salesin [WS94]. En este artículo se presenta una técnica de pintado para simular el pintado con tinta china. Ellos utilizan una serie de texturas con diferentes densidades de trazos para simular la iluminación del modelo. En la figura 3.3 podemos ver tanto la lista de texturas como el resultado final de aplicar esta textura.

3.2. Métodos de asistencia

En esta clase se incluyen los algoritmos que dan asistencia al usuario en el proceso artístico. A continuación se citan algunos sistemas que están dentro de este grupo.

Paul Haeberli [Hae90] presento en 1990 un sistema interactivo que permitía al usuario crear imágenes a partir de fotografías. Haeberli también mostró como este sistema se podía expandir para usar escenas generadas por ordenador en lugar de fotografías. Podemos ver los resultados obtenidos en la figura 3.4.

Figura 3.4: *Paint by Numbers: Abstract Image Representation* [Hae90].

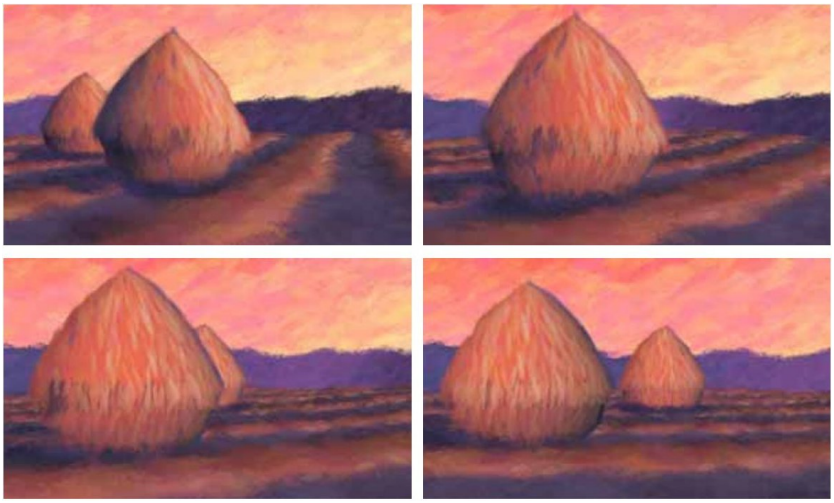


Figura 3.5: Painterly rendering for animation [Mei96].

Otra técnica fue la presentada por Barbara Meier [Mei96], que se dedicaba a renderizar en un estilo pictórico. Este método proporciona una coherencia frame a frame modelando las partículas en espacio 3d y utilizándolas para posicionar los pinceles en espacio de pantalla. Podemos ver el resultado de aplicar esta técnica en la imagen 3.5.

3.3. Sistemas automáticos

Dentro de este grupo de algoritmos están incluidos todos aquellos algoritmos que aplican alguna técnica de visualización ilustrativa de forma automática sin interacción del usuario. Algunos algoritmos dentro de este grupo son los siguientes:

Dentro de este grupo está la detección de siluetas, que es una técnica

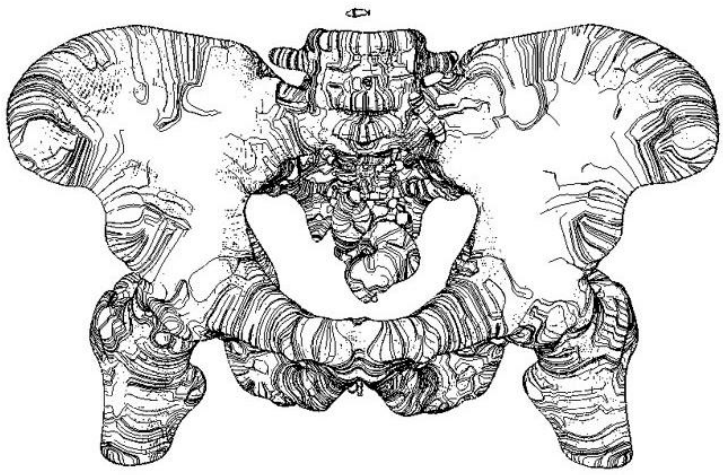


Figura 3.6: *Line Direction Matters: An Argument For The Use Of Principal Directions In 3D Line Drawings* [AGL00].

muy estudiada dentro del campo de la visualización ilustrativa. Esta técnica la hemos utilizado en nuestra aplicación, por lo que ya detallaremos diferentes técnicas para detectar la silueta en un capítulo posterior.

Otra técnica dentro de este grupo es la presentada por Ahna Girshick et al. [AGL00], que utilizaban las principales curvaturas del modelo para dar más información al usuario cuando la silueta no aporta la suficiente. Podemos ver el resultado obtenido de aplicar esta técnica en la imagen 3.6.

Capítulo 4

Visualización Médica

4.1. Introducción

La visualización médica es un campo que estudia la visualización de datos orgánicos. Normalmente, estos datos están representados como un conjunto de imágenes 2D que representan un modelo 3D, obtenidas a partir de escáners CT o escáners de resonancia magnética (MRI). Podemos ver la reconstrucción de un cerebro a partir de este conjunto de imágenes 2D en la figura 4.1.

A partir de este conjunto de imágenes se construye un modelo de volumen para poder visualizarlo. Este modelo de volumen es una matriz tridimensional donde cada posición tiene un valor de densidad.

Existen diferentes opciones para visualizar estos modelos:

- **Renderizado directo del volumen:** Esta opción requiere que se asigne a cada voxel un valor de opacidad y color. Esto se consigue con lo que se llama *función de transferencia*. Esta función mapea un color y una opacidad a cada valor de densidad. Existen varias técnicas para realizar la visualización de esta manera. Una de ellas es la que se conoce como *ray casting*. Esta técnica lanza un rayo que se inicia en cada pixel. Este rayo atraviesa el volumen y para cada muestra calcula su color y opacidad. Una vez el rayo sale del volumen se compone el color final utilizando los colores y opacidades obtenidas de estas muestras. Podemos ver el resultado de aplicar esta técnica sobre un modelo de volumen en la imagen 4.2.

- ***Maximum intensity projection*:** Esta técnica es parecida al renderizado directo de volumen, pero solo se proyectan los voxels que contienen un valor de densidad definido dentro de un rango. El problema de esta técnica es que no proporciona una sensación de profundidad

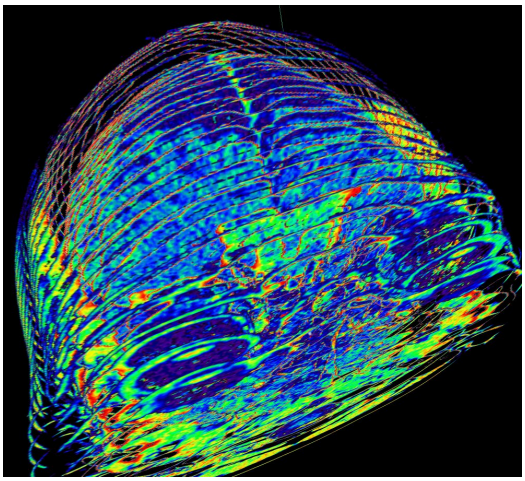


Figura 4.1: Reconstrucción de un cerebro a partir de imágenes 2D.

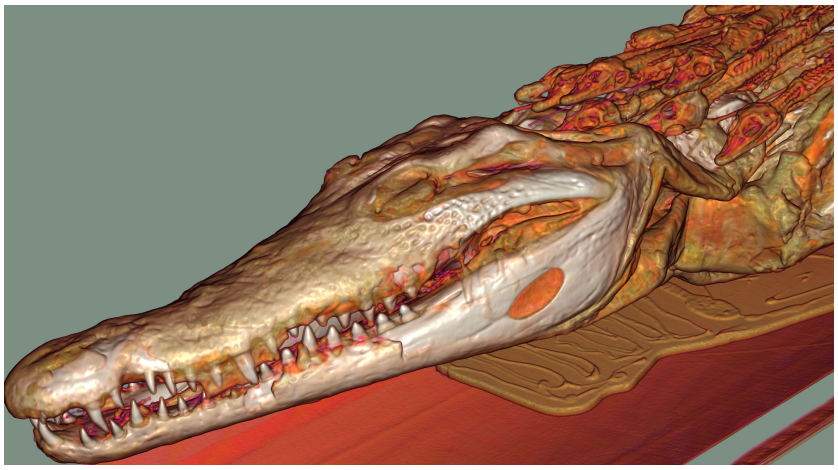


Figura 4.2: *Volume ray casting.*

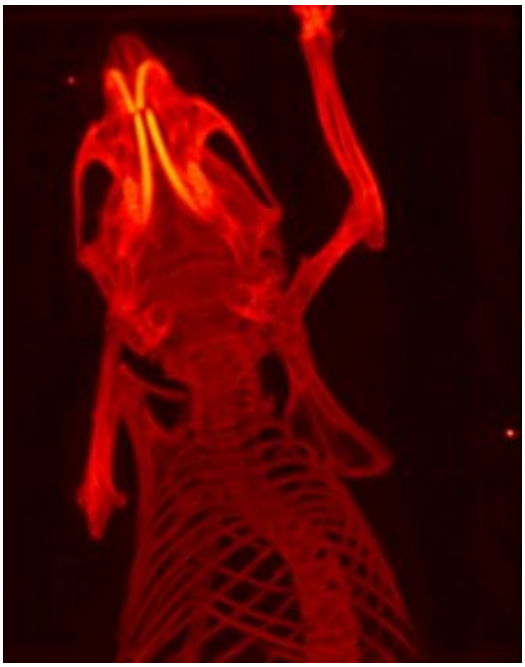


Figura 4.3: *Maximum intensity projection.*

correcta. Podemos ver una imagen obtenida a través de esta técnica en la figura 4.3.

- **Técnicas optimización:** El renderizado de volumen es muy costoso, ya que requiere muchos accesos a textura por cada uno de los rayos que se generan (uno por pixel). Por eso existen numerosas técnicas que intentan optimizar este proceso. A continuación describiremos algunas de ellas:
  - **Saltarse espacios vacíos:** Existen sistemas para saltarse espacios vacíos.
  - **Finalización del recorrido:** Otra opción es finalizar el recorrido del rayo una vez encontramos una muestra con un valor de densidad suficientemente alto.
  - **Estructuras jerárquicas:** Esta optimización utiliza una subdivisión del espacio (Octree, BSP, etc) para acelerar el recorrido del volumen.
  - **Construcción de un modelo 3D:** Otra opción para visualizar un modelo de volumen es generar una malla 3D a partir de una función de transferencia. De esta manera la visualización se acelera considerablemente, ya que solo hemos de enviar a pintar los

Figura 4.4: *Illustrative Rendering of Dense Line Data* [MHEI09].

triángulos de esta malla. El problema que tiene esta opción es que no podemos modificar en tiempo real la función de transferencia, sino que se tendría que generar un nuevo modelo cada vez que se modifique la función de transferencia.

4.2. Visualización de Fibras

Existen numerosas especialidades dentro de la visualización médica. La visualización de fibras cerebrales es la visualización de conexiones entre zonas del cerebro. Como he explicado en la introducción de esta tesis, este modelo de fibras se obtiene realizando una reconstrucción a partir de un modelo de difusión de moléculas de agua, obtenido a partir de la técnica de imagen *Diffusion Tensor Imaging* (DTI).

4.2.1. Visualización simple

Existen varios métodos publicados para visualizar este grupo de fibras. En el artículo de Everts et al. [MHEI09] se describe un algoritmo para la visualización de fibras cerebrales utilizando técnicas de visualización ilustrativa. Esta técnica crea una serie de rectángulos para representar las fibras y añade un halo alrededor de ellas en función de la profundidad de cada una. Podemos ver una imagen de esta técnica en la figura 4.4.

Otra técnica para visualizar fibras es la presentada por Ron Otten et al. [ROvdW10]. Ellos proponen un algoritmo que utiliza también técnicas de visualización ilustrativa para resaltar diferentes *clusters* de fibras. En el artículo utilizan esta técnica para visualizar las fibras sobre las que no estamos centrando la atención y proporcionar una visión general de todo el grupo de fibras. En la figura 4.5 se muestra el resultado de aplicar esta técnica.

