Universidad de Valladolid

UNIVERSIDAD DE VALLADOLID

ESCUELA DE INGENIERIAS INDUSTRIALES

GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

DISEÑO E IMPLEMENTACIÓN DE UN EXPLORADOR DE ARCHIVOS DICOM PARA REALIDAD VIRTUAL Y AUMENTADA

Autor:

Sánchez Brizuela, Guillermo

Tutor:

De la Fuente López, Eusebio

Dpto. Ingeniería de Sistemas y Automática

Valladolid, julio 2020.

Resumen y palabras claves

Resumen

A la hora de visualizar los resultados de una imagen de resonancia magnética o de una tomografía computarizada, los profesionales médicos deben buscar detalles en múltiples planos paralelos que se visualizan en forma de imagen bidimensional. Con el objeto de encontrar alternativas más cómodas, intuitivas y eficaces a esta forma de visualización, en este documento se plantea el desarrollo de una herramienta de exploración de archivos DICOM que genera una reconstrucción volumétrica de los resultados de las pruebas comentadas anteriormente, permitiendo al usuario explorar un modelo tridimensional del paciente en un entorno de realidad virtual, mejorando de esta forma la interacción y visualización de este tipo de archivos respecto a los métodos actuales.

Palabras clave

- Reconstrucción 3D
- Realidad Virtual
- DICOM
- Renderización Volumétrica
- Imagen médica

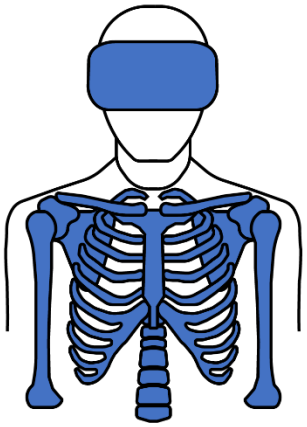
Abstract and keywords

Abstract

When it comes to visualizing results of a magnetic resonance or a computed tomography scan, medical professionals have to look for details in multiple parallel slices, bidimensionally-displayed. Aiming to find more intuitive, more efficient, comfortable-to-use visualization alternatives, this document presents the development of a DICOM file explorer that generates a volumetric reconstruction of the results of the earlier mentioned procedures, allowing the user to explore a 3D model of the patient in a virtual reality environment, improving interaction and visualization of this type of files compared to actual methods.

Keywords

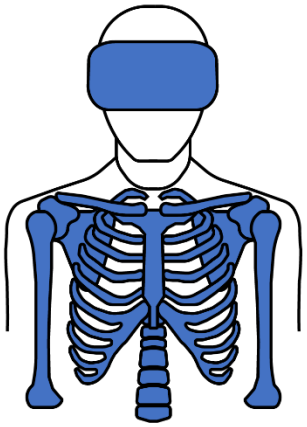
- 3D Reconstruction
- Virtual Reality
- DICOM
- Volumetric Rendering
- Medical Imaging



Índices

Contenido

Ilustraciones



Capítulo 1: Introducción

En este primer capítulo, se introduce el trabajo de fin de grado realizado. Para esto se presenta la justificación del trabajo, los objetivos y subobjetivos que se esperan alcanzar una vez se haya completado su desarrollo, la situación actual en la que se encuentran tecnologías similares, y por último, la estructura que sigue el propio documento.

1.1 Justificación del estudio

Vivimos en un mundo cada vez más conectado donde la interacción en entornos virtuales está cada vez más presente. El estudio que se lleva a cabo en el presente documento persigue avanzar un paso hacia delante en la visualización de archivos de imagen médica con el desarrollo de una herramienta que presente una mejora en las funcionalidades que ofrecen los métodos actuales, y pueda sirva de base para aplicaciones más complejas como podrían ser:

- Herramientas en línea de telemedicina, que permitan a varios expertos discutir y planear diagnósticos y operaciones, compartiendo un mismo espacio virtual en el que comentar sobre una reconstrucción del paciente, independientemente del lugar del mundo en el que estén.
- Herramientas educativas, que permitan a los estudiantes visualizar modelos tomados de pacientes reales, para poder visualizar más claramente síntomas de ciertas patologías o claves de un diagnóstico realizado previamente por profesionales.

Los entornos de realidad virtual proporcionan al profesional sanitario una nueva dimensión de libertad y de posibilidades, además de abandonar las restricciones de un entorno de trabajo en el mundo real. Estas características contrastan con los monitores bidimensionales con los que se ha trabajado hasta ahora, que trasladan de una forma más pobre la información que contiene un modelo 3D al usuario.

1.2 Objetivos

El objetivo básico de este trabajo es el desarrollo de una herramienta funcional que pueda emplearse para explorar y visualizar archivos médicos en entornos virtuales, para esto, se definen una serie de subobjetivos:

- Reconstrucción tridimensional:

Capacidad de generar una reconstrucción tridimensional a partir de un archivo obtenido de una tomografía computarizada o de una resonancia magnética.

- Renderización nítida de la reconstrucción:

Para su correcto estudio, es indispensable que los modelos generados se visualicen de forma que no se generen artefactos que puedan dar lugar a un diagnóstico erróneo.

- Distintas herramientas de ayuda:

Para la visualización de los modelos, se desarrollarán distintas herramientas que permitan interactuar con el modelo, como pueden ser la realización de cortes, la traslación y rotación, o el cambio del tejido que se desea observar.

- Comodidad de uso e interfaz intuitiva:

La herramienta debe contar con una interfaz de usuario intuitiva para requerir al personal no familiarizado con entornos virtuales del menor entrenamiento posible para su correcto uso.

- Compatibilidad con el estándar DICOM para su integración en sistemas médicos.

Como se verá más adelante, el estándar DICOM está integrado en todos los aparatos de generación y almacenamiento de imágenes médicas en hospitales y centros sanitarios, es por esto y por la cantidad de información que contiene que se espera que la aplicación funcione siguiendo este estándar.

1.3 Estado del arte

Actualmente, existen varias soluciones de renderización volumétrica para archivos médicos, algunas de ellas, similares a la desarrollada en el presente proyecto, para realidad virtual:

- **RadiAnt – Medixant** [1]: RadiAnt es un explorador de archivos DICOM para monitores tradicionales que incorpora las siguientes características:
  - Renderización de archivos en 3D (Ilustración 1).
  - Reconstrucciones multiplanares.
  - Soporte de múltiples archivos DICOM.
  - Exportación de archivos a imágenes.
  - Capacidad de conectarse a servidores médicos que siguen el estándar DICOM.
  - Herramientas de manipulación como ajuste de centro y anchura de ventana, ajuste de brillo, herramienta para dibujar, etc.

Ilustración 1: Reconstrucción 3D dentro de la aplicación RadiAnt viewer. Fuente: RadiAnt DICOM viewer [En línea] Disponible en https://www.radiantviewer.com/dicom-viewer-manual/3d-volume-rendering.html

- **DICOM VIEWER XR – Medicalholodeck** [2]: Esta herramienta permite la exploración de archivos DICOM en realidad virtual y presenta las siguientes características:
  - Soporte de archivos DICOM.
  - Visualización 3D (Ilustración 2).
  - Corte de los archivos.
  - Aplicación de gamas de color.
  - Escala y rotación (no traslación).
  - Colaboración multiusuario.

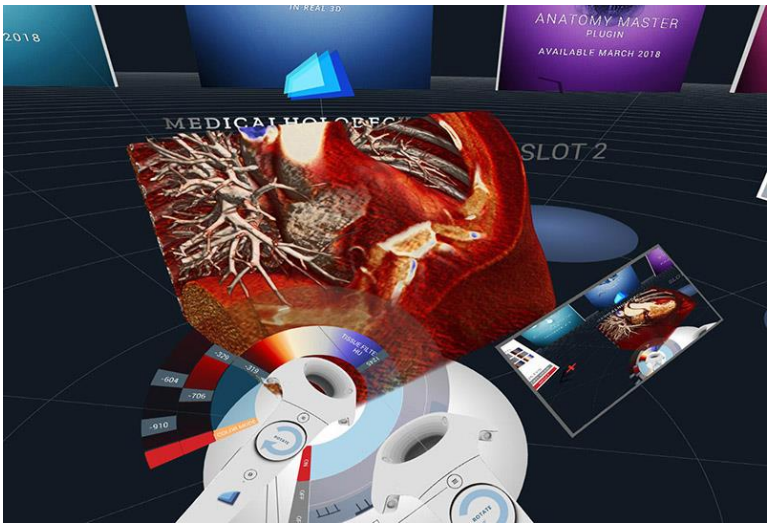


Ilustración 2: Interior de la aplicación DICOM Viewer XR. Fuente: DICOM VIEWER XR [En línea] Disponible en https://www.medicalholodeck.com/medical-virtual-augmented-reality-images/medical-virtual-augmented-reality-dicom-images.html

- **DICOM VR** [3]: Desarrollado por investigadores del Instituto del Cáncer Dana-Farber de Boston, permite, entre otros:
  - Visualizar reconstrucciones tridimensionales en realidad virtual.
  - Trabajar con archivos DICOM.
  - Escalar, rotar y trasladar dichos archivos.
  - Segmentación interactiva de zonas.
  - Modificación en tiempo real de la reconstrucción (retirar secciones y abrir agujeros) (Ilustración 3).

Ilustración 3: Visualización de la reconstrucción en realidad virtual de DICOM VR con una sección recortada. Fuente: DICOM VR [En línea] Disponible en http://www.dicomvr.com/

1.3.1 Crítica al estado del arte

Si bien es cierto que las aplicaciones mencionadas representan archivos DICOM de forma volumétrica, solo algunas de ellas permiten explorarlos en realidad virtual, conservando de esta forma la información tridimensional perdida al visualizarlo en un monitor convencional. Podemos observar también cómo algunas de estas últimas emplean algoritmos de renderización que generan artefactos, ofrecen reconstrucciones pobres o no permiten la manipulación libre del modelo. Además de esto, estas soluciones son aplicaciones cerradas, las cuales no permiten su uso como base para futuros desarrollos e investigaciones.

1.3.2 Propuesta

Este trabajo, busca, entre otros: Encontrar el equilibrio entre una representación fiel de los archivos médicos y un uso eficiente de los recursos computacionales, una interfaz cómoda e intuitiva para el usuario, y la creación de una base abierta para continuar el desarrollo de distintas aplicaciones o modificaciones de esta misma incorporando nuevas funcionalidades como pueden ser las expuestas en el apartado **5.2 Líneas futuras**.

1.4 Estructura del trabajo

Este trabajo de fin de grado se ha dividido en cinco capítulos, donde se encuentran los siguientes temas:

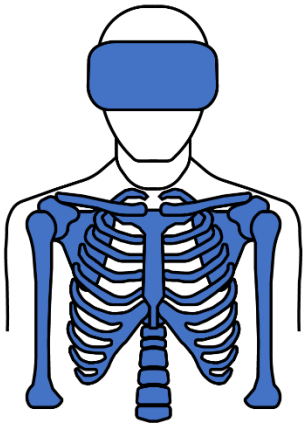
- **Introducción**: Justificación del proyecto, sus objetivos, el estado del arte del tema que se trata en el documento y esta descripción de la estructura que se ha seguido.

- **Fundamento teórico**: Descripción de los temas centrales del trabajo, que son: Imagen médica, el estándar DICOM, el proceso de renderización y la Realidad virtual. Se hace hincapié tanto en la parte de DICOM como en la de Realidad virtual, debido a la escasa literatura que existe, especialmente en castellano.

- **Herramientas empleadas**: Herramientas que se han usado a lo largo del proyecto para su desarrollo y prueba, tanto software como hardware.

- **Desarrollo**: Lógica y relaciones internas de las distintas partes que componen la aplicación, así como las decisiones tomadas durante el proceso de desarrollo y sus justificaciones. También se encuentra en este capítulo un apartado con los resultados del proyecto.

- **Conclusiones**: Conclusiones del proyecto, además, un apartado con las posibles líneas de investigación sobre las que se podría continuar trabajando a partir de este trabajo de fin de grado.



Capítulo 2: Fundamento teórico

Se introduce en este capítulo el marco teórico en el que se desarrolla este trabajo de fin de grado, definiendo los conceptos, bases, modelos, y algoritmos fundamentales de las distintas materias de las que trata el trabajo.

2.1 Imagen médica

El término imagen médica agrupa el conjunto de técnicas empleadas para generar representaciones visuales del interior de un cuerpo con el objetivo de descubrir, controlar o diagnosticar patologías. Dentro de este conjunto de técnicas, se introducen a continuación las que son de especial relevancia para el documento por ser las que generan las imágenes con las que trabaja la aplicación desarrollada en este trabajo de fin de grado.

2.1.1 Tomografía computarizada [4]: Las tomografías axiales computarizadas (*X-ray computed tomography, CT*) obtienen representaciones visuales de secciones transversales del cuerpo empleando las propiedades de atenuación de rayos x de las distintas partes de este, es un procedimiento rápido, indoloro y no invasivo. Los rayos x son producidos por un tubo de rayos x, atenuados por el paciente y medidos por un detector. Este proceso se repite en cantidad de ángulos, almacenando las medidas. Una vez se tienen suficientes medidas, es posible reconstruir la atenuación en cada punto del plano transversal escaneado.
Las imágenes obtenidas presentan los valores medidos en unidades Hounsfield (HU), definidos por la Ecuación 1.

$$\text{Valor de la TC (HU)} = \frac{\mu - \mu_{H_2O}}{\mu_{H_2O}} * 1000$$

Ecuación 1: Cálculo de los valores de la tomografía.

En dicha ecuación, μ es el coeficiente de absorción. Para su correcta visualización, es necesario trasladar estos valores a una escala de grises, lo cual, se consigue a través de las transformaciones detalladas en el apartado **2.2.2.3 Características de visualización.**

2.1.2 Resonancia magnética [5]: Las resonancias magnéticas (*Magnetic Resonance imaging, MRI*) emplean la respuesta de los protones del cuerpo ante campos magnéticos externos para generar la imagen. Este proceso se realiza en dos partes, primero se manipula la orientación del spin del protón empleando un conjunto de campos magnéticos y después se miden cambios en la orientación del campo magnético del protón con un sensor inductivo aplicando una señal de radiofrecuencia. Esta lectura, dependiendo de cómo se interprete, permite obtener imágenes potenciadas en T1 o en T2, que destacan distintos tejidos. Este procedimiento no emplea radiación y produce información más detallada que las tomografías computarizadas a cambio de ser un proceso más lento.

A la hora de almacenar y transmitir archivos generados a través de estos estudios, se emplea el estándar DICOM, que se expone a continuación.

2.2 DICOM

DICOM[1] (*Digital Imaging and Communication In Medicine*) es el estándar para transmisión y almacenamiento de imágenes y datos médicos entre hardware y software de propósito médico. Permite integrar sistemas médicos como escáneres, servidores, impresoras, PACS (*Picture Archiving and Communication Systems* – Un sistema computarizado que permite el almacenaje de imágenes y su transmisión a través de una red informática) de diferentes fabricantes, asegurando su compatibilidad.
DICOM ha provisto a la comunidad médica, entre muchas otras cosas, de [6]:

- Un estándar universal para la medicina digital. Todas las máquinas de adquisición de imagen médica producen imágenes DICOM, y se comunican a través de redes DICOM.

- Muy alta calidad de imagen. DICOM soporta 16 bits para almacenar tonalidades de gris, esto significa que puede almacenar 65536 tonalidades de gris distintas para imágenes monocromáticas. En comparación, los formatos típicos de imagen como JPEG están limitados a 256 valores.

- Soporte para una gran cantidad de parámetros asociados al estudio y al proceso de adquisición de imágenes. El formato DICOM, además de la imagen como tal, almacena información como puede ser la posición del paciente, su orientación, filtros aplicados, etc. Para nuestro estudio, por ejemplo, son de vital importancia los parámetros asociados a la distancia entre cada uno de los planos adquiridos en las resonancias o tomografías, ya que sin ellos sería imposible realizar una reconstrucción fiel a la realidad.

- Codificación de la información. Los archivos y mensajes DICOM almacenan la información en más de 2000 atributos estandarizados. Estos atributos permiten almacenar los parámetros mencionados anteriormente, información sobre el paciente y el estudio que se le está realizando, o incluso sobre el diagnostico que se está llevando a cabo.

- Funcionalidad e interfaz clara. DICOM define interfaces muy concisas e independientes del tipo de aparato para hacer que el proceso de trabajar con equipos médicos a través de sus interfaces DICOM sea un proceso robusto y predecible que reduzca lo máximo posible la posibilidad de que aparezcan errores.

[1] Es posible consultar el estándar DICOM en su página web, https://www.dicomstandard.org/

2.2.1 Términos básicos de DICOM

- IODs (*Information Object Definitions*) [7]: Los IODs son una abstracción de información orientada a objetos que especifican información sobre elementos del mundo real. Los IODs no representan una instancia especifica de un objeto, si no una clase de objetos reales que comparten las mismas propiedades. Podemos pensar en los IODs como una colección de atributos que describen ciertas propiedades, un IOD de paciente, tendrá ciertas propiedades como nombre, edad, sexo, peso, etc.

- AEs (*Application Entities*): Cada uno de los dispositivos y programas que siguen DICOM. Las entidades de aplicación se proveen servicios mutuamente.

- SOPs (*Service-Object Pairs*): Asociación entre un servicio y el IOD que procesan, estas asociaciones se agrupan en Clases SOP. Por ejemplo, guardar una radiografía desde un equipo de rayos X en un PACS corresponde a un SOP de almacenaje, en este ejemplo, la radiografía sería el IOD, mientras que el equipo de rayos X haría una petición del servicio de almacenaje al PACS, que respondería con dicho servicio al equipo de rayos X.

- SCU (*Service Class User*) y SCP (*Service Class Provider*): Dentro de una petición/respuesta de un servicio, se denota como SCU a la entidad que realiza una petición de un servicio y como SCP a la entidad que responde con el servicio en cuestión. Cada intercambio de parejas SCU/SCP se llama asociación, y cada transferencia en la red comienza con un establecimiento de asociación (*Association Establishment*).

- Declaración de conformidad: Cada entidad DICOM, viene acompañada de este documento, que especifica sus capacidades funcionales, como pueden ser qué servicios soporta la entidad, si actúa como SCU, SCP o ambas, etc.

- VR[2] (*Value Representation*) [8]: Los atributos DICOM, en los que se profundiza a continuación, mantienen un formato acorde a 28 tipos de representación de valores, que incluyen datos típicos como enteros con y sin signo, cadenas de texto, números en coma flotante, y tipos más complejos como fechas, horas, nombres, etc.

[2] Se puede consultar la lista completa de VRs en [5]

2.2.2 Atributos

Los atributos codifican la información del archivo DICOM y cuentan con dos elementos fundamentales:

- Una etiqueta que identifica el atributo, en formato (XXXX, XXXX) donde las X son números hexadecimales, siendo los cuatro primeros dígitos el número de grupo y los cuatro últimos el número de elemento.
- Un VR (*Value Representation*) que indica el tipo de dato y el formato de dicho atributo.

Se comentarán a continuación los atributos especialmente relevantes para el desarrollo de la aplicación, agrupándolos según su función dentro de esta.

2.2.2.1 Características digitales de las imágenes:

- Número de muestras (0028, 0002): Número de canales en la imagen.

- Número de frames (0028, 0008): Número de secciones en una imagen de múltiples secciones.

- Número de filas (0028, 0010): Número de filas en la imagen.

- Número de columnas (0028, 0011): Número de columnas en la imagen.

- Bits asignados (0028, 0100): Define cuanta memoria se reserva para cada valor de píxel almacenado (Ilustración 4).

- Bits almacenados (0028, 0101): Define cuantos de los bits asignados son realmente usados para almacenar el valor del píxel (Ilustración 4). El resto de los bits deben ser despreciados usando una máscara ya que pueden almacenar información ajena al valor del píxel.

- Bit más alto (0028, 0102): Número que identifica el bit más alto usado de los bits asignados, en el estándar está fijado al número de bits almacenado menos uno (Ya que el primer bit se denomina cero) (Ilustración 4).

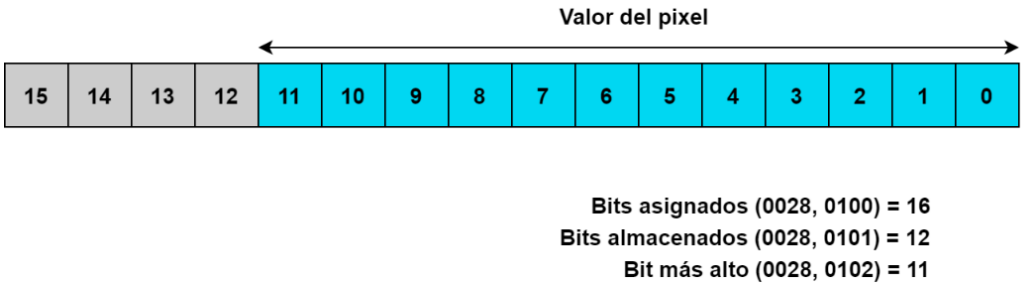


Ilustración 4: Ejemplo de la estructura del valor de un pixel almacenado en un archivo DICOM. Fuente: Elaboración propia.

2.2.2.2 Características físicas de las imágenes:

- Grosor de sección (0018, 0050): Grosor de cada sección del estudio en milímetros.

- Espacio entre secciones (0018, 0088): Distancia entre las secciones de centro a centro en milímetros.

- Posición de la imagen (0020, 0032): Coordenadas x, y, z de la esquina superior izquierda de la sección en milímetros, es preferible usar esta etiqueta para calcular las distancias entre cada una de las secciones en vez de "Espacio entre secciones" (0018, 0088) ya que en caso de que existan varias distancias distintas, este último valor no sería fiable .

- Orientación de la imagen (0020, 0037): Cosenos de dirección de la primera fila y de la primera columna respecto del paciente.

- Espacio entre píxeles (0028, 0030): Distancias en milímetros entre filas y columnas adyacentes especificadas por dos valores numéricos.

2.2.2.3 Características de visualización:

A la hora de visualizar las imágenes DICOM, estas pueden tener que someterse a una serie de operaciones [9] (Ilustración 5) para transformar los valores almacenados en los píxeles en valores de visualización (*Presentation values* o *p-values*). Dentro de estas transformaciones, se tratarán los atributos relacionados con imágenes en niveles de gris.

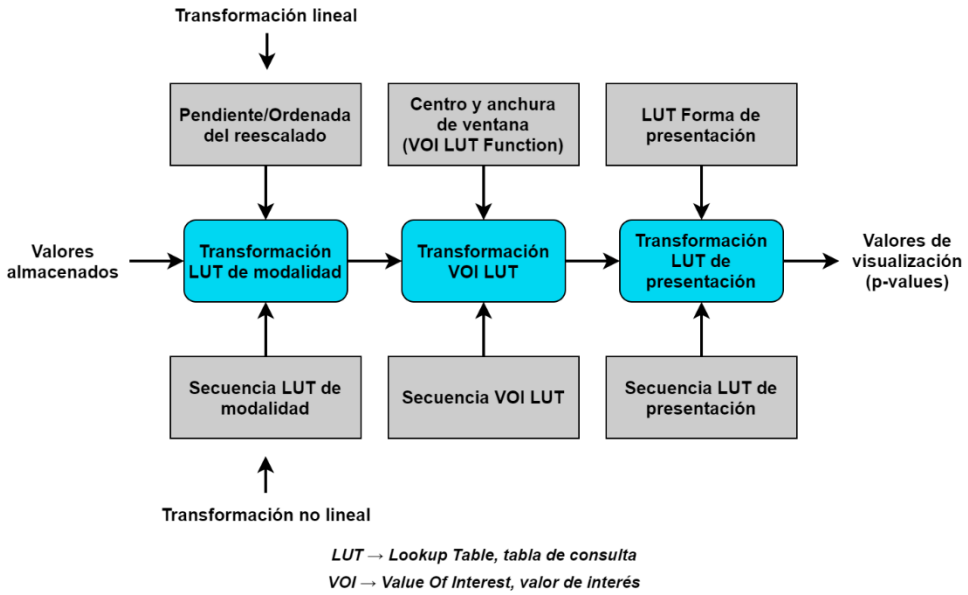


Ilustración 5: Secuencia simplificada de transformaciones para convertir los valores almacenados en valores de visualización. Fuente: Elaboración propia.

Transformación LUT de modalidad (*Modality LUT Transformation*) [10]**:** Esta transformación se encarga de convertir los valores recogidos por el fabricante en valores con significado para cada modalidad de imagen, por ejemplo, los valores de Hounsfield para tomografías. Si la transformación es lineal, Resultado = valor_almacenado*a + b los parámetros serán:

- Ordenada en el origen de reescalado (0028, 1052): Unidades de modalidad equivalentes cuando el valor almacenado es cero.

