Proyecto Fin de Carrera

OPTIMIZACIÓN DEL RENDERIZADO DE VOLUMEN MEDIANTE LA TÉCNICA *SHEAR-WARPING*

VOLUME RENDERING OPTIMIZATION BY MEANS OF A SHEAR-WARP FACTORIZATION

Autor/es

José Ángel Tavira Rodríguez de Liébana

Director/es

Carlos Monserrat Aranda
UNIVERSIDAD POLITÉCNICA DE VALENCIA

Ponente/es

Sandra Baldassarri
ESCUELA DE INGENIERÍA Y ARQUITECTURA
UNIVERSIDAD DE ZARAGOZA

ESCUELA DE INGENIERÍA Y ARQUITECTURA
2017

DATOS TÉCNICOS

Título

Optimización del renderizado de volumen mediante la técnica *shear-warping*.

Autor

José Ángel Tavira Rodríguez de Liébana

DNI

29119232-J

Titulación

Ingeniería Informática

Ponente

Sandra Baldassarri

Director

Carlos Monserrat Aranda

Departamento

Sistemas Informáticos y Computación

Centro

Universidad Politécnica de Valencia

Universidad

Universidad Politécnica de Valencia

Fecha

Junio de 2017

OPTIMIZACIÓN DEL RENDERIZADO DE VOLUMEN MEDIANTE LA TÉCNICA *SHEAR-WARPING*

RESUMEN

El renderizado de volumen es una técnica para representar objetos tridimensionales en una imagen, a partir de las muestras obtenidas de dichos objetos.

Debido a la gran cantidad de muestras necesarias para obtener una imagen nítida, el coste temporal de los diferentes algoritmos desarrollados en la actualidad no permite la representación de los objetos de forma eficiente, obligando a dichos algoritmos a recurrir a técnicas que requieren el uso de hardware específico y/o la disminución de la calidad de la imagen final.

Este proyecto tiene como objetivo la implementación de un algoritmo que permita visualizar objetos tridimensionales en proyección paralela, de forma eficiente, esto es, con un coste temporal inferior a un segundo, a partir de un volumen de muestras de tamaño habitual (256^3 *voxels*), en ordenadores de propósito general, sin comprometer la calidad de la imagen ni utilizar hardware específico.

Para ello, se estudian las estrategias de los diferentes algoritmos presentes en la literatura, extrayendo sus ventajas e inconvenientes, así como diferentes técnicas de optimización del rendimiento de dichos algoritmos.

A continuación, se diseñan las características de un algoritmo, además de las estructuras de datos apropiadas, para aplicar la técnica de descomposición matricial *shear-warp*, la cual permite combinar las ventajas de los algoritmos analizados junto con las técnicas de optimización de rendimiento.

Por último, se implementa el algoritmo y se analizan los resultados obtenidos en cuanto a rendimiento y calidad de imagen, emitiendo las conclusiones pertinentes y proponiendo diversas alternativas a desarrollar en un futuro.

VOLUME RENDERING OPTIMIZATION BY MEANS OF A SHEAR-WARP FACTORIZATION

ABSTRACT

Volume rendering is a technique for displaying the image of three-dimensional sampled objects.

Due to the high number of samples needed to obtain a clear image, the computational cost of the current algorithms does not let visualize objects efficiently, which makes those algorithms use some techniques requiring specialized hardware and/or seriously reducing image quality.

The goal of this project is to develop an algorithm able to visualize three-dimensional objects in parallel projection, efficiently, that is to say, with a computational cost of less than one second, from an average-sized sampled volume (256^3 *voxels*), on general purpose workstations, without compromising image quality and not using specialized hardware neither.

In order to achieve this goal, the methods applied in algorithmic literature are analysed to draw their advantages and disadvantages, as well as several acceleration techniques to improve the performance of those algorithms.

Next, the features of an algorithm, and the most suitable data structures, are designed to apply the 'shear-warp' matrix factorization, which lets us combine the advantages of the analysed algorithms and the performance improvement techniques.

Finally, the algorithm is implemented, and the achieved results about performance and image quality are analysed, explaining the corresponding conclusions and suggesting several future improvements.

AGRADECIMIENTOS

Me gustaría mostrar mi gratitud a todas aquellas personas que han hecho posible la realización de este proyecto.

Quiero dar las gracias a mi tutor Carlos Monserrat por su dedicación y formación aportadas a mi persona, y a Sandra Baldassarri, por su extraordinaria labor como ponente y maestra durante mi formación como ingeniero informático.

Por último, y especialmente, quiero dar las gracias a mi padre Jerónimo y a mi madre Francisca, por la gran confianza depositada en mí y su apoyo incondicional mostrados durante toda mi vida, en concreto, durante la elaboración de este proyecto, el cual les dedico.

ÍNDICE

1 INTRODUCCIÓN

En este capítulo se describen los motivos que impulsan la realización de este proyecto, dentro de la situación contextual en el que se desarrolla. A continuación, se explican los objetivos generales, las herramientas y material utilizados, así como la aplicación final del producto obtenido. Finalmente se describe la estructura del contenido de esta memoria.

1.1. CONTEXTO Y MOTIVACIÓN

Desde su origen, el hombre siempre ha estado interesado en la cuantificación de los parámetros por los que se rige la naturaleza que le rodea. Con el paso del tiempo ha ido desarrollando sistemas de medición y simulación cada vez más precisos, los cuales producen como resultado enormes conjuntos de datos que son difíciles de manejar e interpretar.

En medicina, tecnologías de imagen médica tales como la resonancia magnética o la tomografía asistida por ordenador, generan grandes series tridimensionales de datos que contienen una representación muy detallada de los órganos internos del cuerpo humano, permitiendo a los cirujanos establecer diagnósticos sin utilizar la cirugía invasiva. En geología, la exploración sísmica se vale de las ondas acústicas producidas por una explosión para generar mapas 3D de estructuras geológicas debajo de la superficie terrestre. En física, la simulación por ordenador de la dinámica de fluidos genera información en forma de una red tridimensional de datos que se estudia para predecir el comportamiento aerodinámico de un vehículo, por ejemplo, dentro de un túnel de aire.

Es necesario, por lo tanto, extraer la información contenida en estas inmensas series de números de una forma útil e inteligible para el hombre, de manera que se pueda interpretar y manipular con facilidad. Una manera de conseguirlo es mediante la visualización tridimensional de dichos datos. Entre las técnicas más utilizadas se encuentra el **renderizado de volumen**, denominada también con el término inglés ***volume rendering***.

El renderizado de volumen es un método de generación de imágenes a partir de una serie 3D de datos escalares. La **Ilustración 1** muestra una de estas imágenes producida a partir de una tomografía asistida por ordenador (TAC) de una cabeza humana. El *volume rendering* es un método robusto y versátil para visualizar series 3D, pero tiene el inconveniente de su alto coste computacional, lo cual ha desviado su uso exclusivo para las grandes compañías.

Este alto coste se debe al gran tamaño de los conjuntos de datos utilizados para generar las imágenes. Renderizar tal cantidad de información de forma suficientemente rápida para las aplicaciones interactivas, requiere una elevada potencia computacional y un gran ancho de banda en memoria.

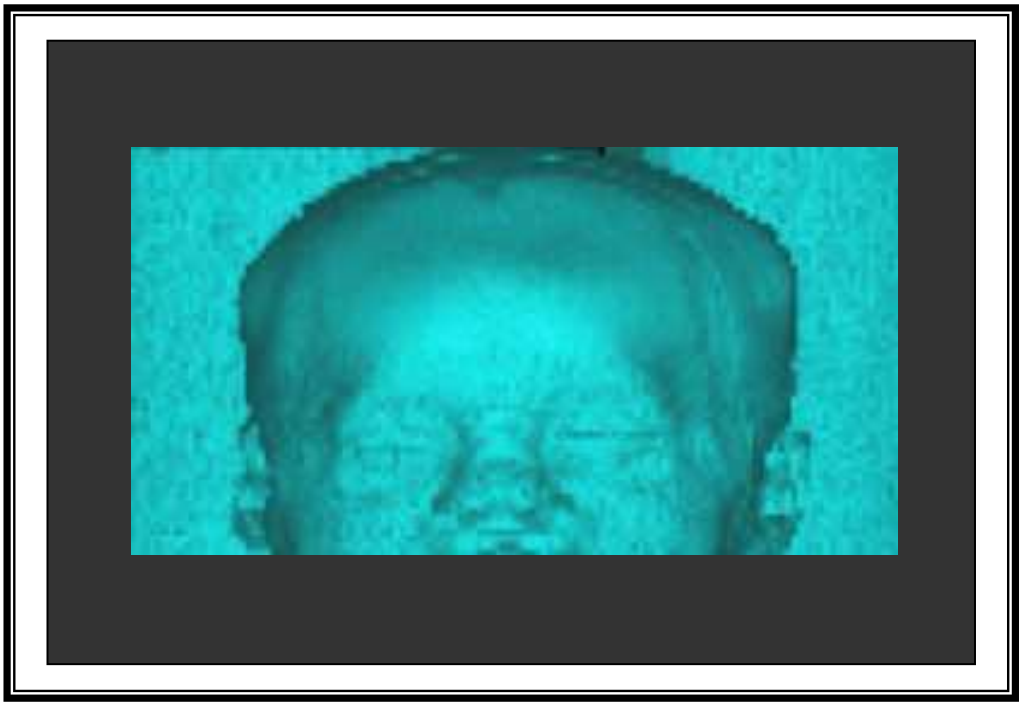


Ilustración 1.- Renderizado de volumen obtenido a partir de una tomografía asistida por ordenador de una cabeza humana.

Para aumentar la velocidad de renderizado se han utilizado diferentes aproximaciones, pero todas ellas se han basado, bien en reducir la calidad de la imagen final, mediante el truncamiento de la información original, o bien en utilizar hardware especializado de elevado coste, como el uso de tarjetas gráficas avanzadas; dando lugar todo ello a aplicaciones software de poca utilidad, con bajo aporte de información al usuario final e incluso fuera del alcance económico de éste. Este proyecto propone un algoritmo de renderizado eficiente, sin comprometer la calidad de la imagen y sin requerimientos específicos de hardware.

1.2. OBJETIVO DEL PROYECTO

La finalidad de este proyecto es, basándose en el estudio de las ventajas e inconvenientes de las soluciones propuestas en el estado del arte referente al renderizado de volumen, implementar un algoritmo de *volume rendering* que cumpla las siguientes premisas:

- Generación de imágenes, en proyección paralela, de alta calidad, con un coste temporal inferior a un segundo.
- Información original consistente en volúmenes de datos estándar (256^3 *voxels*).
- Destinado al uso en pequeñas estaciones de trabajo de propósito general.
- Ejecución sin requerir hardware especializado.

1.3. ALCANCE

El algoritmo generado en este proyecto pretende ser incorporado a aplicaciones informáticas comerciales de diagnóstico médico. La información visual proporcionada por este tipo de aplicaciones es de un valor crucial para la correcta detección de patologías, por lo que requieren una alta calidad en las imágenes generadas.

Por otra parte, estos programas informáticos requieren una interacción persona-ordenador constante, para elaborar un correcto diagnóstico del paciente, lo que implica que el coste temporal de renderizado debe ser mínimo para permitir una rápida manipulación del conjunto de datos volumétricos del paciente, cuyo tamaño es similar al establecido en las premisas de este proyecto.

Por último, es importante señalar que estas aplicaciones suelen son utilizadas por facultativos en diferentes lugares de trabajo (despachos, quirófanos, salas de conferencia, etc.), donde no siempre tienen acceso físico o económico a potentes ordenadores, sino a máquinas de propósito general, plataforma en la que se debe poder ejecutar este algoritmo.

1.4. HERRAMIENTAS UTILIZADAS

Para la realización de este proyecto se dispone de los siguientes medios tecnológicos y material de apoyo.

ENTORNO DE DESARROLLO

- **Aplicación de desarrollo:** Como herramienta para la codificación del algoritmo y pruebas de ejecución se utiliza el programa de desarrollo de aplicaciones informáticas *Visual Studio* (módulo C++).
- **Plataforma de ejecución:** Sistema operativo *Microsoft Windows* (versión actual).

MATERIAL

- **Volúmenes de datos:** Estudios médicos procedentes de pacientes reales y modelos anatómicos.

1.5. CONTENIDO DE LA MEMORIA

Para poder implementar un algoritmo de renderizado de volumen es necesario conocer todos los detalles, desde su fundamento físico hasta las estructuras de datos requeridas para su implementación, así como las posibles estrategias de incremento de eficiencia y gestión adecuada en el uso de la memoria.

Todos estos aspectos se abordan en los siguientes capítulos de la presente memoria:

- En el segundo capítulo se detalla la técnica de *volume rendering* a nivel físico, esto es, desde la estructura volumétrica del objeto a representar, hasta el modelo físico en el que se basa para calcular el color de cada *pixel* representado en la imagen final.

- El tercer capítulo despliega un estudio sobre las familias de algoritmos existentes en la actualidad, mostrando sus ventajas y desventajas, así como las técnicas de aceleración que mejorarían su rendimiento.

- El capítulo cuarto explica detalladamente la técnica *shear-warping* para proyecciones paralelas, demostrando su idoneidad para combinar las ventajas y mejoras de los algoritmos estudiados en la sección anterior.

- El quinto capítulo describe de forma completa y detallada el algoritmo implicado en la técnica *shear-warping*, mostrando todas sus fases de desarrollo: especificación de requisitos, diseño e implementación.

- El capítulo seis de este proyecto está destinado a las pruebas realizadas para evaluar el rendimiento y calidad de imagen conseguidos por el algoritmo.

- En el séptimo capítulo se exponen las conclusiones en cuanto a rendimiento y calidad en la imagen obtenida, así como las posibles mejoras para adaptar el algoritmo a diferentes aplicaciones.

La información contenida en esta memoria se complementa con los siguientes anexos:

- ANEXO I: GALERÍA DE IMÁGENES: Muestra de algunas imágenes de ejemplo obtenidas por el algoritmo de renderizado a partir de diferentes volúmenes de datos.

- ANEXO II: CÓDIGO DEL ALGORITMO: Contiene las principales rutinas del algoritmo de renderizado codificadas en lenguaje C++.

2 CONCEPTOS DE *VOLUME RENDERING*

Este capítulo explica los conceptos teóricos que permiten entender la técnica de renderizado de volumen. Para ello, se detallan las características de la información de partida, los fenómenos físicos subyacentes, terminando con la ecuación matemática que modela el comportamiento de la luz y la visualización de un objeto en una imagen.

2.1. VOLUMEN DE DATOS

Un algoritmo de *volume rendering* genera una imagen tomando como información de entrada una serie tridimensional de datos escalares. Esta serie de datos se llama **volumen** y cada elemento de la serie se conoce como ***voxel***. El volumen puede representarse como una función escalar continua y cada *voxel* representa un punto **muestreado** de dicha función.

La información (*voxels*) contenida en el volumen tras el muestreo depende de las directrices llevadas a cabo para extraer las muestras del objeto a renderizar. Estas directrices determinan la **rejilla de muestreo**, la cual puede ser de cuatro tipos:

- **Regular**: en ella los *voxels* obtenidos corresponden a muestras del volumen con una conectividad uniforme.
- **Curvilínea**: obtenida a partir de una rejilla regular que ha sido deformada utilizando una transformación no lineal, de manera que los *voxels* obtenidos pertenecen a muestras del volumen localizadas siguiendo dicha transformación.
- **No estructurada**: formada por una colección arbitraria de *voxels* donde la conectividad entre muestras no sigue una regla explícita.
- **Híbrida**: Formada por una composición de los tres tipos de rejilla anteriormente mencionados.

La **Ilustración 2** muestra un ejemplo de cada tipo de rejilla. Este proyecto centra su atención en la **rejilla regular**, porque es el mecanismo de muestreo más simple y rápido y, por lo tanto, permite la implementación de un algoritmo de renderizado más eficiente, objetivo final de este proyecto.

La secuencia de pasos básicos de todo algoritmo de renderizado de volumen consiste en primero **asignar** un color y una opacidad (o nivel de transparencia) a cada *voxel* del volumen, después **proyectar** estos *voxels* en una imagen plana y, por último, **combinar** en dicha imagen los *voxels* coincidentes tras la proyección.

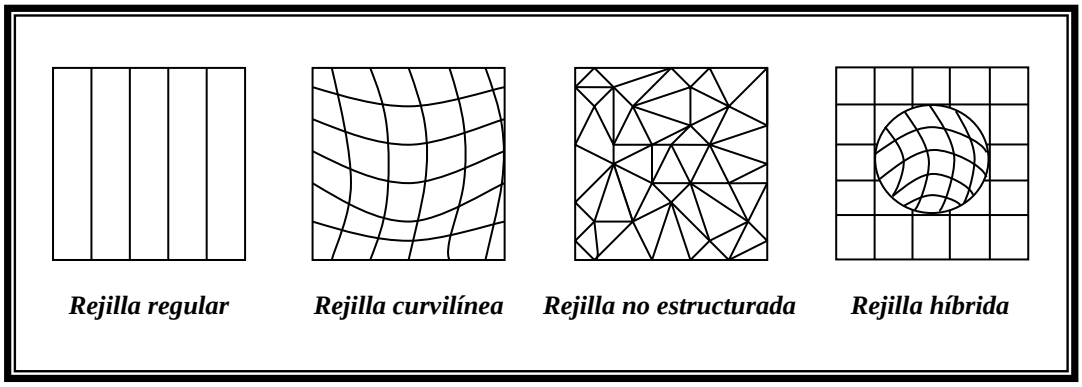


Ilustración 2.- Ejemplos de tipos de rejilla.

2.2. REMUESTREO

Los puntos de muestreo que un algoritmo de renderizado de volumen necesita para realizar su función, no se correspondan exactamente con los datos del volumen (representados, como se ha mencionado en el apartado anterior, mediante una serie discreta de muestras formando una rejilla). Por ello, cuando el algoritmo requiere un punto de muestreo que no corresponde a ninguno de los *voxels* que forman el volumen, recurre a los *voxels* más cercanos en la rejilla para obtener el valor de color y opacidad del punto de muestreo deseado.

Esta forma de obtener puntos arbitrarios del volumen a partir de las muestras discretas que lo representan es una técnica llamada **reconstrucción del volumen**. En la práctica, existen dos factores que condicionan la perfecta reconstrucción de un volumen:

- **Características de la función continua**: la frecuencia de muestreo utilizada para obtener la serie de muestras del volumen debe ser, al menos, el doble que la máxima frecuencia espacial (frecuencia de Nyquist) de la función continua que representa el objeto a renderizar (Teorema de Nyquist; [Bracewell 1986]). Si este teorema se cumple, entonces la reconstrucción perfecta del volumen es posible. Por lo tanto, el volumen de muestras debe corresponder a una función continua limitada en banda, para poder ser reconstruido a partir de las muestras obtenidas. Es decir, la función continua no puede contener frecuencias arbitrariamente altas o, de lo contrario, es necesario aplicarle un filtro de paso bajo para eliminarlas, obteniendo una versión "filtrada" de la función original; siendo esta versión la única que se puede reconstruir. Toda función que no cumpla esta característica produce el efecto de *aliasing* en la imagen final del volumen renderizado.

- **El filtro de reconstrucción**: La reconstrucción perfecta de una función se obtiene aplicando un filtro senoidal [Bracewell 1986]. Éste utiliza la contribución ponderada de todas las muestras del volumen para obtener cada uno de los puntos de muestreo requeridos. Computacionalmente, esta técnica es demasiado costosa, de modo que se utilizan otros métodos más rápidos (aunque menos precisos) para la reconstrucción. Así, en vez de utilizar filtros que requieran el examen de todas las muestras del volumen, se usan otros que sólo

necesitan un número de muestras espacialmente localizadas en una vecindad próxima al punto de muestreo que se quiere obtener. Los filtros comúnmente más utilizados son los que aplican interpolación trilineal e interpolación por vecino más próximo, éste último es el utilizado por el algoritmo de este proyecto. Dichos métodos obtienen resultados muy aceptables siempre y cuando actúen sobre volúmenes que cumplan el teorema anteriormente mencionado.

Ambos factores influyen en la calidad del renderizado final, dado que cuanto mejor sea la reconstrucción del volumen original, más fidedignas son las muestras obtenidas. El proceso de reconstrucción más muestreo se denomina **remuestreo**.

2.3. ECUACIÓN DE *VOLUME RENDERING*

El renderizado de volumen es una simulación aproximada de la propagación de la luz a través de un medio participativo representado por el volumen. Dicho medio puede entenderse como un bloque de gel coloreado y semi-transparente en el cual el color y la opacidad son funciones dependientes de los valores contenidos en la serie tridimensional que representa dicho volumen. A medida que la luz fluye a través de éste, interacciona con él por medio de diferentes procesos: absorción, dispersión, reflexión, fosforescencia (absorción de la luz y reemisión tras un periodo de tiempo) y fluorescencia (absorción de la luz y reemisión a una frecuencia diferente). Pero, dado que la meta del *volume rendering* es la visualización de datos (no la simulación precisa de un fenómeno físico), en este proyecto se omiten muchas partes del modelo óptico completo que no son necesarias para la visualización. Con un modelo óptico simplificado, **un algoritmo de *volume rendering* trata de obtener una imagen calculando cuánta luz llega a cada punto del plano de la imagen**.

En la **Ilustración 3** se muestra gráficamente un modelo simplificado de la propagación de la luz.

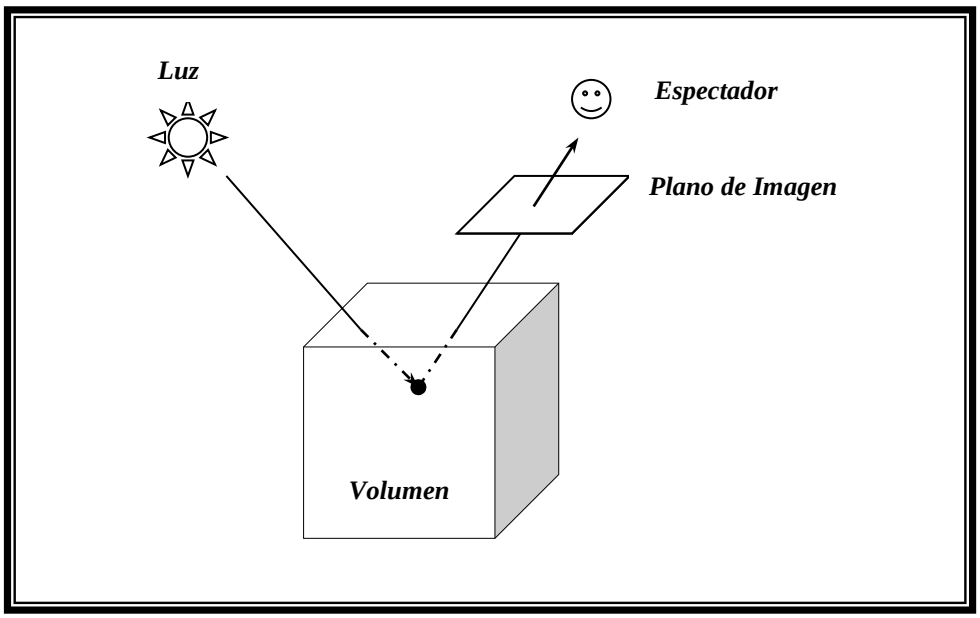


Ilustración 3.- Modelo físico simplificado de la propagación de la luz a través de un volumen colisionando con el plano de imagen del espectador.

El transporte de la luz se rige por un caso especial de la ecuación de Boltzmann extraído de su "Teoría del Transporte", estudio basado en cómo una distribución estadística de partículas (en este caso fotones) fluye a través de un entorno[1].

Así, una forma de calcular la cantidad de luz que llega hasta el plano de imagen es mediante el estudio del flujo de fotones que atraviesa el volumen. Pero, en vez de contar fotones, se puede escribir una **Ecuación de equilibrio de la energía** en términos de **radiancia**. Ésta describe la densidad de potencia transmitida por los fotones siguiendo una dirección determinada en un punto concreto; su expresión, dada una serie de suposiciones [Lacroute 1995], es la siguiente[2]:

$$L(x) = \int_{x}^{x_B} e^{-\int_{x}^{x'} \phi_t(x'')dx''} \varepsilon(x')dx'$$

En esta ecuación se ha reparametrizado la radiancia en función de una variable unidimensional, *x*, que representa la distancia a lo largo del rayo de luz (también llamado rayo de vista). El límite superior de integración es x_B, punto en el cual el rayo sale del volumen (ver **Ilustración 4**). Φ_t es el coeficiente de absorción o dispersión de un fotón en el volumen y ε es el coeficiente de emisión de fotones hacia el volumen.

