UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

Graduado en Ingeniería del Software

Desarrollo de una aplicación web para la generación de imágenes fotorrealistas a partir de escenas modeladas en 3D

Development of a web application for the generation of photorealistic images from 3D modeled scenes

Realizado por
Jose Mª Sánchez Fernández

Tutorizado por
Rafael Marcos Luque Baena

Departamento
Lenguajes y Ciencias de la Computación

MÁLAGA, junio de 2021

Resumen

Actualmente los navegadores se han convertido en una herramienta que nos facilita ejecutar todo tipo de aplicaciones directamente en la nube sin necesidad de instalarlas o almacenarlas en nuestra máquina. Esto facilita la capacidad de ofrecer todo tipo de aplicaciones con mucho potencial que se pueden acceder desde cualquier navegador moderno. Pudiendo hacer aplicaciones que necesiten una potencia de computación mayor al público común que no posea una máquina que alcance estas características o hacerla accesible desde cualquier dispositivo, existiendo así, una gran oportunidad de investigación e innovación.

Así, este proyecto se centrará en el desarrollo de una aplicación web, la cual, a partir de un modelo 3D, que se podrá cargar desde cualquier máquina, renderizará el modelo de manera que genere una visión fotorrealista, con objeto de que sea lo más parecida a una imagen real. Teniendo como objetivo principal la obtención de una imagen que parezca real a partir de un modelo generado por ordenador.

Este renderizado de imágenes fotorrealista aporta valor y uso en diferentes mercados, como podrían ser: en la arquitectura, para mostrar un diseño realista de una futura construcción; en el diseño de interiores, pudiendo mostrar una distribución de muebles concreta; e incluso en la medicina, para cargar y visualizar modelos de órganos. Hay muchos más ejemplos de las áreas que podrían beneficiarse de esto, siendo estas una de las pocas que se beneficiarían de esta aplicación.

La necesidad de este proyecto surge de la falta de aplicaciones web que consigan este objetivo de manera sencilla, ya que las herramientas actuales necesitan una máquina potente para esta carga de trabajo. El hecho de desarrollar la herramienta como una web permitirá derivar la carga de trabajo a un servidor externo, lo que aliviará de una computación intensiva al equipo que está ejecutando la aplicación web. Todo esto buscando un acercamiento sencillo y gratuito.

Palabras claves: desarrollo web, fotorrealismo, node.js, Blender, three.js

Abstract

Currently, browsers have become a tool that makes easier running all kinds of applications directly in the cloud without having to install or store them on our machine. This facilitates the ability to offer applications with great potential that can be accessed from any modern browser. Also, allowing us to make applications that need a greater computing power than the common house machine, making it accesible for all that does not have a machine with these characteristics or accesing it from any device, moreover, creating a great opportunity for research and innovation.

Thus, this thesis will focus on the development of a web application, which from a 3D model that can be loaded from any machine, will render the model in a way that generates a photorealistic image, in order to be as real as possible. The main goal is obtaining an image that is indistinguishable from reality.

This photorealistic rendering provides value and use in different markets, such as: architecture, to show a realistic design of a future construction; interior design, being able to show a specific furniture distribution; or in medicine, to show an organ or some part of the human body. There are many more examples of areas that could benefit from this, these being one of the few that would benefit from this application.

The need for this thesis arises from the lack of web applications that achieve this objective in a simple way, since the tools that allow us to render models usually need a powerful machine. The fact of developing the tool as a web will allow to derive the workload to an external server, which will relieve the team that is executing the web application from an intensive computation. All this looking for a simple and free approach.

Keywords: web development, photorealism, node.js, Blender, three.js

Índice de contenidos

CAPÍTULO 1

Introducción

El renderizado es el proceso por el cual, podemos obtener una imagen de dos dimensiones a partir de un modelo de tres dimensiones, haciendo uso de programas de ordenador. Esta técnica es usada en la construcción de diseños arquitectónicos, en el cine, en los videojuegos y en muchos otros sectores. Las características de este proceso y los pasos que sigue dependen del proyecto, aunque normalmente todas las opciones comparten la misma idea, podemos incrementar la velocidad o eficiencia del proceso a costa de la calidad del producto final. Existen dos categorías de renderizado:

- **Renderizado en tiempo real:** donde las imágenes son generadas al momento y suele usarse en aplicaciones que necesitan hacer uso de gráficos con los que se puedan interactuar.
- **Pre-renderizado:** en esta categoría las imágenes son generadas a una menor velocidad que el renderizado a tiempo real, pero esto nos permite obtener una calidad mejor. Usada cuando la interacción o el tiempo de procesamiento no son primordiales, como en la animación o los renderizados de escenas, donde el *fotorrealismo* necesita obtener la mayor calidad posible.

Entonces, las técnicas de *pre-renderizado*, o renderizado a partir de ahora, nos permiten obtener una mayor calidad de imagen llegando a la posibilidad de obtener una imagen fotorrealista. El fotorrealismo, en lo referido a la renderización de imágenes, se trata de una técnica que busca obtener una imagen que sea capaz de representar una imagen del mundo real de la manera más cercana posible. Por lo que, el renderizado es capaz de obtener una imagen que se asemeje a la realidad, y desde hace un par de décadas, el avance de la tecnología de componentes gráficos y la globalización de los mismos, ha permitido que se innove en la eficiencia y calidad de estas imágenes.

Por lo que en este capítulo vamos a analizar los diferentes factores que propician la necesidad del desarrollo del proyecto que se tratará en este documento. Asimismo, todos los objetivos y la metodología que se seguirá en la creación del proyecto serán descritos a continuación.

1.1. Motivación

Como hemos comentado con anterioridad, en los últimos años los componentes gráficos para el renderizado han aumentado y se han normalizado, pero aún así suelen ser caros para el presupuesto medio de la población y los programas de renderizado suelen ser complejos y muy técnicos, por lo que el usuario debe estar predispuesto a la adquisición del material necesario y al aprendizaje de unos conocimientos nuevos para poder trabajar o tener la capacidad de obtener resultados fotorrealistas.

Existen empresas de mobiliario que tienen catálogos en los que ofrecen todo su inventario, pero este catálogo no es un simple listado de muebles con sus precios, sino que normalmente suelen estar colocados en una habitación en la que podemos observar como quedaría con otros muebles del fabricante o para qué tipo de espacios es más apropiada una decoración. Generar estos catálogos de manera real sería un trabajo muy costoso, tanto en tiempo utilizado como en materiales, entonces, estas *escenas* son obtenidas a través de un modelo generado por ordenador. De esta manera, tenemos un modelo 3D de todos los muebles del inventario de la empresa y podemos colocarlos de las maneras que consideremos necesarias sin la necesidad de que esa habitación exista o que realmente tengamos los muebles.

Aunque el renderizado de estos modelos sea un trabajo más sencillo que montar la distribución de la habitación en la realidad, sigue siendo un proceso complicado y en ciertas ocasiones muy costoso. Los costes de estos procesos aparecen a raíz de que se necesita una máquina capaz de procesar tales cargas de trabajo y es necesario del conocimiento propio para operar con este software.

Por ello, la motivación principal de este proyecto es ofrecer una aplicación que sea capaz de facilitar el proceso de renderizado de escenas, tanto por la derivación de la carga de trabajo a otras máquina como por el uso sencillo e intuitivo. La creación de este catálogo del que hemos hablado anteriormente solo necesitaría de una conexión a internet a través de un navegador y tomar la perspectiva que deseamos renderizar de nuestra escena.

1.2. Objetivos

El objetivo principal de este proyecto es desarrollar una aplicación web en la que el usuario podrá subir un modelo y navegarlo libremente, de manera que seleccione la perspectiva que desea de la escena y con la acción de un botón, la aplicación procese ese modelo en una imagen que sea fotorrealista. Para este renderizado se usarán los parámetros más óptimos que nos permita el software de renderizado para el procesado del modelo, eliminando la necesidad del usuario de conocer como funciona el programa. Este objetivo está dividido en diferentes subobjetivos que nos permitirá afrontar el proyecto de una manera más simple:

- Desarrollar una aplicación cliente que permita cargar modelos 3D y navegarlos libremente a través de un navegador web.

- Desarrollar una API, de manera que cada ruta de esa API sea una funcionalidad que pueda usar el cliente.
- Desarrollar una aplicación servidor que ofrezca la API antes mencionada y sea capaz de procesar los diferentes modelos que le mande el cliente.
- Desarrollar un script que automáticamente renderice de manera fotorrealista el modelo que le pasemos.

Además, la aplicación del cliente deberá ser simple e intuitiva para el usuario. Esto se conseguirá haciendo uso de interfaces de usuario minimalistas, con la mínima cantidad de información técnica en pantalla y que toda interacción con la aplicación aparezca con naturalidad. Asimismo, el servidor estará alojado en una máquina diferente al del usuario en el mejor caso para conseguir desviar la carga del procesamiento de la imagen a un ordenador que esté preparado para ello. En capítulos posteriores analizaremos los diferentes requisitos específicos que deberá satisfacer la aplicación y se realizarán diferentes estudios de su viabilidad y cuales serían las mejores tecnologías y estructuras del software para satisfacerlos, con el fin de cumplir estos objetivos.

1.3. Metodología

Actualmente el desarrollo de software está liderado por el uso de metodologías ágiles, ya que por la naturaleza de los proyectos es el mejor acercamiento actual, permitiendo a las empresas y desarrolladores mantener el ritmo de evolución de la tecnología. Una de estas metodologías es el *desarrollo iterativo e incremental*[1] el cual ha sido usado generalmente para la creación de grandes proyectos. Una repetición de ciclos (iteración) y el desarrollo de pequeñas porciones de software (incremental) permite hacer uso de lo aprendido en ciclos anteriores y poder comenzar con un sistema simple el cual irá haciéndose más complejo hasta tener el sistema completo.

Esto nos permitirá dividir el proyecto en diferentes iteraciones, y tener un prototipo funcional al final de cada iteración con las características implementadas hasta entonces. Así se alcanzará la solución final tras varias iteraciones, que se podrán descomponer en tres factores principales: diseñar y modelar si es necesario las funcionalidades de la iteración; implementar las funcionalidades requeridas para la iteración actual; y hacer pruebas de calidad de las funcionalidades.

Una vez implementadas todas las funcionalidades y comprobado el correcto funcionamiento en un entorno de desarrollo de estas, se continuará a probar la totalidad de la aplicación y todos sus requisitos en un entorno de producción real. En paralelo al desarrollo de la aplicación, se escribirá la memoria con los diferentes conocimientos adquiridos de cada iteración, completándola en su totalidad en la última iteración o en su defecto en una adicional ya que será necesario tener completada y probada la aplicación web. También se generarán los respectivos documentos como el manual de usuario y diferentes explicaciones en el código fuente para facilitar la lectura y comprensión de este.

1.4. Estructura de la memoria

En esta sección vamos a explicar como se va a estructurar esta memoria y que abarcará cada capítulo:

2. Estado del arte En este capítulo vamos a estudiar y analizar el estado actual del mercado para aprender en que contexto se encuentra nuestra aplicación.

2.1. Visualizadores de modelos En esta sección se estudia el mercado actual en lo referido a los visores de modelos 3D en línea.

2.1. Granjas de renderizado En esta sección se estudia que son las granjas de renderizado y el mercado actual de estas.

3. Tecnologías a utilizar En este capítulo se van a estudiar las diferentes tecnologías que se van a utilizar para desarrollar este proyecto.

4. Requisitos En este capítulo se hará un análisis de los requisitos que debe satisfacer el proyecto.

4.1. Análisis de los Requisitos En esta sección se analizarán tanto los requisitos funcionales como los no funcionales que deberá satisfacer la aplicación.

4.2. Casos de Uso En esta sección se generarán los diferentes casos de uso que describirán la interacción entre el usuario y el sistema.

5. Diseño y modelado En este capítulo se estudiará y definirá tanto la arquitectura ideal para la aplicación como el posterior modelado de la misma.

5.1. Arquitectura de la aplicación En esta sección se analizará a partir de los requisitos del sistema la arquitectura más adecuada para usar en la aplicación.

5.2. Modelado de la aplicación En esta sección se generarán los diferentes modelos que resultan del estudio de la arquitectura de la aplicación para facilitar el desarrollo de la misma.

6. Estudio del fotorrealismo En este capítulo se estudiarán los factores que consiguen que una imagen parezca fotorrealista y como los usará el usuario.

6.1. Luz En esta sección se estudiará como se comporta la luz con las imágenes y como obtener el mejor resultado en un renderizado.

6.2. Motor de renderizado En esta sección se describirán los diferentes motores de renderizado disponibles en Blender y que parámetros se usarán.

7. Desarrollo En este capítulo se expondrá el proceso de desarrollo de la aplicación y estará dividido en las respectivas iteraciones que se ejecutaron.

8. Conclusiones En este capítulo concluirá el desarrollo de la memoria con las ideas finales del proyecto, dificultades del mismo y posibles mejoras futuras.

CAPÍTULO 2

Estado del arte

La necesidad de renderizar modelos para el público común no es moderna y existe desde hace tiempo ya, aunque si es verdad, que cada vez más gente posee un ordenador capaz de procesar estas cargas gráficas, las nuevas tecnologías y el uso más común de la web ha facilitado que se puedan crear aplicaciones o servicios que sean capaces de satisfacer esta necesidad. Entonces, la cantidad de empresas que han querido desarrollar una solución para esta necesidad es cada vez mayor y cada una intenta ofrecer un mejor servicio que la anterior pero, ¿cuales son los factores clave que llevan a estas empresas a considerar este mercado como uno viable? Los factores clave para la existencia de estas aplicaciones y servicios son:

- Para la gente común todavía es complicado y caro conseguir una máquina capaz de renderizar imágenes de una manera eficiente.
- Algunos profesionales necesitan más de una máquina para poder renderizar varios modelos a la vez y la adquisición de más máquinas sería imposible de escalar.
- Hay muchos sectores que necesitan de la capacidad de renderizar modelos para la realización de su trabajo pero no poseen los medios y/o conocimientos para hacerlo.

En la antigüedad, el renderizado de modelos o animaciones era un mercado de nicho, en el que comenzaron innovando las diferentes empresas de animación cinematográficas y su uso público no iba más allá de unos pocos entusiastas. Hoy en día el diseño gráfico se ha popularizado y el gran auge de los videojuegos han conseguido que muchas personas empiecen a desarrollar modelos fotorrealistas para los escenarios.

Ante estas necesidades que podemos ver en el mercado sería fácil pensar que está todo ya inventado y bien atado, sin embargo la realidad es bien distinta y podemos encontrarnos todo tipo de aplicaciones tanto de pago como gratuitas, y servicios que compiten entre ellos en ofrecer los mejores precios o resultados. El objetivo principal de este capítulo es estudiar el mercado actual de ofertas de renderizado y/o visualización de modelos, que podemos aprender de ellos y como se podría innovar en este campo.

Como nuestra aplicación busca satisfacer unos requisitos muy concretos, podríamos dividir el mercado actual es las dos principales funcionalidades de nuestra aplicación: la capacidad de visualizar un modelo 3D y navegarlo libremente, y la más importante que sería la opción de renderizar el modelo con los parámetros y perspectivas que deseemos.

2.1. Visualizadores de modelos

Nuestra aplicación pretende, como ya hemos comentado en otras ocasiones, facilitar el renderizado de un modelo. Sin embargo aunque ese sea el objetivo final, para conseguir facilitar el proceso de qué perspectiva va a tomar la imagen se va a implementar un visor que permita navegar libremente el modelo, por lo que debemos analizar que opciones hay en el mercado actualmente que permita visualizar un modelo 3D sin necesidad de descargar ningún programa en nuestra máquina.

Aunque esta funcionalidad sea importante en nuestra aplicación, este tipo de aplicaciones que solo ofrecen esta funcionalidad no son competencia directa con la nuestra, ya que buscamos ofrecer más que esa única funcionalidad. Aún así, es buena idea estudiar estas aplicaciones para poder aprender de las diferentes ventajas y desventajas de cada una. Así, vamos a analizar algunas de las aplicaciones web más famosas para la visualización de modelos 3D.

2.1.1. three.js editor

Three.js, la librería que vamos a usar en la parte del cliente para visualizar los diferentes modelos, posee un editor de modelos llamado *three.js editor*[2] que permite visualizar modelos y editarlos en la misma web. Es totalmente gratuito y permite exportar los modelos que editemos en diferentes formatos. Aunque el rendimiento no es su punto fuerte, ya que la gran cantidad de funcionalidades que junta acaban ralentizando la web, pero ofrece muchas funcionalidades accesibles de una manera sencilla.

Figura 2.1: Interfaz de *three.js editor*

2.1.2. Online 3D Viewer

El nombre de la aplicación es bastante intuitivo por sí mismo. Este visor admite más formatos que el anterior de three.js y nos brinda un mayor rendimiento de manera que es más agradable usar la herramienta. A diferencia de la aplicación de three.js, *Online 3D Viewer*[3] solo permite visualizar y navegar el modelo, aunque también nos ofrece información sobre el mismo y nos permite ocultar los diferentes objetos que forman nuestro modelo.

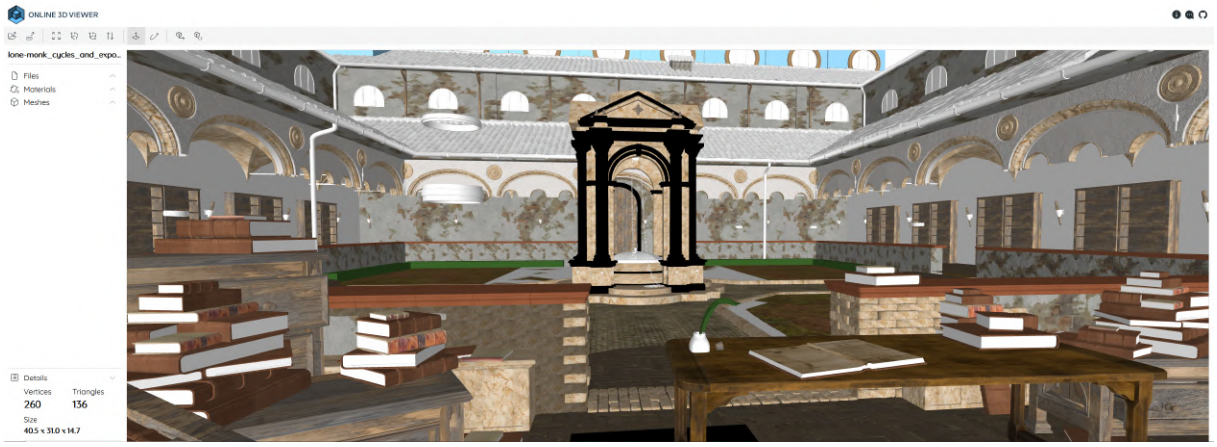

Figura 2.2: Interfaz de *Online 3D Viewer*

2.1.3. Vectary

Por último, *Vectary*[4] es una aplicación web muy completa, requiere que nos registremos en la página y así poder mantener los proyectos que creemos, ya que permite visualizar y editar todo tipo de modelos de una manera sencilla. La parte más importante de esta aplicación es que es capaz de renderizar los modelos siguiendo unos parámetros desde la perspectiva que seleccionemos. Esta aplicación estaría aunando lo que necesitamos para satisfacer nuestros requisitos, sin embargo, la aplicación es más complicada de lo que consideramos necesario y nuestro único objetivo es facilitar la tarea del renderizado, cuando Vectary se centra más en el proceso de modelado.

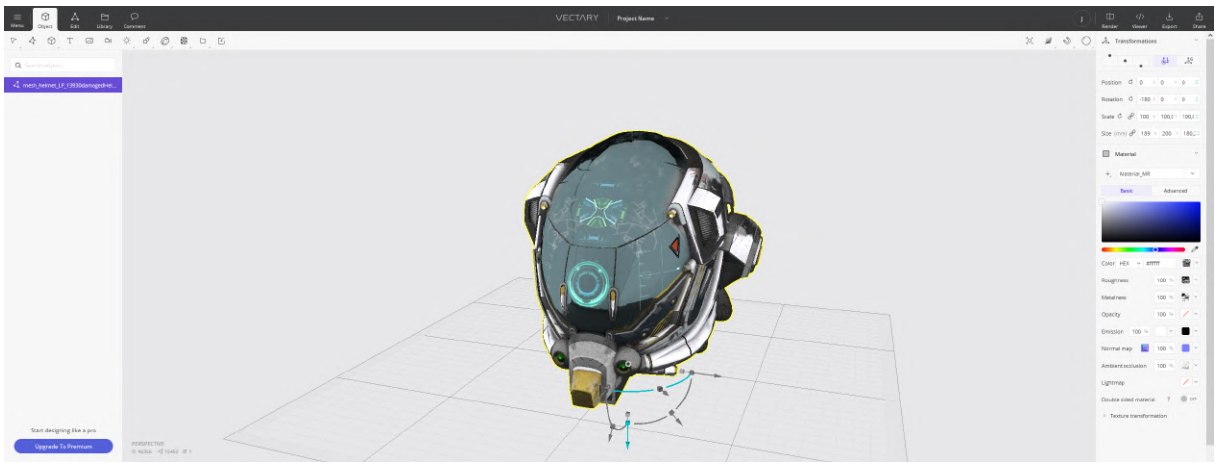

Figura 2.3: Interfaz de *Vectary*

Figura 2.4: Renderizado ejemplo en *Vectary*

2.2. Granjas de renderizado

Una granja de renderizado está constituida por muchos ordenadores que están especializados en el cálculo de imágenes 3D, estos ordenadores suelen llamarse *nodos de renderizado*. Cada uno de estos nodos posee todo el software necesario para desempeñar la tarea de renderizado. Estas granjas suelen hacer uso de múltiples nodos, por lo que cada uno de estos nodos puede trabajar simultáneamente en un proceso de renderizado.

El uso concurrente de muchos nodos de renderizado permite que los renderizados que llevarían días en una máquina única, tardarían solo horas. Aunque estas granjas no solo ofrecen servicios para hacer uso de todas sus capacidades de la manera más óptima, sino que también podemos contratar sus servicios como particulares si no poseemos una máquina lo suficientemente potente para satisfacer nuestras necesidades.

La mayoría de granjas costean sus servicios en un formato de contrato de potencia de GPU o CPU por hora, algunas también tienen a su disposición diferentes configuraciones de máquina para ofrecer una mayor diversidad. Estas granjas son muy cómodas de usar, una vez contratado el servicio suelen poseer un software que se instala en tu máquina local que facilitan su uso. Cuando las escenas están listas, las puedes subir a la granja y automáticamente se distribuye entre los diferentes nodos de la red, y una vez finalizado el renderizado se descarga en la máquina del usuario.

A continuación vamos a analizar algunas de las granjas de renderizado más utilizadas, este estudio nos ayudará en como podemos ofrecer nuestra funcionalidad de renderizado y la diferente retroalimentación que se le es ofrecida a los clientes.

2.2.1. RebusFarm

RebusFarm[5] es un servicio de granja para todos los públicos, tanto para particulares como para estudios. También ofrecen diferentes capacidades computacionales teniendo disponible máquinas con gran capacidad de computación de CPU y de GPU. Esta variedad de oferta hace posible que ofrezcan un servicio muy asequible para todos los públicos, además, tienen una amplia librería de software de renderizado (Blender, Autodesk, Cinema 4D, etc.).

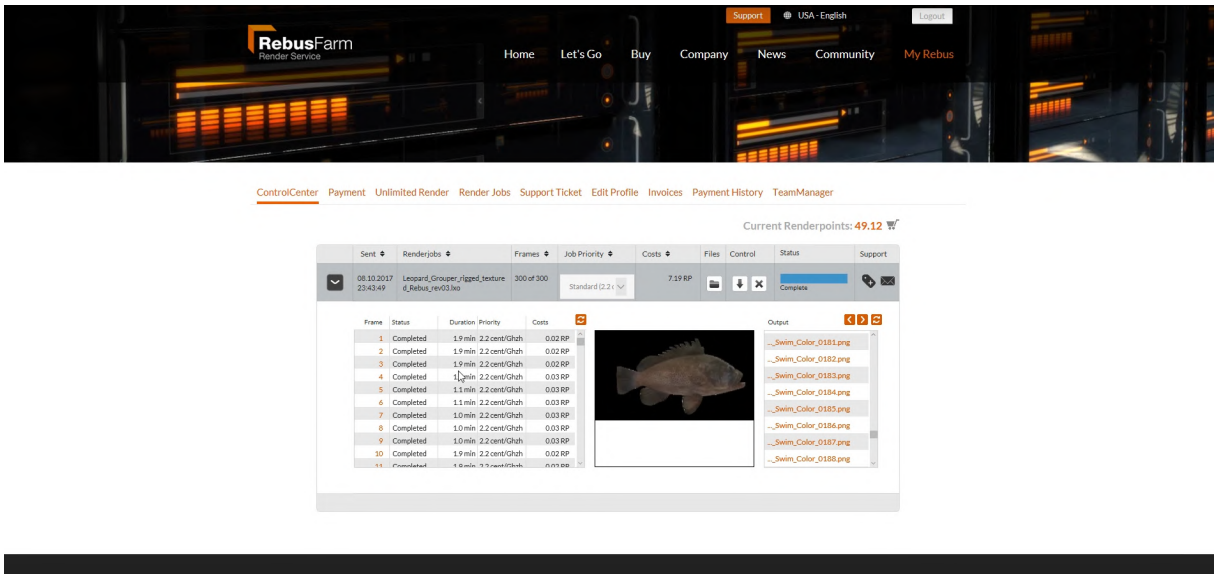

Figura 2.5: Interfaz de proceso de renderizado en RebusFarm

2.2.2. SummuS Render

SummuS Render[6] es una granja de renderizado española que ofrece unos servicios de renderizado *lowcost* y poseen las instalaciones con los equipos que usan, esto es especial ya que con el auge del *SaaS* la mayoría de granjas de renderizado usan máquinas en la nube que orquestan para desempeñar la tarea del renderizado. Además, SummuS posee una herramienta web como podemos apreciar en la figura 2.6 que facilita el uso de la granja y cuenta con tutoriales de formación.

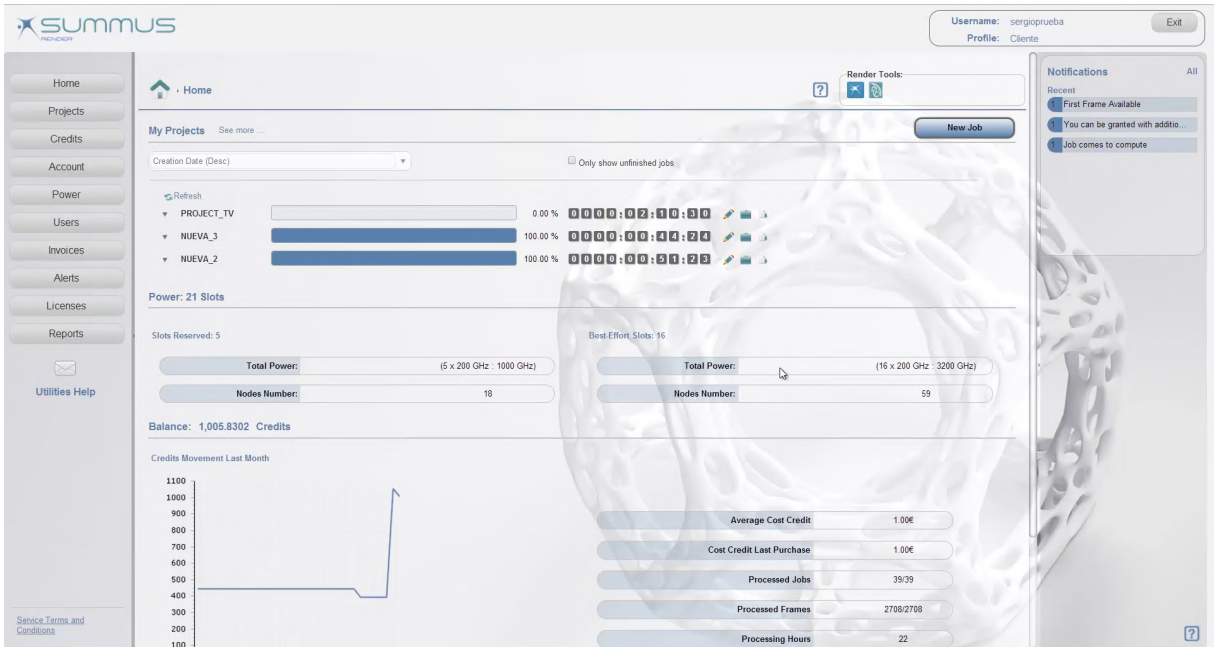

Figura 2.6: Aplicación de la empresa SummuS

Hay muchas más empresas que ofrecen servicios de renderizado pero casi todas cumplen la misma función sin innovar en ningún otro campo. Por tanto, podemos obtener unas cuantas ideas principales de este análisis de mercado:

1. Un visor de modelos puede ser más completo añadiéndole unas cuantas funcionalidades simples pero que aumentan la calidad del uso, como podrían ser la capacidad de editar los objetos del mismo o como podemos visualizar el modelo.