- Pendiente de reescalado (0028, 1053): Pendiente de la recta de la transformación a aplicar para convertir los valores almacenados en valores de modalidad.

Si la transformación no es lineal, la tabla de consulta necesaria se encuentra en la etiqueta "*LUT Data*" (0028, 3006) asociados a "*Modality LUT Sequence*" (0028, 3000).

Transformación VOI LUT (*VOI LUT Transformation*) [11]**:** La transformación VOI (*Value of interest*, valor de interés) transforma los valores específicos de cada modalidad en valores interpretables por el usuario. El tipo de transformación que se aplica viene especificado en la etiqueta "*VOI LUT Function*" (0028, 1056), que puede tomar los valores: LINEAR, LINEAR_EXACT o SIGMOID. Si este atributo no está presente, el tipo de transformación a aplicar es LINEAR. Los parámetros de estas transformaciones vienen especificados en:

- Centro de la ventana (0028, 1050): Valor medio donde se aplicará la transformación.

- Anchura de la ventana (0028, 1051): Amplitud de la transformación.

LINEAR
Siendo x el valor almacenado, c el centro de la ventana y w el ancho de la ventana, el valor de salida, y, viene dado por la
Ecuación 2.

$$\begin{cases} if\ \ x \leq c - 0.5 - \dfrac{w-1}{2} & y = y_{min} \\ if\ x > \ c - 0.5 + \dfrac{w-1}{2} & y = y_{max} \\ else & y = \dfrac{x-(c-0.5)}{(w-1)+0.5} * (y_{max} - y_{min}) + y_{min} \end{cases}$$

Ecuación 2: Transformación LINEAR.

LINEAR_EXACT
Siendo x el valor almacenado, c el centro de la ventana y w el ancho de la ventana, y el valor de salida, y, viene dado por la Ecuación 3.

$$\begin{cases} if\ \ x \leq c - \dfrac{w}{2} & y = y_{min} \\ if\ x > \ c + \dfrac{w}{2} & y = y_{max} \\ else & y = \dfrac{x - c}{w} * (y_{max} - y_{min}) + y_{min} \end{cases}$$

Ecuación 3: Transformación LINEAR_EXACT.

SIGMOID
Siendo x el valor almacenado, c el centro de la ventana y w el ancho de la ventana, el valor de salida, que después se redondea a un entero, viene dado por la Ecuación 4.

$$y = \frac{y_{max} - y_{min}}{1 + e^{\left|-4 * \frac{x-c}{w}\right|}} + y_{min}$$

Ecuación 4: Transformación SIGMOID.

En caso de que la etiqueta "Centro de la ventana" (0028, 1050) no esté definida, lo estará la etiqueta "*VOI LUT Sequence*" (0028, 3010) asociada a "*LUT Data*" (0028, 3006) que definirá una tabla de consulta para realizar esta transformación.

Transformación LUT de presentación (*Presentation LUT Transformation*) [12]**:**
La transformación de presentación, convierte los valores de cada pixel en el valor de presentación (*p-value*), es decir, el que se mostrará a través de la pantalla. Esta transformación puede estar definida por la etiqueta "*Presentation LUT Shape*" (2050, 0020), que puede tomar dos valores:

IDENTITY
No se realiza ninguna transformación, los *p-values* son los obtenidos de la transformación de valor de interés (*VOI LUT Transformation*).

INVERSE
Se realiza una inversión a los valores resultantes de la transformación de VOI LUT para obtener los *p-values*.

En caso de que esta etiqueta no esté presente, lo estará "*Presentation LUT Sequence*" (2050, 0010) con su correspondiente tabla de consulta definida en "*LUT Data*" (0028, 3006).

2.2.2.4 Otros atributos:

- Interpretación fotométrica (0028, 0004): Interpretación esperada para los valores de los píxeles, por ejemplo, monocromático, RGB, YBR, etc. Es posible consultar la lista completa de interpretaciones en [13].

- *Píxel Data* (7FE0, 0010): Conjunto de datos que contiene la información relativa a los valores medidos para cada píxel de la imagen.

Los archivos DICOM, necesitan visualizarse para realizar diagnósticos, planificar operaciones, o estudiar el avance de enfermedades. Para esto, es común generar imágenes en monitores. Este proceso, en el cual se profundiza en el siguiente apartado, se conoce como renderización.

2.3 Renderización

La renderización (del inglés *rendering*) consiste en la síntesis automática de una imagen digital bidimensional a través de un programa informático partiendo de un modelo 3D o un conjunto de ellos en lo que se considera una escena. Esto se hace calculando cuanta luz llega a cada píxel del sensor de una cámara virtual.
Teóricamente, se deberían simular las interacciones físicas de todos los fotones presentes. Dado que esto es claramente inviable, se muestrean algunos de ellos (*sample*) y se generaliza una estimación de la luz que realmente llegaría al sensor.

Para realizar dichos cálculos y generar la imagen, existen numerosos algoritmos, que pueden llevarse a cabo tanto en tiempo real como en una etapa de prerrenderizado.

El renderizado en tiempo real es mucho más rápido que el prerrenderizado y se emplea para generar una sucesión de imágenes en entornos digitales como videojuegos (Ilustración 6) o escenas de realidad virtual y aumentada. Para esto se emplean métodos de aceleración por hardware, como por ejemplo el uso de tarjetas gráficas dedicadas, que facilitan la computación en paralelo en vez de usar la arquitectura de propósito general de una CPU.
Por otro lado, el prerrenderizado genera resultados mucho mejores, a cambio de requerir un coste computacional mucho mayor, este tipo de renderización se emplea en la industria cinematográfica (Ilustración 7) y para la generación de imágenes hiperrealistas.

Debido al tipo de aplicación desarrollada, este documento se centrará en el estudio de la renderización en tiempo real.

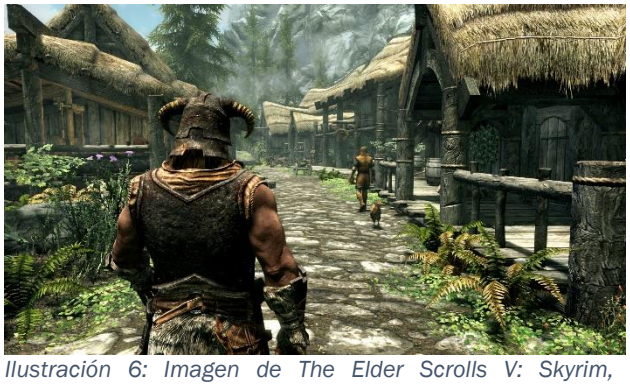


Ilustración 6: Imagen de The Elder Scrolls V: Skyrim, videojuego desarrollado por Bethesda Game Studios. Fuente: "The Elder Scrolls V: Skyrim" de Bethesda Game Studios.

Ilustración 7: Detalle del polo de un personaje de la película de animación Los Increíbles 2, producida por Pixar. Fuente: "Los Increíbles 2" de Pixar.

2.3.1 Conceptos básicos

Se exponen a continuación una serie de apartados que tratan las bases de la renderización, los cuales se emplearan en la descripción del proceso final en el apartado **2.3.3 Render Pipeline.**

2.3.1.1 Algoritmos de renderización [14]

Como simplificación, se considera una cámara virtual estenopeica[3], con un plano virtual delante del centro de proyección y solamente se muestrearán los rayos dentro de un cono cuyo ápice está situado en el estenopo (*pinhole*) de dicha cámara.

Se considera también que cada uno de los píxeles del sensor (distintos en concepto de los píxeles de la imagen) estiman el valor de su lectura usando una sola medida en el centro del área de dicha parte del sensor. De esta forma, los rayos de muestreo serán aquellos que atraviesan cada uno de los centros de los píxeles del sensor con origen en el centro de proyección (el estenopo).

Como una primera aproximación de alto nivel a los algoritmos, se considera que el proceso de renderizado (Ilustración 8) consiste en para cada punto P visible en la escena, con dirección ω_0 hasta el centro de un píxel (x, y), asignar a ese píxel el sumatorio de la luz reflejada que proviene en direcciones ω_i. Se puede ver una simplificación del proceso en el Pseudocódigo 1.

```
Para cada punto visible P con dirección ω0 desde un píxel (x, y):
    suma = 0
    Para cada incidencia de luz con dirección ωi en P:
        suma = suma + Luz reflejada en P desde ωi hasta ω0
    Píxel[x, y] = suma
```

Pseudocódigo 1: Proceso de renderizado simplificado

Las incidencias que se mencionan en el Pseudocódigo 1, serán intersecciones con la geometría virtual de la escena, el cómo se definen estas geometrías, compone la siguiente sección de este fundamento teórico.

[3] Las cámaras estenopeicas o *pinhole cameras*, están formadas por una caja estanca y un pequeño orificio (estenopo), a través del cual entra la luz, que se proyecta en el interior de la caja formando la imagen.

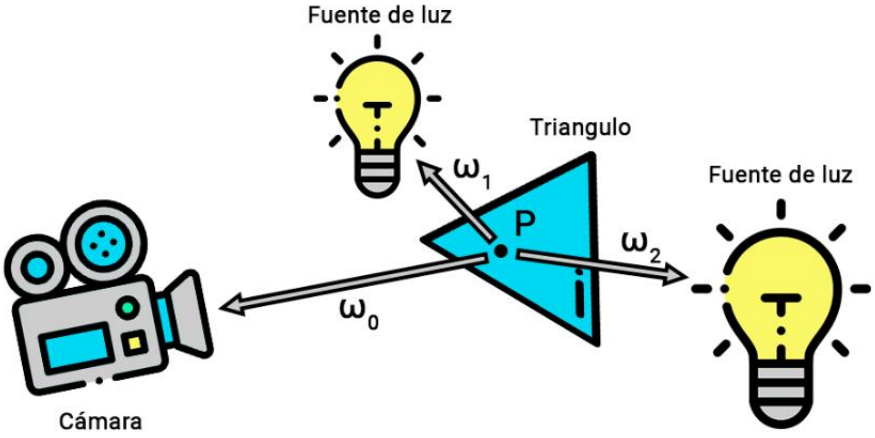


Ilustración 8: Representación de las direcciones ω_i. Fuente: Elaboración propia (Iconos hechos por Freepik disponibles gratuitamente en https://www.flaticon.com/)

2.3.1.2 Definición del conjunto de puntos visibles [14]

Para representar un objeto, se considera suficiente modelar las superficies que forman los límites de este. En los modelos 3D, estas superficies están formadas por mallas de triángulos (Ilustración 9), definidos por sus tres vértices.

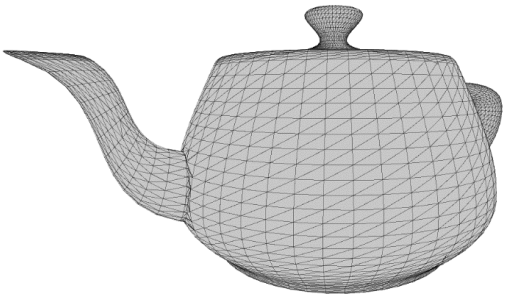


Ilustración 9: Malla de triángulos de la tetera de Utah. Fuente: Elaboración propia. Archivo: Utah teapot, Martin Newell.

Estas mallas, se consideran cerradas, lo que significa que los triángulos solo tendrán una cara visible. Vistos desde fuera, los triángulos se definen por sus vértices en sentido contrario a las agujas del reloj, almacenando de esta forma el sentido que apunta hacia el exterior de la dirección normal al triangulo.

Estas superficies definen el conjunto de posibles puntos visibles. Para obtener el resultado de un determinado píxel, se debe determinar cuáles son los puntos que se proyectan sobre el píxel correspondiente del sensor y elegir el más cercano de todos. Ese punto será el realmente visible para la cámara.

Para encontrar los puntos visibles, la primera iteración del Pseudocódigo 1 se convertirá en dos iteraciones, una sobre los centros de los píxeles y otra sobre los triángulos que conforman las mallas.

Dependiendo de sobre qué se itere primero, se obtienen dos técnicas de renderizado distintas, ray casting y rasterización, expuestas a continuación. Hay que tener en cuenta que estas técnicas son una simplificación de los algoritmos empleados normalmente, además de servir como base para conseguir técnicas de renderizado más complejas.

2.3.1.3 Ray casting

Se itera primero sobre cada uno de los píxeles, creando un rayo por píxel (con dirección estenopo-centro del píxel) y comprobando si interseca con cada triangulo. Se selecciona la intersección con el triángulo a menor distancia y la iluminación reflejada desde las fuentes de luz en dicha superficie. Esto permite procesar cada píxel de forma independiente, lo que facilita el procesamiento paralelo. Los píxeles necesitan acceder a todos los triángulos, por lo que debemos mantener la escena completa cargada en memoria.

Esto es una particularización no iterativa del algoritmo de **ray tracing**, que consiste en trazar rayos en todas las direcciones recursivamente para cada colisión de los rayos (partiendo del inicial) y realizar un cómputo de la luz directa y reflejada que llega hasta cada uno de los puntos de la escena.

2.3.1.4 Rasterización

Se itera primero sobre cada uno de los triángulos de las mallas, para después iterar sobre los píxeles de las filas de la imagen, (*rasters* en inglés). Este método permite procesar cada triangulo independientemente. Al contrario que en el ray casting, no es necesario mantener almacenada la escena en memoria, ya que solo se necesitan las propiedades de un triángulo cada vez.

Por otro lado, ahora se deben comparar las distancias de las intersecciones de los rayos con los triángulos para cada uno de los píxeles. Si se recurre a la computación paralela, es necesario almacenar un recurso compartido en forma de matriz bidimensional del mismo tamaño que la imagen de salida.
Esta matriz, ***z-buffer*** **o buffer de profundidad**, almacenará la menor distancia de un triángulo al píxel correspondiente, permitiendo así la comparación de distancias entre distintos triángulos.

La rasterización, en principio, no define ninguna forma de calcular el color o la iluminación. Para esto se emplearán distintas técnicas de sombreado (*shading*) que podrán estar basados en el comportamiento físico de la luz (por ejemplo, el modelo Lambertiano) o buscar efectos artísticos. Estos cálculos se definirán en **shaders**, programas destinados a ejecutarse en la tarjeta gráfica dedicada. En esto se profundizará en el apartado **2.3.3 Render Pipeline.**

2.3.2 Renderización volumétrica

La renderización volumétrica permite representar una proyección bidimensional de un conjunto de datos muestreados tridimensionalmente. Existen diferentes métodos para conseguir esto:

Nubes de puntos: Al muestrear discretamente un campo escalar, se obtiene un conjunto de puntos con un valor concreto asociado a cada una de las posiciones muestreadas. Estos puntos, que pueden usarse para generar mallas triangulares o como conjunto de datos para otros algoritmos, también pueden visualizarse directamente instanciando partículas en cada uno de los puntos (Ilustración 10). Este método conlleva la aparición de artefactos en la visualización, así como de franjas vacías entre los puntos (Ilustración 11). Cuanto más densa sea la nube de puntos, más habrá que "acercarse" para apreciar estos huecos.

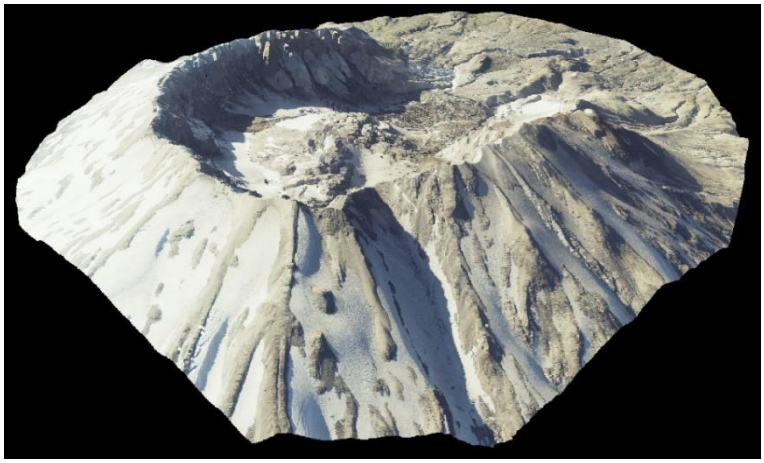


Ilustración 10: Nube de puntos del monte Santa Helena. Fuente: Elaboración propia. Archivo: St. Helens, Howard Butler, Hobu.Inc USGS LIDAR + NAIP

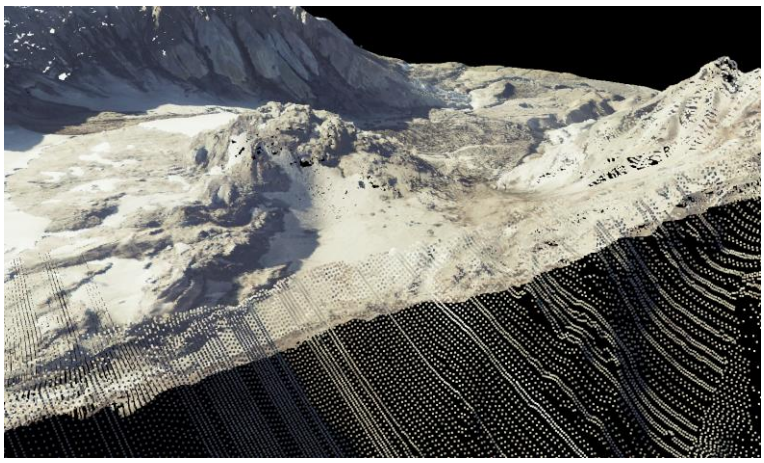


Ilustración 11: La misma nube de puntos vista desde cerca. Fuente: Elaboración propia. Archivo: St. Helens, Howard Butler, Hobu.Inc USGS LIDAR + NAIP

***Marching cubes*:** Presentado por W. E. Lorensen y H. E. Cline [15], este algoritmo permite extraer una malla poligonal a partir de un campo escalar (Ilustración 13). Consiste en:

- Dividir el espacio en voxels (pequeños cubos).
- Comprobar para cada cubo si sus vértices están dentro o fuera del volumen.
- Si en un cubo hay vértices dentro y fuera, significa que la superficie del volumen interseca con ese voxel. y se traza un polígono entre las aristas pertinentes.

En la Ilustración 12, se pueden ver los pasos reducidos a dos dimensiones (*marching squares*) con el objetivo de ilustrar los pasos del algoritmo.

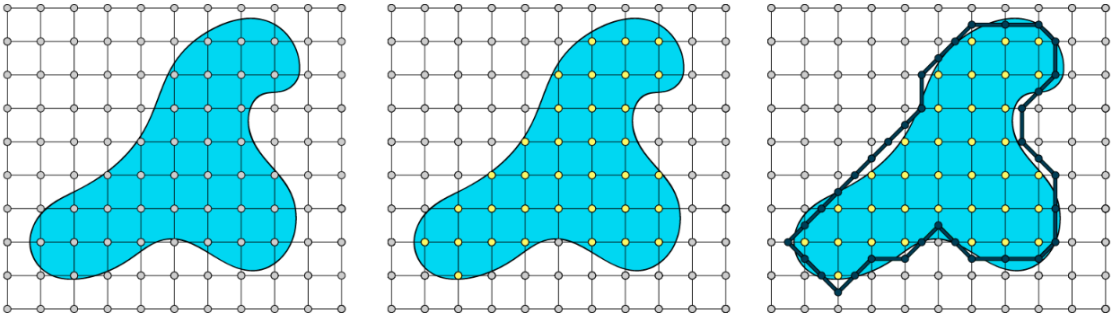


Ilustración 12: Simplificación bidimensional de los pasos del algoritmo. Fuente: Elaboración propia.

La calidad del resultado de este algoritmo depende en gran medida de la resolución que se elija para los voxels, es decir, cuantos más voxels haya, más fiel al campo escalar será la malla generada. Además, al generar una malla triangular, se puede someter fácilmente a técnicas de postprocesado como suavizados o diferentes segmentaciones.

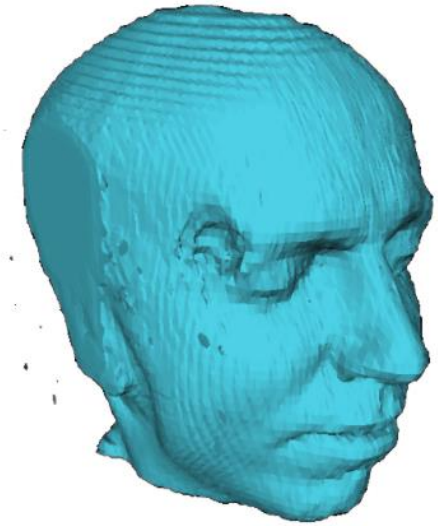


Ilustración 13: Malla resultante empleando marching cubes a partir de un archivo de imagen médica. Fuente: Modificado a partir de "MarchingCubes" de Dake [En línea] Disponible en: https://commons.wikimedia.org/wiki/File:Marchingcubes-head.png

Volumetric ray casting [16]: Este método es similar en concepto al *ray casting* tradicional. Se genera un rayo para cada píxel, con la diferencia de que en vez de parar o rebotar (*ray tracing*) al intersecar con uno de los triángulos de la malla, este atraviesa el volumen. Los pasos del algoritmo (Ilustración 14) se exponen a continuación:

- Se traza un rayo desde el origen de la cámara atravesando cada uno de los píxeles del sensor hasta el volumen, esta vez, a diferencia de en el ray casting tradicional, el rayo atraviesa el volumen (volumen constituido por datos discretos encerrados normalmente en paralelepípedos de ángulos rectos).
- Una vez en el interior del volumen, se toman n puntos equidistantes, situados sobre cada uno de los rayos trazados, después, se calculan los valores correspondientes a dichos punto por interpolación de los valores definidos más cercanos.
- Por último, se combinan estos valores y el del fondo en un proceso de *compositing* (Combinación visual de diferentes elementos en uno solo) para calcular el valor final que será asignado al píxel. Este proceso, se puede hacer de adelante hacia atrás o de atrás hacia adelante.

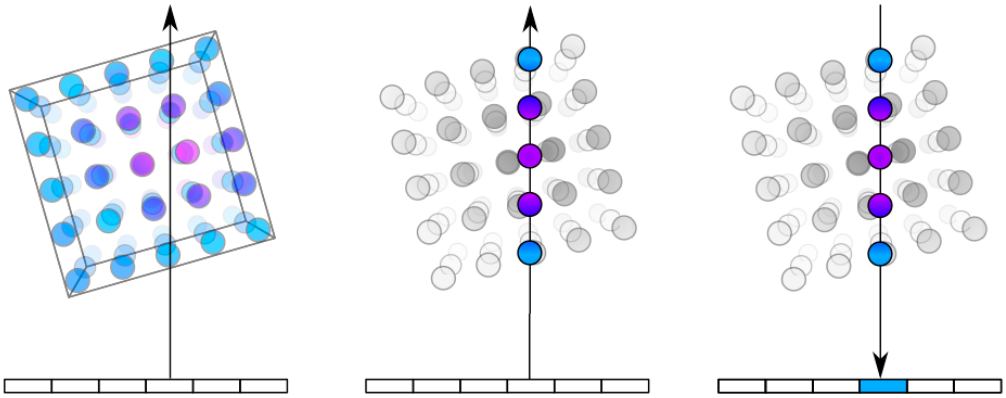


Ilustración 14: Pasos del algoritmo de volumetric ray casting. Fuente: Modificado a partir de "Volume ray casting" de Thetawave [En línea] Disponible en: https://commons.wikimedia.org/wiki/File:Volume_ray_casting.png

De esta forma, es posible obtener representaciones de muy alta calidad (Ilustración 15)

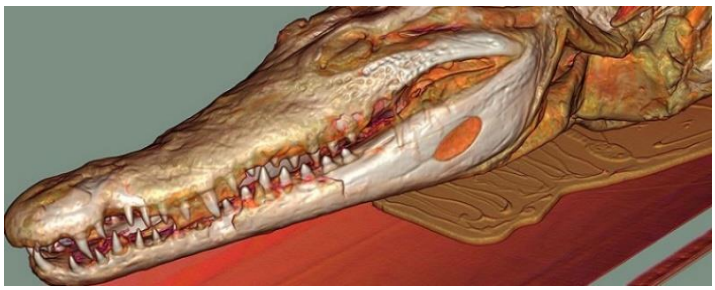


Ilustración 15: Renderizado empleando volumetric ray casting de la momia de un cocodrilo. Fuente: "Croc" de Stefanbanev [En línea] Disponible en: https://commons.wikimedia.org/wiki/File:Croc.5.3.10.a_gb1.jpg

2.3.3 Render pipeline

Se entiende por *render pipeline* la serie de etapas de procesamiento de las cuales se obtiene la imagen renderizada. A partir de la aparición de la aceleración por hardware, estos procedimientos se implementan directamente en la electrónica de las tarjetas gráficas, permitiendo la posibilidad de programar ciertas etapas a través de *shaders*, programas que se ejecutan en la tarjeta gráfica.