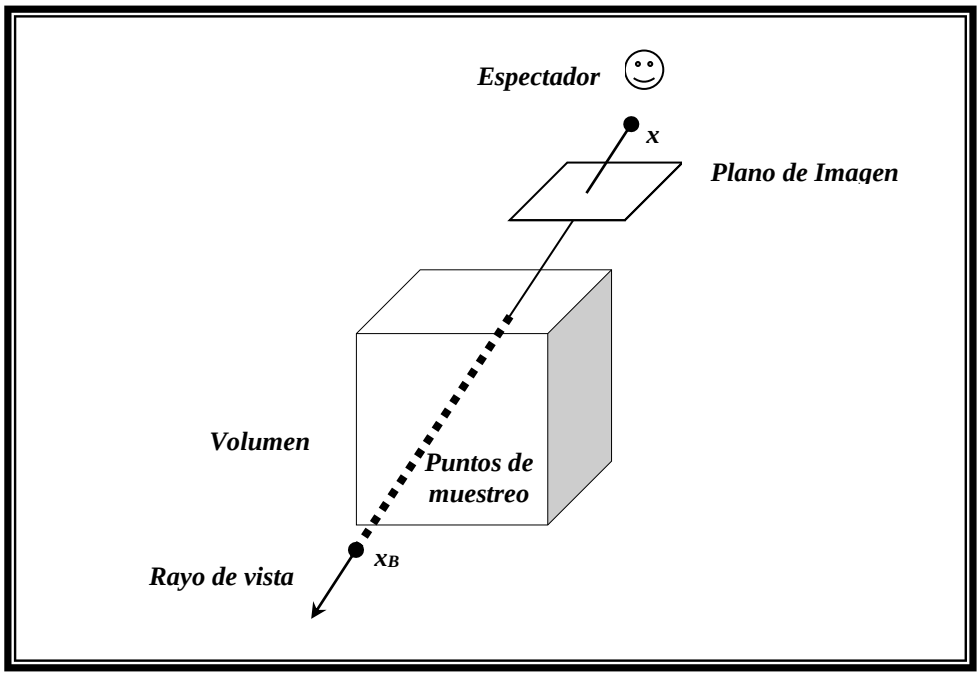


Ilustración 4.- Modelo simplificado para el renderizado de volumen: el valor de un *pixel* es calculado remuestreando los valores de cada *voxel* a lo largo del rayo de vista, desde un punto *x* de la imagen hasta un punto x_B, límite opuesto del volumen, y evaluando numéricamente la ecuación de *volume rendering*.

[1] Escritos como [Arvo 1993], [Glassner 1995] y [Cohen & Wallace 1993] contienen introducciones a la Teoría del Transporte y su aplicación a la informática gráfica. En otros como [Siegel & Howell 1992], se realiza un estudio más detallado de las leyes físicas subyacentes.

[2] La ecuación de *volume rendering* obtenida en este apartado es equivalente a la propuesta en los modelos de renderizado de volumen expuestos en [Blinn 1982], [Levoy 1989] y [Sabella 1988].

Un sencillo algoritmo de trazado de rayos, basado en la ecuación de *volume rendering*, opera lanzando rayos que atraviesan el volumen, paralelos a la dirección de vista, como muestra la **Ilustración 4.- Modelo simplificado para el renderizado de volumen: el valor de un *pixel* es calculado remuestreando los valores de cada *voxel* a lo largo del rayo de vista, desde un punto *x* de la imagen hasta un punto x_B, límite opuesto del volumen, y evaluando numéricamente la ecuación de *volume rendering*.**

. El algoritmo, a continuación, evalúa numéricamente la integral correspondiente a dicha ecuación para cada rayo. En la siguiente sección se describe un método común para evaluar dicha integral, método llamado **composición volumétrica**.

2.3.1. Ecuación de composición volumétrica

A continuación, se presenta un método aproximado para calcular la ecuación de *volume rendering*. Dicha ecuación, para un espacio de muestreo Δx a lo largo de los rayos de vista puede evaluarse usando la regla del rectángulo[3]:

$$L(x) = \sum_{i=0}^{n-1} e^{-\sum_{j=0}^{i-1}\phi_j \Delta x} \cdot \varepsilon_i \Delta x = \sum_{i=0}^{n-1} \varepsilon_i \Delta x \prod_{j=0}^{i-1} e^{-\phi_j \Delta x}$$

donde:

$$\varepsilon_i \equiv \varepsilon(x + i\Delta x)$$
$$\phi_i \equiv \phi_t(x + i\Delta x)$$

La expresión de la derecha de la ecuación puede ser escrita utilizando el operador *over* extraído de la composición digital en [Porter & Duff 1984], tal y como se puede observar realizando las siguientes substituciones:

$$\alpha_i \equiv 1 - e^{-\phi_i \Delta x}$$
$$C_i \equiv (\varepsilon_i / \alpha_i)\Delta x$$
$$c_i \equiv C_i \alpha_i$$

donde α_i, C_i *y* c_i son respectivamente la opacidad, el color y el producto de ambos, correspondientes a la muestra *i*. Obteniendo como resultado la **Ecuación de Composición Volumétrica**:

$$L(x) = \sum_{i=0}^{n-1} c_i \prod_{j=0}^{i-1} 1 - \alpha_j =$$
$$= c_0 + c_1(1-\alpha_0) + c_2(1-\alpha_0)(1-\alpha_1) + \ldots + c_{n-1}(1-\alpha_0)\ldots(1-\alpha_{n-2}) =$$
$$= c_0 \otimes c_1 \otimes c_2 \otimes \ldots \otimes c_{n-1}$$

[3] Diferentes reglas han sido propuestas para obtener una aproximación más precisa a la ecuación de *volume rendering*: en [Upson & Keeler 1988] se usa la regla trapezoidal y en [Novins & Arvo 1992] las reglas de cuadratura de alto orden.

donde ⊗ es el operador *over* de composición digital.

Por lo tanto, el algoritmo de trazado de rayos introducido en la sección anterior renderizaría un volumen de la siguiente manera:

1º) Para cada *pixel* de la imagen, lanzar un rayo dirigido hacia el volumen.

2º) En puntos de muestreo regularmente espaciados a lo largo del rayo, calcular el color C_i y la opacidad α_i a partir del valor escalar en el volumen.

3º) Combinar los colores y opacidades a lo largo del rayo mediante la ecuación de composición volumétrica.

Éste es un ejemplo de algoritmo de fuerza bruta en el renderizado de volumen. Si un volumen contiene *n voxels* por cara y la imagen se calcula lanzando n^2 rayos con *n* puntos de muestreo por rayo, entonces **el algoritmo requiere un número de operaciones del orden de n^3**.

Este proyecto propone un método para calcular la misma imagen con un menor coste computacional[4].

[4] Existen además dos alternativas muy comunes a la ecuación de composición volumétrica como son "las proyecciones de rayos X" y "las proyecciones de máxima intensidad", tratadas en [Laub & Kaiser 1988], [Keller 1989] y [Laub 1990], y que también requieren un coste computacional menor que el citado algoritmo de fuerza bruta.

3 ESTUDIO DE TRABAJOS PREVIOS

Los algoritmos existentes de *volume rendering* se pueden clasificar en cuatro grupos: algoritmos de **trazado de rayos**, algoritmos de **salpicado**, algoritmos de **proyección de celdas** y algoritmos de **remuestreo con multipaso** [Lacroute 1995]. Las dos características que distinguen a cada grupo son:

- El orden en el que se recorre el volumen de datos.
- El método que aplica para proyectar los *voxels* en la imagen.

Cada grupo de algoritmos tiene sus ventajas e inconvenientes y, además, algunos son más sensibles que otros a algunas técnicas de aceleración del rendimiento.

Como la literatura referente al renderizado de volumen es muy extensa, en esta memoria sólo se tienen en consideración los algoritmos y técnicas de aceleración más representativos para ilustrar las diferencias entre ellos.

3.1. ALGORITMOS DE *VOLUME RENDERING*

A continuación, se describen las cuatro clases de algoritmos de renderizado. Para diferenciarlos se recurre al orden establecido en la estructura de bucles que todo algoritmo de *volume rendering* requiere.

Dicha estructura consta de seis bucles: tres de ellos iteran sobre las tres dimensiones del volumen o del recorrido del rayo, mientras que los otros tres restantes lo hacen sobre los *pixels* de la imagen o los puntos del volumen utilizados por el filtro de remuestreo.

Estos bucles pueden ser intercambiados (de acuerdo con las restricciones impuestas por la ecuación de renderizado de volumen correspondiente) dando lugar a las diferentes clases de algoritmos. El orden de los bucles es importante en nuestra búsqueda de un algoritmo de renderizado eficiente, porque dicho orden influye directamente en las características de rendimiento de cada uno de los cuatro tipos de algoritmos.

3.1.1. Algoritmos de trazado de rayos

Los algoritmos de trazado de rayos generan una imagen mediante la proyección de un rayo a través del volumen para cada uno de los *pixels* de dicha imagen. A continuación, combinan el color y la opacidad de los *voxels* atravesados, a lo largo del rayo, tal y como se ha descrito en el apartado referente a la ecuación de composición volumétrica [Levoy 1988], [Sabella 1988] y [Upson & Keeler 1988].

Este tipo de algoritmos se denomina “algoritmos dirigidos por la imagen” puesto que sus bucles exteriores iteran sobre los *pixels* de la imagen. Su estructura de bucles es la mostrada en la **Ilustración 5.- Estructura de bucles característica de los algoritmos de trazado de rayos.**

.

```
Para cada yi desde 1 hasta altura de la imagen
    Para cada xi desde 1 hasta la anchura de la imagen
        Para cada zi desde 1 hasta la longitud del rayo

            Para cada x0 requerido por el filtro de remuestreo en función de (xi, yi , zi)
                Para cada y0 requerido por el filtro de remuestreo en función de (xi, yi , zi)
                    Para cada z0 requerido por el filtro de remuestreo en función de (xi, yi , zi)
                        Combinar color y opacidad del voxel (x0, y0, z0)
                        sobre el pixel (xi, yi) de la imagen
                    fPara
                fPara
            fPara

        fPara
    fPara
fPara
```

Ilustración 5.- Estructura de bucles característica de los algoritmos de trazado de rayos.

Los dos bucles exteriores iteran sobre los *pixels* de la imagen. El siguiente lo hace sobre los puntos de muestreo a lo largo del rayo (rayo de vista). Finalmente, los tres bucles interiores iteran sobre los *voxels* que el filtro de remuestreo necesita para reconstruir un punto de muestreo.

El cuerpo del bucle anidado multiplica el valor del *voxel* por un peso del filtro de remuestreo y añade el resultado (color y opacidad) al *pixel* de la imagen.

Este grupo de algoritmos también es llamado “algoritmos de proyección hacia atrás” porque proyectan los rayos de vista desde la imagen al volumen, justo en sentido contrario al seguido físicamente por los rayos de luz.

La principal desventaja de estos algoritmos es que no acceden de una forma ordenada al volumen puesto que los rayos de vista pueden atravesarlo en cualquier dirección. Como consecuencia, pierden mucho tiempo obteniendo la localización del punto de muestreo (es decir, los índices del *voxel* utilizado en el cuerpo del bucle anidado) y calculando una gran aritmética de direccionamiento.

Otra desventaja reside en la baja localidad espacial. Así, al acceder desordenadamente al volumen, las memorias caché de los procesadores resultan poco efectivas en su misión de disminuir la latencia de la memoria principal.

3.1.2. Algoritmos de salpicado

En contraste con los algoritmos de trazado de rayos, esta clase de algoritmos operan iterando sobre los *voxels* del volumen [Westover 1990], calculando la contribución a la imagen de cada *voxel* mediante un filtro que distribuye el valor de dicho *voxel* entre una vecindad de *pixels* determinada. Este método se denomina "salpicado", por el cual estos algoritmos reciben ese nombre [Westover 1989]. Debido a su forma de operar partiendo de los *voxels* del volumen, se denominan "algoritmos dirigidos por el volumen". En la **Ilustración 6.- Estructura de bucles comúnmente utilizada en los algoritmos de salpicado.**
se detalla su estructura de bucles.

```
Para cada z0 desde 1 hasta la profundidad del volumen
    Para cada y0 desde 1 hasta la altura del volumen
        Para cada x0 desde 1 hasta la anchura del volumen

            Para cada zi requerido por el filtro de remuestreo en función de (x0, y0, z0)
                Para cada yi requerido por el filtro de remuestreo en función de (x0, y0, z0)
                    Para cada xi requerido por el filtro de remuestreo en función de (x0, y0, z0)
                        Combinar color y opacidad del voxel (x0, y0, z0)
                        sobre el pixel (xi, yi) de la imagen
                    fPara
                fPara
            fPara

        fPara
    fPara
fPara
```

Ilustración 6.- Estructura de bucles comúnmente utilizada en los algoritmos de salpicado.

Como se puede observar, comparando este algoritmo con el anterior de trazado de rayos, se han intercambiado los bucles exteriores y los interiores. Así, los tres bucles exteriores iteran sobre los *voxels*, el siguiente lo hace sobre la extensión en profundidad requerida por el filtro de remuestreo, reservándose los últimos bucles para iterar sobre los *pixels* de la imagen.

Otra denominación para los algoritmos de salpicado es la de "algoritmos de proyección hacia delante" puesto que los *voxels* son proyectados en el mismo sentido en el que lo hacen los rayos de luz.

A diferencia de los algoritmos de trazado de rayos, los algoritmos de proyección hacia delante recorren el volumen en orden de almacenamiento evitando así las desventajas citadas anteriormente.

Sin embargo, los filtros de remuestreo que utiliza esta técnica son computacionalmente muy costosos si se desea obtener una alta calidad, debido a que dependen directamente de la transformación de coordenadas requerida para pasar del volumen a la imagen, es decir, son dependientes del punto de vista del objeto.

En teoría, en lo que se refiere a calidad de imagen, los algoritmos de salpicado generan exactamente las mismas imágenes que los algoritmos de trazado de rayos.

3.1.3. Algoritmos de proyección de celdas

Un tercer tipo de algoritmos consiste en la utilización de técnicas de proyección de celdas [Upson & Keeler 1988], [Max 1990], [Shirley & Tuchmann 1990] y [Wilhelms & Van Gelder 1991]. Este método a menudo se utiliza en volúmenes muestreados mediante rejillas no regulares.

El primer paso es descomponer el volumen en poliedros cuyos vértices son los puntos de muestreo (utilizando por ejemplo la triangulación de Delauney o algoritmos de decimación). A continuación, el algoritmo ordena los poliedros en profundidad, y finalmente evalúa la ecuación integral de *volume rendering* entre las caras de poca profundidad y las de alta profundidad de cada poliedro, con el fin de calcular el color y la opacidad de cada *pixel* de la imagen.

Aunque se trata de algoritmos dirigidos por el volumen y son muy similares a los anteriormente citados algoritmos de salpicado, la implementación del software correspondiente lleva a programas de alto coste computacional y, a menudo, se necesita hardware especializado para el tratamiento de los poliedros.

En este proyecto se toman como referencia aquellos algoritmos cuya implementación no requiera hardware especializado, por lo que esta clase de algoritmos no es tenida en consideración.

3.1.4. Algoritmos de remuestreo con multipaso

Este grupo de algoritmos dirigidos por el volumen basa su técnica en el remuestreo del volumen completo, de manera que sus *voxels* queden alineados uno detrás de otro en la dirección del eje de vista. Es entonces cuando éstos pueden ser tratados de la misma manera que haría un algoritmo de trazado de rayos, con la particularidad de que los rayos de vista ya están alineados con los ejes del volumen remuestreado.

Esta familia de algoritmos utiliza métodos multipaso paso remuestrear el volumen: la transformación de vista es factorizada en una secuencia de deformaciones y escalados que, a continuación, se aplican al volumen en diferentes pasos. El remuestreo con multipaso, introducido en [Catmull & Smith 1980] para imágenes 2D, fue aplicado por primera vez en el renderizado de volumen en [Drebin 1988].

Como ocurre con la anterior familia de algoritmos, el remuestreo con multipaso también requiere un hardware especializado. Además, la complejidad de los filtros de remuestreo utilizados para evitar la degradación de la imagen en cada uno de los pasos, unida a la dependencia del punto de vista del objeto, hacen a este grupo de algoritmos poco apropiados para tomarlos como referencia, de cara a conseguir un algoritmo eficiente de *volume rendering* en ordenadores de propósito general.

3.2. TÉCNICAS DE ACELERACIÓN

Aunque son muchas las optimizaciones propuestas para acelerar el renderizado de volumen, en este proyecto sólo se tienen en cuenta aquellas dirigidas a los algoritmos de trazado de rayos y salpicado, puesto que, como se ha explicado anteriormente, son las familias de algoritmos más adecuadas para renderizar volúmenes muestreados mediante una rejilla regular en ordenadores de propósito general.

Así mismo, las técnicas de aceleración consideradas en este proyecto no disminuyen la calidad de la imagen a favor de una mayor velocidad de renderizado[5].

3.2.1. Estructuras espaciales de datos

Una técnica de aceleración establecida en *volume rendering* consiste en explotar la coherencia en el volumen, manejando **estructuras espaciales de datos**. Para la visualización de un conjunto de datos, normalmente hay unos grupos de *voxels* que aportan gran información a la imagen y otros que son irrelevantes. El propósito de una estructura espacial de datos es codificar este tipo de coherencia, de tal manera que los *voxels* irrelevantes puedan ser eliminados eficientemente.

Estas estructuras son a menudo utilizadas para codificar la localización de *voxels* con alta opacidad en volúmenes clasificados. Ejemplos de estructuras son los *octrees* y las pirámides de [Meagher 1982] y [Levoy 1990], los árboles k-d de [Subramanian & Fussell 1990], la estructura basada en codificación *run-length* de [Reynolds 1987] y [Montani & Scopigno 1990]. Los algoritmos de renderizado usan estas estructuras para ignorar los *voxels* transparentes de forma rápida.

Desgraciadamente, los algoritmos que utilizan estas estructuras de datos requieren una etapa de preprocesamiento para rellenarlas con los datos del volumen. No obstante, el coste de esta etapa es aceptable si las estructuras precalculadas pueden ser utilizadas en varios renderizados de un mismo volumen. Sin embargo, ninguna de las optimizaciones de coherencia existentes es útil para la clasificación interactiva de volúmenes, puesto que todas ellas requieren preprocesamiento cada vez que la opacidad del *voxel* cambia.

La mejora de rendimiento alcanzable con estas optimizaciones de coherencia es siempre dependiente de los datos, pero los volúmenes clasificados que se utilizan normalmente tienen un alto grado de coherencia. Más aún, el porcentaje de *voxels* transparentes oscila entre un 70% y un 80% tal y como se refleja en [Levoy 1990] y [Subramanian & Fussell 1990], así que la coherencia es muy útil para eliminar gran parte del trabajo realizado por un algoritmo de renderizado. Por ejemplo, en [Levoy 1990] se afirma que, en imagen médica, el uso de un *octree* para explotar la coherencia en un algoritmo de trazado de rayos, consigue una mejora de 3 a 5 veces sobre el rendimiento de un algoritmo de trazado de rayos de fuerza bruta. Además, la utilización de este tipo de estructuras supone una gran ventaja para los algoritmos dirigidos por el volumen.

[5] Técnicas como el submuestreo del volumen [Laur & Hanrahan 1991] o muestreo adaptativo [Levoy 1990], [Danskin & Hanrahan 1992], causan pérdida o emborronamiento de la imagen.

Para diseñar el algoritmo de este proyecto se utiliza la estructura basada en la **codificación *run-length***, debido a que su sencillez de implementación y mantenimiento conducen a un algoritmo más rápido frente a las complejas estructuras citadas anteriormente.

3.2.2. Terminación temprana de rayos

Otra de las técnicas comunes en *volume rendering* es la llamada **terminación temprana de rayos**. A diferencia de la anterior, esta optimización está orientada, por su facilidad de implementación, a los algoritmos de trazado de rayos.

El método consiste en hacer que cada uno de los rayos emitidos por el algoritmo termine su recorrido en cuanto la opacidad acumulada alcance un umbral determinado, cercano a la opacidad total[6]. Cualquier *voxel* que quede por recorrer queda así descartado y, por lo tanto, no es analizado por el algoritmo, con el consiguiente ahorro en tiempo de computación.

El objetivo de esta técnica es reducir o eliminar muestras residentes en regiones ocultas del volumen. En [Levoy 1990] se afirma que, para imágenes médicas, un algoritmo de trazado de rayos con esta optimización, utilizando un umbral del 95% de opacidad, consigue una mejora del rendimiento de entre 1.6 y 2.2 veces más que sin ella. Además, combinándola con el uso de un *octree*, la mejora alcanzada oscila entre 5 y 11.

Como se ha mencionado previamente, la implementación de esta técnica es más difícil en los algoritmos de salpicado ya que es necesario comprobar la opacidad de cada *pixel* de la imagen antes de añadir la contribución del siguiente *voxel* procesado. Esto reduce el número de operaciones de tonalidad, remuestreo y adición de contribución, pero no reduce el tiempo empleado en atravesar las regiones ocultas del volumen ya que el algoritmo debe atravesar de todos modos el volumen completo.

En [Reynolds 1987] se propone una técnica muy eficiente llamada "pantalla dinámica" para implementar esta técnica en algoritmos dirigidos por el volumen. Consiste en utilizar la codificación *run-length* para obtener la coherencia tanto en la imagen como en el volumen. Además, los autores afirman que: "*si la transformación del punto de vista se basa en una proyección paralela, entonces líneas paralelas en el volumen se convierten en líneas paralelas en la imagen*". Por tanto, para cualquier rotación en la imagen, cada trayectoria lineal del volumen recorrida por el algoritmo es proyectada en la imagen siguiendo una trayectoria paralela a la correspondiente trayectoria lineal de la imagen rotada.

Esta observación conduce a un eficiente algoritmo de renderizado: *en cada trayectoria lineal del volumen recorrida por el algoritmo (en orden de menor a mayor profundidad) se recorre al mismo tiempo la correspondiente trayectoria lineal en la*

[6] Existen dos generalizaciones adicionales en esta técnica: la llamada "ruleta rusa", [Arvo & Kirk 1990] y [Danskin & Hanrahan 1992], en la cual cada rayo termina su recorrido de acuerdo con una probabilidad que aumenta en función de la opacidad acumulada por el rayo; y la denominada "aceleración β", [Danskin & Hanrahan 1992], la cual decrementa la frecuencia de muestreo a lo largo del rayo a medida que aumenta la distancia óptica desde el observador.

imagen. Durante el recorrido de la trayectoria por el volumen el algoritmo usa una codificación run-length para esquivar voxels transparentes; y durante la trayectoria de la imagen, otra para esquivar los voxels ocultos. Después de recorrer todas las trayectorias del volumen, la imagen 2D resultante es rotada en la orientación correcta.

Como consecuencia, con esta técnica sólo se analiza la contribución de *voxels* no totalmente transparentes a proyectar en *pixels* no completamente opacos.

3.3. COMBINACIÓN DE MÉTODOS

Una vez revisadas las diferentes familias de algoritmos, se pueden extraer las siguientes conclusiones:

ALGORITMOS DIRIGIDOS POR LA IMAGEN

- **Ventajas**:
  - Filtros de remuestreo más simples y de mayor calidad.
  - Técnica de aceleración aplicable: "Terminación temprana de rayos".
- **Inconvenientes**:
  - Alto tiempo de cómputo para atravesar el volumen.

ALGORITMOS DIRIGIDOS POR EL VOLUMEN

- **Ventajas**:
  - Recorren el volumen en orden de almacenamiento, evitando así problemas de localidad espacial en memoria.
  - Técnica de aceleración aplicable: "Estructuras espaciales de datos".
- **Inconvenientes**:
  - Filtros de remuestreo muy costosos en cálculo por ser dependientes del punto de vista.

En el siguiente capítulo se describe un método que permite combinar las ventajas de los dos grupos de algoritmos, incluyendo sus técnicas de aceleración. Dicha técnica se llama **factorización *shear-warp*** [Lacroute 1995].

4 LA FACTORIZACIÓN *SHEAR-WARP*

En el capítulo anterior se explica que los algoritmos dirigidos por la imagen, tales como los de trazado de rayos, tienen las ventajas de eficiencia, remuestreo de alta calidad y permiten una implementación muy sencilla de la optimización basada en la "terminación temprana de rayos".

Por otra parte, los algoritmos dirigidos por el volumen, como los de salpicado, requieren una aritmética de direccionamiento más simple y consiguen grandes mejoras en rendimiento, gracias a la utilización de estructuras espaciales de datos, debido a que recorren el volumen en su orden de almacenamiento natural en memoria.

El algoritmo que se desarrolla en este proyecto combina todas estas ventajas. Para ello, se basa en una técnica existente denominada "**factorización *shear-warp***" [Lacroute 1995] para transformaciones afines del punto de vista[7]. Dicha técnica consiste en la descomposición de la transformación del punto de vista, que simplifica la proyección de los *voxels* del volumen en la imagen, permitiendo así construir un algoritmo dirigido por el volumen con las mismas ventajas que los dirigidos por la imagen.

En este capítulo se define primero la factorización *shear-warp* y se motiva su uso con un sencillo algoritmo de renderizado de fuerza bruta. A continuación, se exponen las propiedades de esta factorización, que se aplican en el siguiente capítulo para implementar el algoritmo de renderizado, cuyas indicaciones generales se describen en [Lacroute 1995].

4.1. GENERALIDADES

La relación arbitraria existente entre el sistema de referencia asociado al volumen y el correspondiente a la imagen complica, en los algoritmos dirigidos por el volumen, la eficiencia y alta calidad tanto de los filtros de remuestreo como de la proyección de *voxels*. Por otra parte, esta relación exige a los dirigidos por la imagen una aritmética de direccionamiento extra. Este problema puede ser resuelto transformando el sistema de referencia del volumen en uno intermedio, elegido de tal manera que requiera la aplicación de una transformación muy simple entre ambos, junto con una proyección eficiente en la imagen final. Esta técnica se denomina ***shearing*** y al sistema de referencia intermedio: "**sistema de referencia trasladado**" cuya definición se expone a continuación:

> ***Sistema de referencia trasladado:*** *Aquél en el que todos los rayos de vista son paralelos al tercer eje de coordenadas del volumen.*

[7] Existe también una variante de esta factorización, aplicada a transformaciones de perspectiva del punto de vista tal y como se explica en [Lacroute 1995].