2. Las granjas de renderizado actuales son algo complicadas de usar para el usuario medio o para el público sin demasiados conocimientos en la tecnología (potencia, precios, procesos, etc.). Por lo que podríamos facilitar este proceso al mínimo, mediante la selección de determinados parámetros y pulsar un botón para tener el modelo renderizado como queríamos.

CAPÍTULO 3

Tecnologías a utilizar

En este capítulo se explicarán las diferentes tecnologías usadas para el desarrollo de este proyecto, tanto lenguajes como herramientas para el desarrollo. Todas estas tecnologías han sido estudiadas y seleccionadas según el análisis de la arquitectura de la aplicación realizado en el capítulo 5.

3.1. JavaScript

JavaScript[7] es un lenguaje de programación imperativo, famoso por su gran flexibilidad y potencia de uso. Usado principalmente para desarrollo web también puede usarse en diferentes sectores gracias a la gran cantidad de librerías disponibles. La programación web permite ejecutar la funcionalidad en el lado del cliente sin la necesidad de contactar con el servidor. Esto nos permitirá generar una página dinámica totalmente funcional sin necesidad de un servidor externo, pero el mayor potencial de JavaScript se encuentra en su interacción asíncrona con un servidor remoto, permitiendo la comunicación con este sin interrumpir la interacción del usuario.

Por estas razones este será el lenguaje de programación principal que se usará para el desarrollo tanto de la parte del cliente como la del servidor, esta última aun no siendo el uso más común de JavaScript es posible gracias a node.js y express.js.

3.2. Node.js

Node.js[8] es un entorno de ejecución en tiempo real que permite construir y ejecutar aplicaciones. Es ligero, escalable y multiplataforma. Además, usa un modelo de entrada y salida de datos que hace posible la creación de aplicaciones de red eficientes y escalables.

Usando este entorno como base, los dos pilares principales de la aplicación serán las librerías three.js para el cliente y express.js para el servidor. Estas librerías se pueden instalar de manera sencilla gracias al sistema de gestión de paquetes que trae node.js consigo (npm[9]).

3.2.1. Three.js

Esta librería[10] nos permite mostrar gráficos animados en el navegador web, será usada para el desarrollo del cliente donde podremos generar figuras y formas, animarlas o incluso importar un modelo completo. Esta última funcionalidad será la principal que se use en el proyecto, permitiéndonos crear un visor que muestre el modelo en el navegador. Aparte de las herramientas de generación de figuras y modelos, three.js trae consigo diferentes funcionalidades de control de navegación, edición de modelos y métricas de eficiencia.

Three.js usa *WebGL* para dibujar en 3D, esta API implementada en JavaScript fue diseñada para la renderizar gráficos 3D en el navegador sin necesitar hacer uso de otros componentes adicionales y lo más importante, WebGL está integrada en todos los estándares web, por lo que permite funciones como el aceleramiento del hardware y el procesamiento de imágenes como parte de un elemento HTML.

Esta librería distingue las diferentes partes de un modelo y su visualización en diferentes elementos que harán referencia a la figura 3.1:

- **Rendererizador:** este sería el elemento principal, ya que a partir de una *escena* y una *cámara*, el renderizador dibuja una porción de la escena 3D que sería la visión de la cámara en una imagen 2D.
- **Escena:** este objeto tiene una estructura de árbol y puede estar compuesta por diferentes elementos como se puede ver en la figura 3.1 como *mallas*, *luces*, *objetos 3D*, *grupos de objetos* y cámaras.
- **Malla:** los objetos malla representan una *geometría* con un *material* en concreto, estos últimos pueden ser usados por múltiples mallas.
- **Geometría:** este objeto representa la información de los vértices de alguna geometría en particular, como una esfera, un cubo, un árbol, etc. Three.js ofrece unas geometrías primitivas, pero se pueden añadir adicionales.
- **Material:** es un objeto que representa las propiedades de la superficie que se usan para dibujar una geometría, como el color que tiene o como debe comportarse la luz con ella.
- **Textura:** estos objetos representan imágenes y pueden ser referenciadas por los materiales para que sean mostrados en los diferentes objetos.

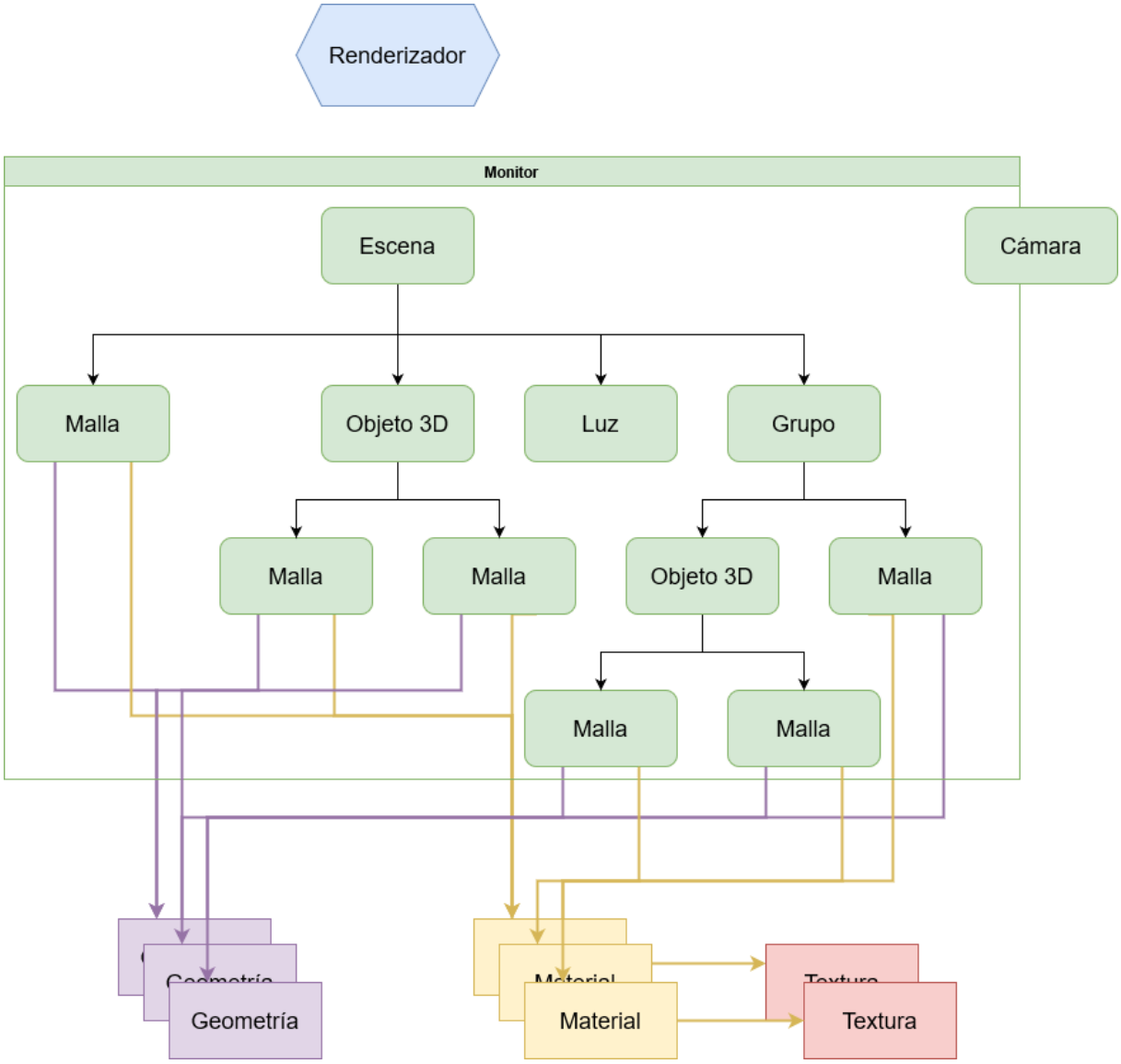

Figura 3.1: Elementos de three.js

Por lo que la forma en que haremos uso de esta librería será generando una escena a partir del modelo que el usuario desee y mostrarla en el navegador mediante el uso de la cámara que tuviese el modelo, o en su defecto, una generada automáticamente. Para después colocar las luces que necesite y se exporte el modelo con las nuevas cámaras o luces, si es que existen.

3.2.2. Express.js

Express.js[11] es una librería del *framework* node.js que nos permite crear aplicaciones de servidor de manera rápida y sencilla. Es simple, minimalista, flexible y escalable, características que ha heredado en su mayoría de node.js.

Aunque su uso principal sea para la creación del lado del servidor, usaremos express.js también en el lado del cliente, para poder generar un cliente de manera sencilla y estructurada, definiendo las diferentes rutas de los archivos y librerías que se usarán. De

esta forma, express.js nos permite construir una aplicación que sirva y maneje peticiones HTTP de manera sencilla, como se ha mencionado antes, será usado también para la creación del cliente para facilitar obtención de archivos estáticos del mismo, como la página principal *index.html*.

```
const express = require('express')
const app = express()
const path = require('path')
const PORT = process.env.PORT || 8080

app.use(express.static(__dirname + '/public'))
app.use(
  '/build/',
  express.static(path.join(__dirname, 'node_modules/three/build'))
)
app.use(
  '/jsm/',
  express.static(path.join(__dirname, 'node_modules/three/examples/jsm'))
)

app.listen(PORT, () => console.log(`Listening on ${PORT}`))
```

Figura 3.2: Código fuente del servidor del cliente generado con Express.js

En la figura 3.2 podemos observar como se genera el servidor del cliente, solo necesitamos importar la librería y crear un objeto nuevo de la misma, después comprobamos si hay algún puerto como variable de entorno y sino ponemos uno por defecto, y ya solo nos quedaría definir las rutas que usará la aplicación (como la librería three.js y los archivos estáticos) y comenzar a aportar servicio en la dirección por defecto.

3.3. HTML5

HTML5[12] es un lenguaje de marcado usado para definir y mostrar el contenido de una web, haciendo uso a su vez de CSS para personalizar por completo todos sus elementos nos permite desarrollar web de manera sencilla.

Se utilizará para desarrollar la parte del cliente y se procurará hacer la página web de la manera más simple y básica, usando únicamente la librería Bootstrap para facilitar el estilo de la página.

3.4. CSS

CSS[13] o *Cascading Style Sheets* nos permite describir como se muestran los diferentes elementos HTML en la página web. Podemos usarlo dentro de los mismos elementos que queramos alterar o podemos generar un archivo independiente donde esté todos los estilos del cliente.

3.5. Blender

Blender[14] es un programa de código abierto y gratuito que nos ofrece todo lo necesario para el desarrollo 3D: modelado, animación, simulación y renderizado. A parte de su uso común con la interfaz gráfica, este programa ofrece una API propia que permite ejecutar *scripts* desarrollados en Python, permitiéndonos desarrollar herramientas automáticas o ejecutar Blender de manera *headless*.

Por lo que en nuestra aplicación utilizaremos Blender para el renderizado de los modelos 3D que emita el servidor a través de la API de Blender y mediante *scripts* de Python. El objetivo principal es desarrollar llamadas a la API que se encarguen de todo el procesamiento del modelo, sin necesidad de la intervención del usuario, más que para la selección de parámetros.

Como hemos comentado, el uso que le hará la aplicación de Blender será en todas las ocasiones de manera *headless* o el término en español, sin cabeza. Cuando hablamos de sistemas electrónicos, este término indica que el sistema no tiene una interfaz local o lo que es lo mismo, no posee ningún tipo de periférico que permita el intercambio de información directamente. Esto es muy útil, ya que el servidor procesará el modelo automáticamente y de manera remota, por lo que no existe la necesidad de estas funcionalidades, lo que nos confiere un pequeño aumento de rendimiento.

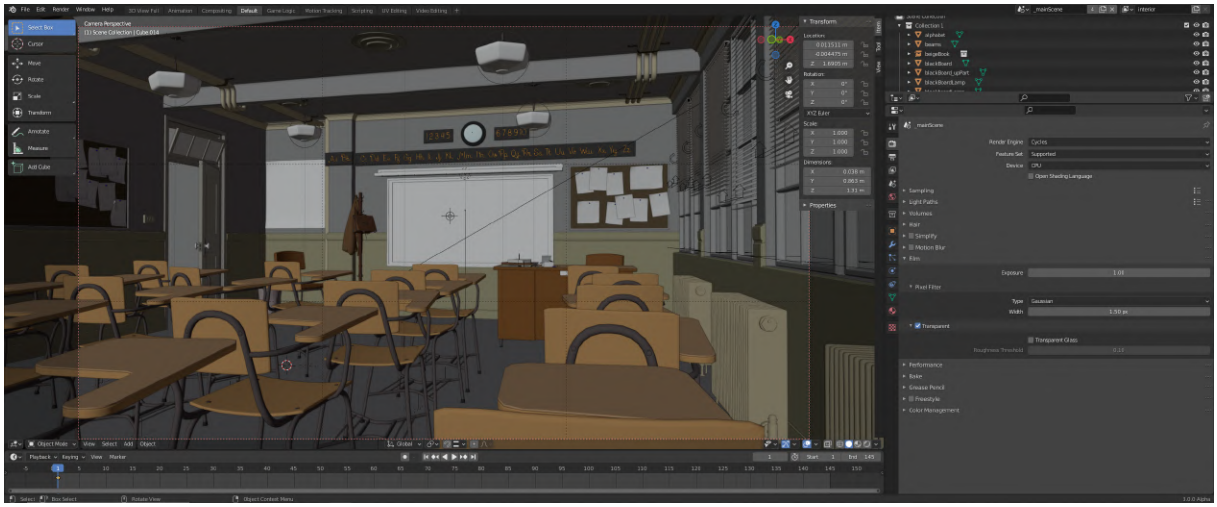

Figura 3.3: Interfaz de Blender

Sin embargo, el estudio, las pruebas y la documentación que se tome del programa, se harán a través de su interfaz gráfica la mayoría de las veces, ya que ofrece una facilidad mayor a la hora de trabajar con ella a diferencia de su API. Y como podemos ver en la figura 3.3, la interfaz de Blender ofrece todo tipo de herramientas para facilitar el proceso de modelado, como la adicción de formas predeterminadas a la escena, transformaciones de objetos, generación de scripts, etc.

Por lo que el uso de Blender en esta aplicación será siempre el mismo, se importará un modelo que anteriormente fue procesado en el servidor, se establecerán los parámetros que el usuario solicitó y comenzará el proceso de renderizado, y en la finalización de este, el servidor cerrará la instancia del programa sin almacenar ningún dato de esa sesión.

3.6. Docker

Docker[15] nos permite aislar nuestras aplicaciones del entorno donde se están ejecutando, de manera que facilita la portabilidad y usabilidad de esta en cualquier máquina. Con Docker podremos aislar tanto la parte del cliente como la del servidor y poder ejecutarlas en cualquier máquina o servidor externo como AWS, Heroku o Netlify (solo el cliente).

```
FROM node:14.15

# Crea el directorio de la aplicacion
WORKDIR /usr/src/app

# Instala las dependencias de la aplicacion que estan en el archivo
COPY package*.json ./
RUN npm install

# Copia toda la informacion al contenedor
COPY . .

# Abre el puerto 8080
EXPOSE 8080

# Ejecuta el comando al inicio
CMD [ "node", "app.js" ]
```

Figura 3.4: Dockerfile de la imagen del cliente

En nuestro caso haremos uso de Docker y de *Docker Compose*[16] que nos permite definir aplicaciones compuestas por múltiples contenedores, los cuáles pueden estar contenidos en

una red virtual común pudiendo comunicarse entre ellos. Docker permite crear las imágenes de las aplicaciones haciendo uso de un único fichero llamado normalmente *Dockerfile*, en estos ficheros podemos comenzar a partir de una imagen ya existente o desde cero, en nuestro caso empezaremos desde una distribución de Linux para facilitar el trabajo de instalación. Desde esta imagen, podemos ejecutar comandos como si estuviésemos en la propia máquina contenida y copiar archivos desde la máquina local al contenedor, como se puede apreciar en la figura 3.4.

Entonces, Docker nos facilitará todo el proceso de despliegue de la aplicación, necesitando únicamente tener instalado Docker en la máquina que deseemos, ya que tanto el cliente como el servidor serán orquestados por Docker Compose. Además de que las aplicaciones individualmente estén en contenedores, el contenedor del servidor tendrá instalado y preparado Blender, de manera que no se tendrá que hacer una instalación complementaria a las dos partes de la aplicación.

3.7. Visual Studio Code

Para la aplicación y desarrollo de todas estas tecnologías se usará el editor de código[17] desarrollado por Microsoft, ya que es totalmente gratuito y ofrece múltiples ventajas, como extensiones para el formato del lenguaje, integración con Git, resaltado de sintaxis y muchas más.

3.8. GitHub

GitHub[18] es una versión web del programa de gestión de versiones *Git*. Esta web nos facilita el desarrollo de la aplicación, generando diferentes versiones, pudiendo hacer diferentes ramas de trabajo e incluso desarrollando diferentes *pipelines* que nos permitirán desplegar nuestra aplicación en un servicio en la nube con cada *push* que hagamos.

CAPÍTULO 4

Requisitos

En este capítulo vamos a centrarnos en como deberá ser el comportamiento y las diferentes interacciones de la aplicación, estudio que se realizará mediante la definición de requisitos y casos de uso. Por lo que vamos a analizar y especificar las diferentes funcionalidades principales que se consideran que debería poseer el sistema para que cumpla los diferentes objetivos antes especificados.

A parte de la propia utilidad en los requisitos funcionales, se estudiarán los requisitos no funcionales que nos permitirán mejorar la experiencia de usuario más allá de la utilidad del producto. El objetivo principal de este capítulo es tener claros como debe actuar el sistema para ser capaces de diseñarlo e implementarlo usando las mejores técnicas y tecnologías.

Entonces, este capítulo estará dividido en dos secciones principales: el análisis de los requisitos, en ambos se analizarán los requisitos potenciales y serán clasificados por diferentes prioridades de implementación; y los casos de uso donde se estudiarán las diferentes interacciones que pueden tener los usuarios con el sistema.

4.1. Análisis de los Requisitos

En esta sección se analizarán las necesidades que pueden tener los usuarios objetivo de esta aplicación para hacer un correcto estudio de los diferentes requisitos del sistema. Como antes se ha mencionado, se analizarán los requisitos funcionales (aquellos que definen el comportamiento del sistema o de una parte de él) y los requisitos no funcionales (aquellos que definen restricciones al sistema en forma de cualidades del sistema).

4.1.1. Requisitos Funcionales

Los requisitos funcionales son las descripciones de los servicios que un sistema debe proporcionar, pudiendo ser la lógica que el sistema debe tener u otras funcionalidades de las cuáles dependan el sistema. Los requisitos serán identificados con un código tal que *RFXY*, donde *X* es la distinción de si es un requisito del cliente (C) o si es del servidor (S), e *Y* es la numeración del requisito, esto nos permitirá referirnos a ellos de una manera más sencilla.

RFC01 - Cargar un modelo desde una máquina

El cliente nos permitirá cargar un modelo desde nuestra máquina local. Este modelo se subirá desde la susodicha máquina y se cargará en el cliente de la aplicación, permitiendo su posterior uso por parte del usuario.

RFC02 - Importar un modelo

El cliente nos permitirá tomar un archivo de un modelo e importarlo en una escena de three.js, permitiendo su posterior manipulación.

RFC03 - Exportar un modelo

El cliente nos permitirá, a partir de una escena de three.js, exportar un modelo de esta para que pueda ser enviado al servidor.

RFC04 - Visualizar un modelo 3D

El cliente permitirá visualizar en el navegador web una recreación de un modelo 3D.

RFC05 - Navegar un modelo 3D

El cliente permitirá navegar un modelo 3D a través del uso de diferentes controles como podrán ser el teclado y el ratón, en un ordenador de escritorio común.

RFC06 - Conexión con la API de renderizado

El cliente permitirá la conexión con la API de renderizado que se encuentra en la parte del servidor, de manera que pueda comunicarse con él a través de peticiones HTTP como un servidor REST.

RFC07 - Selección de parámetros de renderizado

El cliente permitirá modificar, desde el navegador, diferentes parámetros que modificarán el proceso de renderizado para obtener diferentes resultados en el servidor. Estos parámetros podrán ser:

- Motor de renderizado: Eevee o Cycles
- Oclusión ambiental
- Resplandor

- SSGI (si se hace uso de Eevee)

Todos estos parámetros de renderizado se desarrollarán de una manera más exhaustiva en el capítulo 6.

RFC08 - Añadir luces al modelo

El cliente permitirá añadir objetos de luz a la escena que esté cargada, pudiendo elegir si son luces direccionales o de punto y posteriormente, modificar su posición e intensidad.

RFC09 - Descarga de la imagen renderizada

El cliente permitirá descargar en la máquina local la imagen del renderizado anteriormente procesado por el servidor.

RFS10 - Aplicar parámetros de renderizado al modelo 3D

El servidor será capaz de aplicar diferentes parámetros de renderizado para adquirir un procesamiento de imagen lo más fotorrealista posible. Estos parámetros son los mencionados anteriormente en el requisito RFC07 y algunos más como el *sampling*.

RFS11 - Ejecución de scripts en Blender

El servidor será capaz de ejecutar diferentes scripts escritos en Python que se aprovechen de la API de Blender y así poder modelar, parametrizar y renderizar modelos en Blender en un estado *headless*.

RFS12 - Renderizado del modelo 3D

El servidor será capaz de renderizar un modelo 3D de manera que genere una imagen fotorrealista del mismo. Esta imagen será almacenada temporalmente para poder ser enviada o procesada.

RFS13 - Cola de usuarios

El servidor será capaz de ofrecer servicio a más de un usuario simultáneamente, de manera que los que vayan llegando serán colocados en una cola FIFO (*First In First Out*) y esperarán a que se resuelva su petición. Todo este proceso vendrá apoyado por una correcta retroalimentación en el cliente.

RFS14 - Estimación temporal

El servidor será capaz de generar una estimación de cuanto podrá tardar en procesarse el modelo actual según los parámetros usados para el proceso de renderizado.

4.1.2. Requisitos No Funcionales

Los requisitos no funcionales son restricciones sobre la funcionalidad o servicios, definiendo propiedades y restricciones del sistema y suelen englobar todo el sistema. Como en los requisitos funcionales, usaremos un código que haga referencia a cada uno de estos requisitos no funcionales, siendo tal que, *RNFX* donde *X* es el número diferenciador.

RNF01 - Cliente fácil e intuitivo

El cliente deberá ser implementado de manera que cualquier usuario no técnico sea capaz de usar la aplicación sin necesidad de conocimientos avanzados en la materia.

RNF02 - Cliente implementado de forma sencilla

El cliente deberá desarrollarse usando la menor cantidad de librerías para generar el HTML, o como se suele expresar, deberá ser implementado de la forma más *vanilla* posible. Por lo que se hará únicamente uso de HTML, CSS y Bootstrap, como se ha mencionado en un capítulo anterior.

RNF03 - Fiable

La aplicación deberá ser fiable, permitiendo que diferentes usuarios puedan usar el sistema simultáneamente sin que esto perjudique a los demás usuarios.

RNF04 - Arquitectura cliente-servidor

La aplicación deberá hacer uso de la arquitectura cliente-servidor, por la cuál, las tareas serán repartidas entre el servidor y el cliente. Esto será desarrollado en profundidad en el capítulo 5.

RNF05 - La instalación y despliegue de la aplicación será simple

La aplicación al completo deberá poder ser desplegada con cierta facilidad en la mayoría de máquinas y hará uso de la mínima cantidad de dependencias de sistema.

4.2. Casos de Uso

Los casos de uso son una descripción, normalmente creada con ayuda de diagramas, de como interactúan los usuarios con un sistema, mostrando desde el punto de vista del usuario como responde el sistema a una petición. Cada caso de uso[19] está representado por una secuencia de pasos, comenzando con un objetivo para el usuario y finalizando con ese objetivo completado.

Así, estos diagramas están compuestos por diferentes elementos:

- **Actores:** cualquier objeto o persona que interactúe con el sistema. Entre ellos se encuentra el **actor principal** que es aquel que comienza la interacción con el sistema.
- **Precondiciones:** condiciones que deben cumplirse antes y después de que ocurra el caso de uso.
- **Disparador o acción de inicio:** es aquella acción que comienza el caso de uso.
- **Escenario de éxito principal:** es el flujo correcto del caso de uso en caso de que todo salga bien.
- **Escenarios alternativos:** estos escenarios son variaciones del escenario principal, estas excepciones ocurren cuando algo no sale bien a nivel de sistema o el escenario de éxito tiene diversos caminos.

Como la clasificación de los requisitos anterior, estos casos de uso estarán marcados con un código diferenciador, tal que, CUX donde X se trata de el número del caso de uso.

CU1 - Subir un modelo

El actor principal de este caso de uso será el usuario final de la aplicación, el cuál, quiere conseguir subir su modelo 3D a la aplicación, para poder aprovecharse de todas sus funcionalidades posteriormente. Por lo que el escenario de éxito principal sería:

1. Entrar en la página de la aplicación a través de un navegador.
2. Seleccionar el botón para subir un modelo o hacer un *drag&drop* del archivo.
3. Esperar a que se cargue el modelo.

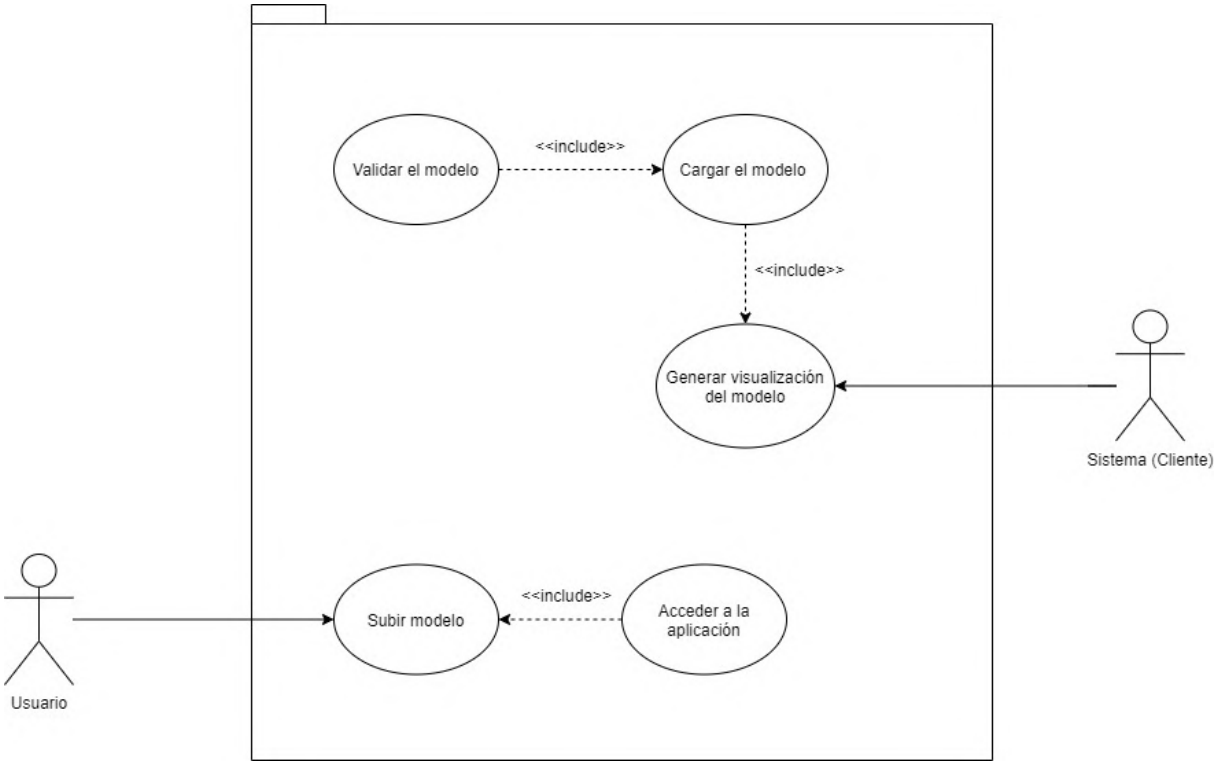

Figura 4.1: Diagrama de caso de uso - Subir un modelo

En un escenario alternativo, la validación del modelo podría dar error, ya sea porque el formato del archivo no es compatible o porque el archivo esté corrupto y no pueda ser cargado. En este caso, el cliente informaría al usuario del problema y cancelaría la carga del modelo.