Este *pipeline* o tubería, difiere dependiendo de que API (Interfaz de programación de aplicaciones) de gráficos se use, (OpenGL, Direct3D, Vulkan, etc.), es por esto por lo que se expone a continuación una vista simplificada de los pasos que componen estos pipelines.

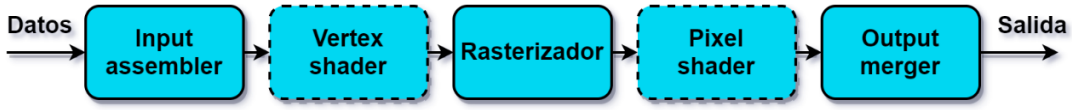


Ilustración 16: Pasos de un render pipeline simplificado (en línea de puntos las etapas programables). Fuente: Elaboración propia.

Las principales etapas son:

- ***Input assembler*** [17]**:** Se encarga de leer los datos (puntos, triángulos, líneas) almacenados en los *buffers* de memoria y de convertirlos en distintas primitivas geométricas que serán la entrada de las siguientes etapas.
- ***Vertex shader*** [18]**:** Este primer *shader* es el encargado de procesar los vértices provenientes del *Input assembler*, transformando sus atributos cuando sea pertinente. Tanto la entrada como la salida del *vertex shader* será siempre un único vértice cada vez.
- ***Rasterizador*****:** Empleando el algoritmo visto anteriormente, se generan los fragmentos de cada triangulo asociados a los pixeles de la imagen (Ilustración 17). En esta etapa también se interpola la información de los vértices cercanos y se calcula la profundidad del fragmento. Hay que recordar que a la salida de esta etapa, los pixeles aún **no** tienen un color definido.

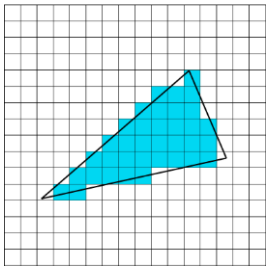


Ilustración 17: Fragmentos a la salida del rasterizador. Fuente: Elaboración propia.

- ***Pixel shader*** [19]: Segunda parte programable del *pipeline*, puede emplear variables, texturas, valores interpolados de los vértices cercanos y otros datos para generar un color asociado a cada fragmento que sale del *rasterizador*.
- ***Output merger*** [20]**:** Genera el color final que será mostrado en cada píxel usando una combinación del resultado del pixel shader y distintos elementos como máscaras o el buffer de profundidad.

2.4 Realidad virtual

Si se atiende a la definición provista por Fuchs, Moreau & Guitton [21], la realidad virtual (VR, *Virtual Reality*) es un dominio científico-técnico que emplea la informática y una serie de interfaces para simular el comportamiento de entidades tridimensionales en un entorno virtual, que interactúan entre ellas y con uno o más usuarios creando una sensación de inmersión. Dichas interfaces están compuestas por distintos sensores y actuadores, encargados de registrar las interacciones de los usuarios y de retroalimentarlas.

Es fundamental contar con una interfaz visual: Monitores, sistemas CAVE[4] (Ilustración 18), Visores montados en la cabeza (*Head Mounted Display*, HMD de aquí en adelante, Ilustración 19), etc. Pese a que todos estos tipos de pantallas entran dentro de la definición de realidad virtual, este documento se centrará en el uso de HMD (Visores montados en la cabeza).

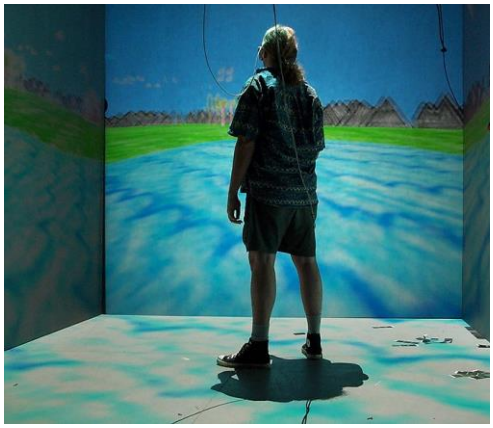


Ilustración 18: Usuario en un entorno de realidad virtual de tipo CAVE. Fuente: "CAVE" de Davepape [En línea] Disponible en: https://commons.wikimedia.org/wiki/File:CAVE_Crayoland.jpg

Ilustración 19: Investigador de la Agencia Espacial Europea (ESA) probando un prototipo de software de entrenamiento para astronautas en un HMD. Fuente: "Reality check" de European Space Agency [En línea] Disponible en http://www.esa.int/ESA_Multimedia/Images/2017/07/Reality_check

2.4.1 Distintos tipos de HMD

Dentro de los visores montados en la cabeza, se encuentran tres grandes grupos:

- Adaptadores para móviles: Este tipo de visores permiten el uso de un teléfono móvil a modo de pantalla y dependen de las capacidades de dichos dispositivos. Normalmente emplean mandos bluetooth con funcionalidades muy limitadas y registran movimientos de rotación valiéndose de los acelerómetros y giróscopos equipados en los smartphones. Debido a que básicamente son estructuras de plástico o cartón con las lentes adecuadas, estos equipos tienen un coste muy

[4] CAVE (*Cave Automatic Virtual Environment*) es un entorno inmersivo de realidad virtual que consiste en una habitación con proyectores dirigidos a las paredes, normalmente entre tres y seis de ellas.

bajo en comparación con los otros tipos de visores. Visores de este tipo son Google Cardboard (Ilustración 20) o las gafas VRBox, entre otros.

- Visores para uso en equipos de sobremesa: Este tipo de visores, asociados a lo que se conoce como realidad virtual de escritorio (*Desktop VR*) son los más populares. En comparación con los otros equipos, y gracias a que tienen acceso a los recursos computacionales y energéticos de ordenadores externos, estos dispositivos consiguen mejores características tanto visuales como funcionales. Esto es debido a que integran pantallas de mayor resolución, tienen un campo de visión mayor y pueden permitirse el uso de periféricos más sofisticados o de instalaciones de sensores fijos con las que conseguir registrar las interacciones de los usuarios de forma muy fiable. Entre estos dispositivos están, por ejemplo, el Valve Index o las Oculus Rift (Ilustración 20).

- Visores todo en uno: Por último, y con un prometedor futuro, se encuentran los visores que no dependen de ningún tipo de computador externo, este tipo de dispositivo incorpora las unidades de procesamiento necesarias para ejecutar aplicaciones directamente, cuenta con una batería, e integran en su interior los sensores necesarios para registrar los movimientos del usuario. Este tipo de HMD, a pesar de depender de una batería y de estar limitado por unidades de procesamiento móviles, permite un espacio de trabajo teóricamente ilimitado, sin restricciones debido a instalaciones alámbricas entre el dispositivo y el ordenador. Visores todo en uno son las Oculus Quest o el Vive Focus Plus (Ilustración 20).

Ilustración 20: De izquierda a derecha, Google Cardboard, Oculus Rift, Vive Focus Plus. Fuentes: "Google Cardboard" de Evan-Amos [En línea] Disponible en https://commons.wikimedia.org/wiki/File:Google-Cardboard.jpg ; "CV1" de Evan-Amos [En línea] Disponible en https://es.m.wikipedia.org/wiki/Archivo:Oculus-Rift-CV1-Headset-Front.jpg ; "Focus plus" de HTC Corporation [En línea] Disponible en https://enterprise.vive.com/mx/product/focus-plus/

2.4.2 Posición y orientación en la realidad virtual

Para conseguir una interacción en tiempo real entre el usuario y el sistema de realidad virtual, es necesario que los movimientos del usuario en el mundo real se registren y trasladen al entorno virtual.

Simplificando el concepto de grado de libertad (*Degree of Freedom*) como el número de parámetros independientes que definen la configuración espacial de un sistema, se puede decir que un cuerpo en el espacio tiene seis GDL, tres componentes para la orientación y tres componentes para la traslación. Dentro de los sistemas de realidad virtual se puede diferenciar entre:

- **Sistemas de seguimiento de tres grados de libertad (3 DOF):** Registran solamente cambios en la orientación (Ilustración 21) empleando acelerómetros y giróscopos, por lo que si bien es cierto que registran los movimientos angulares de la cabeza o los controladores, no permiten al usuario desplazarse dentro del entorno virtual, solamente mirar alrededor. Es por esto por lo que se han visto relegados a experiencias visuales donde no es necesario interactuar con el mundo virtual como la visualización de imágenes 360º o películas adaptadas a realidad virtual. Estos sistemas, han ido quedado obsoletos a medida que han avanzado las tecnologías de seguimiento, exceptuando su uso en dispositivos basados en teléfonos móviles.

- **Sistemas de seguimiento de seis grados de libertad (6 DOF):** Capaces de seguir los cambios en orientación y localización de los periféricos (Ilustración 21), permiten al usuario moverse dentro de un entorno virtual e interactuar con otras entidades, tanto simuladas como otros usuarios. Este tipo de seguimiento lo incorporan la gran mayoría de equipos de escritorio y todo en uno actuales tanto en los visores como en los controladores, empleando diferentes técnicas, expuestas a continuación.

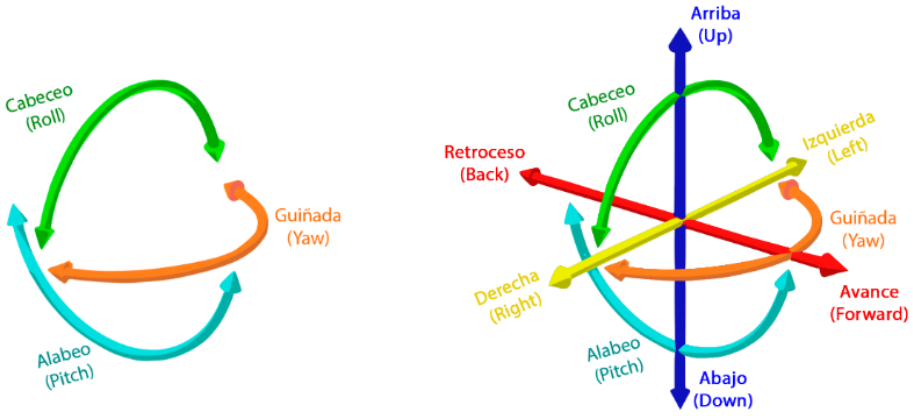


Ilustración 21: Diferentes magnitudes seguidas por sistemas de 3GDL (izquierda) y de 6GDL (Derecha). Fuente: Modificado a partir de "Seis grados de libertad" de Horia Ionescu [En línea] Disponible en: https://commons.wikimedia.org/wiki/File:6DOF_en.jpg

Dentro de los sistemas de seis grados de libertad, la gran mayoría de periféricos de realidad virtual actuales (visores y controladores), emplean una unidad de medida inercial (IMU) compuesta por una combinación de acelerómetros y giróscopos electromecánicos, funcionando típicamente a una frecuencia de 1000Hz para calcular los cambios en la orientación y localización del dispositivo empleando navegación por deriva (*Dead Reckoning*) [22] [23]. Esto se consigue integrando las lecturas de las aceleraciones a las que se ve sometido el dispositivo para calcular los incrementos en la posición. Pese a esto, las lecturas de los IMU son insuficientes y demasiado ruidosas debido a la doble integración como para registrar con la exactitud suficiente los movimientos del usuario, por lo que se emplean distintos métodos de corrección:

- **Seguimiento de fuera a dentro**: Este tipo de seguimientos requiere de la instalación de sensores externos, conectados o no a un ordenador, que se encargan de procesar información proveniente de los periféricos o de generar información que será procesada por estos. Los más comunes son:
    - **Sistemas de visión:** Los sensores exteriores son cámaras que operan bien en el espectro de la luz visible o en el espectro infrarrojo. En los dispositivos que se desean localizar se sitúan patrones de luces (Ilustración 22) conocidos por el software. De esta forma, las cámaras capturan las imágenes y las transmiten hasta el ordenador, donde se procesan y se identifican las posiciones de las luces y los patrones (*Led Matching*) [24]. Hay que tener en cuenta que estos sistemas conocen los datos de los IMU, y por lo tanto saben cuál es la aceleración y rotación del periférico, además de, normalmente, conocer la posición en el fotograma anterior, lo que facilita la tarea de localización del visor y los controladores. Este sistema es el empleado por PlayStation VR y la primera generación de Oculus Rift.

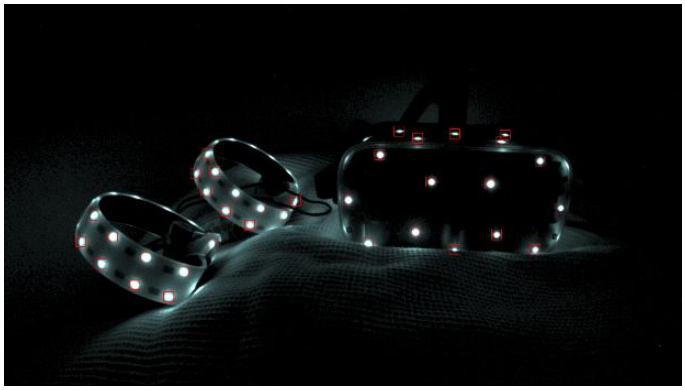


Ilustración 22: Leds en un visor Oculus y sus controladores vistos a través de una cámara infrarroja. Fuente: "cv1" de Open HMD [En línea] Disponible en http://www.openhmd.net/index.php/blog/

- **Sistemas de barrido de luz (*Lighthouse* de Valve)** [25]: Este método emplea unas balizas (*beacons*) infrarrojas que se deben instalar de forma fija, sin necesidad de conectarlas al ordenador. Estas balizas se iluminan y después barren un haz laser, proyectándolo sobre todas las superficies con las que se encuentra, primero horizontalmente, y después verticalmente, de forma periódica. Al contrario que en el caso anterior, los sensores, que son fotodiodos, están montados sobre el visor y los controladores. Estos sensores registran el paso de los haces laser y triangulan su posición dependiendo de los incrementos de tiempo que miden entre los barridos y la iluminación inicial. Este sistema fue diseñado por la empresa Valve para SteamVR y su visor HTC Vive, pero otros fabricantes lo han adoptado, como por ejemplo los visores de Varjo, empresa dedicada a la manufactura de HMDs para uso profesional en diferentes industrias.

- **Seguimiento de dentro a fuera:** Al contrario que en el seguimiento de fuera a dentro, los sensores que hacen posible la localización en el espacio de los dispositivos están incorporados en los mismos, normalmente en el visor.

  Para conseguir esto, se emplean técnicas de SLAM (*Simultaneous Localization and Mapping*) [26] [27] que fusionan la información proveniente de los sensores inerciales (IMU) del visor junto con información de las imágenes recogidas por las cámaras montadas en este, de donde se extraen puntos característicos del entorno que permiten calcular la orientación y posición del dispositivo,
  Para la localización de los controladores, se emplean las lectura de los IMU equipados en dichos periféricos en conjunto con sistemas de visión artificial de *Led Matching* como los vistos en el apartado de seguimiento de fuera a dentro [28], con la diferencia de que está vez se emplean las cámaras que equipa el visor para detectar los patrones de luz en vez de cámaras externas.

  Este tipo de seguimiento, pese a (de momento) ofrecer resultados ligeramente inferiores al seguimiento de fuera a dentro en términos de precisión, evitan la instalación de sensores exteriores y facilitan el transporte de los dispositivos. Se puede encontrar en visores más modernos como las Oculus Rift S o en los visores del tipo todo en uno.

2.4.3 Controladores y otros periféricos

Con el objetivo de posibilitar la interacción natural con el entorno virtual, se emplean controladores, ya mencionados anteriormente. Lo más normal es que estos controladores tomen la forma de mandos, que a través de sensores capacitivos, botones, *joysticks*, paneles táctiles, etc., registran tanto interacciones discretas: la pulsación de un botón, como interacciones "continuas": la distancia de los dedos a los mandos o la fuerza que se está ejerciendo con cada dedo.

Existen otros dispositivos, como guantes hápticos, que pueden ofrecen al usuario una retroalimentación del entorno virtual, por ejemplo una resistencia física al sujetar un objeto virtual, o dispositivos diseñados específicamente para entornos de simulación con los que se consiguen altos niveles de fidelidad en comparación con la actividad real.

Cabe mencionar también el auge del uso de dispositivos que rastrean las manos del usuario (*Hand tracking*) y emplean esta información para registrar el movimiento y las interacciones sin necesidad de usar otros controladores. Esto libera las manos del usuario y permite una interacción más natural e intuitiva. Para lograr esto existen dispositivos que se pueden añadir a los visores como son los sensores de Ultraleap (anteriormente Leap Motion), aunque algunos HMD ya lo implementan empleando las cámaras que equipa el propio visor (Oculus Quest).

2.4.4 Problemas de la realidad virtual

El uso prolongado de tecnologías de realidad virtual, con una fuerte dependencia de lo que se esté visualizando a través del visor y del propio usuario, puede llegar a causar malestar, conocido como *cybersickness* [29]. Este malestar puede venir acompañado de síntomas como nauseas, mareos, dolor de cabeza o desorientación.

Las principales causas son:

- Movimiento no correspondido, bien porque es lo que se pretende (ej: Simulador de conducción) o por un retardo entre los movimientos del usuario y la respuesta del sistema.
- Aspectos técnicos, como puede ser una tasa de refresco en las pantallas inadecuada.

Pese a que es un problema que hoy en día continúa siendo estudiado, se puede paliar, respectivamente, con:

- Distintas formas de transporte, como puede ser el teletransporte [30], o la introducción de un marco estático de referencia en la escena [31].
- Cumplimiento de los requisitos computacionales mínimos para el correcto funcionamiento del sistema y su correcta configuración.

2.4.5 Realidad virtual en el ámbito médico

El uso de dispositivos y entornos de realidad virtual en el ámbito médico pretende optimizar costes, mejorar la calidad de la educación y las terapias [32], además de permitir nuevas técnicas más eficientes y seguras en distintos campos, como son:

- **Entrenamiento:** El entrenamiento quirúrgico y de profesionales médicos actual presenta una relación directa entre su realismo y su complejidad, coste y riesgos. El uso de la realidad virtual para práctica y desarrollo de habilidades se presenta como un escalón intermedio que a través de dispositivos equipados con tecnologías hápticas, retroalimentan al usuario sensaciones similares a las que se experimentaría en una práctica real, a la vez que están inmersos visualmente en dicho entorno gracias al uso de un HMD. Este tipo de prácticas han sido bien acogidas en diferentes estudios [33], permitiendo a los usuarios la repetición de procesos y actividades, dejando de lado las limitaciones espaciales (lugar donde se realiza la práctica o entrenamiento) o de recursos (disponibilidad de materiales para realizar las prácticas, como por ejemplo maniquíes de práctica o animales).

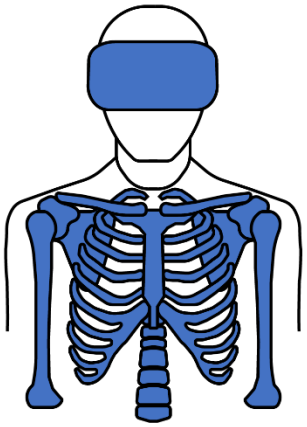
- **Rehabilitación:** Los procesos de rehabilitación, se pueden ver beneficiados del uso de visores de realidad virtual al crear un entorno donde realizar los ejercicios correspondientes. Pacientes sujetos a este tipo de tratamiento aplicado a rehabilitación pulmonar, declaran un aumento de motivación al realizar los ejercicios en un entorno virtual y una sensación de satisfacción al completarlos [34].

- **Tratamiento de fobias:** Ha sido exitoso también el uso de tecnologías de realidad virtual en psicoterapia para el tratamiento de fobias como el miedo a lugares cerrados o el miedo a volar [35], creando un entorno en el que el paciente puede enfrentarse a diferentes grados del estímulo fóbico, añadiendo la posibilidad de terminar la sesión de forma inmediata en cualquier momento.

- **Visualización y planificación:** Dejando de lado los aspectos hápticos de la exploración, ya que simular las respuestas de los tejidos es realmente complicado y depende en gran medida de cada paciente, la realidad virtual puede emplearse para visualizar las imágenes médicas recogidas. Es posible explorarlas o estudiar diferentes vías de acceso para las operaciones. También puede emplearse para explicar al paciente su patología o el procedimiento al que se va a someter de una forma más intuitiva y visual.

Si clasificásemos la aplicación desarrollada en estos campos, sería una base a partir de las cuales se podrían construir aplicaciones tanto de entrenamiento/educativas, como de visualización y planificación.

Antes de tratar el desarrollo de esta aplicación, se comentarán en el siguiente capítulo las herramientas empleadas durante este proceso.



Capítulo 3: Herramientas empleadas

En este capítulo, se comentarán las distintas herramientas, tanto Software como Hardware, que se han usado durante el desarrollo de este trabajo de fin de grado.

3.1 Software

3.1.1 Unity3D

Desarrollado por Unity Technologies, Unity3D (Ilustración 23) es una plataforma gratuita (para uso no comercial) de desarrollo de aplicaciones en tiempo real. Está orientado al desarrollo de experiencias interactivas y emplea el lenguaje C# como herramienta de desarrollo. Para este trabajo de fin de grado, se ha empleado la versión 2018.3.11f1.

Ilustración 23: Logo de Unity3D. Fuente: "UnityLogo" de Unity Technologies [En línea] Disponible en https://unity.com/es

3.1.2 VRTK

Virtual Reality Toolkit (VRTK) [36] (Ilustración 24) es un conjunto de herramientas desarrolladas para Unity3D que facilitan el uso e implementación de distintas soluciones enfocadas a entornos espaciales como los que se dan en la realidad virtual y aumentada. Está publicada bajo la licencia MIT de software libre y de uso gratuito. En este proyecto se ha empleado la versión VRTK v4.

Ilustración 24: Logo de VRTK. Fuente: "VRTKLogo" de ExtendRealityLtd [En línea] Disponible en https://github.com/ExtendRealityLtd/VRTK

3.1.3 EvilDICOM

Publicada bajo la licencia MIT de software libre, EvilDICOM [37] (Ilustración 25) es una librería de C# destinada a facilitar la lectura y manipulación de archivos DICOM, ofreciendo un interfaz a través del cual acceder a los atributos de este tipo de archivos.

Ilustración 25: Logo de EvilDICOM. Fuente: "EvilDICOMLogo" de Rex Cardan [En línea] Disponible en https://github.com/rexcardan/Evil-DICOM

3.2 Hardware

3.2.1 Visor de Realidad Virtual

El visor elegido para el desarrollo y prueba del proyecto ha sido el Oculus Rift S (Ilustración 26), de la empresa Oculus, proporcionado por el ITAP (Instituto de las Tecnologías Avanzadas de la Producción). Este HMD cuenta con las siguientes características:

- Visor para equipos de sobremesa
- Seis grados de libertad gracias a un sistema de seguimiento dentro a fuera (Oculus Insight).
- Visera de tipo halo para máxima comodidad.
- Controladores Touch.
- Pantallas LCD 1280x1440 @ 80Hz

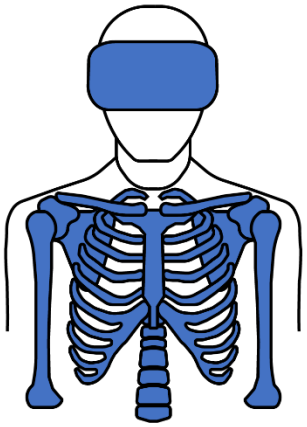
Ilustración 26: Visor Oculus Rift S. Fuente: "Oculus Rift S" de Oculus VR [En línea] Disponible en https://www.oculus.com/rift-s/

3.2.2 Ordenador

El ordenador empleado es un ordenador portátil MSI GL72VR 7RFX, cuyas especificaciones son las siguientes:

- Procesador: Intel Core i7 7700HQ @ 2.80 GHz
- Tarjeta gráfica dedicada: NVIDIA GeForce GTX 1060, 6 GB de VRAM GDDR5
- Memoria RAM: 16 GB DDR4
- Sistema operativo: Windows 10

Con estos equipos, se ha llevado a cabo tanto el desarrollo, en el que se profundiza en el siguiente capítulo, como las pruebas de la aplicación.