Tras la transformación, el volumen queda trasladado paralelo a aquel conjunto de cortes más perpendicular a la dirección de vista, resultando los rayos de vista perpendiculares a dichos cortes. La **Ilustración 7** muestra la transformación entre ambos sistemas. Como se puede observar, consiste simplemente en trasladar cada corte para proyectarlo sobre la imagen, operación eficiente y muy sencilla de implementar.

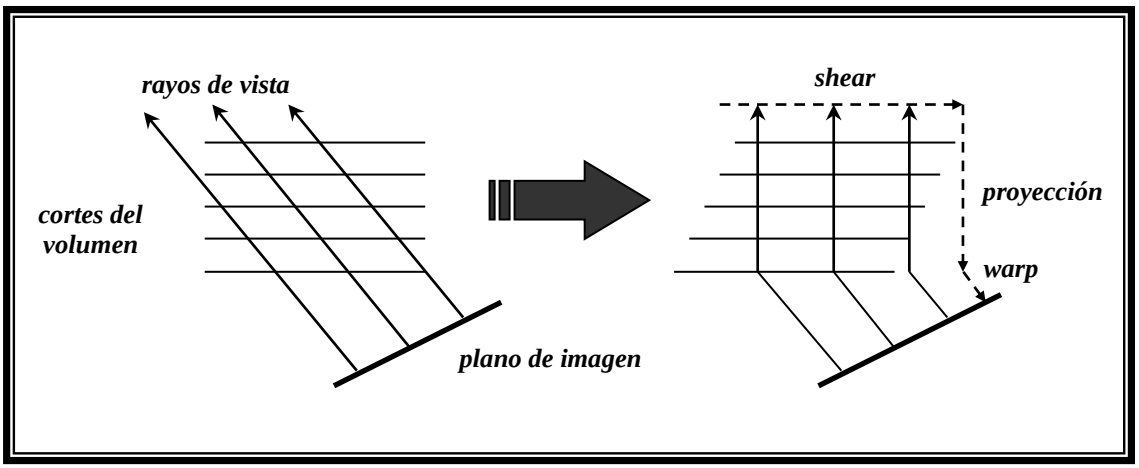


Ilustración 7.- Las líneas horizontales representan los cortes que forman el volumen en sección transversal. Para transformar el volumen al sistema de referencia trasladado basta con trasladar cada corte. En este sistema de referencia se pueden proyectar fácilmente los cortes sobre la imagen.

4.2. LA FACTORIZACIÓN *SHEAR-WARP* AFÍN

La factorización afín se utiliza para renderizar volúmenes con proyección paralela. Matemáticamente, este método parte de una matriz de transformación 4x4, obteniendo como resultado la descomposición de dicha matriz en dos factores: la **matriz *shear*** y la **matriz *warp*** [Lacroute 1995].

4.2.1. Definiciones y sistemas de coordenadas

La posición y orientación del observador con respecto al objeto a renderizar se expresa mediante la **matriz de transformación de vista**. Dicha matriz transforma coordenadas del volumen en coordenadas de la imagen final. Los sistemas de coordenadas involucrados en dicha transformación corresponden a: coordenadas del objeto, coordenadas estándar del objeto, coordenadas del sistema de referencia trasladado y coordenadas de la imagen. La **Ilustración 8** detalla dichos sistemas.

El **sistema de coordenadas del objeto** es el sistema natural asociado al volumen, es decir, aquél en el que están expresadas las coordenadas de cada uno de los *voxels* que lo componen. Su origen está localizado en uno de los vértices, y la unidad en cada eje es exactamente la longitud de un *voxel* a lo largo de dicho eje. Los ejes son: x_o, y_o y z_o.

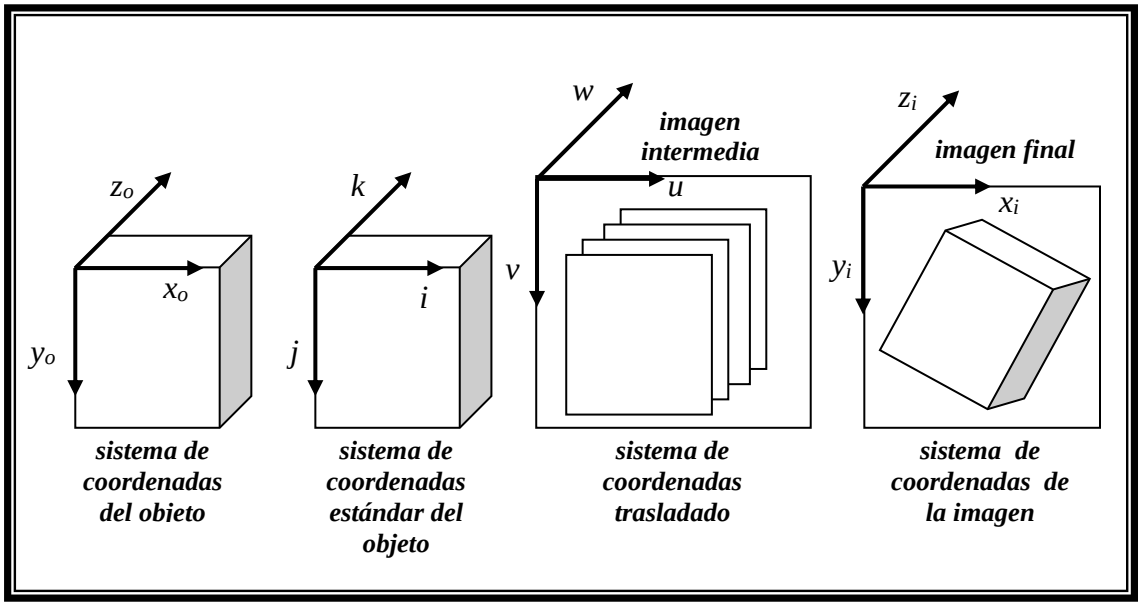


Ilustración 8.- Sistemas de coordenadas utilizados en la factorización *shear-warp*.

Para obtener el **sistema de coordenadas estándar del objeto** se define el "**eje principal de vista**" como aquel eje de coordenadas del sistema de coordenadas del objeto más paralelo a la dirección observador-objeto o dirección de vista. Se obtiene reorientando los ejes del anterior sistema de forma que el eje principal de vista se convierta en el tercer eje de coordenadas. Los nombres de sus ejes son: *x*, *j* y *k*.

El **sistema de coordenadas trasladado** se genera, como se explicó anteriormente, aplicando la técnica de *shearing* al sistema de coordenadas estándar del objeto. Dicha técnica es la transformación que aplica una de las dos matrices finales de esta factorización: la matriz *shear*. Este sistema es también denominado **sistema de coordenadas de la imagen intermedia**. El origen está situado en el vértice superior izquierdo de dicha imagen y sus ejes son: *u*, *v* y *w*.

El **sistema de coordenadas de la imagen** es el correspondiente a la imagen final, objetivo del algoritmo de renderizado. La técnica de *warping*, aplicada por la segunda de las matrices obtenidas al final de esta factorización, es la responsable de transformar el sistema de coordenadas trasladado en éste. El origen de este sistema se encuentra en el vértice superior izquierdo de la imagen final y sus ejes son: x_i, y_i y z_i.

A partir de ahora, $\vec{v}$ representa un vector y v_x una de sus componentes. La matriz de transformación de vista es una matriz 4x4 denominada M_{vista}, que transforma coordenadas del objeto en coordenadas de la imagen final:

$$\begin{bmatrix} x_i \\ y_i \\ z_i \\ w_i \end{bmatrix} = M_{vista} * \begin{bmatrix} x_o \\ y_o \\ z_o \\ w_o \end{bmatrix}$$

4.2.2. Fases de la factorización

El objetivo final de la factorización *shear-warp* afín es descomponer la matriz de transformación de vista M_{vista} de la siguiente manera:

$$M_{vista} = M_{warp} * M_{shear}$$

siendo M_{shear} la matriz *shear* y M_{warp} la matriz *warp*.

A continuación, se detallan cada uno de los pasos a seguir para obtener dichas matrices.

4.2.2.1. Obtención del eje principal de vista

Se define el **eje principal de vista** como aquel eje del sistema de coordenadas del objeto que forma el ángulo más pequeño con el vector director de vista[8]. En coordenadas de la imagen, el vector director de vista $\vec{v}_i$ es:

$$\vec{v}_i = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Sea $\vec{v}_o$ el vector director de vista expresado en coordenadas del objeto, se obtiene el siguiente sistema lineal de ecuaciones:

$$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} * \begin{bmatrix} v_{o,x} \\ v_{o,y} \\ v_{o,z} \end{bmatrix}$$

donde los m_{ij} (con $i, j \in \{1, 2, 3\}$) son los elementos de la matriz M_{vista}. Como $\vec{v}_i$ y $\vec{v}_o$ son vectores, sólo son necesarias las componentes correspondientes a la matriz 3x3 del menor complementario m_{33}. Dicha matriz se denota $M_{vista,3x3}$. Utilizando la regla de Cramer se obtiene:

$$\vec{v}_o = \begin{bmatrix} m_{12}m_{23} - m_{22}m_{13} \\ m_{21}m_{13} - m_{11}m_{23} \\ m_{11}m_{22} - m_{21}m_{12} \end{bmatrix}$$

[8] Esta definición no es equivalente a la elección del vector normal al conjunto de cortes del volumen más perpendicular a la dirección de vista.

El coseno del ángulo formado por el vector director de vista y cada uno de los ejes del sistema de coordenadas del objeto, es proporcional al producto escalar de $\vec{v}_o$ con cada uno de los vectores unitarios de este sistema de coordenadas. El producto escalar más grande corresponde al ángulo más pequeño. De esta manera se obtiene el eje principal de vista:

$$c = max\left(\left|v_{o,x}\right|, \left|v_{o,y}\right|, \left|v_{o,z}\right|\right)$$

Según sea c igual a $\left|v_{o,x}\right|$, $\left|v_{o,y}\right|$ ó $\left|v_{o,z}\right|$, el eje de principal de vista es x_o, y_o ó z_o respectivamente.

4.2.2.2. Transformación al sistema de coordenadas estándar

El algoritmo de renderizado que utiliza esta factorización opera remuestreando y componiendo el conjunto de cortes más perpendicular al eje principal de vista.

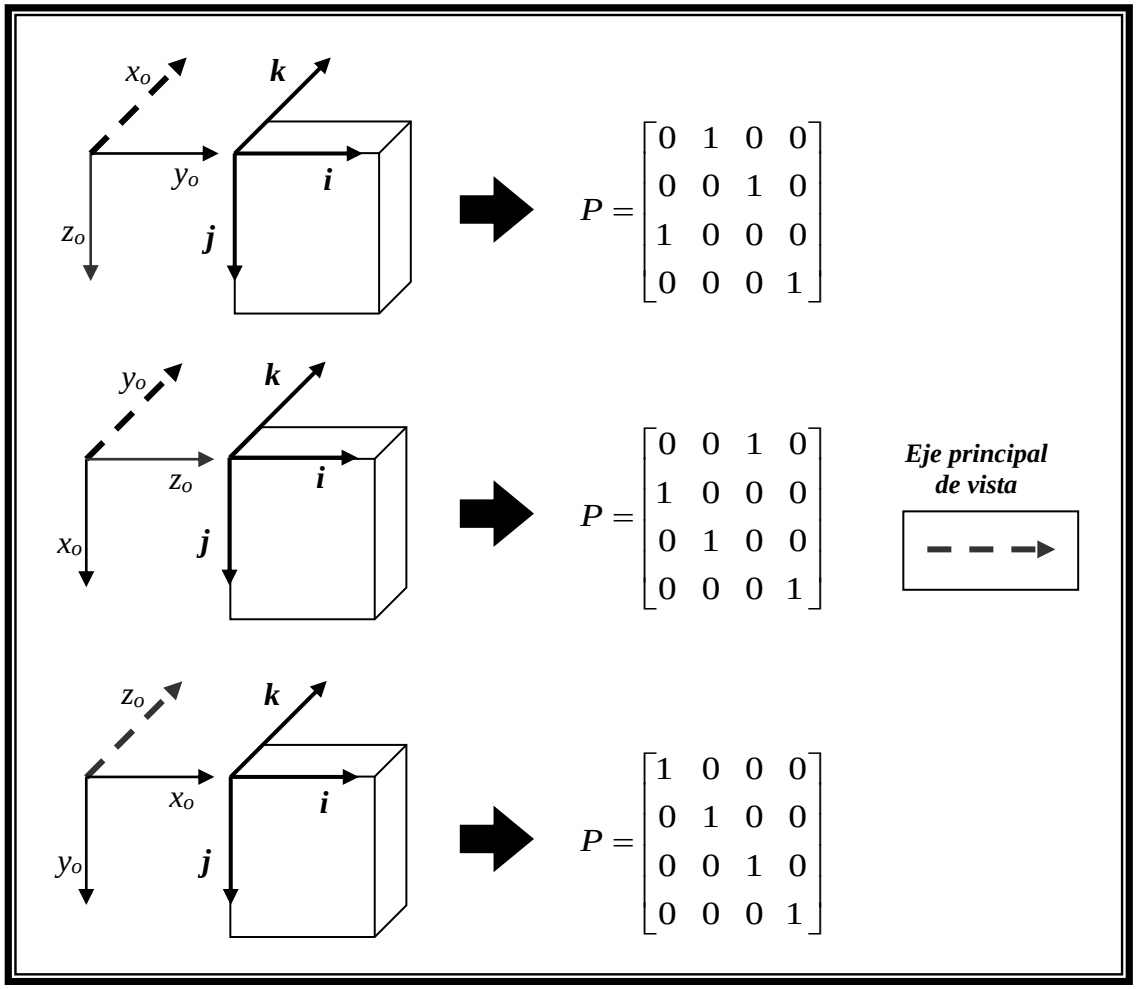


Ilustración 9.- Correspondencia entre los ejes del sistema de coordenadas del objeto (x_o, y_o, z_o) y el sistema de coordenadas estándar (i, j, k), con la respectiva matriz de transformación P aplicada sobre aquél.

Para eliminar casos especiales en cada uno de los tres ejes, se transforma el volumen a coordenadas estándar. Para ello, se reorienta el sistema de coordenadas del objeto según sea el eje principal de vista. En la **Ilustración 9** se muestra cada una de las reorientaciones, la respectiva correspondencia entre los ejes de ambos sistemas de coordenadas, así como la transformación geométrica *P* necesaria.

El resultado de esta transformación altera la matriz de vista M_{vista} pasándose a llamar M'_{vista}:

$$M'_{vista} = M_{vista} * P^{-1}$$

donde P^{-1} es la inversa de la matriz *P*.

Esta nueva matriz transforma coordenadas estándar del volumen en coordenadas de la imagen. En el sistema de coordenadas estándar el eje *k* es siempre el eje principal de vista.

4.2.2.3. Los coeficientes de *shearing* y *warping*

El siguiente paso en el proceso consiste en factorizar la nueva matriz de vista M'_{vista} en la matriz M'_{shear} y la matriz M'_{warp}. Esta factorización debe satisfacer la siguiente condición: *después de la transformación shear, el vector director de vista debe quedar perpendicular al plano (i, j).*

En coordenadas estándar, el vector director de vista es:

$$\vec{v}_{so} = P * \vec{v}_o = \begin{bmatrix} m'_{12}m'_{23} - m'_{22}m'_{13} \\ m'_{21}m'_{13} - m'_{11}m'_{23} \\ m'_{11}m'_{22} - m'_{21}m'_{12} \end{bmatrix}$$

donde los m'_{ij} (con $i, j \in \{1, 2, 3\}$) son los elementos de la matriz M'_{vista}.

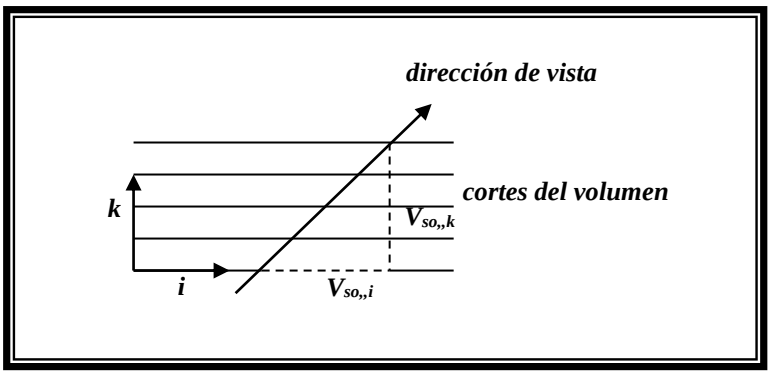


Ilustración 10.- Determinación de los coeficientes de *shear*: el diagrama muestra una sección transversal del volumen y la dirección de vista en el sistema de coordenadas estándar del objeto.

La proyección del vector director en el plano *(i, k)* tiene una inclinación igual a $v_{so,i} / v_{so,k}$ (ver **Ilustración 10**). Por lo tanto, la traslación de los cortes en la dirección *i* necesaria para hacer el vector director de vista perpendicular a dichos cortes es igual a $-v_{so,i} / v_{so,k}$. El mismo planteamiento sirve para obtener la traslación en la dirección *j*. Así, los coeficientes de *shearing* son:

$$s_i = -\frac{v_{so,i}}{v_{so,k}} = \frac{m'_{22}m'_{13} - m'_{12}m'_{23}}{m'_{11}m'_{22} - m'_{21}m'_{12}}$$

$$s_j = -\frac{v_{so,j}}{v_{so,k}} = \frac{m'_{11}m'_{23} - m'_{21}m'_{13}}{m'_{11}m'_{22} - m'_{21}m'_{12}}$$

La matriz M'_{vista} queda, de momento, factorizada de la siguiente manera:

$$M'_{vista} = M'_{vista} * \begin{bmatrix} 1 & 0 & -s_i & 0 \\ 0 & 1 & -s_j & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & s_i & 0 \\ 0 & 1 & s_j & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} m'_{11} & m'_{12} & \left(m'_{13} - s_i m'_{11} - s_j m'_{12}\right) & m'_{14} \\ m'_{21} & m'_{22} & \left(m'_{23} - s_i m'_{21} - s_j m'_{22}\right) & m'_{24} \\ m'_{31} & m'_{32} & \left(m'_{33} - s_i m'_{31} - s_j m'_{32}\right) & m'_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & s_i & 0 \\ 0 & 1 & s_j & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

El factor de la izquierda es una matriz afín de *warping* y el factor de la derecha un factor de *shearing*. Se han construido las dos matrices de manera que, después de aplicar la transformación *shear* a un volumen, sea posible calcular una proyección 2D mediante la composición de sus cortes con *k* constante en el corte con $k = 0$. Esta composición da como resultado la imagen intermedia, a la que se aplica después la transformación de *warping*.

Al aplicar la transformación *shear* al sistema de coordenadas estándar se obtiene un nuevo sistema trasladado para la imagen intermedia. Pero este sistema no es conveniente porque el origen no está localizado en un vértice de la imagen. Es necesario, por tanto, trasladar el nuevo sistema para reponer el origen en el vértice superior izquierdo. Con esta nueva traslación queda definido el sistema de coordenadas de la imagen intermedia.

La **Ilustración 11** muestra los cuatro posibles casos y sus correspondientes fórmulas para calcular la traslación correspondiente. Cada caso se identifica por el signo de los dos coeficientes de *shear*.

El algoritmo de renderizado necesita conocer, además, el orden de apilamiento de los cortes proyectados en la imagen intermedia. Éstos están ordenados por su coordenada *k*. Así, un recorrido de los cortes en orden de delante a atrás se corresponde con un bucle que itere sobre ellos en orden ascendente o descendente dependiendo del sentido del vector director de vista. El orden de apilamiento se deduce examinando la componente de este vector que corresponde al eje principal de vista: $v_{so,k}$. Si ésta es positiva, entonces el corte situado en el plano $k = 0$ es el corte frontal; si no, lo es el situado en $k = k_{max}$.

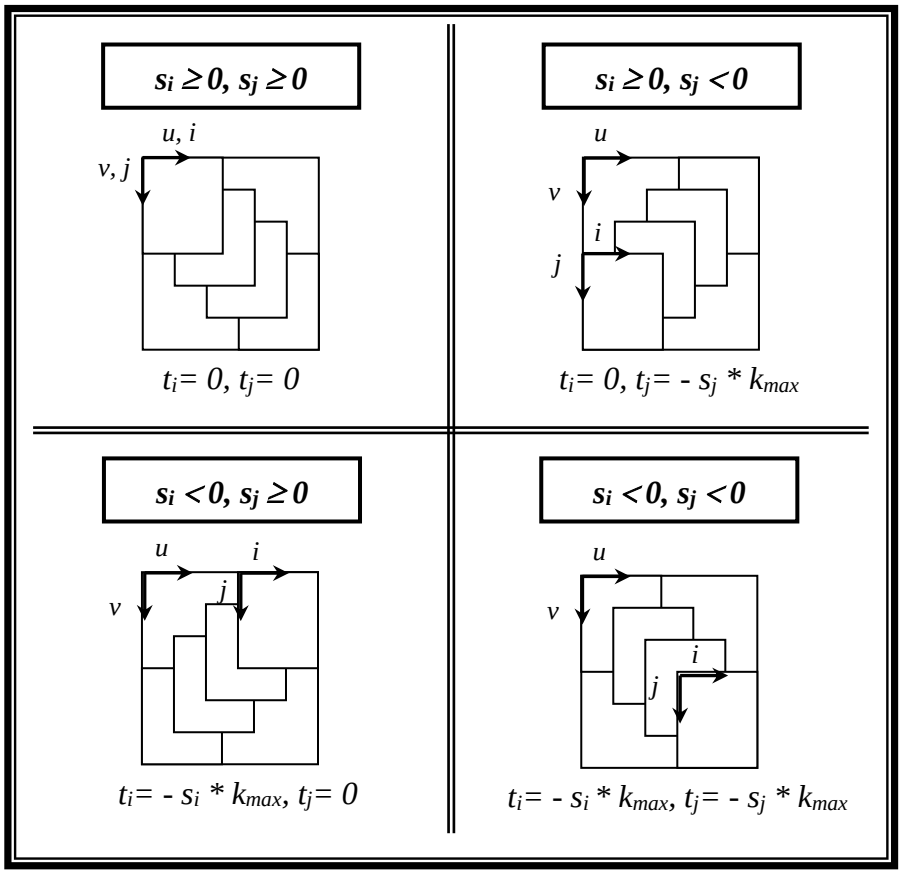


Ilustración 11.- Definición del sistema de coordenadas de la imagen intermedia. Los signos de s_i y s_j distinguen cada caso. Las traslaciones t_i y t_j especifican el desplazamiento desde el origen del sistema de coordenadas estándar ($i = 0, j = 0$) hasta el origen del sistema de coordenadas de la imagen intermedia ($u = 0, v = 0$). k_{max} es el índice del último corte del volumen.

Una vez conocida la traslación adecuada, se pueden reescribir las matrices de *shear* y *warp*. La matriz de *shear*, que transforma coordenadas estándar en coordenadas de la imagen intermedia, es:

$$M_{shear} = \begin{bmatrix} 1 & 0 & 0 & t_i \\ 0 & 1 & 0 & t_j \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & s_i & 0 \\ 0 & 1 & s_j & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & s_i & t_i \\ 0 & 1 & s_j & t_j \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Y la matriz de *warp*, que transforma coordenadas de la imagen intermedia en coordenadas de la imagen final, queda así:

$$M_{warp} = \begin{bmatrix} m'_{11} & m'_{12} & \left(m'_{13} - s_i m'_{11} - s_j m'_{12}\right) & m'_{14} \\ m'_{21} & m'_{22} & \left(m'_{23} - s_i m'_{21} - s_j m'_{22}\right) & m'_{24} \\ m'_{31} & m'_{32} & \left(m'_{33} - s_i m'_{31} - s_j m'_{32}\right) & m'_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -t_i \\ 0 & 1 & 0 & -t_j \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Pero, dado que esta técnica de *warping* se realiza en 2D, es necesario eliminar la tercera fila y columna de M_{warp}, obteniendo la matriz M_{warp2D}:

$$M_{warp2D} = \begin{bmatrix} m'_{11} & m'_{12} & \left(m'_{14} - t_i m'_{11} - t_j m'_{12}\right) \\ m'_{21} & m'_{22} & \left(m'_{24} - t_i m'_{21} - t_j m'_{22}\right) \\ 0 & 0 & 1 \end{bmatrix}$$

Esta matriz 3x3 transforma la imagen intermedia en la imagen final de esta forma:

$$\begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} = M_{warp2D} * \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

4.2.3. La factorización shear-warp afín completa

En resumen, la factorización *shear-warp* de una matriz de transformación de vista afín cualquiera, requiere una reorientación de ejes *P*, una transformación *shear* 3D M_{shear} y una transformación *warp* 2D M_{warp2D}:

$$M_{vista} = M_{warp2D} * M_{shear} * P$$

Para conseguirla, es necesario seguir los siguientes pasos:

1. Encontrar el eje principal de vista y realizar la reorientación adecuada de los ejes del sistema de coordenadas del objeto.
2. Calcular los coeficientes de *shearing*.
3. Obtener la traslación entre el origen del sistema de coordenadas estándar y el del sistema de coordenadas de la imagen intermedia.
4. Calcular la matriz de *shear* 3D y la matriz de *warp* 2D.

El algoritmo de renderizado utiliza el eje principal de vista para elegir el conjunto de cortes a remuestrear y componer. El signo de $v_{so,k}$ determina el orden de

apilamiento de los cortes en el momento de atravesar el volumen en orden de delante a atrás. El algoritmo calcula la localización de cada corte en la imagen intermedia usando la matriz de *shear* y produce la imagen final mediante la matriz de *warp*.