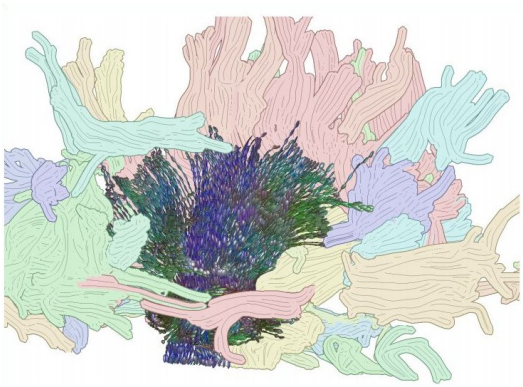


Figura 4.5: *Illustrative White Matter Fiber Bundles* [ROvdW10].

4.2.2. Modelado de la incertidumbre

Estas dos técnicas se centran en visualizar las fibras sin más, pero, como he explicado en la introducción de esta tesis, estas fibras tienen un cierto error o incertidumbre, que viene provocado tanto por el método de obtención de datos como por el algoritmo para reconstruir este modelo. Los métodos de reconstrucción se pueden dividir en dos grupos:

- **Algoritmos deterministas:** Para una cierta entrada siempre devuelven el mismo resultado.
- **Algoritmos probabilísticos:** El resultado de aplicar el algoritmo no solo depende de los datos de entrada sino también de un proceso pseudo-aleatorio.

Existen numerosas opciones para modelar esta incertidumbre y todas ellas pertenecen a uno de estos dos grupos: *Métodos empíricos* o *Métodos de modelado matemático.*

- **Métodos empíricos:** Una manera de modelar la incertidumbre de un modelo es realizar un gran número de *diffusion-weighted imaging* (DWI)[1]. De esta manera el ruido generado por la obtención de datos será diferente, y podremos determinar que fibras tienen más incertidumbre que otras. El problema de este proceso es que requiere mucho tiempo para obtener las diferentes DWI. Una variación de este método es el conocido como *bootstrap*, que modela la incertidumbre a partir

[1]Diffusion-weighted imaging: *Imagen obtenida a través de resonancia magnética donde cada voxel tiene una intensidad que refleja la cantidad de difusión de moléculas de agua en esa zona.*

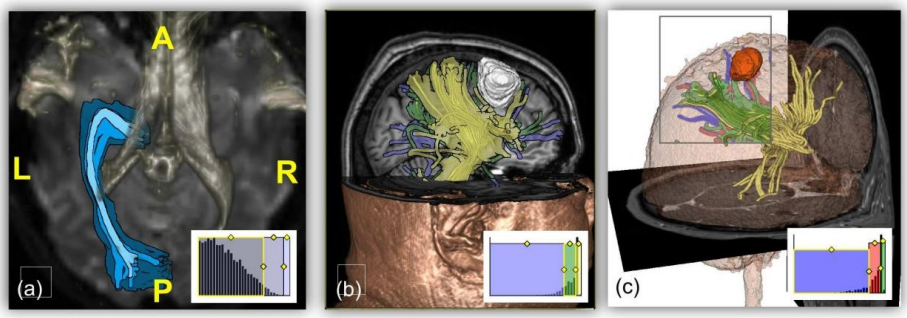


Figura 4.6: *Illustrative Uncertainty Visualization for DTI Fiber Pathways* [BRA11].

de un número muy menor de escaneos DWI [SP03]. Esta técnica, pese a reducir el tiempo necesario para la obtención de las DWI, requiere mucho tiempo para ejecutar el algoritmo. Otro algoritmo para calcular la incertidumbre es *Wild bootstrap* [D.08]. Este algoritmo genera variaciones aleatorias a partir de una sola imagen DWI. Este método se combina normalmente con una reconstrucción determinista para obtener las variaciones en la forma de la fibra.

- **Métodos de modelado matemático:** Un algoritmo de este grupo es el propuesto por Klein et al. [ea06], con el que generan variaciones sobre un solo DWI añadiendo niveles incrementales de *Gaussian noise*. Estos métodos tienen el inconveniente que asumen un modelo de ruido (*Gaussian noise*) los cuales son una simplificación del ruido real. Estos métodos no se suelen utilizar con una reconstrucción determinista.

4.2.3. Visualización de la incertidumbre

En el campo de la visualización de la incertidumbre de fibras queremos destacar el trabajo de Brecheisen [BRA11]. Ellos utilizan un algoritmo *wild bootstrap* para calcular la incertidumbre de las fibras y a través de una interfaz de usuario permiten visualizarla, agrupando las fibras en niveles de incertidumre y permitiendo modificar estos grupos de forma interactiva. Podemos ver los resultados en la figura 4.6.

Nuestro objetivo es proporcionar nuevos modos de visualización de la incertidumbre usando la GPU.

Capítulo 5

Técnica de Visualización

5.1. Objetivos

Queremos diseñar un algoritmo que visualice un grupo de fibras cerebrales utilizando técnicas de renderizado no fotorrealista. El objetivo es poder visualizar la orientación de las fibras y poder distinguir los diferentes niveles de incertidumbre. Para ello queremos utilizar un texturado orientado a partir de la dirección de las fibras. Este texturado debe permitir distinguir los diferentes niveles de incertidumbre utilizando diferentes tonos para cada uno de los niveles. El resultado de este algoritmo debe ser visualmente atractivo y debe dar una apariencia de haber sido realizado a lápiz.

Al implementar nuestro algoritmo sobre la aplicación DTITool, nos proporcionan muchas funcionalidades. Esta aplicación nos proporciona el cargado de fibras y de los valores de incertidumbre. Estos valores de incertidumbre están calculados utilizando el algoritmo *Wild bootstrap* y se basa en la distancia de cada fibra a una *mean fiber*, que representa a un grupo de ellas. También nos facilita una interfaz de usuario para definir los rangos de incertidumbre que queremos visualizar (*QHistogramWidget*).

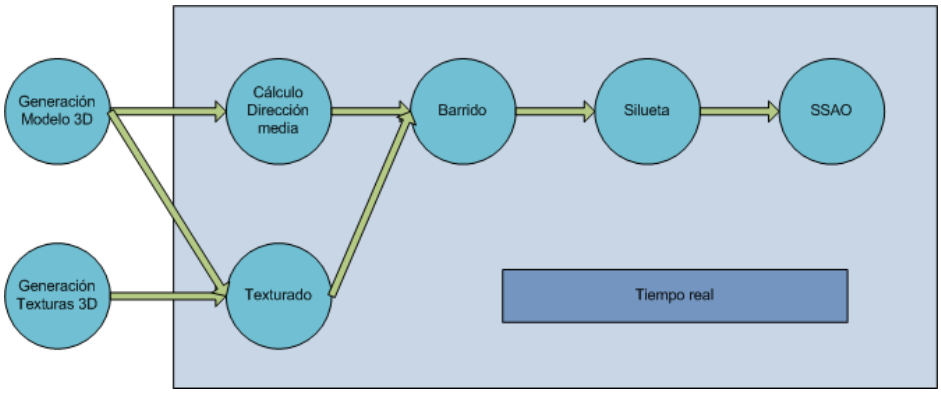


Figura 5.1: Proceso de visualización.

5.2. Descripción General

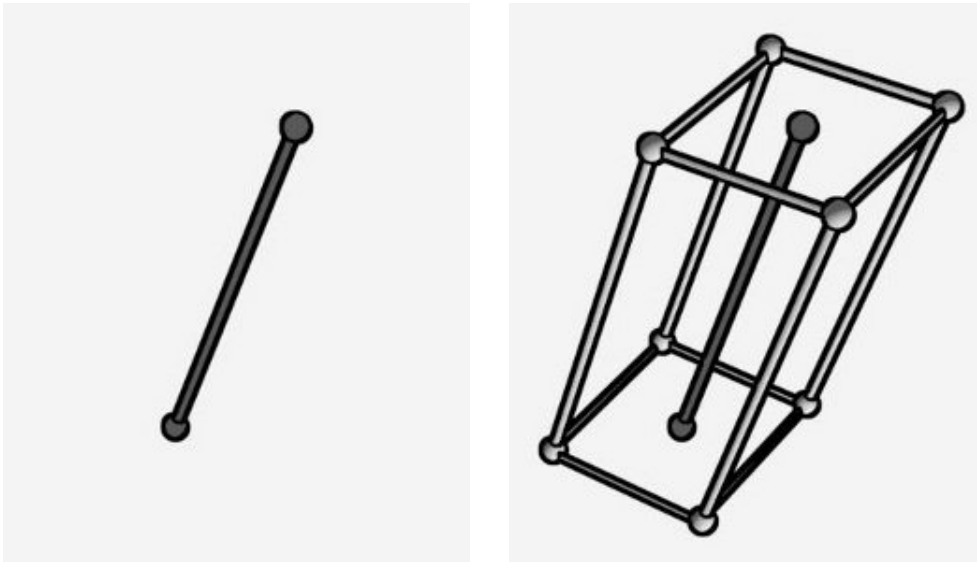
A continuación damos una pequeña descripción del proceso de visualización (Imagen 5.1):

- **Generación del Modelo 3D:** Pre-proceso en el que se genera un modelo 3D a partir de una lista de líneas que representan las diferentes fibras del cerebro.
- **Generación de Textura 3D:** Pre-proceso en el que se genera una textura 3D utilizada para visualizar el modelo de fibras.
- **Algoritmo de visualización:** A partir del modelo y la textura se genera la imagen en varios pasos de rendering:
  - **Cálculo de direcciones:** En el paso inicial se calcula una dirección media en espacio de pantalla para, en un paso posterior, poder orientar la textura.
  - **Texturado :** En este paso se utiliza la posición 3D del modelo para obtener el color de cada pixel a través de la textura 3D.
  - **Barrido:** En el paso final se utiliza la dirección media calculada en el primer paso para desplazar el color de cada pixel en espacio de pantalla.
- **Técnicas adicionales:** Esta nueva técnica se combina con varias técnicas existentes para obtener el resultado final:
  - **Siluetas:** Esta técnica se utiliza para resaltar los contornos del modelo y dar al usuario más información de su forma.

Figura 5.2: Modelo de fibras visualizado con dos niveles de incertidumbre. Las zonas de color azul tienen un nivel de incertidumbre más alto que las zonas de color naranja.

- **Screen Space Ambient Occlusion:** Esta técnica resalta la forma 3D del modelo oscureciendo las partes ocluidas del modelo.

En la imagen 5.2 podemos ver el resultado de aplicar este proceso de visualización sobre un modelo de fibras con dos niveles de incertidumbre.



(a) Segmento (b) Geometría

Figura 5.3: Generación de geometría a partir del segmento.

5.3. Generación del Modelo 3D

Los datos que recibimos de la aplicación principal y que representan las diferentes fibras del cerebro son una lista de líneas, formada cada una por una lista de segmentos (Figura 5.3 (a)). Estas líneas no disponen de un volumen para ser visualizadas, por lo que generamos un modelo 3D a partir de ellas.

Este modelo se genera formando cuatro rectángulos por cada segmento de línea, desplazando los vértices con dos vectores perpendiculares entre sí y, perpendiculares a su vez, con la dirección del segmento (tal como se muestra en la figura 5.3 (b)). En el inicio y final de cada línea, se crea un rectángulo para tapar el inicio y final de la línea.

A cada uno de estos vértices generados se le asigna un vector que se corresponde con la dirección del segmento. Esta será la dirección que utilizaremos más adelante para orientar nuestra textura.

Cada una de las fibras generadas se distribuyen entre diferentes modelos, dependiendo del rango de incertidumbre en el que se encuentra la fibra. De esta manera tenemos un modelo por cada nivel de incertidumbre.

Este algoritmo se ejecuta una sola vez antes de que el modelo sea visualizado, por lo que no se repite hasta que se carga otro modelo o se modifican los rangos de incertidumbre.

La geometría resultante de ejecutar este algoritmo sobre una lista de líneas se muestra en la imagen 5.4.

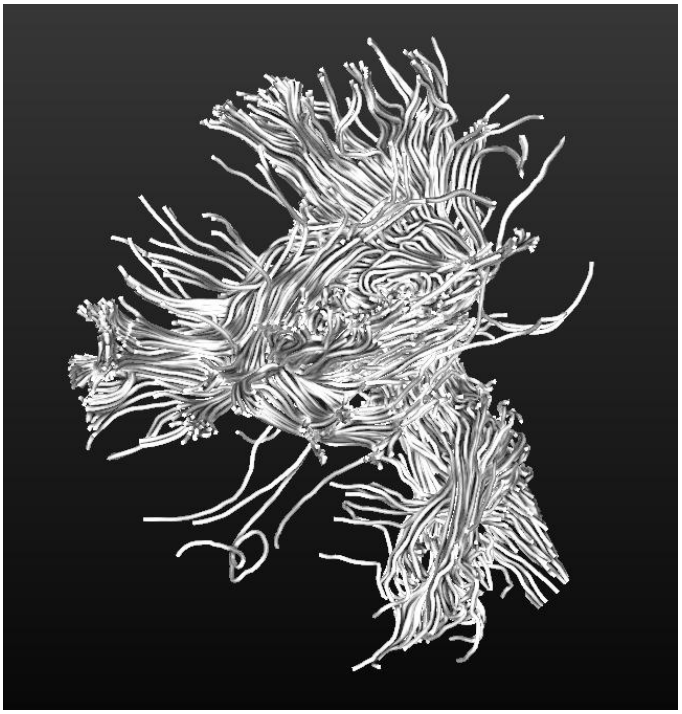


Figura 5.4: Modelo final utilizado para representar la lista de fibras.

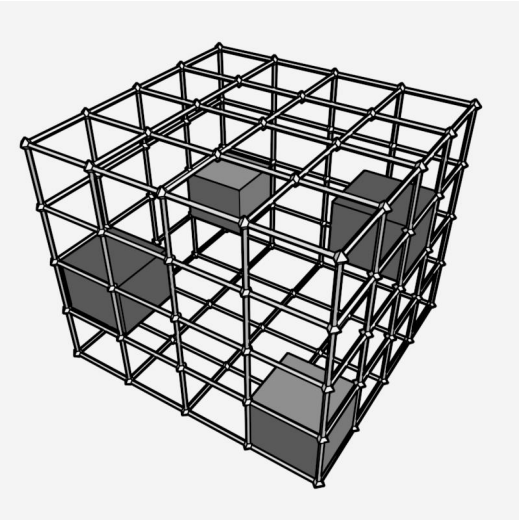


Figura 5.5: Textura 3D.

(a) Nivel 1 (b) Nivel 2

Figura 5.6: Rangos de tonos para dos niveles de incertidumbre diferentes.

5.4. Textura 3D

Para texturar el modelo utilizamos una textura 3D generada aleatoriamente, una para cada nivel de incertidumbre. Estas texturas se inicializan con color blanco para todos los pixels. Una vez inicializadas, se escogen aleatoriamente diferentes pixels de las texturas y se actualizan con un tono de gris escogido también aleatoriamente, tal como se muestra en la imagen 5.5. Cada nivel de incertidumbre tiene un rango de grises diferente para poder diferenciarlos. Podemos ver un ejemplo de los rangos de grises permitidos para dos niveles diferentes en la imagen 5.6. Estos niveles de grises se mezclan más adelante con el color del nivel de incertidumbre para componer la imagen final.

El problema de las texturas 3D, es que ocupan mucha memoria, y al tener una por cada nivel de incertidumbre se hace impracticable para el hardware actual. Por eso hemos diseñado un sistema para almacenar estas texturas reduciéndolas a texturas 2D.

Cada pixel de estas texturas 2D contiene 3 valores que se utilizan más ade-

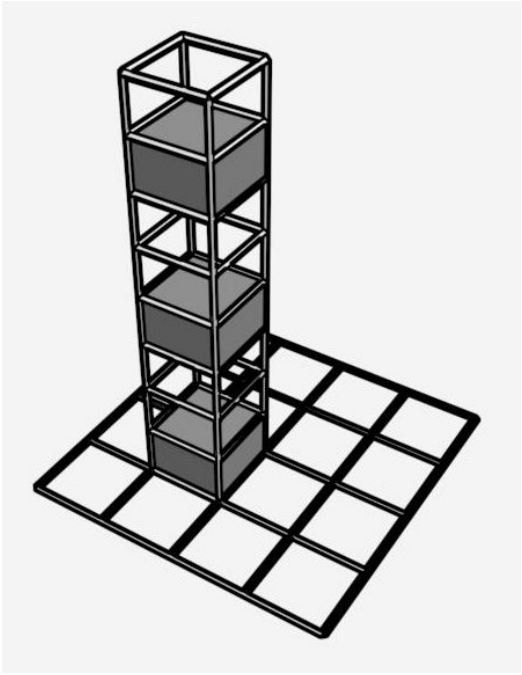


Figura 5.7: Textura 3D comprimida.

lante para calcular el valor final de cada columna vertical de pixels (Imagen 5.7):

- **Canal Rojo:** En este canal se almacena el nivel de gris del pixel, entre 0.0 y 1.0.
- **Canal Azul:** En este canal se almacena la distancia entre diferentes puntos grises a lo largo de esta columna de pixels. Este valor está comprendido entre 0.0 y 1.0, donde 0.0 es el valor mínimo del eje y de la caja contenedora del modelo y 1.0 el valor máximo.
- **Canal Verde:** En este canal se almacena el tamaño de cada punto gris. Este valor está codificado de la misma manera que el valor del canal verde.

De esta manera podemos tener almacenada una textura 3D ocupando el espacio de una textura 2D, a cambio de un pequeño cálculo a la hora de pintar. El trozo de código de shader necesario para calcular el color final del pixel a partir de esta codificación es el siguiente:

```
1   //Obtenemos el valor de la textura
2   vec4 texture =  texture2D(rndTexture,texCoord.xz);
```

```
3   vec4 color = vec4(1.0);
4   //Si es una columna de pixels grises
5   if(texture.r<0.999){
6       //Calculamos si el pixel corresponde a un punto gris
7       float rest = (texCoord.y/texture.b);
8       rest = rest -  floor(rest);
9       float porcFull = texture.g/texture.b;
10      if(rest<porcFull)
11          color = vec4(texture.r,texture.r,texture.r,1.0);
12  }
```

5.5. Algoritmo

Una vez tenemos el modelo y las diferentes texturas, podemos empezar el proceso de visualización. Este proceso consta de tres pasos: Cálculo de dirección media, Texturado y Barrido.

5.5.1. Cálculo de dirección media

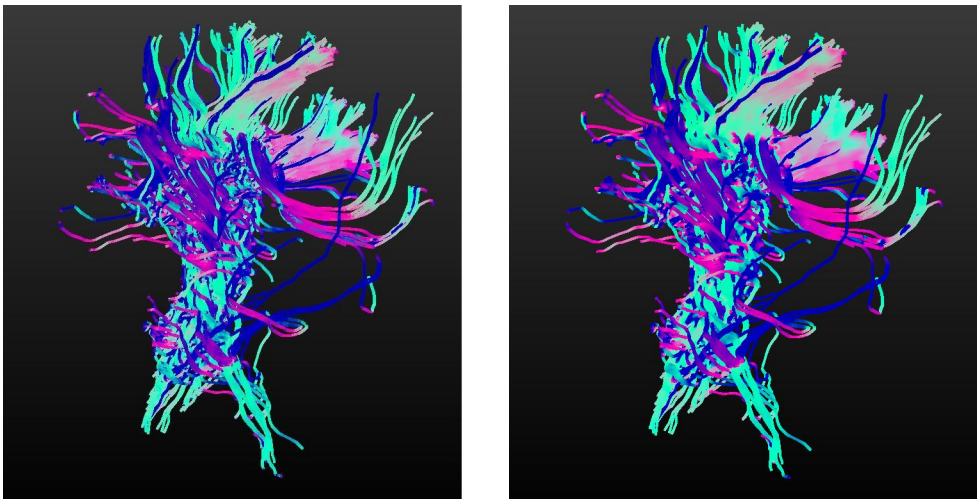
Cada segmento de fibra tiene una dirección. Esta dirección es la que se utiliza para orientar las texturas. El problema es que puede haber zonas en las que segmentos cercanos tengan direcciones muy diferentes, provocando que el texturado no sea uniforme.

Por eso, en este paso del proceso de visualización, se realiza una media en espacio de pantalla de las diferentes direcciones de pixels vecinos. De esta manera el texturado es más uniforme en estas zonas de direcciones diferentes.

Esta media se realiza recorriendo los pixels vecinos al pixel para el que estamos calculando la dirección. Hemos de tener en cuenta que los pixels vecinos que no tienen una profundidad cercana a la profundidad del pixel actual, no deberían contribuir a la media final, ya que si no segmentos lejanos en el espacio desviarían el resultado final.

El código de shader para realizar esta media es el siguiente:

```
1  //Obtenemos la direccion y la profundidad inicial
2  vec4 currentDirection =  texture2D(directionTex,coord);
3  vec4 currentDepth =  texture2D(depthTex,coord);
4  vec2 auxDirection = currentDirection.xy;
5  //Realizamos la media
6  for(int i = -kernelSize; i <= kernelSize; i++)
7  {
8     for(int j = -kernelSize; j <= kernelSize; j++)
9     {
10       vec2 auxCoord =  vec2((gl_FragCoord.x+float(i))/
11         screenWidth,(gl_FragCoord.y+float(j))/screenHeight);
12       vec4 auxDepth =  texture2D(depthTex, auxCoord);
13       vec4 direction2 =  texture2D(directionTex, auxCoord);
14       if( abs(currentDepth.w-auxDepth.w)<depthThreshold)
15          auxDirection += direction2.xy;
16    }
17 }
18 auxDirection =  normalize(auxDirection);
```



(a) Direcciones (b) Media de Direcciones

Figura 5.8: Resultado de realizar la media de las direcciones.

Podemos ver el resultado de esta media en las imágenes 5.8.

5.5.2. Texturado

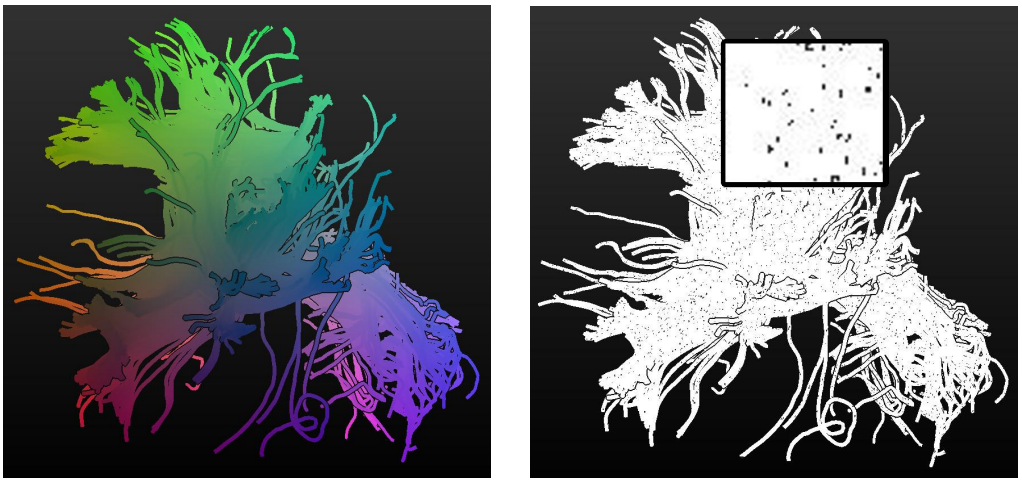
Para poder determinar el color de cada pixel sobre el que se proyecta el modelo de fibras, se ha de realizar una correspondencia entre la posición 3D del modelo y un pixel dentro de la textura.

En nuestro caso la correspondencia es muy sencilla. La coordenada más pequeña de la caja contenedora del modelo 3D se corresponde con la coordenada más pequeña de la textura 3D, y la coordenada más grande de la caja se corresponde con la coordenada más grande de la textura 3D.