Capítulo 4: Desarrollo

Este capítulo compone sus secciones a partir de los diferentes subsistemas en los que se ha dividido el desarrollo del trabajo de fin de grado, incluyendo su lógica, componentes, relaciones entre ellos, etc. Además de esto, se presenta también un último apartado donde se exponen y comentan los resultados obtenidos de este desarrollo.

4.1 Introducción

Se introducen a continuación, antes de profundizar en el desarrollo del proyecto, tanto una serie de conceptos que se consideran de importancia para facilitar la comprensión de los distintos apartados de este capítulo, como un diagrama que indica el nombre de cada uno de los botones de los controladores usados (Ilustración 29).

4.1.1 Términos

- Editor: Entorno gráfico de Unity (Ilustración 27), permite abrir diferentes pestañas, como pueden ser la jerarquía de objetos, los archivos del proyecto, el inspector de objetos, ventana de iluminación, de animación, de audio, etc.

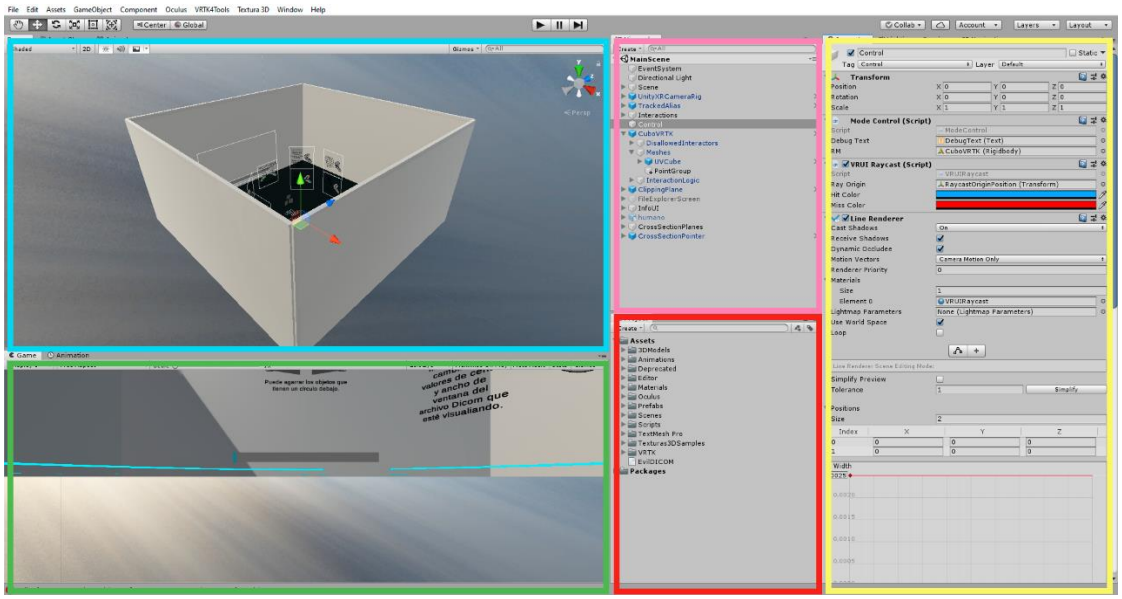


Ilustración 27: Editor de Unity. (Recuadros: Azul-Escena, Verde-Vista de la cámara, Rosa-Jerarquía, Rojo-Archivos del proyecto, Amarillo-Inspector). Fuente: Elaboración propia.

- Escena: Entorno virtual dentro de Unity, aquí se encuentran los objetos, modelos 3D, comportamientos definidos, etc.

- GameObject: Elementos base de Unity, todos los elementos dispuestos en la escena son GameObjects, que contienen distintos componentes.

- Componente: Cada uno de los elementos que definen el comportamiento de un GameObject. Hay una serie de componentes predefinidos en Unity que añaden distintas funcionalidades al GameObject (mostrar texto o imágenes, detectar colisiones, verse afectado por las leyes físicas, etc.). Estos componentes también pueden

programarse, pudiendo añadir a los GameObjects los comportamientos que se definan en archivos escritos en C#.

- Transform: Los GameObjects siempre cuentan con un componente Transform, que almacena y permite modificar su posición en la escena (x, y, z), su orientación (representada para el usuario como ángulos de Euler, internamente en cuaternios), y su escala. Los componentes Transform pueden tener también padres e hijos, configurables en la jerarquía del editor de Unity y vía código. En la Ilustración 28 se puede apreciar la relación padre – hijo entre el GameObject "CuboVRTK" y los GameObjects anidados en su interior.

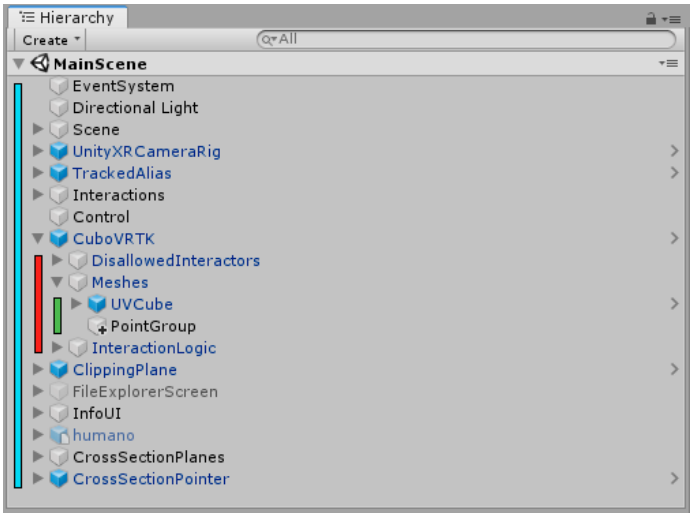


Ilustración 28: Detalle de la jerarquía de Unity. Fuente: Elaboración propia.

- RigidBody: Componente encargado de hacer que el GameObject al que se asocia se vea influido por el sistema de físicas de Unity. De esta forma se puede dotar al objeto de masa, inercia, fuerzas que actúan sobre él, restricciones de movimiento, etc.

- Textura (2D/3D): Es un conjunto de datos en forma de matriz. Puede ser bidimensional, una imagen en la que se proyectan los triángulos de las mallas, o tridimensional, en cuyo caso son necesarias técnicas especiales para visualizarlas.

- Lepio: Nombre interno de la aplicación. Pese a que no se hará referencia directa a este nombre, aparece en el código y en la documentación desarrollada.

- Material: Los materiales definen el shader que usará un determinado objeto y las variables de dicho shader. Unity proporciona una serie de shaders, entre ellos el "*Standard Shader*", que permite simular distintas superficies y comportamientos de la luz. También se pueden escribir shaders empleando HLSL (*High-Level Shader Lenguage*) que se ejecutaran directamente en la GPU permitiendo modificar los comportamientos de la renderización.

- *Prefab*: GameObjects guardados con una serie de componentes y configuraciones que pueden instanciarse fácilmente. Cualquier cambio realizado en el prefab original, se verá reflejado en las instancias de este.

- *Canvas*: Componente que convierte un GameObject en la base para situar todo tipo de elemento de UI (*User Interface* – Interfaz de usuario) como texto, imágenes, botones, etc.

- Inspector: El inspector es una ventana de Unity que permite modificar las propiedades (públicas o serializadas) de los componentes asociados a un GameObject, así como añadir o eliminar componentes.

- Sistema de eventos: Emplea sucesos (eventos) en una serie de objetos publicadores (*Publisher*) que al ocurrir invocan una serie de métodos o funciones en otros objetos a la escucha de dichos sucesos (*Listeners*).

4.1.2 Distribución de los botones en los controladores

La botones que equipan los controladores (Oculus Touch) del equipo Oculus Rift S son los siguientes (Ilustración 29):

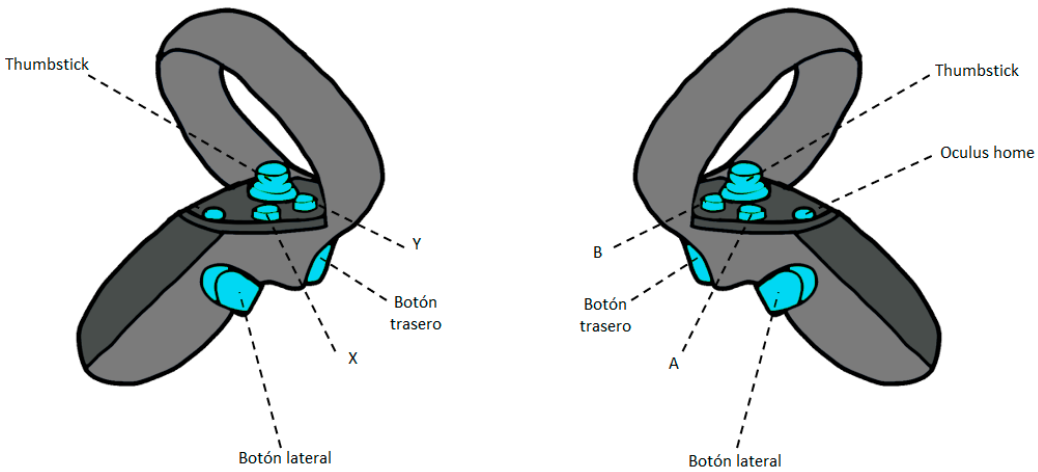


Ilustración 29: Botones en los controladores Oculus Touch. Fuente: Elaboración propia.

4.2 Entorno virtual

4.2.1 Entorno 3D

El entorno tridimensional en el que se desarrolla la aplicación está delimitado por cinco prismas cuadrangulares, que hacen las veces de paredes y suelo (Ilustración 30). Los materiales asociados a estos elementos estructurales son cercanos al blanco y al negro, buscando un contraste máximo con las reconstrucciones. También se ha cambiado la textura del cielo (Ilustración 31) por un archivo publicado bajo licencia CC0 por el estudio RUST LTD.

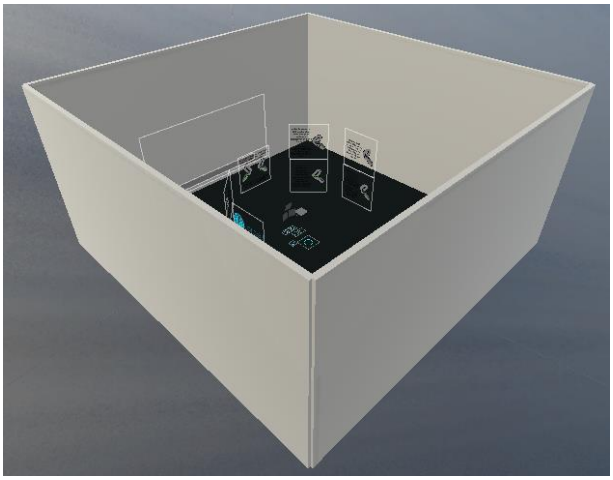


Ilustración 30: Entorno visto desde fuera. Fuente: Elaboración propia.

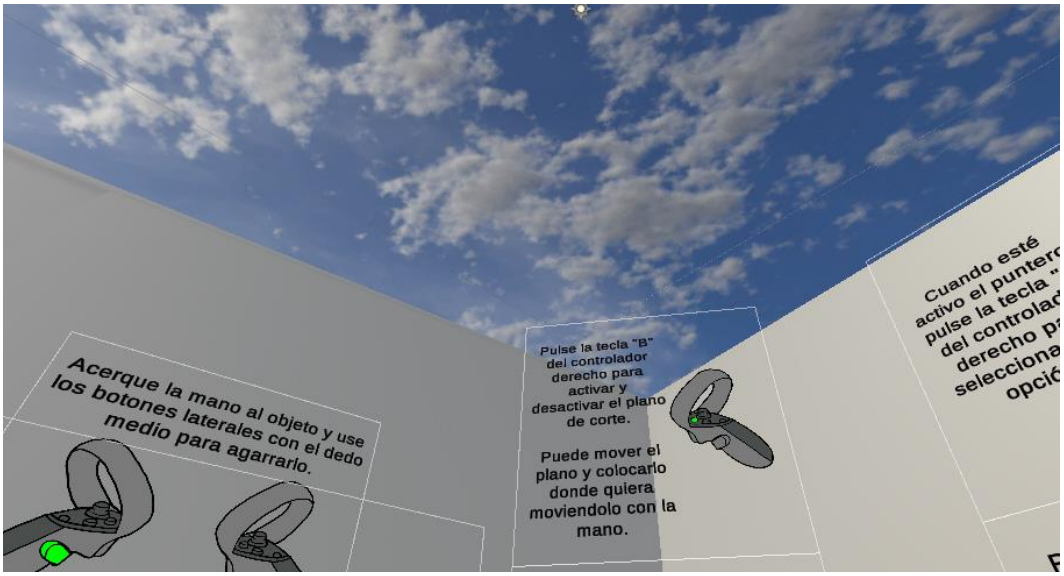


Ilustración 31: Entorno visto desde el punto de vista del usuario (mirando hacia arriba). Fuente: Elaboración propia.

4.2.2 Marcadores de posición

Se han dispuesto marcadores circulares en el suelo que señalan la posición tanto de las herramientas y la reconstrucción tridimensional del archivo como del propio usuario (marcador más grande) (Ilustración 32). Esto se consigue con un script encargado de actualizar cada frame el Transform de cada GameObject marcador, igualando sus coordenadas "x", "z" a las del objeto a seguir y manteniendo la "y" constante a la altura del suelo. También se modifica la orientación con respecto del eje "y" para simular un efecto de rotación continuo y hacerlo de esta forma más visual. El objetivo de estos marcadores es distinguir claramente los objetos con los que se puede interactuar, además de mantener un punto de visibilidad para cuando las herramientas se encuentren ocluidas y poder localizarlas fácilmente.

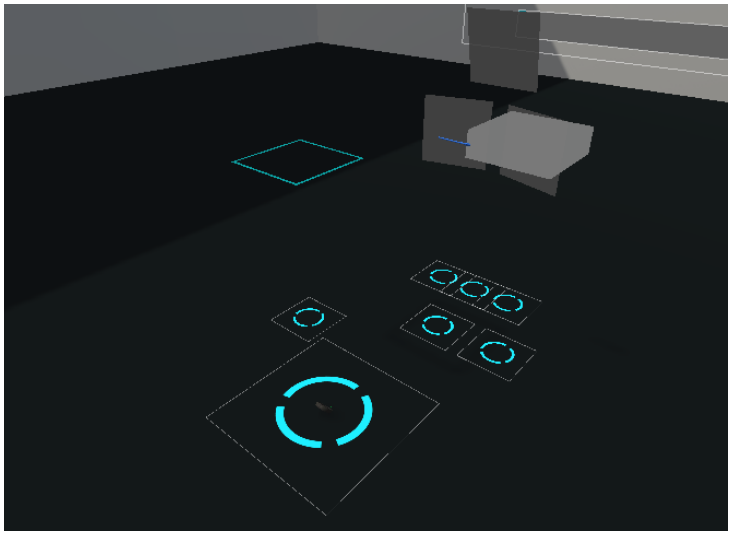


Ilustración 32: Marcadores de posición en el suelo del entorno. Fuente: Elaboración propia.

4.2.3 Barra de carga

A modo de indicación durante los procesos de generación del modelo a partir de un archivo DICOM (apartado **4.4 Generación del modelo tridimensional**), se ha dispuesto en el entorno una barra de carga (Ilustración 33) que se muestra más o menos llena en función del progreso realizado y el que queda por realizar. Esto se consigue con un componente que actualiza las dimensiones de una barra superpuesta sobre otra dependiendo de los valores que reciba a través de la llamada a una de sus funciones.

Ilustración 33: Barra de carga en el entorno virtual de la aplicación. Fuente: Elaboración propia.

4.3 Menú radial

Al tocar con el dedo el Thumbstick (acción registrada por los sensores capacitivos del mando), se despliega un menú radial de seis opciones (Ilustración 34), de las cuales están en uso cuatro. Al mover el Thumbstick con el dedo, se determina sobre que sección del menú se encuentra el usuario dependiendo de las coordenadas x e y del Thumbstick. Al situarse sobre cada una de las opciones, esta se resalta para indicar preselección y muestra un pequeño mensaje que indica su función (Ilustración 35). Para elegir la opción preseleccionada, es necesario apretar el botón del Thumbstick.

Ilustración 34: Menú radial desplegado. Fuente: Elaboración propia.

Ilustración 35: Menú radial con una opción preseleccionada y texto de información. Fuente: Elaboración propia.

Para crear este menú radial, se ha diseñado en 3D el modelo de la sección seleccionable, para las cuales se ha creado una animación de activación y una textura con un icono adecuado a la función que desempeña.

Se ha desarrollado para el menú un sistema de eventos que se disparan al preseleccionar, al dejar de preseleccionar y al seleccionar cada una de las opciones, permitiendo de esta forma configurar de forma rápida las acciones que realiza cada opción en dichos sucesos desde el inspector de Unity sin tener que modificar el código (Ilustración 36).

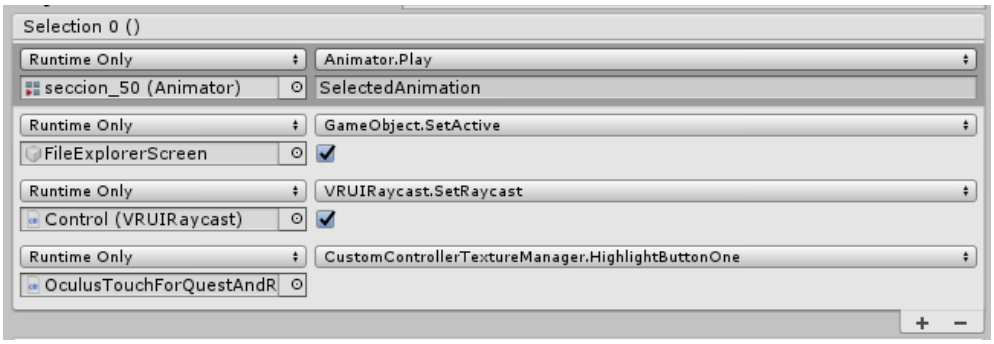


Ilustración 36: Detalle de las funciones configuradas a la escucha del evento asociado a la selección de la primera opción del menú radial (Abrir explorador de archivos). Fuente: Elaboración propia.

Las distintas opciones de este menú son:

4.3.1 Abrir explorador de archivos

Al elegir esta opción, se abre un panel (Ilustración 37) a modo de explorador de archivos creado para la aplicación y se activa un puntero que permite interactuar a distancia con los elementos de dicho panel, estas dos acciones se pueden ver en el sistema de eventos de la Ilustración 36. Este explorador de archivos, permite visualizar las distintas carpetas que hay en el equipo, preseleccionar una de ellas y confirmar la opción a través de un botón de "Seleccionar". También puede interrumpirse la selección empleando el botón "Cancelar" o la cruz de la esquina superior derecha. En caso de que se seleccione una carpeta, se pasa a través de una llamada a LoadTexture() (función de la que se hablará en el siguiente apartado) su ruta hasta el proceso de generación. No se profundizará en el desarrollo del subsistema asociado al funcionamiento del puntero y su interacción con los elementos del explorador de archivos, ya que es relativamente complejo y se centraría en el desarrollo de componentes de Unity que escapan del alcance de este documento.

Ilustración 37: Explorador de archivos en Unity. Fuente: Elaboración propia.

4.3.2 Alternar entre modos de manipulación

Como se profundizará más adelante en el apartado **4. 6 Manipulación del modelo** es posible trasladar, rotar, y escalar la reconstrucción tridimensional empleando los controladores. Al activar esta opción se actualizan las restricciones del RigidBody asociado a la reconstrucción, alternando entre los siguientes modos:

- Libre: Modo por defecto, permite trasladar, rotar y escalar la reconstrucción.
- Traslación: En este modo, la rotación se encuentra bloqueada, por lo que al agarrar la reconstrucción con una mano solo podrán verse afectados sus valores x, y, z.

- Rotación: De forma similar al anterior, esta vez se bloquea la traslación, por lo que solo será posible cambiar la orientación de la reconstrucción al agarrarla con una mano.
- Escala: Este modo bloquea tanto la traslación como la rotación, por lo que agarrar con una mano no modificara ni posición ni orientación, siendo solo posible cambiar la escala (agarrando con las dos manos).

4.3.3 Reestablecer posiciones iniciales

Activar esta opción devuelve a su posición inicial todos los objetos con los que se puede interactuar. El objetivo de esta opción es que exista una forma de volver a localizar las herramientas si se pierden de vista o llegan a algún punto de difícil acceso. Para esto, un script almacena las posiciones iniciales de un conjunto previamente configurado con los GameObjects a reestablecer. Al invocar dicha funcionalidad, se recorre el conjunto, modificando la posición actual de cada GameObject por la inicial almacenada.

4.3.4 Activar/Desactivar paneles de información

Se han creado para la aplicación una serie de paneles informativos (Ilustración 38). Estos paneles exponen los diferentes controles, resaltando el botón que hay que apretar, con el objetivo de facilitar el uso de la aplicación. Las imágenes se han creado en Adobe Illustrator y el texto es un componente de Unity, lo que facilita su cambio (por ejemplo para traducciones) sin tener que volver al programa de edición fotográfica. Cada uno de estos controles se comentará en su respectivo apartado.

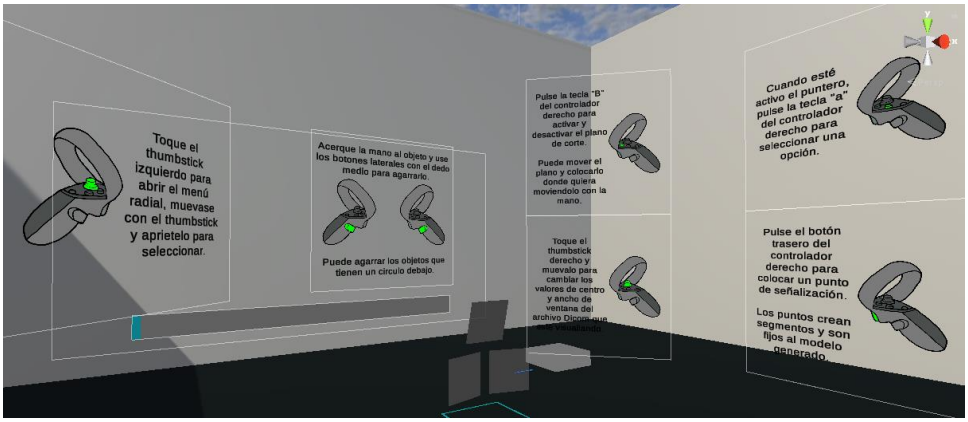


Ilustración 38: Paneles informativos con los controles en el interior de la aplicación. Fuente: Elaboración propia

Una vez comentadas las distintas opciones del menú y la interfaz de usuario de la aplicación, se trata a continuación el proceso de generación del modelo tridimensional partiendo del archivo a visualizar.

4.4 Generación del modelo tridimensional

El objetivo de este proceso es, a partir de una ruta seleccionada en el explorador de archivos interno, generar una textura tridimensional, es decir, una matriz de dimensiones f * c * n, donde f y c son las filas y columnas de cada una de las secciones del archivo DICOM, y n son el número de secciones del archivo. Esta textura, será la que se pase a la etapa de visualización, donde se renderizará su información.

4.4.1 Proceso general

Primero, se comentará el funcionamiento del proceso general de generación del modelo (Ilustración 39), para después profundizar en las funciones más complejas de este.

Tal y como se ha visto en el apartado **4.3.1 Abrir explorador de archivos**, al seleccionar una carpeta desde el explorador de archivos, se invoca el método `LoadTexture()` del componente `Texture3DGenerator` con la ruta de dicha carpeta como argumento.

Una vez dentro de `LoadTexture()`, se llama a la función `FileType` (miembro también de `Texture3DGenerator`), que extrae la extensión de los archivos localizados en la carpeta con la siguiente lógica:

- Si todos los archivos comparten extensión, se devuelve dicha extensión.
- Si hay archivos con diferentes extensiones, se devuelve “Multiple”.
- Si la carpeta está vacía, se devuelve “Empty”.

Esta respuesta, se sanea pasando la cadena de texto a minúsculas, evitando así problemas con comparaciones de cadenas de texto en mayúsculas.

En caso de que la extensión extraída sea “.dcm”, esto significará que la carpeta contiene archivos DICOM, mientras que si es “.jpg” o “.png” serán imágenes. En cualquiera de los otros dos casos, “Multiple” o “Empty”, se imprime un mensaje de error explicando el problema ocurrido al cargar el archivo en los registros de errores (logs).