4.2.4. Propiedades de la factorización shear-warp

La proyección desde el volumen sobre la imagen intermedia tiene algunas propiedades geométricas que simplifican el algoritmo de renderizado.

1.- Las líneas de *pixels* en la imagen intermedia son paralelas a las líneas de *voxels* en el volumen. Esto es debido al hecho de que la matriz de *shearing* no contiene ningún elemento de rotación.

2.- Todos los *voxels* de un corte son escalados con el mismo factor. Esta propiedad también tiene su origen en la matriz de *shearing*: dicha matriz escala cada corte uniformemente.

3.- Cada corte del volumen tiene el mismo factor de escala cuando es proyectado en la imagen intermedia; y este factor puede ser elegido arbitrariamente. En particular, se puede elegir como factor la unidad, de manera que dada una línea de *voxels*, existe una relación biyectiva con los *pixels* de la imagen intermedia.

Una implicación muy útil derivada de estas propiedades es que, para proyecciones paralelas, cada *voxel* de un determinado corte del volumen tiene los mismos pesos de remuestreo (ver **Ilustración 12**). Cada corte es simplemente trasladado, de manera que el conjunto de pesos puede ser precalculado y reutilizado para cada *voxel* de dicho corte. Este hecho elimina los problemas asociados con el remuestreo eficiente en los algoritmos dirigidos por el volumen.

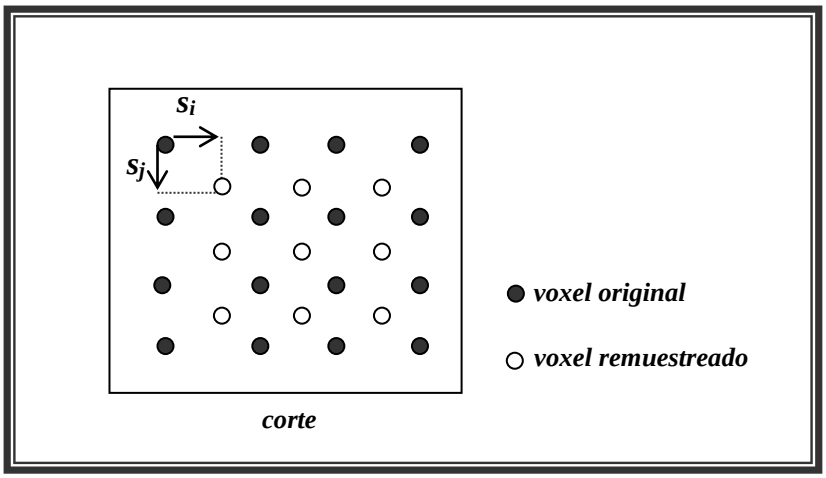


Ilustración 12.- Cada corte del volumen es simplemente trasladado, por eso todos los *voxels* de un mismo corte tienen los mismos pesos de remuestreo.

4.3. ALGORITMO RESULTANTE

En base a las propiedades anteriormente mencionadas, se describe, a continuación, un algoritmo dirigido por el volumen que las aplica[9]:

1º) Transformar el volumen al sistema de referencia trasladado mediante la técnica de ***shearing*** expuesta anteriormente. Como los parámetros de traslación no tienen por qué ser números enteros, cada corte ha de ser remuestreado adecuadamente.

2º) Componer los cortes remuestreados, en orden de menor a mayor profundidad en el sentido del vector director de vista, utilizando el operador de composición (también llamado operador *over*), que se define más adelante. Con este paso se consigue proyectar los *voxels* del volumen en una **imagen intermedia** distorsionada con coordenadas expresadas en el sistema de referencia trasladado.

3º) Transformar la imagen intermedia distorsionada en la imagen final mediante una deformación. Esta deformación es una técnica de remuestreo llamada ***warping***.

La razón por la que se calcula primeramente la imagen intermedia distorsionada es resultado de las propiedades de la factorización. Éstas conducen a una implementación eficiente, tanto de los bucles de remuestreo, como del correspondiente a la adición de contribución de color y opacidad[10]. En concreto, la conclusión a la que llevan estas propiedades es que las filas de *voxels* de cada uno de los cortes son paralelas a las filas de *pixels* de la imagen intermedia, haciendo muy sencilla la transformación desde el sistema de referencia del volumen al sistema de referencia trasladado y su posterior proyección en la imagen intermedia.

4.3.1. Ventajas del algoritmo

Aplicando las propiedades anteriores, la estructura general del algoritmo, expresada en el esquema de anidación de bucles, queda como muestra la **Ilustración 13**, y cuenta con las siguientes ventajas:

- **Algoritmo dirigido por el volumen:** como muestra el primer bucle, los cortes son recorridos en orden de almacenamiento natural, con las ventajas en cuanto a localidad espacial de los datos en memoria, disminución del tiempo de cálculo empleado en la aritmética de direccionamiento y el uso de estructuras de datos espaciales.

- **Remuestreo simple en 2D:** dado que los coeficientes de *shearing* implican sólo una traslación dentro del mismo corte; lo que hace que el remuestreo sea similar al de un algoritmo de trazado de rayos, pero en dos dimensiones, como se puede apreciar en los dos bucles más internos (x_0, y_0).

[9] Tal y como se expone en [Klein & Kübler 1985] y [Cameron & Undrill 1992].
[10] Bucles expuestos en el capítulo anterior.

- **Proyección simple:** debido a que las líneas de *pixels* son paralelas a las de *voxels*, ambas pueden ser recorridas de forma simultánea para su composición en la imagen intermedia, por lo que se pueden intercalar los bucles centrales que recorren cada *pixel* (x_i, y_i), y aplicar en ellos la aceleración por terminación temprana de rayos.

  El uso de la imagen intermedia permite eliminar el bucle (z_i), presente en los algoritmos de trazado de rayos.

- **Warping eficiente:** puesto que es una operación en 2D, que se aplica a la imagen intermedia.

```
Para cada z0 desde 1 hasta la profundidad del volumen

    Para cada yi desde 1 hasta altura de la imagen
        Para cada xi desde 1 hasta la anchura de la imagen

            Para cada y0 requerido por el filtro de remuestreo en función de (xi, yi)
                Para cada x0 requerido por el filtro de remuestreo en función de (xi, yi)
                    Combinar color y opacidad del voxel (x0, y0, z0) sobre el pixel (xi, yi)
                    de la imagen
                fPara
            fPara

        fPara
    fPara

fPara
```

Ilustración 13.- Algoritmo resultante con las propiedades de la factorización *shear-warp*.

En el siguiente capítulo se utilizan estas propiedades, junto con las optimizaciones del capítulo anterior, para diseñar el algoritmo de renderizado.

5 ALGORITMO DE RENDERIZADO CON PROYECCIÓN PARALELA

En este capítulo se describen, de forma detallada, las fases llevadas a cabo para la elaboración del algoritmo de renderizado.

5.1. ANÁLISIS DE REQUISITOS

Tras el estudio de los trabajos previos (capítulo 3) y de la técnica *shear-warping* (capítulo 4), se describen, a continuación, los requerimientos del algoritmo.

5.1.1. Requisitos funcionales

El algoritmo de renderizado de volumen debe realizar su propósito bajo los siguientes condicionantes:

- **Datos de entrada:** el algoritmo tiene que tomar como información de entrada objetos representados por volúmenes de muestras distribuidas en forma de rejilla regular, con un tamaño máximo de 256^3 *voxels*.
- **Datos de salida:** la calidad de la imagen generada debe ser suficiente para la localización de estructuras anatómicas en imagen médica, y en ningún caso se debe comprometer la calidad de la imagen, a favor de una mejora de rendimiento.
- **Coste temporal:** el tiempo de renderizado debe permitir la manipulación interactiva de objetos, es decir, ser inferior a 1 segundo.

5.1.2. Requisitos de metodología

El algoritmo de renderizado de volumen se debe desarrollar aplicando los siguientes técnicas y optimizaciones:

- **Técnica de renderizado**: Transformación de la matriz de vista utilizando la factorización *shear-warp*.
- **Técnicas de optimización**:
    - Estructura de datos espaciales.
    - Terminación temprana de rayos.

5.1.3. Requisitos de entorno

El algoritmo debe cumplir los siguientes imperativos:

- **Lenguaje de programación**: El algoritmo tiene que estar codificado en lenguaje C++.
- **Contexto de ejecución:** el algoritmo tiene que poder ser ejecutado en máquinas de propósito general, sin uso de hardware especializado.

5.2. DISEÑO DEL ALGORITMO

En esta sección se describen, justificadamente, las decisiones tomadas para construir el algoritmo, tanto a nivel funcional como estructural.

5.2.1. Visión general del algoritmo

Como se explicó en el capítulo anterior, la primera propiedad de la factorización *shear-warp* establece que las líneas de *voxels* del volumen trasladado están alineadas con las líneas de *pixels* de la imagen intermedia; lo que significa que volumen e imagen intermedia pueden ser atravesados simultáneamente. Por lo tanto, la utilización de estructuras que exploten la coherencia de los datos, tales como las basadas en la codificación *run-length* son una buena elección.

La primera estructura de datos a usar por el algoritmo es una codificación *run-length* del volumen. Dicha estructura evita el procesamiento de los *voxels* transparentes durante el renderizado. Así, el algoritmo atraviesa el volumen, corte a corte, consultando esta estructura de la misma forma que lo hace el algoritmo de fuerza bruta descrito en capítulos anteriores. Para cada corte, el algoritmo hace uso de la matriz de *shear* para calcular la traslación correspondiente a dicho corte. A continuación, recorre la estructura para determinar qué *voxels* no son transparentes. Sólo dichos *voxels* son trasladados, remuestreados y proyectados en la imagen intermedia. Y todo ello se puede realizar en un único bucle; evitando el procesamiento de porciones transparentes del volumen.

La segunda estructura a crear es también del tipo *run-length* y se aplica a la imagen intermedia. Se genera a medida que el algoritmo va construyendo dicha imagen intermedia. En este caso, la codificación consiste en asociar a cada *pixel* opaco un valor (*offset*). Éste apunta al siguiente *pixel* no opaco dentro de la misma línea de *pixels* (ver **Ilustración 14.- Codificación *run-length* de la imagen intermedia. Los *offsets* almacenados en los *pixels* opacos de la imagen intermedia permiten ignorar de una manera eficiente los *voxels* ocultos del volumen.**). Del mismo modo que en la estructura anterior, se define un umbral de opacidad que determina cuándo un *pixel* es o no opaco. De esta forma, los *voxels* pendientes por procesar, que han de ser proyectados sobre dicho *pixel*, son ignorados por el algoritmo, con el consecuente ahorro en procesamiento. Los *offsets* sirven para no procesar tramos completos de *pixels* opacos. Se debe almacenar un *offset* para cada *pixel* del tramo opaco (en vez de un único *offset* al comienzo de éste), debido a que es posible saltar al interior de dicho tramo después de atravesar uno de *voxels* transparentes.

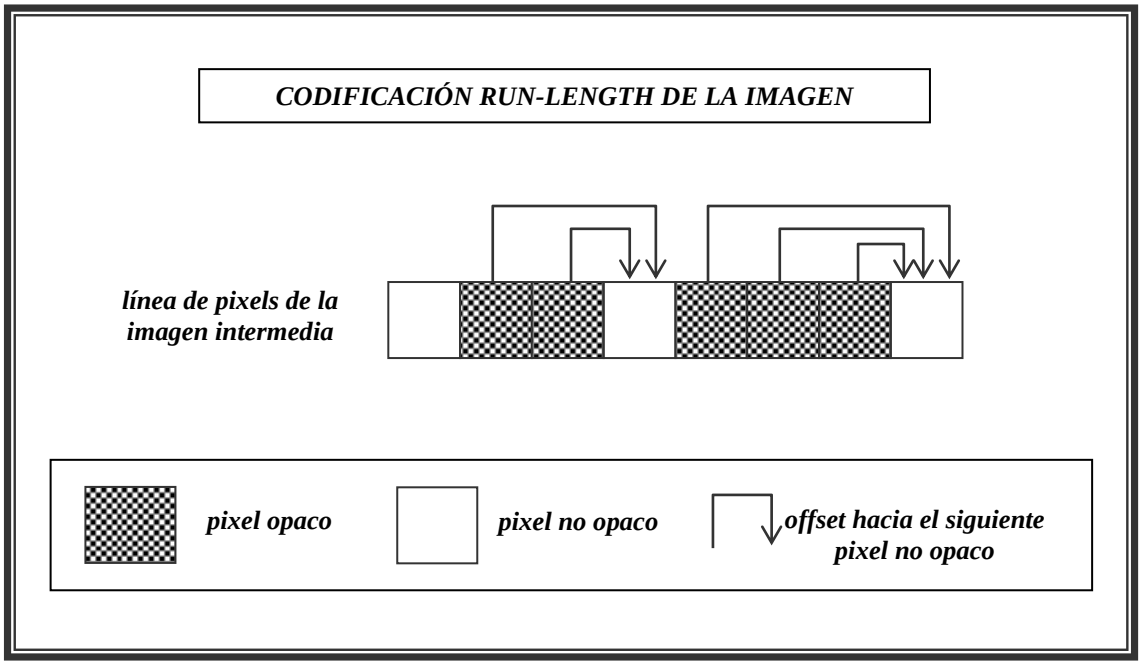


Ilustración 14.- Codificación *run-length* de la imagen intermedia. Los *offsets* almacenados en los *pixels* opacos de la imagen intermedia permiten ignorar de una manera eficiente los *voxels* ocultos del volumen.

Estas dos estructuras de datos y la primera propiedad de la factorización conllevan a un rápido algoritmo de renderizado (ver **Ilustración 15**). Éste recorre al mismo tiempo los *voxels* del volumen y los *pixels* de la imagen intermedia siguiendo un orden, línea a línea, utilizando las estructuras comentadas anteriormente para evitar trabajo de procesamiento. De esta manera se consiguen dos objetivos:

- Al avanzar siguiendo este orden, se reduce la aritmética de direccionamiento.
- Utilizando estas estructuras, se procesan sólo los *voxels* que son visibles y no transparentes.

Para los *voxels* que se han de procesar, se utiliza un bucle que realiza el remuestreo, cálculo de tonalidad y composición. Las propiedades segunda y tercera permiten simplificar la fase de remuestreo en este bucle, ya que los pesos de remuestreo utilizados son los mismos para todos los *voxels* de un mismo corte. Para la tonalidad de los voxels, se utiliza un sistema basado en tablas de normales, con el fin de calcular el color de cada *voxel*.

Una vez que el volumen ha sido compuesto sobre la imagen intermedia, el algoritmo la transforma en la imagen final. Puesto que se trata de una transformación 2D, más pequeña que el volumen, esta parte del algoritmo es relativamente rápida.

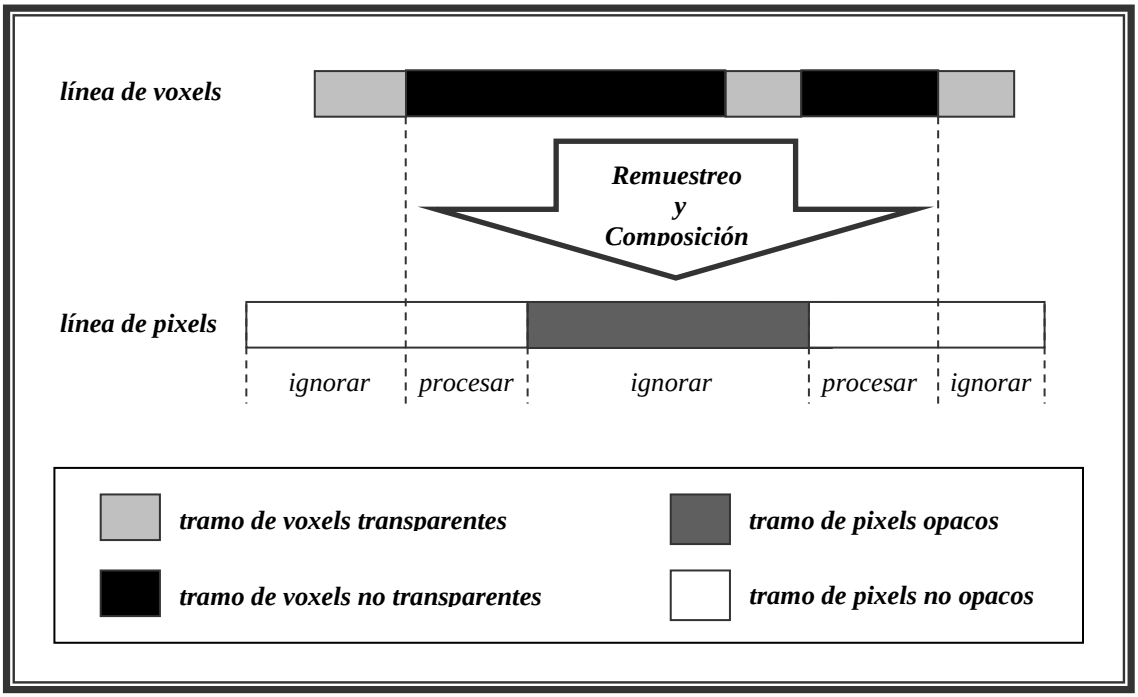


Ilustración 15.- El remuestreo y la composición son realizados recorriendo al mismo tiempo *voxels* y *pixels* línea a línea, ignorando los *voxels* transparentes y los *pixels* opacos.

5.2.2. Codificación run-length del volumen

Se trata de una estructura formada por dos tipos de tramos: transparentes y no transparentes, en función de un umbral de opacidad especificado. Cada tramo (*run*) está representado por la longitud del tramo (*length*) [Foley 1990]. El algoritmo crea esta estructura durante un preprocesamiento del volumen previo al renderizado siguiendo los siguientes pasos:

1º) Se visita cada *voxel* del volumen, en orden de almacenamiento, y se le asigna una opacidad según el tipo de tejido a visualizar.
2º) Dicha opacidad se compara con el umbral especificado para determinar si se trata de un *voxel* transparente o no.
3º) Según el resultado de la comparación, el *voxel* se inserta en el tramo correspondiente (transparente o no transparente) de la estructura.

De esta forma, el algoritmo organiza los *voxels* en tramos. Un tramo consiste en una secuencia de *voxels* contiguos, todos ellos transparentes o no transparentes. De cada tramo se guarda su longitud, pero sólo se almacenan los *voxels* no transparentes.

Si un tramo de *voxels* es más largo que la línea del volumen a recorrer, dicho tramo es partido en dos, asegurando así que la siguiente línea a recorrer comienza con un tramo nuevo.

Físicamente, la estructura de datos utilizada está formada por tres vectores (ver **Ilustración 16**):

- Vector de longitudes de tramo.
- Vector de *voxels* no transparentes
- Vector de punteros: dirigidos a las componentes de los dos primeros vectores.

El vector de *voxels* no transparentes simplemente almacena dichos *voxels* de forma contigua. Los *voxels* transparentes no son almacenados. Cada *voxel* de este vector contiene los datos necesarios de opacidad y tonalidad (requeridos para calcular su color). Esto es así porque el color de cada *voxel* depende del punto de vista y la localización de las fuentes de luz.

Como el acceso a los *voxels* no siempre se realiza de forma secuencial, se necesita un mecanismo que permita el recorrido aleatorio del volumen. Para conseguir este tipo de acceso se utiliza el vector de punteros. Cada entrada de este vector apunta a la longitud del primer tramo y al primer *voxel* no transparente, para un corte determinado del volumen, permitiendo así el acceso aleatorio a cualquier corte.

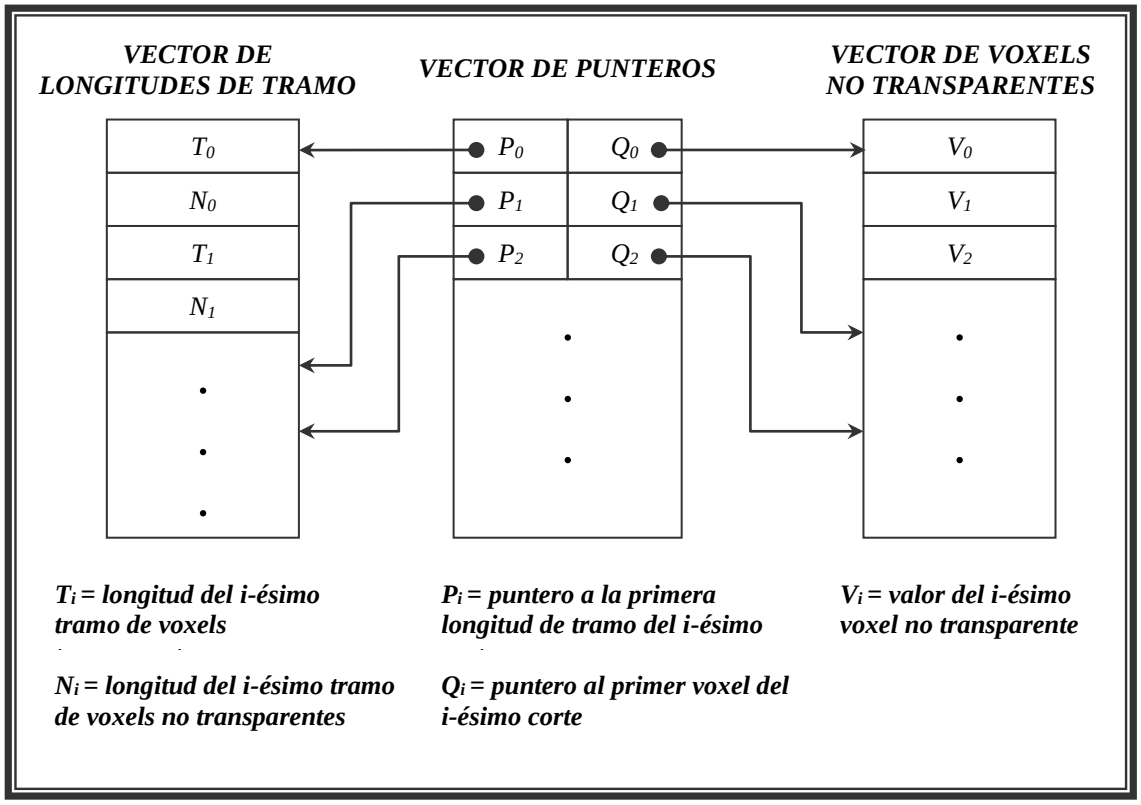


Ilustración 16.- Codificación *run-length* del volumen.

Por otra parte, en este tipo de codificación, los *voxels* quedan ordenados siguiendo la dirección de un único eje del volumen. Si este eje no coincide con el eje principal de vista requerido en un momento dado por la factorización *shear-warp*, el algoritmo se vería obligado a visitar los *voxels* en un orden diferente al de almacenamiento, contradiciendo una de las ventajas de la factorización *shear-warp*

expuesta en el capítulo anterior. Por ello, se precalculan tres codificaciones diferentes, cada una en función de uno de los tres ejes principales de vista posibles. Más tarde, el algoritmo elige la apropiada, dependiendo del eje de vista a utilizar. Es preferible cargar las tres copias en memoria simultáneamente para evitar fallos de página cuando el eje principal de vista cambia; sin embargo, mantener las tres copias en memoria no altera el rendimiento de la caché, puesto que sólo una copia es necesaria para cada renderizado. Además, dado que los *voxels* transparentes no son almacenados en la estructura, el tamaño del volumen codificado es siempre menor al original, incluso con las tres copias cargadas en memoria.

5.2.3. Codificación run-length de la imagen intermedia

La imagen intermedia también está codificada utilizando una estructura de datos del tipo *run-length*; en este caso, los tramos a codificar contienen *pixels* opacos y no opacos. Sin embargo, los requerimientos son diferentes a los exigidos por la codificación del volumen, ya que ésta puede ser precalculada, mientras que la imagen intermedia varía de un renderizado a otro. Se necesita, por lo tanto, una estructura de datos que soporte la creación dinámica de tramos de *pixels* opacos, además de ofrecer la capacidad de unir tramos adyacentes y encontrar el final de un tramo.

Este problema es equivalente al denominado *UNION-FIND*[11] [Aho 1974] y [Cormen 1990]. Este problema versa sobre un conjunto de objetos agrupados en conjuntos disjuntos. En este proyecto, los objetos son los *pixels* opacos y los conjuntos son los tramos formados por ellos. Cada conjunto contiene un representante que da nombre al grupo. Siempre se elige como representante al último *pixel* de un tramo. Además, se definen tres operaciones:

- **Crear**: genera un nuevo conjunto que contiene un único elemento.
- **Unir**: mezcla dos conjuntos en uno nuevo con un único representante.
- **Encontrar**: dado un elemento cualquiera de un conjunto, encuentra el representante de dicho conjunto.

Una estructura de datos muy común para representar conjuntos es un bosque de árboles. En el algoritmo de este proyecto cada tramo de *pixels* opacos se representa mediante un árbol (ver **Ilustración 17**). Estos árboles se implementan utilizando *offsets* almacenados con cada *pixel* en la imagen intermedia. Por lo tanto, la estructura de datos elegida para representarla será un vector 2D de *pixels*. Cada uno contiene un color, una opacidad y un *offset* relativo. Éste último valor contiene el número de *pixels* a saltar para alcanzar el nodo padre de dicho *pixel*. La raíz de cada árbol es el *pixel* no opaco situado al final del tramo (aunque dicho *pixel* no forma parte realmente del tramo por ser no opaco). Cada *pixel* no opaco tiene un *offset* igual a cero.