CU2 - Navegación del modelo y edición

El actor principal de este caso será el usuario final, el cuál, quiere poder navegar por el modelo que está visualizando en el monitor de su máquina y ha cargado anteriormente en el cliente con éxito. Esta navegación le proporcionará diferentes perspectivas del modelo. A parte de la posibilidad de navegar el modelo libremente, el usuario podrá añadir luces al modelo que permitan visualizar mejor la escena y seleccionar los diferentes parámetros que desea para el proceso de renderizado posterior. El escenario principal de éxito sería:

1. Navegar el modelo con los controles de navegación apropiados.
2. Añadir luces al modelo y modificar las propiedades de estas.
3. Modificar los parámetros de renderizado del modelo.

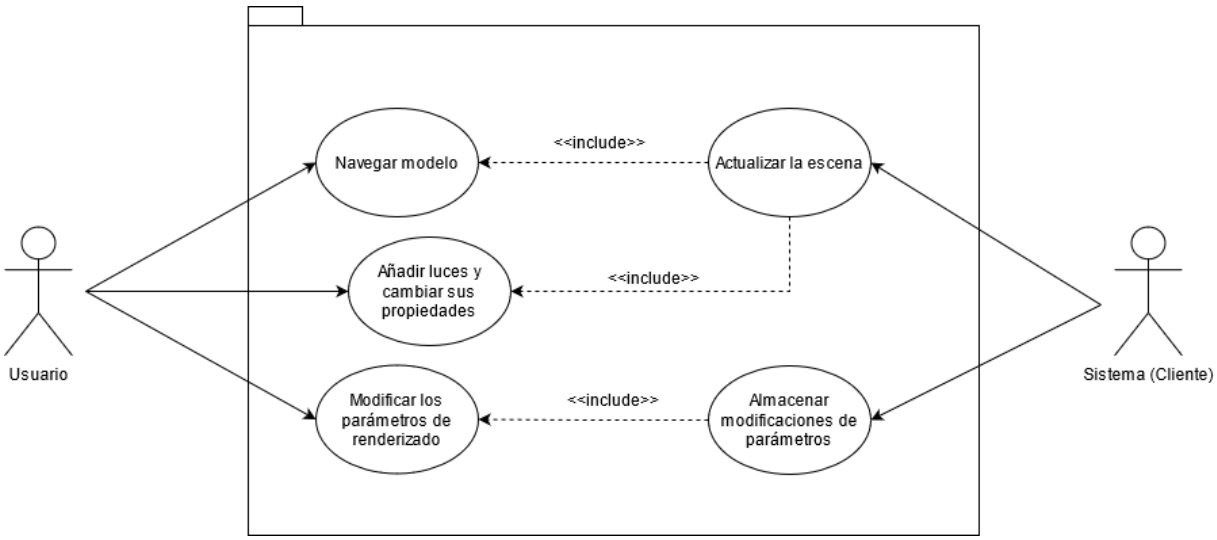

Figura 4.2: Diagrama de caso de uso - Navegación del modelo y edición

CU3 - Renderizado y descarga

El actor principal de este caso será otra vez el usuario final, esta vez, tras haber cargado el modelo satisfactoriamente y haber podido navegar por él usando los controles de navegación que le brinda el cliente, y habiendo seleccionado los diferentes parámetros para el posterior renderizado, quiere renderizar su modelo desde la vista que tiene actualmente del monitor y obtener así una imagen. El escenario principal de éxito sería:

1. Hacer clic en el botón de renderizado.
2. El servidor realiza las labores de renderizado.
3. El usuario pulsa el botón para descargar la imagen.

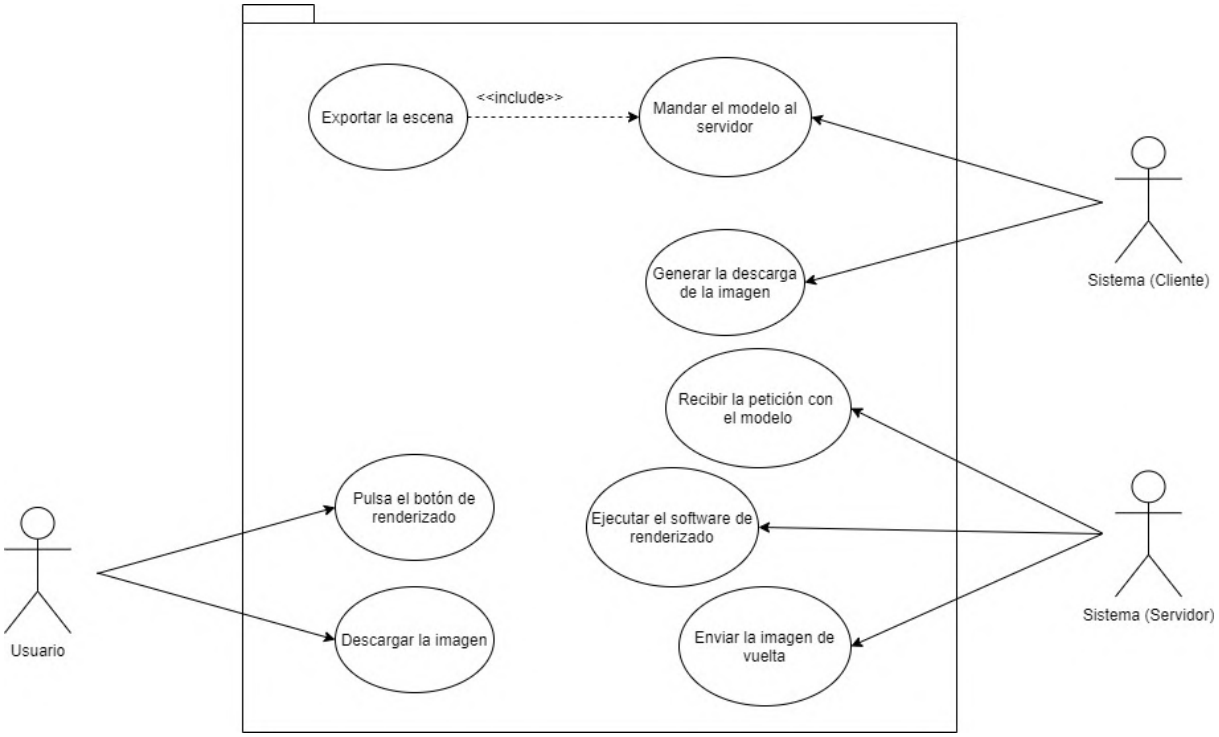

Figura 4.3: Diagrama de caso de uso - Renderizado y descarga

CAPÍTULO 5

Diseño y modelado

En este capítulo se van a desarrollar el modelado y diseño de la aplicación, todo esto se hará en función a el estudio realizado en el capítulo 4 sobre los diferentes requisitos de la aplicación. El diseño previo de una aplicación consiste en definir la estructura e interacción de la aplicación para que la funcionalidad resultante satisfaga los requisitos que hemos analizado. Este diseño deberá incluir una descripción de la arquitectura usada, incluyendo el hardware, las bases de datos y los *frameworks* usados, indicando como interactúa cada parte del sistema entre sí. También se incluirá la definición de la API generada en el servidor siguiendo las normas para su correcta creación.

Tras el estudio del diseño de la aplicación, comenzaremos con el modelado de la aplicación que se centrará en el estudio de las diferentes interfaces de usuario y cuáles serán las distintas interacciones del sistema con los usuarios. Ayudándonos a especificar las abstracciones sobre el mundo real, descubrir los diferentes problemas antes de desarrollar la aplicación y facilitar el desarrollo de la aplicación.

Por lo que este capítulo se dividirá en dos secciones principales:

- Arquitectura de la aplicación, donde diseñaremos la estructura de la aplicación, a raíz de los requisitos analizados.
- Modelado de la aplicación, donde diseñaremos las diferentes interfaces e interacciones de la aplicación.

5.1. Arquitectura de la aplicación

La arquitectura de la aplicación nos permite definir soluciones comprensibles basadas en principios, conceptos, y propiedades relacionadas lógica y consistentemente entre ellas. Esta arquitectura posee características que nos permiten satisfacer, en la medida de lo posible, el problema u oportunidad expresados por los requisitos que hemos analizado, el ciclo de vida de la aplicación y cuáles de estas son implementables.[20]

A raíz del estudio de los requisitos se ha decido hacer uso de un modelo cliente-servidor. Este modelo es una estructura de aplicación distribuida que divide las tareas entre los generadores del contenido (los servidores) y los que lo piden (los clientes). En esta arquitectura, un cliente envía una petición de datos al servidor y este la acepta y devuelve esta información al cliente en paquetes de datos.

Por lo que tenemos dos partes que se dividen la carga de trabajo, veamos una definición más detallada de cada una:

- **Cliente:** en el contexto del software, nos referimos a la máquina *host*, el cuál es capaz de recibir información o usar un determinado servicio que provea un servidor.
- **Servidor:** en el contexto del software, hablamos de un servidor como una máquina, normalmente remota, que provee de información o permite el acceso a determinados servicios.

Esta arquitectura nos brinda multitud de ventajas como que la división de carga entre el cliente y el servidor nos permite ejecutar y obtener diferentes servicios que necesitan una gran potencia de procesamiento sin tener que usar la máquina *host* para ello, ofrece una gran escalabilidad, es seguro y permite el acceso desde cualquier parte de Internet.

A parte de este del uso de este modelo por parte de toda la aplicación, el servidor ofrecerá una *API REST*. En servicio *RESTful* nos permite interpretar el servidor como recursos que pueden ser invocados mediante una URI, estos recursos tienen: un identificador (URI), hiperenlaces hacia él, un estado y una representación. También ofrecen operaciones para que los clientes pueden invocarlas sobre estos recursos. Y como estructura, está formado por capas, de manera que un cliente no puede saber si está conectado al servidor final, reforzando la seguridad.

Para hacer la API REST se usarán tres tecnologías diferentes:

- **HTTP:** como la base de la comunicación que ofrece una interfaz de servicio mediante peticiones GET, POST, PUT DELETE, etc.
- **JSON:** para el intercambio de datos entre el cliente y el servidor.
- **JavaScript:** para hacer la programación tanto del lado del cliente como del servidor, más concretamente el servidor usará el *framework express.js* del que ya hablamos en el capítulo 3.

Con la API REST del servidor ya creada, el cliente será capaz de comunicarse con él a partir de las diferentes peticiones, pudiendo enviar el modelo para que sea renderizado o recibir información sobre la cola de procesamiento del servidor.

En el lado del cliente, haremos uso de una estructura de *aplicación de página única* o en inglés *Single-page application* (SPA). Este tipo de aplicaciones nos permite mantener toda la funcionalidad de la aplicación en una única página en el navegador, de manera que todas las actualizaciones necesarias se harán en tiempo real en la misma página a través de JavaScript. Esta estructura nos brinda multitud de ventajas:

1. El usuario solo se tiene que descargar un archivo HTML del cliente para acceder a toda la aplicación.

2. Acelera la carga de la aplicación en el navegador, ya que es este mismo el que procesa los datos y peticiones del usuario.

3. Mejora la experiencia del usuario al ofrecer una web continua que no tiene saltos entre enlaces o navegaciones complicadas.

Y por último, el formato de modelo que procesará la aplicación será *glTF*. Este formato nos permite trabajar con modelos 3D de manera eficiente y de manera interoperable en las tecnologías web actuales. A parte de la información común que contiene un modelo en un formato más popular como *OBJ*[21], glTF nos permite almacenar información sobre la escena, como la posición de cámaras o luces, y hacer uso de mejores materiales y texturas. Además, la librería three.js posee distintos módulos para trabajar de manera sencilla con este formato, como módulos de importación o exportación.

5.2. Modelado de la aplicación

Una aplicación web es un sistema de información que es capaz de interactuar con el usuario a través de interfaces web. Y como ya hemos visto está dividida en cliente y servidor, estos servidores pueden necesitar de una base de datos para el uso común (*CRUD*), por ello se podría considerar los servidores de aplicaciones como tuplas (servidor y base de datos). Sin embargo, en este caso no es necesario hacer uso de una base de datos, ya que ni los modelos ni las imágenes renderizadas se almacenan, por lo que nuestro modelo no contendrá información de la base de datos.

Así, una vez recogidos los requisitos de la aplicación, el modelado de esta aplicación estará dividido en tres secciones diferentes:

- **El modelo lógico:** que describirá la lógica de la aplicación.
- **El modelo de navegación:** que describirá la composición y navegación del cliente.
- **El modelo gráfico:** que contendrá los detalles de la apariencia de el cliente.

5.2.1. Modelo lógico

El modelo lógico tratará de describir la organización de la lógica de la aplicación, se usarán diagramas *UML* para representar los diferentes componentes del servidor, sus atributos, funciones y relaciones. Este modelo será útil para generar la parte lógica de la aplicación de una manera sencilla. Como nuestra aplicación no poseerá datos persistentes, tampoco poseerá clases como tal, por lo que el modelado UML se separará en los diferentes componentes y como se relacionan.

Por lo que vamos a dividir el modelo en dos partes principales, una será la parte del cliente y la otra será el servidor y los servicios que usa.

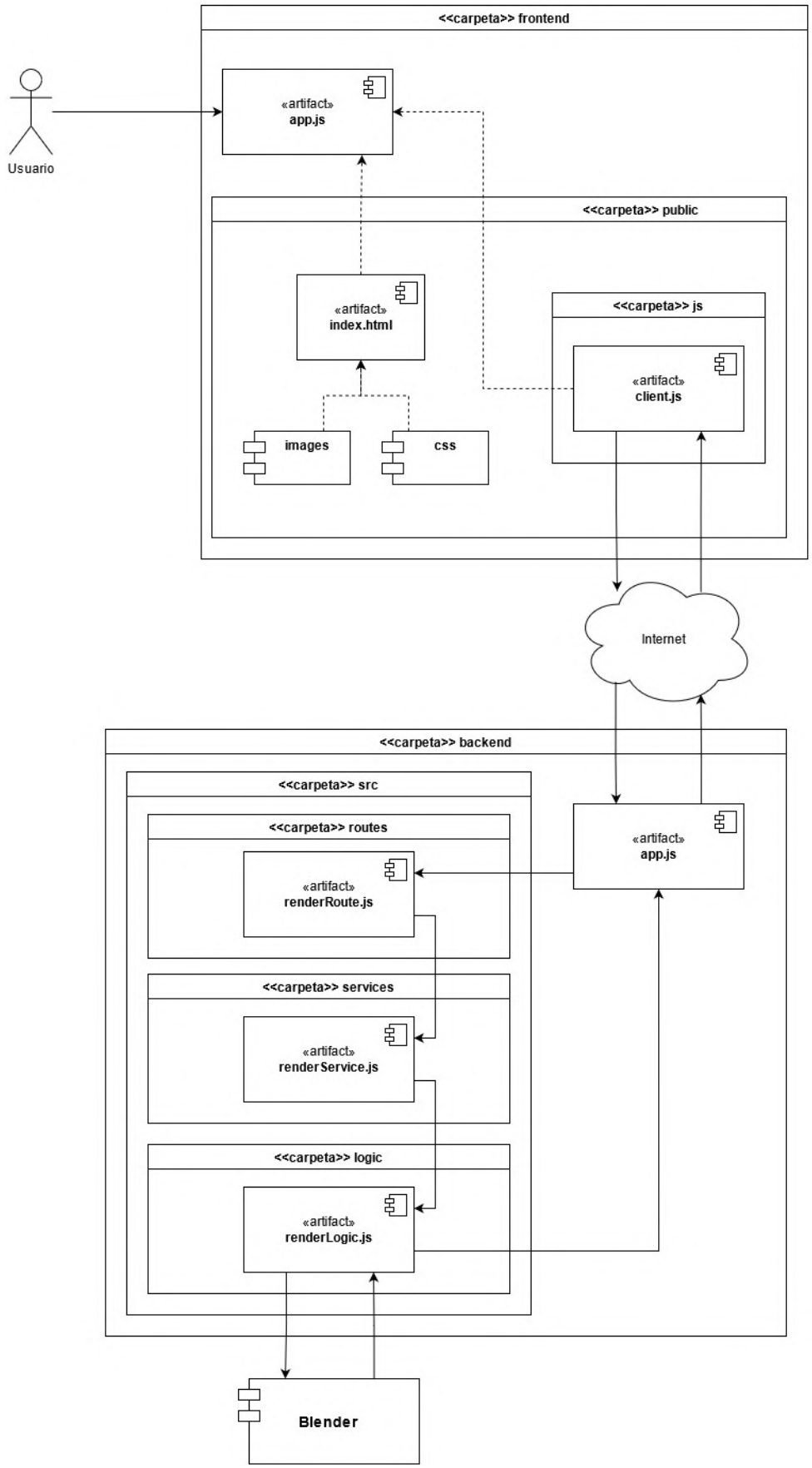

Figura 5.1: Modelo lógico de la aplicación

Como se ha mencionado antes, la aplicación no posee clases diferenciadas para la realización de un diagrama de clases típico, por lo que como se puede ver en la figura 5.1 hemos indicado la estructura principal que va a seguir la aplicación, tanto para el cliente (en el diagrama llamado *frontend*) como para el servidor (en el diagrama llamado *backend*).

Cliente

La parte del cliente tendrá dos partes principales: el proveedor del cliente, que generará la conexión en el puerto correspondiente donde oirá las peticiones; y la parte lógica, la cuál se encargará de generar la página y comunicarse con el servidor.

El proveedor del cliente se encargará de crear una conexión para permitir a cualquier navegador conectarse a él mediante el protocolo HTTP en el puerto que se seleccione. Además, proveerá al resto de la parte del cliente del enrutamiento necesario para acceder a todos los archivos estáticos del sistema, como los scripts JS, los archivos CSS para las páginas o las imágenes que se usen para el visor.

Por otra parte, la parte lógica del sistema se encargará de toda la lógica de negocio necesaria para cargar el modelo, habilitar la navegabilidad de este en un visor y comunicarse con el servidor mediante su API para el envío de modelos y recepción de imágenes. Esta parte también altera los diferentes elementos de la página principal, ya que estamos usando una arquitectura de *Single-page Application* y no necesitaremos más que un archivo HTML para generar toda la web.

Servidor

El servidor tendrá una estructura modular que permitirá diferenciar por completo las partes del servidor en la funcionalidad que cumplen, de esta forma tendremos cada funcionalidad del servidor trabajando independientemente la una de la otra. Esta modularidad nos permite tener la funcionalidad dividida en categorías, lo que nos facilita el desarrollo, la legibilidad y una posterior escalabilidad de una manera sencilla.

Como en el cliente, una parte principal del servidor es la que se encarga de crear la API REST para la recepción de peticiones HTTP, todo esto se hará en *app.js*. Esta parte no solo permitirá el acceso a las diferentes rutas del servidor, también se encargará de enrutar los errores del servidor para poder devolverlos en la respuesta y convertir todos los archivos que sean enviados para su correcto procesamiento.

Por lo que toda la parte lógica de la API se desestructura en tres partes diferentes:

1. **Rutas:** esta parte del servidor será la encargada de generar las diferentes rutas para cada funcionalidad, describirá la URI de las peticiones HTTP y como actuar ante ellas llamando a los diferentes métodos de los servicios.

2. **Servicios:** los servicios serán los encargados de tomar la petición de la ruta, adecuar el cuerpo de la petición en el caso de que fuera necesario y generará la respuesta según los datos obtenidos de la lógica de negocio.

3. **Lógica:** la parada final de la petición se encuentra aquí, en la lógica se procesa la información de la petición y se genera el cuerpo de la respuesta. Aquí será donde el servidor se comunique con la API de Blender para ejecutar la importación y renderizado del modelo.

5.2.2. Modelo de navegación

Para este modelado vamos a usar el lenguaje IFML[22] (*Interaction Flow Modeling Language*). IFML es un lenguaje de modelado inspirado en *WebML* y *WebRatio*, y fue adoptado como estándar por el OMG; nos permite usar diferentes herramientas para la edición gráfica de los modelos e incluso la generación automática de código. Nos permite expresar como interactúa el usuario con la aplicación, como se muestra el contenido de esta y como se comportan los clientes de las aplicaciones web. Sin embargo, no es capaz de describir la representación gráfica del cliente y solo podremos definir la navegabilidad de la aplicación.

El cliente puede ser modelado teniendo un *contenedor* que será la página principal o con una multitud de estos, uno por cada página dinámica. Cada uno de estos contenedores pueden poseer una jerarquía de subcontenedores, generando los diferentes paneles de una página. Estos contenedores pueden contener a su vez *componentes* que denotan los diferentes contenidos o la interfaz de elementos para la introducción de datos como formularios.

Un contenedor y un componente pueden ser asociados mediante *eventos*, los cuales describen la interacción del usuario, como por ejemplo, un componente que representa una lista en la que se pueden seleccionar sus valores. Los eventos son dependientes de la plataforma, ya que por ejemplo usar la rueda del ratón para bajar en la página no será lo mismo que usar el dedo en un teléfono móvil. El efecto de un evento se representa mediante una *flujo de interacción* la cual conecta el evento al contenedor o componente que se ve afectado por él.

Estos eventos pueden activar a su vez *acciones*, estas son ejecutadas antes de la actualización de la interfaz de usuario. Y también existen *dependencias de entrada/salida* entre elementos (contenedores y componentes) o entre elementos y acciones, estas últimas estarán asociadas a un flujo de interacción.

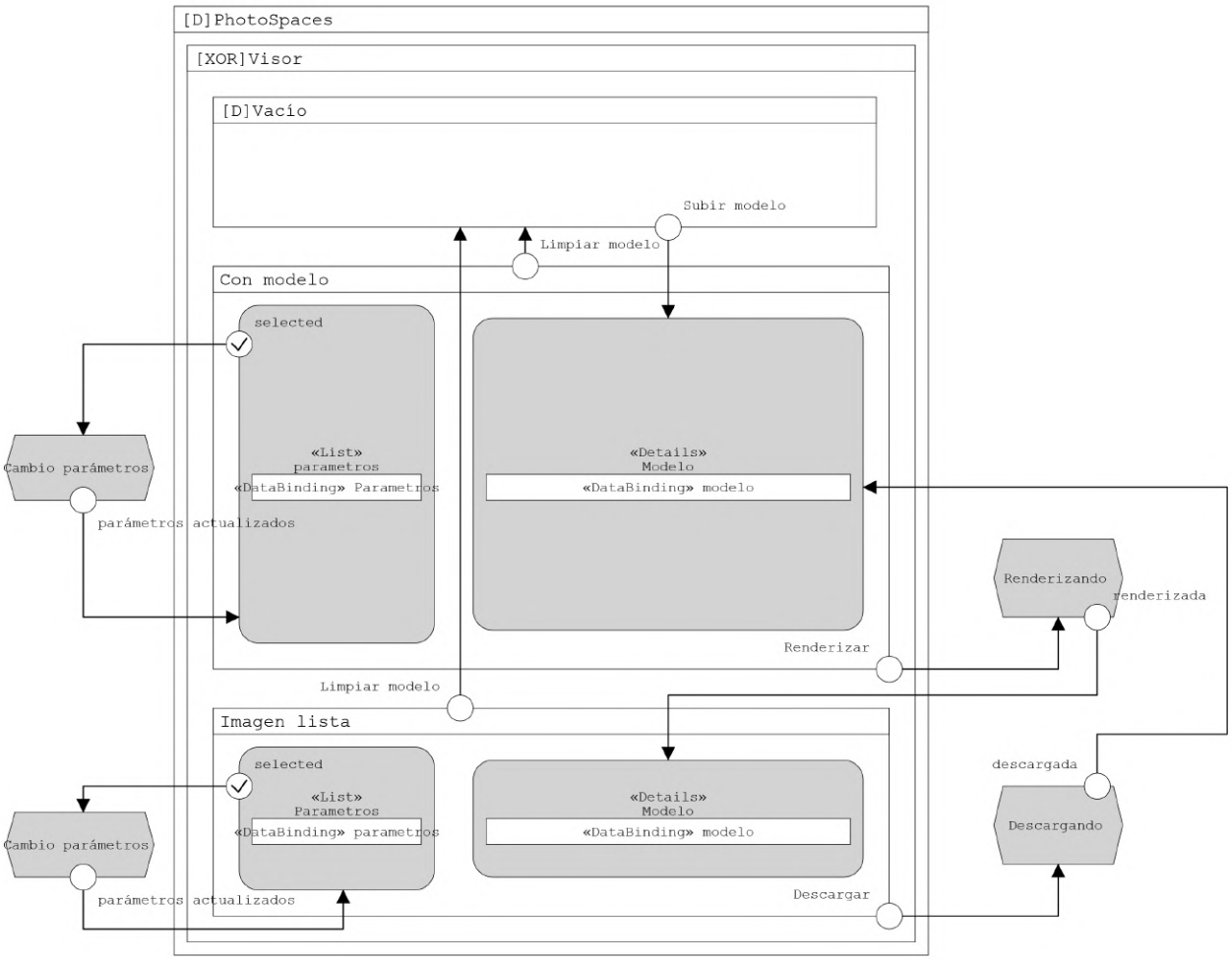

Figura 5.2: Modelo navegable de la aplicación

En este modelo podemos ver que la navegabilidad de la aplicación se ha intentado hacer lo más simple y amigable posible para los usuarios, de manera que, como la aplicación usa una arquitectura *single-page* los elementos de la interfaz se irán alterando conforme el usuario interactúe con ella. La aplicación comienza con el visor vacío y en botón que nos permite subir un modelo a la web, si subimos el modelo y es correcto, el visor mostrará el modelo y se habrá desplegado un panel con instrucciones y parámetros. Desde aquí, podremos movernos por el modelo, editar los diferentes parámetros que tenemos a nuestra disposición o vaciar el visor para subir un nuevo modelo. Una vez tengamos la vista y parámetros que deseemos podemos pulsar en renderizar y cuando se finalice el proceso, descargar la imagen.

5.2.3. Modelo gráfico

Y por último, debemos desarrollar un modelo gráfico que definirá como se va a representar la información en la página web. Para este proceso existen diferentes métodos para satisfacer nuestras necesidades:

1. **Boceto:** estos representan las primeras ideas sobre una funcionalidad o las posibles interfaces que se usarán aunque no son muy extensos ni detallados.

2. **Prototipo software:** se trata de una implementación de un software que reproduciría la funcionalidad de nuestra interfaz, aunque sea muy detallado consumiría mucho tiempo y recursos.

3. **Maqueta digital:** son representaciones de la interfaz en formato digital, para esto se utilizan herramientas de prototipado. Este será el método que usaremos por la velocidad y veracidad de los resultados.

Para generar esta maqueta digital haremos uso de una herramienta en línea gratuita llamada *Marvel*[**mar**]. Esta herramienta nos permite generar rápidamente modelos gráficos de nuestra aplicación e incluso hacer diferentes pruebas con ellos, aunque en nuestro caso solo usaremos su faceta gráfica, ya que la navegabilidad ha sido estudiada con anterioridad en IFML.

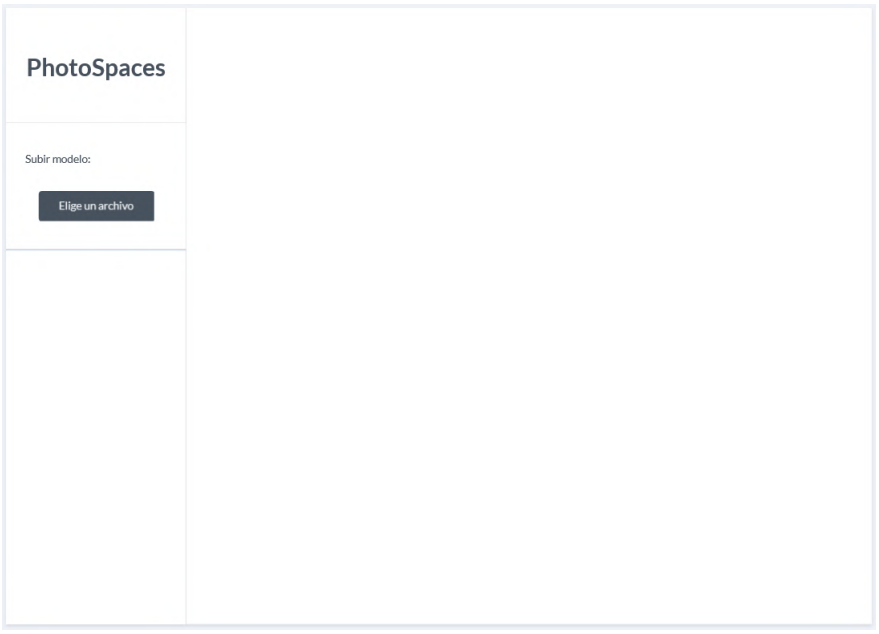

Figura 5.3: Modelo gráfico - Página principal

Como podemos ver en la figura 5.3, la página principal de la aplicación tiende a ser minimalista y concisa, tenemos un panel que contiene las opciones y botones de acción en el lado izquierdo, y nuestro visor (en primera instancia) vacío en el lado derecho. Si pulsamos el botón para subir un modelo se nos abrirá el explorador de archivos de nuestra máquina donde podremos seleccionar el archivo en cuestión.