Si bien es cierto que la reconstrucción tridimensional a partir de imágenes está implementada en la aplicación, fue parte de las etapas iniciales del proyecto, quedando obsoleta al introducir las funcionalidades de DICOM. Es por esto por lo que **el documento se centrará en la reconstrucción a partir de archivos DICOM**, siendo la reconstrucción a partir de imágenes una tosca simplificación del proceso en la que se pierde tanto la información almacenada en las etiquetas (distancia entre secciones, relaciones de aspecto, etc.), como la posibilidad de modificar el rango de valores visualizados.

A continuación, en el caso de tener extensión .dcm, se invoca a la función `LoadAsynchronousFromDicom()` con la ruta de la carpeta elegida como argumento. Esta llamada se hace a través de `StartCoroutine`, permitiendo así que la generación de la textura tridimensional sea *asíncrona*[5]. Si este proceso no se llevase a cabo de forma asíncrona, durante el tiempo que se estuviesen leyendo archivos y generando la textura, las pantallas del visor quedarían congeladas, lo que podría originar incomodidad y mareos.

Para poder invocar una función a través de `StartCoroutine`, es necesario que esta incluya en algún momento la palabra clave `yield`, que se encargará de "pausar" su ejecución, liberando de esta forma el procesador y permitiendo que el resto de procesos se ejecuten (como por ejemplo los procesos encargados de registrar movimiento, refrescar las pantallas, etc.). Una vez finaliza el procesamiento del frame, la ejecución vuelve al punto en el que quedó, repitiéndose el proceso.

La función `LoadAsynchronousFromDicom()`, cuyo proceso se describe en el apartado **4.4.2 LoadAsynchronousFromDicom**, genera un array de Texturas2D que contienen la información necesaria para la representación. Una vez generado, se pasa como argumento a la función `Load3DFrom2DArrayAsync()`, que se invoca a través de `StartCoroutine()` para conseguir una vez más una ejecución asíncrona.

Esta función, descrita en el apartado **4.4.4 Load3DFrom2DArrayAsync**, es la encargada de convertir el array de texturas2D en una textura3D.

Una vez la textura3D está lista, se invoca la función `UploadTexture3D()` con dicha textura como argumento, que se encargará de configurar la textura3D generada como valor en las propiedades de los shaders pertinentes, de forma que estos puedan acceder a su información para el proceso de visualización.

A continuación, tal y como se ha mencionado anteriormente, se comenta con mayor detalle el funcionamiento de las partes más importantes de este proceso general.

[5] Un proceso síncrono permanece a la espera de su resultado, mientras que un proceso asíncrono, permite la ejecución de otros procesos mientras este está ejecutándose.

Proceso general

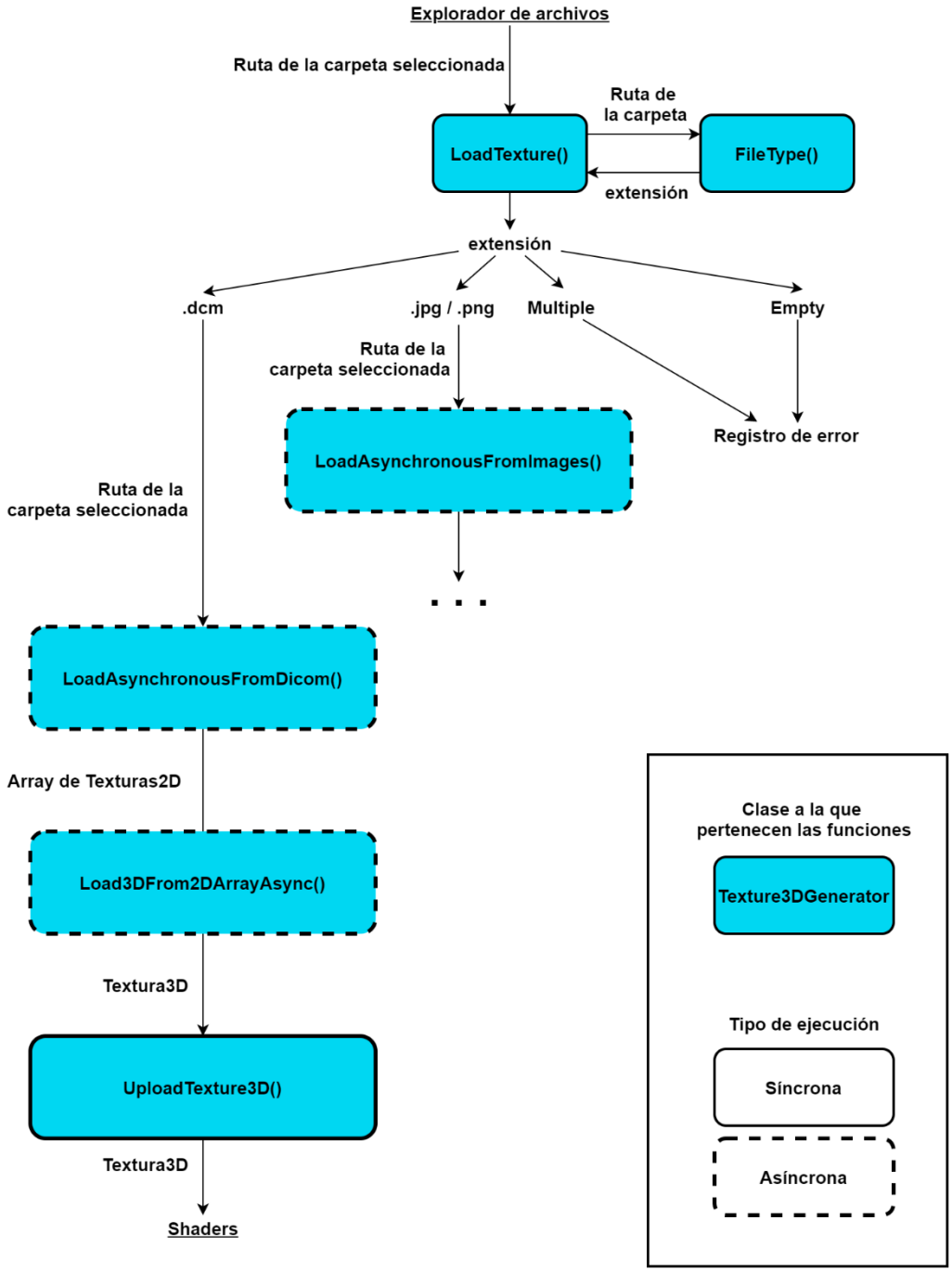


Ilustración 39: Esquema del proceso general de generación del modelo tridimensional. Fuente: Elaboración propia.

4.4.2 LoadAsynchronousFromDicom

La función `LoadAsynchronousFromDicom()`, recibe la ruta de la carpeta seleccionada y genera un array de texturas2D, que pasará como argumento a la función `Load3DFrom2DArrayAsync()`. Para esto lleva a cabo los siguientes pasos (Ilustración 40):

Primero, realiza una llamada a la función `GetDicomPaths()`, que devuelve un array que contiene las rutas de todos los archivos ubicados en la carpeta que se le pase como argumento (las secciones de resonancias y tomografías pueden estar tanto agrupadas en un solo archivo .dcm como cada una en un .dcm distinto).

Después, se instancia una lista de Texturas2D y un objeto de la clase `DicomLoader` vacío, de la que se hablará más adelante. A continuación, se itera sobre cada una de las rutas de los archivos del interior de la carpeta:

Para cada ruta, se asigna al objeto de la clase `DicomLoader` un nuevo objeto cuyo constructor se invoca con la ruta pertinente como argumento. Este constructor emplea funciones de la librería `EvilDICOM` para leer el archivo y almacenarlo en memoria, además, llama a `RefreshTags()`, función miembro de `DicomLoader`, que actualiza los campos de la instancia de la clase asociados a las etiquetas DICOM que interesa conocer para generar correctamente las texturas. La lectura de estas etiquetas, debido a que algunas pueden no estar presentes, se ha estructurado en bloques try-catch, de forma que si algún valor no está disponible salte una excepción, se registre el error y se asigne un valor por defecto adecuado a la etiqueta.

Una vez que el objeto de la clase `DicomLoader` tiene toda la información necesaria, se invoca su función `LoadDicom()`, que devuelve un array de Texturas2D. Dentro de `LoadDicom()`, se crea un array de Texturas2D con tantos elementos como secciones tenga el archivo Dicom[6] (Etiqueta "Número de frames (0028, 0008)"). Cada una de las texturas en este array, se definen a través de su respectiva llamada a la función `GenerateColors32()`, cuyo funcionamiento se especifica en el apartado **4.4.3 GenerateColors32**. Una vez generada la textura de cada archivo, se añaden al array de texturas, que será el que después de convierta en una textura3D.

[6] Hay que recordar que un archivo DICOM puede estar definido por un solo archivo de múltiples secciones o múltiples archivos de una sola sección.

Después de la generación de cada una de las texturas, se actualiza la barra de carga en función del archivo que se haya cargado y del número total de archivos en la carpeta y se usa la palabra clave yield para permitir que la ejecución sea asíncrona.

Una vez ha finalizado el bucle que itera sobre los archivos ubicados en la carpeta, se invoca a través de `StartCoroutine()` la función `Load3DFrom2DArrayAsync()`, pasándole como argumento el array de texturas2D generado.

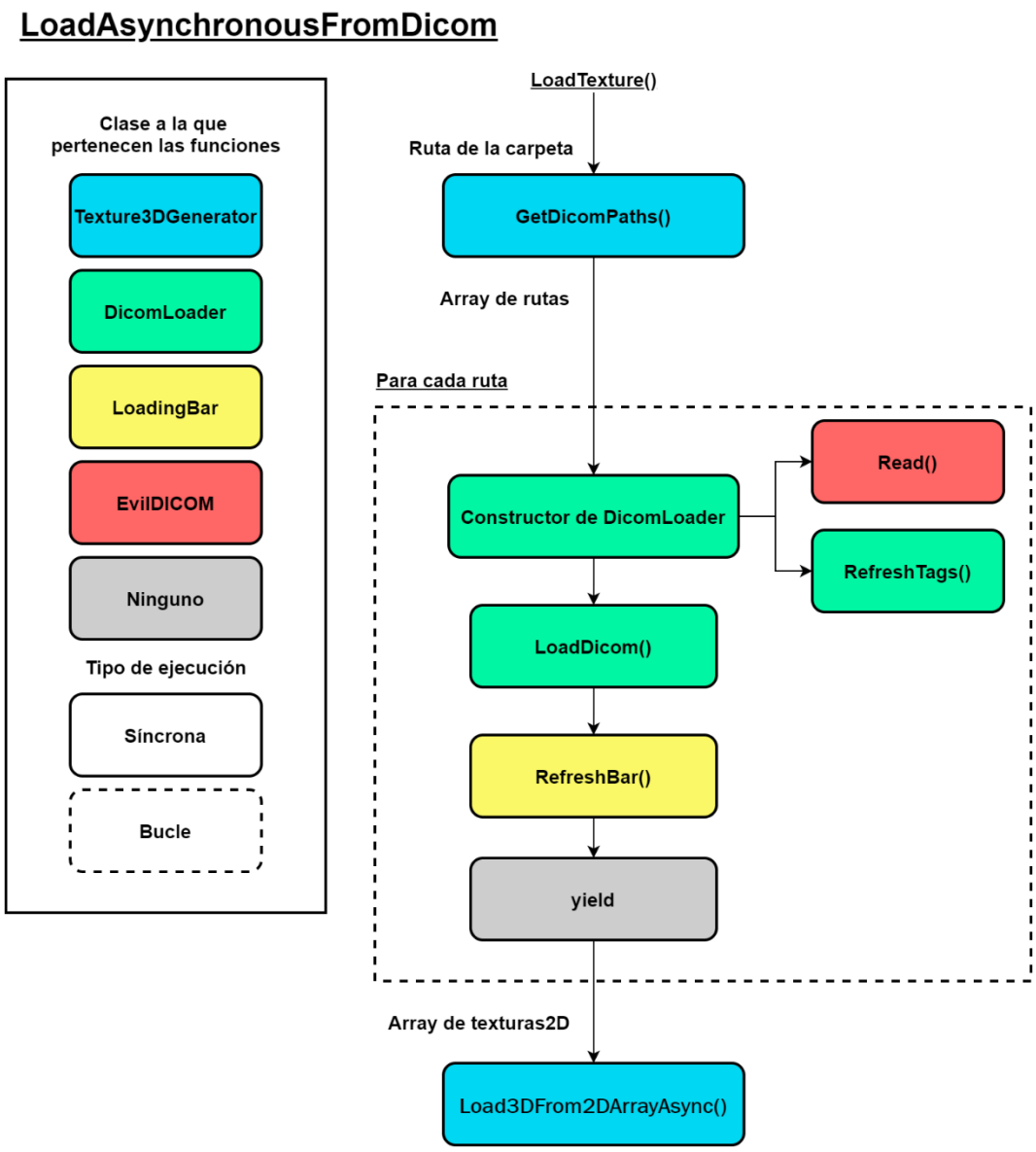


Ilustración 40: Esquema del proceso interno de la función LoadAsynchronousFromDicom(). Fuente: Elaboración propia.

4.4.3 GenerateColors32

Esta función, que es llamada por `LoadDicom()`, es la encargada de generar los valores que definirán cada una de las secciones del archivo. Dichos valores, se almacenarán en forma de textura2D. El proceso que sigue esta función (Ilustración 41) es el siguiente:

Primero, comprueba si la etiqueta de "Interpretación fotométrica (0028, 0004)" es de tipo "MONOCHROME2" (el más usual y actualmente, el único implementado). En caso de que el archivo tenga información en color, se registra un error, sin detener la ejecución del programa pero sí la carga del archivo. En caso contrario, el proceso continúa, comprobando ahora el valor de la etiqueta "Bits asignados (0028, 0100)", se valoran aquí dos casos, que el archivo asigne a cada valor 8 bits o 16 bits.

En caso de que sean 8 bits, el proceso continua con la llamada a la función `GenerateColor32bits8()`, encargada de generar los colores que se devolverán a `LoadDicom()`. En caso de que sean 16 bits, el proceso continuará con la llamada a `GenerateColor32bits16()`.

Aunque está implementado en la aplicación, el formato de 8 bits rara vez se emplea en resonancias y tomografías debido a que se busca la mayor resolución posible. Por esto y porque el proceso es bastante similar al de 16 bits, se profundizará directamente en este último.

`GenerateColor32bits16()`, de forma similar a la versión de 8 bits, devuelve un array de datos Color32. Su proceso interno es el siguiente:

- Los valores almacenados en la etiqueta "Píxel Data (7FE0, 0010)" vienen dados en datos de 8 bits, por lo que se construyen cada uno de los valores de 16 bits a partir de dos elementos consecutivos en dicha etiqueta.
- Para cada valor (ya almacenado en 16 bits) se llama a la función `MaskData()` (sobrecargada para funcionar con elementos de 8 y 16 bits), que emplea las etiquetas "Bits asignados (0028, 0100)" y "Bits almacenados (0028, 0101)" para crear una máscara que elimina los bits no deseados de cada valor.
- A continuación, se les aplica a estos valores la **Transformación de modalidad.** Solo se encuentra implementada la transformación lineal, por lo que se emplean las etiquetas "Ordenada en el origen de reescalado (0028, 1052)" y "Pendiente de reescalado (0028, 1053)" para transformar estos valores. El resultado de esta aplicación puede ser negativo, por lo que se almacena en un dato de tipo double.

- Debido a que Unity trabaja más eficientemente con datos de tipo Color32 en texturas, es decir, 8 bits para cada canal (rojo, verde, azul, alfa), y que era necesario transmitir un valor mayor que 255 y con signo al shader para realizar la **Transformación VOI LUT**, se ha decidido codificar el valor a la salida de la transformación de modalidad en el dato Color32 de la siguiente forma:
    - Rojo: 255 si es negativo, 0 en caso contrario.
    - Verde: Byte más significativo del valor absoluto representado en 16 bits.
    - Azul: Byte menos significativo del valor absoluto representado en 16 bits.
    - Alfa: 255
- La lista de valores codificados se devuelve a la función `GenerateColors32()`, desde donde es devuelta a `LoadDicom()`.

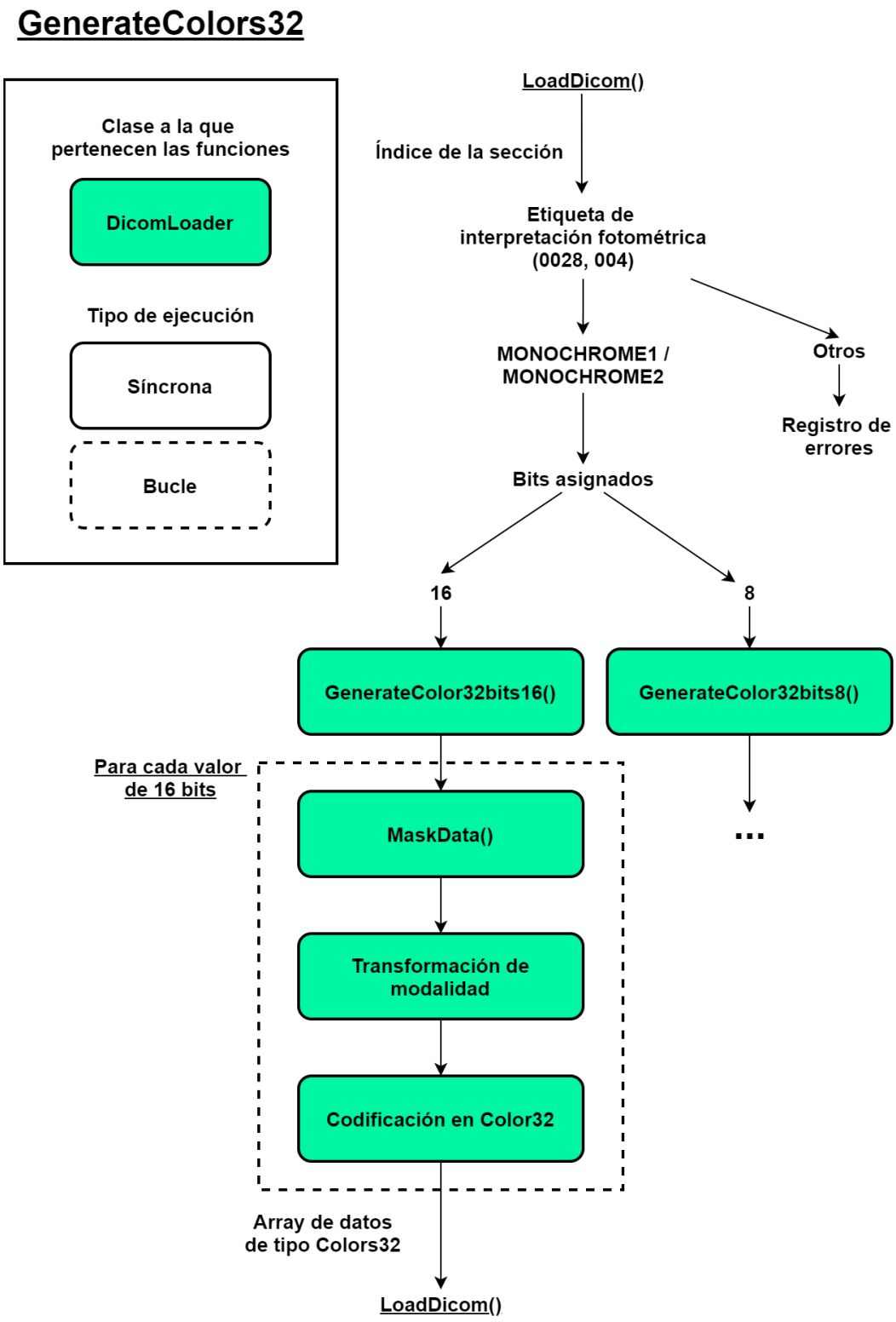


Ilustración 41: Esquema del proceso interno de la función GenerateColors32(). Fuente: Elaboración propia.

4.4.4 Load3DFrom2DArrayAsync

Esta función, es la encargada de convertir el array de texturas2D en una sola textura3D, su proceso interno (Ilustración 42) es el siguiente:

Primero, se llama a las funciones `GetPixelSpacingX()`, `GetPixelSpacingY()` y `GetSpacingBetweenSlices()`, que devuelven los valores de las etiquetas "Espacio entre píxeles (0028, 0030)" y "Espacio entre secciones (0018, 0088)", los cuales se usaran en conjunto con el número de secciones y el tamaño en pixeles de las imágenes para calcular las medidas reales. A partir de estas medidas, se escala el cubo donde se realizará la visualización, cuyas relaciones de aspecto deben coincidir con las reales para una correcta representación.

Después, se instancia una Textura3D, con la anchura y altura de las imágenes, y tanta unidades de profundidad como secciones haya más dos (Se verá más adelante el porqué). El tipo de esta textura será RGBA32 y su *WrapMode* será *Clamp* (El *WrapMode* indica como debe repetirse la textura si se accede a ella fuera de sus límites, en este caso, *Clamp* repite el "último" valor).

A continuación, se define esta textura3D a partir de las texturas2D generadas anteriormente, además de esto, se añaden dos secciones más generadas con la función `GenerateAlphaArray()` (de aquí la suma de dos en el tamaño de la textura3D) que serán totalmente transparentes, para que si se accede a la textura fuera de sus límites, lo que se repita sea un valor totalmente transparente. De forma similar, en las secciones centrales se realiza un marco de un píxel transparente a través de la función `MakeBorder()` para evitar la repetición de valores no transparentes, que darían lugar a líneas blancas que se extienden hacia el infinito.

En esta parte del programa, sería donde se podrían **incluir tratamientos de visión artificial para cada una de las imágenes**, en caso de querer ampliarse la aplicación con algoritmos de segmentación, filtros, etc. Nótese que para realizar estas operaciones tiene que reconstruirse el modelo, lo cual lleva cierto tiempo.

Se refresca la barra de carga para ilustrar este segundo proceso y se usa la palabra clave yield para permitir la ejecución asíncrona.

Por último, se invoca la función `UploadTexture3D()`, ya mencionada en el proceso general, que carga la textura3D generada en los shaders pertinentes.

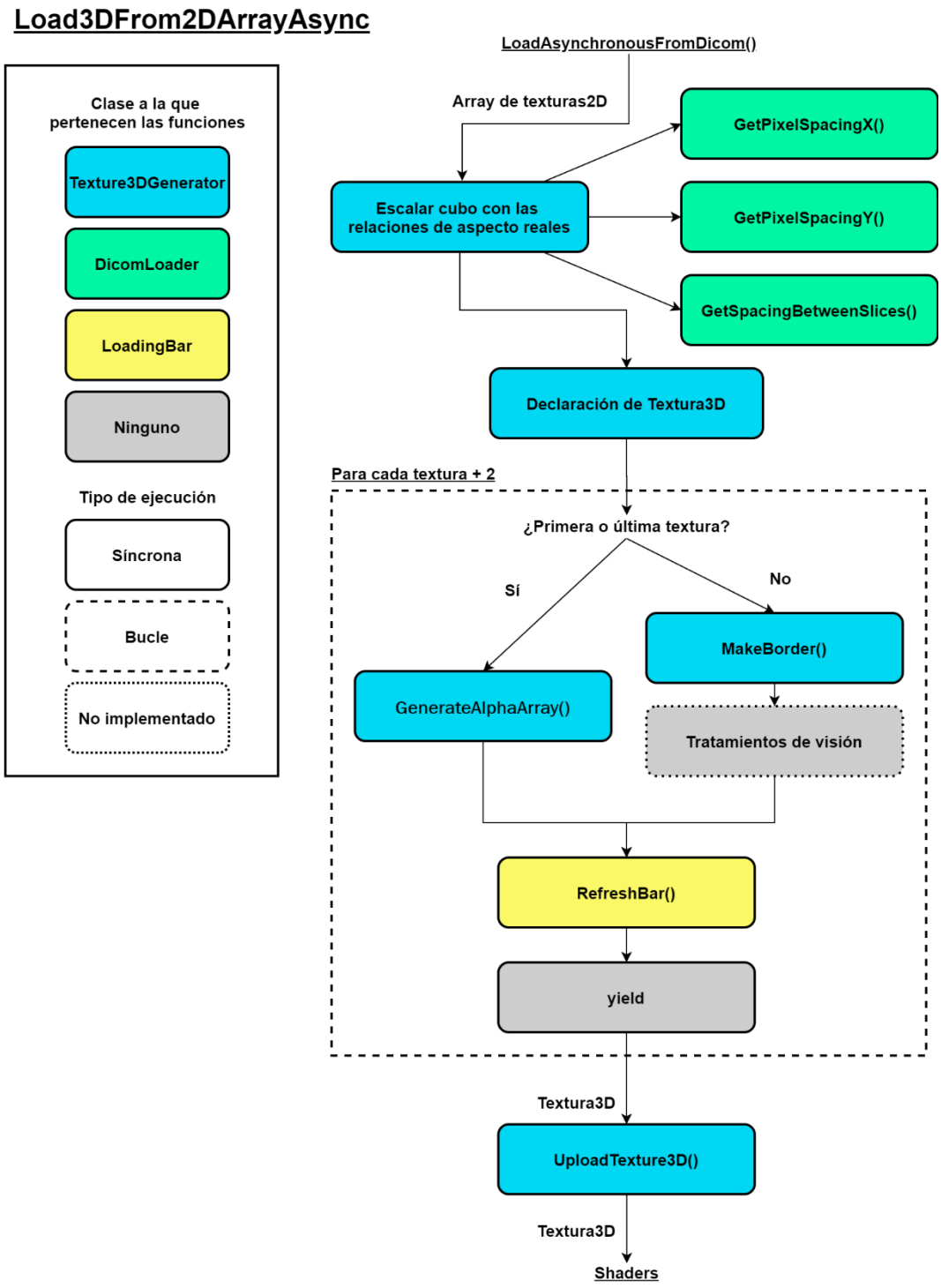


Ilustración 42: Esquema del funcionamiento interno de Load3DFrom2DArrayAsync(). Fuente: Elaboración propia.