El algoritmo de renderizado utiliza esta estructura siguiendo varios pasos.

[11] También llamado “Problema de unión de conjuntos disjuntos”.

- Primero inicializa la imagen intermedia estableciendo la opacidad y el *offset* de todos los *pixels* a cero, es decir, genera la imagen intermedia con todos los *pixels* transparentes.
- Durante el renderizado, antes de componer un *voxel* sobre un *pixel*, comprueba el valor del *offset* asociado a dicho *pixel*. Si el *offset* no es cero, entonces el *pixel* es opaco, de manera que el algoritmo realiza la operación de "Encontrar", para localizar el final del tramo y saltar a él. Por otra parte, si el *offset* es cero, el algoritmo completa la operación de composición.
- Si la nueva opacidad del *pixel* compuesto excede el umbral de opacidad establecido, se realiza la operación de "Crear", para generar un nuevo tramo de *pixels* opacos. Posiblemente se realice, a continuación, la operación "Unir", para unir el tramo nuevo con los adyacentes. La operación "Crear" requiere una implementación muy simple: crear un nuevo tramo cuando un *pixel* se convierte en opaco consiste en establecer su *offset* a uno.
- La operación de "Unir" se realiza automáticamente gracias a la operación "Crear": si ya existe un tramo de *pixels* opacos a la izquierda del recién creado, entonces el *offset* de su último *pixel* estará apuntando al nuevo tramo. De la misma forma, el *offset* del *pixel* que forma el nuevo tramo apunta al siguiente *pixel*; si éste es opaco, entonces el tramo recién creado se convierte automáticamente en parte del siguiente.

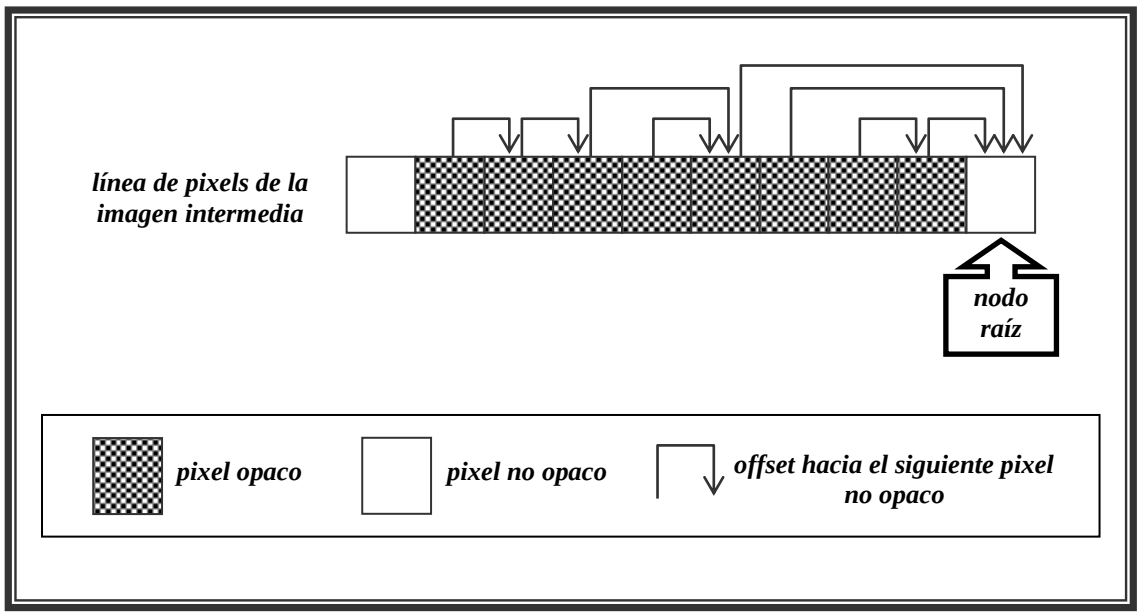


Ilustración 17.- Estructura de árbol para representar un tramo de *pixels* opacos: cada *pixel* opaco tiene un *offset* que apunta a otro *pixel* del tramo o al final de éste (nodo raíz).

- El algoritmo usa la operación "Encontrar" para localizar el final de un tramo partiendo desde cualquier *pixel* contenido en ese tramo. Esta operación puede lograrse siguiendo la cadena de *offsets* hasta llegar al primer *pixel* no opaco, que es el que tiene su *offset* igual a cero. Sin embargo, esta implementación es ineficiente porque las operaciones "Crear" y "Unir" sólo producen *offsets* iguales a la unidad, dando como resultado cadenas muy largas. Existe una

optimización muy efectiva llamada "**compresión del camino**", que reduce en gran medida la longitud media de las cadenas [Cormen 1990]. Esta optimización puede ser implementada de la siguiente manera: después de que el algoritmo atraviesa una cadena de *offsets* para encontrar el final de un tramo, vuelve a atravesar la misma cadena y actualiza cada uno de los *offsets*, haciéndolos apuntar directamente al final del tramo (ver **Ilustración 18**). Esta optimización es efectiva porque disminuye la cantidad de *pixels* a visitar en futuras composiciones sobre dichos caminos comprimidos.

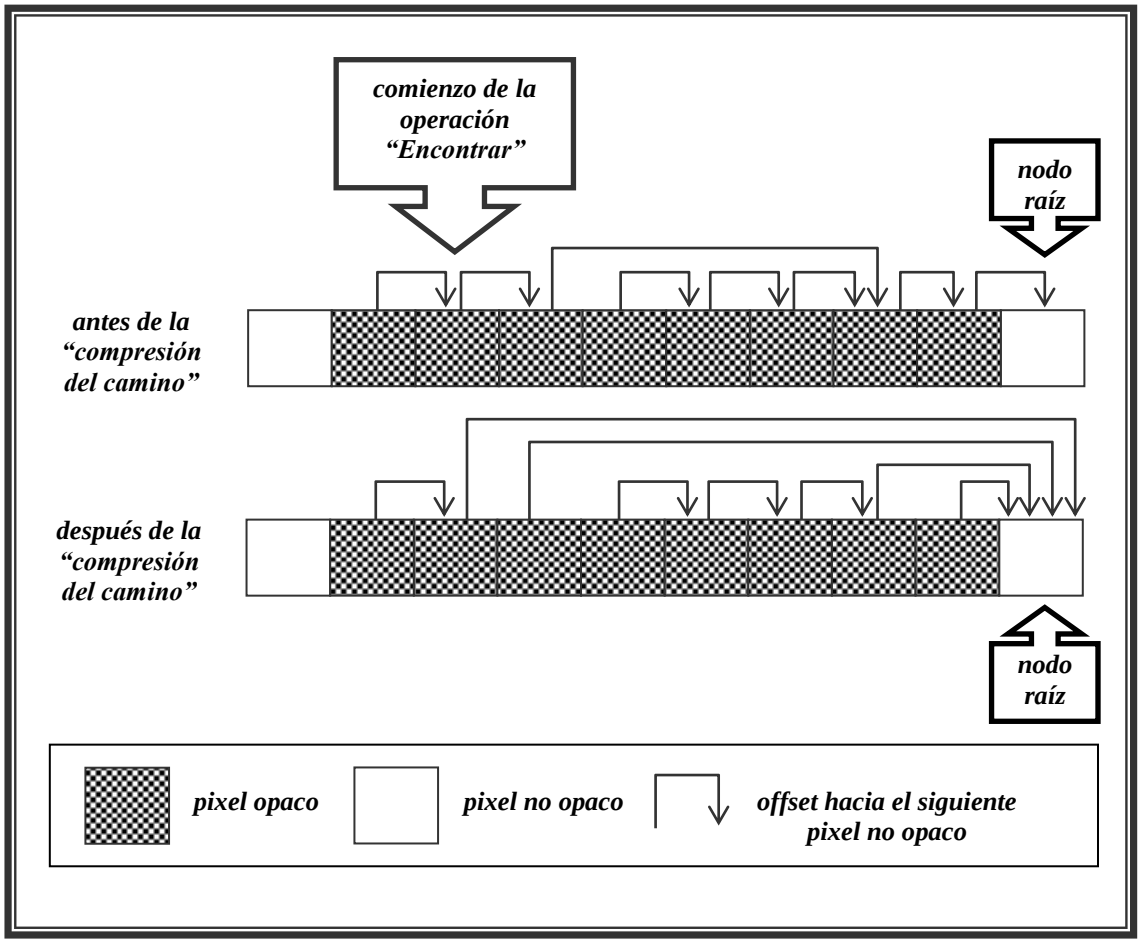


Ilustración 18.- Optimización "compresión del camino": después de recorrer la cadena de *offsets* para encontrar el final de un tramo de pixels opacos, el algoritmo vuelve a recorrerla y actualiza cada *offset* para que apunte al final del tramo.

Sin esta optimización, la complejidad asintótica de actualizar y atravesar la imagen intermedia es de $O(m^2)$ operaciones por línea, donde m es el número total de llamadas a las operaciones "Crear", "Unir" y "Encontrar". Sin embargo, con la "compresión del camino" la complejidad disminuye hasta $O(F * log(1+ F/C) * C)$, donde C es el número de operaciones "Crear" y F el número de operaciones "Encontrar" [Cormen 1990].

Puede realizarse una optimización adicional, llamada "unión por rango", para reducir la complejidad asintótica. Ésta se aplica a la operación de "Unir": cuando dos conjuntos son "unidos", el más pequeño debería apuntar al más grande. El tamaño (o

rango) de un conjunto se define como el número de niveles en el árbol que lo representa. Esta optimización tiende a mantener los árboles balanceados, reduciendo por tanto el máximo número de pasos desde un nodo hoja hasta la raíz del árbol. Para añadir la "unión por rango" al algoritmo de este proyecto debe almacenar el rango de cada árbol en el pixel localizado en la raíz y cambiar la implementación de la operación "Unir".

La combinación de las optimizaciones comentadas da como resultado una complejidad asintótica de $O(m * A(m, C))$, donde $A(m ,C)$ es la inversa de la función de Ackermann (una función que crece tan lentamente que se iguala a cuatro para este tipo de problemas). Sin embargo, el coste adicional de la operación "Unir" conlleva a la aparición de una constante más alta en el coste polinomial del algoritmo. Así, con ambas optimizaciones el algoritmo es un 5-15% más lento que con la optimización "compresión del camino" únicamente. Además, para volúmenes con 256 o 512 *voxels* por corte, el término logarítmico que aparece en la expresión de la complejidad asintótica referente a esta optimización, no es peor que $A(m, C)$: ambos crecen extremadamente despacio para las dimensiones de los volúmenes comúnmente utilizados. Por todo ello, en este proyecto se utiliza solamente la optimización "compresión del camino".

La estructura de datos utilizada por Reynolds, en su técnica de "pantalla dinámica" (ver apartado 3.2.2), tiene la misma función que el vector de *pixels* opacos, pero en vez de enlaces entre *pixels*, utiliza listas enlazadas con punteros explícitos para representar tramos de *pixels* de la imagen. En su implementación cada línea de la imagen está formada por una lista enlazada de tramos opacos y no opacos; dicha lista inicialmente contiene un tramo simple no opaco. Su algoritmo explora simultáneamente las listas enlazadas, que corresponden a una línea codificada del volumen y la correspondiente línea de la imagen, para evitar componer *voxels* transparentes y ocultos, igual que en el algoritmo desarrollado en este proyecto.

Sin embargo, la lista enlazada debe ser accedida secuencialmente, a diferencia del vector de *offsets*, que puede ser indexado en orden aleatorio. Como consecuencia, su algoritmo debe atravesar todos los tramos de una línea de la imagen, incluso si la mayoría de los *voxels* correspondientes a proyectar en ella son transparentes. Más aún, actualizar la lista enlaza requiere el uso de un de gestor de memoria dinámica. El coste de mantener un espacio de bloques libres de memoria añade sobrecarga al algoritmo, teniendo en cuenta, además, que las listas enlazadas no suelen tener una buena localidad espacial. En resumen, su algoritmo puede tener un coste computacional y de memoria superior al de este proyecto.

En general, el uso de estas estructuras de codificación hace que el algoritmo desarrollado tenga una sobrecarga computacional mayor que la "terminación temprana de rayos" en un algoritmo de trazado de rayos, ya que es necesario atravesar estructuras de datos codificadas incluso en regiones ocultas del volumen. Además, la efectividad de los enlaces entre *pixels* opacos depende de la coherencia entre las regiones opacas de la imagen. Sin embargo, los tramos de *pixels* opacos son normalmente largos, de manera que muchos *pixels* pueden ser saltados de una vez y el número de *pixels* a examinar individualmente es relativamente pequeño.

5.2.4. Remuestreo del volumen

El algoritmo de renderizado debe remuestrear cada corte del volumen cuando los desplaza al sistema de referencia trasladado. La implementación actual utiliza un filtro de interpolación bilineal, de manera que se necesitan dos líneas de *voxels* para producir una línea remuestreada. Utiliza una técnica de convolución del tipo *gather*, típica de los algoritmos de proyección hacia atrás explicados en la sección 3.1.1. Dicha técnica exige al algoritmo atravesar y decodificar las dos líneas de *voxels* simultáneamente. En consecuencia, cada una debe ser recorrida dos veces (una vez para cada una de las líneas remuestreadas a las que contribuye).

Una alternativa a esta técnica de convolución es utilizar otra del tipo *scatter*, comúnmente utilizada en los algoritmos de proyección hacia delante (ver sección 3.1.2). Con ella, el algoritmo atraviesa cada línea de *voxels* sólo una vez y calcula su contribución sobre dos líneas remuestreadas. Sin embargo, los *voxels* parcialmente remuestreados deben ser almacenados en un *buffer* temporal, hasta que se hayan procesado completamente, para evitar reordenar los pasos a seguir por los filtros de composición y reconstrucción. Más aún, el *buffer* temporal debe ser codificado utilizando la técnica *run-length* para mantener los beneficios del resto de estructuras de coherencia entre datos.

Durante el remuestreo, el algoritmo ignora aquellas regiones donde ambas líneas contienen tramos de *voxels* transparentes. Además, utiliza los enlaces entre pixels opacos para ignorar las regiones ocultas, de manera que realmente sólo los *voxels* que no son transparentes ni ocultos se remuestrean. El coste del remuestreo se reduce todavía más porque los pesos de remuestreo son siempre los mismos para cada *voxel* dentro de un mismo corte, permitiendo al algoritmo calcular dichos pesos sólo una vez por corte.

5.2.5. Warping de la imagen intermedia

La imagen calculada en la fase de composición del algoritmo es una proyección oblicua del volumen. En general, una deformación afín arbitraria debe ser aplicada a la imagen intermedia para producir la correcta imagen final. El segundo factor de la factorización *shear-warp* es el que determina la transformación necesaria (ver sección 4.2.2.3).

La implementación de esta deformación 2D es directa [Heckbert 1986] y [Wolberg 1990]. El algoritmo de este proyecto aplica un filtro bilineal que utiliza una técnica de recolección de un paso, usada en los algoritmos de proyección hacia delante.

5.2.6. Corrección de opacidad

La resolución del volumen determina la frecuencia de muestreo usada por el algoritmo para evaluar la ecuación de composición volumétrica (ver sección 2.3.1). Como consecuencia, la distancia entre muestras a lo largo de un rayo es constante,

siempre que nos movamos dentro del espacio del objeto, pero varía en el de la imagen dependiendo de la transformación de vista actual. Esto conlleva a un problema. Como se menciona en la sección 2.3.1, la opacidad α_i en un punto de muestreo se define como:

$$\alpha_i = 1 - e^{-\phi_i \Delta x}$$

donde Δx es la distancia entre muestras en el sistema de la imagen. Puesto que esta distancia de muestreo cambia con el punto de vista, la opacidad también lo hace, de modo que las opacidades almacenadas en cada *voxel* deben ser corregidas para cada dirección de vista.

Dado un volumen con forma de cubo, relleno con *voxels* idénticos de baja opacidad, si un espectador mira justo al centro de una de las caras del cubo, la distancia entre puntos de muestreo en el espacio de la imagen, a lo largo del rayo de vista, es igual a la longitud de un lado de un *voxel*. Si el espectador gira el volumen 45º alrededor de su eje central, entonces el número de muestras permanece igual, pero la distancia entre ellas en el espacio de la imagen aumenta (ver **Ilustración 19)**. Por ello, la atenuación a lo largo del rayo también se debería incrementar realizando una corrección de la opacidad. Otros algoritmos dirigidos por el volumen también deben corregir la opacidad según el ángulo de vista [Laur & Hanrahan 1991].

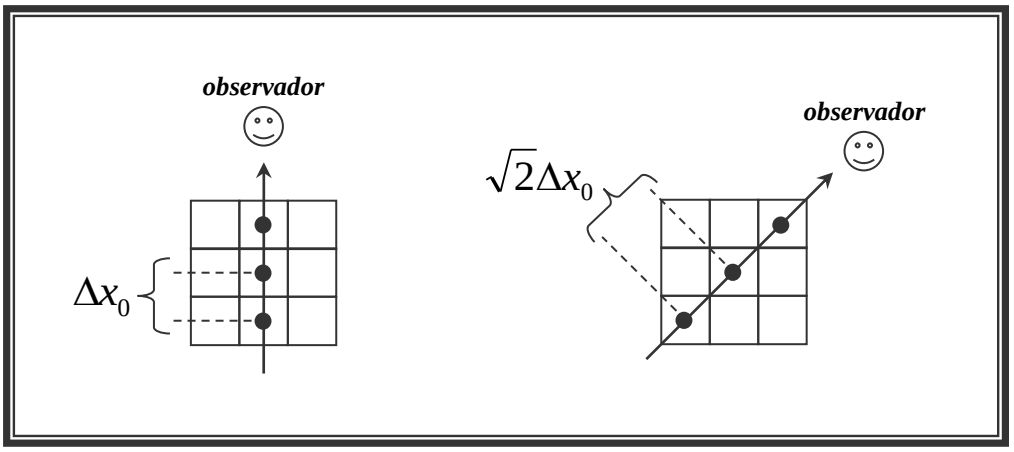


Ilustración 19.- El espacio entre muestras en un rayo de vista depende de su dirección, por lo que la opacidad de los *voxels* debe ser corregida.

Sea Δx_0 la anchura de un *voxel* y, asumiendo que la opacidad calculada y almacenada en un *voxel* ($\alpha_{almacenada}$) utiliza Δx_0 como el espacio entre muestras, ésta queda se puede expresar como:

$$\alpha_{almacenada} = 1 - e^{-\phi \Delta x_0}$$

Luego, para cualquier otro espacio entre muestras, Δx, la opacidad corregida puede calcularse de esta manera:

$$\alpha_{almacenada} = 1 - e^{-\phi\Delta x} = 1 - \left[e^{-\phi\Delta x_0}\right]^{\frac{\Delta x}{\Delta x_0}} = 1 - \left[1 - \alpha_{almacenada}\right]^{\frac{\Delta x}{\Delta x_0}}$$

La opacidad corregida es una función de la opacidad almacenada y la ratio entre los espacios de muestreo (Δx / Δx_0), pero la función es la misma para todos los *voxels*. La **Ilustración 20** muestra una gráfica de la función de corrección de opacidad para diversos valores de la ratio entre los espacios de muestreo. Sin corrección de opacidad, un *voxel* con forma de cubo aparecería un 30% más transparente de lo que debería visto desde un ángulo de 45º, una diferencia visualmente significante.

El algoritmo de renderizado realiza esta corrección de opacidad para cada *voxel* justo antes del remuestreo de dicho *voxel*.

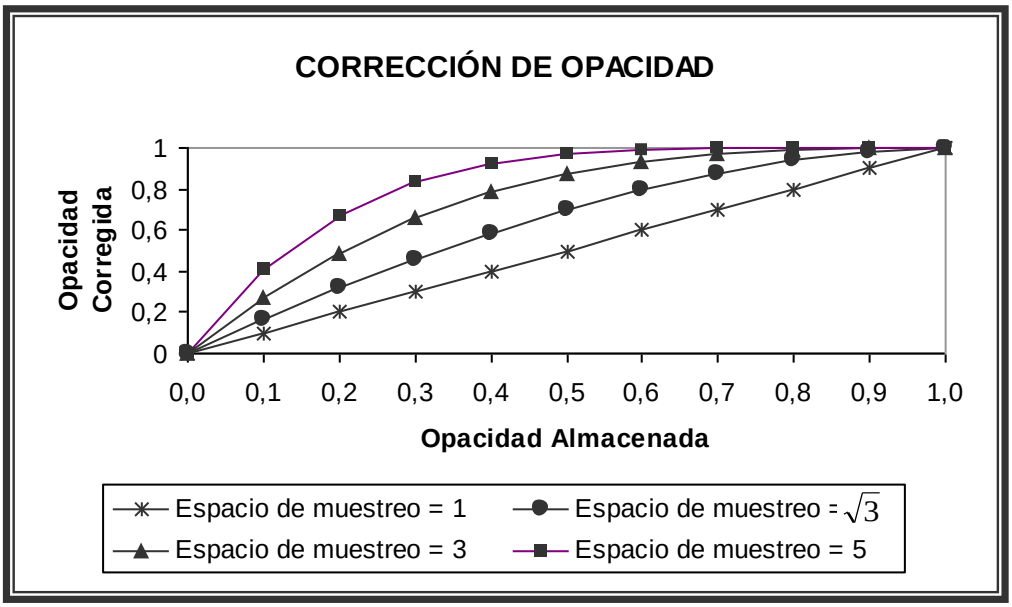


Ilustración 20.- Gráfica de la función de corrección de opacidad en función de la ratio entre los espacios de muestreo (tomando $\Delta x_0 = 1$).

5.2.7. Codificación de normales

Para calcular el color de cada *voxel* se aplica un sistema de codificación-decodificación de normales:

- **Tabla de codificación:** dada una normal, permite obtener un índice en el mapa de colores.
- **Tabla de decodificación:** dado un índice, se obtiene la normal correspondiente.
- **Mapa de colores:** nivel de gris asociado a un índice.

El método de codificación se basa en la utilización de una rejilla, como la mostrada en la **Ilustración 21**. En dicha rejilla, cada celda sombreada corresponde a una

normal y está identificada por su vértice superior izquierdo. El resto de celdas no son utilizadas.

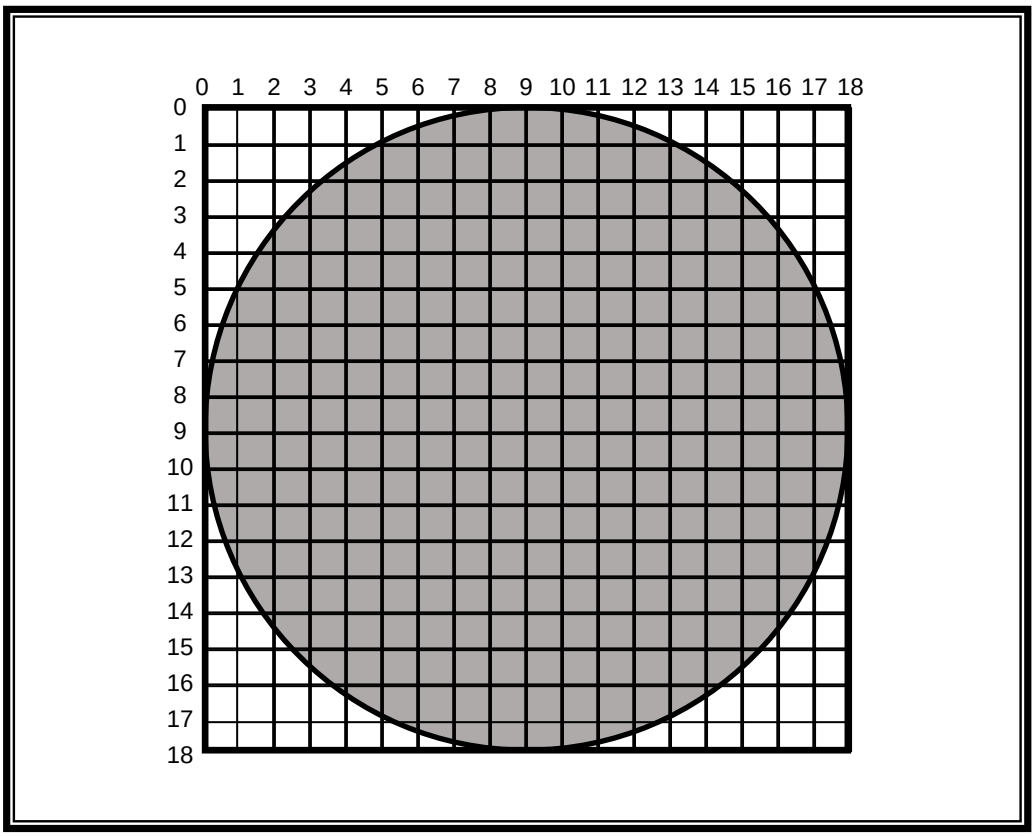


Ilustración 21.- Ejemplo de rejilla de 18x18 celdas.

Para entender el mecanismo de codificación y decodificación, considérese este ejemplo: partiendo de la rejilla de arriba, con centro *C(9, 9)* y, dada la celda de vértice *V(14, 5)*, la normal correspondiente *(N_x, N_y, N_z)* se calcula con la siguiente fórmula:

$$N_x = (V_x - C_x)/C_x = 0{,}56$$
$$N_y = (V_y - C_y)/C_y = -0{,}44$$
$$N_z = \sqrt{1 - (N_x^{\,2} + N_y^{\,2})} = 0{,}70$$

Dicha normal se almacena en la tabla de decodificación, bajo un índice *i*. La tabla de codificación es una matriz que almacena por su parte dicho índice en la componente *[14][9]*.