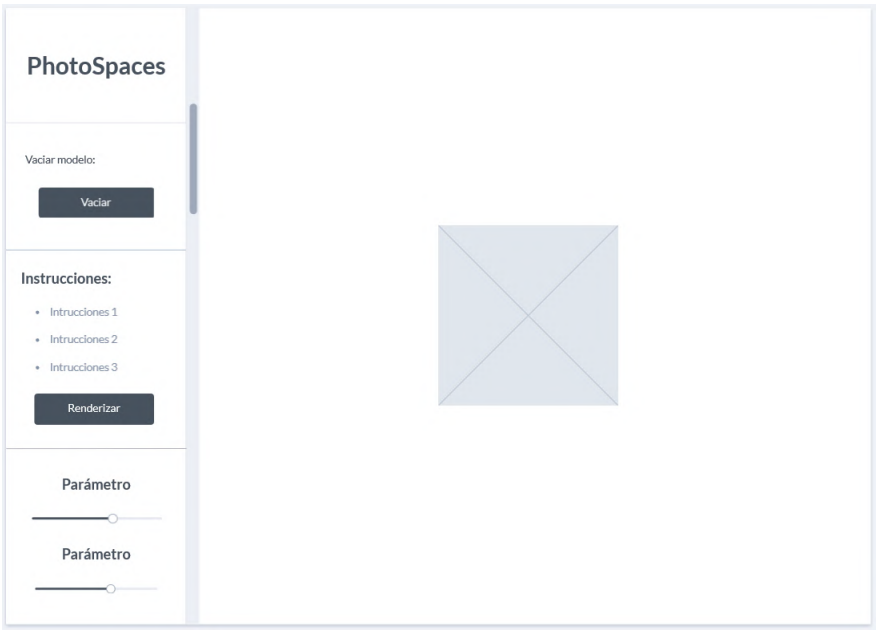

Figura 5.4: Modelo gráfico - Modelo cargado

Una vez cargado correctamente el modelo, este será mostrado en el visor el cuál se podrá navegar siguiendo las instrucciones que han aparecido en el panel. También podremos editar los parámetros del modelo desde el panel de control y si no caben todos en la pantalla, ya sea por la cantidad de estos o por el tamaño del monitor, este panel permitirá un desplazamiento vertical en el mismo. Una vez decidida la perspectiva y parámetros de la imagen, podremos pulsar sobre renderizar.

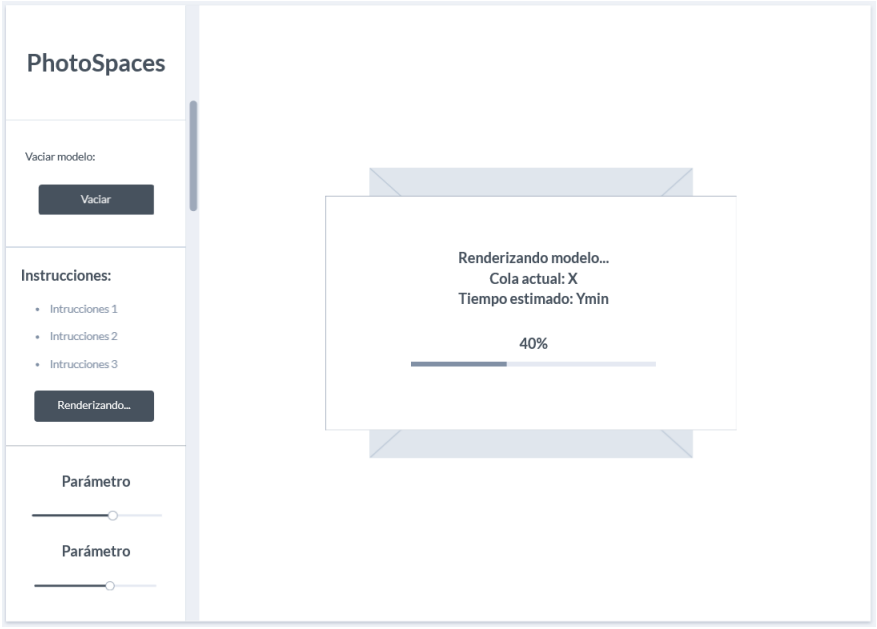

Figura 5.5: Modelo gráfico - Renderizando modelo

Como podemos ver en la figura 5.5 al pulsar en renderizar aparecerá una ventana que nos indicará la cola de peticiones actual del servidor, el tiempo estimado de procesamiento y una barra de progreso. El botón de renderizado estará deshabilitado para evitar que el usuario lo pulse.

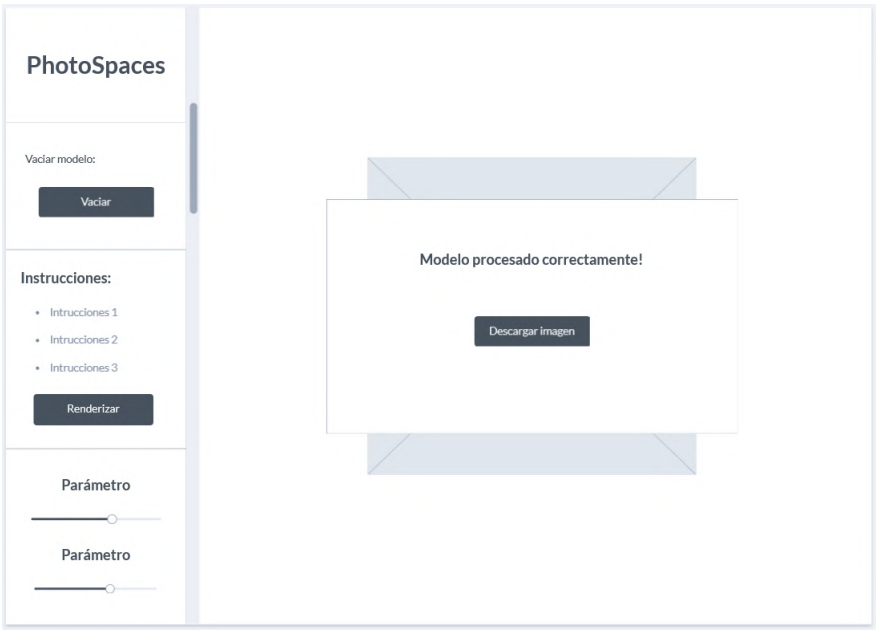

Figura 5.6: Modelo gráfico - Modelo listo para descargar

Finalmente cuando el procesamiento de la imagen haya finalizado la aplicación nos dará la opción de descargar la imagen a nuestra máquina de manera local, desde aquí podemos continuar usando la aplicación cargando un modelo nuevo o renderizando diferentes perspectivas del modelo actual.

CAPÍTULO 6

Estudio del fotorrealismo

Una vez estudiado cuales son las funcionalidades deseadas del sistema y que arquitectura satisfaría estas, debemos analizar una parte muy importante de la aplicación, quizás el factor más importante de todos, el fotorrealismo. El fotorrealismo es, en el contexto del arte, un estilo de pintura en el que se dibuja con multitud de meticulosos detalles que hacen casi imposible distinguir estas obras de una fotografía real; entonces, en nuestro caso no vamos a pintar la imagen como tal, sino que vamos a procesar un modelo 3D para que se asimile en su totalidad a una fotografía real.

Entonces la cuestión sería, ¿que necesita un renderizado para hacerlo indistinguible de la realidad? Como hemos comentado antes una imagen fotorrealista es aquella que no se puede diferenciar de una fotografía real, así que para saber como renderizar una imagen similar a la de una cámara tendremos que saber como funcionan estas. Las partes principales de una imagen serían los materiales y la luz. La luz es el factor condicionante de que podamos diferenciar una imagen real de una que no lo es, ya que la luz interactúa de una forma muy específica con los objetos de nuestro entorno, y nuestro cerebro sabe cuando la luz no está interactuando como debería.

6.1. Luz

La luz es la parte de energía electromagnética que puede percibir el ojo humano y está compuesta por fotones, los cuáles viajan en una línea recta (la mayoría de las veces) hasta toparse con una superficie o un gas. Cuando los fotones se encuentran con un objeto, estos pueden ser absorbidos, rebotar o transmitirse. Por lo que, ¿como interactúa una cámara con la luz? El funcionamiento de las cámaras es similar al de nuestros ojos, y al igual que nosotros poseemos una retina que capta la luz y la transmite al cerebro en forma de señales eléctricas, las cámaras digitales tienen unos sensores digitales que captan la luz y la procesan en una imagen.

Si en una fotografía se encuentran dos elementos, uno brillante y otro oscuro, una cámara solo es capaz de captar la *exposición* de luz de uno de ellos en la fotografía, esto es debido a factores como la velocidad de obturación, luminiscencia de la escena y la apertura del

objetivo. Para recalibrar la imagen, las cámaras deben llevar al objeto oscuro a lo que se llama como *middle gray* o gris medio en español.

Figura 6.1: Ejemplo de objetos en exposición[1]

Como podemos observar en la figura 6.1, la foto de la izquierda esta usando un nivel de exposición adecuado para captar el fondo de la imagen que es luminoso; sin embargo en el lado derecho, al intentar exponer a la persona, la cámara necesita llevarla al gris medio para poder distinguir los detalles, captando más luz a su vez del fondo por lo que este se ve muy brillante y pierde definición. Una cámara es capaz de mostrar elementos más claros y más oscuros del gris medio, pero solo hasta un cierto punto, este rango de visualización es llamado el *rango dinámico*. Cuanto mayor sea el rango dinámico, más detalles se podrán apreciar tanto para las partes claras como para las oscuras.

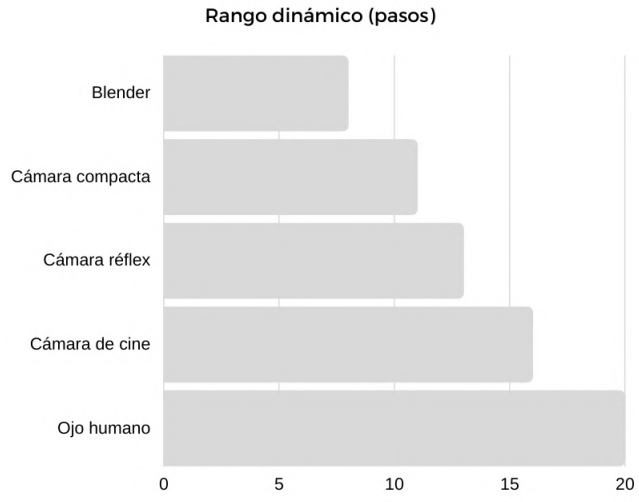

Figura 6.2: Rango dinámico

[1]Fuente de la imagen: Bad Backlighting, 6 and 1/2 ways to deal with it!

Por lo que si las cámaras tienen un rango dinámico que toma una imagen muy veraz de la realidad, solo deberíamos conseguir que Blender tuviera un rango dinámico similar para que los detalles de la imagen fueran más reales. Sin embargo, aunque las cámaras ronden entre los 11 y 16 pasos (como se mide el rango dinámico), Blender en sus preferencias por defecto solo tiene 4 pasos, haciendo que la mayoría de imágenes queden muy oscuras o muy brillantes si no tenemos cuidado en como posicionamos las luces.

Entonces el paso principal para obtener una imagen fotorrealista en Blender sería encontrar una solución al rango dinámico del programa. La solución principal sería modificar como Blender interpreta los colores, esto es debido a que, cuando una cámara toma una foto, esta información pasa por un proceso de *transformación de color* para que los dispositivos puedan mostrar la imagen. Blender funciona igual, y antes de que podamos ver la imagen renderizada, se necesita hacer una transformación de color en colores que nuestro monitor pueda mostrar.

Por defecto, Blender usa una transformación de color llamada *sRGB*, esta fue diseñada para los antiguos monitores EGA, por lo que nunca fue pensada para el trabajo de renderizado y menos para obtener uno fotorrealista. Desde la versión 2.8, Blender trae por defecto la transformación de color llamada *Filmic*, este tipo de transformación es usada actualmente en cinematografía y permite a Blender pasar de un rango dinámico de 8 a 25, pudiendo obtener incluso una mejor visibilidad que las cámaras convencionales.

Figura 6.3: sRGB vs. Filmic

Usando esta transformación de color podemos obtener un mayor rango dinámico como podemos ver en la figura 6.3 teniendo al lado izquierdo la transformación sRGB por defecto que genera un renderizado en el que para obtener una habitación iluminada debemos aumentar mucho la luz, cosa que hace que las partes expuestas a la luz se vean muy blancas y la luz apenas rebote en la habitación. Por el otro lado, usando la transformación de Filmic, podemos aumentar mucho la potencia de la luz para simular el sol y el rango dinámico permite que las zonas expuestas no pierdan detalle y las que no estén expuestas no se vean tan oscuras.

Así, haciendo uso de las nuevas transformaciones de color de Blender podemos conseguir una interacción de la luz más natural, de manera que estemos un paso más cerca del fotorrealismo.

6.2. Motor de renderizado

Renderizar es el proceso de transformar una escena 3D en una imagen 2D, y para esto Blender trae tres motores de renderizado por defecto: *Eevee*, *Cycles* y *Workbench*, aunque el propósito de este último es su uso para el modelado y diseño así que no lo consideraremos. Además de estos motores por defecto, Blender nos permite desarrollar o instalar otros a través de los *add-ons* (son scripts que ofrecen diferentes funcionalidades a las que trae por defecto un programa), sin embargo solo estudiaremos los que trae por defecto Blender ya que son los más estables y son capaces de satisfacer nuestros requisitos por completo.

Cada motor de renderizado ofrece unos parámetros diferentes para modificar la calidad de renderizado y su rendimiento. La calidad de un renderizado viene definida principalmente por las luces (como ya hemos hablado en la sección 6.1) y los materiales, estas características son normalmente compartidas entre los motores, sin embargo algunas funciones solo se encuentran en algunos.

Por lo que en esta sección vamos a analizar que motores gráficos son los mejores para usar, como podríamos usarlos y los diferentes parámetros que podríamos modificar.

6.2.1. Eevee

Eevee es un motor de renderizado en tiempo real que hace uso de *OpenGL*[23], su principal objetivo es que el proceso de renderizado sea rápido e interactivo mientras consigue renderizar satisfactoriamente *materiales PBR* [2]. Este motor nos ofrece una visión de un renderizado real mientras estamos modelando nuestra escena, aunque es una característica muy útil, al usar nuestra aplicación Blender a través de su API esto no nos sirve de nada.

Sin embargo, que no hagamos uso de su renderizado en tiempo real no quita que no podamos aprovechar la increíble velocidad de renderizado del mismo. A diferencia del motor Cycles, que es un motor de trazado de rayos como más tarde comentaremos, Eevee usa un procedimiento llamado *rasterización*. La rasterización[24] es un proceso que se ha usado desde hace mucho tiempo y actualmente obtiene unos resultados muy buenos y su punto fuerte es su velocidad; consiste en convertir los objetos del modelo en píxeles a los cuales se les asigna en primer lugar un valor de color y luego es procesado para adaptarlo a como influye la luz de la escena sobre él.

Aunque Eevee no es perfecto, su velocidad de procesamiento puede ser muy útil como opción seleccionable en la aplicación, de esta forma, el usuario puede elegir si desea que el proceso de renderizado tarde menos tiempo aunque la calidad pueda ser peor. De todas formas, vamos a estudiar como podemos sacar el máximo partido de este motor con los diferentes parámetros que nos ofrece.

[2]Los materiales PBR son aquellos materiales que permiten simular casi cualquier material existente usando un único formato compacto

Oclusión ambiental

Es una característica que genera sombras donde diferentes superficies se juntan, como esquinas o grietas. Esta categoría esta compuesta por la *distancia*, que es como de lejos se tomará en cuenta los objetos para que influyan en la sombra; el *factor*, que mezclará la existencia del efecto y su inexistencia, con valores de 0 para no tener el efecto y 1 para que esté activado por completo; y la *precisión*, a mayor valor, más precisión en el cálculo de las sombras. La oclusión ambiental será un valor que el usuario podrá activar o desactivar. Como podemos ver en la figura 6.4 las sombras bajo el casco se calculan más detalladamente, aunque en este ejemplo no se aprecia mucho ya que en este modelo no existen muchos huecos o grietas.

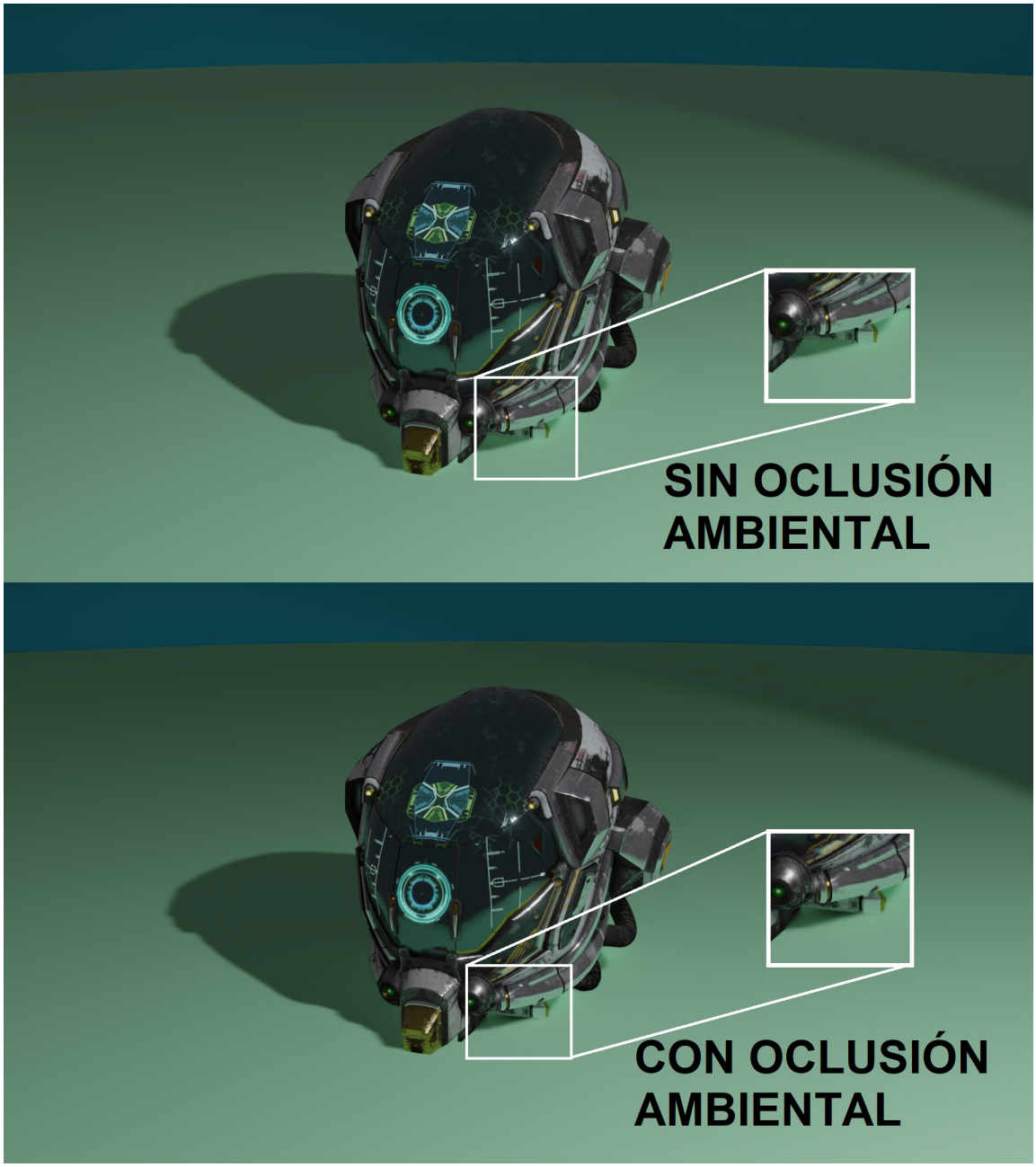

Figura 6.4: Renderizado ejemplo - Oclusión ambiental

Resplandor

Esta característica que se puede activar o desactivar permite hacer un efecto de resplandor en superficies en las que pueda rebotar la luz o que estén muy iluminadas. Para resumir esta característica, se le permitirá al usuario activarla y la intensidad de la misma. En la figura 6.5 podemos observar como al usar este parámetro los reflejos de luz brillan más.

Figura 6.5: Renderizado ejemplo - Resplandor o Bloom

SSGI

Screen Space Global Illumination es una característica que busca crear un sistema de luz que parezca natural, añadiendo luminosidad indirecta a los objetos. Esta característica no es propia del motor Eevee, ya que este proceso suele obtenerse usando el trazado de rayos y ya hemos visto que este motor hace uso de la rasterización; sin embargo, existe un add-on que nos permite añadir esta funcionalidad al motor, obteniendo unos resultados similares.

Figura 6.6: Renderizado ejemplo - SSGI

Este parámetro apenas afecta al rendimiento y el usuario será capaz de activarlo o desactivarlo y con ello también podrá activar la refracción de la luz en diferentes superficies. Este add-on hace un trabajo increíble con el motor por defecto y como podemos ver en la figura 6.6 es capaz de simular como incidiría la luz que rebota del suelo en la parte baja del casco.

Sampling

Sampling o muestreo en español, es el proceso en el cual Eevee recoge datos de la escena por pasadas y luego los une en el renderizado, cuanto más aumentemos este valor más datos sobre la escena obtendremos y en este caso las sombras y texturas estarán más suavizadas. En nuestro caso como el motor es muy eficiente y llegados a un número de muestras el renderizado no mejora, será un valor estático que el usuario no podrá modificar de 128 muestras.

6.2.2. Cycles

Cycles es otro de los motores de renderizado que trae por defecto Blender, a diferencia de Eevee, este motor usa el trazado de rayos para el proceso de renderización. Está diseñado para ofrecer una gran calidad de imagen por defecto y si lo deseamos, ajustar los parámetros para obtener mejores resultados. La capacidad de Cycles de hacer uso del trazado de rayos conlleva a que aparezcan dos factores que serían antagónicos al motor Eevee, este proceso permite hacer una simulación real de como la luz se comporta con los diferentes materiales de la escena; para este proceso podemos elegir la cantidad de muestras que se toman, pudiendo seleccionar cuanta información es usada para el procesamiento.

Este motor se le ofrecerá al usuario en los parámetros de renderizado como una opción que tardará más tiempo en procesarse que haciendo uso de Eevee, pero a cambio, obtendrá una mayor calidad de imagen. Los parámetros que ofrece Cycles son distintos a los que ofrece Eevee en su mayoría, sin embargo comparte algunos como el *sampling*. Así, los parámetros que ofrece Cycles para modificar como se renderiza la imagen no podrán ser modificados desde la aplicación, esto es debido a que este motor tiene un conjunto de parámetros que ofrece la mayor calidad que puede ofrecer este motor al mejor rendimiento, por ello sería inútil que el usuario pudiera aumentar valores sin que estos repercutan en la calidad final pero sí en el rendimiento y una disminución de estos parámetros no obtendrían un mayor rendimiento. Estas razones nos llevan a definir las mejores opciones para el motor, como podemos ver a continuación.

Sampling

De nuevo nos encontramos con la opción de variar el rango de muestreo que hace el motor, pero en este caso a diferencia de Eevee, el número de muestras se refiere cuantos rayos saldrán de la cámara del modelo hacia la escena. Como en Eevee, la idea de este proceso es obtener información sobre todos los objetos que hay en la escena, pudiendo determinar el color de cada píxel de la imagen.

Cycles ofrece una opción de muestreo adaptativo, que consiste en que el propio motor renderiza las diferentes zonas de la imagen hasta que el ruido del sector es mínimo, obteniendo el número óptimo de muestras. Además, tras acabar el renderizado, Cycles ejecuta un proceso para eliminar el posible ruido que haya quedado en la escena, sin afectar apenas al rendimiento final.

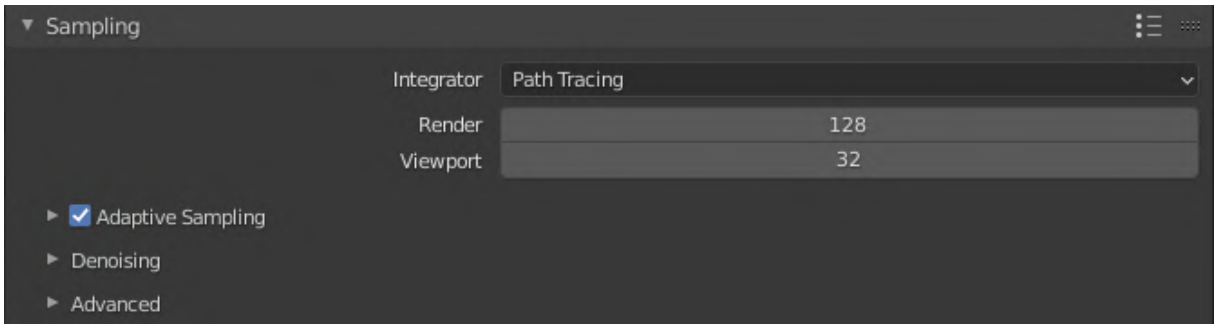

Figura 6.7: Parámetros de *sampling* en Cycles

Trayectoria de la luz

Esta sección agrupa como se comportarán los rayos de luz que son lanzados para el muestreo del modelo pudiendo decidir, como vemos en la figura 6.8, el número máximo de rebotes que puede dar un rayo, a partir de este número de rebotes en superficies, el rayo será eliminado. En la mayoría de escenas un número total de rebotes de cuatro es más que suficiente, ya que no necesitamos tantos rebotes para tomar la información necesaria y a partir de este número, los rebotes adicionales no aportan más información. A parte, el número de rebotes de la luz está respaldado por el muestreo, ya que al usar más rayos para tomar la información, no son necesarios tantos rebotes.

El *clamping* nos permite contener la máxima cantidad de luz que se permite tomar en una muestra, ayudando a reducir la cantidad de muestras de luz que puede llevar en ocasiones a píxeles blancos. En esta parte, la luz directa tendrá desactivada esta contención, pudiendo así obtener un resultado más realista de los objetos que interactúan con luz directa; pero, para la luz indirecta se pondrá un valor de contención, ya que si por alguna razón, una parte de la muestra recibe muchos rebotes de los rayos de luz, esta parte puede quedar mucho más brillante de lo que debería por el efecto de la luz indirecta.

Las cáusticas son el efecto óptico de la luz que ocurre cuando esta atraviesa un objeto trasparente como un vaso de agua y genera un patrón de luz al otro lado. Este efecto es muy deseable y bonito, sin embargo en esta parte de los parámetros suele provocar zonas blancas de luz por la cantidad de luz indirecta y se deben generar muchas muestras para que sea procesado, por ello estará desactivado. Esto no significa que nuestros renderizados se queden sin ese efecto, ya que Cycles lo calcula de una manera simulada (no hace pasar realmente los rayos por ese objeto) para que no afecte al rendimiento y con unos resultados muy buenos.

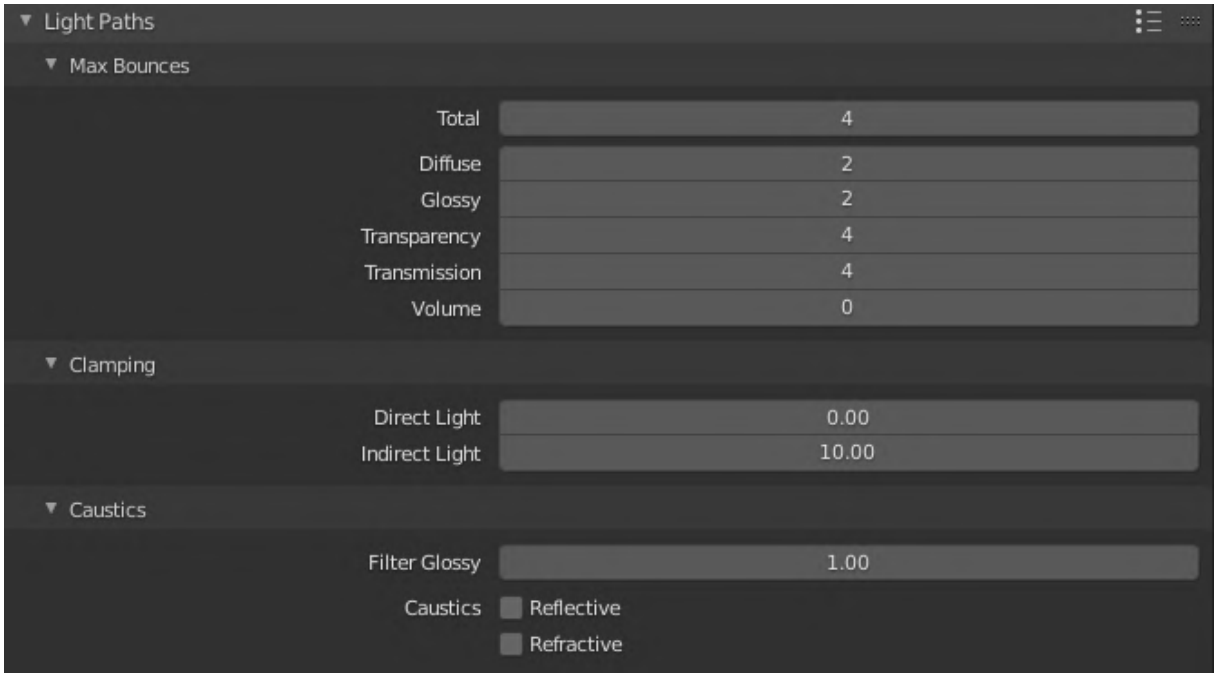

Figura 6.8: Parámetros de la trayectoria de la luz en Cycles

Simplificar

Este parámetro permite a Cycles simplificar el modelo para que el rendimiento mejore sin eliminar calidad del resultado final. La parte del *viewport* la podemos ignorar, ya que estos valores serán aplicados a lo que podría ver un usuario al usar la interfaz de Blender, pero como nuestro uso será *headless* no aplica. En el renderizado tenemos la opción de reducir el número de partículas que son renderizadas o la calidad de las texturas de los objetos, en nuestro caso no se reducirá ninguna de las dos, ya que afectaría a la calidad y no tanto al rendimiento.