Estas funciones de generación, cuentan con pruebas unitarias, que han permitido un proceso de depuración del código sencillo y rápido, permitiendo cargar texturas ya generadas rápidamente o generarlas sin tener que usar el visor de realidad virtual para seleccionar el archivo.

Una vez se ha generado el modelo correctamente y se dispone de una textura tridimensional con la información codificada correctamente, es el momento de descodificarla y mostrarla por pantalla. Este proceso se trata en el siguiente apartado: Visualización.

4.5 Visualización

A la hora de visualizar la información generada en la reconstrucción, se valoraron tres algoritmos diferentes (descritos en la sección **2.3.2 Renderización Volumétrica**). En este apartado, se exponen los distintos factores por los cuales se ha elegido o no cada método, así como el proceso que sigue el método elegido.

4.5.1 Algoritmos no empleados

Se consideró primero la opción de implementar el algoritmo de ***marching cubes***. Esta idea se desechó, ya que este tipo de visualización representa los archivos pobremente a menos que se empleen resoluciones muy grandes de voxels, en cuyo caso, se requiere una gran cantidad de recursos computacionales para manejar mallas con tantos triángulos.

Después, a modo de primera implementación y prueba, se decidió llevar a cabo la visualización con una **nube de puntos** (Ilustración 43). Para implementar este método, el proceso fue el siguiente:

- A partir de la textura tridimensional del modelo (dicha textura almacena unos valores fijos sin codificar, debido a que la nube de puntos se implementó en una etapa del proyecto en la que la codificación de valores no estaba presente, y por lo tanto, el centro y anchura de ventana era fijo), se genera proceduralmente una malla geométrica en la que se crea un vértice por cada uno de los pixeles de cada una de las secciones.
- A estos vértices, se les dota de un campo que almacena el color de su píxel correspondiente.
- A la hora de mostrar esta malla por pantalla, se escribió un shader que rasterizaba el color de los vértices como puntos en vez de rasterizar los triángulos que estos formaban.

Esté método, aunque tiene un buen rendimiento, presenta los problemas asociados a las nubes de puntos ya mencionados en la sección **2.3.2 Renderización Volumétrica**, como son artefactos dentro de las nubes y franjas de espacios al acercarse. Estos defectos pueden verse claramente en la Ilustración 44.

Para una correcta visualización, más aún en un entorno crítico (como el diagnóstico de enfermedades o la planificación de procedimientos quirúrgicos), es inaceptable la aparición de tan notorios artefactos, por lo que esta opción también se despreció como método de visualización.

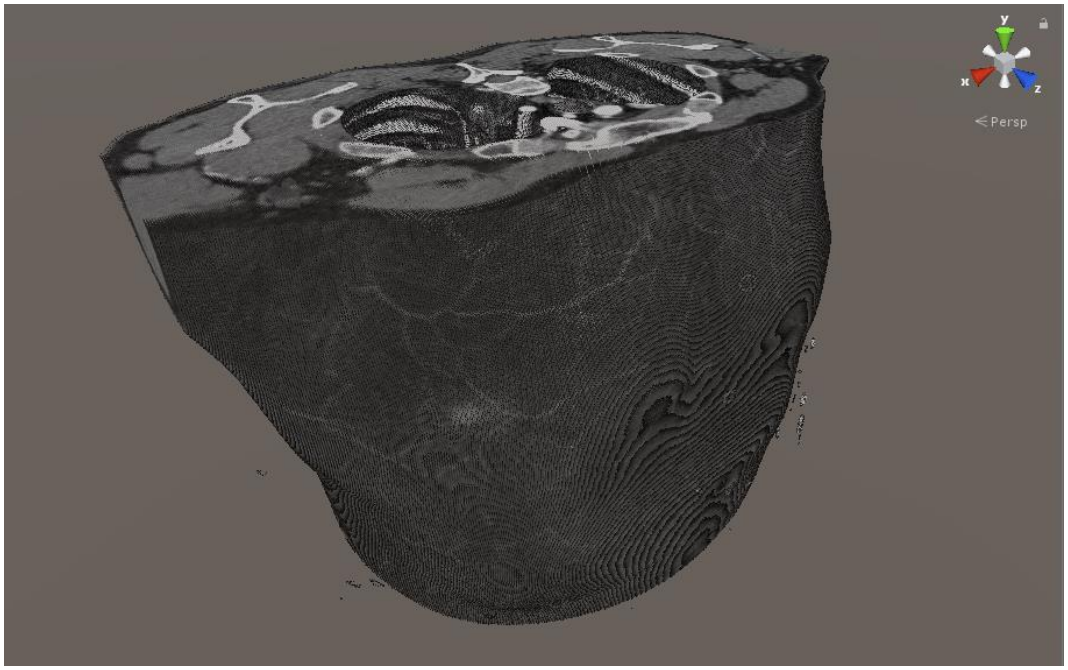


Ilustración 43: Nube de puntos del modelo generado a partir de un archivo DICOM. Fuente: Elaboración propia. Archivo: Abdominal CT disponible como dominio público en IDoImaging [38]

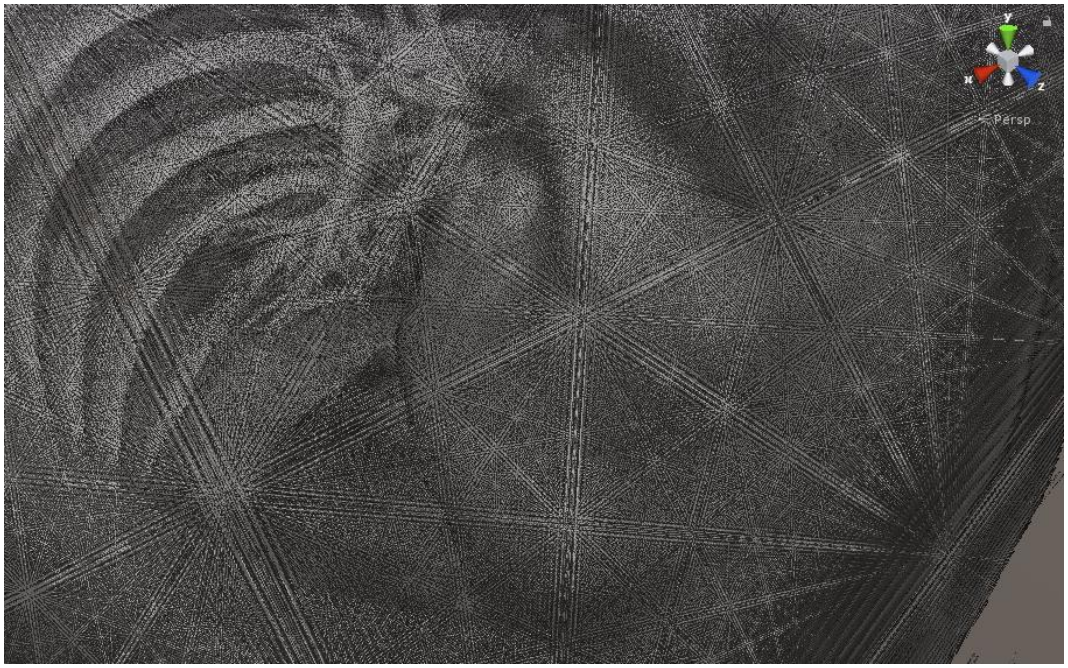


Ilustración 44: Detalle de la nube de puntos del modelo generado a partir de un archivo DICOM. Fuente: Elaboración propia. Archivo: Abdominal CT disponible como dominio público en IDoImaging [38]

4.5.2 Algoritmo empleado

Definitivamente, se decidió solucionar la visualización implementando *volumetric ray casting*, que presenta la mejor relación calidad de visualización / recursos empleados.

Para implementar este algoritmo, se ha creado un GameObject con un componente MeshFilter que tiene una malla con forma de cubo, (esta malla es a la que se modifican las dimensiones en la función descrita en el apartado **4.4.4 Load3DFrom2DArrayAsync**). Esta malla, es rasterizada, obteniendo los fragmentos que corresponden a cada píxel, tal y como se vio en el apartado **2.3.3 Render pipeline**. Estos fragmentos pasan al *pixel shader*.

En el *pixel shader*, si trabajásemos con el shader estándar de Unity, podríamos darle color al cubo, efecto metalizado, sombreado, etc. En cambio, se ha escrito un *pixel shader* que extrae información de la matriz tridimensional que es la textura generada en el apartado anterior, para computar el valor (color) que se mostrará en el pixel correspondiente de las pantallas del visor de realidad virtual.

El proceso que lleva a cabo este shader para cada uno de los fragmentos del cubo es el siguiente:

- Primero, a partir de las coordenadas transformadas de los vértices (que pasan del espacio de la malla al espacio global de la escena en el *vertex shader*), se obtienen por interpolación las coordenadas respecto del sistema de referencia global de cada uno de los fragmentos. También, una vez se conoce la posición del fragmento y la posición de la cámara, se calcula la dirección cámara-fragmento a modo de vector normalizado.

- A continuación, conocidos la posición del fragmento y la dirección del rayo que pasa por el centro de la cámara y por dicha posición, se ejecuta el *volumetric ray casting*, que generará el valor del color a mostrar. La implementación consiste en:

    - Se calcula un vector *step* multiplicando la dirección cámara-fragmento por la constante *STEP_SIZE* (configurable antes de la ejecución de la aplicación). A continuación, se itera hasta un máximo de pasos *STEP*, aumentando en cada iteración la posición de un muestreador que leerá los valores de la textura. Este bucle consta de tres partes:

- Se realiza el producto escalar entre el vector *centro del plano de corte – posición del muestreador* y el vector normal de la superficie del plano (en este plano de corte se profundizará en la sección **4.7.1 Plano de corte**). Si el resultado es menor que cero, significa que el punto está por encima del plano (respecto de su normal) y el color muestreado en dicho punto será el (0, 0, 0, 0), haciendo que ese punto sea transparente totalmente.
  En caso contrario, se extrae el valor de la textura (codificado), que se pasa a través de la función VOILUT.

- Esta función, descodifica el valor de la textura, obteniendo el valor original a la salida de la transformación de modalidad, al cual se le aplica la lógica descrita en la Ecuación 2 del apartado **2.2.2.3 Características de visualización** a partir de los valores seleccionados en el variador de centro y anchura en el que se profundizará en el apartado **4.7.3 Variador de centro y anchura**.
  De esta forma, es posible cambiar la ventana visualizada en tiempo real, en vez de tener que reconstruir el modelo cada vez que cambie.

- Se acumula el valor devuelto multiplicado por un parámetro _Alpha con los valores de las anteriores iteraciones y se comprueba que la transparencia de la acumulación no supere la unidad (el valor sería opaco y por lo tanto no tendría sentido "mirar" detrás). Si la transparencia aún no ha llegado a uno, se le suma a la posición de muestreo el vector *step* para avanzar a través del rayo, realizando el mismo proceso hasta llegar a una opacidad unidad o hasta superar el límite de iteraciones *STEP*.

- Una vez finalizado el bucle, el color obtenido se muestra en el píxel correspondiente, generando de esta forma la imagen que se muestra en las pantallas.

En este proceso, influyen en gran medida ciertos parámetros, cuyos valores elegidos se discuten en la siguiente sección.

4.5.3 Valores de los parámetros

En cuanto a los parámetros no modificables durante la ejecución de la aplicación, se encuentra el tamaño del paso del muestreador *STEP_SIZE*, el número máximo de pasos *STEP*, y el factor por el que se multiplican los valores acumulados, *_Alpha*.

Cuanto más pequeño sea *STEP_SIZE*, mayor resolución tendrá la visualización, ya que se están muestreando más valores. Por otro lado, si el tamaño del paso es demasiado pequeño, se llegará al límite de pasos *STEP* antes de atravesar el modelo por completo, causando que la representación pueda visualizarse incompleta (Ilustración 45).

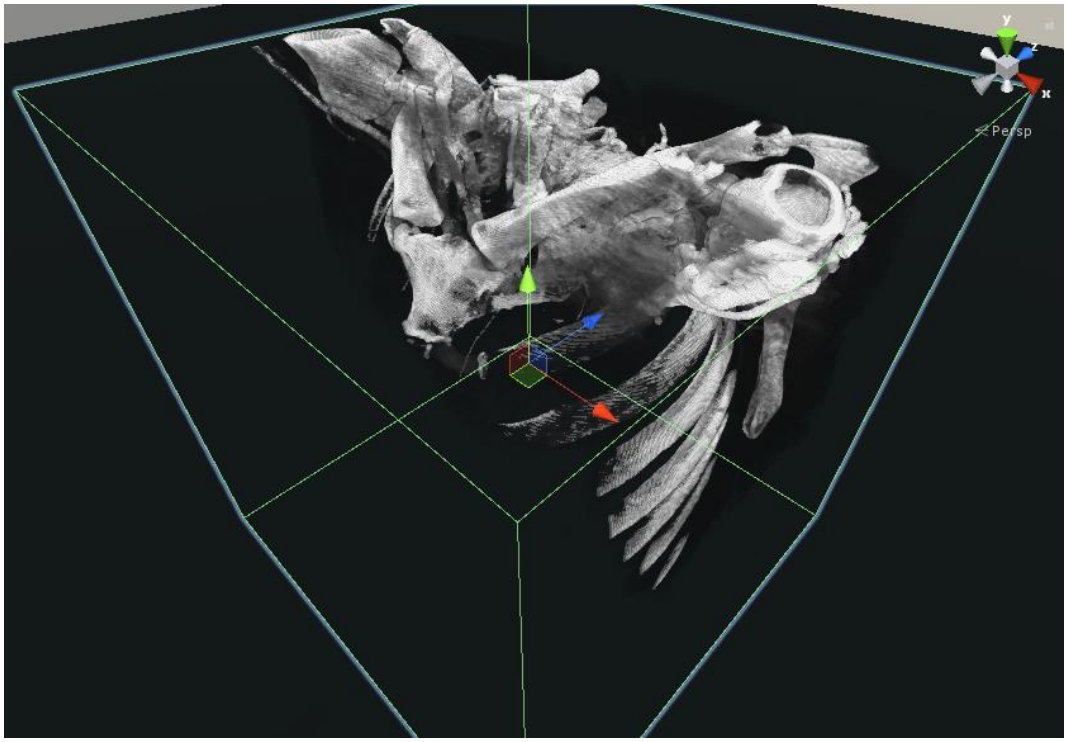


Ilustración 45: Renderización incompleta debido a un número insuficiente de pasos. Fuente: Elaboración propia. Archivo: Abdominal CT disponible como dominio público en IDoImaging [38]

Esto puede solucionarse aumentando el límite de pasos máximo, pero entonces, se estará realizando un número mucho mayor de operaciones cada vez que se refresca la pantalla, para lo que se necesitará una tarjeta gráfica de mejores características. En la Ilustración 46, la Ilustración 47 y la Ilustración 48 puede apreciarse que a medida que disminuye el tamaño de paso, el resultado presenta una calidad mayor.

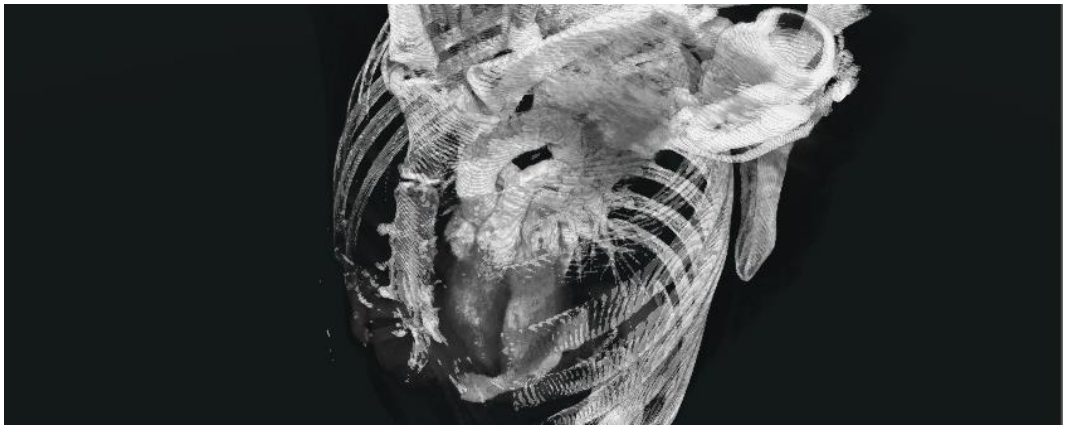


Ilustración 46: Tamaño de paso 0.004, número máximo de pasos 125. Fuente: Elaboración propia. Archivo: Abdominal CT disponible como dominio público en IDoImaging [38]

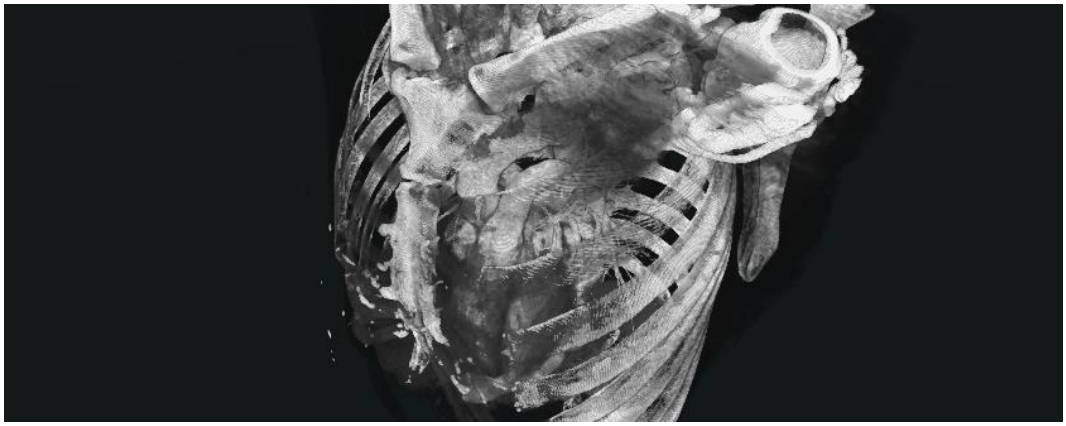


Ilustración 47: Tamaño de paso 0.002, número máximo de pasos 512. Fuente: Elaboración propia. Archivo: Abdominal CT disponible como dominio público en IDoImaging [38]

Ilustración 48: Tamaño de paso 0.001, número máximo de pasos 1024. Fuente: Elaboración propia. Archivo: Abdominal CT disponible como dominio público en IDoImaging [38]

En la aplicación actualmente se usa un tamaño de paso de 0.002 y un número de pasos máximo de 512 para asegurar una tasa de refresco de las pantallas que resulte cómoda para el usuario. En caso de que se emplease otro equipo de mayores prestaciones, podría emplearse un tamaño de paso de 0.001 y un número de pasos máximo de 1024 para conseguir unos resultados superiores.

Ilustración 49: Valor de _Alpha 0.15. Fuente: Elaboración propia. Archivo: Abdominal CT disponible como dominio público en IDoImaging [38]

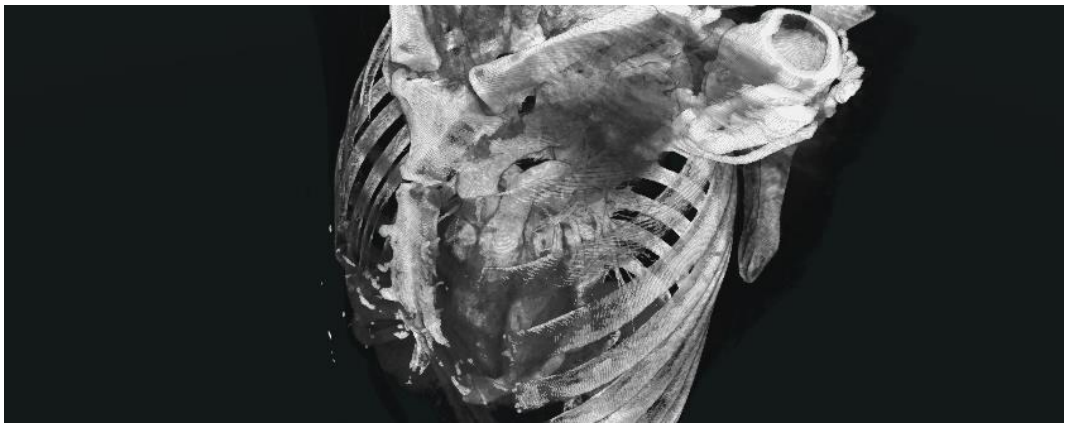


Ilustración 50: Valor de _Alpha 0.5. Fuente: Elaboración propia. Archivo: Abdominal CT disponible como dominio público en IDoImaging [38]

Ilustración 51: Valor de _Alpha 0.95. Fuente: Elaboración propia. Archivo: Abdominal CT disponible como dominio público en IDoImaging [38]

En cuanto al valor de *_Alpha*, cuanto mayor es, menos transparencia tiene la representación y menor recorrido tiene el rayo. Esto puede apreciarse en la Ilustración 49, la Ilustración 50 y la Ilustración 51**.** El valor elegido es 0.5, que presenta una opacidad suficiente para ver la reconstrucción correctamente sin llegar a saturar el resultado.

4.6 Manipulación del modelo

Para permitir que el usuario interaccione con la reconstrucción y las herramientas se han empleado las funcionalidades de *VRTK4* (*Virtual Reality Toolkit*) para convertir dichos GameObjects en objetos interactivos.

Entre las muchas funcionalidades de *VRTK4*, destaca la posibilidad de configurar fácilmente GameObjects para hacer posible la interacción entre los controladores y ellos. En el caso de esta aplicación, se encuentran dos variaciones de estos objetos interactivos:

- Acción secundaria escalar: Esta configuración permite coger el objeto y girarlo de forma que siga los movimientos del controlador con el que se ha agarrado. Además, si el objeto está agarrado con una mano y se agarra a la vez con la otra, se podrá modificar la escala del objeto, haciéndolo más o menos grande en función de la separación entre las manos. El cubo donde se realiza la visualización del modelo reconstruido es un objeto de este tipo.

- Acción secundaria cambio de mano: En este caso, si el objeto es agarrado con una sola mano, el funcionamiento es el mismo que en "Acción secundaria escalar", pero al intentar agarrar con la otra mano, el objeto deja de estar asociado a la primera y sigue los movimientos de la última mano en agarrarlo. De esta forma se impide que ciertos objetos puedan cambiar su tamaño manteniendo una experiencia de usuario intuitiva. El resto de objetos interactivos de la escena (herramientas) son objetos de este tipo.

En cuanto al motor de simulaciones físicas de Unity, se han desactivado tanto las colisiones entre objetos interactivos para evitar rebotes, como la inercia y la masa de todos los elementos. De esta forma, se impide que se vean afectados por la gravedad o que permanezcan moviéndose por el espacio al soltarlos.

En este apartado se ha nombrado un conjunto de objetos interactivos como herramientas, estos objetos, cuya finalidad es mejorar la experiencia de visualización y exploración dentro del entorno virtual, se exponen en la siguiente sección.

4.7 Herramientas

A continuación, se profundizará en las características y principios de funcionamiento de cada una de las herramientas desarrolladas para añadir funcionalidades a la aplicación.