Por otro lado, el índice *i* es la componente del mapa de colores cuyo nivel de gris se corresponde con la normal codificada.

5.3. IMPLEMENTACIÓN DEL ALGORITMO

En la **Ilustración 22** se muestra la implementación en pseudo-código de las principales rutinas del algoritmo de renderizado con factorización *shear-warp* para proyecciones paralelas.

La codificación de la estructura *run-length* del volumen se detalla en la **Ilustración 33** del ANEXO II de esta memoria, mientras que la correspondiente a la imagen intermedia se encuentra en la **Ilustración 34**. La clase correspondiente a un *pixel* se presenta en la **Ilustración 35** del mencionado anexo.

El procedimiento "Renderizar_Volumen" es la rutina de más alto nivel. Primero llama a "Factorizar_Matriz" para calcular los coeficientes de *shearing* y *warping* de la matriz de transformación de vista. A continuación, se ejecuta "Corregir_Opacidad" para realizar la corrección de opacidad según la orientación actual del objeto. La posterior llamada a "Inicializar_Imagen_Intermedia" asigna los valores iniciales a los pixels de la imagen intermedia. Seguidamente, el bucle de la línea número 6 compone cada corte del volumen sobre la imagen intermedia en orden de delante a atrás, llamando a "Componer_Corte". Finalmente, "Renderizar_Volumen" realiza el *warping* de la imagen intermedia y muestra la imagen final. El código correspondiente a esta rutina se muestra en la **Ilustración 36** del ANEXO II.

La rutina "Componer_Corte" implementa el corazón del algoritmo de renderizado. Comienza inicializando la estructura de datos "voxel_ptr", un vector de punteros a dos líneas adyacentes de *voxels*, codificadas siguiendo la técnica de codificación *run-length*. Aplicando el filtro de remuestreo a estas dos líneas se genera una línea remuestreada de *voxels* (alineada con la imagen intermedia). La rutina "Encontrar_Tramos_de_Voxel" utiliza el vector de punteros a cortes del volumen codificado en forma *run-length* (ver sección 5.2.2) para encontrar las dos primeras líneas de *voxels* de un corte determinado. Éste método aparece codificado en la **Ilustración 37** del ANEXO II.

El bucle más externo itera sobre las líneas de *voxels* del corte. La línea 16 calcula las coordenadas del pixel superior izquierdo de la imagen intermedia a partir del cual se proyecta el actual par de líneas de *voxels* apuntadas por "voxel_ptr". El bucle más interno remuestrea y compone tramos de *voxels* sobre la línea de la imagen intermedia, hasta que alcanza el final de las dos líneas de *voxels*. El cuerpo del bucle discierne tres casos:

- Si el pixel actual de la imagen intermedia es opaco entonces se llama a "Saltar_Tramo_de_Pixels" para ignorar todo el tramo de pixels opacos y los correspondientes *voxels*. Su codificación se detalla en la **Ilustración 38** del ANEXO II.
- En caso contrario, el algoritmo calcula "longitud_tramo", el valor del mínimo número de *voxels* restantes hasta el comienzo del siguiente tramo, en las dos líneas de *voxels*; si los dos tramos actuales son transparentes entonces "Componer_Corte" llama a "Saltar_Tramo_de_Voxels" para ignorar ambos tramos.

- Si la condición del caso anterior no es cierta, se llama a "Procesar_Tramo_de_Voxels" para remuestrear y componer los *voxels* sobre la imagen intermedia.

Una vez que el bucle interno finaliza, los punteros de "voxel_ptr" han avanzado respectivamente una línea de *voxels* y, de esta manera, están listos para calcular la siguiente línea de la imagen intermedia. El bucle más externo continúa procesando líneas, hasta que todos los tramos de *voxels* del corte han sido recorridos.

"Saltar_Tramo_de_Voxels" es la rutina que evita procesar un tramo de *voxels* transparentes. Simplemente llama a "Avanzar_Voxel_Ptr" para avanzar los punteros de las dos líneas actuales de *voxels* y la posición actual en la línea de pixels de la imagen intermedia.

"Saltar_Tramo_de_Pixels" ignora un tramo de *pixels* opacos. El bucle de la línea 40 recorre los *offsets* de los *pixels* opacos hasta el siguiente *pixel* no opaco (el cual debe tener un *offset* igual a cero). El bucle de la línea 44 realiza la "compresión del camino" descrita en la sección 5.2.3. Finalmente, avanza los punteros de las dos líneas actuales de *voxels* llamando a "Avanzar_Voxel_Ptr", haciendo lo mismo con el índice del *pixel* actual "im_x", sumándole la longitud del tramo de *pixels* opacos.

La rutina "Procesar_Tramo_de_Voxels" remuestrea y compone un tramo de *voxels*.

- Primero comprueba si el punto a remuestrear se va a proyectar sobre un *pixel* opaco, en cuyo caso se ha alcanzado el comienzo de un tramo de *pixels* opacos y la rutina retorna a "Componer_Corte".
- En caso contrario, se calcula el valor del *voxel* a remuestrear llamando a "Muestra". Esta función toma cuatro *voxels* (dos de cada línea de *voxels*), realiza la corrección de opacidad y calcula la tonalidad, multiplica los colores y opacidades por los pesos precalculados del filtro de remuestreo, y finalmente, devuelve el color y la opacidad del *voxel* remuestreado. Además, utiliza la información contenida en la estructura *run-length* de *voxels* codificados para evitar el procesamiento de cualquiera de los cuatro *voxels* que sea transparente.
- Después, el procedimiento "Componer_Voxel" compone el *voxel* remuestreado sobre la imagen intermedia.
- Finalmente, el bucle avanza hasta el siguiente punto de muestreo del volumen.

"Componer_Voxel" utiliza el operador "Over" (ver sección 2.3.1) para componer un *voxel* sobre la imagen intermedia. Solamente compone un *voxel* si su opacidad remuestreada excede el umbral de opacidad utilizado al crear la estructura *run-length* del volumen. El procedimiento también crea un nuevo tramo de *pixels* opacos si el *pixel* de la imagen intermedia se vuelve opaco. La **Ilustración 39** del ANEXO II despliega el código del operador *over*.

```
1   Procedimiento Renderizar_Volumen ()
    {
      Factorizar_Matriz (matriz_de_vista);
      Corregir_Opacidad ();
5     Inicializar_Imagen_Intermedia (imagen_intermedia);

      Para k desde 1 hasta Tamaño_Volumen [eje_z]
        Componer_Corte (k);
      fPara

      imagen= Warp (imagen_intermedia);
10    Visualizar (imagen);
    }

    Procedimiento Componer_Corte (k)
    {
      voxel_ptr [1..2]= Encontrar_Tramos_de_Voxel (k,2);
15    Para v desde 1 hasta Tamaño_Volumen [eje_y]
        (im_x, im_y)= (Trasladar_en_X (k), Trasladar_en_Y (k) + v);

        Mientras (No (Final_de_Línea (voxel_ptr)))
          Si (Es_Opaco (imagen_intermedia [im_x][im_y]))
              Saltar_Tramo_de_Pixels (im_x, im_y, voxel_ptr);
20        Sino
              longitud_tramo= Min (voxel_ptr [1]. longitud_tramo, voxel_ptr [2].longitud_tramo);
              Si ( (Es_Transparente (voxel_ptr [1])) Y (Es_Transparente (voxel_ptr [2])) )
                Saltar_Tramo_de_Voxels (im_x, voxel_ptr, longitud_tramo);
25            Sino
                Procesar_Tramo_de_Voxels (im_x, im_y, voxel_ptr, longitud_tramo);
              fSi
          fSi
        fMientras

30    fPara
    }

    Procedimiento Saltar_Tramo_de_Voxels (im_x, voxel_ptr, longitud_tramo)
    {
      Avanzar_Voxel_Ptr (voxel_ptr, longitud_tramo);
35    im_x= im_x + longitud_tramo;
    }
```

Ilustración 22.- Implementación en pseudo-código del algoritmo de renderizado con factorización *shear-warp* para proyecciones paralelas.

```
Procedimiento Saltar_Tramo_de_Pixels (im_x, im_y, voxel_ptr)
{
  longitud_tramo= 0;
  Mientras (imagen_intermedia [im_x + longitud_tramo][im_y].offset > 0)
    longitud_tramo= longitud_tramo + imagen_intermedia [im_x + longitud_tramo][im_y].offset;
  fMientras

  longitud_camino= 0;

  Mientras (imagen_intermedia [im_x + longitud_camino][im_y].offset > 0)
    offset= imagen_intermedia [im_x + longitud_camino][im_y].offset;
    imagen_intermedia [im_x + longitud_camino][im_y].offset= longitud_tramo – longitud_camino;
    longitud_camino= longitud_camino + offset;
  fMientras

  Avanzar_Voxel_Ptr (voxel_ptr, longitud_tramo);
  im_x= im_x + longitud_tramo;
}

Procedimiento Procesar_Tramo_de_Voxels (im_x, im_y, voxel_ptr, longitud_tramo)
{
  Para i desde 1 hasta longitud_tramo
    Si (Es_Opaco (imagen_intermedia [im_x][im_y]))
      Retornar;
    fSi

    (color, alfa)= Muestra (voxel_ptr);
    Componer_Voxel (im_x, im_y, color, alfa);
    Avanzar_Voxel_Ptr (voxel_ptr,1);
    im_x= im_x + 1;
  fPara
}

Procedimiento Componer_Voxel (im_x, im_y, color, alfa)
{
  Si (No (Es_Transparente (alfa)))
    imagen_intermedia [im_x][im_y]= imagen_intermedia [im_x][im_y] Over (color, alfa);
    Si (Es_Opaco (imagen_intermedia [im_x][im_y]))
      imagen_intermedia [im_x][im_y].offset= 1;
    fSi
  fSi
}
```

Ilustración 22 (continuación).- Implementación en pseudo-código del algoritmo de renderizado con factorización *shear-warp* para proyecciones paralelas.

5.4. COMPLEJIDAD TEMPORAL DEL ALGORITMO

El objetivo final de este proyecto es mejorar el rendimiento observado en los algoritmos tradicionales de renderizado de volumen.

Si se considera como información de entrada un volumen de tamaño *n* en las tres dimensiones, entonces cualquier algoritmo de renderizado de fuerza bruta tiene una complejidad asintótica de $O(n^3)$. El objetivo de este proyecto es disminuir ese orden.

Para poder estimar la complejidad temporal de este algoritmo, es necesario tener en cuenta los diferentes retardos en cada una de sus fases críticas:

- **Recorrido de *voxels*:** El coste de recorrer los *voxels* del volumen viene condicionado por la cantidad de tramos de *voxels* a recorrer, la cual depende arbitrariamente del objeto a renderizar.

  Estimando una media de c_v tramos de *voxels* en cada una de las n^2 líneas de escaneo, la penalización temporal es de $O(c_v n^2)$.

- **Recorrido de *pixels*:** El tiempo necesario para visitar los *pixels* de la imagen intermedia depende de la cantidad de tramos de *pixels* creados. Como se detalla en la sección 5.2.3, la codificación de esos tramos también debe ser mantenida con la operación de "compresión del camino", a la que hay que sumar la operación de búsqueda del siguiente *pixel* no opaco.

  Dada una imagen intermedia con las siguientes características:

  - Número de *pixels*: n
  - Número de tramos creados: n_c
  - Número de operaciones de búsqueda del *pixel* opaco: n_b

  El coste total para recorrer toda la imagen intermedia es $O(c_v n^2 log_{1+nb/nc}\ (c_v))$ [Cormen 1990], penalización a tener en cuenta en la complejidad del algoritmo.

- **Remuestreo y composición:** El número de remuestreos y composiciones de *voxels* es de orden $O(dn^2)$, siendo *d* la media de superficies atravesadas hasta que un *pixel* se convierte en opaco.

- ***Warping* 2D:** La imagen intermedia requiere un procesado que afecta a la totalidad de sus *pixels*, lo que supone un coste de orden $O(n^2)$.

De esta forma, la complejidad temporal del algoritmo implementado en este proyecto es inferior a la de un algoritmo de renderizado clásico de fuerza bruta ($O(n^3)$):

$$O\left[n^2 \left(d + c_v \log_{1+\frac{n_b}{n_c}} (c_v) \right) \right]$$

6 RESULTADOS

Una vez implementado el algoritmo, se procede a evaluar los resultados de su ejecución. A continuación, se describen las pruebas realizadas para ese fin.

6.1. DISEÑO DE LAS PRUEBAS

6.1.1. Objetivo de las pruebas

El conjunto de pruebas se ha diseñado para tener una perspectiva del comportamiento del algoritmo desde una doble vertiente:

- Comprobar que el algoritmo cumple con los requisitos de rendimiento y calidad de imagen impuestos en la sección 5.1.
- Evaluar el impacto que tiene para el algoritmo la alteración de los siguientes parámetros:
  - Tamaño del volumen de datos de entrada.
  - Porcentaje de *voxels* transparentes del volumen.
  - Tipo de procesador en el que el algoritmo es ejecutado.

Para llevar a cabo estas pruebas, se utilizan dos tipos de volúmenes de datos:

- **Reales**: Conjunto de *voxels* procedentes de un objeto o individuo real. Su uso está destinado a evaluar la calidad de la imagen renderizada.
- **Sintéticos**: conjunto de *voxels* generados al azar para aumentar la frecuencia de aparición de voxels no transparentes y, de esta manera, simular el caso más desfavorable, aunque prácticamente infrecuente, a la hora de evaluar el rendimiento del algoritmo.

6.1.2. Descripción de las pruebas

A continuación, se detallan las pruebas a realizar para evaluar los diferentes objetivos mencionados.

PRUEBA Nº 1	
Descripción	Medida del coste temporal del algoritmo sobre volúmenes de datos de diferentes tamaños, todos ellos con el 100% de los *voxels* no transparentes.
Datos de entrada	Volúmenes de datos sintéticos obtenidos de forma aleatoria: - Volumen 1: 128 x 128 x 64 *voxels*. - Volumen 2: 256 x 256 x 64 *voxels*. - Volumen 3: 256 x 256 x 128 *voxels*. - Volumen 4: 256 x 256 x 256 *voxels*. - Volumen 5: 512 x 512 x 128 *voxels*.
Metodología	Ejecución del algoritmo con cada uno de los volúmenes generados y medición del tiempo invertido en obtener la imagen final. La ejecución se realiza en un mismo ordenador, prestando atención a la diferencia de tiempos entre los diferentes volúmenes, más que el tiempo individual de cada renderizado. Para evitar casos particulares, cada volumen se genera aleatoriamente 3 veces, obteniendo como coste temporal la media de las 3 ejecuciones para cada tamaño.

Tabla 1.- Prueba de medición del coste temporal sobre volúmenes de diferentes tamaños.

PRUEBA Nº 2	
Descripción	Medida del coste temporal del algoritmo sobre un volumen de datos de un mismo tamaño, con diferente número de *voxels* transparentes.
Datos de entrada	Volumen de datos sintéticos obtenidos de forma aleatoria de tamaño 256 x 256 x 256 *voxels*.
Metodología	Ejecución del algoritmo con el volumen generado y medición del tiempo empleado, considerando transparentes el 0%, 25%, 50% y 75% de los *voxels*, respectivamente. Para evitar casos particulares, el volumen se genera aleatoriamente 3 veces, obteniendo como coste temporal la media de las 3 ejecuciones para cada porcentaje de *voxels* no transparentes.

Tabla 2.- Prueba de medición del coste temporal sobre volúmenes con diferente número de *voxels* transparentes.

PRUEBA Nº 3	
Descripción	Medida del coste temporal del algoritmo con diversos volúmenes de datos, en procesadores de propósito general con alta diferencia de prestaciones.
Datos de entrada	Volumen de datos sintéticos obtenidos de forma aleatoria con el 100% de los *voxels* no transparentes. - Volumen 1: 128 x 128 x 64 *voxels*. - Volumen 2: 256 x 256 x 64 *voxels*. - Volumen 3: 256 x 256 x 128 *voxels*. - Volumen 4: 256 x 256 x 256 *voxels*. - Volumen 5: 512 x 512 x 128 *voxels*.
Metodología	Ejecución del algoritmo y medición de tiempos de renderizado de los volúmenes anteriores en los siguientes procesadores: - Procesador 1: Intel Core i7-7700HQ 2.80 GHz; RAM: 16GB - Procesador 2: Intel Core 2 Quad Q8200 2.33 GHz; RAM: 4 GB

Tabla 3.- Prueba de medición del coste temporal sobre diferentes procesadores de propósito general.

PRUEBA Nº 4	
Descripción	Evaluación de la calidad de la imagen obtenida por el algoritmo con estudios volumétricos reales.
Datos de entrada	Diversos estudios volumétricos reales.
Metodología	Ejecución del algoritmo y evaluación visual de la imagen generada.

Tabla 4.- Prueba de evaluación de la calidad de la imagen generada.

6.2. RESULTADOS EN RENDIMIENTO

A continuación, se muestran los resultados de rendimiento, correspondientes a las pruebas del nº 1 al nº 3.

6.2.1. Resultados de la prueba nº 1

La **Ilustración 23** demuestra que, aunque evidentemente el coste temporal del algoritmo aumenta con el tamaño del volumen, éste lo hace atendiendo a un crecimiento claramente por debajo del orden $O(n^3)$, como se deduce de la expresión obtenida en el apartado 5.4.

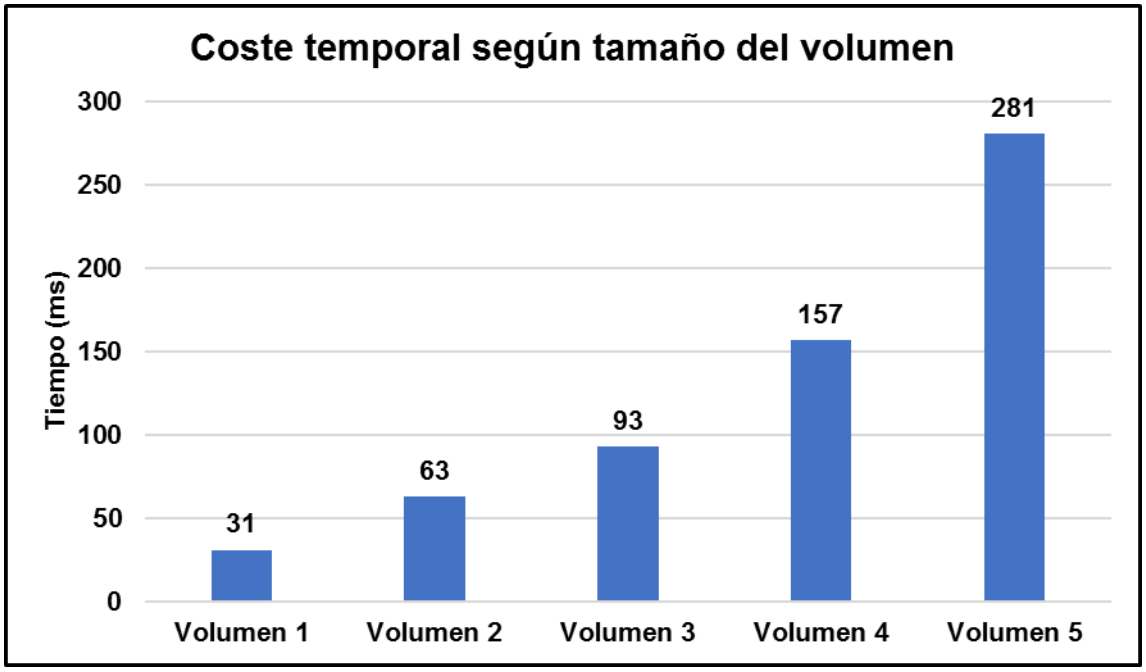


Ilustración 23.- Resultados de la prueba nº 1.

6.2.2. Resultados de la prueba nº 2

En esta prueba se puede observar que el aumento de *voxels* transparentes no mejora obligatoriamente el tiempo de renderizado (ver **Ilustración 24**).

De hecho, el algoritmo es más eficiente cuando la cantidad de *voxels* de un mismo tipo (transparentes o no transparentes) es mayoritariamente superior a la cantidad del otro tipo.

En esta prueba se utiliza deliberadamente un volumen de datos sintéticos generados de forma aleatoria. De esta manera se asegura de que el volumen de datos no contiene *voxels* del mismo tipo, dando lugar a la generación de tramos de *voxels* (y *pixels*) de distinta naturaleza, arbitrariamente localizados dentro del volumen (y de la imagen intermedia), afectando directamente al coste temporal del renderizado.

Esta prueba demuestra que el algoritmo es más rápido cuanto más homogéneo es el conjunto de *voxels*.

La explicación de este hecho se fundamenta en la expresión calculada en el apartado 5.4: el coste temporal del algoritmo viene penalizado por el número, tanto de tramos de *voxels* (transparentes y no transparentes), como de tramos de *pixels* (opacos y no opacos).

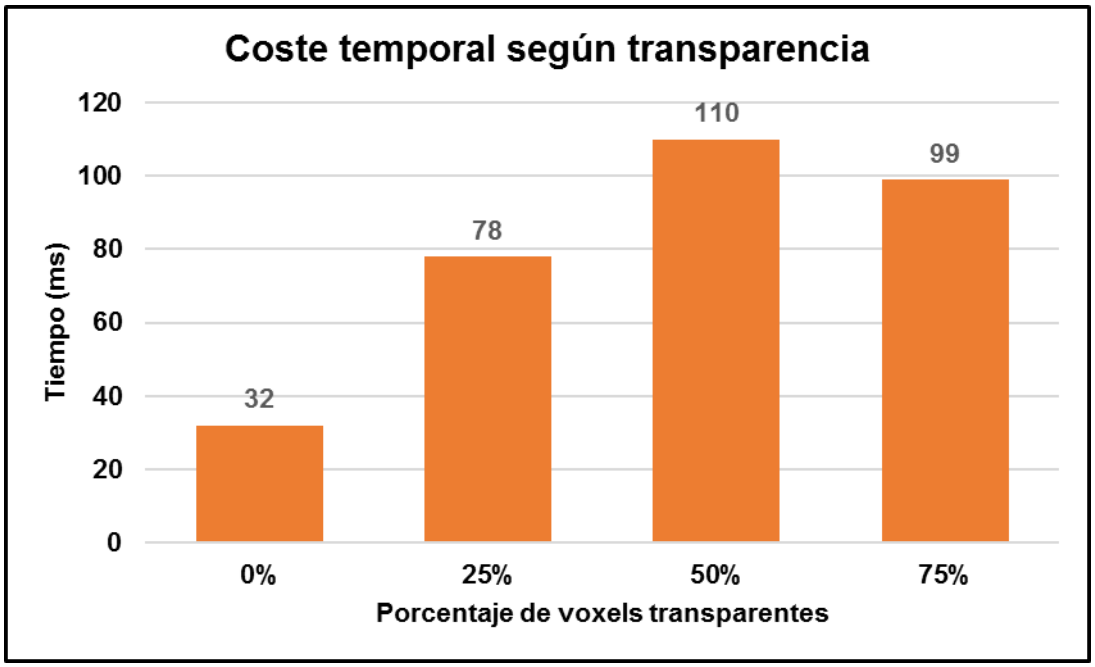


Ilustración 24.- Resultados de la prueba nº 2.

6.2.3. Resultados de la prueba nº 3

La tercera prueba aporta una doble información (ver **Ilustración 25**).

- Por una parte demuestra el cumplimiento de uno de los objetivos principales de este proyecto: el coste temporal de nuestro algoritmo de renderizado para volúmenes de datos estándar (256^3 *voxels*), ejecutado en máquinas de propósito general, es inferior a un segundo.

  Más aún, incluso duplicando ese tamaño (volumen 5), el coste sigue estando por debajo del tiempo exigido en este proyecto.

- Por otra parte, ante la gran diferencia entre ambos procesadores y, por lo tanto, en sus tiempos de ejecución, esta prueba demuestra que la máquina donde se ejecute el algoritmo no tiene porqué ser de alto rendimiento, dentro de su propósito general.

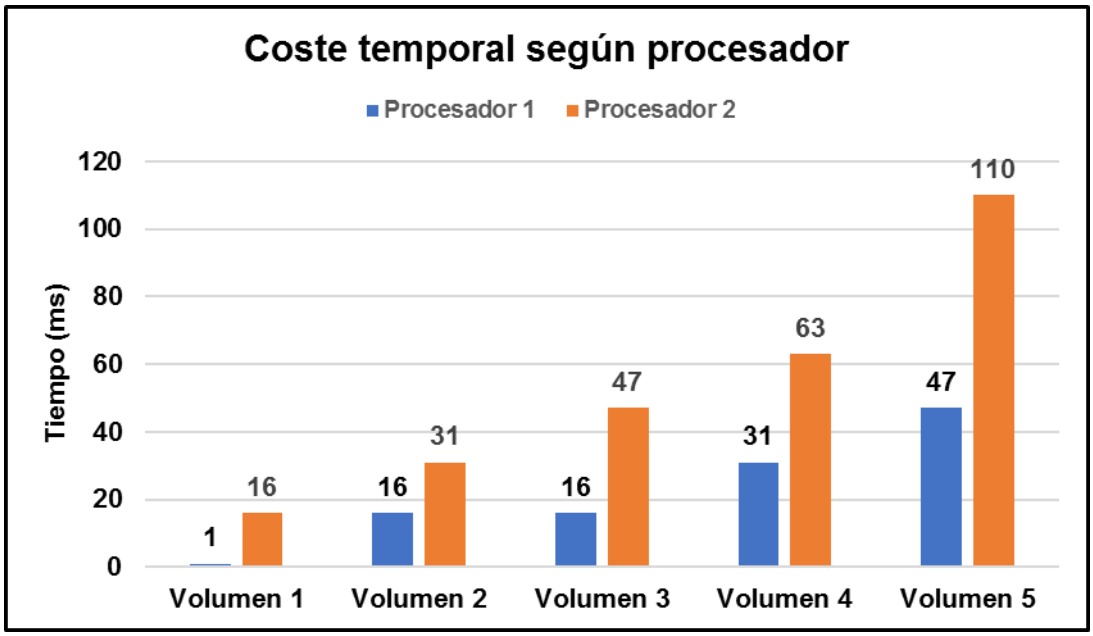


Ilustración 25.- Resultados de la prueba nº 3.