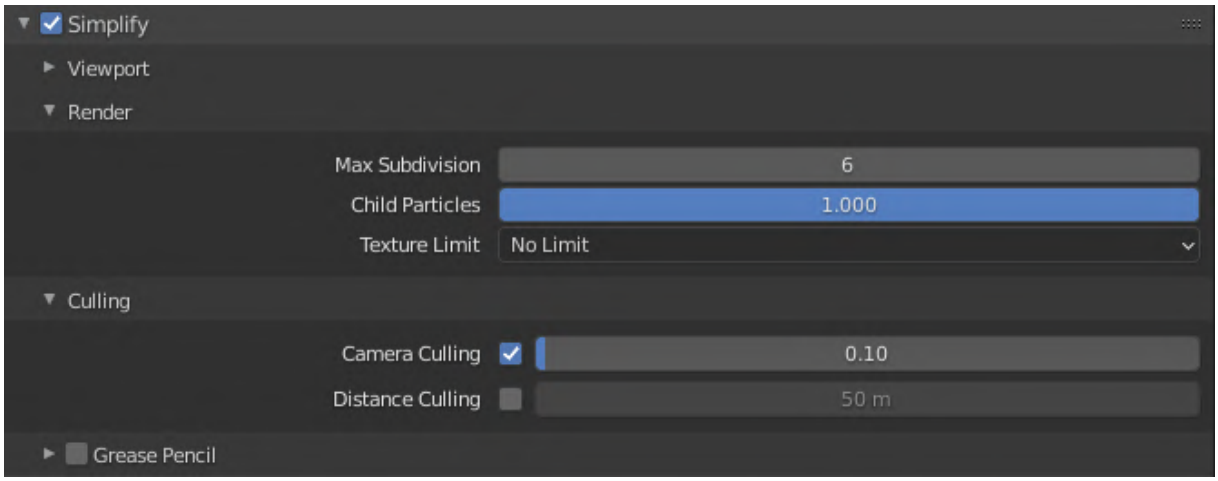

Figura 6.9: Parámetros de *culling* en Cycles

Por último la opción principal en esta sección es la de *culling* o *sacrificar* en español, a pesar de su grotesca atribución, este parámetro permite que en el momento de renderizado, todos los objetos que no estén en el rango de la cámara o muy alejados de ella, sean eliminados o lo que es lo mismo, no renderizados. Esto nos permite renderizar menos objetos en la mayoría de casos, obteniendo una mejora de rendimiento y una continuidad de la calidad, ya que son objetos que no aparecerían en el renderizado final. Entonces, como podemos ver en la figura 6.9 activaremos el *culling* de la cámara a una distancia mínima de su visión, para que elimine la mayor cantidad de objetos.

Tratamiento del color

Como hemos estudiado y analizado anteriormente en la sección 6.1, Blender no posee por defecto un rango dinámico muy bueno pero podíamos aumentarlo haciendo uso de la transformación de color *filmic*, por lo que en esta categoría de los parámetros podemos activarla para obtener unos renderizados más realistas, que al principio serán más oscuros pero al aumentar la intensidad de la luz nos devolverá la información suficiente para las áreas que no están iluminadas directamente por la luz. Estos parámetros también estarán designados de esta forma por defecto con el uso de Eevee.

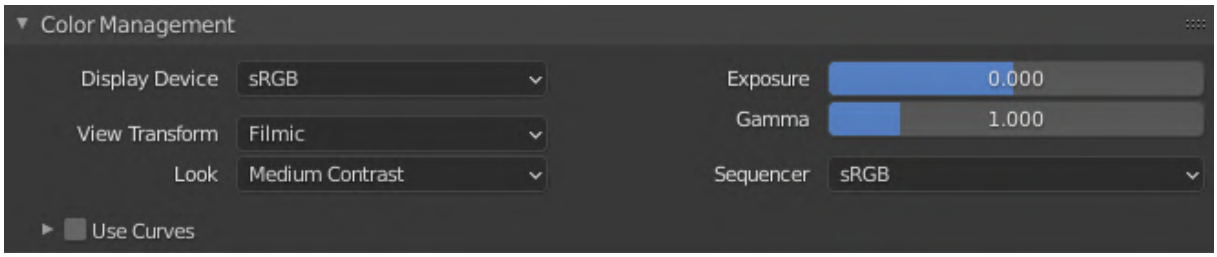

Figura 6.10: Parámetros de tratamiento del color en Cycles

6.3. Conclusiones del estudio

Como hemos podido ver, uno de los factores principales de que un renderizado consiga una calidad fotorrealista es la luz y como los diferentes objetos de la escena interactúan con ella, ya que para nuestro cerebro es muy sencillo detectar cuando la luz no se comporta como debería en una imagen. A partir de aquí analizamos el por qué de que Blender posea un rango dinámico tan bajo por defecto y encontramos la solución de cambiar el método de transformación de color para que obtengamos unos resultados más reales.

A partir de aquí nos hemos centrado en los motores de renderizado que trae Blender por defecto, entre los cuáles hay dos que el usuario podrá elegir a la hora del renderizado. Tenemos Eevee, que aunque inicialmente estuviera creado para renderizar un modelo rápidamente para tener un resultado a tiempo real de nuestro modelo, con los parámetros adecuados y haciendo uso de la extensión SSGI, podemos conseguir unos resultados decentes a cambio de un rendimiento espectacular. Como segunda opción hemos visto el motor de Cycles, muy superior en temas de calidad a Eevee al usar la tecnología de trazado de rayos para obtener unos resultados fotorrealistas con los parámetros por defecto.

Así con todos estos conocimientos podemos implementar un renderizado de imágenes automático que consiga los mejores resultados posibles sin la ayuda humana. Y para finalizar este capítulo, en la figura 6.11 podemos ver un ejemplo de la aplicación de estos parámetros al modelo de una habitación, obteniendo un resultado automático casi perfecto.

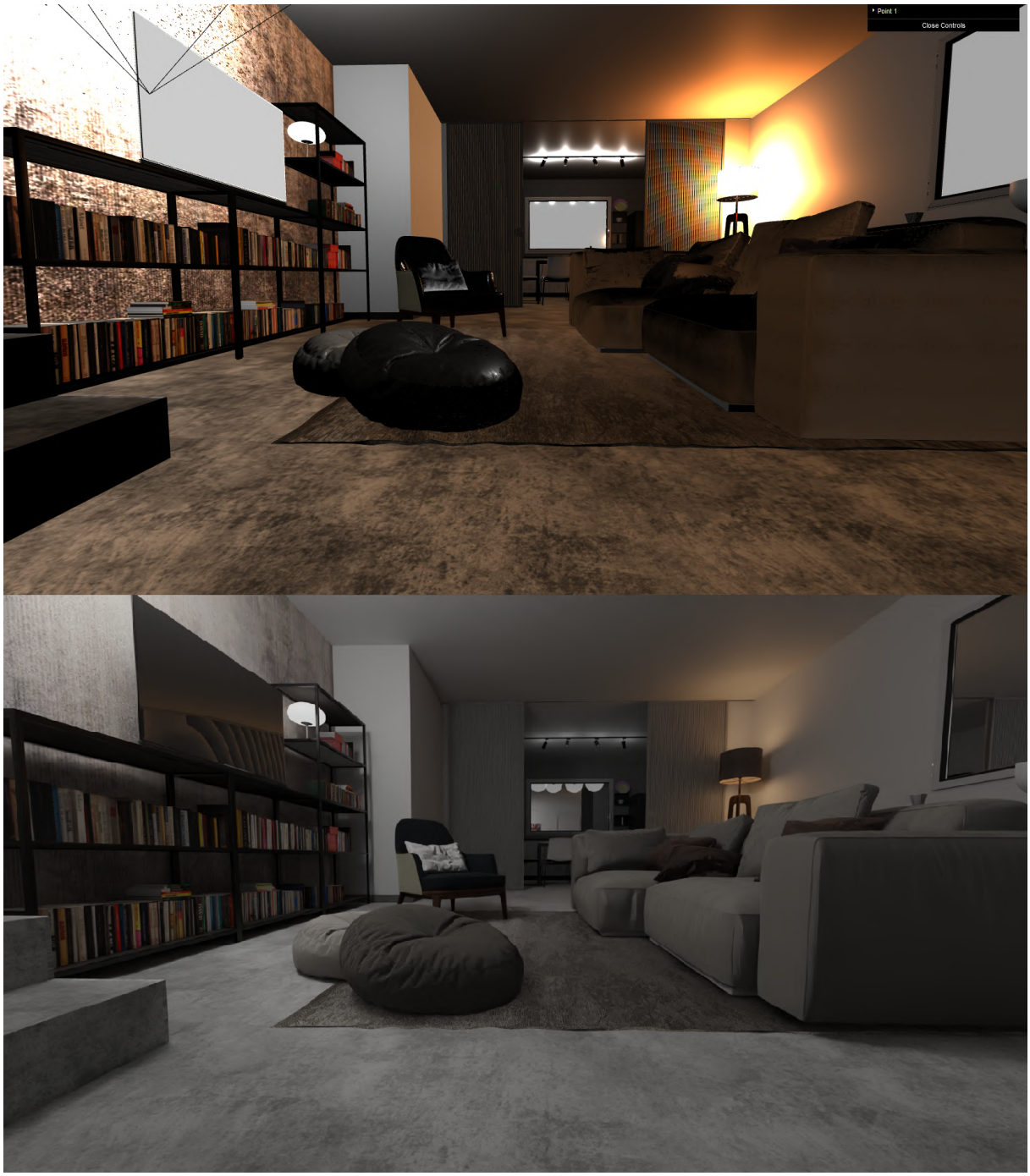

Figura 6.11: Antes y después del renderizado usando Cycles y los parámetros discutidos en la sección 6.2.2

CAPÍTULO 7

Desarrollo

En este capítulo se describirá como se ha procedido al desarrollo de la aplicación para la correcta satisfacción de los requisitos descritos en el capítulo 4. Será dividido en tres secciones diferentes, en las que se analizarán los pasos realizados en cada una de las iteraciones en las que ha consistido el desarrollo de esta aplicación, ya que la metodología usada ha sido el desarrollo iterativo e incremental. Además de la propia implementación de funcionalidades de la aplicación, en cada sección se expondrán los objetivos planteados en la iteración correspondiente y los diferentes retos que se presentaron en cada una de ellas.

7.1. Primera Iteración

En líneas generales, esta primera iteración buscaba obtener unas nociones básicas de como sería trabajar con las tecnologías y arquitectura que propusimos en las primeras fases de desarrollo, a parte de buscar comprender cuales eran las mejores formas de trabajar con la librería de *three.js*.

A parte de esto, también se buscaría obtener una primera API del servidor con la que en iteraciones futuras, el cliente, podrá comunicarse para las diferentes peticiones como podrían ser: tiempo estimado de espera, cola de peticiones y el renderizado.

7.1.1. Objetivos

Los objetivos de esta iteración entonces, eran claros, realizar un primer acercamiento a todas las tecnologías mencionadas y conocer cómo será su correcta manera de interactuar entre ellas. Como prototipo final de la iteración se buscaba obtener:

1. Establecer el flujo de trabajo de la aplicación, realizando un estudio de la estructura de la aplicación y creando un repositorio *Git*.

2. Una aplicación cliente a la que nos pudiésemos conectar y que nos sirviese los documentos estáticos necesarios, además de un visor de modelos mínimamente funcional (pudiera cargar y navegar un modelo).

3. Una aplicación servidor con una API definida y estructurada.

7.1.2. Aplicación del cliente

Para comenzar la implementación de la aplicación se comenzó por la parte del cliente, uno de los pilares más importantes de la aplicación, ya que esta es la parte que contendrá el visor que nos permitirá navegar y visualizar los diferentes modelos que subamos a él, y esta será la perspectiva que será procesada. Entonces el paso principal fue implementar la estructura de ficheros que fue estudiada en el capítulo 5 y hacer la instalación de *node.js* y posteriormente de *express.js*.

Con la estructura de trabajo ya organizada y las herramientas listas, comenzamos con la página de la aplicación, que será la única ya que como hemos mencionado con anterioridad hacemos uso de la arquitectura de *aplicación de página única* o *single-page application*. Esta página principal tenía como objetivo ser muy simple a la vista e intuitiva para el usuario, así que, haciendo uso de los modelos básicos planteados en el diseño de la aplicación se implementó una página para el cliente con una barra de navegación lateral y a su lado, ocupando la mayor parte de la pantalla, lo que será el visor del modelo.

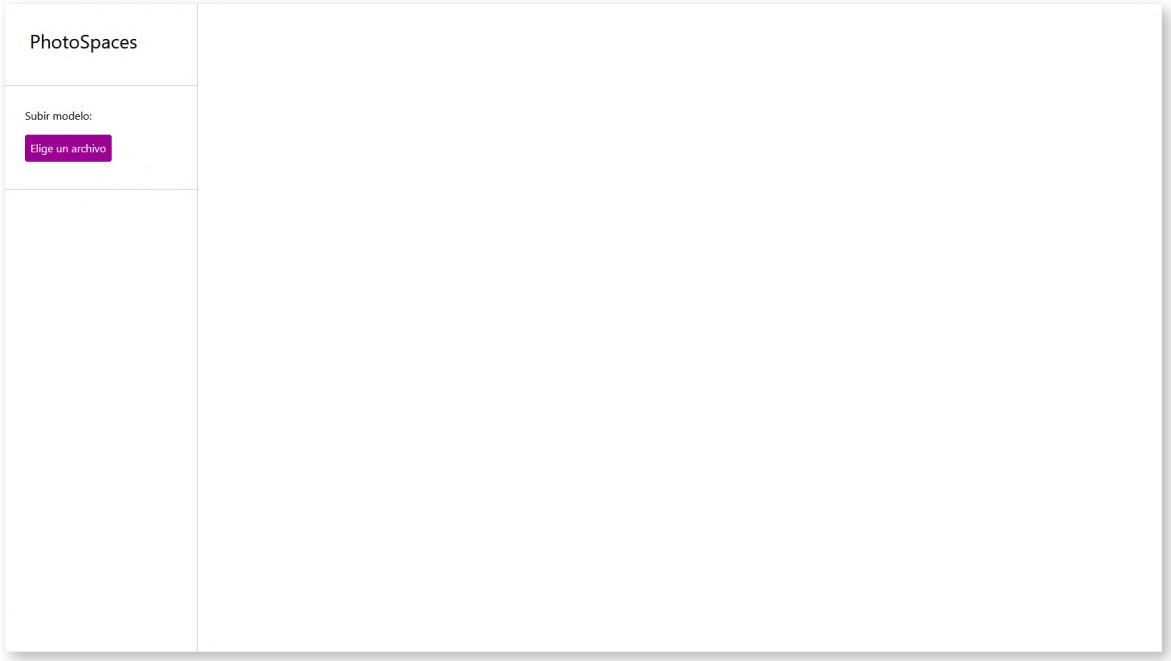

Figura 7.1: Página principal del cliente

Así, como podemos observar en la figura 7.1, haciendo uso de un par de elementos HTML y CSS para colocar los elementos en sus posiciones tenemos una página principal de la aplicación. Sin embargo, con un fichero HTML no podemos tener una aplicación que se sirva fácilmente a las peticiones HTTP que reciba. Por lo que pasamos a la implementación de el servidor de la aplicación con la ayuda de la librería *express.js*. Esta librería ha sido instalada en la aplicación haciendo uso de *npm*, la mayor ventaja de este gestor de librerías es que al generar una aplicación base tenemos un fichero JSON[25] (llamado *package.json*) donde vienen especificadas todas las propiedades de la aplicación y las librerías que usa, de esta forma únicamente con este fichero y *node.js* podremos instalar todas las librerías automáticamente sin tener que subirlas a un repositorio, reduciendo en gran medida el tamaño de la aplicación.

Con *express.js* ya instalado podemos generar el fichero *app.js* que se encargará de suplir los diferentes archivos y librerías al cliente, facilitando el acceso de la página principal a los diferentes *scripts* que usemos y a estos, las librerías que usen. Como podemos ver en la figura 3.2, la implementación de este servidor es muy simple, constando el fichero de tres partes diferenciadas:

1. En la primera parte del archivo se importa la librería de express.js y se genera una aplicación con la sentencia *express()*. A partir de aquí se almacena la ruta de la aplicación para su uso posterior y por último se obtiene en puerto donde se generará la conexión, si existe uno como variable de entorno se usará y sino se usará uno por defecto (8080).

2. En la segunda parte se describen las rutas que va a usar cada carpeta o módulo que deseemos que sirva el servidor haciendo uso del método *use()* de la instancia de aplicación de express.js. Por lo que en este servidor definimos: la carpeta *public*, donde se encuentran todos los archivos estáticos del servidor; y, para su fácil acceso, tanto la carpeta donde se encuentra la aplicación de three.js como una de sus subcarpetas.

3. Y por último, con el método *listen()*, comenzamos a dar servicio de la aplicación en el puerto que hayamos asignado, además de incluir un método en esta escucha, que en nuestro caso, es únicamente imprimir por consola el puerto en el que está conectada la aplicación.

Con la aplicación del cliente ya implementada, solo tenemos que indicar a node.js que fichero usará para ejecutar la aplicación, así con un simple comando en la consola como `node app.js` podemos comenzar la aplicación en la dirección localhost:8080 por defecto. Este comando se puede abreviar aún más haciendo uso del fichero *package.json* que comentamos antes, en el que podemos describir scripts y resumirlos en un único comando, en nuestro caso, el comando anterior será reducido a `npm start`, de esta forma si necesitamos más parámetros en el comando se quedará más simple.

La página principal de la aplicación será modificada y actualizada mediante JavaScript, por lo que toda la lógica de la aplicación cliente estará ubicada en el fichero *client.js*. La lógica del cliente tendrá dos responsabilidades: actualizar la página principal según el usuario interactúe con ella y procesar todo lo que conlleva el visor del modelo. Por lo que en una primera instancia solo tenemos que importar la librería de *three.js* según la ruta

indicada antes en el fichero *app.js* e indicarle el elemento HTML en el que mostrará los escenarios.

```
// 1. Asignar el elemento HTML que será el visor
// Y asignar un renderizador a ese elemento
const canvas = document.querySelector("#canvas");
const renderer = new THREE.WebGLRenderer({ canvas });

// 2. Crear escena principal
const scene = new THREE.Scene();

// Crear una cámara
const camera = new THREE.PerspectiveCamera(fov, aspect, near, far);

// 3. Añadir los controles de navegación a la escena
const controls = new PointerLockControls(camera, canvas);
scene.add(controls.getObject());

// 4. Añadir el modelo a la escena
const gltfLoader = new GLTFLoader();
gltfLoader.parse(file, './', (gltf) => {
  const root = gltf.scene;
  scene.add(root);
});

// 5. Y renderizar la escena
renderer.render(scene, camera);
```

Figura 7.2: Código fuente de la carga del modelo (simplificado)

En resumidas cuentas, en la figura 7.2 podemos ver cuáles serían las bases principales para cargar un modelo en el visor:

1. Asignar el elemento que contendrá el visor y crear con él el renderizador.
2. Crear la escena principal a la que se añadirá el modelo y la cámara que se usará.
3. Crear unos controles de navegación y añadirlos a la escena. Estos controles actuarán según los como movamos el ratón o pulsemos las teclas en el visor.
4. Añadir el modelo a la escena a partir del archivo que haya subido el usuario y haciendo uso del cargador de glTF que trae three.js.
5. Y por último renderizar la escena.

Si en este último paso no hiciéramos nada más, nuestro modelo quedaría estático, ya que solo lo renderizamos una vez, así que este método es llamado repetidamente al hacer uso del método `requestAnimationFrame()`. Con esto tendríamos la funcionalidad básica del visor finalizada, pudiendo cargar un modelo y navegarlo libremente.

7.1.3. API del servidor

Para el servidor vamos a diseñar su API con una herramienta de código libre llamada *Swagger Editor*[26], esta herramienta nos permite diseñar las rutas de una API de manera sencilla y muy visual, además posee funcionalidades de generación de código a partir de esta definición. Como podemos ver en la figura 7.3, la API del servidor será muy simple, consistiendo únicamente de tres rutas diferentes.

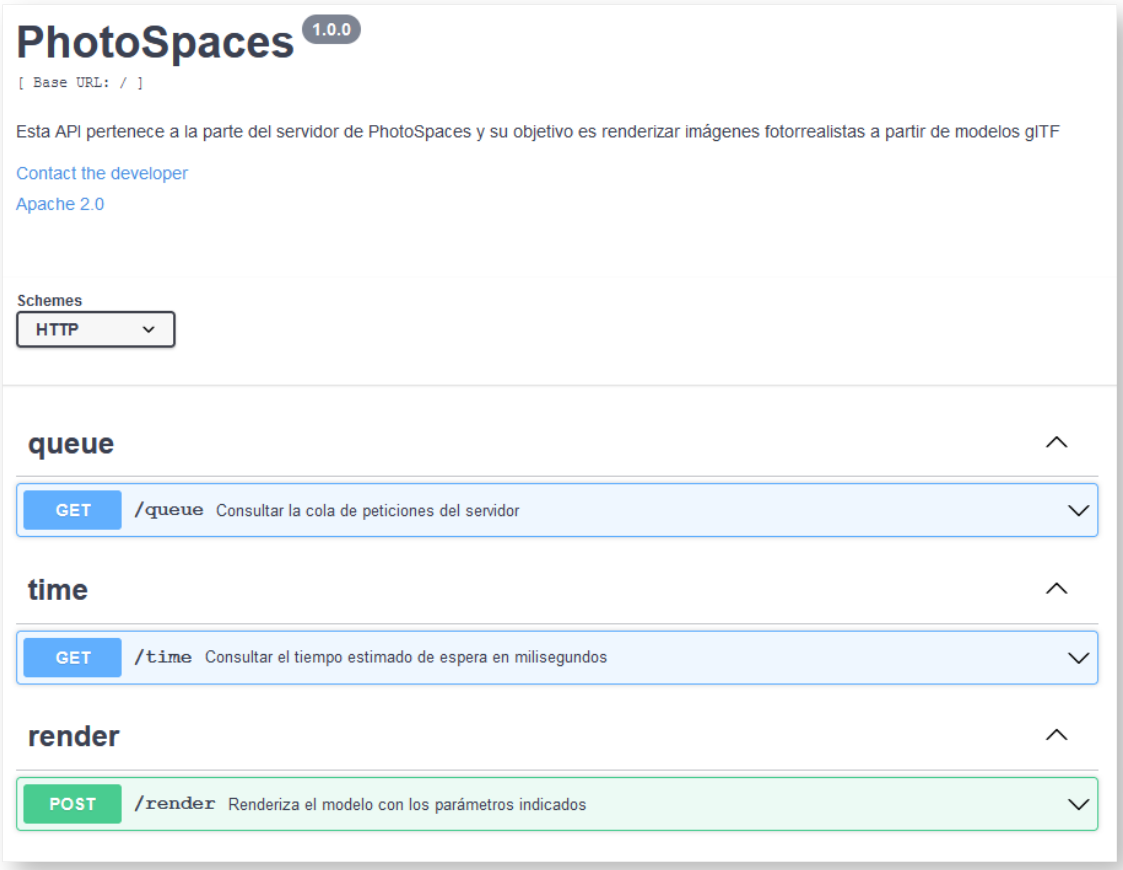

Figura 7.3: Definición de la API del servidor en Swagger

1. La ruta GET de *queue* nos devuelve el número de peticiones actuales que tiene el servidor en cola.

2. La ruta GET de *time* nos devuelve el tiempo estimado de espera en milisegundos para que el proceso actual finalice.

3. La ruta POST de *render* recibe el archivo del modelo y una cadena de texto en formato JSON con los parámetros deseados para el renderizado, en respuesta, el servidor nos devuelve la imagen procesada.

De esta manera, y haciendo uso de express.js como en la parte del cliente, podemos implementar la aplicación del servidor sin demasiadas complicaciones. La única diferencia con la aplicación del cliente es que se añaden las rutas antes mencionadas a la aplicación, también se añade una ruta adicional de control de errores y un procesamiento de datos para poder recibir todo tipo de datos en las peticiones HTTP. Por lo que el grueso de esta parte se encuentra en la lógica que implementan estas rutas, haciendo uso de la estructura de ficheros y directorios estudiada en el capítulo 5, podemos abstraer cada funcionalidad de la aplicación en tres partes distintas:

- **Ruta:** que se encarga de definir la ruta y que métodos de servicio se le asignará a cada una.
- **Servicio:** que procesará la información recibida y preparará la respuesta a partir de los datos obtenidos de la lógica.
- **Lógica:** que realiza la parte analítica y de procesamiento de las rutas que, una vez terminadas, le son devueltas al servicio correspondiente.

En esta primera iteración las rutas son definidas y existen, pero la lógica de las mismas no está implementada, esta lógica será desarrollada en la tercera iteración. Con estas rutas acabaríamos los objetivos principales para esta primera iteración, de la cual hemos obtenido una aplicación cliente con un visor funcional y una aplicación servidor con la API definida.

7.1.4. Retos

Dejando a un lado el proceso de aprendizaje de las diferentes tecnologías usadas como pueden ser three.js o Swagger, el reto principal fue servir los diferentes recursos de la aplicación cliente, ya que en un principio no se iba a usar la librería express.js para la definición de las rutas de recurso, pero después de un tiempo intentando implementarla sin ella, se consideró oportuno su uso, facilitando por completo el servicio de recursos para la aplicación.

7.2. Segunda Iteración

Tras dar los primeros pasos en la implementación de la aplicación, toca continuar con los conocimientos aprendidos que han sido obtenidos de la primera iteración. Ya conocemos como usar la librería three.js para importar un modelo y tenemos desarrollada las rutas de la API del servidor, por lo que ha llegado el momento de comunicar la aplicación del cliente con la aplicación del servidor. En esta iteración abordaremos como enviará el cliente el modelo y los parámetros necesarios al servidor, además, comenzaremos la creación del script en Python para el renderizado de Blender.

7.2.1. Objetivos

Entonces en esta iteración buscaremos principalmente tener un prototipo muy básico de la aplicación pero funcional, mejorando las funciones que ya existían previamente de manera paralela. Al final de esta iteración habremos conseguido subir un modelo desde nuestra máquina local, navegarlo a libertad en el visor, enviarlo al servidor y recibir de vuelta una imagen renderizada del modelo. De esta manera, vamos a ver los objetivos de manera más concreta:

1. Una aplicación cliente que sea capaz de exportar tanto la escena que hemos estado navegando en el visor como la posición de la cámara que será la visión que hemos tenido. Este modelo será posteriormente enviado en una petición HTTP al servidor, para más tarde descargar una imagen en la respuesta.

2. Una aplicación servidor que reciba la petición HTTP con el modelo a renderizar y los diferentes parámetros para esto, además de poder ejecutar un script que renderice automáticamente el modelo para devolverlo posteriormente en la respuesta HTTP.

Los objetivos son bastante claros y buscan tener un prototipo del que podamos aprender al final de la iteración y que pueda ser totalmente finalizado en la tercera, aunque en el caso de necesitar más iteraciones para el desarrollo podrían hacerse hasta que los requisitos de la aplicación sean satisfechos. Así, comencemos a analizar el proceso de desarrollo de la segunda iteración.