4.7.1 Plano de corte

Se ha añadido un objeto interactivo que hace de plano de corte, esto significa que puede ser colocado en cualquier posición por el usuario para ocultar una porción del modelo reconstruido, permitiendo de esta forma visualizar el interior de las reconstrucciones con mayor detalle (Ilustración 52).

Como se ha mencionado en el apartado anterior, estos objetos interactivos permiten que el usuario use los controladores para agarrarlos, haciendo de esta forma posible modificar su posición en cualquier momento.

Cada vez que el plano es movido por el usuario, se actualizan las variables del shader del cubo de visualización que almacenan la posición del plano y el vector normal a su superficie. Con estas variables se opera en el shader de la forma descrita en el apartado **4.5.2 Algoritmo empleado** para conseguir que los valores que queden por encima (respecto del sistema de coordenadas local del plano) no se tengan en cuenta en la renderización.

El efecto de este plano, puede activarse y desactivarse pulsando el botón B (Es posible consultar los nombres de los botones en el apartado **4.1.2 Distribución de los botones en los controladores**).

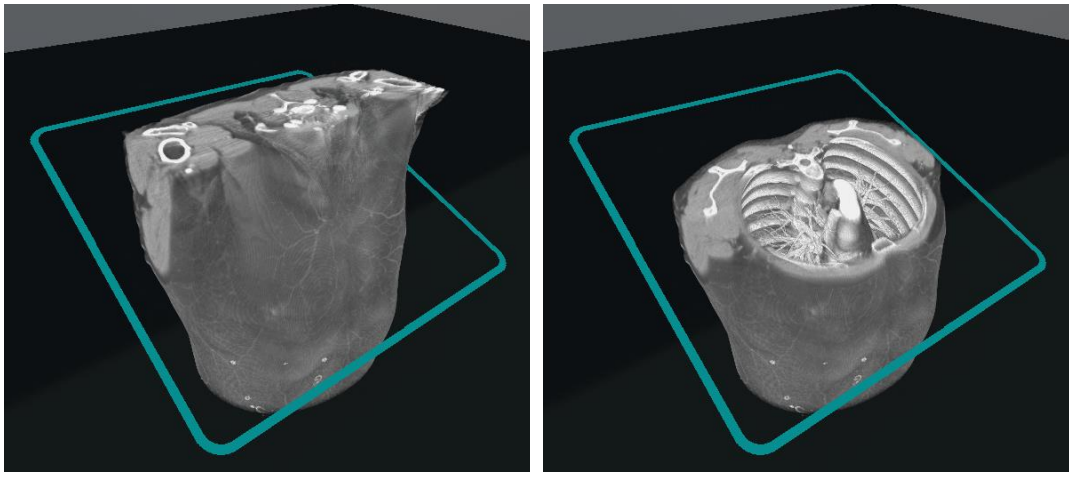


Ilustración 52: Plano de corte desactivado (izquierda) y activado (derecha). Fuente: Elaboración propia. Archivo: Abdominal CT disponible como dominio público en IDoImaging [38]

4.7.2 Puntero y planos anatómicos

También se ha desarrollado para la aplicación un puntero (configurado como objeto interactivo) que puede ser colocado por el usuario en cualquier posición. La punta de dicho puntero, indica la posición a partir de la cual se generarán los tres planos anatómicos (Ilustración 53).

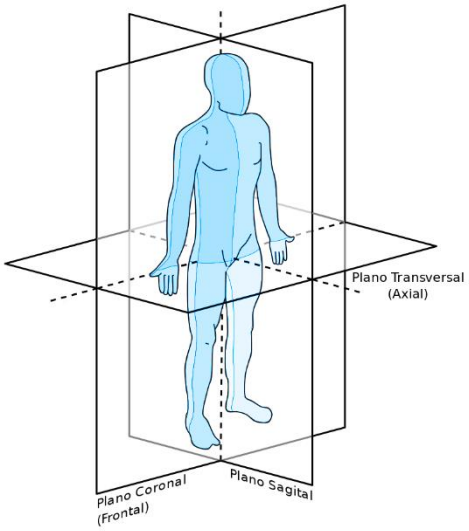


Ilustración 53: Planos anatómicos del cuerpo. Fuente: Modificado a partir de "Planos anatómicos" de Edoarado [En línea] Disponible en:https://commons.wikimedia.org/wiki/File:Planos_anat%C3%B3micos.svg

Estas reconstrucciones de los planos anatómicos, se visualizarán en tres paneles (también objetos interactivos) (Ilustración 54) que se renderizan accediendo a una sola sección bidimensional de la textura tridimensional del modelo y haciendo la transformación VOI LUT adecuada en función de los valores de centro y anchura de ventana (también modificables en tiempo real).

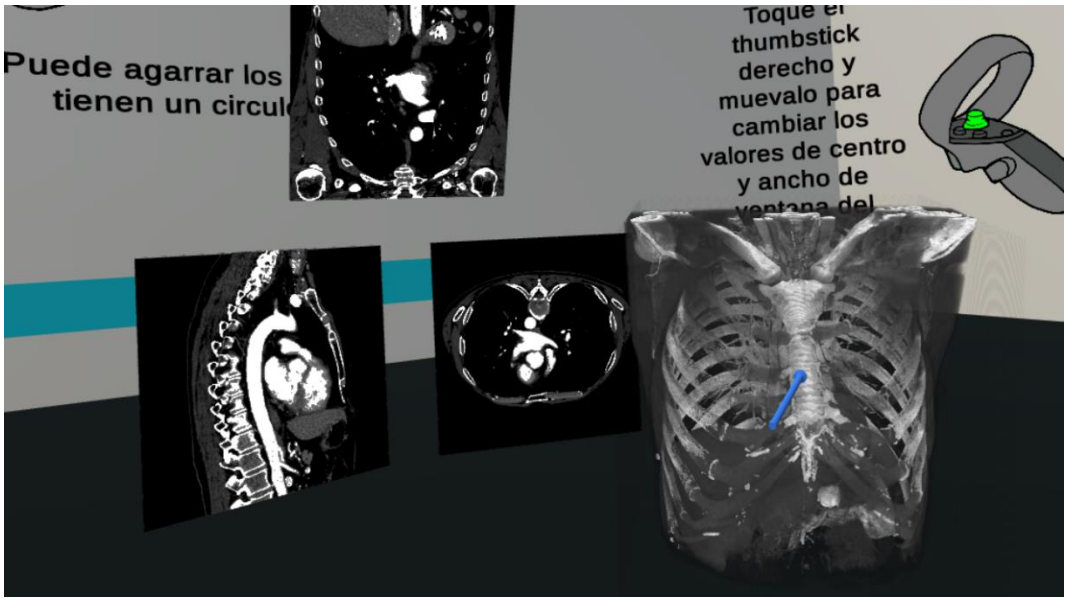


Ilustración 54: Paneles con los planos anatómicos generados en el punto señalado por el puntero (flecha azul). Fuente: Elaboración propia. Archivo: Abdominal CT disponible como dominio público en IDoImaging [38]

4.7.3 Variador de centro y anchura

Al tocar el *Thumbstick* derecho, se despliega un pequeño panel informativo que indica al usuario que al mover dicho *Thumbstick*, se variará tanto la anchura como la localización de la ventana (referentes a la transformación VOI LUT descrita en el apartado **2.2.2.3 Características de visualización**). También indicará que es posible reestablecer los valores por defecto si se pulsa el *Thumbstick*.

Para conseguir esto, se emplean funciones de *VRTK4* para registrar tanto el contacto con el *Thumbstick* como su movimiento, este movimiento (registrado en forma de coordenadas x, y) pasa a una función donde se procesa y se actualizan los valores de anchura y localización de la ventana dependiendo de si es un movimiento vertical u horizontal. Una vez se han actualizado los valores, se pasan a los shaders pertinentes (el del cubo de visualización y el de los paneles de planos anatómicos) que realizan la transformación VOI LUT.

Con este proceso, se dota al usuario de la capacidad de cambiar en tiempo real el rango de valores mostrados, permitiendo así la visualización de diferentes tejidos (Ilustración 55).

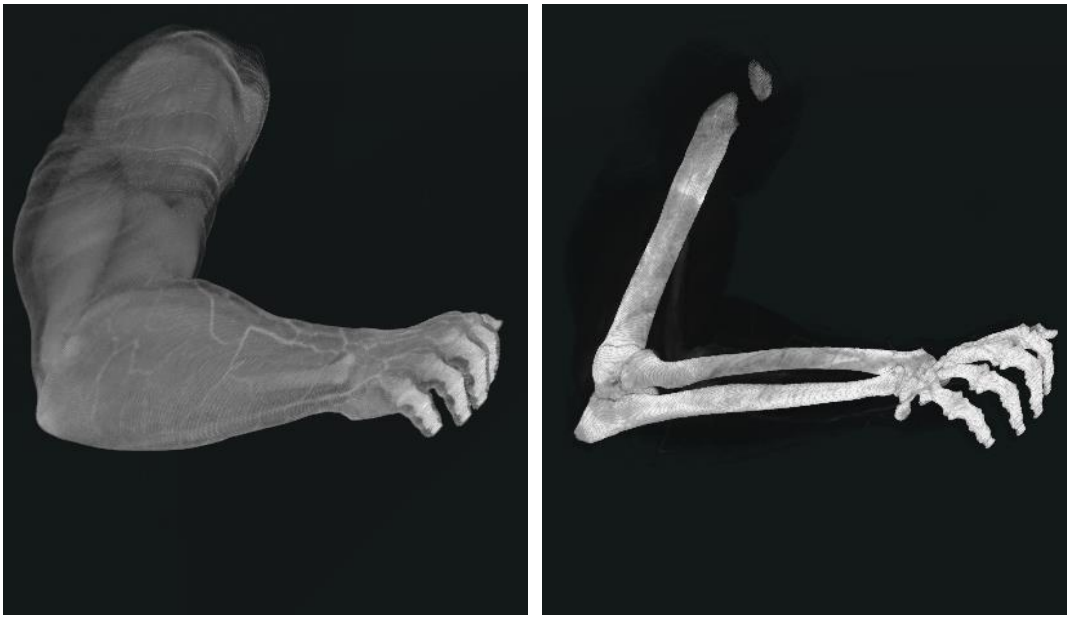


Ilustración 55: Visualización en la aplicación de diferentes tejidos (izquierda – músculo, derecha – hueso) a partir de un mismo archivo DICOM. Fuente: Elaboración propia. Archivo: Arm CT disponible como dominio público en IDoImaging [38]

4.7.4 Botones destacados

A modo de ayuda a la interfaz entre la aplicación y el usuario, se ha desarrollado un componente que cambia el color del botón del controlador que habría que pulsar para llevar a cabo ciertas acciones basándose en los actos del usuario. Por ejemplo, cuando la mano del usuario entra en contacto con un objeto interactivo, se destaca el botón que habría que apretar para coger dicho objeto (Ilustración 56), ya que se entiende que es la acción que quiere llevar a cabo el usuario. Otros ejemplos son el botón que hay que apretar en cada caso para seleccionar una opción en el explorador de archivos (Ilustración 57) o en el menú radial.

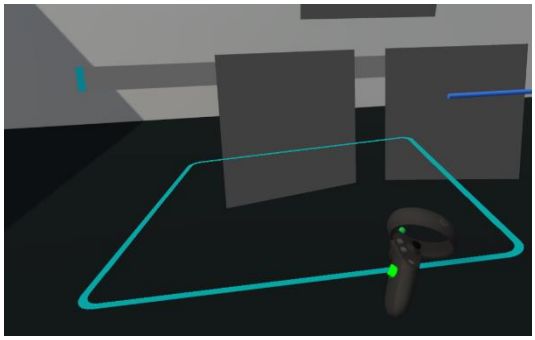


Ilustración 56: Botón de agarrar destacado al entrar en contacto el controlador y el objeto interactivo. Fuente: Elaboración propia.

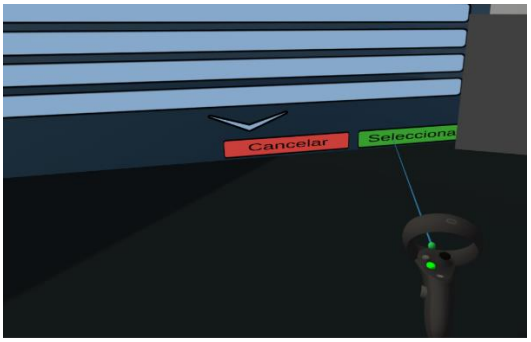


Ilustración 57: Botón de seleccionar destacado al abrir el explorador de archivos. Fuente: Elaboración propia.

Para conseguir esto, el componente está suscrito a una serie de sistemas de eventos, que notifican las colisiones con los objetos interactivos o determinados estados de la aplicación (por ejemplo, seleccionando opción en el explorador de archivos). Cuando se registran estos cambios, se cambian las texturas asociadas a los controladores, que han sido previamente modificadas mediante un software de edición fotográfica (Ilustración 58).

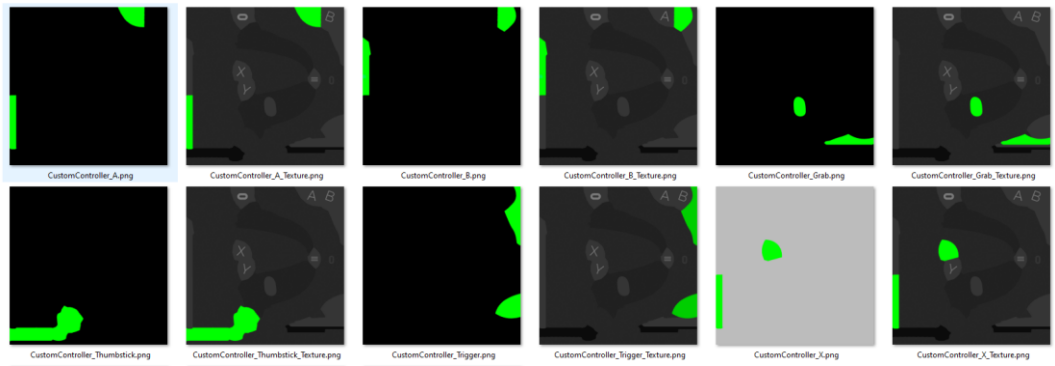


Ilustración 58: Texturas modificadas para resaltar cada uno de los botones. Fuente: Elaboración propia.

4.8 Documentación generada

4.8.1 Código

A lo largo del código escrito durante el desarrollo de este proyecto (22 archivos con más de 3150 líneas de código en total) se han empleado comentarios de documentación XML (Ilustración 59) para especificar la función, parámetros, y devoluciones de las funciones y métodos desarrollados, así como también documentar los componentes, clases, variables y espacios de nombres.

```
/// <summary>
/// Genera una textura3D a partir de un array de texturas2D (Dicom)
/// </summary>
/// <param name="_frames">Array de texturas2D</param>
/// <param name="dl">Instancia de DicomLoader del que estamos generando la textura.</param>
/// <returns>Textura3D generada a partir del array</returns>
1 referencia
private Texture3D Load3DFrom2DArraySync(Texture2D[] _frames, DicomLoader dl) {
```

Ilustración 59: Ejemplo de una función del proyecto documentada con comentarios XML. Fuente: Elaboración propia.

Estos comentarios XML permiten el uso de etiquetas y enlaces internos dentro de los programas, facilitando la navegación y comprensión del código. Además, emplear estos comentarios XML permiten el posterior procesamiento del código de forma automatizada para generar páginas web de consulta como se verá en el apartado **4.8.3 Generación de página web**.

4.8.2 Unity

En cuanto al proyecto de Unity, dado que no existe una forma estándar de documentar sus proyectos, se ha escrito un documento (adjunto en anexos) en el que se especifican los componentes de cada GameObject, así como su función y sus relaciones internas (Ilustración 60).

TrackedAlias : Aliases : LeftControllerAlias : RadialUI : Tooltips : Tooltip : Text <Lepio> <TMPro>

- Texto del Tooltip.
- Tag: Ninguno.
- Capa: UI.
- Componentes:
  - RectTransform: Encargado de manipular la localización, rotación y escala de un objeto bidimensional.
  - CanvasRenderer: Encargado de renderizar un componente gráfico dentro del Canvas.
  - TextMeshProUGUI: Texto TextMeshPro del Tooltip.

Ilustración 60: Ejemplo de la documentación de un GameObject en el archivo de documentación del proyecto de Unity. Fuente: Elaboración propia.

Se especifican también en dicho documento las capas añadidas al motor físico de Unity, así como la matriz de colisiones del proyecto (relación entre las capas que pueden colisionar entre sí). Además, se puede encontrar también una pequeña explicación para configurar elementos de interfaz de usuario con los que interactuar empleando el puntero.

4.8.3 Generación de página web

Tal y como se ha mencionado en el apartado **4.8.1 Código**, es posible generar una página web a partir de los comentarios XML. Para esto, se ha empleado Doxygen, un programa de software libre que permite la generación automática de documentación de diferentes lenguajes de programación, configurado en este caso para trabajar con C#.

Esta página web (Ilustración 61) estructura los espacios de nombre y clases, además de facilitar la navegación a través de los diferentes elementos de la documentación del proyecto.

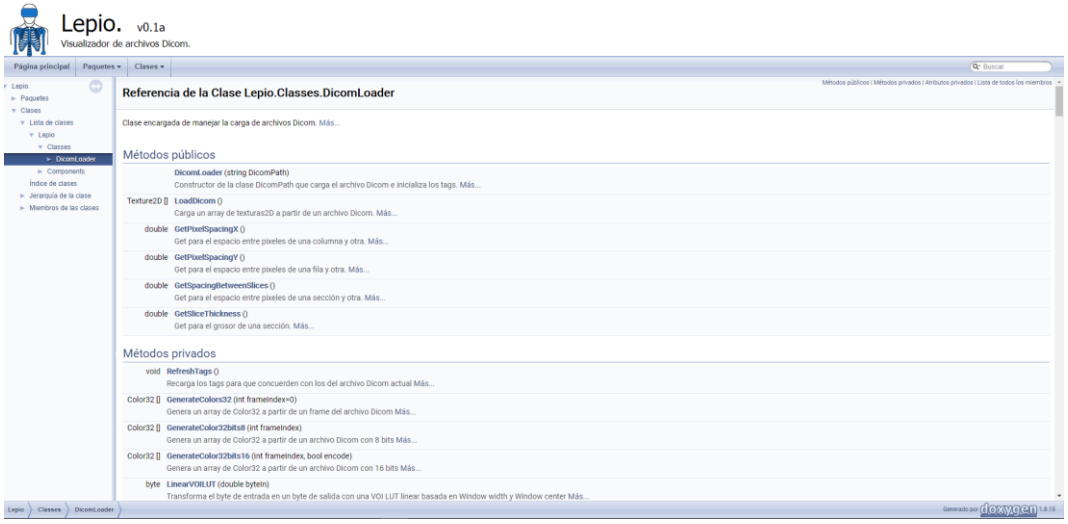


Ilustración 61: Página web generada con Doxygen. Fuente: Elaboración propia.

Los archivos necesarios para explorar dicha página web, se encuentran adjuntos en los anexos.

4.9 Resultado

A continuación se exponen los resultado de este capítulo de desarrollo, divididos en implementación de DICOM, visualización y experiencia del usuario en la aplicación.

4.9.1 Implementación de DICOM

Una vez finalizado el desarrollo de la aplicación, se ha tenido la oportunidad de probar variedad de archivos, tanto de resonancias magnéticas como de tomografías. Estos archivos han sido generados en diferentes localizaciones como son el Hospital Clínico Universitario de Valladolid, el Hospital Universitario Quirónsalud de Madrid, e incluso el Hospital Veterinario Universidad de León.

Dichos archivos se han conseguido leer y reconstruir sin problema, probando que la porción del estándar DICOM implementada en este trabajo cubre gran parte de las posibles variaciones que existen entre los estudios de distintos hospitales, con distintas patologías e incluso diferentes especies. Además, se ha podido probar la aplicación con archivos procedentes de distintas bases de datos de internet con resultados igualmente satisfactorios.

4.9.2 Visualización

La implementación de los algoritmos de visualización ha resultado satisfactoria, ofreciendo una visualización nítida del modelo reconstruido. Se exponen en las ilustraciones siguientes los resultados de visualizar diferentes archivos.

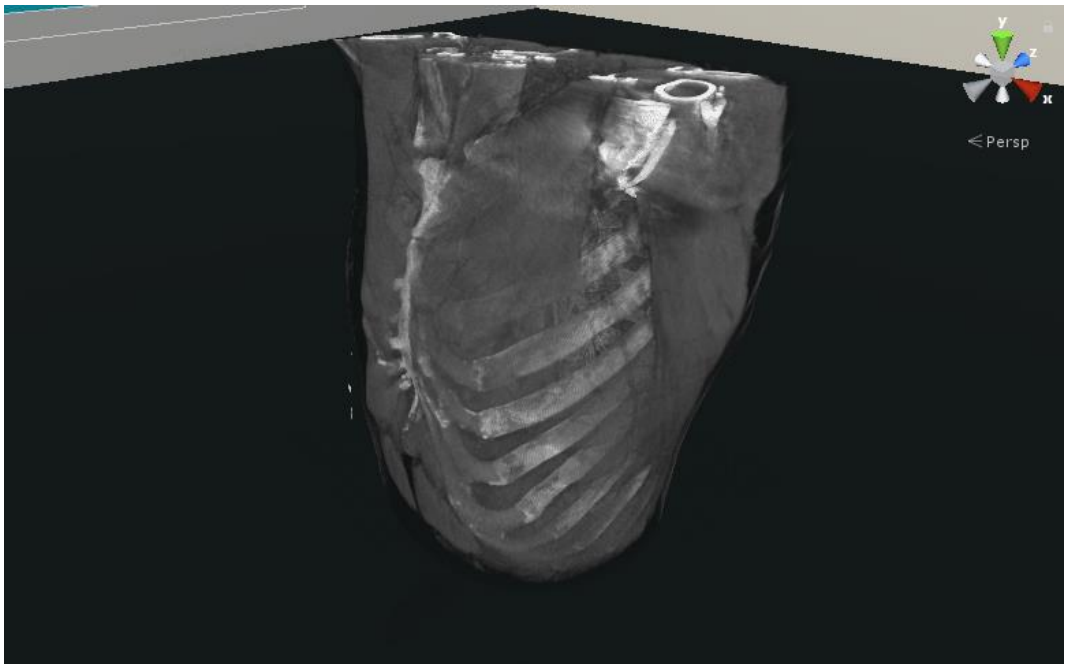


Ilustración 62: Tomografía abdominal. Fuente: Elaboración propia. Archivo: Abdominal CT disponible como dominio público en IDoImaging [38]

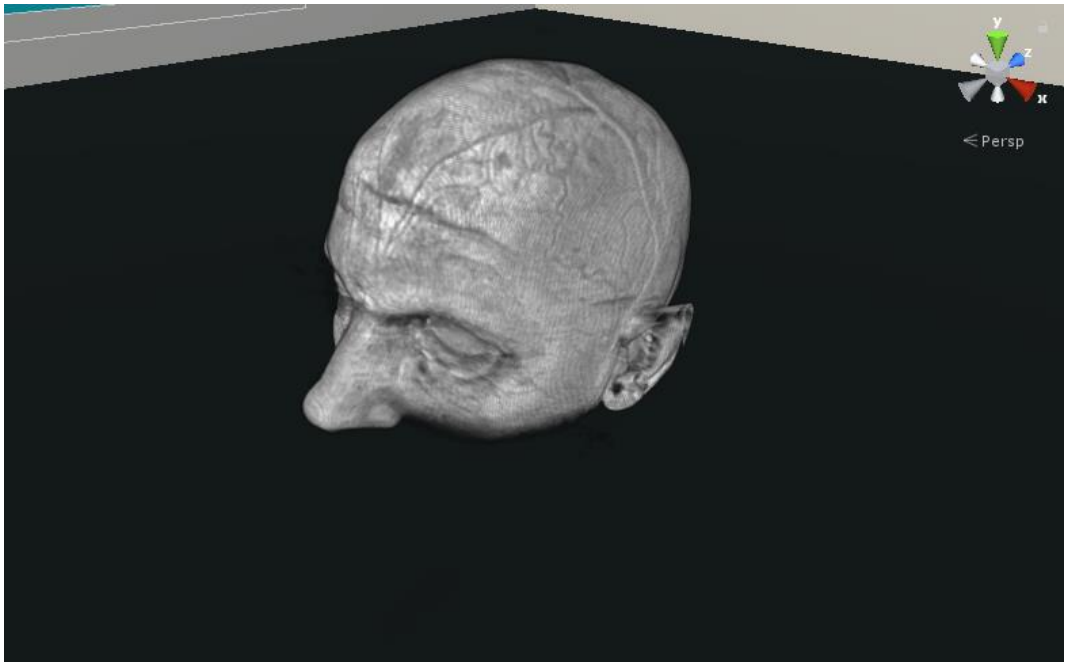


Ilustración 63: Resonancia magnética del cerebro. Fuente: Elaboración propia. Archivo: Brain MR disponible como dominio público en IDoImaging [38]

Al comparar la Ilustración 63 (gran cantidad de secciones con poca distancia entre ellas) y la Ilustración 64 (menos secciones con una distancia mayor) se comprueba que a menor distancia entre secciones, la reconstrucción deberá interpolar menos valores intermedios y los resultados serán mejores.

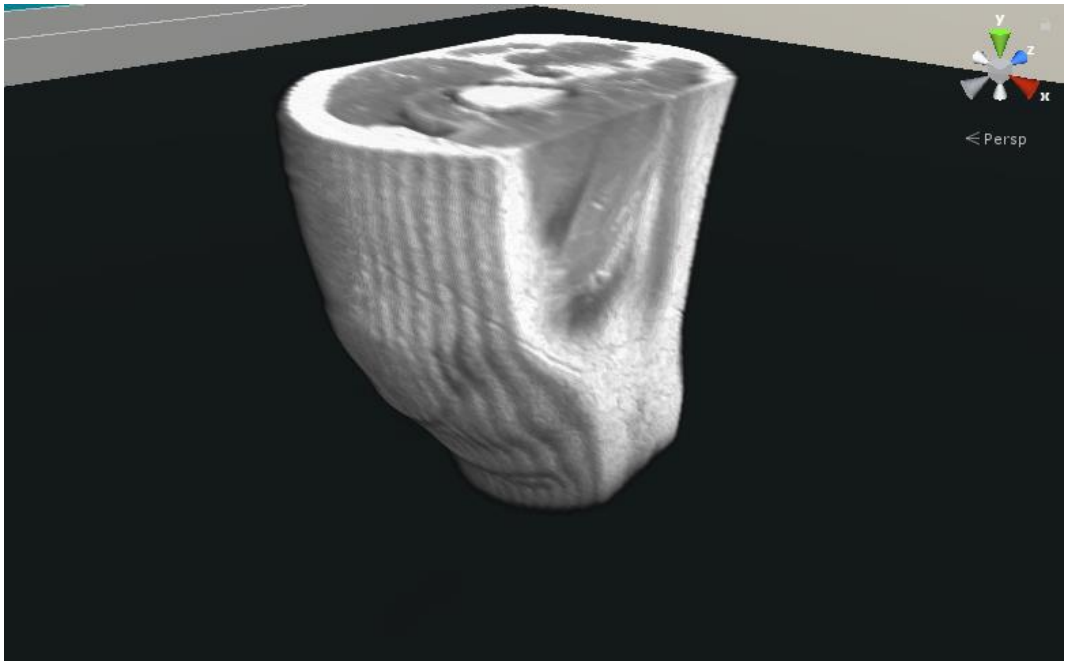


Ilustración 64: Resonancia magnética de una rodilla. Fuente: Elaboración propia. Archivo: Knee MR disponible como dominio público en IDoImaging [38]

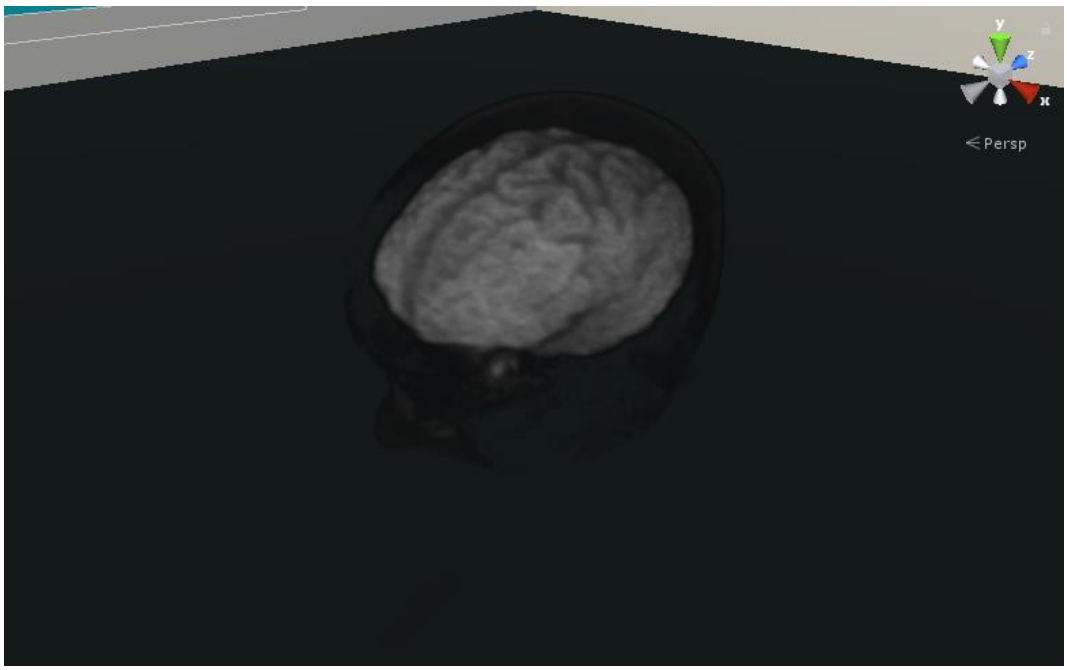


Ilustración 65: Tomografía por emisión de positrones de un cerebro. Fuente: Elaboración propia. Archivo: Brain PET disponible como dominio público en IDoImaging [38]

Se puede observar también como aun variando la localización y la anchura de la ventana de la transformación VOI LUT, los archivos siguen visualizándose correctamente, ofreciendo además la ya mencionada posibilidad de ver más claramente partes ocultas por otros tejidos (Ilustración 65, Ilustración 66).

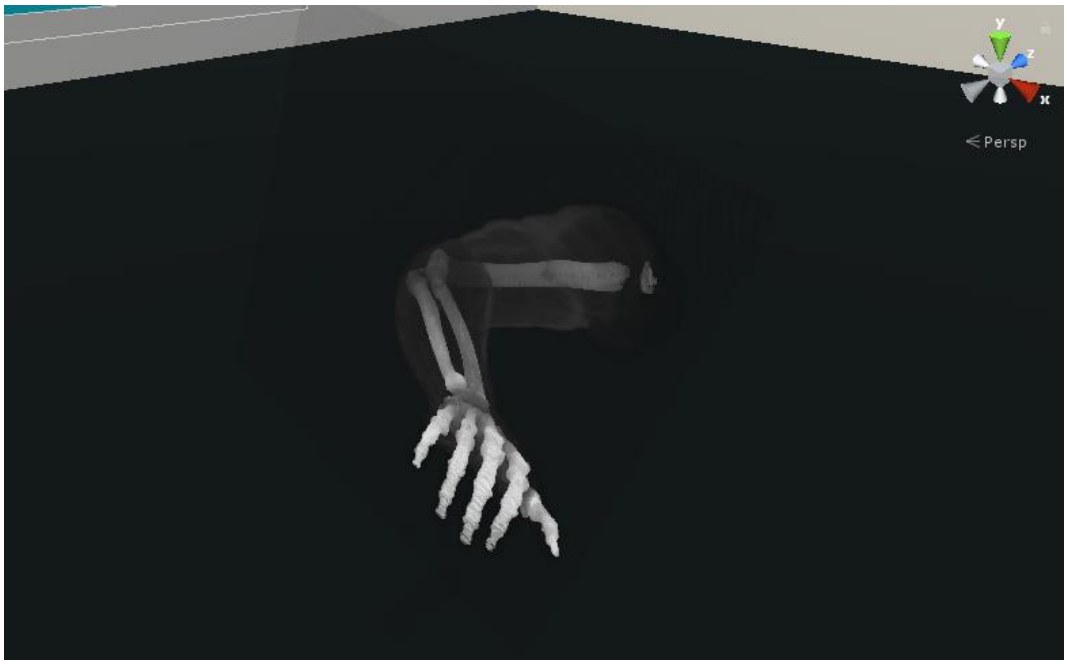
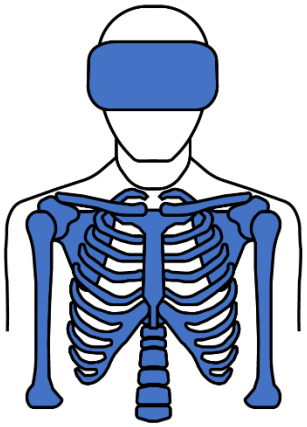


Ilustración 66: Tomografía de un brazo. Fuente: Elaboración propia. Archivo: Arm CT disponible como dominio público en IDoImaging [38]

4.9.3 Experiencia del usuario en la aplicación

Debido a la naturaleza dinámica de la aplicación desarrollada en este trabajo de fin de grado y aun teniendo en cuenta que se pierde de esta forma el carácter inmersivo del uso de un visor de realidad virtual, se adjunta en los anexos del documento un vídeo grabado desde el punto de vista del usuario en el que se manipula la reconstrucción tridimensional y se emplean las distintas herramientas desarrolladas.



Capítulo 5: Conclusiones

En este capítulo se concluye este trabajo de fin de grado exponiendo tanto las conclusiones como las posibles líneas de investigación sobre las que se podría continuar trabajando.

5.1 Conclusiones

A lo largo de este trabajo de fin de grado se ha desarrollado tanto una aplicación que permite la exploración de archivos que siguen el estándar DICOM como una base de conocimiento sobre el estándar y sobre el correcto procesamiento de la información que contiene.

Además, este proyecto constituye una base apropiada para el desarrollo de futuras investigaciones, al integrar tanto la lectura de archivos DICOM como la posibilidad de renderizar dichos archivos, controlando y pudiendo modificar el investigador todos los detalles de este proceso. Son estos detalles los que muchas veces incorporan dificultades a la hora de llevar a cabo proyectos de investigación, por ejemplo, en:

- Ciertos trabajos de fin de grado, desarrollados en este mismo departamento, que han trabajado con imágenes en vez de archivos DICOM debido a la barrera que suponía la integración de este estándar, perdiendo de esta forma toda la información que incluyen los archivos.

- Proyectos que emplean software propietario o con métodos de visualización más sencillos, donde no es posible modificar el funcionamiento de las partes más internas de estos procesos, haciendo imposible la incorporación de, por ejemplo, algoritmos de tratamiento de imágenes, o donde si es posible incorporarlos pero se obtienen visualizaciones demasiado básicas.

La aplicación desarrollada, cumple los objetivos propuestos para este trabajo de fin de grado, los cuales eran la compatibilidad con el estándar DICOM, la reconstrucción de modelos a partir de los archivos resultantes de distintas técnicas de escáner, la representación nítida de dichas reconstrucciones, el desarrollo de una serie de herramientas que ampliasen las funcionalidades de la aplicación y una interfaz de usuario intuitiva.

A continuación, se proponen una serie de líneas de investigación en las que se podría trabajar partiendo de este proyecto.

5.2 Líneas futuras

Una vez finalizado este trabajo de fin de grado y conseguidos sus objetivos principales, se exponen una serie de líneas de investigación a través de las cuales se podría mejorar la aplicación desarrollada. Algunas de estas líneas son:

5.2.1 Integración total con DICOM

Tal y como se ha comentado a lo largo del trabajo, el estándar DICOM es un estándar muy amplio que cubre todo tipo de casos límite. Es por esto, y pese a que la aplicación desarrollada cubre las partes más comunes y empleadas del estándar relativo a imagen, que podría ampliarse para trabajar con distintas transformaciones no lineales, permitir el uso de formatos de imágenes comprimidos, habilitar la lectura de imágenes en color, y un largo etcétera.

Además, podría ampliarse la aplicación para seguir la parte del estándar DICOM de comunicaciones, haciendo que la integración en la red de, por ejemplo, un hospital, fuese tan sencilla como la instalación de un nuevo equipo informático. Esto permitiría al usuario acceder a la base de archivos médicos del hospital de una forma segura y cómoda para cargarlos posteriormente en la aplicación.

5.2.2 Incorporación de algoritmos de tratamiento de imágenes

La aplicación se ha dejado preparada para incluir en la reconstrucción de los modelos algoritmos de visión artificial. Estos algoritmos podrían llevar a cabo segmentación de partes del cuerpo, detección de anomalías, o planificación automática de procedimientos quirúrgicos, que podrían ayudar a los profesionales médicos a detectar patologías o a realizar dichos procedimientos.

5.2.3 Integración Online

Sería posible dotar a la aplicación de un sistema multiusuario en línea que permitiese colaborar en un mismo entorno a varios usuarios a la vez, manipulando el mismo modelo y con un sistema de comunicación por voz incorporado dentro de la propia aplicación.

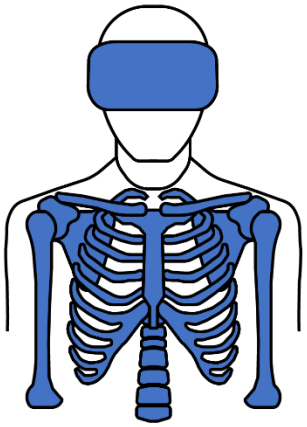
5.2.4 Realidad aumentada

Una vez desarrollada la aplicación y gracias a que Unity permite exportar los proyectos a diferentes plataformas, haría falta un pequeño conjunto de cambios para ejecutar la aplicación en unas gafas de realidad aumentada, que permitirían visualizar e interactuar con los modelos reconstruidos en el mundo real en vez de en un entorno virtual. De esta forma se podría explorar el interior de un paciente en medio de una operación o planificar una cirugía superponiendo el archivo escaneado sobre el paciente real.

Esta línea de investigación se ha empezado a desarrollar empleando un visor de realidad aumentada impreso en 3D de código abierto, *Project North Star* Este visor fue inicialmente publicado por la empresa Leap Motion bajo la licencia GPL 3.0 (Ilustración 67). A través de este visor, se ha conseguido ejecutar la reconstrucción y visualización del modelo, faltando por adaptar los controles de interacción con los objetos interactivos (Ilustración 68).

Ilustración 67: Visor de realidad aumentada Project North Star. Fuente: Elaboración propia.

Ilustración 68: Reconstrucción a través del visor de realidad aumentada. Fuente: Elaboración propia. Archivo: Abdominal CT disponible como dominio público en IDoImaging [38]



Bibliografía

[1] Medixant, «RadiAnt DICOM viewer,» [En línea]. Available: https://www.radiantviewer.com/. [Último acceso: 13 Junio 2020].

[2] Medicalholodeck, «DICOM VIEWER XR,» [En línea]. Available: https://www.medicalholodeck.com/dicom-viewer-virtual-augmented-reality/dicom-viewer-xr-medical-virtual-reality.html. [Último acceso: 13 Junio 2020].

[3] C. Williams y K. Kovtun, «DICOM VR,» [En línea]. Available: http://www.dicomvr.com/. [Último acceso: 14 Junio 2020].

[4] P. Suetens, «X-ray computed tomography,» de *Fundamentals of Medical Imaging*, Nueva York, EEUU, Cambridge University Press, 2009, pp. 33-63.

[5] R. W. Brown, Y.-C. N. Cheng, E. M. Haacke, M. R. Thompson y R. Venkatesan, «Magnetic Resonance Imaging: A Preview,» de *Magnetic Resonance Imaging, Physical Principles and Sequence Design*, New Jersey, EEUU, Wiley Blackwell, 2014, pp. 1-12.

[6] O. S. Pianykh, «What is DICOM?,» de *Digital Imaging and Communications in Medicine (DICOM)*, Heidelberg, Alemania, Springer, 2012, pp. 3-7.

[7] National Electrical Manufacturers Association (NEMA), «6 DICOM Information Model,» 2013. [En línea]. Available: http://dicom.nema.org/dicom/2013/output/chtml/part04/chapter_6.html. [Último acceso: 23 Abril 2020].

[8] National Electrical Manufacturers Association (NEMA), «6.2 Value Representation (VR),» 2013. [En línea]. Available: http://dicom.nema.org/dicom/2013/output/chtml/part05/sect_6.2.html. [Último acceso: 23 Abril 2020].

[9] National Electrical Manufacturers Association (NEMA), «N.2 Pixel Transformation Sequence,» 2013. [En línea]. Available: http://dicom.nema.org/medical/dicom/current/output/html/part04.html#sect_N.2. [Último acceso: 22 Abril 2020].

[10] National Electrical Manufacturers Association (NEMA), «C.11.1 Modality LUT Module,» 2013. [En línea]. Available: http://dicom.nema.org/dicom/2013/output/chtml/part03/sect_C.11.html#sect_C.11.1.1. [Último acceso: 26 Abril 2020].

[11] National Electrical Manufacturers Association (NEMA), «C.11.2 VOI LUT Module,» 2013. [En línea]. Available: http://dicom.nema.org/dicom/2013/output/chtml/part03/sect_C.11.html#sect_C.11.2. [Último acceso: 25 Abril 2020].

[12] National Electrical Manufacturers Association (NEMA), «C.11.4 Presentation LUT Module,» 2013. [En línea]. Available: http://dicom.nema.org/dicom/2013/output/chtml/part03/sect_C.11.html#sect_C.11.4. [Último acceso: 27 Abril 2020].

[13] National Electrical Manufacturers Association (NEMA), «C.7.6.3.1.2 Photometric Interpretation,» 2013. [En línea]. Available: http://dicom.nema.org/medical/dicom/current/output/chtml/part03/sect_C.7.6.3.html#sect_C.7.6.3.1.2. [Último acceso: 25 Abril 2020].

[14] J. F. Hughes, A. Van Dam, M. McGuire, D. F. Sklar, J. D. Foley, S. K. Feiner y K. Akeley, «Ray Casting and Rasterization,» de *Computer Graphics Principles and Practice*, New Jersey, EEUU, Addison-Wesley, 2014, pp. 387-451.

[15] W. E. Lorensen y E. Cline, Harvey, «Marching cubes: A high resolution 3D surface construction algorithm,» *ACM SIGGRAPH Computer Graphics,* vol. 21, pp. 163-169, 1987.

[16] J. Pawasauskas, «Volume Visualization With Ray Casting, CS563 - Advanced Topics in Computer Graphics,» 18 Febrero 1997. [En línea]. Available: http://web.cs.wpi.edu/~matt/courses/cs563/talks/powwie/p1/ray-cast.htm. [Último acceso: 15 Junio 2020].

[17] Microsoft, «Microsoft Docs: Input-Assembler Stage,» 31 Mayo 2018. [En línea]. Available: https://docs.microsoft.com/en-us/windows/win32/direct3d11/d3d10-graphics-programming-guide-input-assembler-stage. [Último acceso: 15 Junio 2020].

[18] Microsoft, «Microsoft Docs: Vertex Shader Stage,» 31 Mayo 2018. [En línea]. Available: https://docs.microsoft.com/en-us/windows/win32/direct3d11/vertex-shader-stage. [Último acceso: 15 Junio 2020].

[19] Microsoft, «Microsoft Docs: Pixel Shader Stage,» 31 Mayo 2018. [En línea]. Available: https://docs.microsoft.com/en-us/windows/win32/direct3d11/pixel-shader-stage. [Último acceso: 15 Junio 2020].

[20] Microsoft, «Microsoft Docs: Output-Merger Stage,» 31 Mayo 2018. [En línea]. Available: https://docs.microsoft.com/en-us/windows/win32/direct3d11/d3d10-graphics-programming-guide-output-merger-stage. [Último acceso: 15 Junio 2020].

[21] P. Fuchs, G. Moreau y P. Guitton, «Introduction to virtual reality,» de *Virtual Reality: Concepts and Technologies*, Boca Raton, FL, CRC Press, 2011, pp. 3-10.

[22] D. Heaney, «VentureBeat,» 5 Mayo 2019. [En línea]. Available: https://venturebeat.com/2019/05/05/how-virtual-reality-positional-tracking-works/. [Último acceso: 16 Abril 2020].

[23] S. LaValle, «Oculus Developer Blog,» 22 Mayo 2013. [En línea]. Available: https://developer.oculus.com/blog/sensor-fusion-keeping-it-simple/. [Último acceso: 15 Abril 2020].

[24] A. Melim, «Oculus Developer Blog,» 4 Noviembre 2019. [En línea]. Available: https://developer.oculus.com/blog/tracking-technology-explained-led-matching/. [Último acceso: 16 Abril 2020].

[25] E. Williams, «Hackaday,» 21 Diciembre 2016. [En línea]. Available: https://hackaday.com/2016/12/21/alan-yates-why-valves-lighthouse-cant-work/. [Último acceso: 16 Abril 2020].

[26] Microsoft, «Microsoft Docs: Inside-out tracking,» 10 Diciembre 2017. [En línea]. Available: https://docs.microsoft.com/en-us/windows/mixed-reality/enthusiast-guide/tracking-system. [Último acceso: 17 Abril 2020].

[27] J. Heshch, A. Kozminksi y O. Linde, «Facebook AI,» 8 Agosto 2019. [En línea]. Available: https://ai.facebook.com/blog/powered-by-ai-oculus-insight/. [Último acceso: 17 Abril 2020].

[28] A. Melim, «Oculus Developers Blog,» 20 Septiembre 2019. [En línea]. Available: https://developer.oculus.com/blog/increasing-fidelity-with-constellation-tracked-controllers/. [Último acceso: 17 Abril 2020].

[29] P. Fuchs, «Comfort and Health,» de *Virtual Reality Headsets – A Theoretical and Pragmatic Approach*, Leiden, Paises Bajos, CRC Press, 2017, pp. 97-115.

[30] E. Langbehn, P. Lubos y F. Steinicke, «Evaluation of Locomotion Techniques for Room-Scale VR: Joystick, Teleportation, and RedirectedWalking,» de *VRIC '18*, Laval, Francia, 2018.

[31] D. Whittinghill, B. Ziegler, J. Moore y T. Case, «Nasum Virtualis: A simple Technique for Reducing Simulator Sickness in Head Mounted VR,» de *GDC*, San Francisco, EEUU, 2015.

[32] R. Riener y M. Harders, «Introduction to Virtual Reality in Medicine,» de *Virtual Reality in Medicine*, Londres, Inglaterra, Springer, 2012, pp. 1-12.

[33] A. L. Butt, S. Kardong-Edgren y A. Ellertson, «Using Game-Based Virtual Reality with Haptics for Skill Acquisition,» *Clinical Simulation in Nursing,* vol. 16, pp. 25-32, Marzo 2006.

[34] N. Moorhouse, T. Jung, X. Shi, F. Amin, J. Newsham y S. McCall, «Pulmonary Rehabilitation in Virtual Reality for COPD Patients,» de

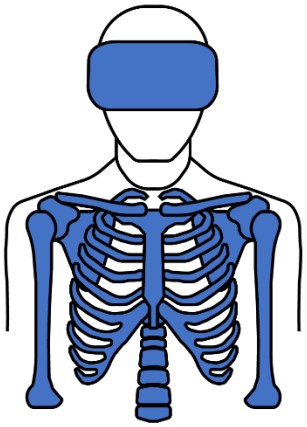
Augmented Reality and Virtual Reality: The Power of AR and VR for Business, Cham, Suiza, Springer, 2019, pp. 277-290.

[35] C. Botella, J. Bretón-Lopez, B. Serrano, A. García-Palacios, S. Quero y R. Baños, «Treatmen of flying phobia using virtual reality exposure with or without cognitive restructuring: Participants' preferences,» *Revista de Psicopatología y Psicología Clínica ,* vol. 19, nº 3, pp. 157-169, 2014.

[36] Extend Reality Ltd., «VRTK,» 29 Marzo 2019. [En línea]. Available: https://github.com/ExtendRealityLtd/VRTK. [Último acceso: 20 Julio 2019].

[37] R. Cardan, «Evil-DICOM,» 17 Abril 2017. [En línea]. Available: https://github.com/rexcardan/Evil-DICOM. [Último acceso: 22 Julio 2019].

[38] A. Crabb, «I Do Imaging Wiki,» 14 Mayo 2017. [En línea]. Available: https://wiki.idoimaging.com/index.php?title=Sample_Data. [Último acceso: 23 Junio 2019].



Anexos

Tal y como se ha comentado a lo largo del documento, se pueden encontrar en los anexos de este trabajo de fin de grado los siguientes documentos:

- Documentación del código en formato HTML.
- Documentación del proyecto de Unity.
- Video del uso y funcionalidades de la aplicación.