6.3. RESULTADOS EN CALIDAD DE IMAGEN

Los resultados en calidad de imagen se detallan en la prueba nº 4.

6.3.1. Resultados de la prueba nº 4

Los resultados gráficos de la cuarta prueba se muestran en el ANEXO I: GALERÍA DE IMÁGENES.

La coherencia espacial de los datos tiene la cualidad de acelerar el renderizado de volumen sin alterar la calidad de la imagen. Por ello, una de las características de este algoritmo es que mejora el rendimiento, con respecto a los algoritmos tradicionales, sin degradación de la imagen final.

Sin embargo, la nitidez de la imagen generada por este algoritmo sí se ve afectada por otros factores: la **función de iluminación** y los **filtros de remuestreo**.

En primer lugar, la función de iluminación utilizada influye directamente en la calidad del tono de los *voxels*. Así, para la implementación del algoritmo se ha hecho uso de una técnica que, aunque permite la iluminación interactiva del objeto a renderizar, ataca levemente a la calidad del 3D obtenido [Glassner 1990]. Esta técnica se basa en la codificación de las normales del objeto mediante tablas de búsqueda y mapas de colores. De esta forma, para obtener la tonalidad de un *voxel* basta con consultar el valor de su color en el mapa de colores, el cual se genera cada vez que cambia el ángulo de visión, haciendo uso de las tablas de búsqueda precalculadas. Estas tablas contienen el valor de todas las normales y son creadas justo antes del comienzo del algoritmo, permaneciendo invariables, independientemente del ángulo de visión, hasta el final de éste.

Este método, aunque eficiente, produce el conocido efecto de *banding* (bandas de regiones con el mismo valor de la normal) debido a la cuantización de las normales (ver **Ilustración 26**).

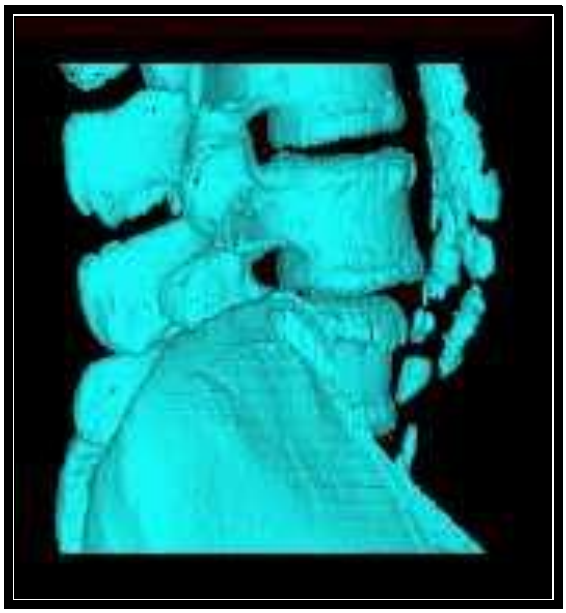


Ilustración 26.- Ejemplo del efecto *banding*.

En segundo lugar, a pesar de la similitud en calidad con las imágenes generadas por un algoritmo de trazado de rayos [Lacroute 1995], la técnica *shear-warp* impone severas limitaciones al filtro utilizado para remuestrear el volumen, y estas limitaciones pueden producir degradación en la imagen.

La primera limitación es que esta técnica requiere dos pasos de remuestreo: primero el algoritmo remuestrea los cortes del volumen (*shearing*), y después hace lo mismo con la imagen intermedia (*warping*), para producir la imagen final. En la literatura referente al renderizado de volumen[12] se refleja que múltiples pasos en el remuestreo pueden producir el efecto de *blurring* (imagen borrosa) y la pérdida de detalle.

La segunda limitación es que el remuestreo del volumen se realiza mediante un filtro de reconstrucción bilineal en lugar de trilineal. Se podría utilizar un filtro de mayor calidad para reconstruir cada corte del volumen, pero nuestro algoritmo no interpola datos entre cortes. Como consecuencia, el fenómeno de *aliasing* (efecto de escalonamiento de *pixels* y bordes dentados) puede aparecer si las funciones de opacidad y color del volumen contienen altas frecuencias en la dimensión perpendicular a los cortes (aunque, si dichas frecuencias exceden la frecuencia de Nyquist, este fenómeno es inevitable). La **Ilustración 27** refleja este fenómeno.

[12] Como por ejemplo en [Lacroute 1995].

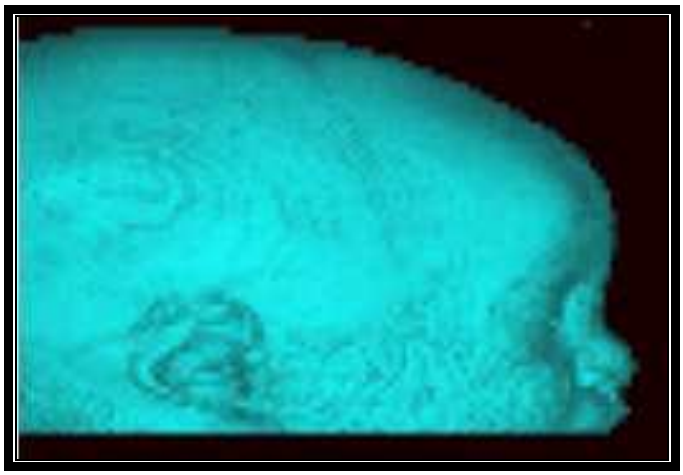


Ilustración 27.- Ejemplo de efecto *aliasing*.

De todos modos, si el volumen original está limitado en banda y las funciones de color y opacidad no contienen altas frecuencias, el filtro bilineal produce buenos resultados (ver **Ilustración 28**).

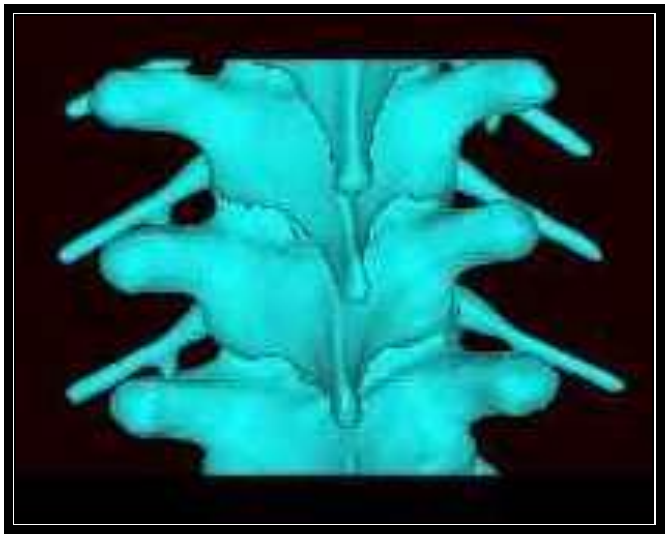


Ilustración 28.- Ejemplo de imagen generada por el algoritmo de renderizado.

La tercera, y última, limitación impuesta por el filtro de remuestreo tiene su origen en la resolución del volumen de entrada. Ésta fija la frecuencia de muestreo durante el renderizado independientemente de la resolución de la imagen final. Así, si la resolución de ésta es mucho mayor que la del volumen, entonces el resultado es una imagen 3D muy pixelada o emborronada, dependiendo del filtro utilizado para el *warping* 2D. Este efecto podría solucionarse muestreando (a mayor resolución) cada corte del volumen en el momento del *shearing*. Sin embargo, este muestreo afectaría al rendimiento del algoritmo de forma significativa. Además, es imposible realizar este remuestreo en la dimensión perpendicular a los cortes del volumen sin cambios significativos en el algoritmo. Una mejor solución sería preescalar el volumen a una resolución mayor antes del renderizado. Puesto que sólo se realizaría una vez, podría

incluso utilizarse un filtro de reescalado de alta calidad. La única desventaja vendría dada por un mayor requerimiento de memoria.

De la misma manera, el volumen ha de ser preescalado si la resolución en los tres ejes no es la misma. Si no se hiciese así, la frecuencia de muestreo en el espacio de la imagen no sería uniforme causando, bien posibles efectos de *aliasing*, o bien una excesiva frecuencia de muestreo en los *voxels*, reflejada en ciertas regiones de la imagen final. Los algoritmos de trazado de rayos no tienen este problema, porque la localización de las muestras a lo largo del rayo de vista puede ser distribuida de forma uniforme, independientemente de la resolución del volumen.

El siguiente capítulo ofrece algunas pautas a seguir para poder corregir las desventajas del algoritmo de renderizado, expuestas en esta sección.

7 CONCLUSIONES Y TRABAJOS FUTUROS

En este capítulo se realiza una reflexión del trabajo realizado, exponiendo las conclusiones obtenidas tras la implementación del algoritmo, las posibles acciones a llevar a cabo para mejorar sus puntos débiles, y una opinión personal.

7.1. CONCLUSIONES

Tras la implementación del algoritmo, objetivo final de este proyecto, se puede deducir que, ciertamente, la técnica *shear-warp* permite desarrollar un algoritmo eficiente para el renderizado de volúmenes de datos de tamaño estándar con proyección paralela.

La simplicidad de los cálculos requeridos por esta técnica, complementada con los métodos de aceleración basados en la terminación temprana de rayos y el uso de estructuras espaciales que explotan la coherencia de los datos entre *voxels* y entre *pixels*, convierten al algoritmo en un candidato ideal para el manejo interactivo de objetos en tres dimensiones.

Durante el desarrollo de este proyecto se han estudiado diferentes opciones de implementación para aumentar la simplicidad del algoritmo y/o disminuir las penalizaciones en cuanto a tiempo de cálculo. Así, acciones como la supresión del uso de las estructuras espaciales de datos en el volumen, o la cancelación de la operación de "compresión de camino" en la imagen, se han llevado a cabo con resultados poco alentadores en cuanto a rendimiento. Por ello, una conclusión resaltable es que la eficiencia del algoritmo es alcanzable si, y sólo si, se implementa la técnica *shear-warp* junto con todos los métodos de aceleración mencionados en este proyecto.

Por su gestión de la memoria, se trata de un algoritmo bastante independiente de la política de gestión del sistema informático donde se ejecute, dado que recorre el volumen en orden de almacenamiento y libera la memoria de los *voxels* transparentes.

El tipo de sistemas informáticos donde puede ser ejecutado abarca un amplio rango, ya que el algoritmo funciona perfectamente en ordenadores de propósito general, razón que también lo hace económicamente accesible a cualquier usuario.

Sin embargo, las desventajas aparecen al analizar de forma minuciosa la calidad de la imagen. Aunque los resultados obtenidos son más que aceptables para destinar su aplicación a campos como el diagnóstico en medicina, arquitectura y otras disciplinas que requieran un renderizado de volumen casi inmediato; para otras aplicaciones más exigentes, la nitidez de la imagen no es suficiente, ya que ésta se ve afectada por la función de iluminación y el filtro de remuestreo utilizados.

Para mejorar visualmente el renderizado, se han implementado mejoras en el filtro de remuestreo, utilizando un filtro de interpolación trilineal básico, consiguiendo

evitar en gran medida el efecto *aliasing*, pero comprometiendo seriamente la eficiencia del algoritmo.

Por otra parte, esta técnica requiere la previa clasificación del volumen antes de cualquier renderizado, lo que exige la reconstrucción de las estructuras de codificación del volumen cada vez que cambia la función de segmentación. Este hecho hace impracticable el uso de este algoritmo para procesos médicos, como la visualización simultánea de diferentes tejidos anatómicos o la detección en tiempo real de cuerpos extraños.

7.2. TRABAJOS FUTUROS

Este algoritmo tiene la restricción de uso para proyecciones paralelas. Una mejora importante sería su adecuación a proyecciones en perspectiva, tal y como se propone en [Lacroute 1995], con rendimientos muy aceptables, aunque algunas limitaciones en calidad de imagen.

La incorporación de un sistema de clasificación inmediata del volumen, sin necesidad de recalcular las estructuras volumétricas, sería una estrategia idónea complementaria para el uso interactivo de este algoritmo. En [Lacroute 1995] se propone un algoritmo de renderizado basado en estructuras de datos de tipo min-max *octree*, que permite al usuario ajustar interactivamente las funciones clasificación de volumen y ver los resultados inmediatamente.

En busca de una imagen de más alta calidad, la mejora del remuestreo del volumen, usando por ejemplo un sistema de interpolación trilineal optimizado, evitaría la aparición de *aliasing* en la imagen. En [Sweeney & Mueller 2002] se desarrolla una versión revisada del algoritmo de este proyecto, que mejora el efecto de *aliasing* entre cortes.

Por último, un sistema de cuantización de normales más preciso, aunque más costoso, evitaría el conocido efecto de bandas.

7.3. OPINIÓN PERSONAL

La elaboración de este proyecto ha supuesto para mi persona una experiencia enriquecedora a nivel intelectual, porque me ha permitido, no sólo aplicar gran parte de los conocimientos aprendidos durante mi etapa de formación, sino también adentrarme en el mundo de la investigación para ampliarlos y comprenderlos desde diferentes puntos de vista.

Desde un enfoque profesional, gracias al desarrollo de este trabajo, he descubierto nuevas tecnologías informáticas existentes en la actualidad, desconocidas por mí, y entender su protagonismo en campos como el diagnóstico médico, disciplina a la que va orientado este proyecto.

A nivel personal, esta experiencia me ha brindado una oportunidad para interaccionar con profesionales de otras disciplinas y, aunque ha sido una prueba de esfuerzo y dedicación dentro de una temática concreta, me ha abierto las puertas para saber y poder afrontar nuevos retos informáticos en un futuro.

8 ANEXO I: GALERÍA DE IMÁGENES

A continuación, se muestran ejemplos con los resultados de renderizado obtenidos por nuestro algoritmo:

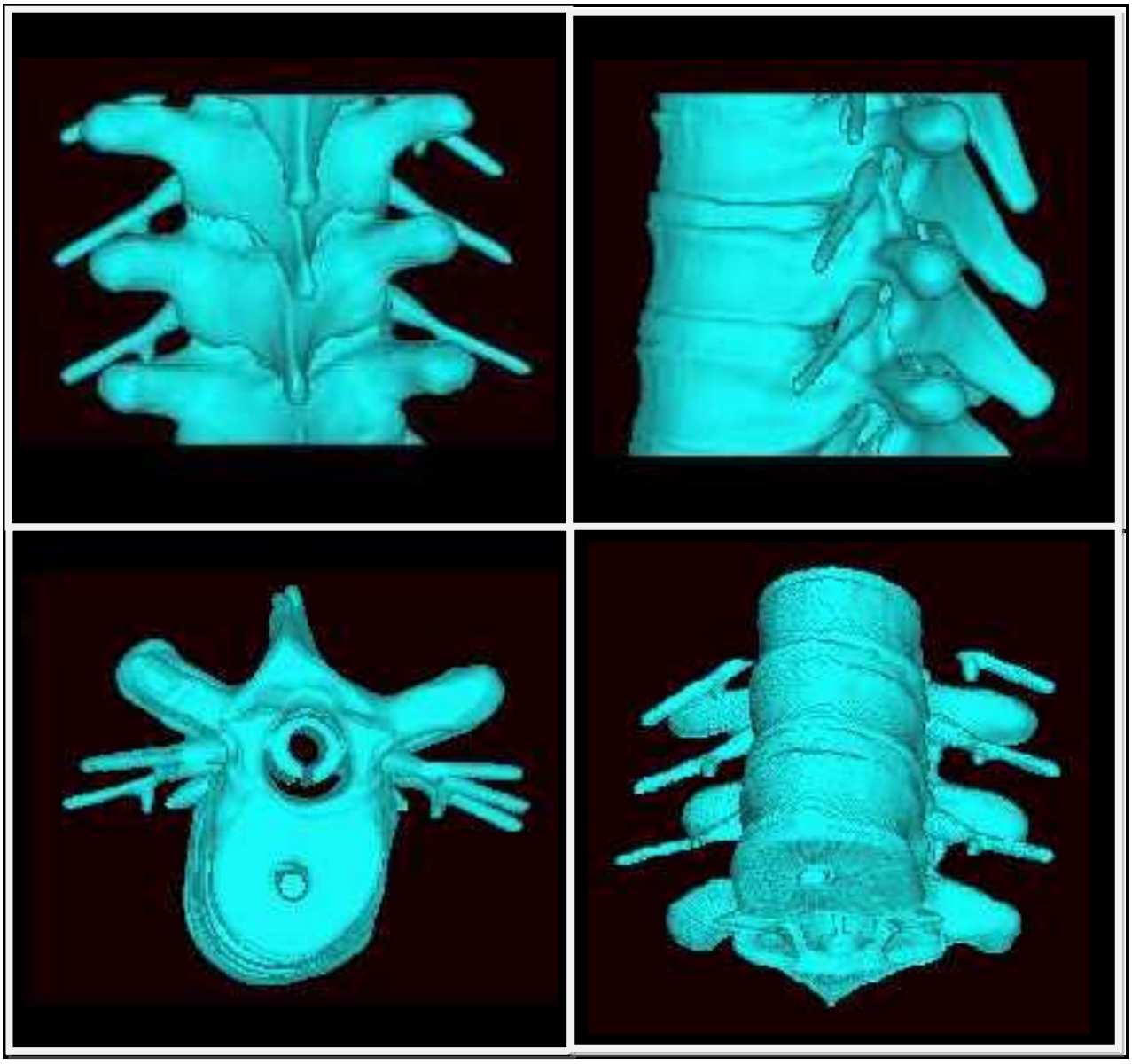


Ilustración 29.- Ejemplo de imágenes generadas por el algoritmo de renderizado.

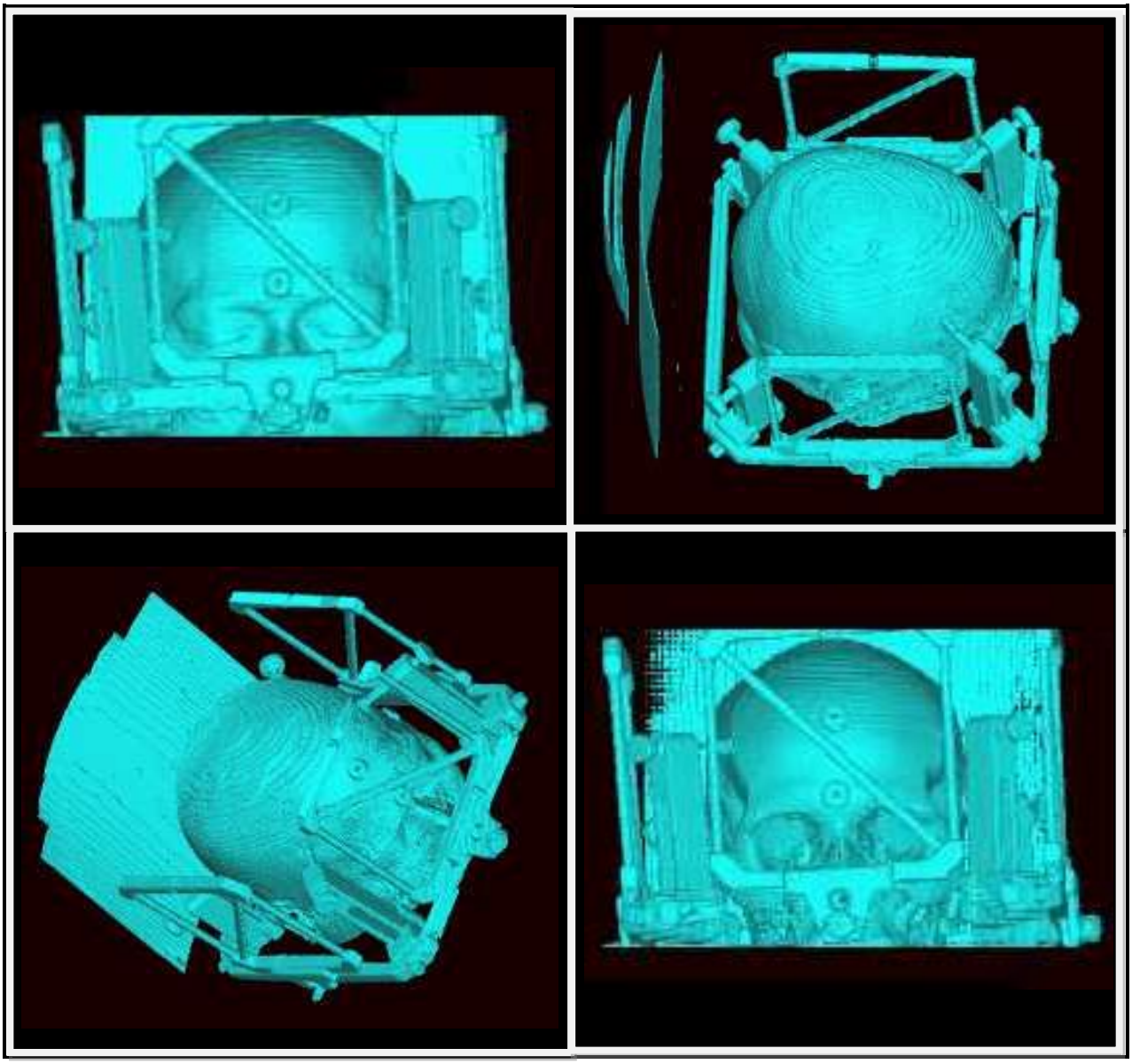


Ilustración 30.- Ejemplo de imágenes generadas por el algoritmo de renderizado.

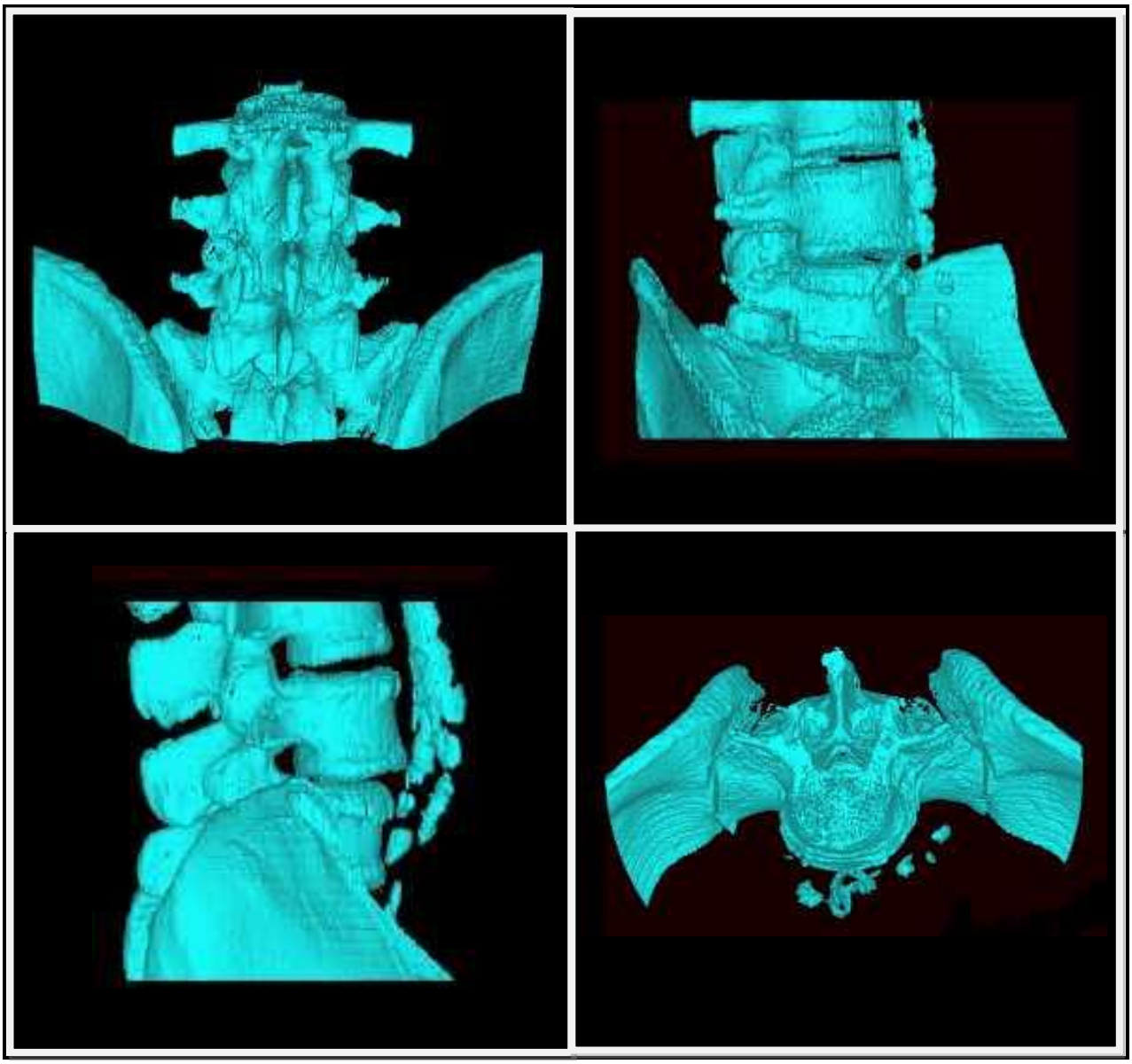


Ilustración 31.- Ejemplo de imágenes generadas por el algoritmo de renderizado.