7.2.2. Aplicación del cliente. Exportar la escena

En la anterior iteración conseguimos tener un visor que cargara nuestro modelo y nos permitiera navegarlo a libertad usando el teclado y el ratón. Una vez decidida la perspectiva que tomará nuestra imagen queda el paso de exportar ese modelo e indicar de alguna manera, la posición en la que deseamos que esté nuestra cámara para obtener la misma posición que en el visor. Anteriormente, al importar el modelo a la escena de la aplicación, creábamos una cámara nueva para la escena, en este caso se va a añadir una modificación a eso, de forma que si el modelo importado traía una cámara por defecto, la escena usará esa cámara, facilitando el posicionamiento de la misma si ya teníamos una.

Como podemos ver en la figura 7.4 el proceso de exportación de la escena y de mandarla al servidor es el que sigue:

1. Para comenzar creamos los objetos de exportador, el objeto de *FormData*, el *Quaternion* y el objeto con los datos de la cámara.

2. Después, usamos el exportador de glTF para convertir la escena en un archivo que podamos mandar.

3. Este modelo ya exportado es introducido en el *FormData* junto con los datos de la cámara.

4. Luego creamos la petición HTTP al servidor y la ejecutamos.

5. Y por último recibimos en la respuesta de la petición la imagen renderizada que podremos descargar desde el navegador.

Como hemos mencionado antes, para el envío de información hacemos uso de un objeto *FormData*[27], esta es una interfaz con la que podemos construir pares de clave y valor que podemos enviar fácilmente en peticiones HTTP, de esta forma lo usamos en la exportación del modelo para enviar un paquete de datos con toda la información necesaria, en este caso, los datos de la cámara y el modelo glTF. Como hemos comentado con anterioridad, el formato glTF está descrito en JSON, por lo que para facilitar la manipulación del modelo lo convertimos en una cadena de texto que puede adjuntarse a una petición.

Posteriormente hemos creado una instancia de *quaternion* (cuaternión en español), este objeto se usa en three.js para representar las rotaciones de un objeto, en este caso, la cámara. No solo necesitamos conocer la posición de la cámara en el modelo y sus características como el campo de visión o el aspecto, la rotación nos indicará hacía que lugar y con que inclinación está ubicada la cámara. Three.js considera en sus coordenadas tridimensionales la coordenada Y como la dirección vertical y los cuaterniones son representados en grados sexagesimales; sin embargo, Blender considera la coordenada vertical como la Z y representa los cuaterniones en radianes. Por lo que para pasar la información de la cámara de three.js a Blender debemos multiplicar el cuaternión por la coordenada X y pasarlo a radianes, para que el modelo se cargue como debería en Blender.

De esta forma nuestro cliente ya es capaz de exportar la escena que hemos navegado, almacenando la posición exacta de la perspectiva que hemos elegido y creando una petición HTTP con el modelo que es enviada al servidor. En iteraciones posteriores, esta petición contendrá los diferentes parámetros elegidos por el usuario para el proceso de renderizado y se dividirá la petición en dos diferentes además de parametrizar la dirección de la misma por si es usada en un servidor distinto. Por lo que en esta iteración el renderizado será el que viene por defecto y el usuario no podrá elegir los parámetros del mismo. Ahora solo queda que nuestro servidor reciba la petición HTTP y procese nuestro modelo.

```
const sendModel = async (cam) => {
  // Creamos el exportador, el FormData donde irá la información,
  // el quaternion y los datos de la cámara
  const exporter = new GLTFExporter();
  const formData = new FormData();
  const quaternion = new THREE.Quaternion();
  quaternion.setFromAxisAngle(new THREE.Vector3(1, 0, 0), Math.PI / 2);
  const newQ = quaternion.multiply(cam.quaternion);

  const camData = {
    lens: cam.getFocalLength(),
    clip_start: cam.near,
    clip_end: cam.far,
    location: { x: cam.position.x, y: -cam.position.z, z: cam.position.y },
    qua: newQ,
    motor: eeveeEngine.checked ? "BLENDER_EEVEE" : "CYCLES",
    gtao: gtao.checked,
    bloom: bloom.checked,
    ssr: ssr.checked,
  };

  // Usamos el exportador para convertir la escena en un modelo glTF
  exporter.parse(scene, async (gltf) => {
    // Añadimos el modelo al FormData
    formData.append(
      "model",
      new Blob([JSON.stringify(gltf, null, 2)], { type: "text/plain" })
    );

    // Añadimos los datos de la cámara
    formData.append("data", JSON.stringify(camData));

    btnRender.style.display = "none";
    btnLoading.style.display = "block";
    // Hacemos la petición
    fetch(
      "http://localhost:3030/render",
      {
        method: "POST",
        body: formData,
      }
    )
      .then((res) => {
        // Procesamos la imagen de la respuesta
        btnLoading.style.display = "none";
        btnRender.style.display = "block";
        downloadImage(res.body);
      })
      .catch((e) => console.log(e));

    // Controlar el tiempo y la cola
    updateTimeAndQueue();
  });
```

Figura 7.4: Exportar y enviar el modelo glTF

7.2.3. Recepción de la petición HTTP y renderizado

Una vez enviado el modelo y la información desde la aplicación del cliente, debemos recibirlo íntegramente en el servidor y procesar esa información. Con la ayuda de una librería llamada *fileupload* de npm, podemos aceptar peticiones HTTP con objetos FormData como cuerpo. A partir de aquí la petición es derivada a la ruta */render* la cuál se encarga de seleccionar el método necesario de la capa de servicio que, como podemos ver en la figura 7.5, se encarga generar la respuesta de la petición. En primera instancia, el servicio pide a la lógica del renderizado procesar el modelo y una vez hecho, lo devuelve en una respuesta satisfactoria o lanza un error que será procesado por la ruta de errores en *app.js*.

```
import { Worker } from "worker_threads";
import { saveTempFile, deleteTempFiles } from "../logic/fileLogic.js";
import { setTimeEstimation } from "../logic/timeLogic.js";

export async function upload(req, res, next) {
  try {
    // Cogemos el modelo y lo guardamos en el servidor de forma temporal
    const model = req.files.model;
    saveTempFile(model);

    // Creamos el Worker y le pasamos los datos que usará
    const worker = new Worker("./src/logic/renderLogic.js", {
      workerData: {
        data: req.body.data,
        fileName: model.md5,
      },
    });

    // Actualizamos con la información del worker el tiempo estimado
    // y cuando finalice respondemos a la petición con la imagen
    worker.on("message", (message) => {
      if (message.includes("file")) {
        setTimeEstimation(JSON.parse(message));
      } else {
        res.status(201).sendFile(`${message}.png`, options);
  } catch (e) {
    next(e);
  }
}
```

Figura 7.5: Respuesta de la petición

Paralelamente, la parte lógica de la ruta de renderizado se encarga de ejecutar el script de Blender con los parámetros de la cámara, almacenar temporalmente el modelo y la imagen procesada para que Blender lo pueda usar y se pueda devolver en la respuesta de

manera sencilla. Para que esta solicitud no bloquee el servidor creamos un *Worker* que se encarga de ejecutar la petición que le hagamos en un nuevo hilo, en este caso le pedimos que ejecute el script de *renderLogic.js*. Para ejecutar el script de Blender, en el fichero *renderLogic.js* se utiliza el modulo de node.js llamado *Child process*, concretamente una parte de el mismo llamada *spawn*, de esta forma podemos invocar una consola y ejecutar el comando correspondiente, que en este caso sería pedirle a Blender ejecutar un script con unos parámetros concretos. Por lo que el comando usado quedaría tal que así:

```
blender -b -P [localizacion script] -- ${filename} ${data}
```

A parte de el nombre del fichero, se pueden concatenar al comando los diferentes parámetros que se vayan a usar en el script, como la posición de la cámara o su orientación. Este script funciona a través de la API de Blender[28], la cuál está escrita en Python y posee una librería para el mismo llamada *bpy*, los módulos que usaremos en el script serán normalmente las siguientes:

- `bpy.data` - Este modulo nos permite acceder a todo tipo de datos de Blender pudiendo crear objetos y modificar sus propiedades. Lo usamos, por ejemplo, para crear la cámara que se usa en el modelo.

- `bpy.ops` - Este modulo nos permite ejecutar diferentes métodos de Blender como la importación de modelos o el renderizado de ellos.

- `bpy.context` - Este modulo nos permite acceder a los datos de la escena en la que estamos trabajando actualmente y editarla, pudiendo añadir o eliminar objetos de la misma.

Al ejecutar el comando como hemos comentado antes, Blender se ejecutará sin la interfaz gráfica de usuario y comenzará a implementar las modificaciones que están descritas en el script, tras esto, se almacenará la imagen renderizada temporalmente en un directorio del que podrá ser enviada de vuelta en la petición HTTP, que podrá ser descargada desde el navegador del cliente.

7.2.4. Retos

En esta iteración se han presentado más retos que en la anterior y no ha sido tan sencillo hacer la implementación de la manera definitiva que se ha descrito anteriormente. El primero de los obstáculos que se presentó fue pasar los cuaterniones de three.js a Blender, ya que no había ninguna documentación de que estándares usaba cada uno o como podrían convertirse los cuaterniones; después de un tiempo de búsqueda, encontramos una página web que describía el mismo problema y como se puede solucionar[29], aunque la solución fue parcial, ya que en vez de multiplicar el cuaternión en la coordenada Y como está en la respuesta, lo conseguimos con la X. El otro obstáculo fue que el servidor aceptara FormData sin modificarlos en el proceso, al final encontramos la librería que mencionamos anteriormente y se solucionó.

Y con esto ya tendriamos una versión mínimamente funcional de la aplicación web, por lo que lo único que nos queda en la iteración final es satisfacer unos cuantos requisitos algo más secundarios y finalizar la aplicación para tener un producto completo y funcional.

7.3. Tercera Iteración

En esta última iteración del desarrollo terminaremos la aplicación, implementando los requisitos que nos quedan, a partir de aquí el proceso es más simple ya que tenemos la base de la aplicación y solo tendremos que ir añadiéndole mejoras. A parte de esto, construiremos la aplicación en *Docker* para facilitar su despliegue en servicios de nube de terceros y para aquellos que quieran hacer la instalación de la aplicación en su propio sistema.

7.3.1. Objetivos

El objetivo principal como ya hemos mencionado antes será terminar de implementar todos requisitos que necesitaba la aplicación y paralelamente, poder desplegar tanto el cliente como el servidor en contenedores de Docker. Esta iteración busca mejorar la calidad de uso de la aplicación, con funcionalidades que mejoren las ya existentes como podría ser una mejor retroalimentación para el usuario, más opciones o un sistema más estable. Por lo que si lo dividimos en partes más tangibles tendriamos:

1. Una aplicación cliente que nos permita añadir y posicionar objetos de luces a la escena, en la que podamos cambiar el fondo de visualización del modelo, que devuelva información sobre los tiempos de espera estimados y que permita seleccionar los parámetros de renderización del modelo.

2. Una aplicación servidor que tenga la lógica de su API implementada por completo, que lleve una cola de peticiones y permita editar más parámetros de renderizado a la hora de ejecutar el script de Blender.

3. Un despliegue sencillo de toda la aplicación al completo, haciendo uso de *Docker* y *Docker compose*.

7.3.2. Luces y parámetros de renderizado

Para el comienzo de esta última iteración, comenzaremos de nuevo con la parte del cliente, en este caso vamos a implementar la funcionalidad de que podamos añadir y mover libremente objetos de luces, de esta forma, si nuestro modelo no contiene luces o queremos añadir alguna más, podremos hacerlo directamente desde el navegador web. La idea principal es que sea muy intuitivo, por ello, la manera de seleccionar y añadir una luz será reemplazando el menú habitual que aparece al pulsar el botón derecho del ratón por un menú personalizado en el que podamos seleccionar que luces deseamos añadir.

Además de añadir las luces, haciendo uso de un modulo de three.js, permitiremos al usuario cambiar sus propiedades, para ubicar la luz en la zona que más le satisfaga. Para poder implementar un menú personalizado, debemos bloquear el evento que sucedería habitualmente al pulsar el botón derecho del ratón y sustituirlo por un elemento HTML que elijamos. En nuestro caso, este menú solo aparecerá si se pulsa mientras estamos dentro del visor, para no eliminar por completo la funcionalidad común del navegador.

A partir de este menú personalizado solo tenemos que indicar que ocurre al pulsar sobre cada elemento del menú, que en este caso diferenciará entre una *luz de punto* o una *luz direccional*; las luces de punto se podrían describir como una bombilla común, son una esfera que emite luz por igual en todas las direcciones; y las luces direccionales, tienen un área que es dirigida a una parte de la escena, iluminando solo esta zona. En la figura 7.6, podemos ver como en el modelo tenemos el menú desplegado y podemos elegir entre las dos luces que hemos mencionado, en la parte superior derecha encontramos los controles para modificar las propiedades de estos objetos y abajo a la izquierda, podemos ver la lista de las luces que tenemos y eliminarlas si lo deseamos. Estas luces son representadas por unas líneas que ayudan a comprender donde está la luz y en el caso de la direccional, a donde apunta, estas líneas son solo de apoyo y no serán renderizadas en el producto final.

Figura 7.6: Página principal del cliente con el menú personalizado y objetos de luz

También se ha implementado la opción de cambiar el fondo donde se encuentra el modelo. Este fondo se puede seleccionar desde un menú desplegable y el fondo es cambiado en la escena, estos fondos son archivos prerenderizados tridimensionales que están almacenados como archivos estáticos en el cliente. Estos fondos no son renderizados posteriormente en el producto final, sino que son usados para facilitar el visionado del modelo o incluso, para crear una mejor representación de donde se encontraría ese modelo en la realidad.

Con esto, solo nos quedan dos cambios importantes en la aplicación del cliente para tenerla finalizada, estos son: elegir los diferentes parámetros para el renderizado y la retroalimentación de lo que tardará el proceso en completarse. Empecemos con los parámetros del renderizado; como ya hemos estudiado en el capítulo 6, el usuario tendrá la opción de seleccionar que motor de renderizado usará y más allá, si hace uso del motor Eevee, podrá seleccionar los diferentes parámetros que quiere que se le aplique al procesamiento. Entonces, si el usuario selecciona Eevee como motor de búsqueda, aparecerán los diferen-

tes parámetros que puede seleccionar como vemos en la figura 7.7: la oclusión ambiental, el resplandor y la luz global. Y si selecciona el motor de Cycles, los parámetros usados serán los más óptimos planteados en el estudio del fotorrealismo y solo podrá seleccionar la visión de campo (FOV).

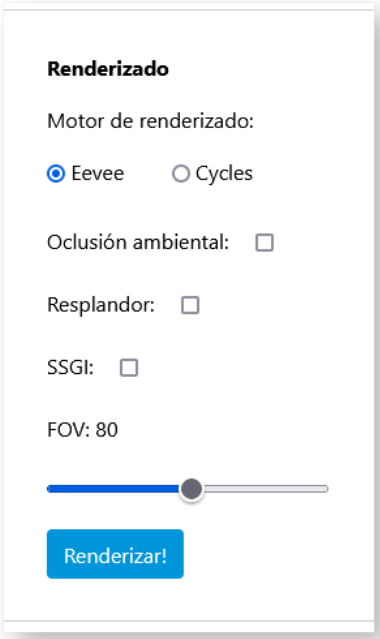

Figura 7.7: Parámetros de renderizado en la aplicación del cliente

Y por último, cuando el usuario ya tenga la perspectiva de la cámara que desea y los parámetros de renderizado seleccionados, podrá pulsar el botón de renderizar. En ese momento aparecerá una ventana, como podemos ver en la figura 7.8, que le indicará cuantos procesos hay en el momento en cola y el tiempo estimado del proceso que está en ejecución; cuando su proceso termine, se cerrará la ventana de información y podrá descargarse la imagen desde el botón de descarga en el panel izquierdo. Estos datos de cola y tiempo son tomados a partir de la API del servidor, por lo que en el momento de hacer la petición de procesamiento, el cliente continua haciendo peticiones HTTP para obtener los datos sobre el estado del procesamiento, como podemos observar en la figura 7.9.

Como podemos ver en la figura 7.8, la información de renderizado nos muestra que no hay peticiones en la cola actualmente, esto significa que no hay ninguna petición esperando ser procesada actualmente y que es nuestra petición la que está siendo renderizada en el servidor. El tiempo estimado indicado hace referencia al procesamiento que está haciendo actualmente el servidor y no al acumulativo de estas peticiones, ya que para hacer esta estimación necesitamos empezar a renderizar el modelo, como veremos más adelante.

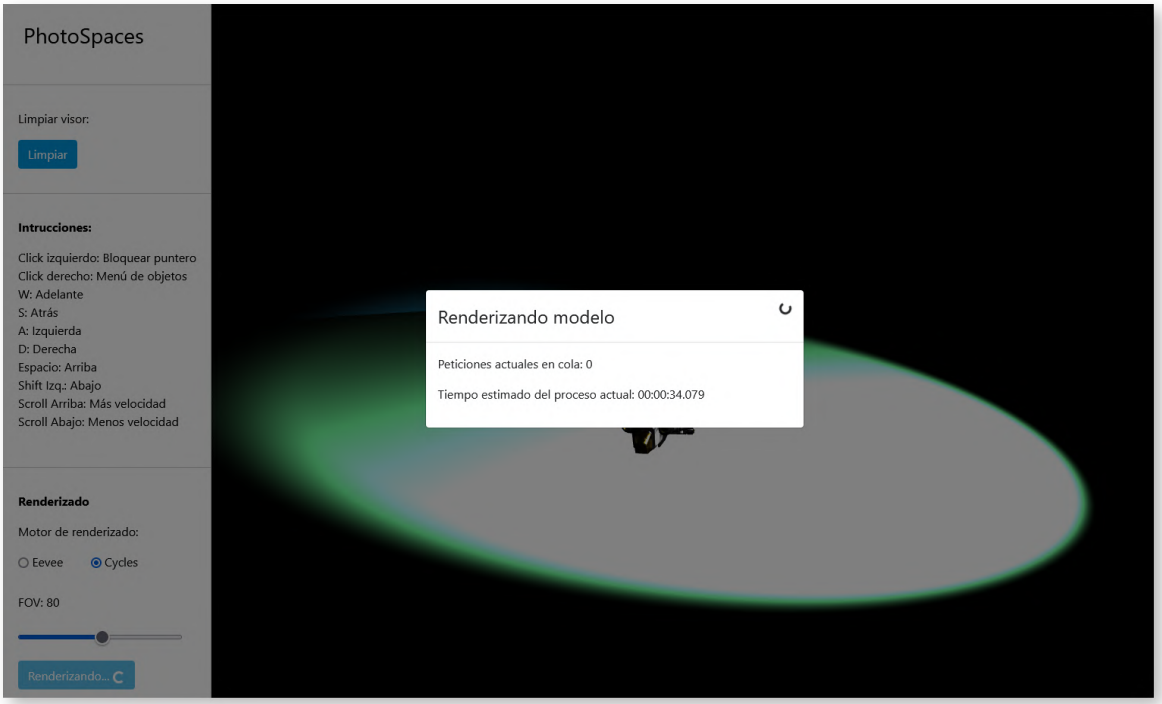

Figura 7.8: Información del renderizado

```
const updateTimeAndQueue = () => {
  const actualQueue = document.getElementById("actualQueue");
  const actualTime = document.getElementById("actualTime");

  $("#renderInfoModal").modal("show");

  // Ejecutamos el intervalo cada 2 segundos
  intervalUpdate = setInterval(() => {
    // Obtenemos la información de la cola y la actualizamos
    fetch("http://localhost:3030/queue", { method: "GET" })
      .then((res) => res.json())
      .then((queue) => (actualQueue.innerHTML = queue));

    // Obtenemos la información del tiempo y la actualizamos
    fetch("http://localhost:3030/time", { method: "GET" })
      .then((res) => res.json())
      .then(
        (time) =>
          (actualTime.innerHTML = new Date(time.remaining)
            .toISOString()
            .slice(11, -1))
      );
  }, 2000);
};
```

Figura 7.9: Actualización de la información del proceso

Con estas últimas funcionalidades la aplicación del cliente ya estaría completa y sería totalmente funcional. Pudiendo pasar así a describir el desarrollo en paralelo de la última iteración de la aplicación del servidor.

7.3.3. Aplicación del servidor final

En esta tercera iteración debemos terminar la funcionalidad de la API del servidor para que todas las rutas que definimos en la primera iteración sean totalmente funcionales. De las tres rutas de la API tenemos una ya implementada que es la ruta del proceso de renderizado, por lo que nos queda las otras dos, la ruta que nos devuelva cuantos procesos están en espera en la cola de peticiones (*queue*) y la ruta que nos devuelve el tiempo estimado que le resta al proceso actual (*time*). Empecemos con la definición de una cola de peticiones.

Cola de peticiones

Nuestra cola de peticiones busca poder administrar correctamente las peticiones que se le harán a nuestro servidor de manera que podamos definir cuáles y cuantas peticiones podrán ser procesadas simultáneamente y cuáles tendrán que esperar en la cola. Por lo que la implementación de esta funcionalidad tiene dos puntos claves:

- Qué peticiones podrán hacerse simultáneamente a las demás peticiones.
- Qué peticiones deberán esperar a la cola para procesarse de manera secuencial.

Para esta funcionalidad usaremos una librería de node.js que se llama *express-queue* y es una implementación de una cola básica que nos permite indicar cuantas peticiones podremos tener activas simultáneamente y cuantas podremos tener en cola antes de rechazar una petición. Aunque *express.js* por defecto ya controla la entrada de peticiones simultáneas, esta librería nos permitirá parametrizar el número límite de ellas y tener un registro a tiempo real de las mismas. Esta cola se crea en *app.js* indicando el número máximo de peticiones concurrentes y el número máximo de peticiones en la cola; a partir de aquí, exportamos esta variable para poder acceder a ella desde otras partes del código y hacemos uso de esta cola únicamente en la ruta de renderizado.

Con esta interfaz ya creada y exportada, solo nos queda crear un servicio para la ruta de cola en el que devuelva el número de peticiones actuales en cola en la respuesta de la petición, para esto solo tenemos que acceder a la variable exportada anteriormente y consultar la longitud de la misma con la función *getLength()*.

Estimación del tiempo de procesado

Para el tiempo de procesado el enfoque que tomamos es directo, Blender en el proceso de renderizado nos notifica sobre la información del renderizado como podría ser tiempo renderizando, cuadrantes renderizados o el tiempo estimado restante. Sin embargo, esta

información de renderizado no es proporcionada de ninguna forma en la API de Blender, por lo que tenemos que usar otro acercamiento para tomar esa información.

Al ejecutar Blender en segundo plano, como hacemos nosotros a la hora de renderizar los modelos, este va imprimiendo las diferentes estadísticas en la consola. Y como hacemos uso de la funcionalidad de *spawn* para ejecutar el comando, esta interfaz nos permite escuchar las diferentes actualizaciones de la consola, por lo que lo único que tenemos que hacer en la lógica de renderizado es, tomar las diferentes actualizaciones de la consola que tengan información sobre el tiempo restante y modificar una variable a la que tiene acceso la ruta de servicio de *time*, de esta forma tendremos datos en tiempo real de el tiempo restante del procesamiento.

Aunque esta funcionalidad de estimación de tiempo solo la calcula Blender para animaciones o renderizados que hagan uso del motor Cycles, por lo que solo podremos dar información del tiempo restante al hacer uso de este motor. Pero esto no es mucho problema, ya que Eevee es capaz de renderizar la mayoría de modelos en menos de cinco segundos, por lo que cualquier petición que haga uso de este motor durará muy poco tiempo en la cola como para que sea útil saber el tiempo estimado del mismo.

Con estas dos últimas funcionalidades ya tenemos la aplicación del servidor completada en su totalidad, estas dos últimas rutas de la API son las que son usadas por la aplicación cliente en sus funcionalidades de información sobre el proceso de renderizado. Ahora, veamos como podemos facilitar toda la instalación y despliegue de la aplicación con Docker.

7.3.4. Aplicación en Docker

Como ya hemos mencionado antes, Docker nos permite aislar nuestros programas en su propio entorno, de esta forma conseguimos que sean compatibles en cualquier sistema que pueda tener Docker y facilitamos el despliegue de la aplicación a un simple comando para arrancar los contenedores. Más allá de solo usar Docker, también usaremos Docker Compose, esta herramienta nos permite ejecutar varios contenedores simultáneamente, así podremos tener nuestra aplicación completa en una máquina usando únicamente un comando para comenzar los contenedores.

Por la parte del cliente es simple tener una aplicación en node.js en una máquina cualquiera, pero este proceso disminuye aun más la cantidad de pasos necesarios para ejecutar esta aplicación. Pero la parte que realmente tiene más importancia para crear un contenedor de la misma es el servidor, ya que hacemos uso de Blender, este programa añade una dificultad más al despliegue común, puesto que debemos instalar la versión del programa adecuada e instalarla en nuestra máquina. Así, introduciendo en un mismo contenedor tanto la aplicación del servidor como Blender, reducimos todos los pasos de un despliegue satisfactorio de nuestra aplicación a tener Docker y ejecutar el comando necesario. Ahora veamos que pasos son necesarios para *dockerizar* esta aplicación.

Dockerfile

Docker puede construir imágenes automáticamente leyendo las instrucciones de un archivo llamado *Dockerfile*. Un Dockerfile es un archivo que contiene todos los comandos que un usuario podría usar en la línea de comandos para generar una imagen. Con este archivo y ejecutando el comando `docker build` podemos generar una imagen automáticamente con los comandos del archivo. Hay muchos comandos que pueden usarse en un Dockerfile, pero los más importantes son:

FROM Esta instrucción inicializa una nueva fase de construcción y define una imagen base para las instrucciones siguientes. Esta imagen puede ser de un repositorio de Docker o local.

RUN Esta instrucción ejecuta cualquier comando en la imagen interna que estamos construyendo.

COPY Esta instrucción copia archivos y ficheros de nuestro sistema y los añade al contenedor.

Existen más instrucciones que podemos usar para construir una imagen, todas ellas vienen definidas en la documentación de Docker[30]. A raíz de estos comandos podemos desarrollar dos ficheros distintos que contengan las instrucciones para construir una imagen del cliente y una del servidor. El Dockerfile del cliente lo podemos ver en la figura 3.4 y es más simple que la del servidor, explicando la construcción de la imagen brevemente tendríamos: comenzamos con una imagen que tiene node.js ya instalado, definimos el directorio donde estará la aplicación, copiamos nuestro *package.json* de la máquina local al contenedor y ejecutamos la instalación de las librerías, copiamos el resto de archivos del cliente, y por último abrimos el puerto 8080 y ejecutamos el comando de arranque.

El Dockerfile del servidor, a parte de tener las mismas instrucciones que el del cliente tenemos que instalar Blender. En esta imagen comenzamos desde una imagen base de Ubuntu, a partir de aquí actualizamos las librerías de Ubuntu e instalamos algunas herramientas para el futuro, tras ello nos descargamos Blender y lo descomprimimos en el contenedor y por último también tenemos que instalar node.js ya que esta imagen no comenzaba con él. Esta descarga de Blender, se hace a través de un enlace a *Dropbox*. Aunque sea algo más complicado que el del cliente, sigue valiendo mucho la pena ya que se reduce la complejidad total de la aplicación. Con las dos imágenes de Docker ya creadas solo nos queda ejecutarlas a la vez.

Docker Compose

Esta herramienta nos permite definir y ejecutar aplicaciones de múltiple contenedores. A partir de un fichero en formato *YAML* podemos configurar todos los servicios de nuestras aplicaciones y con un simple comando, crear y ejecutar todos los servicios simultáneamente. Los archivos *docker-compose* pueden llegar a ser complejos en sistemas grandes, pero como nosotros estamos construyendo una aplicación que consiste únicamente de dos contenedores, el archivo es mucho más simple.

```yaml
version: "3.8"
services:
  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile
    container_name: frontend
    ports:
      - 8080:8080

  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile
    container_name: backend
    ports:
      - 3030:3030
```

Figura 7.10: Fichero *docker-compose.yml*

Como podemos observar en la figura 7.10, con unas pocas líneas de código podemos orquestar toda nuestra aplicación al completo. Comenzando el fichero con la versión de Docker compose que será usada, a continuación listamos los servicios de los que va a constar la aplicación y las propiedades de estos. En nuestro caso, tenemos dos servicios distintos: *frontend*, que es la aplicación del cliente; y *backend*, que es la aplicación del servidor. Dentro de ellas definimos que serán construidas con la instrucción *build* y dentro de ella indicamos el contexto de la aplicación y el Dockerfile que usará para ello, por último solo tenemos que indicar el puerto que va a usar cada aplicación y a cuál queremos que equivalga en nuestra aplicación.

Con este archivo ya definido solo tenemos que ejecutar el comando `docker-compose up` para crear y lanzar nuestra aplicación al completo sin preocuparnos por las librerías o que funcione en nuestro sistema en concreto. Haciendo uso de las ventajas que nos proporciona Docker, podemos subir estos archivos a una página como *Heroku*[31] que es una plataforma como servicio (PaaS) que nos permite desplegar nuestra aplicación para hacerla pública sin tener que configurar nuestros servidores.