El trozo de código que genera la coordenada de textura a partir de la posición 3D del modelo es el siguiente:

```
1   texCoord.xyz = (gl_Vertex.xyz - aabbMin)/(aabbMax - aabbMin);
```

En la imagen 5.9 (a) se puede ver el valor de la coordenada de textura para cada uno de los puntos del modelo 3D y en la imagen 5.9 (b) se puede ver el resultado de aplicar una textura de las generadas en el pre-proceso utilizando estas coordenadas de textura.



(a) Coordenadas de textura (b) Modelo texturado

Figura 5.9: Resultado de aplicar las coordenadas de textura para texturar el modelo.

5.5.3. Barrido

En el paso final del proceso se utiliza la información calculada anteriormente para realizar la imagen final.

Este último paso recorre cada uno de los pixels y obtiene la dirección calculada en ese punto. Una vez tenemos la dirección del pixel, la utilizamos para recorrer los pixels cercanos en esa dirección y nos quedamos con el tono gris de valor más bajo de todos.

De esta manera, expandimos los colores de la textura a través de la dirección en la que se desplazan las fibras, dando al usuario información sobre la forma y dirección de las fibras del modelo.

En este recorrido tenemos que tener en cuenta la profundidad de cada pixel vecino que consultamos. Pixels vecinos con una gran diferencia de profundidad con el pixel actual, no debemos tenerlos en cuenta a la hora de realizar el barrido. Si no estaríamos transmitiendo colores entre fibras lejanas en el espacio, y esto provocaría errores en la visualización a la hora de desplazar la cámara.

Una vez tenemos el tono de gris que corresponde a este pixel, lo mezclamos con el color del nivel de incertidumbre para poder diferenciarlos. La ecuación 5.1 es la utilizada para realizar el *blending* entre el tono de gris y el color del nivel de incertidumbre.

$$color\_final = (tono\_gris * 0.5) + (color\_nivel * 0.5) \tag{5.1}$$

El código para calcular el color final es el siguiente:

```
1  //Obtenemos la direccion y la profundidad del pixel
2  vec4 currentTexel =  texture2D(colorTex,coord);
3  vec4 currentDir =  texture2D(directionTex,coord);
4  vec4 color =  vec4(currentTexel.rgb,1.0);
5  //Realizamos la busqueda
6  for(int i = -kernelSize; i <= kernelSize; i++)
7  {
8     float auxI =  float(i);
9     vec2 auxCoord =  vec2((gl_FragCoord.x+(currentDir.x*auxI))/
10      screenWidth,(gl_FragCoord.y+(currentDir.y*auxI))
11      /screenHeight);
12    vec4 auxColor = texture2D(colorTex,auxCoord);
13    if(( abs(currentTexel.w-auxColor.w)<depthThreshold)
14      && (auxColor.r<color.r))
15       color = vec4(auxColor.r,auxColor.r,auxColor.r,1.0);
16 }
17 if(color.r <1.0)
18    color = color*0.5+0.5*vec4(uncertainColor.rgb,1.0);
```

En la imagen 5.10 podemos ver el resultado final de aplicar este proceso.

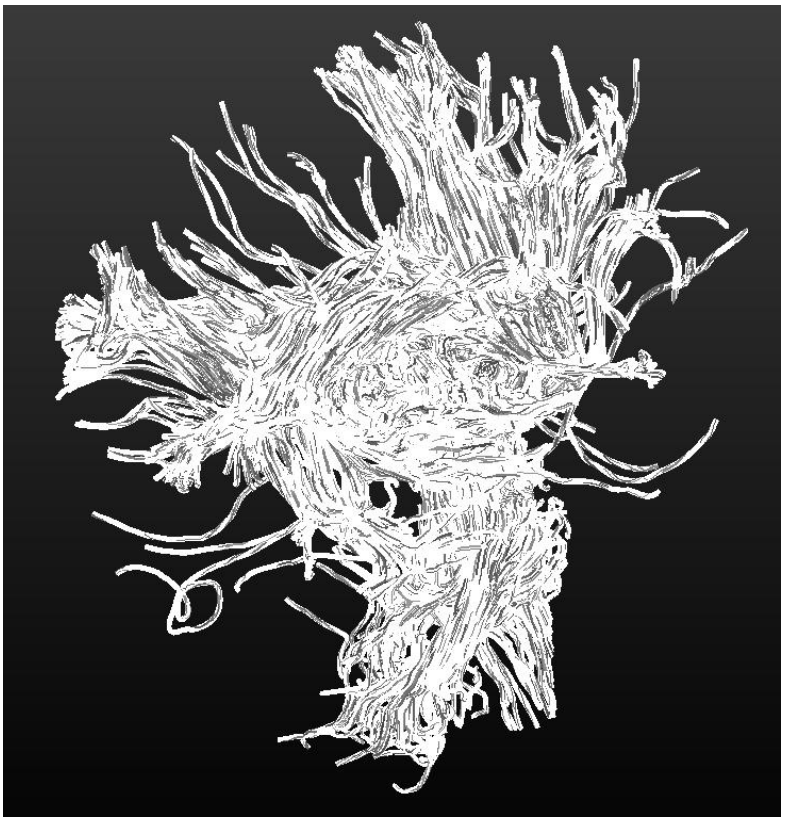
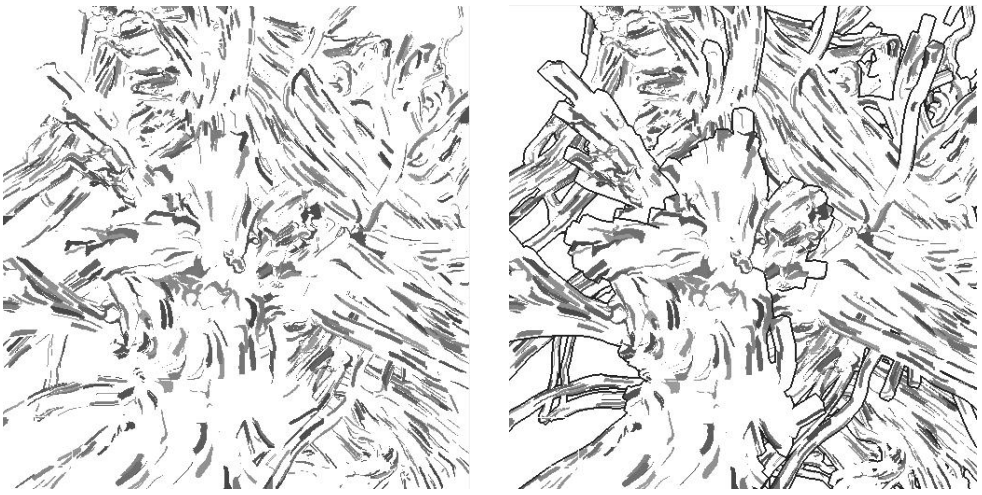


Figura 5.10: Modelo texturado.



(a) Sin silueta (b) Con silueta

Figura 5.11: Imagen renderizada con y sin silueta.

5.6. Siluetas

5.6.1. Introducción

El renderizado de siluetas es una técnica muy común dentro de la visualización ilustrativa, ya que es una herramienta que ayuda a destacar información del modelo que otras técnicas pasan por alto (Figura 5.11).

En el contexto de las mallas triangulares, la silueta está formada por las aristas compartidas por dos triángulos, uno de los cuales está orientado a la cámara (es visible desde el punto de vista) y el otro no está orientado a la cámara (no es visible desde el punto de vista). Esto provoca que la silueta se haya de calcular para cada nuevo punto de vista.

La extracción de silueta es un tema que ha sido estudiado en profundidad. Existen numerosos algoritmos para la detección de la silueta, y todos ellos se dividen en dos categorías: algoritmo en espacio de pantalla y algoritmos en espacio de objeto.

De los algoritmos en espacio de objeto podemos destacar el de John W. Buchanan y Mario C. Sousa, en el que construían una estructura de datos auxiliar en un pre-cálculo de la malla para facilitar la detección de la silueta en tiempo de ejecución [BS00].

Con la aparición de nuevo hardware gráfico se han desarrollado nuevos algoritmos que aceleran la detección de la silueta. Un algoritmo híbrido, tanto en espacio de objeto como en espacio de pantalla, que aprovecha los últimos avances en las tecnologías gráficas, es el propuesto en mi proyecto

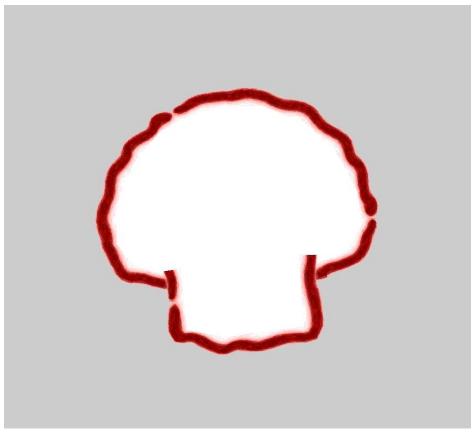


Figura 5.12: Single Pass GPU Stylized Edges.

final de carrera de Ingeniería Técnica en Informática de Gestión [HV09]. Este algoritmo detecta la silueta en espacio de objeto, haciendo uso del Geometry Shader, y genera coordenadas de textura en espacio de pantalla. Podemos ver el resultado de aplicar este algoritmo en la imagen 5.12.

Un algoritmo a destacar, que se calcula en espacio de pantalla, es el T. Saito y T. Takahashi, que obtenían la silueta a partir de la textura de profundidad utilizando un post-proceso [ST90]. Este es el algoritmo que hemos decidido implementar debido a su bajo coste computacional que es independiente de la complejidad de la escena.

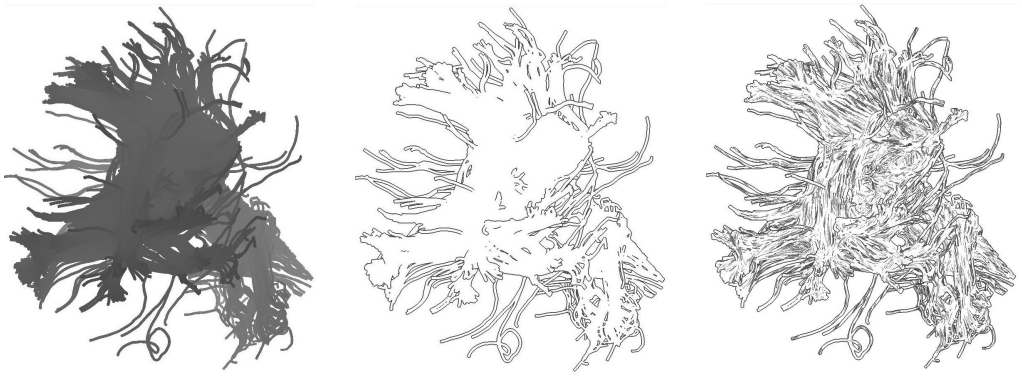
5.6.2. Algoritmo

En un primer paso, se almacena en un buffer la profundidad de la escena calculada de forma lineal (Figura 5.13 (a)), ya que la profundidad utilizada por el proceso de visualización no está definida de forma lineal, sino que se utilizan más bits para definir la profundidad de objetos más cercanos a la cámara.

Una vez tenemos almacenadas las profundidades, se realiza un segundo paso sobre este buffer y para cada pixel, se compara su profundidad con la profundidad de los pixels vecinos. Si algún pixel vecino tiene una diferencia de profundidad más grande que un cierto valor, marcamos este pixel como perteneciente a la silueta. Podemos ver el resultado de aplicar este proceso en las imágenes 5.13.

El código para calcular la silueta es el siguiente:

```
1   //Obtenemos el valor inicial
```



(a) Buffer de Profundidad (b) Silueta (c) Silueta + Textura

Figura 5.13: Proceso de detección de silueta.

```
2   vec4 currentDepth =  texture2D(depthTex,coord);
3   //Cálculo de silueta
4   for(int i = -kernelSize; i <= kernelSize; i++)
5   {
6      for(int j = -kernelSize; j <= kernelSize; j++)
7      {
8         vec2 auxCoord =  vec2((gl_FragCoord.x+float(i))/
9           screenWidth,(gl_FragCoord.y+float(j))/screenHeight);
10        vec4 auxDepth =  texture2D(depthTex, auxCoord);
11        if( length( vec2(i,j)) <= float(outlineSize))
12           if((currentDepth.w - auxDepth.w)
13             >outlieDepthThreshold)
14              //Pixel pertenece a la silueta
15     }
16  }
```

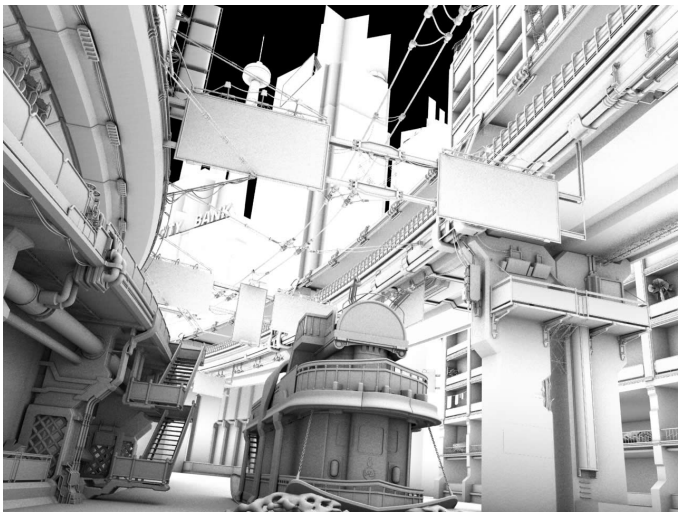


Figura 5.14: Ambient Occlusion.

5.7. Screen Space Ambient Occlusion

5.7.1. Introducción

Ambient Occlusion (AO) es una técnica de pintado que simula la aportación de iluminación indirecta de la escena. Este algoritmo fue desarrollado inicialmente por Zhukov con el nombre de obscurances [ZSK98], y se basa en un pre-proceso independiente del punto de vista en el que por cada triángulo se lanzan una serie de rayos y se comprueba cuántos de ellos intersecan con otros triángulos. Podemos ver el resultado de aplicar este algoritmo en la imagen 5.14.

Con el tiempo y los avances en el hardware gráfico aparecieron nuevos algoritmos para aproximar este cálculo en tiempo real. Todos estos algoritmos se basan en un post-proceso sobre un buffer de profundidad. El algoritmo que aproxima la oclusión ambiente usando cálculos en espacio de pantalla fue desarrollado por la compañía Crytek en 2007 para aplicarlo en su juego Crysis (Figura 5.15). Este algoritmo consta de tres pasos:

- **Cálculo por pixel:** Se utiliza una lista de vectores aleatorios para realizar una comparación de profundidades para calcular una aproximación del factor de oclusión.
- **Blurring:** Una vez calculada la aproximación del factor de oclusión, se realiza un blurring, en dos pasos (uno horizontal y otro vertical), puesto que el cálculo se ha hecho muestreando solamente una serie de vecinos, no todos.

Figura 5.15: Screen Space Ambient Occlusion.

5.7.2. Algoritmo

El algoritmo que hemos implementado es una aproximación del algoritmo propuesto por Crytek. El algoritmo de Crytek utilizaba una lista de vectores aleatorios para desplazarse en espacio de pantalla y consultar la profundidad en estos pixels. Esto provocaba ruido en el resultado obtenido y este es el motivo para aplicar los pasos de blurring.

Debido a la naturaleza de nuestra escena podemos sustituir la lista de vectores aleatorios por un recorrido uniforme por los pixels cercanos, ya que la geometría está formada por fibras muy pequeñas y las diferencias de profundidad las podemos encontrar en los pixels cercanos. Este cambio en el algoritmo evita la aparición de ruido, por lo que podemos eliminar los pasos de blurring.

El cálculo del factor de oclusion que provoca un pixel sobre otro se calcula utilizando la ecuación 5.7, donde $currentDepth$ es la profundidad del pixel actual y $auxDepth$ es la profundidad del pixel vecino. Esta ecuación está definida por rangos de valores definidos por el usuario ($depthThreshold$, $minDepthDistance$ y $maxDepthDistance$).

$$Diff_1 = currentDepth - auxDepth \tag{5.2}$$

$$Diff_2 = Diff_1 - depthThreshold \tag{5.3}$$

$$Diff_3 = Diff_2 - minDepthDistance \tag{5.4}$$

$$AO_1 = (minDepthDistance - Diff_2)/minDepthDistance \tag{5.5}$$

$$AO_2 = 1.0 - ((maxDepthDistance - Diff_3)/maxDepthDistance) \tag{5.6}$$

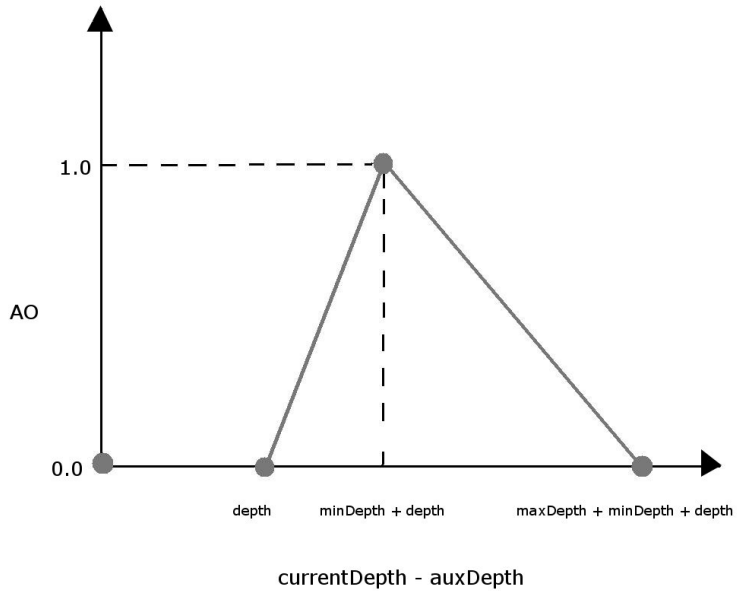


Figura 5.16: Factor de occlusión.

$$AO = \begin{cases} 0.0, & Diff_1 < depthThreshold \\ AO_1, & Diff_2 <= minDepthDistance \\ AO_2, & Diff_3 <= maxDepthDistance \\ 0.0, & Diff_3 > maxDepthDistance \end{cases} \tag{5.7}$$

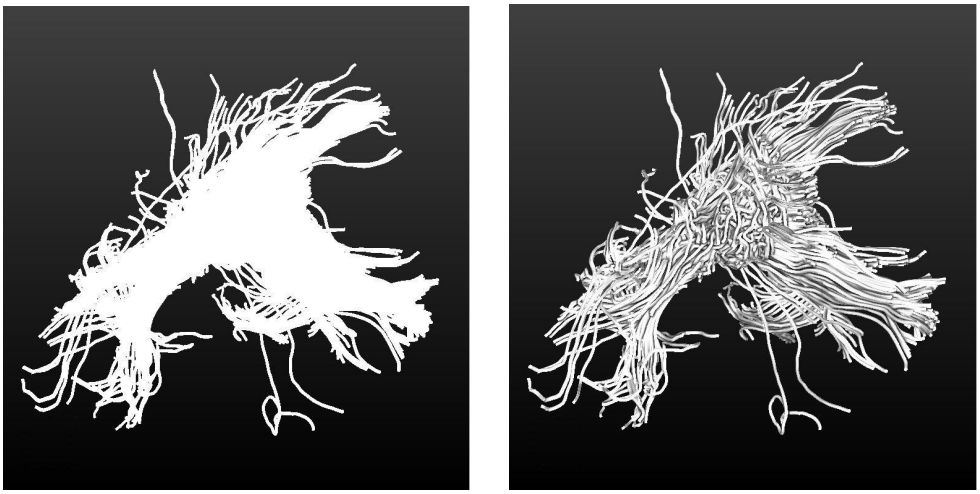
Podemos ver una gráfica del factor de oclusión, obtenido en función de la diferencia de profundidad entre dos pixels, en la gráfica 5.16.

Una vez tenemos todos los factores de oclusión de los pixels cercanos realizamos una media de todos y el resultado es el factor de oclusión para ese pixel.

Podemos ver el resultado de aplicar este algoritmo en las imágenes 5.17.

El código que se encarga de calcular el factor de oclusión para cada uno de los pixels es el siguiente:

```
1   //Obtenemos la profundidad del pixel actual
2   float currentDepth =  texture2D(depthBuffer,coord).x;
3   float ao = 0.0;
```



(a) Sin SSAO (b) Con SSAO

Figura 5.17: Resultado obtenido con el algoritmo de SSAO.

```
4   //Iteramos por los pixels vecinos
5   for(int i = -kernelSize; i <= kernelSize; i++)
6   {
7      for(int j = -kernelSize; j <= kernelSize; j++)
8      {
9         //Nos saltamos el pixel actual
10        if(i != 0 || j != 0)
11        {
12           //Obtenemos la profundidad del pixel vecino
13           vec2 desp =  vec2(i,j);
14           float auxDepth =  texture2D(depthBuffer
15             ,vec2((gl_FragCoord.x+desp.x)/screenWidth
16             ,(gl_FragCoord.y+desp.y)/screenHeight)).x;
17           //Calculamos el factor de occlusion
18           float diff = max(currentDepth - auxDepth,0.0);
19           diff = diff - depthThreshold;
20           if(diff >= 0.0){
21              if(diff <= minDepthDistance)
22                 ao += (minDepthDistance - diff)
23                   /minDepthDistance;
24              else{
25                 diff -= minDepthDistance;
26                 ao += 1.0-((maxDepthDistance -
27                   min(diff,maxDepthDistance))
```

```
28                      /maxDepthDistance);
29                  }
30              }else
31                  ao += 1.0;
32          }
33      }
34  }
35  //Calculamos el factor final de occlusion
36  ao = ao/float(numSum);
```

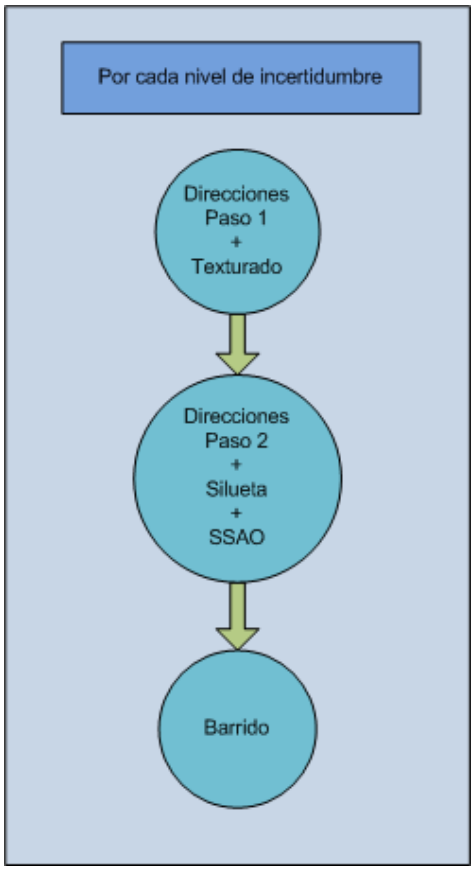


Figura 5.18: Proceso de visualización.

5.8. Optimizaciones

Este proceso de visualización requiere muchos pasos de pintado y se tiene que repetir por cada nivel de incertidumbre, ya que la aplicación da la opción de activar y desactivar interactivamente los diferentes niveles de incertidumbre. El algoritmo también realiza un uso considerable de memoria de video, ya que necesita uno o varios buffers auxiliares para cada paso de rendering.

Para reducir estos costes, hemos agrupado los pasos de tal manera que el coste sea mínimo. Podemos ver el proceso en el diagrama 5.18.

A continuación se describe cada uno de los pasos del proceso de visualización individualmente.

- **Direcciones Paso 1 + Texturado:** En este paso del proceso se dibuja la geometría correspondiente a un nivel de incertidumbre y, para cada pixel sobre el que se proyecta esta geometría, se almacena la dirección del segmento, la profundidad y se realiza el texturado con la textura

correspondiente a este nivel de incertidumbre. Toda esta información se almacena en dos buffers auxiliares. Podemos ver la distribución de valores en los buffers en la tabla 5.1.

Buffer 1 (xyz)	Color de Textura (rgb)
Buffer 1 (w)	Profundidad lineal
Buffer 2 (xy)	Dirección en espacio de pantalla (xy)
Buffer 2 (z)	Profundidad

Cuadro 5.1: Distribución de los valores en los buffers.

Los shaders que se ejecutan en este paso del proceso de visualización son los siguientes:

- **Vertex Shader:**

```
1  varying vec4 texCoord;
2  varying vec2 direction;
3
4  uniform float zNear;
5  uniform float zFar;
6
7  uniform vec3 aabbMin;
8  uniform vec3 aabbMax;
9
10 void main(void)
11 {
12    //Calculamos la profundidad lineal.
13    vec4 auxVertex = gl_ModelViewMatrix * gl_Vertex;
14    texCoord.w = (-auxVertex.z - zNear)
15      /(zFar - zNear);
16    //Calculamos la coordenada de textura.
17    texCoord.xyz = (gl_Vertex.xyz - aabbMin)
18      /(aabbMax - aabbMin);
19    //Calculamos la direccion.
20    vec4 position1 = gl_ModelViewProjectionMatrix *
21      gl_Vertex;
22    vec4 position2 = gl_ModelViewProjectionMatrix *
23      (gl_Vertex+vec4(gl_MultiTexCoord0.xyz,0.0));
24    gl_Position = position1;
25    position1 /= position1.w;
```

```
26      position2 /= position2.w;
27      direction = normalize(position2.xy - position1.xy);
28  }
```

- **Pixel Shader:**

```
1   varying vec4 texCoord;
2   varying vec2 direction;
3
4   uniform sampler2D rndTexture;
5
6   void main(void)
7   {
8      //Obtenemos el valor de la textura
9      vec4 texture = texture2D(rndTexture,texCoord.xz);
10     vec4 color = vec4(1.0);
11     //Si es una columna de pixels grises
12     if(texture.r <0.9){
13        //Calculamos si el pixel corresponde
14        //a un punto gris
15        float rest = (texCoord.y/texture.b);
16        rest = rest -  floor(rest);
17        float porcFull = texture.g/texture.b;
18        if(rest<porcFull)
19            color = vec4(texture.r,texture.r
20              ,texture.r,1.0);
21     }
22     gl_FragData[0] = vec4(color.rgb,texCoord.w);
23     gl_FragData[1] = vec4(direction
24       ,gl_FragCoord.z,0.0);
25  }
```

- **Direcciones Paso 2 + Silueta + SSAO:** En esta parte del proceso se dibuja un rectángulo que ocupa toda la pantalla y se calcula la dirección media de cada pixel iterando sobre los pixels vecinos. Como se realiza un recorrido sobre los pixels vecinos, se aprovecha este proceso para calcular la silueta y el factor de oclusión. De esta manera solo realizamos un acceso a textura por cada pixel vecino. Estos valores se almacenan en un buffer auxiliar (Tabla 5.2).

Buffer (xy)	Dirección media (xy)
Buffer (z)	Profundidad
Buffer (w)	Factor de oclusión

Cuadro 5.2: Distribución de los valores en el buffer.

Estos son los shaders de este paso:

- **Vertex Shader:**

```
1  void main(void)
2  {
3     gl_Position = gl_Vertex;
4  }
```

- **Pixel Shader:**

```
1  uniform sampler2D directionTex;
2  uniform sampler2D depthTex;
3
4  uniform float screenWidth;
5  uniform float screenHeight;
6
7  uniform int outlineSize;
8  uniform float outlieDepthThreshold;
9
10 uniform float ssaoDepthThreshold;
11 uniform float ssaoMinDepthDistance;
12 uniform float ssaoMaxDepthDistance;
13
14 void main(void)
15 {
16    //Obtenemos los valores del pixel.
17    vec2 coord =  vec2(gl_FragCoord.x/screenWidth
18      ,gl_FragCoord.y/screenHeight);
19    vec4 currentDirection =  texture2D(directionTex
20      ,coord);
21    vec4 currentDepth =  texture2D(depthTex, coord);
22
23    vec2 auxDirection = currentDirection.xy;
24    float outlineDepth = 1.0;
```

```
25
26      float ao = 0.0;
27      int numSum = (kernelSize*2)+1;
28      numSum = (numSum*numSum)-1;
29
30      //Iteramos sobre los pixels vecinos.
31      for(int i = -kernelSize; i <= kernelSize; i++)
32      {
33          for(int j = -kernelSize; j <= kernelSize; j++)
34          {
35              //Obtenemos los valores del pixel vecino.
36              vec2 auxCoord =  vec2(gl_FragCoord.x+
37                float(i))/screenWidth,(gl_FragCoord.y+
38                float(j))/screenHeight);
39              vec4 auxDepth =  texture2D(depthTex, auxCoord);
40              vec4 direction2 =  texture2D(directionTex
41                ,auxCoord);
42
43              //Calculamos la direccion media.
44              if(abs(currentDepth.w - auxDepth.w)
45                <depthThreshold)
46                 auxDirection += direction2.xy;
47
48              //Calculamos la silueta.
49              if(length(vec2(i,j))<=float(outlineSize))
50                  if((currentDepth.w - auxDepth.w)
51                    >outlieDepthThreshold)
52                     outlineDepth = min(outlineDepth
48                      ,direction2.z);
53
54              //Calculamos el factor de oclusion.
55              if(i != 0 || j != 0)
56              {
57                  float diff = max(currentDepth-
58                    auxDepth,0.0);
59                  diff = diff - depthThreshold;
60                  if(diff >= 0.0){
61                      if(diff <= minDepthDistance)
62                          ao += (minDepthDistance - diff)
63                            /minDepthDistance;
64                      else{
```

```
65                      diff -= minDepthDistance;
66                      ao += 1.0-((maxDepthDistance -
67                       min(diff,maxDepthDistance))
68                       /maxDepthDistance);
69                  }
70              }else
71                  ao += 1.0;
72          }
73      }
74  }
75
76  auxDirection =  normalize(auxDirection);
77
78  ao = ao/ float(numSum);
79  ao =  pow(ao,3.0);
80
81  if(outlineDepth <1.0)
82      gl_FragColor = vec4(auxDirection
83        ,-outlineDepth,ao);
84  else
85      gl_FragColor = vec4(auxDirection
86        ,currentDirection.z,ao);
87  }
```

- **Barrido:** En el último paso del proceso se envía a pintar un rectángulo que ocupa toda la pantalla y se calcula el color final de cada pixel, realizando el proceso de barrido utilizando la dirección media calculada en el paso anterior. El color final se modula utilizando el factor de oclusión. Los shaders que se ejecutan en este paso final son:

  - **Vertex Shader:**

```
1  void main(void)
2  {
3      gl_Position = gl_Vertex;
4  }
```

  - **Pixel Shader:**

```
1  uniform sampler2D colorTex;
```

```
2   uniform sampler2D directionTex;
3
4   uniform float screenWidth;
5   uniform float screenHeight;
6
7   uniform int kernelSize;
8   uniform float depthThreshold;
9
10  uniform int ssaoEnabled;
11
12  uniform vec3 uncertainColor;
13
14  void main(void)
15  {
16     //Obtenemos los valores del pixel.
17     vec2 coord =  vec2(gl_FragCoord.x/screenWidth
18       ,gl_FragCoord.y/screenHeight);
19     vec4 currentTexel =  texture2D(colorTex,coord);
20     vec4 currentDir =  texture2D(directionTex,coord);
21
22     vec4 color =  vec4(currentTexel.rgb,1.0);
23
24     //Recorremos los pixels en la direccion calculada
25     //en el paso anterior.
26     for(int i = -kernelSize; i <= kernelSize; i++)
27     {
28        float auxI = float(i);
29        vec2 auxCoord = vec2((gl_FragCoord.x+(currentDir.x
30          *auxI))/screenWidth,(gl_FragCoord.y+
31          (currentDir.y*auxI))/screenHeight);
32        vec4 auxColor = texture2D(colorTex,auxCoord);
33
34        if((abs(currentTexel.w - auxColor.w)
35          <depthThreshold) && auxColor.r <color.r)
36           color = vec4(auxColor.r,
37             auxColor.r,auxColor.r,1.0);
38     }
39
40     //Calculamos el color final.
41     if(color.r <1.0)
42        color = color*0.5+0.5*vec4(uncertainColor.rgb,1.0);
```

```
43      if(currentDir.z <0.0){
44          gl_FragColor = vec4(uncertainColor.rgb,1.0);
45          gl_FragDepth = -currentDir.z;
46      }else{
47          if(ssaoEnabled == 1)
48              gl_FragColor = color*currentDir.w;
49          else
50              gl_FragColor = color;
51          gl_FragDepth = currentDir.z;
52      }
53  }
```

De esta manera reducimos el número de pasos solo a tres. El problema que tiene la aplicación es que el coste se incrementa con el número de niveles de incertidumbre que queremos visualizar a la vez, ya que se repite este proceso por cada uno de ellos.

Al agrupar los pasos de rendering y reducirlos a tres, el número de buffers auxiliares también queda reducido a tres.

5.9. Resultados

En la tabla 5.3 podemos ver los tiempos obtenidos en frames por segundo para dos resoluciones de pantalla diferentes. Estos tiempos han sido obtenidos recorriendo una matriz de 7x7 pixels vecinos para el cálculo de la dirección media, la silueta y el factor de oclusión. El coste del algoritmo depende de la resolución de pantalla, ya que el principal coste de este, está en los pasos de post-proceso. Estos datos han sido obtenidos para un modelo de 4.493 fibras y 1.500.000 triángulos.

Incertidumbre	**Resolución(500x500)**	**Resolución(1380x900)**
1 Nivel	116.009	38.2849
2 Niveles	71.377	22.1239
3 Niveles	53.135	15.6715
4 Niveles	42.1763	12.0192

Cuadro 5.3: Tabla de tiempos.

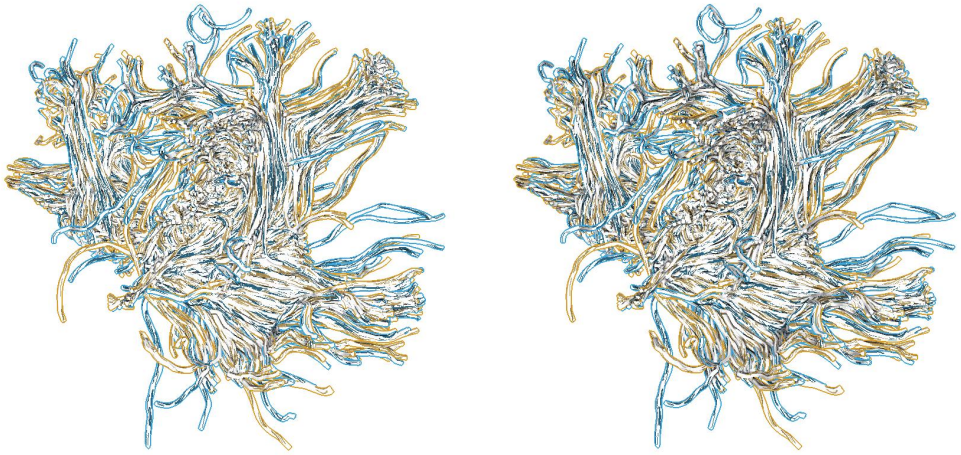
En la tabla 5.4 se muestran los resultados para un escenario igual que el anterior pero recorriendo una matriz de 5x5 pixels vecinos a la hora de realizar el cálculo de la dirección media, de la silueta y del factor de oclusión.

Incertidumbre	**Resolución(500x500)**	**Resolución(1380x900)**
1 Nivel	157.233	67.7461
2 Niveles	110.383	41.0625
3 Niveles	87.7192	28.0112
4 Niveles	71.9425	21.505

Cuadro 5.4: Tabla de tiempos.

Todos estos números han sido obtenidos con el siguiente hardware:

- **Procesador:** Intel Core 2 Duo 3 Ghz
- **Memoria RAM:** 8 Gb
- **Tarjeta gráfica:** NVidia GeForce GTX 280
- **Sistema operativo:** Windows 7



(a) Matriz 5x5 (b) Matriz 7x7

Figura 5.19: En la imagen (a) se muestra el resultado obtenido para una matriz de 5x5 pixels vecinos y en la imagen (b) el resultado para una matriz de 7x7 pixels vecinos.

En la imagen 5.19 se muestra una comparativa entre una imagen obtenida utilizando una matriz de 5x5 para consultar los pixels vecinos y una imagen obtenida utilizando una matriz de 7x7 para consultar los pixels vecinos. Podemos ver que para la matriz de 5x5 la imagen resultante es más clara que la resultante para la matriz de 7x7. Esto se debe a que en la primera imagen consultamos menos pixels vecinos que en la segunda durante el cálculo del factor de oclusión, y por lo tanto detectamos menos oclusores. El resultado obtenido para el cálculo de la dirección media es semejante.

En las imágenes 5.20 y 5.21 podemos ver el resultado de aplicar el algoritmo con diferentes configuraciones y con diferentes niveles de incertidumbre. Los colores seleccionados para estas imágenes los hemos escogido usando colores complementarios para las configuraciones de 2 niveles y colores lo más lejanos posibles en el círculo cromático para las configuraciones con 3 niveles.

Antes de llegar a obtener el resultado final realizamos numerosas pruebas ajustando los diferentes parámetros. Una de estas pruebas que realizamos fue aplicar el color del nivel de incertidumbre solo a la silueta, y a las fibras se le aplicaba el color que estaba almacenado en las texturas 3D. El resultado obtenido era muy atractivo a la vista, pero era muy difícil diferenciar entre diferentes niveles de incertidumbre. Podemos ver el resultado de estas pruebas en las figuras 5.22.

Para llegar a obtener buenos resultados con este algoritmo tuvimos que probar una serie de algoritmos que no nos dieron el resultado deseado. Uno

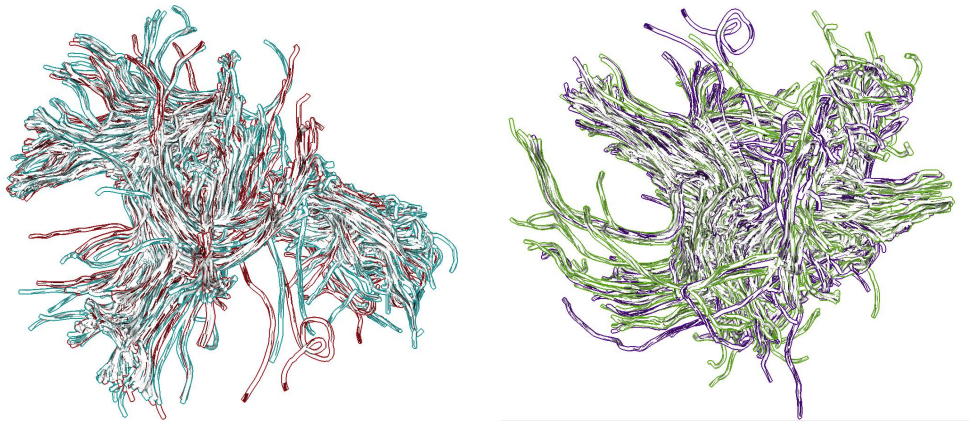


Figura 5.20: Imágenes obtenidas con dos niveles de incertidumbre. En la imagen (a) el color rojo codifica un nivel de incertidumbre alto y el color azul codifica un nivel de incertidumbre bajo, y en la imagen (b) el color lila codifica un nivel de incertidumbre alto y el color verde codifica un nivel de incertidumbre bajo.

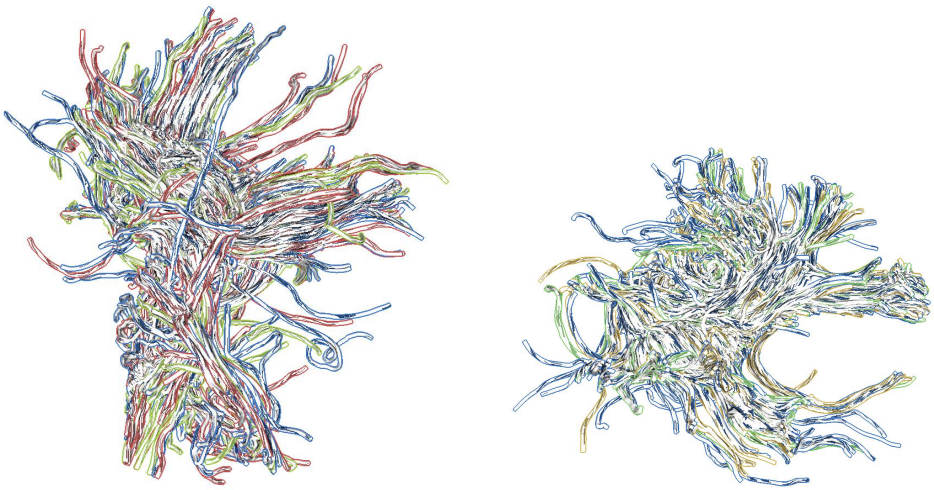


Figura 5.21: Imágenes obtenidas con tres niveles de incertidumbre. En la imagen (a) el color azul codifica el nivel de incertidumbre más alto, el color rojo codifica un nivel de incertidumbre medio y el color verde codifica el nivel de incertidumbre más bajo. En la imagen (b) el color azul codifica el nivel de incertidumbre más alto, el color naranja codifica un nivel de incertidumbre medio y el color verde codifica el nivel de incertidumbre más bajo.

(a) 1 Nivel (b) 2 Niveles

Figura 5.22: Resultado de aplicar directamente el color de las texturas sobre las fibras.

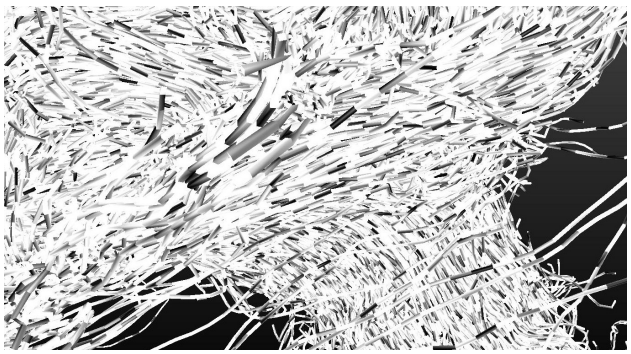
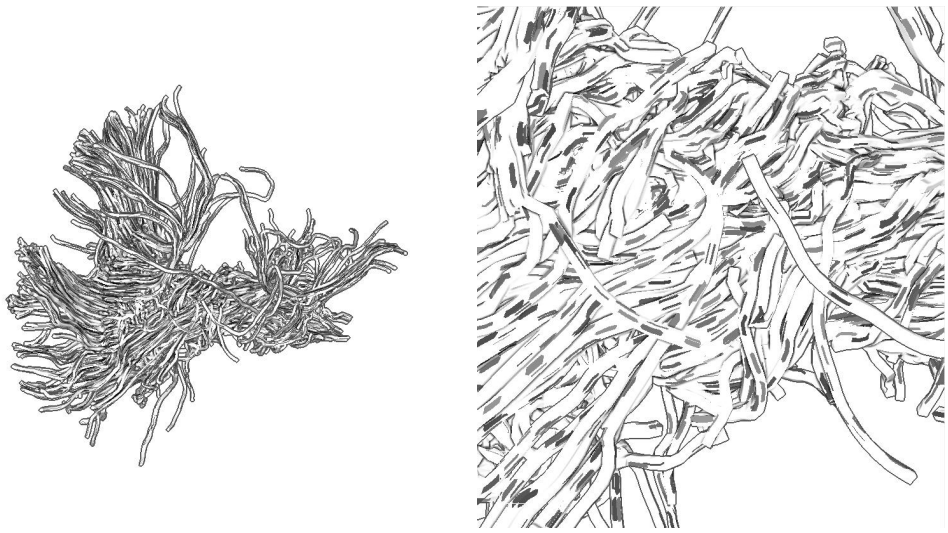


Figura 5.23: Resultado obtenido con una de las primeras versiones de nuestro algoritmo.

de ellos consistía en realizar un texturado en espacio de pantalla, orientando estas texturas utilizando las direcciones de las fibras. Pero el texturado no era continuo a lo largo de las fibras. Otro algoritmo fallido orientaba las texturas 3D utilizando la dirección de cada segmento de las fibras, pero producía discontinuidades entre segmentos continuos (Imagen 5.23).



(a) Imagen lejana (b) Imagen cercana

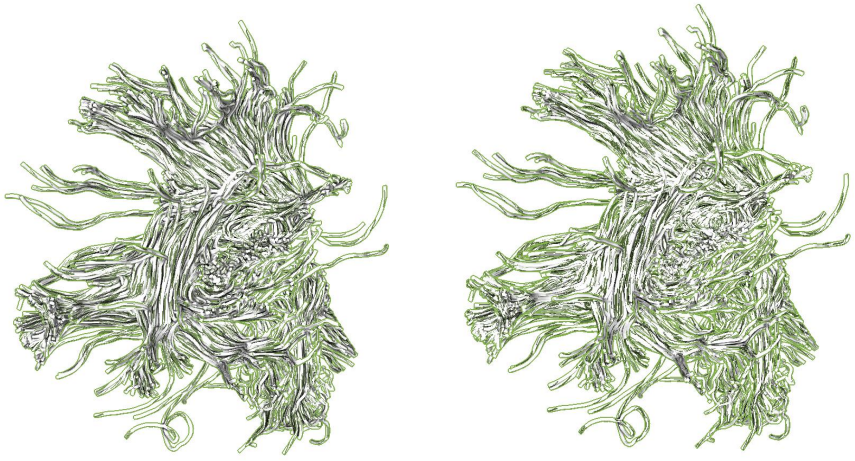
Figura 5.24: Zoom.

5.10. Posibles mejoras

- El paso de barrido se realiza en espacio de pantalla, y por lo tanto el tamaño de las líneas resultantes siempre es el mismo, independientemente de la distancia del observador. Esto provoca que cuando realizamos un zoom sobre el modelo nos dé la impresión que las líneas reducen su tamaño. La imagen 5.24 muestra un ejemplo de este suceso.

  Una posible solución podría ser desarrollar un sistema en el que las coordenadas de textura se multiplicaran por un factor calculado dependiendo de la distancia de la cámara al modelo. De esta manera las líneas siempre tendrían el mismo ancho y la misma longitud independientemente de la distancia a la que estamos visualizando el modelo. Aunque esta solución podría introducir nuevos problemas que provocarían un desplazamiento de la textura mientras realizamos un zoom.

- Al agrupar los diferentes pasos de la visualización en solo tres y realizarlos por cada nivel de incertidumbre independientemente, provoca que el algoritmo de *ambient occlusion* no disponga de información sobre los otros niveles de incertidumbre. Cuando estamos calculado el factor de oclusión para un pixel de un cierto nivel de incertidumbre, no estamos teniendo en cuenta fibras de otros niveles que quizás ocluyen a esta. Este error en el cálculo del factor de oclusión se ve incrementado por el número de niveles de incertidumbre que estamos visualizando, ya



(a) 1 Nivel (b) 4 Niveles

Figura 5.25: Factor de oclusión con diferentes niveles de incertidumbre. En la imagen (a) el factor de oclusión se calcula correctamente, mientras que en la imagen (b) no, ya que se están visualizando 4 niveles de incertidumbre (aunque todos ellos con el mismo color)

que la geometría se va segmentando en una cantidad mayor de grupos. Podemos ver ilustrado el problema en las figuras 5.25.

Esto tiene fácil solución, basta con realizar el cálculo del factor de oclusión de forma independiente, teniendo en cuenta toda la geometría del modelo.

Capítulo 6

Análisis Económico

6.1. Planificación

Al ser un proyecto de investigación la mayoría de horas se invierten en realizar pruebas hasta obtener un resultado. Pese a esto, el tiempo que tenemos para finalizar el proyecto es limitado, por lo que, igualmente, necesitamos realizar una planificación.

En un proyecto de desarrollo de software estándar la planificación siempre sufre un desvío debido a imprevistos. En un proyecto de investigación el desvío suele ser mayor, ya que no podemos predecir cuánto tiempo nos será necesario para obtener un resultado satisfactorio.

El proyecto tiene una duración aproximada de 7 meses, febrero de 2011 a agosto de 2011. Durante este tiempo he estado trabajando a jornada completa y realizando las diferentes asignaturas del Master en Computación, por lo que no he podido dedicarle todo el tiempo que hubiera deseado.

Tarea	**Semanas**	**Horas**
(P)Investigación DTITool	4	80
(A)Especificación	2	40
(D)Diseño del plugin	2	40
(P)Investigación nueva técnica	14	280
(P)Implementación técnica de visualización	2	40
(A)Documentación	4	80
Total	28	560

Cuadro 6.1: Tabla de planificación inicial.

En la tabla 6.1 se muestra una primera planificación del proyecto. En ella, se describen las tareas a realizar, con el indicador del rol de la persona que la

ha de realizar (A - Analista, D - Diseñador y P - Programador). Las tareas descritas en esta son las siguientes:

- **Investigación DTITool:** El objetivo de esta tarea es aprender a crear plugins para la aplicación DTITool.
- **Especificación:** En esta tarea se realiza la especificación de nuestra aplicación, que es lo que queremos desarrollar.
- **Diseño del plugin:** Una vez tenemos la especificación de la aplicación, realizamos un diseño del sistema, como lo desarrollaremos.
- **Investigación nueva técnica:** Esta es la tarea que requiere más tiempo de todas, ya que en ella tenemos que desarrollar la nueva técnica de pintado.
- **Implementación técnica de visualización:** Cuando ya tenemos desarrollada la nueva técnica, la implementamos dentro de nuestro plugin.
- **Documentación:** Por último, tenemos que realizar la documentación del proyecto.

Pese a esta planificación, la tarea de investigación de la nueva técnica ha llevado algunas semanas más de las previstas, por lo que las demás tareas se han visto reducidas en tiempo.

6.2. Coste Económico

El coste económico de llevar a cabo este proyecto se ha calculado en función de la primera planificación. Para este cálculo se han utilizado los siguientes valores en función del rol:

- **Analista:** 35€/hora
- **Diseñador:** 27€/hora
- **Programador:**20€/hora

El coste final del proyecto se muestra en la tabla 6.2.

Gasto	Coste
Analista	4200 €
Diseñador	1080 €
Programador	8000 €
Visual Studio 2008	520 €
Windows 7	310 €
Pc Gama Alta	1500 €
Total	15610 €

Cuadro 6.2: Tabla de costes.

Capítulo 7

Conclusión

Se ha conseguido crear una nueva técnica de visualización para representar modelos de fibras del cerebro. Esta nueva técnica se centra en mostrar la información relevante para el usuario, que en nuestro caso es el nivel de incertidumbre y la dirección de las fibras. De esta manera un usuario puede visualizar un modelo de fibras, diferenciar a qué nivel de incertidumbre pertenece cada zona y cuál es la dirección principal en la que se desplazan las fibras. Esto se consigue gracias a aplicar técnicas de visualización ilustrativa.

A parte de permitir diferenciar al usuario los niveles de incertidumbre y la dirección de las fibras, se ha conseguido un resultado atractivo a la vista con apariencia de pintado a lápiz.

Nuestro algoritmo permite una visualización interactiva en tiempo real, aunque el número de frames por segundo depende del tamaño de la ventana de visualización.

Bibliografía

[AGL00] Steven Haker Anha Girshick, Victoria Interrante and Todd Lemoine. Line direction matters: An argument for the use of principal directions in 3d line drawings. In *SIGGRAPH 00*, 2000.

[BPD94] Mattiello J. Basser P. and Lebihan D. Estimation of the effective self-diffusion tensor from the nmr spin echo. In *Journal of Magnetic Resonance*, 1994.

[BRA11] Romeny B. M. Haar Brecheisen R., Platel B. and Vilanova A. Illustrative uncertainty visualization for dti fiber pathways. In *EuroVis 2011 Poster*, 2011.

[BS00] J. W. Buchanan and M. C. Sousa. The edge buffer: A data structure for easy silhouette rendering. In *NPAR 2000: First International Symposium on Non-Photorealistica Animation and Rendering*, 2000.

[D.08] Jones D. Tractography gone wild - probabilistic fibre tracking using the wild bootstrap with diffusion tensor. In *IEEE Transactions on Medical Imaging*, 2008.

[ea06] Klein J. et al. Eficient visualization of fiber tracking uncertainty based on complex gaussian noise. In *In Proceedings of International Society for Magnetic Resonance Medicine (ISMRM 06)*, 2006.

[GG01] Bruce Gooch and Amy Gooch. *Non-Photorealistic Rendering*. A K Peters, 2001.

[Hae90] Paul Haeberli. Paint by numbers: Abstract image representation. In *SIGGRAPH 90*, 1990.

[HV09] P. Hermosilla and P. P. Vázquez. Single pass gpu stylized edges. In *IV Iberoamerican Symposium in Computer Graphics*, 2009.

[Mei96] Barbara J. Meier. Painterly rendering for animation. In *SIGGRAPH 96*, 1996.

[MHEI09] Jos B. T. M. Roerdink Maarten H. Everts, Henk Bekker and Tobias Isenberg. Illustrative rendering of dense line data. In *IEEE Visualization 2009*, 2009.

[PDB06] Kavita Bal Philip Dutré and Philippe Bekaert. *Advanced Global Illumination*. A K Peters, 2006.

[ROvdW10] Anna Vilanova Ron Otten and Huub van de Wetering. Illustrative white matter fiber bundles. In *Eurographics, Symposium on Visualization 2010*, 2010.

[SB99] M. C. Sousa and J. W. Buchanan. Computer-generated graphite pencil rendering of 3d polygonal models. In *Computer Graphics Forum*, 1999.

[SP03] Pajevic S. and Basser P. Parametric and non-parametric statistical analysis of dt-mri data. In *Journal of Magnetic Resonance*, 2003.

[ST90] T. Saito and T. Takahashi. Comprehensible rendering of 3-d shapes. In *SIGGRAPH '90*, 1990.

[TAMH08] Eric Haines Tomas Akenine-Moller and Naty Hoffman. *Real-Time Rendering, Third Edition*. A K Peters, 2008.

[VAD05] Kindlmann G. Vilanova A., Zhang S. and Laidlaw D. An introduction to visualization of diffusion tensor imaging and its applications. *Visualization and Image Processing of Tensor Fields*, 2005.

[Wat00] Allan Watt. *3D Computer Graphics*. Addison Wesley, 2000.

[WS94] G. Winkenbach and D. H. Salesin. Computer-generated pen-and-ink illustration. In *SIGGRAPH 94*, 1994.

[ZSK98] Iones A. Zhukov S. and G. Kronin. An ambient light illumination model. In *Eurographics Rendering Workshop 98*, 1998.

Apéndice A

Manual de Usuario

Este capítulo es un pequeño manual sobre el uso de la aplicación. En la primera sección explicaremos el funcionamiento de la aplicación DTITool, y en la segunda sección explicaremos el funcionamiento del plugin desarrollado.

A.1. DTITool

En la imagen A.1 podemos ver una captura de la aplicación.

1. **Ventana de visualización:** En esta ventana se muestra la visualización del modelo utilizando el plugin seleccionado. Realizando diferentes acciones con el ratón sobre esta ventana podemos interactuar con la escena:
   - **Rotación:** Haciendo un click con el botón izquierdo del ratón podemos rotar el modelo a nuestro antojo.
   - **Zoom:** Desplazando la rueda del ratón podemos acercarnos al modelo o alejarnos.
   - **Pan:** Haciendo click con el botón izquierdo mientras mantenemos presionada la tecla Shift realizamos un pan del modelo.
   - **Rotación restringida:** Haciendo click con el botón izquierdo mientras mantenemos presionada la tecla Control podemos realizar una rotación sobre el eje de visión.
2. **Menú:** A través del menú podemos realizar diferentes acciones sobre la aplicación.
   - **File:** Este sub-menú nos proporciona opciones para cargar modelos (nuestro plugin no utiliza esta opción, la carga de modelos se

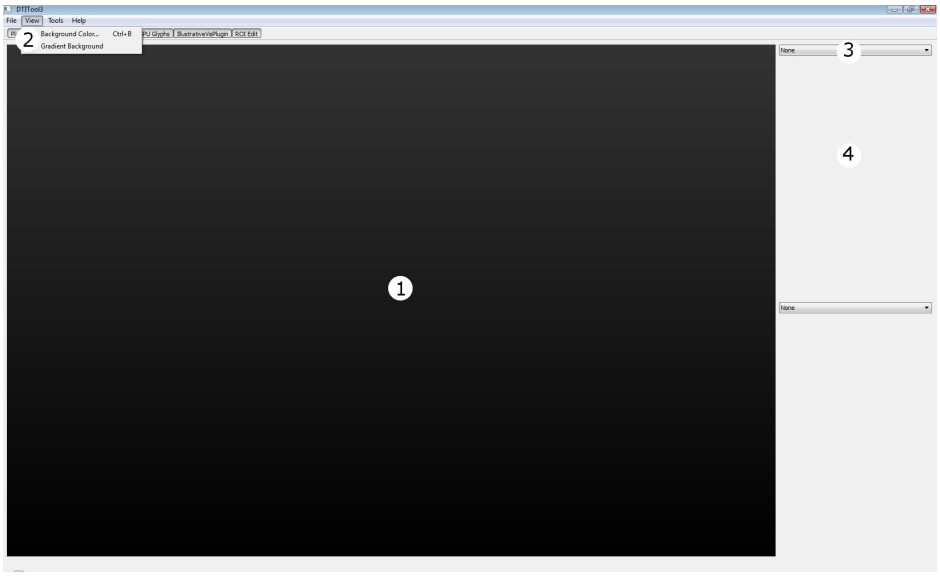


Figura A.1: DTITool.

realiza a través del plugin), ver la lista de modelos cargados, guardar la configuración, definir la carpeta raíz y salir de la aplicación.

- **View:** A través de este sub-menú podemos cambiar el color de fondo de la visualización.
- **Tools:** Este sub-menú nos da la opción de realizar capturas de pantalla y cargar y descargar plugins.
- **Help:** En este último sub-menú están las opciones para consultar la ayuda y obtener información sobre la aplicación.

3. **Selección de plugin:** A través de esta lista seleccionamos el plugin con el que queremos visualizar. Si el plugin que deseamos no está en esta lista, tenemos que cargar el plugin a través del menú.

4. **Ventana de configuración del plugin:** Una vez seleccionamos el plugin con el que deseamos visualizar, en esta pantalla aparecen las diferentes opciones para configurar esta visualización.

A.2. IllustrativeVis Plugin

En esta sección se explica cómo funciona el widget de configuración del plugin que hemos desarrollado.

1. **Load Header:** Botón para cargar el modelo de fibras en formato vtk.

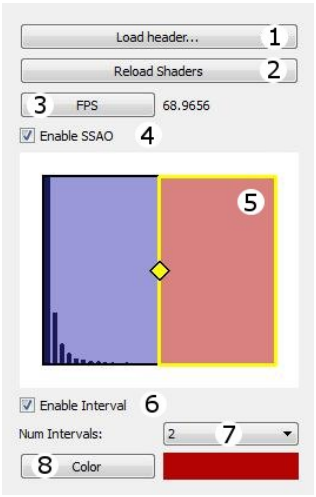


Figura A.2: IllustrativeVis plugin.

2. **Reload Shaders:** Botón para recargar los shaders.

3. **FPS:** Botón para calcular el número de frames por segundo a los que se ejecuta el algoritmo de visualización.

4. **Enable SSAO:** Checkbox para activar y desactivar el algoritmo de *ambient occlusion*.

5. **Confidence Histogram:** A través de este widget podemos modificar los rangos de incertidumbre desplazando el diamante amarillo que separa los diferentes niveles.

6. **Enable Interval:** Con este checkbox podemos activar y desactivar el intervalo de incertidumbre seleccionado.

7. **Num Intervals:** A través de esta lista seleccionamos el número de intervalos de incertidumbre en los que se divide el modelo.

8. **Color:** Botón para cambiar el color del intervalo de incertidumbre seleccionado.