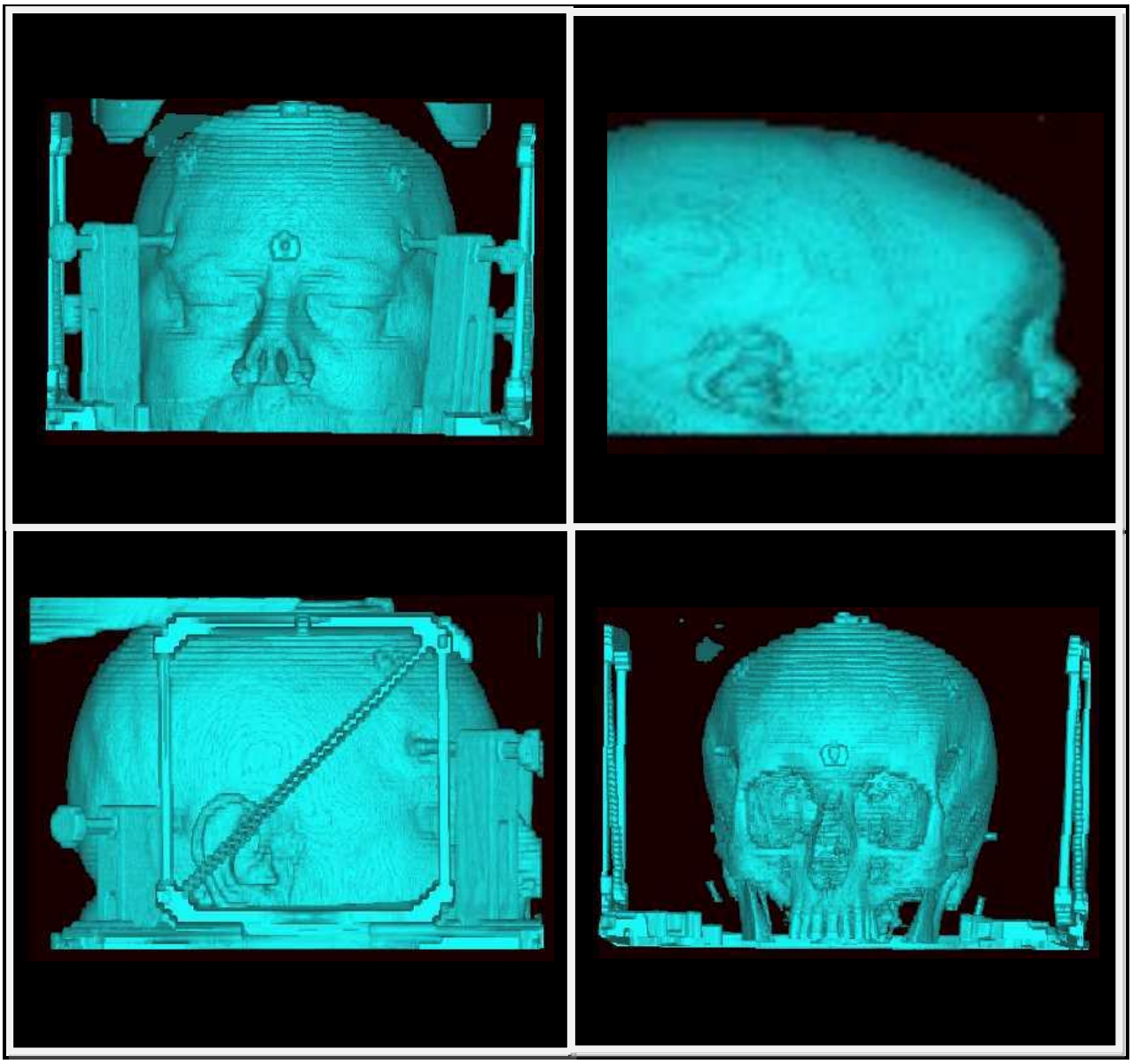


Ilustración 32.- Ejemplo de imágenes generadas por el algoritmo de renderizado.

9 ANEXO II: CÓDIGO DEL ALGORITMO

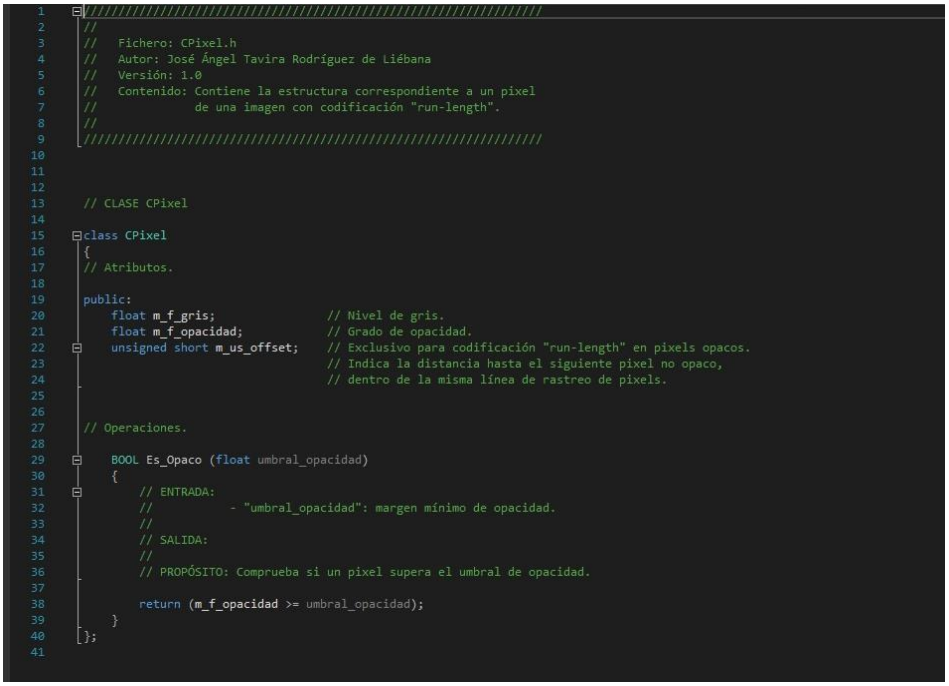
A continuación, se muestran fragmentos del algoritmo codificado en lenguaje C++.

```
// Voxel.

typedef struct
{
    unsigned char gris;     // Nivel de gris en el volumen original.
    short indice_normal;    // Índice de la normal asociada al voxel.
} Voxel;


// Punteros de corte.

typedef struct
{
    unsigned int P,    // Puntero a la primera longitud de tramo de un corte.
                 Q;    // Puntero al primer voxel no transparente de un corte.
} Punteros_de_Corte;


// Vector de punteros de corte.

typedef Punteros_de_Corte* Vector_Punteros_de_Corte;


// Vector de longitudes de tramo.

typedef short* Vector_Longitudes_de_Tramo;


// Vector comprimido de voxels no transparentes.

typedef Voxel* Vector_Comprimido_de_Voxels;


// Estructura de codificación "run-length" del volumen. Recoge los tipos ya definidos anteriormente
// además del número total de tramos.

typedef struct
{
    Vector_Longitudes_de_Tramo tramos;
    unsigned int numero_tramos;
    Vector_Punteros_de_Corte  punteros;
    Vector_Comprimido_de_Voxels voxels;

} Codificacion_Volumen;


// Estructura replicada del Volumen codificado para cada uno de los ejes
// principales de vista.

typedef Codificacion_Volumen Codificacion_Replicada_Volumen [3];
```

Ilustración 33.- Codificación de las estructuras de datos del volumen.

```
//////////////////////////////////////////////////////////////////////
//
//   Fichero: CImagen_Intermedia.h
//   Autor: José Ángel Tavira Rodríguez de Liébana
//   Versión: 1.0
//   Contenido: Contiene la estructura para la imagen intermedia.
//
//////////////////////////////////////////////////////////////////////


// Bibliotecas incluidas.

#include "CPixel.h"


// CLASE CImagen_Intermedia

class CImagen_Intermedia
{
// Atributos.

protected:

    // Dimensiones de la imagen en pixels..

    unsigned short m_us_tamano_x,   // Ancho.
                   m_us_tamano_y;   // Alto.

    unsigned int m_ui_tamano;       // Ancho x Alto.
```

Ilustración 34.- Codificación de la imagen intermedia.

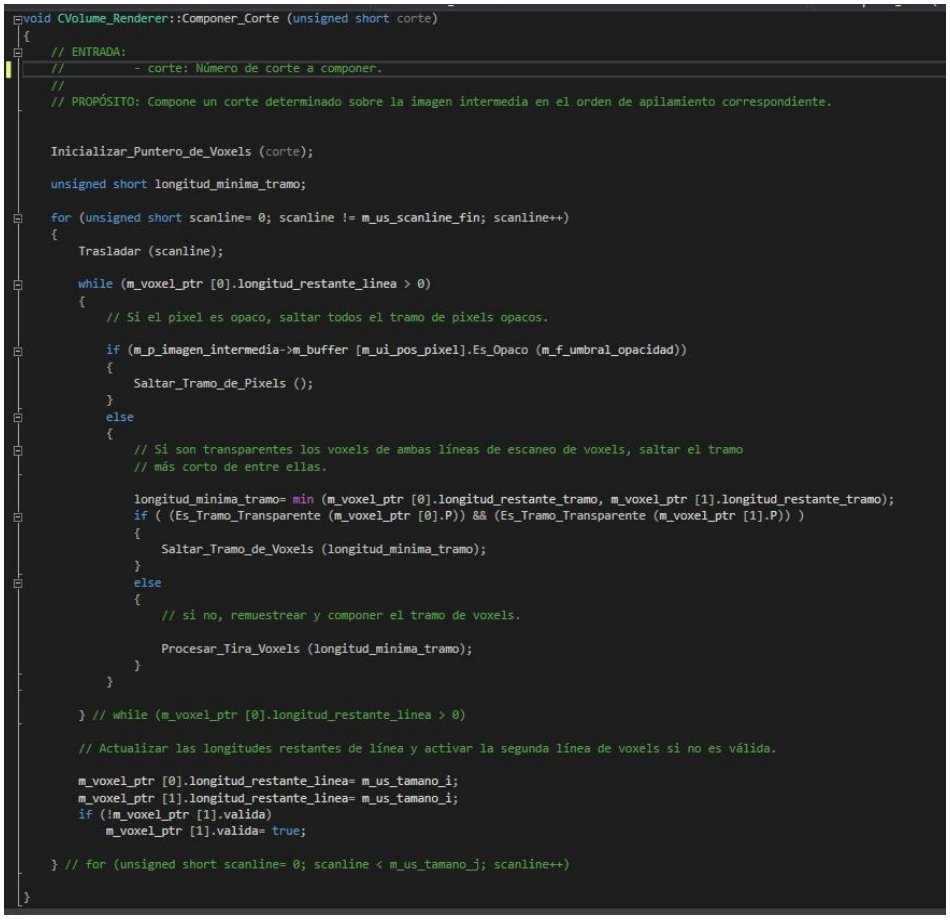


```
////////////////////////////////////////////////////////////////
//
//   Fichero: CPixel.h
//   Autor: José Ángel Tavira Rodríguez de Liébana
//   Versión: 1.0
//   Contenido: Contiene la estructura correspondiente a un pixel
//              de una imagen con codificación "run-length".
//
////////////////////////////////////////////////////////////////



 // CLASE CPixel

class CPixel
{
// Atributos.

public:
    float m_f_gris;               // Nivel de gris.
    float m_f_opacidad;           // Grado de opacidad.
    unsigned short m_us_offset;   // Exclusivo para codificación "run-length" en pixels opacos.
                                  // Indica la distancia hasta el siguiente pixel no opaco,
                                  // dentro de la misma línea de rastreo de pixels.


// Operaciones.

    BOOL Es_Opaco (float umbral_opacidad)
    {
        // ENTRADA:
        //           - "umbral_opacidad": margen mínimo de opacidad.
        //
        // SALIDA:
        //
        // PROPÓSITO: Comprueba si un pixel supera el umbral de opacidad.

        return (m_f_opacidad >= umbral_opacidad);
    }
};
```

Ilustración 35.- Codificación de un *pixel* de la imagen intermedia.

```
void CVolume_Renderer::Renderizar ()
{
    // ENTRADA:
    //
    // SALIDA:
    //
    // PROPÓSITO: Realiza el volume rendering con los parámetros actuales previamente asignados.


    // Realizar la factorización de la matriz de vista, calculando los coeficientes
    // de "shearing" y "warping", así como el eje principal de vista y el orden
    // de apilamiento de los cortes.

    Factorizacion_Shear_Warp ();


    // Corrección de opacidad de los voxels en función de la orientación actual del volumen.

    Correccion_Opacidad ();


    // Calcular el mapa de colores para la orientación actual.

    Calcular_Mapa_de_Colores ();


    // Inicialización de la imagen intermedia a las dimensiones correspondientes según el eje principal.

    Inicializar_Imagen_Intermedia ();


    // Composición de cortes en la imagen intermedia.

    for (int corte= m_i_corte_inicio; corte != m_i_corte_fin; corte+= m_i_orden)
    {
        Componer_Corte ((unsigned short) corte);


        // Aplicación del "shearing" al corte siguiente.

        m_f_shearing_i_actual += m_f_paso_si;
        m_f_shearing_j_actual += m_f_paso_sj;
    }


    // Warping de la imagen intermedia.

    Warp ();
}
```

Ilustración 36.- Código fuente de la rutina “Renderizar”.



```
void CVolume_Renderer::Componer_Corte (unsigned short corte)
{
    // ENTRADA:
    //          - corte: Número de corte a componer.
    //
    // PROPÓSITO: Compone un corte determinado sobre la imagen intermedia en el orden de apilamiento correspondiente.


    Inicializar_Puntero_de_Voxels (corte);

    unsigned short longitud_minima_tramo;

    for (unsigned short scanline= 0; scanline != m_us_scanline_fin; scanline++)
    {
        Trasladar (scanline);

        while (m_voxel_ptr [0].longitud_restante_linea > 0)
        {
            // Si el pixel es opaco, saltar todos el tramo de pixels opacos.

            if (m_p_imagen_intermedia->m_buffer [m_ui_pos_pixel].Es_Opaco (m_f_umbral_opacidad))
            {
                Saltar_Tramo_de_Pixels ();
            }
            else
            {
                // Si son transparentes los voxels de ambas líneas de escaneo de voxels, saltar el tramo
                // más corto de entre ellas.

                longitud_minima_tramo= min (m_voxel_ptr [0].longitud_restante_tramo, m_voxel_ptr [1].longitud_restante_tramo);
                if ( (Es_Tramo_Transparente (m_voxel_ptr [0].P)) && (Es_Tramo_Transparente (m_voxel_ptr [1].P)) )
                {
                    Saltar_Tramo_de_Voxels (longitud_minima_tramo);
                }
                else
                {
                    // si no, remuestrear y componer el tramo de voxels.

                    Procesar_Tira_Voxels (longitud_minima_tramo);
                }
            }

        } // while (m_voxel_ptr [0].longitud_restante_linea > 0)

        // Actualizar las longitudes restantes de línea y activar la segunda línea de voxels si no es válida.

        m_voxel_ptr [0].longitud_restante_linea= m_us_tamano_i;
        m_voxel_ptr [1].longitud_restante_linea= m_us_tamano_i;
        if (!m_voxel_ptr [1].valida)
            m_voxel_ptr [1].valida= true;

    } // for (unsigned short scanline= 0; scanline < m_us_tamano_j; scanline++)

}
```

Ilustración 37.- Código fuente de la rutina “Componer_Corte”.

```
void CVolume_Renderer::Saltar_Tramo_de_Pixels ()
{
    // ENTRADA:
    //
    // PROPÓSITO: Salta el tramo de pixels opacos y los correspondientes voxels, recalculando
    //            la posición del siguiente pixel y voxels a procesar.


    unsigned short offset;
    unsigned short longitud_salto= 0;

    // Salto máximo para no salir de la línea de escaneo de la imagen intermedia.

    unsigned int salto_maximo= m_p_imagen_intermedia->Tamano_X () - (m_ui_pos_pixel_x);

    // Operación de búsqueda del final del tramo de pixels opacos.

    unsigned int aux;
    offset= m_p_imagen_intermedia->m_buffer [m_ui_pos_pixel].m_us_offset;

    while ( (offset > 0) && (longitud_salto < salto_maximo) )
    {
        longitud_salto+= offset;

        aux = m_ui_pos_pixel + longitud_salto;
        if (aux < m_p_imagen_intermedia->Tamano())
        {
            offset = m_p_imagen_intermedia->m_buffer[aux].m_us_offset;
        }
    }
    unsigned short longitud_camino= 0;

    // Operación "compresión de camino".

    offset= m_p_imagen_intermedia->m_buffer [m_ui_pos_pixel].m_us_offset;
    while ( (offset > 0) && (longitud_camino < salto_maximo) )
    {
        m_p_imagen_intermedia->m_buffer [m_ui_pos_pixel + longitud_camino].m_us_offset= longitud_salto - longitud_camino;
        longitud_camino+= offset;

        aux = m_ui_pos_pixel + longitud_camino;
        if (aux < m_p_imagen_intermedia->Tamano())
        {
            offset = m_p_imagen_intermedia->m_buffer[aux].m_us_offset;
        }
        else
        {
            offset = 0;
        }
    }

    // Una vez calculada la longitud del salto, actualizar la posición del siguiente pixel y voxels a procesar.

    Saltar_Tramo_de_Voxels (longitud_salto);
}
```

Ilustración 38.- Codificación de la rutina “Saltar_Tramo_de_Pixels”.

```
void CVolume_Renderer::Operador_Over (CPixel *pixel, float color, float alpha)
{
    // ENTRADA:
    //          - pixel: pixel de la imagen intermedia sobre el que se realiza la composición.
    //          - color: color del voxel a componer.
    //          - alpha: opacidad del voxel a componer.
    //
    // SALIDA:
    //          - pixel: pixel modificado tras la composición.
    //
    // PROPÓSITO: Operación de cálculo del color y de la opacidad del pixel resultante tras la composición
    //            de un voxel sobre él.


    // Cálculo del color.

    pixel->m_f_gris+= (color * alpha) * (1 - pixel->m_f_opacidad);
    if (pixel->m_f_gris > 255.0f) pixel->m_f_gris= 255.0f;


    // Cálculo de la opacidad.

    pixel->m_f_opacidad+= alpha * (1 - pixel->m_f_opacidad);
}
```

Ilustración 39.- Código de la rutina “Operador_Over”.

10 BIBLIOGRAFÍA

- Aho, A. V., Hopcroft, J. E. & Ullman, J. D. [1974]. "The design and analysis of computer algorithms", Addison-Wesley.

- Arvo, J. [1993]. "Transfer equations in global illumination", ed. P. Heckbert, New York.

- Arvo, J. & Kirk, D. [1990]. "Particle transport and image synthesis", Computer Graphics (SIGGRAPH '90 Proceedings), Vol. 24, Dallas.

- Blinn, J. F. [1982]. "Light reflection functions for simulation of clouds and dusty surfaces", Computer Graphics (SIGGRAPH '82 Proceedings), Vol. 16, Boston.

- Bracewell, R. N. [1986]. "The Fourier transform and its applications", McGraw Hill, New York.

- Cameron, G. G. & Undrill, P. E. [1992]. Rendering volumetric medical image data on a SIMD-architecture computer", Proceedings of the third eurographics workshop on rendering, Bristol.

- Catmull, E. & Smith, A. R. [1980]. "3D transformations of images in scanline order", Computer Graphics (SIGGRAPH '80 Proceedings), Vol. 14.

- Cohen, M. F. & Wallace, J. R. [1993]. "Radiosity and realistic image synthesis", Academic Press, Boston.

- Cormen, T. H., Leiserson, C. E. & Rivest, R. L. [1990]. "Introduction to algorithms", MIT Press, Cambridge.

- Danskin, J. & Hanrahan, P. [1992]. "Fast algorithms for volume ray tracing", 1992 workshop on volume visualization, Boston.

- Drebin, R. A., Carpenter, L. & Hanrahan, P. [1988]. "Volume rendering", Computer Graphics (SIGGRAPH '88 Proceedings), Vol. 22, Atlanta.

- Foley, J. D., Van Dam, A., Feiner, S. K. & Hughes, J. F. [1990]. "Computer graphics: Principles and practice", 2nd ed, Addison-Wesley, New York.

- Glassner, A. S. [1990]. "Graphics Gems", Academic Press, San Diego.

- Glassner, A. S. [1995]. "Principles of digital image synthesis", Morgan Kaufmann, San Francisco.

- Heckbert, P. S. [1986]. "Survey of texture mapping", IEEE Computer graphics & applications.

- Keller, P. J., Drayer, B. P., Fram, E. K., Williams, K. D., Dumoulin, C. L. & Souza, S. P. [1989]. “MR angiography with two-dimensional acquisition and three-dimensional display”.

- Klein, F. & Kübler, O. [1985]. “A prebuffer algorithm for instant display of volume data”, Proceedings of SPIE (Architectures and algorithms for digital image processing), Vol. 596.

- Lacroute, P. G. [1995]. “Fast volume rendering using a shear-warp factorization of the viewing transformation”, Technical report: CSL-TR-95-678, Stanford University.

- Laub, G. A. [1990]. “Displays for MR angiography”, Magnetic resonance in Medicine.

- Laub, G. A. & Kaiser, W. A. [1988]. “MR angiography with gradient motion refocusing”, Journal of computer assisted tomography.

- Laur, D. & Hanrahan, P. [1991]. “Hierarchical splatting: a progressive refinement algorithm for volume rendering”, Computer Graphics (SIGGRAPH ’91 Proceedings), Vol. 25, Las Vegas.

- Levoy, M. [1988]. “Display of surfaces from volume data”, IEEE Computer Graphics & Applications.

- Levoy, M. [1989]. “Display of surfaces from volume data”, Ph. D. dissertation, University of North Carolina.

- Levoy, M. [1990a]. “Efficient ray tracing of volume data”, ACM Transactions on graphics.

- Levoy, M. [1990b]. “A hybrid ray tracer for rendering polygon and volume data”, IEEE Computer Graphics & Applications.

- Max, N., Hanrahan, P. & Crawfis, R. [1990]. “Area and volume coherence for efficient visualization of 3D scalar functions”, Computer Graphics (San Diego workshop on volume visualization), Vol. 24, San Diego.

- Meagher, D. J. [1980]. “Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3D objects by computer”, Rensselaer Polytechnic Institute (Technical Report IPL-TR-80-111).

- Meagher, D. J. [1982]. “Efficient synthetic image generation of arbitrary 3D objects”, Proceeding of the IEEE conference on pattern recognition and image processing.

- Montani, C. & Scopigno, R. [1990]. “Rendering volumetric data using the STICKS representation scheme”, Proceedings of the 1992 workshop on volume visualization, San Diego.

- Novins, K. & Arvo, J. [1992]. “Controlled precision volume rendering”, 1992 workshop on volume visualization”, Boston.

- Porter, T. & Duff, T. [1984]. “Compositing digital images”, Computer Graphics (SIGGRAPH ’84 Proceedings), Vol. 18, Minneapolis.

- Press, W. H., Teukolsky, S. A., Vetterling W. T. & Flannery B. P. [1999]. “Numerical recipes in C. The art of scientific computing”, 2nd ed., Cambridge University Press.

- Reynolds, R. A., Gordon, D. & Chen, L. S. [1987]. “A dynamic screen technique for shaded graphics display of slice-represented objects”, Computer vision, graphics and image processing.

- Sabella, P. [1988]. “A rendering algorithm for visualizing 3D scalar fields”, Computer Graphics (SIGGRAPH ’88 Proceedings), Vol. 22, Atlanta.

- Shirley, P. & Tuchmann, A. [1990]. “A polygonal approximation to direct scalar volume rendering”, 1990 workshop on volume visualization, San Diego.

- Siegel, R. & Howell, J. R. [1992]. “Thermal radiation heat transfer”, Hemisphere Publishing, New York.

- Subramanian, K. R. & Fussell, D. S. [1990]. “Applying space subdivision techniques to volume rendering”, Proceedings of visualization ’90, San Francisco.

- Sweeney, J. & Mueller, K. [2002]. “Shear-Warp Deluxe: The shear-warp algorithm revisited”, Department of Computer Science, State University of New York at Stony Brook.

- Upson, C. & Keeler, M. [1988]. “V-BUFFER: Visible volume rendering”, Computer Graphics (SIGGRAPH ’88 Proceedings), Vol. 22, Atlanta.

- Westover, L. [1989]. “Interactive volume rendering”, Chapel Hill workshop on volume visualization, Chapel Hill, North Carolina.

- Westover, L. [1990]. “Footprint evaluation for volume rendering”, Computer Graphics (SIGGRAPH ’90 Proceedings), Vol. 24, Dallas.

- Wilhelms, J. & Van Gelder, A. [1991]. “A coherent projection approach for direct volume rendering”, Computer Graphics (SIGGRAPH ’91 Proceedings), Vol. 25, Las Vegas.

- Wolberg, G. [1990]. “Digital image warping”, IEEE computer society press, Los Alamitos, California.

11 ÍNDICE DE ILUSTRACIONES