El único problema de esta implementación es que no podemos dar el servicio común para la selección del motor de renderizado Eevee. Esto es debido a que este motor esta hecho para ser usado en tiempo real, por lo que para el renderizado de imágenes, Blender nos pide tener un monitor conectado aunque no hagamos uso del mismo porque hacemos el procesamiento en segundo plano y como el entorno simulado del contenedor no posee un monitor no podemos hacer uso de este motor. Aunque intentemos simular la existencia de un monitor virtual a través de Python, las texturas del modelo se procesan en las posiciones incorrectas, por lo que en esta versión de la aplicación solo se podrá hacer uso del motor Cycles.

7.3.5. Retos

Los retos principales en esta iteración se han concentrado en la parte del servidor, esto se debe a dos factores principales:

- En un principio, para la ejecución de el comando de Blender se hacía uso de un único hilo en el servidor, por lo que en la espera de este procesamiento, el servidor se quedaba bloqueado a cualquier otra petición. Pero desde la versión de Node.js 11.7.0, está implementado el uso de los antes mencionados *Workers*, que permiten ejecutar procesos en más de un hilo.

- En la estimación del tiempo, Blender no devuelve a través de su API ninguna de esta información de procesado, por lo que hubo que pensar en otra forma de obtenerla, en este caso, a través de la salida de consola del programa. Por otra parte, como en el procesamiento del modelo creábamos un nuevo hilo para no interrumpir el servidor, este hilo no compartía las mismas variables exportadas que el proceso padre, por lo que se tuvo que replicar esta información al hilo padre para que este pudiera actualizar el objeto de tiempo.

Y como último reto sería el uso del motor de renderizado Eevee en la aplicación montada en Docker, ya que se intentó crear un monitor virtual como se comentó antes para permitir que Blender renderizara los diferentes modelos, pero no está implementado en su funcionalidad que consiga esto sin la presencia de un monitor real, por lo que en última instancia no se pudo hacer.

De esta forma, con esta última iteración habríamos completado todo el desarrollo de la aplicación que hemos estudiado y diseñado en los capítulos anteriores. El uso de la metodología iterativa nos ha permitido implementar la aplicación de una manera sencilla, puesto que hemos ido añadiendo funcionalidades a ella poco a poco de manera que en cada fase del desarrollo estaban claros los objetivos de la misma, pudiendo afrontarlos directamente y de manera modular.

CAPÍTULO 8

Conclusiones

En este capítulo se va a desarrollar las diferentes conclusiones obtenidas del estudio y realización de este proyecto. Esta conclusión tendrá como objetivo evaluar este desarrollo según la satisfacción de los requisitos descritos en el capítulo 4 y las partes más fundamentales de la aplicación desarrollada. Todo esto se realizará de una manera resumida, sin explicar o justificar las diferentes decisiones que se tomaron en el desarrollo, pues, estas están expresadas en los capítulos anteriores.

Además, se describirán las posibles líneas futuras que surgen de este proyecto, buscando explicar la posible evolución del mismo en un futuro, implementando mejoras o nuevas funcionalidades. Estas líneas futuras pretenden cubrir factores no estudiados en este proyecto.

8.1. Líneas futuras

El objetivo principal de la aplicación era derivar la carga computacional del proceso de renderizado a un servidor externo que aliviara este proceso en la máquina del usuario. Sin embargo, actualmente la aplicación está preparada únicamente para recibir todas las peticiones HTTP en un único servidor externo, sin distribución de la carga y solo pudiendo responder a un limitado número de peticiones simultáneamente. Por lo que una mejora directa sería implementar una distribución de carga entre diferentes servidores, esta mejora sería posible ya que la tecnología REST está pensada para esto y una de sus características es que el usuario no pueda saber a que servidor está conectado en cualquier momento.

La versión final de la aplicación desarrollada en este proyecto, nos permite cargar un modelo en la página web y posteriormente navegarlo e incluso, añadir objetos de luz al modelo. Una mejora directa de esto sería la posibilidad de modificar los objetos que ya existen en el modelo, creando así, un editor completo. Esta característica nos permitiría eliminar por completo la necesidad de otras aplicaciones para la creación y modificación de modelos. La viabilidad de esta mejora es muy viable, ya que por defecto *three.js* posee multitud de herramientas para la modificación y creación de nuevos elementos, de las

cuáles, solo hemos visto unas pocas en este trabajo.

La aplicación actual hace uso de modelos en formato *glTF*, como ya hemos descrito en capítulos anteriores. Aún siendo un formato de lo más útil y flexible, solo permitir archivos en este formato limita la aplicación. Una posible mejora sería la adición de poder procesar más formatos de modelos 3D, como podrían ser *OBJ*, *SVG*, *3DM* o *blend*. De nuevo, *three.js* proporciona herramientas para cargar la mayoría de estos formatos, y los que no pudieran ser cargados por defecto, podrían ser convertidos a otros que si estuvieran permitidos.

Actualmente, y por el diseño del proyecto, la aplicación no almacena ningún tipo de datos de forma persistente. Una posible nueva funcionalidad, sería permitir a los diferentes usuarios registrarse e iniciar sesión en la aplicación, ya sea a través de la propia base de datos de los usuarios o haciendo uso de *OAuth*[32]. Con este almacenamiento y continuación de sesiones, se podrían almacenar los diferentes modelos de los usuarios, los resultados de las renderizaciones de estos modelos y sus parámetros determinados o más usados.

El renderizado de modelos es realizado haciendo uso del programa de código abierto Blender, como ya hemos estudiado con anterioridad. Se podría ofertar el uso de más programas de renderizado, de manera que el usuario pudiese elegir que programa usar, existiendo diferentes parámetros para cada uno y diferentes funcionalidades que serían explicadas en la documentación de la página. E incluso, sin modificar mucho la aplicación ya existente, se podrían instalar más motores de renderizado para Blender a través de los *add-ons*.

8.2. Conclusiones

El requisito principal de la aplicación era la generación de una imagen fotorrealista a partir de un modelo 3D. Esta funcionalidad fue dividida en diferentes requisitos que alcanzaran este objetivo al ser todos implementados correctamente. La aplicación está dividida en dos partes, el cliente y el servidor, por lo que el objetivo principal pudo ser dividido aún más. De esta manera, desarrollando paralelamente tanto una aplicación cliente para la navegación del modelo, como una aplicación servidor para el procesamiento de este, hemos obtenido una aplicación que renderiza correcta y efectivamente un modelo 3D. Además de la funcionalidad principal, hemos conseguido implementar diferentes mejoras que apoyan a la calidad de usuario al completo, como podría ser la interfaz gráfica sencilla, la retroalimentación del proceso de renderizado o la implementación de la aplicación en Docker para facilitar su instalación y despliegue.

Como ya hemos mencionado, la aplicación ha sido implementada siguiendo la estructura cliente-servidor, la cuál separa el sistema en dos módulos distintos que se encargan de funcionalidades distintas, esto nos ha ayudado a derivar por completo la carga de trabajo que es el procesamiento de imágenes. Y el servidor ha sido implementado siguiendo una arquitectura REST, lo que nos ha permitido beneficiarnos de ventajas como que no necesitamos almacenar estados entre las sesiones de los usuarios, facilitando la implementación.

Asimismo, la decisión de hacer uso de la librería *express.js* para la implementación de la API fue una decisión correcta, ya que el desarrollo de las diferentes rutas ha sido una tarea simple (en lo referido a la existencia de la ruta en la API, no la lógica de ellas) y la cola de peticiones también es muy versátil.

Por la parte del cliente también hemos hecho uso de express.js y *node.js*, que en este caso nos ha sido muy útil para poder servir los archivos estáticos y el código JavaScript. El objetivo de obtener una interfaz de usuario sencilla e intuitiva ha sido alcanzado y solo necesitamos unas cuantas acciones por parte del usuario, que puede tener conocimientos de la materia o no, para conseguir renderizar un modelo completamente. Más allá, *three.js* nos ha brindado multitud de posibilidades, siendo una librería mucho más flexible y capaz de lo que estudiamos en un principio; pudiendo trabajar con todo tipo de modelos, objetos, movimientos y más.

Los principales retos afrontados han sido por falta de documentación o en algún caso, porque la arquitectura planteada no estaba pensada para esa funcionalidad. Pero en última instancia, todos esos obstáculos han sido superados. En resumen, el desarrollo de este proyecto ha sido realmente un éxito, habiendo cumplido todos los requisitos planteados en el estudio de la aplicación e incluso mejorándolos al final como calidad del usuario. Todo el estudio del fotorrealismo ha sido de gran ayuda a la hora de conseguir obtener con más facilidad renderizados que se asemejen lo máximo a la realidad, y todo el diseño a priori del desarrollo ha facilitado mucho la implementación del mismo, así, como la elección de una metodología iterativa.

Bibliografía

1. V. Paradigm, *Agile Development: Iterative and Incremental*, www.visual-paradigm.com, 2015, (`https://www.visual-paradigm.com/scrum/agile-development-iterative-and-incremental/`).
2. *three.js editor*, threejs.org, (2021; `https://threejs.org/editor/`).
3. *Online 3D Viewer*, 3dviewer.net, (2021; `https://3dviewer.net/`).
4. *Vectary – The Easiest Online 3D Design and 3D Modeling Software*, www.vectary.com, (`https://www.vectary.com/`).
5. *Render Farm - CPU & GPU Online Cloud Render Service — RebusFarm.net*, RebusFarm, (2021; `https://es.rebusfarm.net/en/`).
6. S. Render, *SummuS Render - Render Farm*, SummuS Render, (2021; `https://www.summus.es/`).
7. *Free JavaScript training, resources and examples for the community*, Javascript.com, 2016, (`https://www.javascript.com/`).
8. N. Foundation, *Node.js*, Node.js, 2019, (`https://nodejs.org/en/`).
9. *npm — build amazing things*, Npmjs.com, 2019, (`https://www.npmjs.com/`).
10. *three.js – Javascript 3D library*, Threejs.org, 2019, (`https://threejs.org/`).
11. O. Foundation, *Express - Node.js web application framework*, Expressjs.com, 2017, (`https://expressjs.com/`).
12. *HTML5 - MDN Web Docs Glossary: Definitions of Web-related terms — MDN*, developer.mozilla.org, (2021; `https://developer.mozilla.org/en-US/docs/Glossary/HTML5`).
13. *CSS: Cascading Style Sheets*, MDN Web Docs, jun. de 2019, (`https://developer.mozilla.org/en-US/docs/Web/CSS`).
14. B. Foundation, *blender.org - Home of the Blender project - Free and Open 3D Creation Software*, blender.org, 2019, (`https://www.blender.org/`).
15. *Enterprise Application Container Platform — Docker*, Docker, 2018, (`https://www.docker.com/`).
16. Docker, *Overview of Docker Compose*, Docker Documentation, feb. de 2020, (`https://docs.docker.com/compose/`).
17. Microsoft, *Visual Studio Code*, Visualstudio.com, abr. de 2016, (`https://code.visualstudio.com/`).
18. *GitHub*, GitHub, 2018, (`https://github.com/`).

19. usability.gov, *Use Cases — Usability.gov*, Usability.gov, 2019, (`https://www.usability.gov/how-to-and-tools/methods/use-cases.html`).

20. A. Faisandier y G. Garry, *System Architecture - SEBoK*, Sebokwiki.org, 2011, (`https://www.sebokwiki.org/wiki/System_Architecture`).

21. P. Bourke, *Object Files (.obj)*, paulbourke.net, (2021; `http://paulbourke.net/dataformats/obj/`).

22. IFML, *IFML: the Interaction Flow Modeling Language — the OMG Standard for front-end Design*, IFML: the Interaction Flow Modeling Language, 2018, (2021; `https://www.ifml.org/`).

23. K. Group, *OpenGL - The Industry Standard for High Performance Graphics*, www.opengl.org, (2021; `https://www.opengl.org//`).

24. B. Caulfield, *What's the Difference Between Ray Tracing, Rasterization? — NVIDIA Blog*, The Official NVIDIA Blog, abr. de 2019, (2021; `https://blogs.nvidia.com/blog/2018/03/19/whats-difference-between-ray-tracing-rasterization/`).

25. *JSON*, www.json.org, (`https://www.json.org/json-en.html`).

26. *Swagger Editor*, editor.swagger.io, (`https://editor.swagger.io/`).

27. *FormData - Web APIs — MDN*, developer.mozilla.org, (2021; `https://developer.mozilla.org/en-US/docs/Web/API/FormData`).

28. B. Foundation, *Blender 2.92.0 Python API Documentation — Blender Python API*, docs.blender.org, (`https://docs.blender.org/api/current/index.html`).

29. , *How to convert quaternions from three.js system to Blender system?*, three.js forum, feb. de 2019, (2021; `https://discourse.threejs.org/t/how-to-convert-quaternions-from-three-js-system-to-blender-system/5835/3`).

30. *Docker Documentation*, Docker Documentation, jul. de 2020, (`https://docs.docker.com`).

31. *Cloud Application Platform — Heroku*, www.heroku.com, (`https://www.heroku.com`).

32. *OAuth 2.0 — OAuth*, Oauth.net, 2020, (`https://oauth.net/2/`).

33. *7-Zip*, 7-zip.org, 2019, (`https://www.7-zip.org/`).

Apéndices

APÉNDICE A

Guía de Instalación

En este apéndice se describirá el conjunto de pasos necesarios que hay que seguir para hacer una instalación completa de la aplicación de este proyecto. El ejemplo de instalación se hará en un sistema con Windows 10 de 64 bits, sin embargo, la mayoría de programas que se usaran están disponibles tanto en macOS como en Linux, aunque alguna de las funcionalidades no estarán disponibles en la versión de macOS como explicaremos más adelante. Esta guía estará compuesta por dos secciones diferentes, en una se explicará el proceso de instalación común de la aplicación con todos los programas necesarios en la máquina y en la otra, se explicará como hacer la instalación del programa haciendo uso de Docker.

A.1. Código fuente

Para ambos tipos de instalación, es necesario que nos descarguemos previamente el código fuente de la aplicación. Este se encuentra en un repositorio público de GitHub, al cuál podemos acceder desde este enlace. Una vez abierto el enlace, veremos una página como la siguiente:

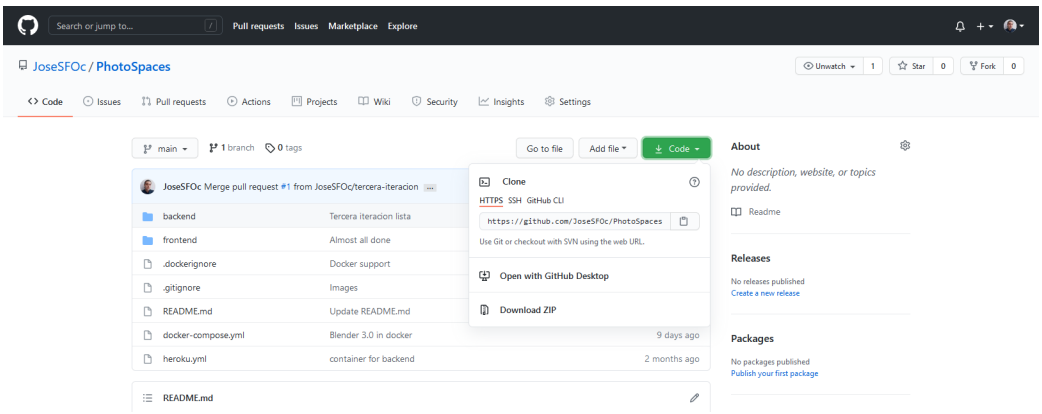

Figura A.1: Repositorio de GitHub

Si pulsamos sobre el botón en el que pone *Code*, se nos desplegará un menú con la dirección de Git del repositorio por si queremos clonarlo a través de Git o, en nuestro caso, pulsaremos sobre el botón *Download ZIP*. Una vez descargado el proyecto, veremos que está en un formato de compresión *.zip*, existen multitud de herramientas para explorar y descomprimir estos archivos, aunque en esta guía se usará la herramienta de código abierto *7zip*[33].

Cuando hayamos extraído el fichero en el directorio de nuestra elección, tendremos todo el código fuente que compone la aplicación, ahora necesitaremos los programas de los que depende.

A.2. Instalación completa

A.2.1. Node.js

La instalación de node.js es muy simple, ya que esta guiada por el instalador que nos descargaremos. La versión que usaremos será la última disponible, ya que en la aplicación ha sido implementada considerando las diferentes versiones de node.js. Accederemos a través de este enlace a la página de descarga de node.js, donde seleccionaremos el instalador correspondiente a nuestra máquina, en nuestro caso será el instalador de Windows de 64 bits.

Figura A.2: Página de descarga de node.js

Con el instalador descargado, podremos ejecutarlo para que comience el proceso de instalación. En la primera ventana aparecerá un mensaje de bienvenida y pulsaremos *Next* para continuar en la instalación. Luego nos pedirá que aceptemos la licencia para continuar, así que marcamos que aceptamos los términos y continuamos.

Figura A.3: Página de términos y condiciones del instalador de node.js

A continuación nos preguntará donde queremos instalar node.js, podemos seleccionar la carpeta que deseemos y continuar. Tras esto nos preguntará que funcionalidades queremos instalar, no cambiaremos ninguna opción y proseguiremos, ya que así, tendremos instalado también el gestor de librerías *npm*.

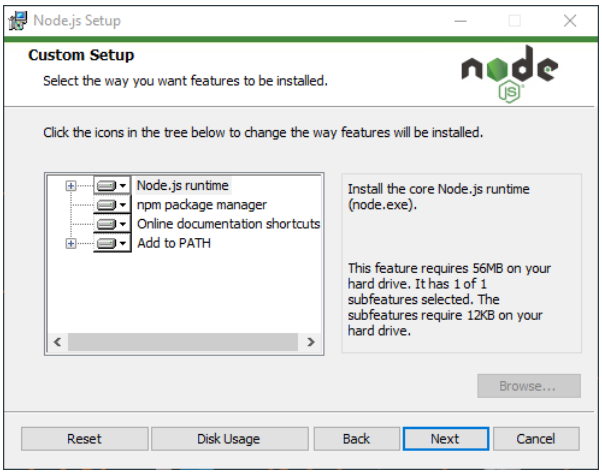

Figura A.4: Página de las funcionalidades del instalador de node.js

Tras esto nos preguntará si deseamos instalar herramientas alternativas para la instalación de diferentes librería de npm, seleccionaremos que sí y continuaremos. Luego nos preguntará si estamos seguros de instalar node.js, y pulsaremos *Install* para continuar, empezando la instalación de node.js. Cuando acabe la instalación nos dirá que node.js ha sido instalado correctamente y podremos cerrar el instalador.

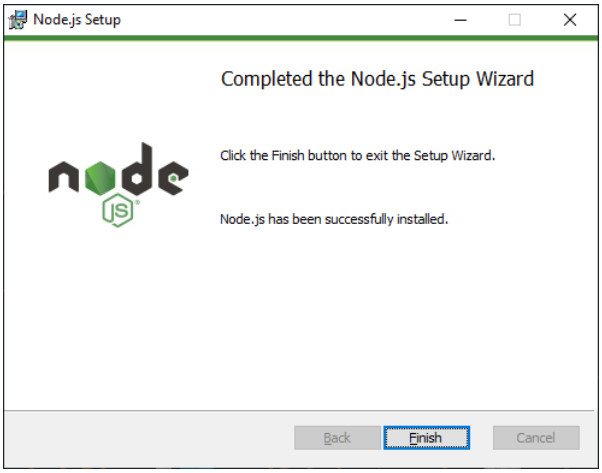

Figura A.5: Página de instalación exitosa del instalador de node.js

Una vez instalado node.js tendremos que reiniciar nuestro ordenador para que se apliquen los cambios correspondientes a la instalación y tras ello podemos pasar a la instalación de Blender.

A.2.2. Blender

La instalación de Blender se hará de dos maneras diferentes en esta guía. La primera será la instalación de la versión del programa que contiene el *add-on* para el motor de Eevee que permite el uso de iluminación global, esta versión es la que ha sido usada en el desarrollo del proyecto y solo tiene dos versiones disponibles, para Windows y para Linux. La segunda versión será la instalación común de Blender con ayuda de su instalador pero no tendremos acceso al uso de *SSGI* en los renderizados que hagamos con Eevee, todas las demás funcionalidades siguen incluidas.

Blender con SSGI

Para acceder a la página de descarga de esta versión de Blender, tendremos que acceder a través de Google Drive y de este enlace. Una vez dentro de la carpeta de Drive, veremos distintas versiones y otros archivos. Depende de la versión de nuestro sistema descargaremos un archivo u otro:

- **Windows:** nos tendremos que descargar el archivo llamado *Windows - SSGI Native 1.13 (New) - 3.00 alpha (Cuda, Optix).zip*.
- **Linux:** nos tendremos que descargar el archivo llamada *Linux - SSGI Native 1.13 (Old) - 3.0 alpha (CPU).7z*.

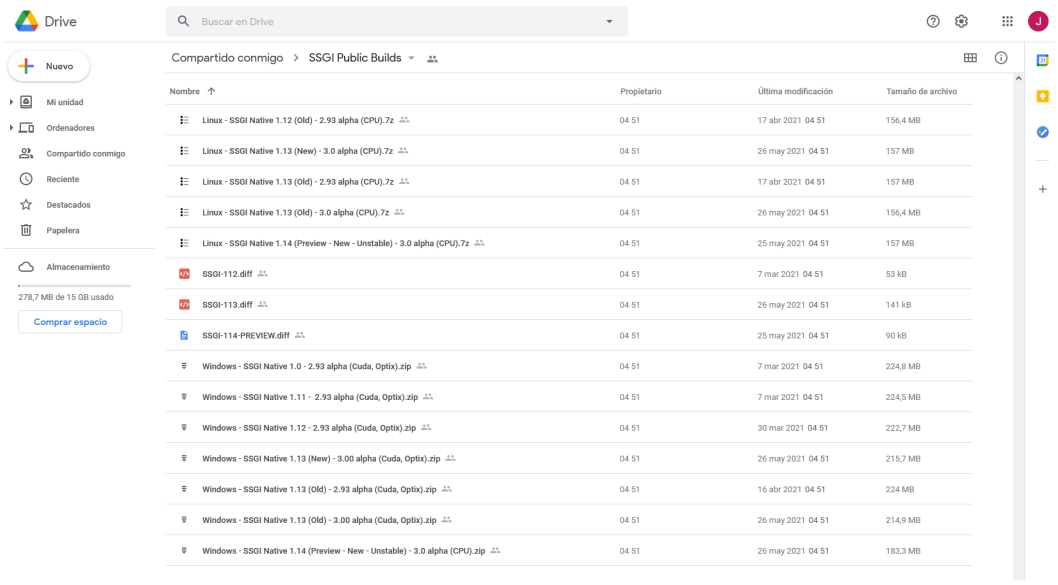

Figura A.6: Descarga de Blender en Google Drive

Blender con instalador

Para instalar Blender a través de su instalador, iremos a la página de descarga de Blender a través de este enlace y una vez dentro descargaremos el instalador dependiendo de la máquina en la que queramos instalarlo.

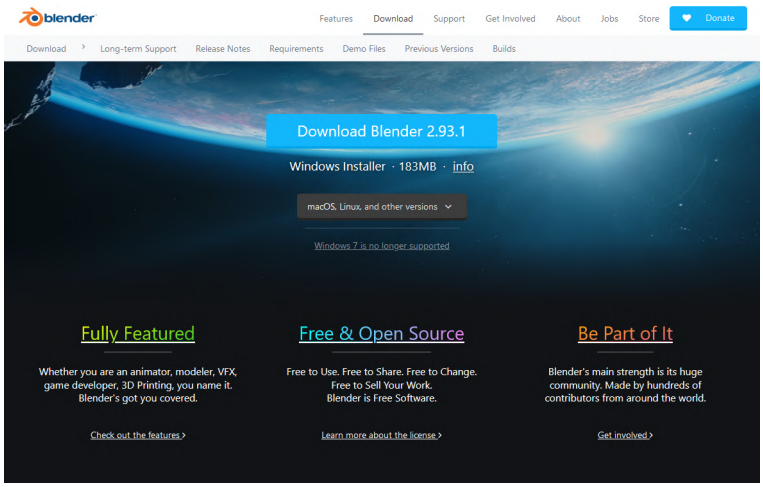

Figura A.7: Página de descarga de Blender

Una vez que hayamos descargado el instalador, procederemos a ejecutarlo. El proceso de instalación en Windows es el que sigue. Después de la página de bienvenida del instalador, tendremos que aceptar las condiciones y términos de uso.

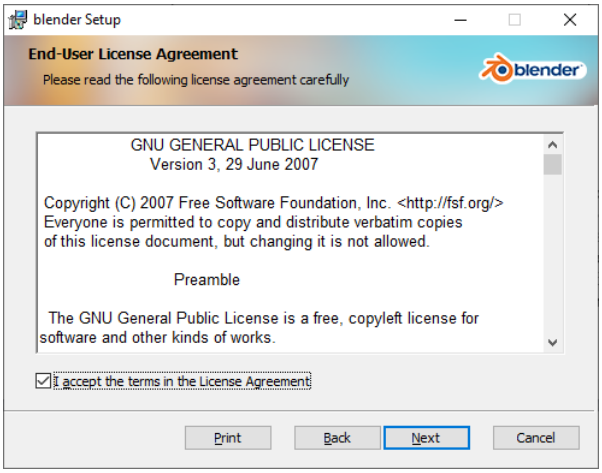

Figura A.8: Página de condiciones del instalador de Blender

Tras esto Blender nos preguntará donde queremos hacer la instalación y que componentes queremos instalar, usaremos la dirección que tiene por defecto el instalador y seguiremos.

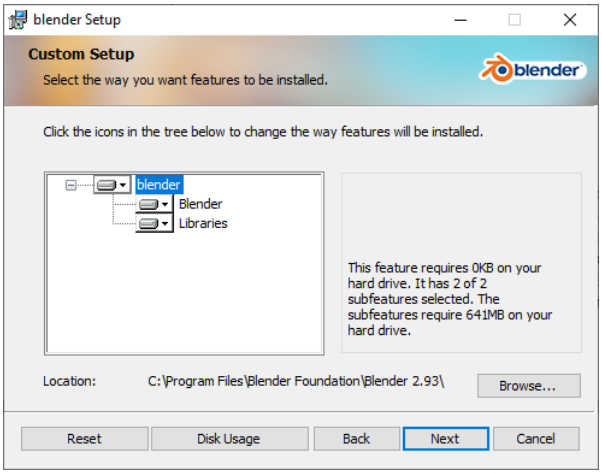

Figura A.9: Página ubicación de la instalación de Blender

A continuación nos pedirá confirmación para realizar la instalación, por lo que pulsaremos *Install* y comenzará a instalar automáticamente Blender. Una vez finalizada la instalación correctamente nos indicará que se ha completado y podremos cerrar el instalador. Y ahora, indiferentemente de la versión de Blender que se haya instalado, debemos añadirla al *PATH* del sistema.

Añadir Blender al PATH del sistema

Cuando hayamos descargado la versión correspondiente tendremos que extraerla como hicimos anteriormente con el código fuente en el directorio que deseemos. A continuación tendremos que añadir Blender al *PATH* de nuestro ordenador. En Windows tendremos que buscar en la barra de navegación *Editar las variables de entorno del sistema* y pulsar *enter*.

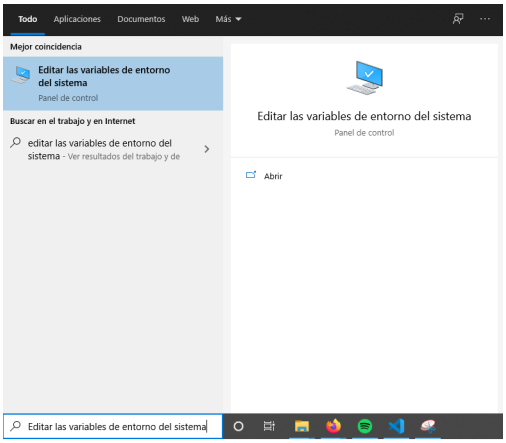

Figura A.10: Variables de entorno en Windows

Se nos abrirá una ventana como la de la figura A.11 y tendremos que pulsar el botón de *Variables de entorno...* En la ventana que se abrirá tenemos que buscar en el recuadro inferior una variable llamada *Path*, la seleccionaremos y pulsaremos el botón *Editar...*

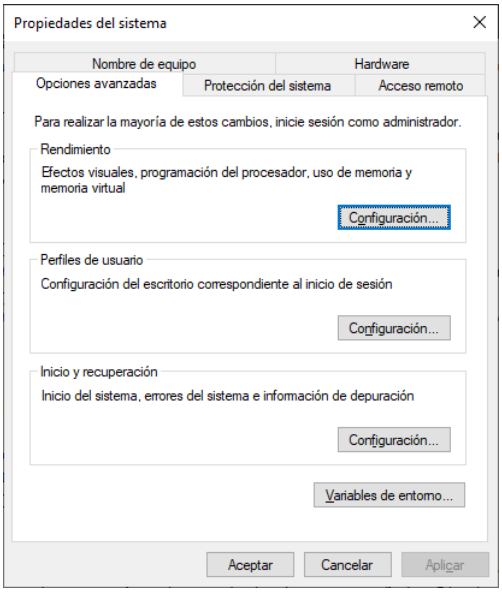

Figura A.11: Ventana principal de variables de entorno

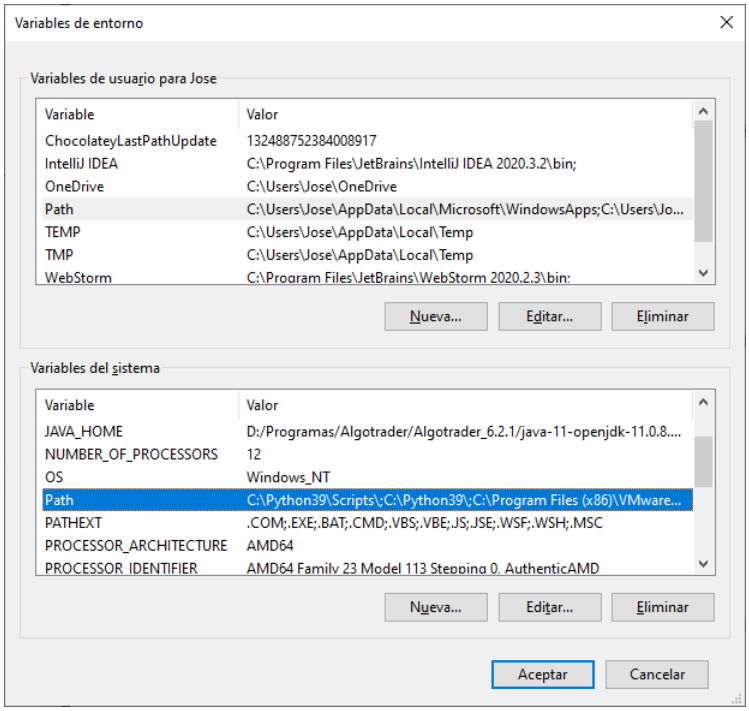

Figura A.12: Ventana que contiene las variables de entorno del sistema con la variable *Path* seleccionada

Al pulsar el botón editar aparecerá una ventana con todas las variables asociadas al *Path* de nuestro sistema. Aquí pulsaremos el botón de *Examinar* para buscar el archivo de ejecución de Blender. En el explorador que se nos ha abierto navegamos hasta la ubicación donde está instalado o descomprimido los archivos de Blender. Tras esto podemos pulsar en aceptar en todas las ventanas que se nos han ido abriendo.

A.2.3. Instalación de las librerías npm y arranque

Una vez completados los pasos anteriores satisfactoriamente, tendremos que instalar las diferentes librerías que compone la aplicación de node.js. Para esto abriremos una consola en el sistema en el que nos encontremos e iremos a la carpeta donde tenemos descomprimido el código fuente de la aplicación, dentro de la carpeta *Photospaces-main* tendremos dos carpetas diferentes (*backend* y *frontend*) y unos cuantos archivos.

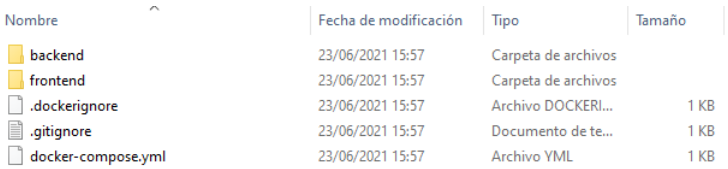

Figura A.13: Carpeta del código fuente

Desde aquí entraremos mediante la consola en la carpeta de *backend* y ejecutaremos el comando `npm install`. Si este comando no funciona, tendremos que añadir node.js al PATH del sistema, siguiendo el mismo procedimiento que para Blender, solo que seleccionando la carpeta donde se instaló node.js.

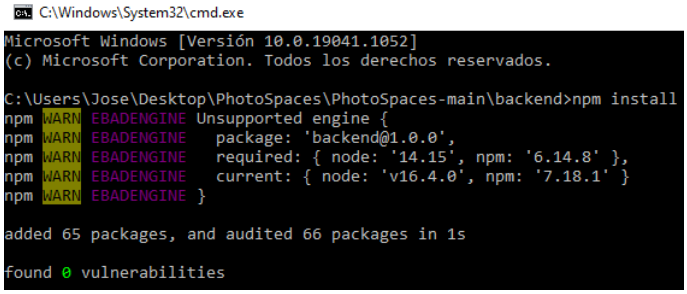

Figura A.14: Instalación de librerías del servidor

Cuando se haya ejecutado el comando correctamente, haremos lo mismo en la carpeta de *frontend*, nos desplazaremos hasta ella en la consola y ejecutaremos el comando `npm install`. Una vez finalizada esta instalación ya tendremos todo lo necesario para ejecutar nuestra aplicación.

A.2.4. Arranque de la aplicación

Con la aplicación ya instalada, solo tendremos que desplegar cada parte por separado para poder usarlas. Para desplegar la aplicación del cliente volveremos a la carpeta *frontend* a través de la consola y ejecutaremos el comando `npm start`.

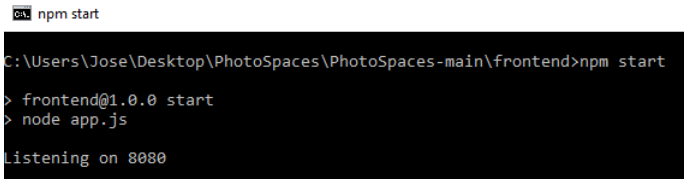

Figura A.15: Arranque de la aplicación del cliente

Así tendremos la aplicación del cliente desplegada y podremos acceder a ella a través de un navegador en la dirección http://localhost:8080/.

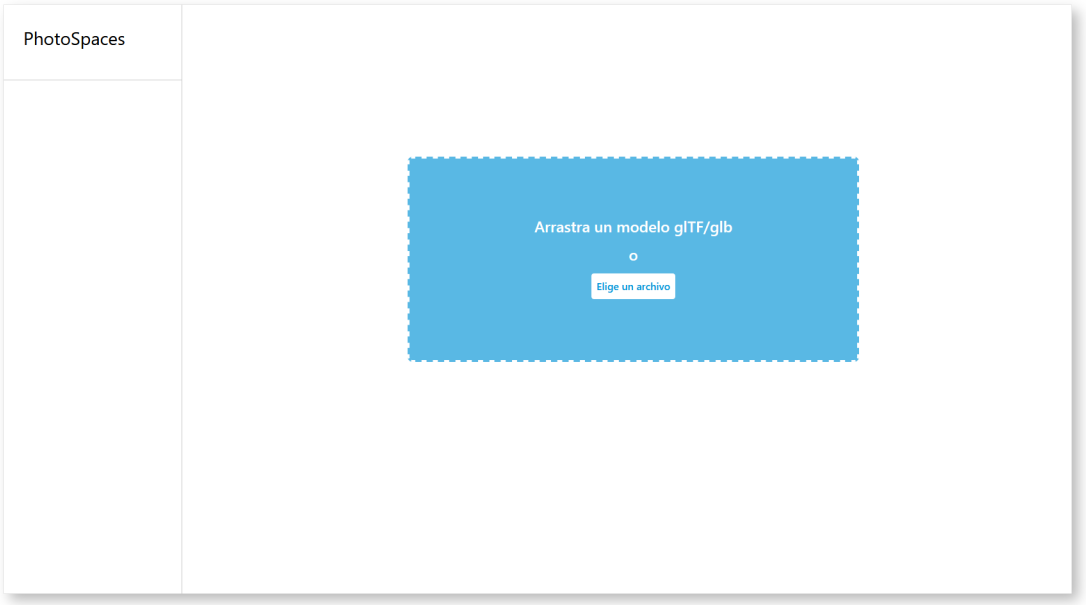

Figura A.16: Página principal de PhotoSpaces

Para el servidor, el procedimiento será el mismo, a través de la consola iremos a la carpeta *backend* y ejecutaremos el comando `npm start` para arrancar el servidor. Una vez iniciado tendremos acceso a la API en la dirección http://localhost:3030/

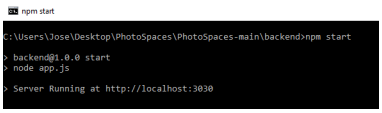

Figura A.17: Arranque de la aplicación del servidor

Así, ya tendríamos la aplicación completa lista para ser usada.

A.3. Instalación con Docker

Aunque Docker sea una herramienta que nos permite aislar las aplicaciones del sistema, las imágenes que pueden ser realmente virtualizadas deben compartir la arquitectura de nuestro procesador. Por lo que esta aplicación ha sido creada para los sistemas con un procesador *x86-64*, quedando fuera del posible uso de la misma procesadores de otro tipo, sin embargo, existe la posibilidad de ejecutar Docker con funciones de emulación, pero esto no será tratado en esta guía.

Para empezar con la instalación debemos ir a la página de descarga de Docker que se encuentra en este enlace.

Figura A.18: Página de descarga de Docker

Una vez descargado el instalador correspondiente para nuestra máquina, lo ejecutaremos y comenzará a descargar algunos archivos adicionales. Cuando la descarga se haya completado aparecerá una ventana con seleccionables de lo que queremos instalar, marcaremos la opción de instalar los componentes *WSL* y la opción de el acceso directo queda a elección del lector.

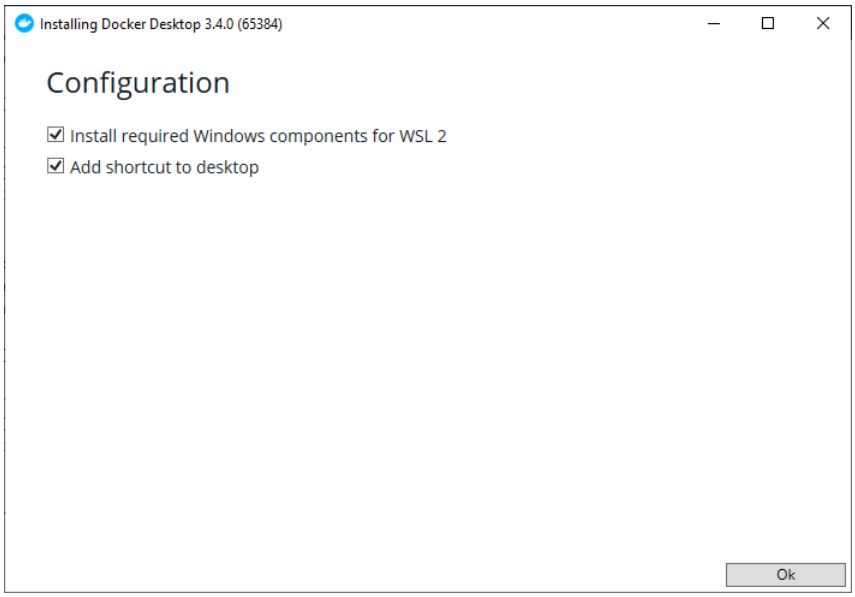

Figura A.19: Página de principal del instalador de Docker

Tras aceptar los componentes que se instalarán comenzará la instalación de Docker. Una vez completada la instalación ya tendremos Docker instalado. Se recomienda reiniciar el ordenador para aplicar los cambios que sean necesarios.

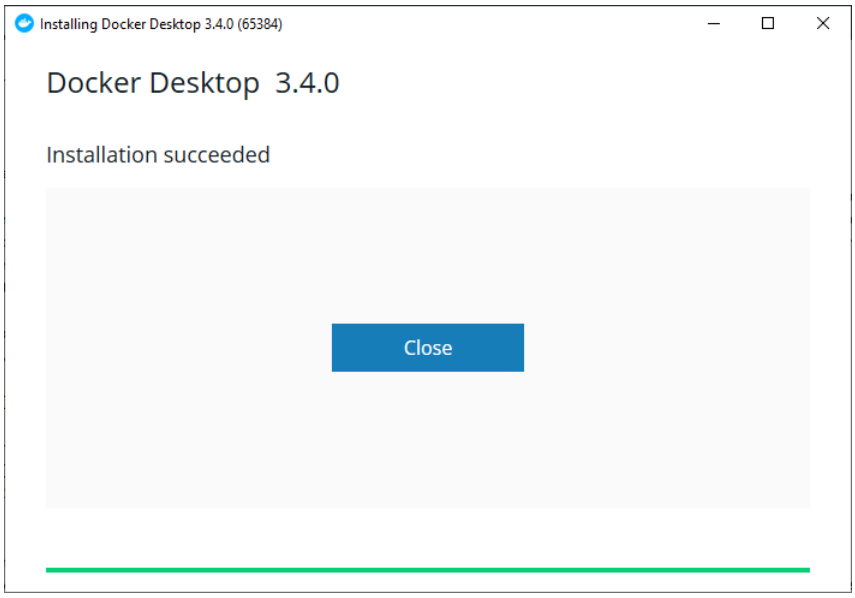

Figura A.20: Página de instalación completada de Docker

A.3.1. Arranque de la aplicación con Docker

Docker Desktop trae consigo Docker Compose, que nos va a permitir crear los contenedores y ejecutarlos. Para comenzar vamos abrir Docker y cuando esté listo abriremos una consola y nos desplazaremos a la carpeta del código fuente que contiene el archivo *docker-compose.yml*. Desde aquí ejecutaremos el comando para construir los contenedores `docker-compose build`.

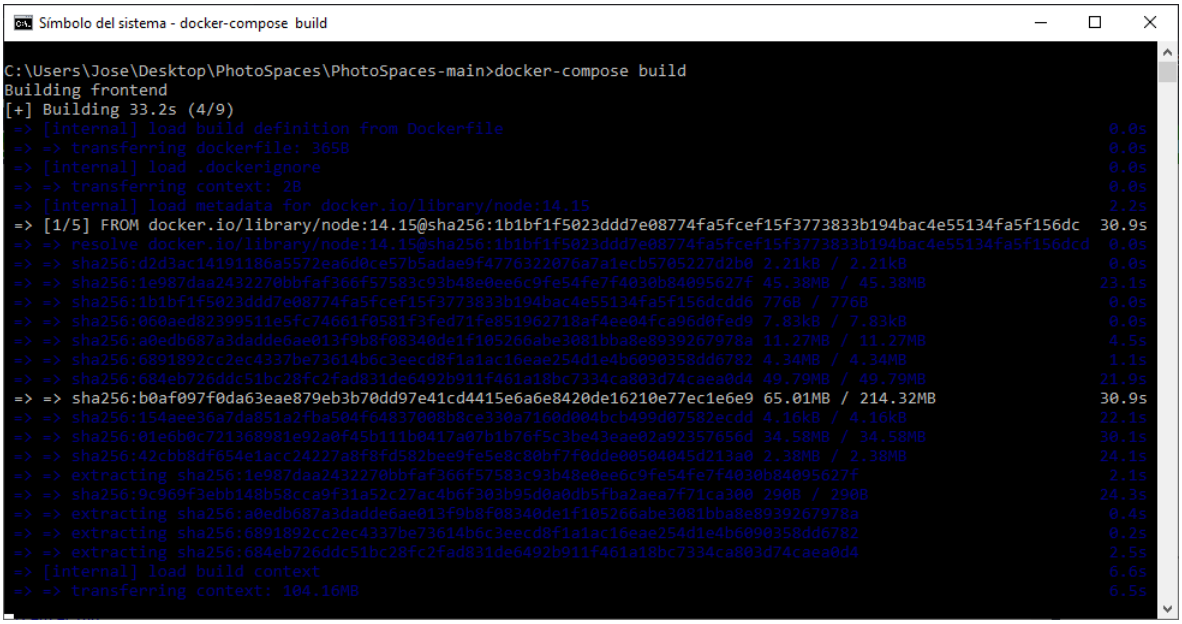

Figura A.21: Construcción de las diferentes imágenes con Docker Compose

Entonces comenzará la creación de las dos imágenes *frontend* y *backend*. Una vez finalizada la construcción de las imágenes podemos ejecutar el comando `docker-compose up` para comenzar la aplicación al completo.

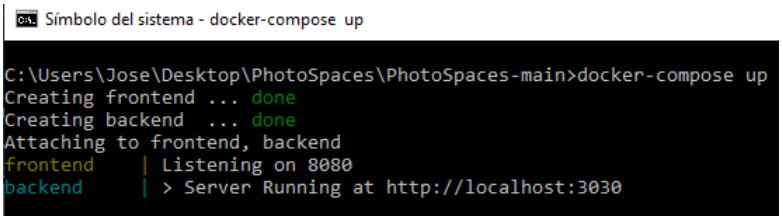

Figura A.22: Aplicación desplegada con Docker Compose

Así ya tendremos desplegada la aplicación al completo y podrá ser usada como la instalación completa, teniendo acceso a la aplicación del cliente en el puerto 8080 y a la aplicación del servidor en el puerto 3030.

APÉNDICE B

Manual de Usuario

En este apéndice se agruparan los pasos necesarios para hacer un uso correcto y completo de la aplicación *PhotoSpaces*. Se abarcarán todas las funcionalidades de la aplicación al completo, sin entrar en detalles de las diferencias entre las versiones existentes.

La página principal de la aplicación nos da la bienvenida con el nombre de la misma en la parte superior izquierda y una caja central, en este elemento central será donde podamos arrastrar un modelo para cargarlo en la aplicación o seleccionarlo de nuestra máquina pulsando el botón.

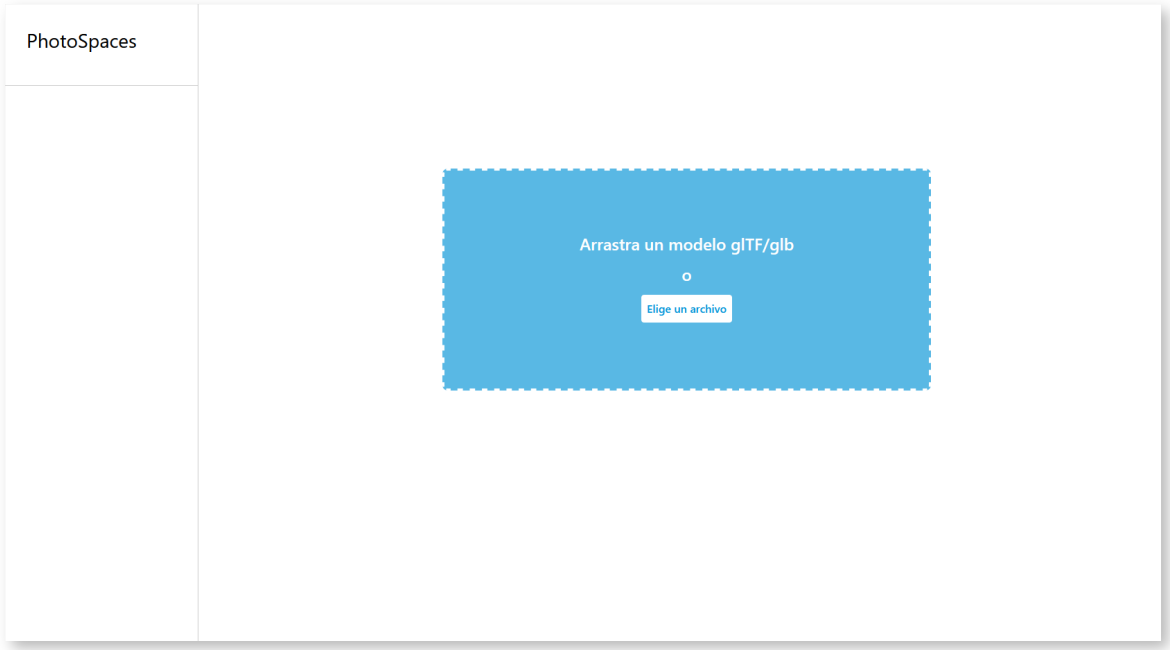

Figura B.1: Página principal de la aplicación

Cuando seleccionemos un archivo para subir, este deberá estar en formato *.gltf* o su versión binaria, *.gbl*. Si el archivo seleccionado no pertenece a ninguno de estos formatos, la aplicación lanzará un mensaje de error y deberemos elegir otro. Si subimos un modelo correcto, la aplicación indicará que está cargando y al terminar, se mostrará el modelo por pantalla y el menú del lado izquierdo tendrá nuevas opciones.

Figura B.2: Modelo cargado correctamente en la aplicación

A la izquierda han aparecido nuevas opciones como podemos apreciar en la figura B.2, expliquemos un poco algunos de estos elementos:

- **Botón limpiar:** su pulsación hará que el modelo actual se elimine del visor y el panel lateral vuelva a su situación anterior, es decir, volveremos a la misma situación que en la que estábamos en la figura B.1.
- **Instrucciones:** en esta pequeña sección encontraremos de manera escrita las siguientes instrucciones para navegar el modelo correctamente.
- **Renderizado:** aquí podemos encontrar la selección de parámetros de renderizado, pudiendo elegir el motor y sus parámetros (en el caso de Eevee) y el campo de visión (FOV).
- **Acciones:** en esta parte encontraremos un menú desplegable que nos permitirá cambiar el fondo que se muestra en nuestra escena, este fondo no será trasladado al renderizado final y será usado únicamente como apoyo para el usuario.

Para navegar el modelo tendremos que hacer clic izquierdo sobre la parte donde se está mostrando el mismo y entonces nuestro cursor se bloqueará, para salir de este bloqueo solo tenemos que volver a hacer clic. Desde aquí, ya podemos movernos por el modelo con los siguientes controles:

- **Ratón:** el movimiento del ratón nos permitirá mover la cámara libremente.
- **Teclas WASD:** nos permitirán movernos en la escena hacia delante (W), hacia atrás (S), hacia la izquierda (A) y hacia la derecha (D).
- **Rueda del ratón:** su movimiento hacia arriba aumentará la velocidad de nuestro movimiento y hacia abajo la disminuirá.
- **Barra espaciadora:** si la pulsamos nos moveremos hacia arriba en la escena.
- **Shift izquierdo:** si pulsamos esta tecla nos moveremos hacia abajo.

Si en vez de hacer clic izquierdo en el visor, hacemos un clic derecho, se desplegará un menú en el que podremos añadir luces al modelo.

Figura B.3: Menú de luces desplegado en el visor

Desde este menú podremos seleccionar entre dos tipos de luces diferentes:

- **Luces de punto:** estas luces emiten luz en todas direcciones como si de una estrella se tratase, podemos modificar su posición e intensidad en el menú del objeto.
- **Luces direccionales:** estas luces emiten luz en una única dirección, podemos modificar su intensidad, posición y la dirección a la que está apuntando.

Si añadimos cualquiera de los dos tipos de luces, estas serán añadidas a la escena y aparecerá un nuevo menú en el visor en el que podremos modificar las propiedades de estos objetos, y una nueva sección en el menú lateral, desde la que podremos ver una lista de las luces que tiene el modelo e incluso, eliminarlas pulsando en el icono de la papelera.

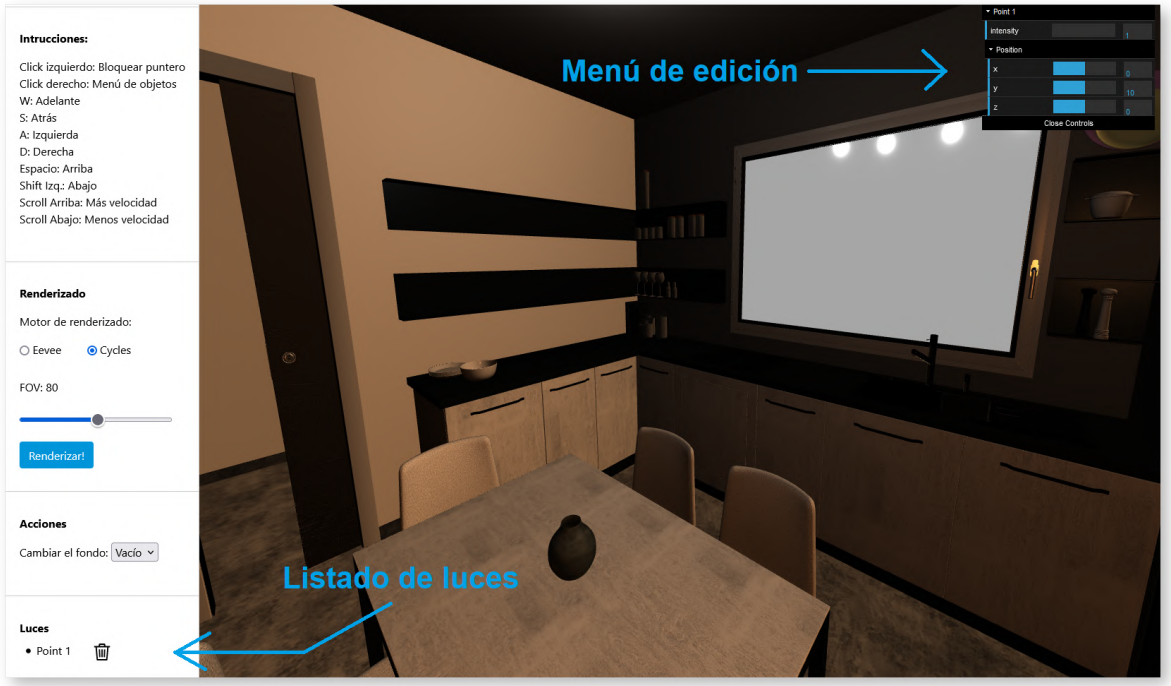

Figura B.4: Listado y propiedades de las luces en el visor

Con las luces posicionadas en la escena como deseamos y la perspectiva de la cámara selecciona, nos queda elegir los diferentes parámetros que serán usados en el renderizado del modelo. En la parte de renderizado del menú lateral podemos encontrar varias opciones:

- **Motor de renderizado:** aquí podremos elegir entre Eevee o Cycles, si seleccionamos Eevee se nos desplegarán las opciones que veremos a continuación.
- **Oclusión ambiental:** esta opción podrá ser marcada o desmarcada, en caso de estarlo el modelo procesado calculará como tienen que estar las superficies afectadas por la luz ambiental.
- **Resplandor:** esta opción que también podrá ser marcada o desmarcada permitirá activar el resplandor de la imagen o el también llamado *bloom*. Este efecto simula el resplandor de ciertas superficies a partir de una exposición de luz.
- **SSGI:** esta opción nos permitirá activar el calculo de luces globales, que permite simular como rebotaría la luz en ciertas superficies, siendo reflejadas por la escena.
- **FOV:** esta barra la podremos aumentar o disminuir, tanto habiendo seleccionado Eevee como Cycles y nos permite aumentar o disminuir el campo de visión de la cámara.

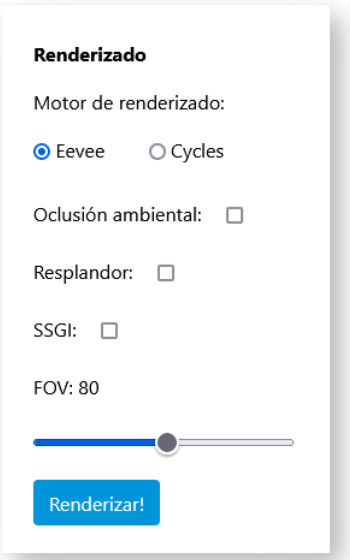

Figura B.5: Parámetros de renderizado

La diferencia principal entre los dos motores de renderizado es la velocidad y la calidad, mientras que el procesado de Eevee es simulado y tiene algo menos de calidad su velocidad de renderizado es casi instantánea; por otro lado, Cycles es más lento ya que calcula como interaccionarían los rayos de luz en la escena, consiguiendo un resultado de una calidad mucho mejor. Una vez seleccionados todos los parámetros que usaremos y tengamos la perspectiva del modelo como nos gusta, podemos proceder a pulsar el botón de renderizar.

Figura B.6: Información del proceso de renderizado

Al comenzar la renderización aparecerá una ventana con información del proceso:

- **Cola:** nos muestra la cantidad de peticiones que están esperando en cola a que se procese su modelo.
- **Tiempo estimado:** es una estimación del tiempo restante que le queda para ser finalizada la imagen que se está procesando actualmente.

Una vez finalizado el proceso de renderizado, la ventana de información se cerrará y en el menú lateral aparecerá un nuevo botón en la sección de *Acciones*. Este botón nos permitirá descargar la imagen que ha sido procesada a nuestro equipo. Así, ya habríamos cubierto la funcionalidad completa de la aplicación y obtenido como resultado nuestra imagen final que podemos ver en la figura B.7.

Figura B.7: Resultado final del proceso de renderizado usando el motor Cycles

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA