



UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Renderizado fotorrealista de carrocerías de automóvil

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Fabregat López, Sergio

Tutor/a: Abad Cerdá, Francisco José

CURSO ACADÉMICO: 2024/2025

Agradecimientos

A mis padres, Carlos y Silvia, por cuidarme todos los días, incluido hoy.

A mi hermano, Jorge, por estar siempre disponible para echar una mano.

A mi tutor, Paco, por haberme ofrecido la oportunidad formativa de mi vida, y por su guía durante el proceso.

A mis amigos, Daniel y Azahar, por todas las madrugadas juntos en Discord haciendo el TFG.

Índice

1. Introducción	5
1.1. Motivación	5
1.2. Objetivos	5
2. Fundamentos teóricos	7
2.1. Radiometría	7
2.2. Ecuación de renderizado	7
2.3. Interacción entre luz y superficie	8
2.3.1. Reflectancia de Fresnel	8
2.3.2. Teoría de microfacetas	9
2.4. Transporte de luz	9
2.4.1. Luces analíticas	10
2.4.2. Luz ambiental	10
2.4.3. Trazado de rayos	10
2.5. Modelo de cámara	11
3. Estado del arte	12
3.1. NVIDIA Iray	12
3.2. Unreal Engine 5	13
3.3. V-Ray	13
3.4. AMD Radeon ProRender	14
4. Análisis del problema	15
4.1. NVIDIA Iray	15
4.1.1. Disponibilidad de renderizado en tiempo real y <i>offline</i>	15
4.1.2. Facilidad de integración e interoperatividad	16
4.1.3. Abundancia de documentación y accesibilidad	16
4.1.4. Licencia de código abierto o gratuita	16
4.1.5. Conclusión	16
4.2. Unreal Engine 5	17
4.2.1. Disponibilidad de renderizado en tiempo real y <i>offline</i>	17
4.2.2. Facilidad de integración e interoperatividad	17
4.2.3. Abundancia de documentación y accesibilidad	17
4.2.4. Licencia de código abierto o gratuita	17
4.2.5. Conclusión	17
4.3. V-Ray	18
4.3.1. Disponibilidad de renderizado en tiempo real y <i>offline</i>	18
4.3.2. Facilidad de integración e interoperatividad	18
4.3.3. Abundancia de documentación y accesibilidad	18
4.3.4. Licencia de código abierto o gratuita	18
4.3.5. Conclusión	19
4.4. AMD Radeon ProRender	19
4.4.1. Disponibilidad de renderizado en tiempo real y <i>offline</i>	19
4.4.2. Facilidad de integración e interoperatividad	19
4.4.3. Abundancia de documentación y accesibilidad	19
4.4.4. Licencia de código abierto o gratuita	19
4.4.5. Conclusión	20
5. Solución propuesta	21
5.1. Tecnologías empleadas	21
5.1.1. Radeon ProRender	21
5.1.2. Visual Studio 2022	21
5.1.3. OpenGL	21
5.1.4. GLAD	22
5.1.5. GLM	22
5.1.6. stb_image_write	22
5.1.7. Herramientas propietarias de la empresa	22
5.1.8. GLFW	22
5.1.9. Dear ImGUI	22

6. Desarrollo de la solución	24
6.1. Diseño de la clase	24
6.1.1. Puntero opaco	24
6.1.2. Funciones y variables miembro	24
6.1.3. Tratamiento de errores	25
6.2. Escena	25
6.2.1. Suelo	25
6.2.2. Luz ambiental	25
6.3. Modelo de la superficie a inspeccionar	26
6.4. Material de la superficie a inspeccionar	27
6.4.1. <i>Base Color</i>	27
6.4.2. <i>Roughness</i>	28
6.4.3. <i>Metallic</i>	28
6.4.4. <i>Clearcoat</i>	29
6.4.5. <i>Clearcoat Roughness</i>	29
6.4.6. Mapa de normales	30
6.5. Luces	32
6.5.1. Material Emisivo	32
6.5.2. Cálculo de la intensidad luminosa	33
6.5.3. Mallas	34
6.6. Cámaras	34
6.6.1. Ajuste de exposición	35
6.7. Renderizado de la escena	36
6.7.1. Creación del contexto	36
6.7.2. Proceso de renderizado	36
6.7.3. Diferencias entre previsualización e imagen final	37
7. Resultados	40
7.1. Captura de fotografías	40
7.1.1. Dispositivo de captura	40
7.1.2. Elementos capturados	41
7.1.3. Condiciones de captura	43
7.2. Recreación mediante PBRRender	44
7.2.1. Recreación del <i>clearcoat</i> blanco	44
7.2.2. Recreación del <i>clearcoat</i> gris	45
7.2.3. Recreación del <i>clearcoat</i> negro	47
7.2.4. Recreación del <i>primer</i>	47
7.3. Comparación	47
7.3.1. Comparación del <i>clearcoat</i> blanco	48
7.3.2. Comparación del <i>clearcoat</i> gris	51
7.3.3. Comparación del <i>clearcoat</i> negro	51
7.3.4. Comparación del <i>primer</i>	51
7.3.5. Conclusión	59
8. Conclusión	62
8.1. Trabajos futuros	63
A. Entorno de pruebas interactivo	70
A.1. Cámara orbital	70
A.2. Cargador de <i>.obj</i>	70
B. Colección completa de capturas y recreaciones	72
B.1. <i>Clearcoat</i> blanco	72
B.2. <i>Clearcoat</i> gris	74
B.3. <i>Clearcoat</i> gris inclinado	76
B.4. <i>Clearcoat</i> negro	78
B.5. <i>Primer</i>	80

C. Objetivos de Desarrollo Sostenible	82
C.1. ODS 9: Industria, Innovación e Infraestructura	82
C.2. ODS 12: Producción y Consumo Responsables	82
C.3. ODS 8: Trabajo Decente y Crecimiento Económico	82
C.4. ODS 4: Educación de Calidad	82

1. Introducción

Una de las mayores áreas de investigación y desarrollo en informática gráfica es el renderizado basado en física (*PBR*), que busca la generación de imágenes sintéticas indistinguibles de la realidad a través de la simulación fiel de los principios físicos que gobiernan la luz. Esta disciplina tiene una colossal aplicabilidad a la resolución de problemas que se plantean en la industria hoy en día. Un claro exponente de ello es *AUTIS Ingenieros*, una empresa de automatización basada en Gandía, que requiere una solución que les permita recrear fielmente imágenes de carrocerías bajo los túneles de inspección de la empresa de manera intuitiva. Este Trabajo de Fin de Grado nace, precisamente, para dar respuesta a dicho problema.

En lugar de desarrollar un motor desde cero, una tarea que excede el alcance de este proyecto, se ha optado por construir sobre una tecnología existente y validada. Tras un análisis de las soluciones del estado del arte, seleccionamos el *SDK Radeon ProRender* de *AMD*. La contribución principal de este trabajo reside en la creación de una librería de abstracción en C++ sobre este motor. Esta capa de software encapsula la complejidad del *PBR* y expone una interfaz de alto nivel, intuitiva y diseñada específicamente para el caso de uso de *AUTIS*, garantizando así su fácil integración en el simulador existente de la empresa y su uso por parte de ingenieros no especializados en computación gráfica.

A lo largo de este documento, detallaremos el proceso completo de concepción y desarrollo de esta solución. Se comenzará estableciendo los fundamentos teóricos de la radiometría, la ecuación de renderizado y los modelos de interacción luz-materia que sustentan el *PBR*. Posteriormente, analizaremos las tecnologías del estado del arte para justificar la elección de *Radeon ProRender*. A continuación, describiremos en profundidad el diseño y la implementación de la librería de abstracción, explicando las decisiones arquitectónicas tomadas. La validación de la solución se presentará a través de una comparación directa entre las imágenes sintéticas generadas y fotografías de una escena real equivalente, demostrando el grado de fotorrealismo alcanzado. Finalmente, expondremos las conclusiones del proyecto, evaluando el cumplimiento de los objetivos y delineando posibles líneas de trabajo futuro.

1.1. Motivación

La motivación de este Trabajo de Fin de Grado es doble, por un lado arraigada en la resolución de un problema tangible en la industria de la automatización, y por otro fundamentada en una vocación personal por la informática gráfica.

El principal impulsor del proyecto es la necesidad de la empresa. Su área de especialidad es la optimización del control de calidad en las líneas de producción de automóviles. Ha implementado túneles de inspección visual cuyo objetivo es la identificación de anomalías superficiales en las carrocerías, como microarañazos, inclusiones de partículas o defectos de aplicación de la pintura. Con este fin, están dotados de una configuración de iluminación controlada y un conjunto de cámaras estratégicamente posicionadas para cubrir toda la superficie inspeccionada. La meta final de la compañía es automatizar el proceso de detección y clasificación de imperfecciones mediante un modelo de visión por computador basado en inteligencia artificial.

Sin embargo, el desarrollo de dicho modelo requiere la adquisición de datos de entrenamiento. Los algoritmos de aprendizaje profundo requieren volúmenes de datos del orden de millones de imágenes. La inviabilidad práctica y económica de dañar deliberadamente un número tan elevado de vehículos para capturar estas imágenes ha llevado a plantear una solución alternativa, la generación de muestras sintéticas. Para realizar esta tarea se requiere un sistema que genere proceduralmente imágenes fieles a la realidad de carrocerías con una diversidad de imperfecciones. El primer paso para construir este sistema es implementar un motor de renderizado fotorrealista capaz de simular con exactitud la apariencia de una carrocería bajo las condiciones de captura específicas de los túneles de la empresa. Este es uno de los principales objetivos de este Trabajo de Fin de Grado.

En paralelo, este desafío impulsado por una necesidad en la industria se alinea con mis intereses académicos y profesionales. Siento una gran afinidad y pasión por el campo de la informática gráfica, con un enfoque particular en la síntesis de imágenes realistas, el modelado de la interacción de la luz con los materiales y la optimización de estos procesos para su ejecución en tiempo real. Por tanto, este proyecto también representa una valiosa oportunidad formativa. Me permite abordar un desafío técnico complejo y tangible, profundizando en tecnologías de renderizado de última generación y aplicando mis conocimientos a un caso de uso con un impacto directo en la industria.

1.2. Objetivos

Para llevar a cabo con éxito este proyecto y dar respuesta a las necesidades planteadas, se han definido los siguientes objetivos específicos.

1. **Sintetizar imágenes generadas mediante gráficos por computador que reproduzcan fielmente fotografías tomadas en el mundo real.** El objetivo principal es construir un motor de renderizado capaz de generar imágenes sintéticas que sean tan visualmente indistinguibles de fotografías reales como sea posible.
 - a) **Generar imágenes que simulen con exactitud capturas de carrocerías bajo los túneles de *AUTIS*.** Además de generar imágenes fotorrealistas en base a escenas genéricas, debemos, específicamente, simular fielmente las imágenes capturadas de carrocerías en los túneles de inspección. El éxito de este punto es crucial para que los datos generados sean útiles para el entrenamiento del modelo de IA, y por consiguiente, resolvamos el problema en la industria.
2. **Diseñar una solución fácilmente integrable en el simulador de la empresa.** La solución desarrollada no debe ser una aplicación aislada, sino un componente diseñado para ser integrado en el *software* de simulación multifuncional que la empresa ya utiliza a modo de “navaja suiza”.
3. **Soportar renderizado en tiempo real y *offline*.** Se reconoce que el flujo de trabajo requiere dos modos de operación distintos. Por un lado, se necesita un modo en tiempo real (o interactivo) que, aunque sacrifique parte de la fidelidad visual, permita a los usuarios configurar la escena, posicionar las cámaras y ajustar los materiales de forma fluida, ágil y con retroalimentación visual instantánea. Por otro lado, se implementará un modo *offline* que priorizará la máxima calidad y fidelidad sobre la velocidad. Este segundo modo será el destinado para ser empleado para la generación final del conjunto de datos de entrenamiento, donde el fotorrealismo es un requisito clave.
4. **Permitir al usuario definir la escena virtual de manera robusta e intuitiva.** Para que la herramienta sea útil para los ingenieros de la empresa, que no tienen por qué ser expertos en computación gráfica, es fundamental abstraer la complejidad inherente al renderizado fotorrealista. Dicha abstracción debe realizarse sin sacrificar en ningún momento el rigor físico y la fidelidad visual que exige la simulación de un entorno tan específico como los túneles de inspección.
 - a) **Exponer parámetros intuitivos y versátiles de control del material que compone la superficie.** Buscamos abstraer los parámetros complejos de los materiales en controles de alto nivel, para que los usuarios no tengan que tener conocimiento sobre *BRDFs*, Fresnel o teoría de microfacetas para poder crear materiales con el aspecto que desean. Estos a la vez deben de ser suficientes y lo suficientemente versátiles como para poder modelar el abanico de materiales distintos que son relevantes en este contexto.
 - b) **Adaptar la configuración de las fuentes de luz de la escena virtual a las condiciones específicas de los túneles.** La iluminación de los túneles de inspección no es convencional, sino un sistema de ingeniería diseñado explícitamente para revelar imperfecciones superficiales. Por tanto, es un requisito indispensable replicar con exactitud la geometría, intensidad y distribución de los arcos de luz para que las simulaciones generen los mismos patrones de reflexión y sombreado que se observan en el entorno real.
 - c) **Simular fielmente las cámaras empleadas en los túneles.** Para obtener imágenes que sean virtualmente indistinguibles de las capturadas en los túneles se debe de modelar cómo funcionan las cámaras en el mundo real, y emplear los parámetros específicos de las cámaras empleadas en los túneles.

2. Fundamentos teóricos

La informática gráfica siempre ha estado interesada en generar imágenes lo más realistas posible. Este problema ha dado lugar a una inmensidad de técnicas, paradigmas, operaciones e investigaciones. De entre ellos, centramos nuestra atención en **PBR** o **Physically Based Rendering**, un paradigma donde se emplean principios físicos para intentar modelar de manera precisa las interacciones entre luz y materia [1]. Basamos nuestra implementación en este paradigma, ya que dado nuestro objetivo de obtener imágenes fotorrealistas, lo más apropiado es basarse en las leyes que modelan cómo funciona el mundo real.

En las siguientes secciones describimos estas leyes físicas y los algoritmos que se emplean para intentar imitarlas.

2.1. Radiometría

La luz es la radiación electromagnética visible [2], pero en Informática gráfica típicamente se trabaja a nivel de la óptica geométrica, donde la luz se modela como rayos en vez de ondas [3].

Independientemente de cómo se modele, la luz lleva energía. La energía es trabajo ejercido en un determinado período de tiempo [4]. Para medir la luz en un instante de tiempo, empleamos el **flujo radiante** ϕ_e , que es la cantidad de energía radiante por unidad de tiempo, medido en vatios (W) [5].

Cualquier medida de flujo requiere un área sobre la que realizar la medición. Dada un área finita A , podemos definir la **irradiancia** E como la densidad media de flujo sobre el área (Ecuación 1), medida en vatios por metro cuadrado (W/m^2) [4]. La irradiancia es, por tanto, directamente proporcional a la cantidad total de energía radiante que llega a una superficie.

$$E = \frac{\phi_e}{A} \quad (1)$$

La irradiancia nos da el flujo por área en un punto, pero no distingue cómo se distribuye esta energía en función de la dirección. La **radiancia** L , por otro lado, mide la irradiancia con respecto a direcciones representadas como ángulos sólidos (Ecuación 2), midiéndose en vatios partido metro cuadrado por estereoradián ($W/(m^2 sr)$) [4].

$$L = \frac{E_\omega}{\omega} \quad (2)$$

Aquí E_ω denota la irradiancia en una superficie perpendicular a ω , la dirección sobre la que se está midiendo la radiancia [4]. Si la fórmula del cálculo de la radiancia (Ecuación 2) se aplica a una superficie perpendicular respecto cualquier otra dirección θ , se ha de corregir, multiplicando el resultado por el coseno del ángulo entre ω y θ [5].

La radiancia se puede interpretar como una medida de la radiación electromagnética en un solo rayo de luz. Al final, es lo que miden sensores como nuestros ojos y cámaras. Por tanto, el proceso de renderizado en el paradigma *PBR* consiste en calcular la radiancia en un rayo determinado, desde el punto que estamos renderizando hasta la cámara [5]. Esta es la **radiancia saliente** L_o .

2.2. Ecuación de renderizado

Cuando la luz llega a una superficie, parte de la energía radiante vuelve a salir de ella. Nos interesa la energía radiante que sale de una superficie en dirección a la cámara, es decir, L_o . Para calcularla empleamos la **ecuación de reflectancia** (Ecuación 3) [3], que describe cómo una distribución de luz incidente en un punto \vec{p} se transforma en radiancia saliente en la dirección determinada por \vec{v} , basándose en las propiedades de dispersión de la superficie [6].

$$L_o(\vec{p}, \vec{v}) = \int_{\vec{l} \in \Omega} f(\vec{l}, \vec{v}) L_i(\vec{p}, \vec{l})(\vec{n} \cdot \vec{l}) d\vec{l} \quad (3)$$

Donde:

- \vec{p} es el punto que estamos renderizando.
- \vec{v} es el **vector de vista**, el que va desde el punto de interés a la cámara.
- \vec{n} es el **vector normal**, perpendicular a la superficie sobre la que se encuentra \vec{p} , que determina la orientación de dicha superficie.
- $L_i(\vec{p}, \vec{l})$ es la **radiancia entrante**, es decir, la luz que llega al punto \vec{p} desde la dirección determinada por el vector \vec{l} .

- \vec{l} es el **vector de luz**, que determina la dirección de la que proviene la radiancia entrante.
- Ω es el conjunto de direcciones que se encuentran en el hemisferio unitario centrado en la normal por encima de la superficie [3]. La integral suma la contribución de la luz proveniente de cada una de las direcciones de este hemisferio para calcular la luz reflejada en la dirección determinada por \vec{v} .
- $f(\vec{l}, \vec{v})$ es el **BRDF** o **Bidirectional Reflectance Distribution Function**, que nos indica cuánta energía se refleja de una dirección de entrada (dada por \vec{l}) a una dirección de salida (dada por \vec{v}). Esta función describe las propiedades visuales del material del que está compuesta la superficie en el punto \vec{p} [7].
- $(\vec{n} \cdot \vec{l})$ es el factor de corrección mencionado en la sección anterior. Cuando el vector normal y el vector de luz no coinciden, la radiancia se está aplicando a una superficie no perpendicular a la dirección de esta. Por tanto, se ha de corregir mediante el coseno del ángulo entre \vec{n} y \vec{l} , que, al ser vectores unitarios, equivale al producto escalar de ambos [8].

La ecuación de reflectancia (Ecuación 3) modela el comportamiento de la gran mayoría de objetos y materiales, pero no el de los que emiten luz. Para modelar el comportamiento de estos empleamos la **ecuación de renderizado** (Ecuación 4)¹ [7].

$$L_o(\vec{p}, \vec{v}) = L_e(\vec{p}, \vec{v}) + \int_{\vec{l} \in \Omega} f(\vec{l}, \vec{v}) L_i(\vec{p}, \vec{l}) (\vec{n} \cdot \vec{l}) d\vec{l} \quad (4)$$

Esta ecuación añade el término $L_e(\vec{p}, \vec{v})$, que modela la emisión de radiancia en el punto \vec{p} en la dirección de la radiancia saliente, \vec{v} . Es decir, la radiancia saliente en un punto, en una dirección es la radiancia emitida en ese punto, en esa dirección, más la radiancia incidente de todas las direcciones en el hemisferio Ω , escalado por el BRDF y un término coseno [7].

Sin embargo, durante la mayor parte del trabajo omitiremos la radiancia emitida L_e , que no suele ser relevante, y nos centraremos en la Ecuación 3, ya que la mayoría de materiales que modelamos en informática gráfica no son emisivos [3].

2.3. Interacción entre luz y superficie

Se puede observar en la ecuación de reflectancia (Ecuación 3) que la radiancia saliente depende únicamente de dos factores: el BRDF y la radiancia entrante. En esta sección profundizamos en el primero.

Al interactuar con una superficie, la luz se divide en dos partes. Una parte se refleja, siendo esta conocida como el **componente especular** de la luz. La restante se refracta y avanza dentro de la superficie. En el interior del material, la luz se dispersa y es parcialmente absorbida. Este fenómeno se conoce como **dispersión subsuperficial** [3].

La luz refractada que después de “rebotar” por dentro de la superficie consigue escapar puede hacerlo en distintos lugares. Si sale muy próxima de donde entró se conoce como **dispersión subsuperficial local**. Si por otro lado la distancia entre el punto de entrada y el punto de salida es significativa, se llama **dispersión subsuperficial global** [3].

Para modelar la luz que sale por dispersión subsuperficial local se asume que esta luz escapa la superficie por exactamente el mismo lugar por el que se refracta. De esta manera, se puede combinar con la luz reflejada (componente especular) a la hora de calcular la radiancia saliente en el punto \vec{p} . Esta parte de la radiancia entrante correspondiente a la dispersión subsuperficial local se denomina **componente difuso** de la luz [3].

2.3.1. Reflectancia de Fresnel

Sabemos que la luz se divide en dos partes cuando choca con una superficie. Ahora, cuánta luz se refleja y cuánta se refracta viene dado por la **reflectancia de Fresnel** F . Más concretamente, F determina qué porcentaje de la luz incidente total se refleja [3]. Y, por tanto, $1 - F$ es el porcentaje que se refracta.

Las ecuaciones de Fresnel describen cómo depende F de θ_i (el ángulo entre \vec{n} y \vec{l}), n_1 y n_2 . Estos dos últimos parámetros son los índices de refracción (IOR) de la sustancia o material a cada lado de la superficie. n_1 es el IOR de la sustancia “arriba” de la superficie, que suele ser aire, cuyo IOR es

¹Más correctamente, en la ecuación de renderizado se integra sobre todas las direcciones de la esfera unitaria, no del hemisferio, alrededor del punto. Y, $f(\vec{l}, \vec{v})$ representa el *BSDF*, no el *BRDF*. Omitimos estos detalles para mantener el hilo de la explicación.

aproximadamente 1,003. Y n_2 es el IOR de la sustancia “debajo” de la superficie, es decir, el material del que está compuesto la superficie [3].

Pero, las ecuaciones de Fresnel no sólo son costosas desde el punto de vista computacional, sino que también dependen del índice de refracción completo, que es un número complejo que consta de una parte real (el índice de refracción) y una parte imaginaria (coeficiente de extinción), y se especifica por longitud de onda. Por ello, las ecuaciones de Fresnel se aproximan mediante la **aproximación de Schlick** ([Ecuación 5](#)) [9], que hace que F dependa solo de θ_i [3].

$$F'(u) = F_0 + (F_{90} - F_0) \cdot (1 - u^5) \quad (5)$$

Donde:

- $u = \cos(\theta_i)$
- F_0 es el valor de F cuando $\theta_i = 0^\circ$, es decir, cuando la luz es perpendicular a la superficie ($\vec{n} = \vec{l}$). F_0 es un valor que es una propiedad del material. Puede considerarse el color espectral característico del material [3].
- F_{90} es el valor de F cuando $\theta_i = 90^\circ$. A medida que θ_i aumenta y la luz incide sobre la superficie con ángulos cada vez más oblicuos, el valor de F tenderá a aumentar, alcanzando un valor de 1 para todas las frecuencias (color blanco) en $\theta_i = 90^\circ$ [3]. Es decir, $F_{90} = 1$ siempre.

Salvo en motores gráficos esenciales, la aproximación de Schlick es igual de precisa que las ecuaciones de Fresnel [10].

2.3.2. Teoría de microfacetas

Hasta ahora hemos asumido que las superficies de los objetos son perfectamente uniformes y suaves. Pero en la vida real eso no es así. Las superficies están plagadas de imperfecciones e irregularidades microscópicas que son tan pequeñas que no podemos ver a simple vista, pero que afectan a cómo reflejan la luz [11].

En la **teoría de microfacetas** la superficie, llamada **macrosuperficie**, se modela como un conjunto de pequeñas superficies, llamadas **microfacetas**, con distinta pendiente y altitud [9]. La idea fundamental de esta teoría es que, en lugar de modelar geométricamente cada una de estas imperfecciones (lo cual sería computacionalmente inviable), se puede modelar su comportamiento agregado de forma estadística [11].

El papel central en este modelo estadístico lo juega el parámetro de rugosidad. Este parámetro determina las características de la **función de distribución de microfacetas** D , que describe la orientación de las normales de dichas microfacetas con respecto a la normal de la macrosuperficie.

- Una rugosidad baja implica que la distribución de normales es muy concentrada. La mayoría de las microfacetas están alineadas con la normal de la superficie principal. Esto produce reflejos especulares potentes y definidos.
- A medida que el valor de rugosidad aumenta, la distribución de normales se vuelve más amplia y dispersa. Las microfacetas presentan una gran variación en su orientación. Como resultado, la luz se refleja en un cono de direcciones mucho más amplio, lo que difumina la distribución direccional de la luz reflejada y produce reflejos borrosos y tenues [11].

Además de la distribución, la rugosidad también influye en el segundo componente crucial de la teoría, la **función de enmascaramiento y sombreado** G . Dado que las microfacetas tienen diferentes alturas y pendientes, pueden ocultarse unas a otras desde el punto de vista del observador (enmascaramiento) o de la fuente de luz (sombreado). Una superficie más rugosa, con mayores variaciones de altura, tendrá una mayor probabilidad de que estos fenómenos ocurran. La función G modela estadísticamente esta visibilidad. [11].

2.4. Transporte de luz

En la sección anterior discutimos el primero de los factores que influencian la radiancia saliente L_o en la ecuación de reflectancia ([Ecuación 3](#)), el *BRDF*. Ahora profundizaremos en el segundo, radiancia entrante L_i .

Recordemos que L_i representa la luz que llega a la superficie. Esta luz puede proceder directamente de una fuente emisora, siendo conocida entonces como **iluminación directa**. O indirectamente, tras rebotar en otras superficies de la escena, nombrada entonces **iluminación indirecta** [12].

2.4.1. Luces analíticas

El modelo más simple de iluminación directa son las llamadas **luces analíticas**. Estas son un conjunto de fuentes de luz especiales que resuelven la integral de la ecuación de reflectancia ([Ecuación 3](#)) analíticamente [\[13\]](#).

Son una idealización matemática de las fuentes de luz, diseñadas para simplificar los cálculos de iluminación directa. Su característica común es que, para cualquier punto de una superficie, asumen que la luz incide desde una única dirección \vec{l} [\[14\]](#). De esta manera, se evita tener que integrar todo el hemisferio de direcciones Ω .

Esto implica que la fuente de luz es un punto infinitesimalmente pequeño, una aproximación que se aleja de la realidad, ya que las fuentes de luz del mundo real tienen un área [\[14\]](#).

Hay varios tipos de luces analíticas. Discutiremos los más relevantes:

- Las **luces direccionales**, el modelo más simple, se caracterizan mediante una dirección \vec{l} y un color o intensidad, ambos constantes en toda la escena. Carecen de posición en el espacio, por lo que todos los rayos de luz que emiten son paralelos entre sí. Se trata de una abstracción útil para simular fuentes de luz muy lejanas, cuyo tamaño es despreciable en relación con la escena, como el sol iluminando un paisaje [\[14\]](#).
- Las **luces puntuales** sí tienen una posición definida en el espacio, pero siguen careciendo de dimensiones físicas como forma o tamaño. Esto significa que el vector de dirección de la luz \vec{l} ya no es constante, sino que varía según el punto \vec{p} de la superficie que se esté renderizando. En estas luces \vec{l} siempre apunta desde \vec{p} hacia la posición de la luz. El tipo más común es la **luz omnidireccional**, que emite luz de manera uniforme en todas las direcciones desde su posición. Una característica clave de estas luces es que su intensidad disminuye con la distancia [\[14\]](#).

Hay más tipos de luces puntuales, como las luces de foco [\[14\]](#), que omitiremos en este trabajo por brevedad.

2.4.2. Luz ambiental

En iluminación indirecta, el modelo más simple es la **luz ambiental**. Este consiste en añadir un término de iluminación constante, independiente de la posición o el ángulo. Esto que hace que parezca que siempre hay algo de luz dispersa por la escena, incluso cuando no hay una fuente de luz directa [\[15\]](#).

Hay métodos más complejos y más físicamente plausibles de simular la iluminación indirecta, como el *environment mapping* que discutiremos en más profundidad en la [Subsección 6.5](#).

2.4.3. Trazado de rayos

La distinción entre iluminación directa e indirecta no está basada en la física. La radiancia que llega a un punto porque ha sido emitida es de la misma naturaleza que la radiancia que llega porque ha sido reflejada desde otro punto. Esta distinción se hace para clasificar algoritmos simples y eficientes que tratan de simular tan solo un aspecto de la radiancia entrante. Los algoritmos que, por otro lado, resuelven la ecuación de renderizado ([Ecuación 4](#)) al completo se conocen como algoritmos de **iluminación global**, y el **trazado de rayos** es uno de ellos [\[16\]](#).

La ecuación de renderizado ([Ecuación 4](#)) es de naturaleza recursiva: la radiancia entrante en un punto es, a su vez, la radiancia saliente de otro punto en la escena. El trazado de rayos se aprovecha de este hecho. En vez de trazar rayos desde la luz y ver cuáles acaban en el observador, se trazan rayos desde la cámara hacia la escena, uno por cada píxel de la imagen [\[17\]](#). En la forma más simple del algoritmo, cuando un rayo intersecta con una superficie, esta emite a su vez otros tres rayos de manera recursiva [\[17\]](#):

- Un rayo en dirección de la fuente de luz. Si intersecta con algún otro objeto antes de llegar a la fuente de luz, la superficie está en sombra ($L_i = 0$).
- El rayo reflejado en la superficie.
- El rayo refractado en la superficie.

Aunque el trazado de rayos clásico fue un gran avance, su enfoque determinista no es la mejor manera de aproximar la integración de la radiancia entrante.

Una alternativa más robusta es la que emplea un tipo avanzado de algoritmo de trazado de rayos, el **path tracing**. Este utiliza un enfoque estocástico basado en la integración de Monte Carlo para resolver la ecuación de renderizado. De esta manera, se resuelve la integral empleando métodos estadísticos. En

lugar de lanzar rayos en direcciones predefinidas, en cada rebote se traza un único rayo en una dirección aleatoria, construyendo un “camino” a través de la escena [18].

La radiancia entrante se obtiene promediando el resultado de cientos o miles de estos caminos aleatorios que se originan en la cámara. Aunque produce un ruido característico con pocas muestras debido a su naturaleza aleatoria, este método es físicamente preciso y converge a la solución correcta a medida que aumenta el número de caminos [18].

2.5. Modelo de cámara

Anteriormente hemos expuesto que el renderizado en el paradigma *PBR* puede verse como calcular la radiancia saliente en dirección de la cámara. Pero, en última instancia, nosotros observamos la escena una vez procesada por la cámara. Entonces, cómo interprete la cámara la radiancia que le llega es igual de importante que la propia radiancia calculada.

Por tanto, un aspecto fundamental del *PBR* es simular una cámara de una manera precisa y fiel a cómo funcionan estos dispositivos en el mundo real.

En el mundo real, las cámaras fotográficas constan de una cámara oscura cerrada, con una abertura en uno de los extremos para que pueda entrar la luz, y una superficie plana de formación de la imagen o de visualización con un elemento fotosensible para capturar la luz en el otro extremo. La mayoría de las cámaras fotográficas tienen un objetivo formado de lentes, ubicado delante de la abertura de la cámara fotográfica para controlar la luz entrante y para enfocar la imagen, o parte de la imagen. El diámetro de esta abertura (conocido como **apertura**) suele poder modificarse [19].

Mientras que la apertura y el brillo de la escena controlan la cantidad de luz que entra por unidad de tiempo en la cámara durante el proceso fotográfico, el obturador controla el lapso en que la luz incide en la superficie de grabación. Por ejemplo, en situaciones con poca luz, la velocidad de obturación será menor (mayor tiempo abierto, es decir, mayor **tiempo de exposición**) para permitir que la película reciba la cantidad de luz necesaria para asegurar una **exposición** correcta [19].

Los elementos más relevantes de las cámaras para su simulación en gráficos son:

- **Distancia focal.** Marca la distancia desde el centro óptico del objetivo hasta el foco de la imagen. A mayor distancia focal del objetivo, se producen imágenes más grandes y, en consecuencia, menor campo de visión [20].
- **Apertura.** El diámetro de la abertura del objetivo. Para cuantificar esta abertura se emplea la relación focal o los *f-stops* N . El valor del diámetro es igual al valor de la distancia focal dividido entre N [20]. A menor *f-stops*, mayor apertura y por tanto mayor brillo o exposición, y viceversa [21].
- **Tiempo de exposición.** Denotado como t , controla el tiempo que el obturador permanece abierto [21]. A mayor tiempo de exposición, más tiempo le damos a la luz para que se concentre en el elemento fotosensible y más brillante saldrá la imagen [20].
- **Sensibilidad.** Los elementos fotosensibles tienen asociado un número denominado *ISO* que indica su sensibilidad a la luz. Cuanto más alto sea este número, mayor será la sensibilidad a la luz del elemento fotosensible [20].

3. Estado del arte

Una vez establecidos los fundamentos teóricos que rigen la simulación física de la luz, esta sección se adentra en su aplicación práctica a través de las herramientas y tecnologías que definen el estado del arte del *PBR* en informática gráfica. El desafío computacional de resolver la ecuación de renderizado ([Ecuación 4](#)) ha impulsado el desarrollo de motores de renderizado altamente sofisticados, cuyo objetivo es implementar los principios de la radiometría, la óptica geométrica y física de la interacción luz-superficie para generar imágenes de alta fidelidad.

La filosofía del *PBR* se manifiesta entre otras cosas, en:

- Motores basados en el *path tracing*, que permiten simular con gran precisión fenómenos complejos como reflexiones especulares múltiples, caústicas, refracción, sombras suaves y dispersión global de la luz, proporcionando un nivel de realismo visual que aproxima lo que captaría una cámara en un entorno físico real.
- Materiales basados en la física (construidos mediante modelos *BRDF* o modelos más generales) y muy versátiles. Permiten representar con fidelidad superficies complejas como metales pulidos, pinturas multicapa, plásticos translúcidos o vidrios esmerilados, todo ello con un conjunto reducido de parámetros intuitivos.
- La capacidad de definir fuentes de luz y cámaras mediante unidades radiométricas, fotométricas y ópticas del mundo real, como radiancia, lúmenes, apertura o distancia focal. Este es un aspecto crucial para la simulación precisa de entornos donde se dispone de datos cuantificables del sistema de iluminación y captura.

En este contexto, se procederá a analizar varias tecnologías representativas que encarnan el estado actual de la industria.

3.1. NVIDIA Iray

NVIDIA Iray es una tecnología de renderizado intuitiva con base física que genera imágenes fotorealistas para flujos de trabajo de renderizado interactivos y por lotes. Hace uso de reducción de ruido potenciada por IA, el motor de trazado de rayos *NVIDIA OptiX*, el lenguaje de definición de materiales *MDL* y la plataforma de programación de GPUs de *NVIDIA*, CUDA [\[22\]](#).

Entre sus funcionalidades se incluyen geometría emisora, luces fotométricas, materiales intuitivos medidos y realistas con *MDL*, profundidad de campo, creación de instancias y la posibilidad de visualizar la distribución de luminancia de una escena [\[22\]](#). *Iray* permite una exploración visual interactiva de la escena, permitiendo previsualizaciones progresivas de alta calidad gracias a su enfoque de rendering incremental [\[23\]](#).

Iray es un motor de renderizado con algoritmos de transporte de luz líderes en la industria que generan resultados realistas rápidamente [\[24\]](#). En cuanto a su arquitectura, destaca por un rendering espectral real [\[24\]](#) (es decir, opera sobre longitudes de onda individuales en lugar de modelos RGB simplificados), lo cual permite simular efectos complejos como la dispersión cromática o la absorción dependiente del espectro [\[25\]](#).

En 2016 *NVIDIA* validó *Iray* con el informe “CIE 171:2006” de la *Commission Internationale de l’Eclairage (CIE)* [\[26\]](#). Este se trata de un conjunto de casos de prueba desarrollados por la comisión para ayudar a los desarrolladores de motores de iluminación a determinar la precisión de sus resultados e identificar las debilidades de su solución [\[27\]](#).

Los resultados obtenidos por *Iray* entraban cómodamente dentro de los márgenes de error establecidos por la CIE, y en los pocos casos en los que el error era superior al esperado, se determinó que era por un diseño del caso de prueba erróneo [\[26\]](#). Por lo tanto, es un motor físicamente correcto.

Además, *Iray* destaca por su soporte nativo para *MDL*, un estándar creado por el propio *NVIDIA* que permite definir materiales de forma agnóstica al motor de renderizado. Esto facilita la creación de materiales físicamente plausibles y portables [\[28\]](#), o permite utilizar materiales de una rica biblioteca de materiales disponibles en la web con *vMaterials* [\[22\]](#).

También, con *MDL* se pueden crear materiales medidos, que representan la medición de superficies del mundo real y se consideran cada vez más una aportación fundamental a las decisiones de diseño y fabricación de productos [\[23\]](#).

En suma, permite la configuración de fuentes de luz y cámaras utilizando unidades fotométricas y ópticas del mundo real (lúmenes, candelas, temperatura de color, distancia focal, *f-stop*) [\[24\]](#), lo cual es crucial para replicar con fidelidad las condiciones de una escena real donde se conocen datos de iluminación y cámaras.

3.2. Unreal Engine 5

Unreal Engine es un motor de gráfico desarrollado por *Epic Games*. Fue desarrollado inicialmente para juegos de disparos en primera persona para PC, pero desde entonces se ha utilizado en diversos géneros de juegos y ha sido adoptado por otras industrias, sobre todo la del cine y la televisión. *Unreal Engine* está escrito en C++ y presenta un alto grado de portabilidad, siendo compatible con una amplia gama de plataformas de sobremesa, móviles, consolas y realidad virtual [29].

Unreal Engine 5 o *UE5* es la última versión de *Unreal Engine*. Fue lanzada en abril de 2022 y, según su desarrollador, es la herramienta de creación 3D en tiempo real más abierta y avanzada del mundo [30].

UE5 posee un motor de iluminación potente llamado *Lumen*. Se trata del sistema de iluminación global y reflejos totalmente dinámico de *UE5*. *Lumen* renderiza interreflexiones difusas con rebotes infinitos y reflejos especulares indirectos en entornos grandes y detallados a escalas que van desde milímetros a kilómetros [31].

Lumen también dispone de materiales emisivos que propagan la luz sin coste adicional de rendimiento. Sin embargo, hay un límite a lo pequeñas y brillantes que pueden ser las áreas emisivas antes de que empiecen a aparecer artefactos de ruido. Además, resuelve rebotes especulares indirectos, o reflexiones, para toda la gama de valores de rugosidad del material [31]. Ambos aspectos importantes dado el contexto de nuestro proyecto.

Gráficamente, *UE5* se apoya en su tecnología de virtualización de geometría llamada *Nanite*, que permite renderizar escenas con billones de triángulos sin una penalización significativa en el rendimiento, eliminando la necesidad de “lods” o mallas de nivel de detalle [32]. Esto mejora la precisión geométrica, lo cual es crucial cuando se combinan materiales físicamente precisos con una geometría detallada.

También, *UE5* permite trabajar con magnitudes del mundo real tanto para luces [33], como para cámaras con *Cine Camera Actors*, que tienen parámetros de distancia focal, *f-stop*, postprocesado de exposición, etc [34].

Aparte, soporta materiales basados en física que funcionan igual de bien en todos los entornos de iluminación. Además, los valores de los materiales pueden ser menos complejos e interdependientes, lo que facilita el flujo de trabajo de creación de materiales [35].

Estos materiales se configuran con cuatro parámetros: color base, rugosidad, metalicidad y reflectividad especular [35]. Además, ofrece soporte a materiales con varias capas [36], que resulta importante para simular los acabados de la pintura de automóviles. Ofrece dos formas principales de superponer materiales y crear mezclas complejas entre tipos de superficie únicos, *layered materials* y *material layers*. Estos flujos de trabajo permiten aplicar distintas propiedades de material a diferentes regiones de una misma malla [36].

3.3. V-Ray

V-Ray, desarrollado por *Chaos Group*, es un motor de renderizado basado en la física muy consolidado y utilizado en arquitectura, diseño de interiores, el diseño de productos y los efectos visuales para cine y televisión [37]. *V-Ray* funciona principalmente como un *plugin* que se integra de forma nativa en las principales aplicaciones de creación de contenido digital, como 3ds Max, Maya, SketchUp y Cinema 4D [38], dotándolas de capacidades de renderizado fotorealista avanzadas.

El núcleo de renderizado de *V-Ray* tiene varias soluciones para conseguir iluminación global, incluso permitiendo configurar distintos motores para los rebotes primarios y secundarios de los rayos. Estos algoritmos son la caché de luz, fuerza bruta, mapeo de fotones y mapa de irradiancia [39]:

- La caché de luz, también conocida como *light cache* o *light mapping* es una técnica desarrollada por *Chaos* específicamente para *V-Ray* para aproximar rápidamente la iluminación global de una escena. Para obtener la iluminación del entorno, *V-Ray* traza muchos rayos desde la cámara hacia la escena. Cada rebote de luz crea una muestra en la caché de luz que puede reutilizarse durante el renderizado. Si un rayo choca con una muestra creada por otro rayo, el proceso de cálculo se detiene y en su lugar se lee la información de la muestra. La capacidad de detectar y leer la información de iluminación creada por otros rayos (en lugar de calcular nueva información) acelera mucho el proceso [40].
- El método de fuerza bruta para calcular la iluminación global vuelve a calcular los valores de iluminación para cada punto renderizado, por separado e independientemente de otros puntos. Este método es muy preciso, especialmente si hay muchos detalles pequeños en la escena [41].
- El mapeo de fotones se emplea exclusivamente para simular las cáusticas, patrones de luz concentrada (reflejos y refracciones intensas) que se forman al atravesar o reflejar superficies curvas. Se trata de una técnica de dos pasadas. La primera consiste en disparar partículas (fotones) desde las

fuentes de luz de la escena, rastreárlas a medida que rebotan por la escena y registrar los lugares en los que los fotones golpean las superficies de los objetos. La segunda pasada es el renderizado final, que es cuando se calculan las cáusticas utilizando técnicas de estimación de densidad sobre los impactos de fotones almacenados durante la primera pasada [42].

- El mapa de irradiancia está en desuso [39], por lo que omitiremos su descripción.

El sistema de materiales de *V-Ray*, está centrado en el versátil *VRayMtl*. Se trata de un material que permite una iluminación físicamente más correcta (distribución de energía) en la escena, un renderizado más rápido y unos parámetros de reflexión y refracción más convenientes. Este material puede simular una enorme variedad de superficies, desde plásticos a metales, vidrio y más con un puñado de ajustes en los parámetros. Además, con *VRayMtl* se puede aplicar diferentes texturas, controlar las reflexiones y refracciones, añadir mapas de relieve y desplazamiento, forzar cálculos directos de iluminación global, y elegir el *BRDF* para la forma en que la luz interactúa con el material de la superficie [43].

Además, también tiene varios *BRDFs* aparte del *VRayMtl* para simular materiales específicos, como pelo, un material con dispersión subsuperficial global o pintura de coches [39].

Al igual que las otras tecnologías analizadas, *V-Ray* fundamenta su realismo en el uso de unidades y magnitudes del mundo real, tanto para cámaras como para luces. La *VRayPhysicalCamera* utiliza ajustes de una cámara del mundo real, tales como *f-stop*, distancia focal, y la velocidad de obturación para configurar la cámara virtual [44].

Del mismo modo, sus fuentes de luz, incluyendo las luces definidas mediante una malla, permiten que se defina su intensidad lumínica mediante unidades como lumens, luminancia, potencia radiante o radiancia [45].

La robustez y precisión de *V-Ray* le han valido un amplio reconocimiento en la industria del cine y televisión. Se emplea como tecnología de rendering en muchas producciones cinematográficas, como *Juego de Tronos*, *Stranger Things* [46], *Doctor Strange*, *Capitán América: Civil War* [47], etc. De hecho, su papel ha sido tal que esta herramienta ha ganado dos premios, un Oscar en 2017 por el “concepto original, diseño e implementación de *V-Ray*” [47], y un Emmy en 2021 por avanzar en la industria el uso de rendering completamente basado en trazado de rayos [46].

3.4. AMD Radeon ProRender

Radeon ProRender o *RPR* es un potente motor de renderización por trazado de rayos con base física que permite que los profesionales creativos generen imágenes fotorrealistas de alta fidelidad [48].

Presenta una potente combinación de capacidades de renderizado, rapidez y compatibilidad entre plataformas. Utiliza *HIP*, *OpenCL* o *Metal*, funciona con *Windows*, *Linux*, *macOS*, y es compatible con las GPU y CPU de *AMD*, así como con las de otros proveedores [48].

Además, forma parte de la suite *AMD GPUOpen*, un paquete de *middleware* que ofrece efectos visuales avanzados para videojuegos. Engloba como componentes principales varias tecnologías gráficas diferentes que antes eran independientes y estaban separadas entre sí. Sin embargo, destaca porque se comercializa bajo una licencia de software permisiva, es decir, que es parcialmente código abierto [49].

Como ya se ha comentado, *RPR* es un renderizador basado en la física. Utiliza algoritmos de *path tracing* para crear renders precisos de iluminación y materiales. Además cuenta con un amplio sistema de materiales con base física nativo [48].

Estos materiales comprenden un abanico de nodos o *shaders* que se pueden combinar para cubrir distintas necesidades, como materiales emisivos, o transparentes [50]. O también se puede recurrir al *Uber Shader*, un *shader* multiusos que combina diferentes funcionalidades con sus distintas capas [51].

Asimismo, también tiene soporte a materiales definidos mediante *MaterialX* [52], un estándar abierto para representar el aspecto de materiales, lo que permite su descripción e integración entre aplicaciones y renderizadores, independientemente de la plataforma [53]. *GPUOpen* cuenta con una extensa biblioteca de materiales definidos mediante este estándar que los usuarios pueden emplear en sus aplicaciones [54].

Desde el punto de vista visual, *RPR* ofrece soporte nativo para efectos como *motion blur*, profundidad de campo, dispersión subsuperficial global y volúmenes participativos (niebla y humo), todos basados en principios físicos y con una configuración directa desde las propiedades del sistema [55].

También tiene un sistema de cámaras realista, con la capacidad de simular profundidad de campo, distancia focal, distancia de enfoque, *motion blur*, entre otras cosas [56]. Y además, cuenta con la posibilidad de definir la potencia de las fuentes mediante magnitudes radiométricas [57].

4. Análisis del problema

Tras haber examinado el estado del arte en renderizado fotorrealista, ahora centramos el análisis en las particularidades del problema que aborda este Trabajo de Fin de Grado. La escena a simular, el túnel de inspección, no es un entorno genérico. Su configuración de iluminación y la disposición de sus cámaras están diseñadas explícitamente para el propósito de detección de imperfecciones en la superficie de carrocerías. Esta naturaleza tan específica, orientada a un caso de uso industrial muy concreto, exige una tecnología que no solo ofrezca fotorrealismo, sino que también permita replicar con exactitud estas condiciones.

Como se ha establecido anteriormente, el fotorrealismo es el principal objetivo sobre el que se sustenta la viabilidad del proyecto. Las imágenes sintéticas deben ser visualmente indistinguibles de las capturas reales para que el conjunto de datos generado sea válido para el entrenamiento del modelo de visión por computador.

Ante este requisito, se podría plantear el desarrollo de un motor de renderizado propietario desde cero. Sin embargo, descartamos esta opción categóricamente. La creación de un motor de renderizado puntero, capaz de competir con las soluciones actuales del estado del arte, es una tarea de tal complejidad y envergadura que excede con creces el alcance y los plazos de un Trabajo de Fin de Grado, requiriendo equipos de desarrollo especializados y años de trabajo.

Por tanto, la estrategia más pragmática y eficiente es emplear un motor de renderizado ya existente como la base de nuestra solución. De esta manera, aseguramos la corrección del renderizado basado en física. Y, al mismo tiempo, en lugar de invertir tiempo en el monumental cometido de desarrollar un *path tracer* de última generación, nos dedicamos a la comparativamente sencilla tarea de adaptar tecnología ya probada y validada a nuestro caso de uso altamente específico.

Para llevar a cabo una evaluación rigurosa, establecemos una serie de requisitos clave que se alinean con los objetivos propuestos y que, por tanto, una solución candidata debe cumplir.

- **Motor de renderizado basado en la física.** Se requiere un motor que implemente un modelo de *PBR* para garantizar el fotorrealismo. Esto permite modelar la interacción de la luz con los materiales de forma físicamente plausible, haciendo uso de magnitudes reales conocidas del túnel de inspección, como la intensidad lumínica o la configuración de las cámaras.
- **Disponibilidad de renderizado en tiempo real y *offline*.** La solución debe ofrecer una dualidad de modos: un renderizado en tiempo real para obtener retroalimentación visual inmediata durante el ajuste interactivo de parámetros de la escena, y un renderizado *offline* de alta fidelidad para la generación de las imágenes finales con la máxima calidad.
- **Facilidad de integración e interoperatividad.** La tecnología debe presentarse como una librería o *SDK*, y no como una plataforma monolítica, para facilitar su integración en el simulador de la empresa. La interoperatividad con *OpenGL* es crucial, dada que ésta es la *API* gráfica que emplea el simulador.
- **Abundancia de documentación y accesibilidad.** Dada la naturaleza y el plazo de un Trabajo de Fin de Grado, es indispensable que la *API* esté bien documentada y sea accesible. Una curva de aprendizaje suave es clave para poder centrarse en la resolución del problema en lugar de en descifrar la herramienta.
- **Licencia de código abierto o gratuita.** Para eliminar las barreras económicas de las soluciones comerciales y asegurar la viabilidad del proyecto tanto a nivel académico como en una futura implementación industrial, la herramienta seleccionada debe ser gratuita.

A continuación, analizaremos las tecnologías discutidas en nuestra exploración del estado del arte del *PBR* en informática gráfica ([Sección 3](#)) para encontrar aquella que mejor se adapte a estos requisitos.

Como conjunto, todas estas soluciones expuestas previamente en nuestro análisis de la industria son tecnología punta de renderizado fotorrealista. Por lo tanto, podemos concluir que todas cumplen el primero de los requisitos propuestos (que la tecnología sea un motor de *PBR*). No es necesario analizar las tecnologías una a una en el caso de este requisito. Nos centraremos, por consiguiente, en los otros cuatro requisitos.

4.1. NVIDIA Iray

4.1.1. Disponibilidad de renderizado en tiempo real y *offline*

Iray ofrece múltiples modos de renderizado que abordan un abanico de casos de uso, que requieren desde retroalimentación interactiva y en tiempo real, hasta visualizaciones fotorrealistas basadas en la

física y simulaciones de iluminación [58]. Admite combinar distintas configuraciones del mismo modo de renderizado, o dos modos de renderizado para permitir previsualizaciones rápidas durante la navegación e imágenes finales de alta calidad [23]. Estos modos son:

- *Photoreal*, un *path tracer* centrado en generar imágenes fotorrealistas con sólo pulsar un botón. Es una opción ideal para cuando se desean efectos complejos que conllevan iluminación global [23].
- *Interactive*, es un modo de renderizado de trazado de rayos interactivo que utiliza algoritmos de renderizado más rápidos pero menos precisos que *Iray Photoreal*. *Iray Interactive* ofrece un aspecto coherente con el resultado físico de *Iray Photoreal*, pero está optimizado para la manipulación interactiva de escenas. Es la mejor opción cuando se buscan efectos de trazado de rayos, como reflejos y refracción, y se puede aceptar un fotorrealismo limitado [59].

Utiliza renderizado progresivo para permitir una retroalimentación rápida. La llamada inicial genera un fotograma de baja calidad. Las siguientes llamadas de renderizado refinan la imagen de forma incremental. Cada fotograma generado por una operación de renderizado progresivo se exporta a un archivo. La visualización de todos los archivos en secuencia muestra el refinamiento incremental de la imagen [23].

4.1.2. Facilidad de integración e interoperatividad

Iray no se distribuye como una aplicación monolítica, sino como un *SDK* con una *API* de C++ para una integración sencilla en aplicaciones 3D y/o la creación de potentes aplicaciones cliente/servidor. Esta *API*, diseñada para ofrecer eficacia y facilidad de uso, proporciona un único punto de acceso a las librerías de software en C++, que pueden cargarse y vincularse dinámicamente a aplicaciones de visualización en tiempo de ejecución para integrar la tecnología de renderizado.

No se ha podido encontrar información específica sobre interoperatividad entre *Iray* y *OpenGL* dada la licencia de la tecnología, pero dado que la documentación oficial señala la facilidad de integración en cualquier aplicación 3D, y que muchas aplicaciones 3D se basan en *OpenGL*, suponemos que será posible integrarla en el simulador.

4.1.3. Abundancia de documentación y accesibilidad

Aparte del manual del programador que se puede encontrar *online* [59], en el paquete que se obtiene con la licencia se proporciona documentación detallada, no solo de *Iray*, sino también de *MDL*, y ejemplos de código. Además, también se incluye una aplicación de ejemplo de visualización en la que se integra *Iray* [23].

En cuanto a la accesibilidad, *NVIDIA* confía en su facilidad de integración [58], por lo que suponemos que la dificultad recae en entender los conceptos de informática gráfica subyacentes, al igual, que cualquier otro motor de rendering avanzado.

4.1.4. Licencia de código abierto o gratuita

Por último, el *SDK* de *Iray* es un producto comercial. Esta disponible a través de los socios de *NVIDIA*, *Siemens Lightworks* y *Migenius RealityServer* [58]. Los costes asociados están orientados al mercado empresarial y resultan prohibitivos para un proyecto de ámbito académico. No existe una versión comunitaria, gratuita o de código abierto que permita su libre utilización.

4.1.5. Conclusión

A pesar de sus notables capacidades técnicas, *Iray* no cumple con todos los requisitos indispensables establecidos para el proyecto.

En el plano técnico, *Iray* demuestra ser una herramienta excepcionalmente potente. Cumple con el requisito fundamental de ser un motor de renderizado basado en la física (PBR). Además, su dualidad de modos, con un renderizado en tiempo real (*Interactive*) y otro de alta fidelidad *offline* (*Photoreal*), se alinea perfectamente con la necesidad del proyecto de obtener retroalimentación inmediata y, a la vez, generar imágenes finales de máxima calidad.

En cuanto a la integración, su formato como *SDK* con una *API* de C++ satisface la necesidad de una librería integrable. No obstante, surge una incertidumbre considerable en lo que respecta a la interoperatividad con *OpenGL*, un punto crucial para la integración con el simulador. Por otro lado, la documentación parece ser exhaustiva y accesible, cumpliendo con este requisito.

Sin embargo, el factor decisivo que invalida la candidatura de *Iray* es su modelo de licenciamiento. Al ser un producto comercial con costes prohibitivos para un ámbito académico, choca frontalmente con el

requisito de ser una solución gratuita o de código abierto. Esta barrera económica es insalvable y hace que, a pesar de sus fortalezas, *Iray no sea una opción factible* para el desarrollo de este Trabajo de Fin de Grado.

4.2. Unreal Engine 5

4.2.1. Disponibilidad de renderizado en tiempo real y *offline*

UE5 está optimizado para renderizado en tiempo real, aprovechando tecnologías como *Nanite* (sistema de geometría virtualizada que emplea un nuevo formato de malla interna y tecnología de renderizado que permite obtener detalles a escala de píxel y una gran cantidad de objetos [32]) y el *Lumen* ya mencionado, para iluminación global y detalles geométricos ilimitados.

Pero también ofrece la posibilidad de romper con las limitaciones de renderizado en tiempo real, y obtener imágenes de aún mayor fidelidad mediante renderizado *offline* con *Movie Render Queue*. Provee la oportunidad de utilizar ajustes y comandos que aumentan enormemente la calidad, la precisión y el aspecto de funciones como iluminación global por trazado de rayos y reflejos por trazado de rayos. También se puede mejorar el desenfoque de movimiento y eliminar artefactos de *aliasing* no deseados [60].

4.2.2. Facilidad de integración e interoperatividad

Unreal Engine es una aplicación monolítica que viene con su propio editor. Dado que su código fuente se encuentra subido en *Github* [61], sería técnicamente posible escoger ciertos componentes del motor, como su motor de rendering, e integrarlos con el simulador, pero sería tarea fuera del alcance de un Trabajo de Fin de Grado.

4.2.3. Abundancia de documentación y accesibilidad

También, el motor cuenta con documentación muy completa. Desde cientos de páginas de documentación oficial explicando la abundante cantidad de componentes y sistemas de *UE5* [62], hasta cientos de proyectos de ejemplo [30] y una biblioteca de vídeos de formación, muchos publicados por el propio *Epic Games* [63].

Pero, la inmensa popularidad de *Unreal Engine* supone una ventaja también para el motor en este aspecto. Se trata del motor de videojuegos comercial más grande en cuanto a ventas, con un 31% de los juegos vendidos en 2025 corriendo sobre *Unreal Engine*. Y es el segundo más grande en cuanto a videojuegos lanzados. Aquellos que emplean *UE5* como motor representan el 28% del número total de juegos publicados en 2025 [64].

Esto conlleva una gran comunidad de desarrolladores que se ayudan unos a otros. Publicando tutoriales en forma de vídeos en *Youtube*, donde hay canales con cientos de miles de suscriptores dedicados exclusivamente a subir contenido sobre el motor [65]. O creando *posts* y respondiendo preguntas en los activos foros de *Epic Games* [66].

4.2.4. Licencia de código abierto o gratuita

Finalmente, *UE5* es gratuito para descarga y uso académico, pero su modelo de licencia no es de código abierto bajo licencias permisivas. Está sujeto a la *End User License Agreement* de *Epic Games*, que exige el pago de un 5% de royalties sobre ingresos por encima de un millón de USD generados directamente por productos que utilicen *Unreal Engine*, con ciertas excepciones (por ejemplo, ventas en *Epic Games Store*) [67].

4.2.5. Conclusión

Unreal Engine 5, a pesar de ser una de las herramientas más avanzadas y completas de la industria, tampoco resulta una solución adecuada para los objetivos del proyecto debido a una incompatibilidad fundamental en su arquitectura.

Técnicamente, *UE5* cumple con creces varios de los requisitos clave. Ofrece un sistema de renderizado *PBR* de última generación y satisface la dualidad de renderizado en tiempo real y *offline* gracias a tecnologías como *Lumen* y *Movie Render Queue*. Además, su documentación es excepcionalmente extensa y está respaldada por una de las comunidades de desarrolladores más grandes y activas del mundo, lo que garantiza una accesibilidad y una curva de aprendizaje muy favorables. Su modelo de licencia, gratuito para uso académico, también elimina las barreras económicas inmediatas para el desarrollo del proyecto, aunque podría suponer un problema a largo plazo.

Sin embargo, el requisito que invalida por completo la viabilidad de *Unreal Engine 5* como solución es su falta de facilidad de integración. El motor está diseñado como una plataforma monolítica y no como una librería o *SDK* que pueda ser incorporada de forma sencilla en una aplicación externa. Como se indica en el análisis, aunque su código fuente está disponible, la tarea de extraer su motor de renderizado para integrarlo en el simulador de la empresa es de una complejidad que excede con creces el ámbito y los plazos de un Trabajo de Fin de Grado.

Por lo tanto, su naturaleza como ecosistema cerrado, en lugar de un componente modular, lo convierte en una **opción inviable** para las necesidades de integración específicas del simulador.

4.3. V-Ray

4.3.1. Disponibilidad de renderizado en tiempo real y *offline*

V-Ray ofrece dos modos de render distintos: *Production* y *Interactive* [68]:

- El modo *Production* es adecuado para obtener la imagen final deseada después de que la escena haya sido cuidadosamente configurada, ya que el proceso de render puede llevar mucho tiempo antes de que la imagen completa esté disponible [68].
- El modo *Interactive*, por su parte, es muy útil para previsualizar rápidamente la imagen y mejorar progresivamente su calidad, lo que lo convierte en la opción perfecta para experimentar con la configuración de la escena y modificar el contenido, así como para recibir rápidamente información sobre el renderizado [68].

Cada modo de renderizado está disponible en dos versiones, CPU y GPU. El primero utiliza los recursos de la CPU, mientras que el segundo aprovecha la potencia de cálculo de la tarjeta gráfica. El tipo de renderizado GPU se subdivide a su vez en dos modos, *CUDA* y *OptiX* (ambos sólo para tarjetas *NVIDIA*) [68].

4.3.2. Facilidad de integración e interoperatividad

Aparte de como añadido a las aplicaciones de generación de contenido digital más populares, *V-Ray* está disponible en forma de *SDK* mediante el *V-Ray Application SDK*. Este permite a terceros iniciar y controlar un proceso de renderizado que haga uso del motor V-Ray. Proporciona una *API* de alto nivel que permite a los usuarios renderizar y manipular una escena *V-Ray* dentro del proceso anfitrión o fuera de él utilizando el renderizado distribuido [68].

El *SDK* ofrece soporte a C++, C#, *Python* y *Node.js* [68]. También permite acceder y manipular la imagen renderizada mediante la clase *VRayImage* [68], lo que asegura la posibilidad de integración en el simulador de *OpenGL*.

4.3.3. Abundancia de documentación y accesibilidad

V-Ray cuenta varios cursos de aprendizaje, pero estos son para que artistas aprendan a usar el *plugin* de *V-Ray* de distintas aplicaciones de creación de contenido digital [69], por lo que no es relevante en nuestro caso.

En el paquete del *SDK* viene documentación extensa de la *API* (que también se encuentra *online*), escenas ya creadas por el equipo de desarrollo, y ejemplos en todos los lenguajes soportados que ilustran las herramientas del motor [68].

Sin embargo, en cuanto a la accesibilidad, dada una lectura rápida de la documentación que se puede encontrar en la web de dicha *API*, concluimos que parece una herramienta más compleja que *Iray* y *RPR* (que analizaremos más tarde).

4.3.4. Licencia de código abierto o gratuita

V-Ray cuenta con varios planes comerciales de menor a mayor precio, con los planes más caros permitiendo acceso a funcionalidad más avanzadas de la herramienta, como renderizado en la nube. También cuenta con un plan educativo por un precio reducido [37]. Sin embargo, la única manera de acceder al motor gratuitamente es mediante un período de prueba de 30 días [37], que no es sostenible a largo plazo, obviamente.

4.3.5. Conclusión

A primera vista, *V-Ray* se presenta como un candidato a solución excepcionalmente fuerte, alineándose de forma notable con varios de los requisitos técnicos más importantes del proyecto.

Su prestigio en la industria, avalado por premios Oscar y Emmy, es un testimonio de la robustez de su motor de renderizado. Cumple con creces el requisito de fotorrealismo, fundamentando su plausibilidad física en el uso de magnitudes del mundo real para cámaras y luces, y en algoritmos avanzados de iluminación global. Desde una perspectiva técnica, su dualidad de modos, *Production* e *Interactive*, satisface a la perfección la necesidad de renderizado en tiempo real y *offline*. También, su arquitectura modular, ofrecida a través del *V-Ray Application SDK*, representa una buena solución para la integración en el simulador de *AUTIS*.

No obstante, la viabilidad de *V-Ray* se ve comprometida por dos factores. En primer lugar, su curva de aprendizaje parece ser más pronunciada en comparación con otras *API*, lo que supone un riesgo para los plazos ajustados de un Trabajo de Fin de Grado. Pero el mayor obstáculo es su modelo de licencia. El requisito de una licencia de código abierto o gratuita es clave para asegurar la viabilidad del proyecto. *V-Ray* es un producto puramente comercial, y su única opción gratuita, un período de prueba de 30 días, es insuficiente para el desarrollo y la futura implementación del proyecto.

Por lo tanto, a pesar de su excepcional motor de renderizado fotorrealista y su idónea arquitectura basada en un *SDK*, su modelo de licencia comercial, al igual que *Iray*, lo convierte en una **opción inviable** para las necesidades y restricciones del proyecto.

4.4. AMD Radeon ProRender

4.4.1. Disponibilidad de renderizado en tiempo real y *offline*

RPR, al igual que otros renderizadores, también soporta dos modos de renderizado, en tiempo real y *offline*, mediante dos *backends* distintos:

- *Northstar* para renderizado *offline* [70]. Soporta todo el espectro de materiales, algoritmos de iluminación y funcionalidades del motor y proporciona las imágenes de mayor fidelidad y fotorrealismo.
- *HybridPro*, por otro lado, se emplea para renderizado en tiempo real [70], que permite darle retroalimentación de manera rápida y eficaz al usuario. Sin embargo, para ello se han de hacer concesiones. Este *backend* solo soporta los materiales del *Uber Shader* y los renders obtenidos son de menor calidad que los de *Northstar* [71].

4.4.2. Facilidad de integración e interoperatividad

Radeon ProRender se ofrece como un *SDK* modular y está diseñado para ser integrado en aplicaciones de terceros [55]. El *SDK* expone una *API* en C que puede ser usada desde programas en C, C++, C# y también tiene *bindings* para *Python* [72]. Como se puede observar por las funciones disponibles de la *API* en la documentación, esta permite un control completo de la escena, cámaras, iluminación y modelos.

Dicho *SDK* es una librería dinámica o *DLL*. Para integrarlo simplemente se deben incluir en el proyecto los archivos de librería propios del sistema operativo sobre el que se este trabajando y los archivos de cabecera correspondientes al lenguaje que se este empleando [72].

4.4.3. Abundancia de documentación y accesibilidad

Esta documentación mencionada anteriormente es extensa, incluyendo todas las funciones de la *API*, tipos de datos, *shaders*, etc. La documentación también presenta un tutorial de interoperatividad entre *RPR* y *OpenGL*, donde se emplea la *API* de *RPR* para bajarse el fotograma generado y subirlo a una textura de *OpenGL* [73]. Además de ese tutorial, en el paquete del *SDK* también se incluyen más proyectos de ejemplo que funcionan a modo de tutorial de la mayoría de funciones disponibles en el motor.

La accesibilidad de esta tecnología se estima muy similar a la de su competidor, *Iray*, por las mismas razones.

4.4.4. Licencia de código abierto o gratuita

Por último, el punto más destacado de *Radeon ProRender*. El kit de desarrollo de software completo y las bibliotecas asociadas se distribuyen bajo la licencia *Apache 2.0* [72]. Esta se trata de una licencia de software libre permisiva, que permite el uso, modificación y distribución del software de forma gratuita, tanto para proyectos académicos como comerciales [74].

4.4.5. Conclusión

A diferencia de las alternativas anteriores, *RPR* se perfila como la candidata ideal, ya que cumple satisfactoriamente con la totalidad de los requisitos establecidos para el proyecto.

En el aspecto técnico, su motor PBR y la dualidad de renderizado mediante los backends *Northstar (offline)* y *HybridPro* (tiempo real) se ajustan perfectamente a las necesidades de fidelidad y retroalimentación interactiva. Su arquitectura como *SDK* modular con una *API* en C garantiza una integración directa y sencilla en el simulador, un punto en el que *Unreal Engine* resultaba inviable.

Además, la existencia de un tutorial específico de interoperatividad con *OpenGL* despeja cualquier duda sobre su compatibilidad, una incertidumbre presente en el análisis de *Iray*. La documentación se presenta como completa y accesible, y su curva de aprendizaje se estima manejable para los plazos del proyecto.

Finalmente, el factor que consolida su idoneidad es su licencia *Apache 2.0*. Al ser de código abierto y totalmente gratuita, elimina por completo las barreras económicas de las soluciones comerciales como *Iray* y asegura la viabilidad del proyecto tanto en su fase académica como en una posible aplicación industrial futura.

Por todo lo expuesto, *Radeon ProRender* no solo cumple con cada uno de los criterios, sino que destaca en los aspectos más críticos de integración y coste, convirtiéndola en **la tecnología seleccionada** para llevar a cabo el proyecto.

5. Solución propuesta

Una vez justificada la selección de *Radeon ProRender* como la tecnología de renderizado más idónea, esta sección se centra en introducir la solución implementada. La solución propuesta no consiste en un uso directo y disperso del *SDK* de *RPR*, sino en la construcción de una capa de abstracción sobre él. Nuestra solución es una librería de software diseñada para actuar como un puente entre la potencia genérica del motor de renderizado y las necesidades altamente especializadas de la empresa.

Se busca crear una interfaz de alto nivel que oculte la inherente complejidad de un motor de renderizado basado en la física. Así, abstraemos operaciones complejas como la creación de contextos, la gestión de la memoria de la GPU, la configuración de nodos de materiales o la sincronización de *framebuffers*. En su lugar, la interfaz expone un conjunto de funciones y estructuras de datos intuitivas, cuya semántica se alinea con el dominio del problema: superficies a inspeccionar, arcos de luz, cámaras industriales y propiedades de materiales visualmente reconocibles. El objetivo es que el equipo de desarrollo del simulador de *AUTIS*, sin necesidad de poseer conocimientos profundos en la teoría del *PBR*, pueda integrar esta interfaz en su *software*.

Esta librería se desarrollará en C++. La elección de este lenguaje está soportada por tres razones:

- Facilita la integración en el simulador existente. Este también está implementado en C++. Por consiguiente, de esta manera permitimos una integración nativa y evitamos problemas de interoperabilidad.
- C++ se diseñó pensando en la programación de sistemas, el software embebido con recursos limitados y los grandes sistemas, con el rendimiento, la eficiencia y la flexibilidad de uso como puntos fuertes de su diseño [75]. Por ello, es el estándar en la computación gráfica, donde la eficiencia y el rendimiento son claves.
- El propio *SDK* de *Radeon ProRender* ofrece una *API* nativa en C, lo que convierte a C++ en el lenguaje natural para interactuar con el motor base de la forma más directa y eficiente posible.

5.1. Tecnologías empleadas

A continuación listaremos y discutiremos los distintos entornos, *SDKs* y librerías utilizadas para desarrollar la solución.

5.1.1. Radeon ProRender

Como hemos discutido en profundidad anteriormente, nuestra solución se basa en *RPR*. Empleamos su *SDK* y todas las *DLL* que vienen incluidas con él.

5.1.2. Visual Studio 2022

Microsoft Visual Studio es un entorno de desarrollo integrado (IDE, por sus siglas en inglés) para *Windows* y *macOS*, compatible con la mayoría lenguajes de programación más populares, como C++, C#, Java y Python, entre otros. *Visual Studio 2022* es la última versión de este editor [76].

La selección de este editor como el *IDE* de elección para el desarrollo de este proyecto se tomó por mi familiaridad con el mismo, al haber aprendido su uso durante mis estudios y haberlo empleado para varios proyectos personales en el pasado.

5.1.3. OpenGL

OpenGL es una especificación estándar que define una *API* multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D [77]. Es una de las principales cuatro *APIs* de gráficos, junto con *Vulkan*, *Direct3D* y *Metal*.

En nuestra solución empleamos esta *API* para postprocesar la imagen generada de una manera más eficiente, aprovechando las capacidades de la GPU (véase la [Subsección 6.7](#)). Seleccionamos *OpenGL*, y no otra *API*, para esta tarea por tres razones que detallamos seguidamente.

- Dado que *Radeon ProRender* es una solución multiplataforma, es deseable que los componentes subsecuentes mantengan esta misma característica. Emplear una *API* gráfica ligada a un único sistema operativo (como *Direct3D* o *Metal*) anularía la flexibilidad del motor base, creando un cuello de botella innecesario. *OpenGL*, como estándar multiplataforma, garantiza que toda la solución sea portable.

- En el simulador existente se emplea *OpenGL* como *API* gráfica. Al utilizar nosotros la misma, evitamos introducir más dependencias externas.
- Una vez más, mi familiaridad con esta herramienta. Cuento con una sólida experiencia previa con esta *API*, adquirida tanto durante mi formación académica como a través del desarrollo de un proyecto personal que consistía en un motor gráfico basado en dicha *API*.

5.1.4. GLAD

Dado que *OpenGL* es una especificación, depende del fabricante del controlador de cada tarjeta gráfica implementar dicha especificación. Puesto que hay muchas versiones diferentes de controladores *OpenGL*, la ubicación de la mayoría de sus funciones no se conoce en tiempo de compilación y necesita ser consultada en tiempo de ejecución. Es tarea del desarrollador recuperar la ubicación de las funciones que necesita y almacenarlas en punteros de función para su uso posterior [78].

GLAD es una librería que se ocupa de realizar este trabajo de carga de funciones [78]. De nuevo, la razón principal para usar esta librería y no otros cargadores de *OpenGL* es mi familiaridad con ella.

5.1.5. GLM

GLM es una librería matemática en C++ para software gráfico basada en la especificación de *OpenGL*. Esta librería funciona perfectamente con *OpenGL*, pero también garantiza la interoperabilidad con otras bibliotecas y *SDK* de terceros. Es un buen candidato para el renderizado por *software*, procesamiento de imágenes, simulaciones físicas y cualquier contexto que requiera una librería de matemáticas simple y conveniente [79].

En nuestra solución se emplea para tener acceso a vectores de distintas dimensiones como tipos de datos y poder calcular las transformaciones necesarias para situar las mallas en la escena.

En el *software* de simulación de la empresa se hace uso de esta librería, por lo que decidimos emplearla como nuestra librería matemática de elección para no incluir nuevas dependencias externas innecesarias en este simulador.

5.1.6. stb_image_write

stb es un conjunto de librerías de dominio público y de un solo archivo para C/C++ desarrolladas por Sean T. Barret. *stb_image_write* es una de estas librerías y, como su propio nombre indica, se emplea para escribir imágenes a disco [80]. En concreto, en nuestra solución hacemos uso de ella para escribir la imagen final renderizada a un archivo *PNG*.

La empleamos por su sencillez y accesibilidad. Es una solución rápida y de buena calidad al problema de escribir imágenes a disco.

5.1.7. Herramientas propietarias de la empresa

Como describiremos en más detalle en la [Sección 7](#), a la hora de validar las imágenes renderizadas hacemos uso del *software* de simulación de *AUTIS*, y de otras herramientas desarrolladas por la empresa, para tareas misceláneas.

5.1.8. GLFW

GLFW es una librería escrita en C y diseñada específicamente para su uso con *OpenGL*. Su propósito es proporcionar las funcionalidades básicas necesarias para poder mostrar gráficos en pantalla, ya que *OpenGL* por sí mismo no se encarga de operaciones que dependen del sistema operativo. Concretamente, *GLFW* permite crear un contexto de *OpenGL*, definir los parámetros de una ventana y gestionar la entrada del usuario [78].

Esta herramienta la empleamos para crear el contexto, la ventana y gestionar la entrada del entorno de pruebas interactivo que se desarrolló junto a este Trabajo de Fin de Grado (véase el [Apéndice A](#)). De nuevo, se eligió por mi familiaridad con ella, dado que también fue empleada en el motor gráfico que he mencionado anteriormente.

5.1.9. Dear ImGui

Dear ImGui es una librería de interfaz gráfica de usuario para C++. Produce *buffers* de vértices optimizados que se pueden renderizar en cualquier momento en una aplicación 3D. Es rápida, portable, agnóstica y autónoma (sin dependencias externas). Está diseñada para permitir iterar rápidamente y

para darle la capacidad a programadores de crear herramientas de creación de contenidos y herramientas de visualización o depuración [81].

Esta librería también se utilizó exclusivamente para el desarrollo del entorno de pruebas interactivo ([Apéndice A](#)). La razón por la que fue elegida se puede discernir en su descripción. Es de licencia permisiva [81] y este entorno de pruebas es el tipo de aplicación para el que fue diseñada la librería.

6. Desarrollo de la solución

En la sección anterior presentamos la solución propuesta, donde se definieron la arquitectura general y las tecnologías seleccionadas. En esta sección nos adentramos en el desarrollo de dicha solución. Aquí se materializa el diseño conceptual en una implementación funcional, detallando los aspectos técnicos y las decisiones tomadas durante la escritura del código.

6.1. Diseño de la clase

Nuestro proyecto está contenido en la clase `PBRRenderer`. Esta ha sido diseñada como una interfaz de alto nivel, cuyo propósito es encapsular toda la lógica de renderizado y facilitar su integración en el software de simulación de *AUTIS*.

6.1.1. Puntero opaco

Más concretamente, nuestro proyecto no está contenido en una clase, sino dos: la interfaz `PBRRenderer` y su implementación `PBRRendererPImpl`. Utilizamos la técnica del *puntero opaco* o *pImpl* (*pointer to implementation*).

pImpl es una técnica de programación C++ que elimina los detalles de implementación de una clase de su representación como objeto, colocándolos en una clase separada, a la que se accede a través de un puntero opaco [82].

Dado que las variables miembro privadas de una clase participan en la representación como objeto, afectando a su tamaño y *layout*, y dado que las funciones privadas de los miembros de una clase participan en la resolución de sobrecargas (que tiene lugar antes de la comprobación de acceso a las variables miembro), cualquier cambio en estos detalles de implementación requiere la recompilación de todos los usuarios de la clase. *pImpl* elimina esta dependencia de compilación. Los cambios en la implementación no provocan la recompilación [82].

En el contexto de nuestro proyecto, una de las principales motivaciones para adoptar esta técnica es el encapsulamiento de las dependencias externas. La implementación de la clase `PBRRenderer` depende de una biblioteca de terceros, en este caso *RPR*, lo que obligaría a incluir las cabeceras de dicha biblioteca en el propio fichero de cabecera. Sin el patrón *pImpl*, esta inclusión propagaría la dependencia de *RPR* a todos los módulos que utilizaran nuestra clase. El uso del puntero opaco confina la inclusión de las cabeceras de *RPR* exclusivamente al fichero de implementación, manteniendo la interfaz (el fichero de cabecera) limpia. Esto no solo acelera drásticamente la compilación, sino que también crea una barrera de abstracción que facilitaría una futura sustitución del motor de renderizado sin impactar al código del usuario.

6.1.2. Funciones y variables miembro

La clase tiene una función de inicialización que debe llamarse en primer lugar. Después tiene las funciones `renderPreview()` y `renderFinal()`, que renderizan la escena en modo de previsualización para tiempo real, o con más detalle *offline*, respectivamente.

Entre las variables miembro de la clase encontramos `structs` que almacenan los parámetros usados para configurar el modelo de la superficie a inspeccionar, el material de la superficie a inspeccionar, las luces y las cámaras. En el caso del modelo se almacena también la información geométrica del mismo, como se detallará en la [Subsección 6.3](#). También podemos encontrar los parámetros de la escena o del renderizado, como la intensidad de la luz ambiente o la resolución de la previsualización.

Estos parámetros de configuración sirven poder recrear la escena y las condiciones de renderizado de la previsualización (en *HybridPro*) en el *backend Northstar* cuando el usuario desee renderizar la imagen final.

`PBRRenderer` contiene funciones `set` para los objetos que son únicos, es decir, que solo hay uno por escena. Y por tanto, solo hay un `struct` variable miembro para almacenar sus parámetros de configuración. Estos son el modelo y el material de la superficie a inspeccionar.

Para cada uno de los objetos de los que hay varios por escena (luces y cámaras), encontramos una función que añade un objeto de ese tipo a la escena y otras que actualizan un objeto de los añadidos previamente.

Los `structs` de las luces y la cámara, a parte de los parámetros de configuración, contienen un campo identificador. En `PBRRenderer` se almacenan en un `map` donde la clave es el identificador y el valor el `struct`. A pesar de que esto duplica el identificador (una vez como clave del `map` y otra como campo del `struct`), facilitaba el diseño de funciones tener el identificador contenido en el `struct`.

Mediante los identificadores el usuario puede indicar cuál es el objeto que quiere modificar cuando llama a las funciones de actualización de parámetros de luces y cámaras. Además, para estos objetos también hay una función para eliminar el objeto especificado mediante su identificador de la escena.

Para todos los objetos que tienen representación de sus parámetros como variables miembro de la clase hay además funciones *getter*. En el caso de las luces y las cámaras, el *getter* devuelve el *map*. Y la clase también contiene funciones para actualizar parámetros de la escena o del renderizado mencionados anteriormente.

En `PBRRenderer` se almacenan también como variables miembro todos los objetos de *RPR* que creamos, ya que necesitamos su referencia para poder borrarlos en el destructor de la clase. De esa manera evitamos que hayan fugas de memoria [83].

6.1.3. Tratamiento de errores

Para tratar errores, la clase tiene una variable `lastError` y una función que devuelve esta variable.

Las funciones de *RPR* devuelven un código. Este es 0 si se ha llevado a cabo con éxito, o, si ha sucedido un error, un número menor que 0 que informa sobre el error [84]. Hacemos uso de una macro `CHECK` con la que se envuelven las llamadas a funciones de *RPR*, de la manera `CHECK(rprAnyFunction())`.

Esta macro llama a una función `errorManager()` con el código de error y la línea actual si la función de *RPR* ha devuelto un código distinto de 0, y después hace `return false`. Todas las funciones de `PBRRenderer` que a su vez llaman a funciones de *RPR* son de tipo `bool`. De esta manera, la macro se encarga de terminar la ejecución de la función de nuestra clase si ha habido un error y de indicarlo devolviendo `false`.

La función `errorManager` escribe en `lastError` la línea del error y el código de error que ha devuelto *RPR*.

6.2. Escena

El primer paso de la inicialización es crear el contexto de *RPR*. Se trata del objeto base para todas las operaciones de renderizado [70]. Es el objeto raíz, y todas las entidades de *RPR* se crean para un contexto y dependen de él [85]. Discutiremos en más profundidad el proceso de creación del contexto en la [Subsección 6.7](#).

Una vez creamos el contexto, podemos crear la escena y asignarla al contexto. Una escena es un contenedor de nodos a renderizar [83]. Es donde situaremos la superficie a inspeccionar, las luces y las cámaras. Siempre que creamos alguno de estos objetos lo deberemos adjuntar a la escena.

Además de la superficie a inspeccionar, las luces y las cámaras, la escena contiene un par de elementos adicionales que detallaremos a continuación.

6.2.1. Suelo

`PBRRenderer` dispone de la posibilidad de incluir un suelo en la escena. Su función es evitar que la superficie a inspeccionar tenga aspecto de estar “flotando”.

El suelo consisten en un *quad* situado en el origen de coordenadas. Al inicializar la clase la malla se crea como se detalla en la sección [Subsección 6.3](#) y se añade a la escena.

El material del suelo es negro, opaco y completamente difuso, de esta manera absorbe casi toda la luz que recibe y la cantidad de luz que refleja es insignificante. El objetivo es que no distraiga de la superficie a inspeccionar ni interfiera en su iluminación.

Llamando a una función, el usuario puede activar o desactivar la presencia del suelo en la escena. Esta función elimina el suelo de la escena si estaba presente, o lo añade a la escena si no lo estaba.

6.2.2. Luz ambiental

Como se ha expuesto anteriormente, tener un factor de iluminación indirecta es importante para obtener escenas realistas. Si no, las secciones no iluminadas o en sombra se verían desproporcionadamente oscuras. Véase la [Figura 1](#)

En *RPR* se puede obtener iluminación indirecta mediante la creación de luces de entorno. Una luz de entorno permite emular radiancia proveniente de toda la esfera que conforma el horizonte de la escena [86]. Normalmente, con estas luces se emplea una textura HDR que se obtiene capturando la iluminación de una escena del mundo real o transformando una escena 3D. Luego, la luz de entorno trata cada *texel* de la textura como un emisor de luz. De este modo podemos captar eficazmente la iluminación global del entorno de una escena, dando a los objetos una mayor sensación de pertenencia [87].

Nosotros no poseemos ni los instrumentos ni la posibilidad de hacer una captura de la iluminación del entorno de los túneles de inspección. Por lo tanto, no podemos emplear una textura que codifique este

entorno para nuestra escena. Tenemos que recurrir a un método más simple de iluminación indirecta, la iluminación ambiental constante explicada en la [Subsección 2.4](#).

Sin embargo, *RPR* requiere que las luces de entorno empleen una textura. Así que, al inicializar la escena, creamos la luz con la función adecuada, cargamos una textura “especial” desde memoria principal a *RPR* y la añadimos a la escena.

Para cargar la textura hacemos uso de una función que crea una imagen en el contexto de *RPR* tomando como argumentos un arreglo con los datos que componen la imagen, un descriptor del formato de la imagen (número de canales de color y tipo de datos de cada canal) y un descriptor del tamaño de la imagen [88].

Esta textura que cargamos consiste en un solo pixel blanco. De esa manera tenemos una radiancia constante a lo largo de la esfera de la luz de entorno.

El usuario puede calibrar la intensidad de este componente ambiental mediante una función, que a su vez llama a una función de *RPR* que escala la contribución de la luz de entorno.

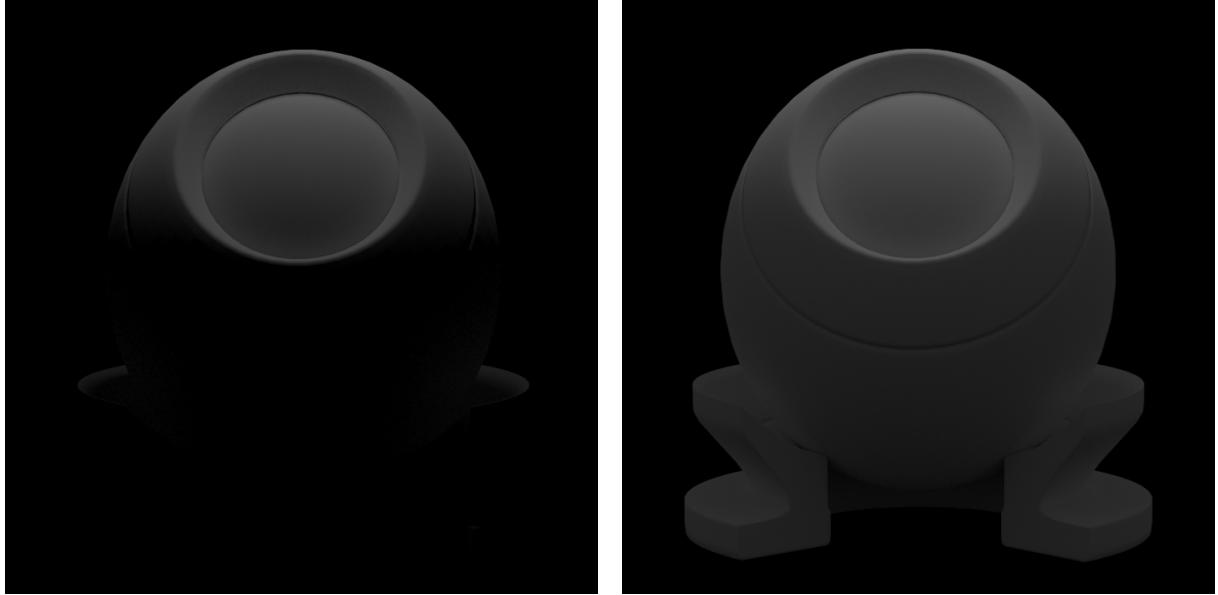


Figura 1: A la izquierda, una escena sin luz ambiental. A la derecha la misma escena con el componente de luz ambiental configurado a la máxima intensidad.

6.3. Modelo de la superficie a inspeccionar

El objeto superficie a inspeccionar en sí es por supuesto un elemento crucial de la escena, y por tanto de **PBRRender**. Y se permite cualquier modelo 3D como superficie a inspeccionar, tan solo hay que proveer la información geométrica en el formato que detallaremos a continuación.

La estructura de datos que representa un modelo 3D consiste en una colección de mallas y una transformación que sitúa la superficie a inspeccionar en la escena. Esta transformación puede ser una translación en cualquier eje, un escalado, tanto uniforme como no uniforme, una rotación también en cualquier eje, o cualquier combinación de estas operaciones.

Una malla se representa internamente como:

- El conjunto de vértices que componen la malla.
- Un conjunto de índices, que determinan la secuencia de dibujado de los vértices.
- El número de caras de la malla.
- Un vector de tamaño igual al número de caras, que en cada elemento contiene el número de vértices que componen esa cara. Esto permite soportar tanto modelos con caras triangulares, como con caras cuadrangulares, como mallas híbridas con caras tanto triangulares como cuadrangulares [89].

Un vértice es un **struct** de 32 bytes formado por un arreglo de 3 **floats** para la posición del vértice, otro arreglo de 3 **floats** para el vector normal asociado al vértice y un último arreglo de 2 **floats** para las coordenadas de textura.

El formato que hemos escogido para representar la información geométrica de una malla no es aleatorio, viene dado por cómo pide *RPR* que se especifiquen las mallas en su función de creación de *shapes* (objeto de *RPR* que representa la geometría en una escena [83]).

6.4. Material de la superficie a inspeccionar

El material del que está compuesto la superficie a inspeccionar que queremos renderizar es otro aspecto fundamental. Más concretamente, las propiedades visuales de este material. El color, el brillo, los acabados, etc. Al final, como hemos explicado, nosotros vemos un objeto si rayos que parten de una fuente de luz rebotan en él y luego llegan a nuestros ojos. Por tanto, definir cómo interactúa la luz con la superficie del objeto es crucial.

A la hora de permitir que el usuario controle esta interacción nos inspiramos en algunos de los principios de diseño del *BRDF* de Burley [90]:

- Los parámetros tienen que ser intuitivos y no necesariamente han de reflejar los principios físicos subyacentes.
- Hemos de exponer suficientes parámetros como para que el usuario pueda obtener el aspecto deseado, pero evitando abrumarlo con una inmensidad de controles a configurar.
- Los valores posibles de los parámetros deben estar en el intervalo $[0, 1]$.
- Cualquier combinación de los valores de los parámetros debería ser lo más convincente y físicamente correcta posible.

Siguiendo estos principios, el material de la superficie a inspeccionar cuenta con los siguientes cinco parámetros: *base color*, *roughness*, *metallic*, *clearcoat* y *clearcoat roughness*. Profundizaremos en ellos en las siguientes subsecciones.

En *RPR* los materiales se componen a partir de nodos. Existen numerosos nodos de material que pueden utilizarse. Los nodos son piezas atómicas de materiales que calculan un color de salida basado en entradas. Las entradas pueden ser valores de coma flotante, texturas o incluso otros nodos. Conectando nodos con entradas, se puede formar un “árbol de nodos” que calcula un material complejo [91].

Por debajo, para implementar el material en *RPR* se hace uso del *Uber Shader*, un nodo multiuso diseñado para crear materiales de aspecto realista, desde simples a complejos. Incluye una serie de capas ajustables, que agrupan ciertas propiedades de los materiales físicos. Cada componente tiene un peso para controlar hasta qué punto sus propiedades contribuirán a la apariencia general del material resultante [51].

El material de la superficie a inspeccionar está compuesto de tres capas: la capa difusa, la capa reflectante y la capa de recubrimiento. La capa difusa representa un material perfectamente difuso, como la tiza. La capa reflectante añade reflejos especulares a la capa difusa. Y, por último, el recubrimiento se utiliza para crear una capa de acabado, como pintura o barniz, sobre otros materiales [51].

La decisión de utilizar únicamente este *shader* de *RPR* para simular el material viene principalmente dada por el hecho de que es el único soportado en el *backend HybridPro* [71]. Sin embargo, también hay una correlación directa entre muchos de nuestros parámetros y los parámetros de configuración del *Uber Shader*, lo cual minimiza la necesidad de realizar un mapeo de nuestros parámetros a los del *Uber Shader*.

Para crear un material en *RPR* se ha de crear primero un sistema de materiales en el contexto, lo cual hacemos al inicializar la clase. Después se ha de crear el material en ese sistema de materiales (que en el caso del material de la superficie a inspeccionar también se hace al inicializar la clase, al ser solo uno), y después ese material se puede asociar a *shapes* o instancias (véase la [Subsección 6.5](#)) [83].

6.4.1. Base Color

Base Color o color base es un triplete de valores RGB que controlan el color del material ([Figura 2](#)).

Su implementación consiste en utilizar estos tres valores como el color de la capa difusa y de la capa reflectante.

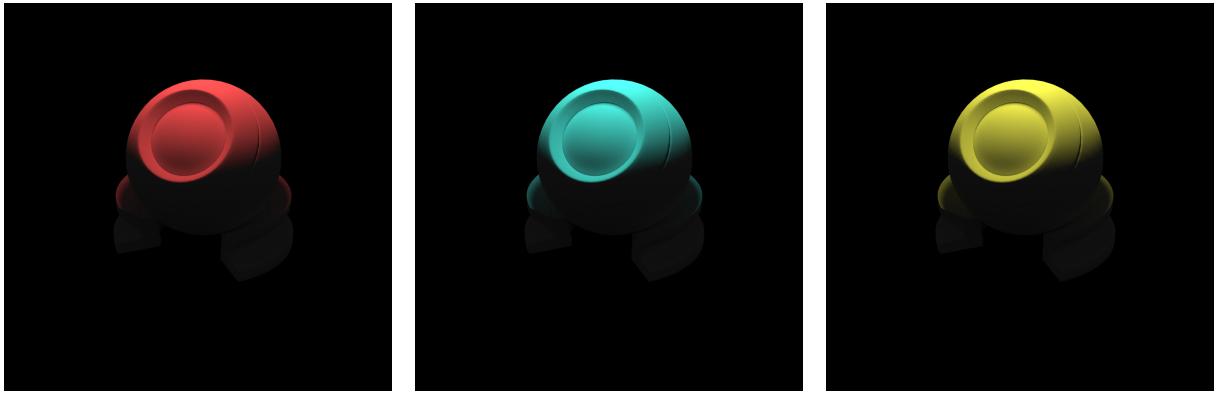


Figura 2: El efecto del parámetro *base color*. De izquierda a derecha, un *base color* rojo, un *base color* cian y un *base color* amarillo.

6.4.2. *Roughness*

Este parámetro representa la irregularidad de la superficie del material. A valores más bajos, más suave la superficie, menos se difumina el reflejo especular y la superficie se aprecia más brillante y reflectante. A valores más altos ocurre lo contrario. La superficie es muy irregular y el reflejo especular se difumina y se pierde. En estos casos el componente difuso domina la apariencia. Observése la [Figura 3](#).

Se trata de la rugosidad mencionada en la [Subsección 2.3](#). Valores altos dispersan la distribución estadística de las micronormales, y valores más bajos alinean las micronormales con la normal de la macrosuperficie.

En *RPR* este parámetro se utiliza como la *roughness* o rugosidad de las capas difusa y reflectante. Como se ha explicado antes, aumentar el valor de este parámetro tiene un efecto visual parecido a reducir el peso de la capa reflectante.



Figura 3: El efecto del parámetro *roughness*. De izquierda a derecha, un *roughness* muy alto, un *roughness* intermedio y un *roughness* mínimo.

6.4.3. *Metallic*

Metallic define si el material es metálico o no, lo cual es importante porque los metales interactúan con la luz de una manera marcadamente distinta a los materiales no conductores. Reflejan la mayor parte de la luz que reciben, y el resto es absorbida sin dispersión subsuperficial. Por tanto, no se aprecia componente difuso y todo el color viene de F_0 . Y, además, a diferencia de los dieléctricos donde el reflejo especular es del color de la luz que reflejan, los metales colorean el componente especular [3], y por lo tanto tienen un F_0 que no es blanco ([Figura 4](#)).

RPR soporta dos modos para las capas reflectantes. El modo IOR y el modo metálico. Para calcular la reflectancia en el modo IOR se emplea la magnitud física del índice de refracción [51]. Por tanto, es el más “físicamente basado” de los dos.

El modo metálico utiliza un parámetro *Metalness*. Esto sigue el sistema establecido en Physically-Based Shading en Disney [90] (en el que también nos basamos nosotros) donde la reflexión puede ser

metálica o dieléctrica. Un *metalness* de 1 da al objeto un color de reflexión especular de metal coloreado, y a 0, el objeto se trata como un dieléctrico que tiene especulares blancos [51], lo cual refleja el comportamiento explicado de los metales.

Nosotros empleamos el modo metálico para la capa reflectante, con una correspondencia directa de nuestro parámetro *metallic* al parámetro *metalness* de *RPR*.



Figura 4: El efecto del parámetro *metallic*. De izquierda a derecha, un *metallic* nulo, un *metallic* intermedio y un *metallic* máximo.

6.4.4. *Clearcoat*

Con este parámetro se puede controlar la presencia de una capa de acabado reflectante sobre el material. A mayor valor, más se nota el recubrimiento en el material, como se puede observar en la Figura 5. Este acabado es el que podemos encontrar en acrílicos, latas de refresco, o, por lo que nos es interesante, en acabados de la carrocería de automóviles.

Para la capa de recubrimiento en *RPR* empleamos el modo IOR, con un IOR igual a 1.5. Este valor es representativo del poliuretano, un compuesto común en barnices [21].

Así como en las otras dos capas (difusa y reflectante) el peso se mantiene a 1, el parámetro *clearcoat* se pasa directamente como el peso de la capa de recubrimiento.

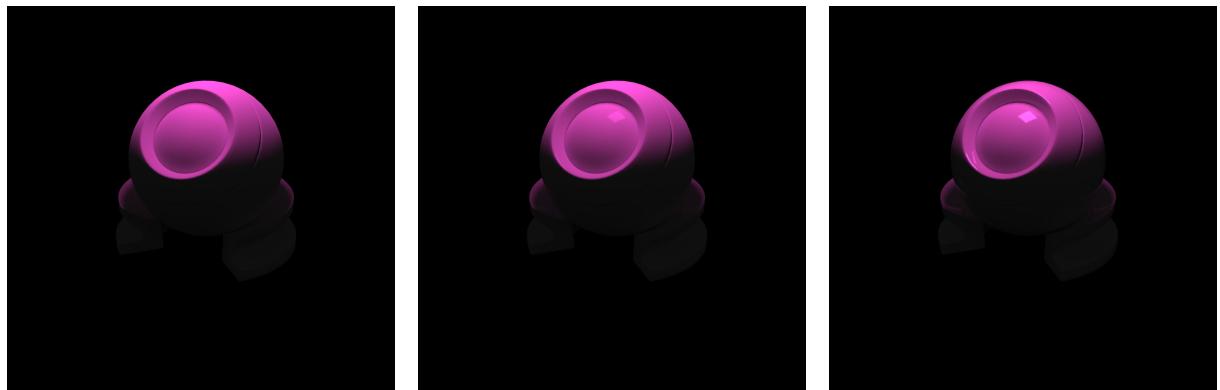


Figura 5: El efecto del parámetro *clearcoat*. De izquierda a derecha, un *clearcoat* nulo, un *clearcoat* intermedio y un *clearcoat* máximo.

6.4.5. *Clearcoat Roughness*

Este parámetro se comporta igual que *roughness* solo que para la capa de recubrimiento (Figura 5). Aunque, dado que esta capa representa barnices y acabados brillantes, se asume que este valor será próximo al cero.



Figura 6: El efecto del parámetro *clearcoat roughness*. De izquierda a derecha, con el valor de *clearcoat* al máximo, un *clearcoat roughness* muy alto, un *clearcoat roughness* intermedio y un *clearcoat roughness* mínimo.

6.4.6. Mapa de normales

La pintura de vehículos presenta un fenómeno característico conocido como piel de naranja o *orange peel*. Se trata una irregularidad visual que aparece en la superficie de ciertos recubrimientos, especialmente aquellos empleados para las carrocerías de automóvil. Se caracteriza por una textura que se asemeja a la piel de una naranja, compuesta por un patrón de pequeñas hendiduras. Estas irregularidades provocan que la superficie refleje la luz de manera no uniforme, creando un mosaico de áreas con brillo y áreas sin él [92].

Dado el contexto de este proyecto, resulta crucial disponer de la posibilidad de emular este fenómeno en el material de la superficie a inspeccionar. Con este fin, implementamos mapas de normales.

Usualmente, el vector normal \vec{n} que se emplea a la hora de calcular la radiancia saliente de un punto con la [Ecuación 3](#) viene determinado por la geometría de la malla [93].

Los mapas de normales son imágenes donde se almacena un vector normal en el triplete RGB de cada píxel que compone la imagen ([Figura 7](#)). Esta imagen se puede emplear como textura para que, en lugar de emplear el vector normal definido por los vértices del modelo, se emplee el vector normal almacenado en la textura a la hora de calcular la iluminación [93].

De esta manera, la iluminación captura detalles de la superficie del objeto que la malla no alcanza a modelar [93], como se puede observar en la [Figura 8](#). Esta técnica es especialmente útil para modelar pequeñas irregularidades en materiales que no se suelen modelar geométricamente, como arañazos, manchas o las hendiduras en la pintura que causan la piel de naranja.

El *Uber shader* de *RPR* soporta mapas de normales mediante una entrada para cada una de las capas difusa, reflectante y de recubrimiento, que permiten alterar los vectores normales que se usan para calcular la radiancia saliente de cada una de estas capas [51].

Para transmitir esta textura a la entrada correspondiente del *Uber Shader* tenemos que emplear otros

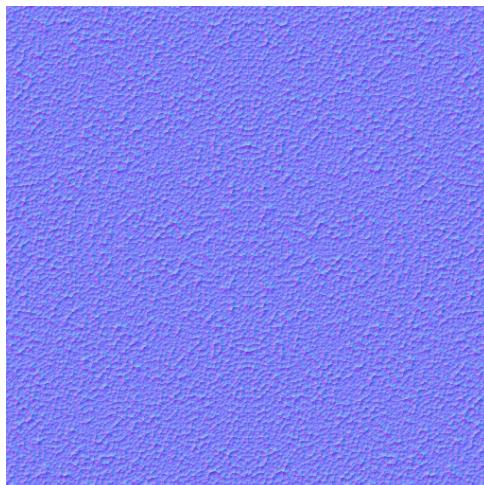


Figura 7: Un mapa de normales que modela la piel de naranja.

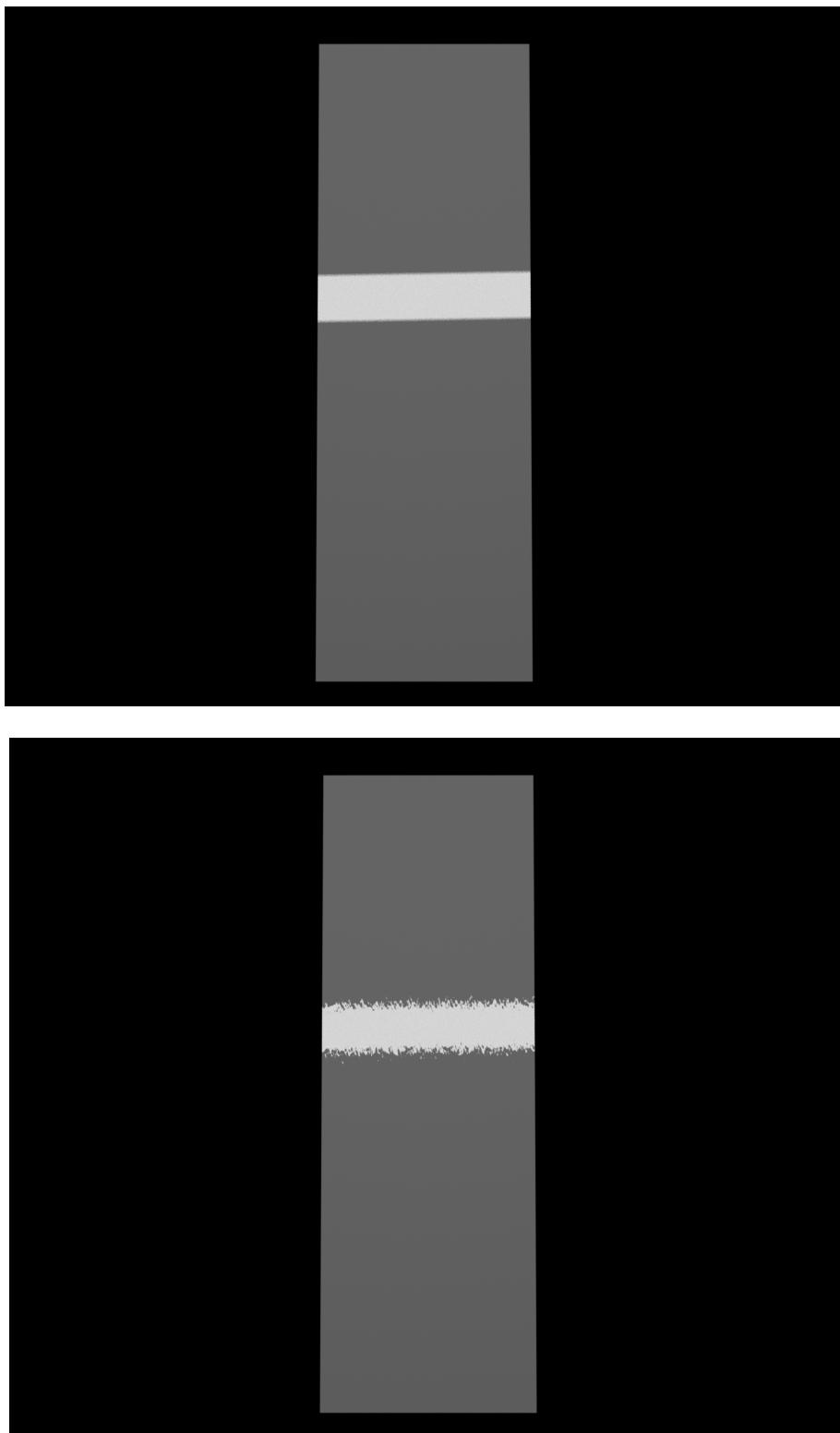


Figura 8: Dos placas renderizadas con el mismo material reflectante. Arriba, sin mapa de normales. Abajo, con un mapa de normales que emula la piel de naranja

dos nodos.

- `RPR_MATERIAL_NODE_IMAGE_TEXTURE` donde almacenamos como textura la imagen que contiene el mapa de normales [91].
- `RPR_MATERIAL_NODE_NORMAL_MAP`, al que le transmitimos el nodo `RPR_MATERIAL_NODE_IMAGE_TEXTURE` como entrada y procesa la textura como un mapa de normales. También podemos escalar el módulo de los vectores normales del mapa mediante una entrada de este nodo [94].

Por último, transmitimos el nodo de mapa de normales a la entrada correspondiente del *Uber Shader*. Para simular el efecto de piel de naranja, este mapa de normales se ha de emplear para la capa de recubrimiento. Nos encontramos en este caso con una limitación de *HybridPro*, ya que en este *backend* no se soportan mapas de normales para las capas de recubrimiento. Por consiguiente, tan solo será observable su efecto al renderizar la imagen final con *Northstar*.

Además, en nuestra solución el usuario dispone de una función para establecer un mapa de normales donde puede elegir si este mapa se aplica a la base de la superficie (capas difusa y reflectante del *Uber Shader*) o al recubrimiento (capa de recubrimiento del *Uber Shader*). Además, puede llamar a otra función para escalar la influencia del mapa.

6.5. Luces

Otro elemento esencial de una escena es la luz. Sin luz ni una cámara ni un ojo puede ver nada. El material no tiene nada para reflejar. Y no solo es que sea imprescindible su presencia, si no que cómo este iluminada la escena es posiblemente el factor más importante a la hora de cómo se verá. Por tanto, es esencial definir adecuadamente las fuentes de iluminación de la escena.

Partiendo de que buscamos renderizar imágenes fotorrealistas, podemos descartar la mayoría de luces analíticas como opciones adecuadas. Lo único que aún podemos considerar son las luces direccionales, ya que si fuéramos a renderizar escenas al aire libre, estas son una buena aproximación de la luz del sol, incluso en un contexto de *PBR* [95]. Pero, recordemos que este trabajo es una librería que será usada para renderizar escenas de carrocerías de vehículos bajo los arcos de luz de túneles de inspección. Por lo tanto, hemos de explorar otras opciones.

En el simulador, el arco de luz se modela mediante un conjunto de mallas que representan cada uno de los segmentos de luz que componen el arco. Aprovechando esta definición de los segmentos como mallas, les aplicaremos los materiales emisivos de *RPR* y estas mallas serán nuestras fuentes de luz.

Recordemos que un objeto con un material emisivo no solo refleja y refracta la luz que le llega, si no que emite luz [51]. Es decir, en este caso la radiancia emitida L_e de la [Ecuación 4](#) no se omite, ya que será distinta de cero. Y, de hecho, en materiales emisivos lo más probable es que la luz emitida sea lo suficientemente fuerte como para eclipsar la luz reflejada.

6.5.1. Material Emisivo

Por debajo, en *RPR* se pueden modelar mallas emisivas con dos materiales distintos. La capa emisiva del *Uber Shader* [51] y el *shader* emisivo dedicado [96]. Usaremos el *Uber Shader* si estamos en previsualización, porque, como hemos expuesto antes, en *HybridPro* solo se puede hacer uso de este *shader*. Y emplearemos el *shader* emisivo dedicado cuando rendericemos la imagen final, ya que este sí que está soportado en *Northstar* y satisface mejor nuestras necesidades.

La capa emisiva del *Uber Shader* permite configurar el color de la luz emitida, la intensidad, y si se emite luz por un lado de la superficie del material o por ambos. Además, también se puede configurar el peso de esta capa, al igual que el resto de capas del *shader*. Para controlar la intensidad, se ha de multiplicar el color por la intensidad deseada, aunque la capa emisiva del *Uber Shader* no está preparada para intensidades superiores a 1. Además, esta intensidad no representa ninguna magnitud del mundo real [51].

Por otro lado, el *shader* emisivo tan solo permite controlar el color y la intensidad, de igual manera que el *Uber Shader*, multiplicando el color deseado por la intensidad. Sin embargo, sí que está diseñado para intensidades superiores a 1 [97] y el valor de intensidad representa radiancia [96].

Configuramos la capa emisiva del *Uber Shader* para que emita luz solo por un lado y tenga peso igual a 1, y pasamos al color, de tanto la capa emisiva del *Uber Shader* como el *shader* emisivo dedicado, el color de la luz indicado por el usuario, multiplicado por la intensidad (en unidades de radiancia). No obstante, esto no da buenos resultados cuando empleamos el *Uber Shader* como material emisivo, ya que no está diseñado para intensidades superiores a 1, ni emplea unidades del mundo real, como hemos expuesto antes. Esto hace que la escena en previsualización tienda a presentar artefactos bajo ciertos ángulos de visión cuando se emplean luces con intensidades altas.

6.5.2. Cálculo de la intensidad luminosa

Los parámetros que exponemos al usuario para controlar el material emisivo de las luces son cuatro; dos de ellos son parte de la especificación técnica de la luz que se está intentado simular y los otros dos permiten al usuario configurarla más en profundidad.

El usuario puede indicar la potencia lumínica de la luz a simular mediante el primero de los parámetros de la especificación técnica, el flujo lumínico ϕ_v , medido en lúmenes (lm). Sin embargo, como se ha explicado anteriormente, *RPR* espera que se defina la intensidad en $W/(m^2sr)$ (radiancia). Por lo tanto, hemos de realizar la conversión.

El flujo lumínico es una unidad fotométrica. La fotometría es el estudio de la radiación electromagnética visible desde el punto de vista de su percepción por el sistema visual humano. Todas las medidas radiométricas, como el flujo, la radiancia, etc., tienen sus correspondientes medidas fotométricas. El flujo lumínico es el equivalente fotométrico del flujo radiante ϕ_e (medido en W) [4]. Para realizar la conversión entre ambos flujos necesitamos exponer otro el parámetro de la especificación técnica de la luz, la eficacia luminosa K , también conocida como *LER*.

La eficacia luminosa de la radiación, definida en la [Ecuación 6](#), mide la fracción de potencia electromagnética útil para la iluminación. Se obtiene dividiendo el flujo luminoso por el flujo radiante. Las longitudes de onda de la luz fuera del espectro visible reducen la eficacia luminosa, porque contribuyen al flujo radiante, mientras que el flujo luminoso de esa luz es nulo. Las longitudes de onda cercanas al pico de la respuesta del ojo contribuyen en mayor medida que las cercanas a los bordes [98].

$$K = \frac{\phi_v}{\phi_e} \quad (6)$$

Para obtener el flujo radiante, entonces, tan solo tenemos que dividir el flujo lumínico por la eficacia luminosa. Y para pasar de flujo radiante a radiancia tenemos que dividir por π y por el área de la superficie de la malla de la luz [99].

El cálculo del área de la superficie emisora es algo complejo, ya que la geometría de la luz puede estar definida por polígonos de cualquier número de lados (no solo triángulos), como se discutió en la [Subsección 6.3](#). Para abordar esto, nuestra implementación itera sobre cada cara de la malla y la descompone en un conjunto de triángulos mediante una técnica de triangulación en abanico. El área de cada uno de estos triángulos se calcula como la mitad de la magnitud del producto vectorial de dos de sus aristas. El área total se obtiene sumando las áreas de todos los triángulos resultantes.

Dado que el cálculo del área de la superficie de las mallas es un proceso computacionalmente costoso, buscamos que se haga el mínimo número de veces. Por tanto, la radiancia de la luz se calcula a partir de estos parámetros solamente cuando se añade una luz a la escena. El resultado se guarda en el `struct` de dicha luz. También se recalcula cuando se cambia la escala del modelo, ya que al que soportar la posibilidad de un escalado no uniforme, es necesario recalcular el área cuando se aplica esta transformación.

Los otros dos parámetros que permiten al usuario configurar la luz son el color ([Figura 9](#)) y la intensidad. Este último es un valor en el rango $[0, 1]$ que escala la radiancia de la luz ([Figura 10](#)). Por tanto, el valor que se pasa al parámetro color de los materiales emisivos de *RPR* es el color especificado por el usuario multiplicado por la intensidad y por la radiancia.

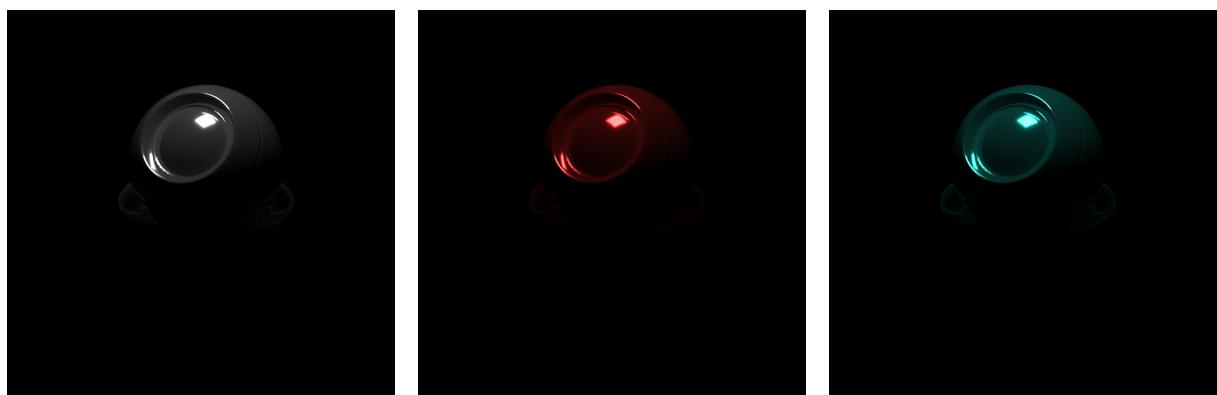


Figura 9: El efecto del parámetro del color luz. Con un material metálico con el mismo color base blanco para todas las imágenes, de izquierda a derecha, una luz de color blanco, rojo y azul celeste

Recordemos que en el contexto de *PBR* es importante emplear magnitudes del mundo real para obtener coherencia visual y poder validar los algoritmos empleados [21]. La razón para utilizar flujo lumínico es, por tanto, porque este es valor que conocemos de los segmentos de luz de los túneles de inspección.



Figura 10: El efecto del parámetro de intensidad de la luz. De izquierda a derecha, intensidades de 0,2, 0,5 y 0,9

6.5.3. Mallas

Como se ha explicado, hemos escogido que las fuentes de luz sean mallas con materiales emisivos porque esto es físicamente correcto y encaja con el diseño del simulador existente.

Para añadir una fuente de luz a la escena, se debe primero proveer la malla que se quiere utilizar, con el formato descrito en la [Subsección 6.3](#), junto con un identificador para la malla. Después, se puede agregar una luz a la escena pasando un **struct** que contiene:

- Un identificador para la luz.
- El identificador de la malla previamente cargada que empleará esta luz.
- Las mismas transformaciones de translación, rotación y escalado vistas en la [Subsección 6.3](#) para situar la luz en la escena.
- El parámetro color.
- El parámetro intensidad.
- El parámetro flujo lumínico.
- El parámetro eficacia luminosa.

Cuando se carga una malla para después ser usada como fuente de luz, lo que se está haciendo es crear el objeto *shape* en el contexto de *RPR*, es decir, subir la información geométrica de la malla a la memoria de la GPU. Después, cuando se añade una luz que hace uso de esa malla, se crea una instancia de esa malla y se añade a la escena, sin añadir nueva información geométrica a la GPU.

Las instancias están pensadas para reutilizar geometría con diferentes transformaciones y ajustes del *shader*. Aunque las instancias son una buena forma de ahorrar en almacenamiento, no es la única ventaja que ofrecen. Las instancias también pueden ayudar a mejorar significativamente el rendimiento en tiempo de ejecución. Como la mayoría de los trazadores de rayos, *RPR* preprocesa la escena para construir una estructura de aceleración antes de comenzar el renderizado. Cuanta más geometría se utilice, más tiempo se tardará en construir esta estructura y más tiempo se tardará en recorrerla en cada pasada de renderizado [\[100\]](#).

La decisión de emplear instancias para las mallas de las fuentes de luz se tomó porque los segmentos de luz usados en los túneles tienen todos la misma geometría. Entonces, en lugar de llenar la memoria con información geométrica de cientos de segmentos idénticos, subimos a la GPU la información geométrica de uno y creamos instancias de él para formar el arco de luz.

6.6. Cámaras

Las cámaras son el último aspecto crucial a tener en cuenta, al ser el receptor de la luz de la escena. En última instancia, siempre vamos a ver la escena a través de la cámara, por lo que su configuración y simulación es muy relevante.

En una escena se pueden añadir tantas cámaras como el usuario desee, pero obligatoriamente habrá de establecer, llamando a una función de la clase, una de las cámaras que haya añadido como la cámara

de renderizado. Si no, las llamadas a las funciones `renderPreview()` y `renderFinal()` devolverán un error y no producirán una imagen. Esta cámara de renderizado es, como su propio nombre indica, la cámara desde la cual se va a renderizar la escena.

En `PBRRenderer` las cámaras de posicionan y orientan en la escena mediante tres parámetros:

- La posición de la cámara en la escena, definida mediante un vector de tres componentes al igual que la posición de las luces y la superficie a inspeccionar.
- El vector *up*, el vector que representa el eje *y* positivo del espacio de la cámara.
- El punto de interés, que es el punto al que mira la cámara. Se emplea para determinar la dirección de visión de la cámara.

La elección de estos parámetros, una vez más, viene determinada por la función que se emplea por debajo en *RPR* para determinar la transformación de sus cámaras. Esta toma estos mismos tres argumentos [101].

En nuestra clase también se pueden determinar el *near plane* y *far plane* de cada cámara. Estos se tratan de los planos que determinan el inicio y el final del volumen de recorte de la cámara. Y el volumen de recorte es la región del espacio 3D que la cámara “ve” y que, por tanto, será procesada y finalmente renderizada [102]. Estos parámetros también se configuran directamente pasando el valor a las funciones *setters* correspondientes de las cámaras de *RPR* [56].

Para terminar la configuración de una cámara, también se definen sus parámetros físicos: distancia focal, distancia de enfoque, apertura (*f-stop*), tamaño del sensor y tiempo de exposición y sensibilidad. Los tres primeros se pueden pasar también directamente a la cámara de *RPR*, pero los restantes dos no tienen soporte en *RPR* [56].

La exposición en *RPR* se emplea para simular *motion blur*. Es una unidad de tiempo en el rango [0, 1], con 0 siendo el principio del fotograma y 1 siendo el final [103]. A pesar de que entre los parámetros de las cámaras en la documentación de *RPR* aparezca listada la exposición con la descripción “Controla el brillo de la imagen resultante.” [56], pudimos comprobar experimentalmente que esto no era así. En el mismo tutorial del *motion blur* [103] cambiamos la exposición final de 1 a 0,5, y observamos el mismo brillo en la escena resultante.

Por tanto, hemos de implementar el ajuste de exposición nosotros postprocesando la imagen que obtenemos de *RPR*. En la [Subsección 6.7](#) discutiremos la implementación del postprocesado, en esta sección nos centraremos en cómo ajustar la exposición.

6.6.1. Ajuste de exposición

Para calcular la exposición a aplicar a la imagen generada según los parámetros de la cámara de los que poseemos nos basamos en *Radiance*, una herramienta validada de simulación de iluminación [104]. Hacemos uso de la [Ecuación 7](#) [105].

$$EV = \frac{K \cdot t \cdot S^\circ}{N^2} \quad (7)$$

Donde:

- K es un factor empírico de conversión para transformar de unidades radiométricas a fotométricas. Su valor es igual a 2,81.
- t es el tiempo de exposición en segundos.
- S° es la sensibilidad en el estándar ISO, pero en escala logarítmica. Para convertir nuestra sensibilidad en escala lineal a la escala logarítmica empleamos la [Ecuación 8](#) [106], donde S es la sensibilidad en estándar ISO lineal.

$$S^\circ = 10 \cdot \log_{10}(S) + 1 \quad (8)$$

- N es la apertura en *f-stops*.

Después, para ajustar la exposición de la imagen multiplicamos cada píxel obtenido de *RPR* por la exposición calculada EV [107].

6.7. Renderizado de la escena

Una vez se han añadido y configurado todos los elementos de la escena, se llama a renderizar el contexto y este se encargará de renderizar la escena que tiene asociada.

Recordemos que `PBREnderer` soporta dos modos de renderizado. En tiempo real con `renderPreview()` y *offline* con `renderFinal()`. El proceso de creación del contexto y del renderizado serán ligeramente distintos según qué modo, como ilustraremos en el final de la subsección.

6.7.1. Creación del contexto

Para crear un contexto primero se registran los *backends* que se utilizarán para el renderizado. Eso es, *Northstar* para renderizado *offline* o *HybridPro* para renderizado en tiempo real. Después, creamos un objeto `rpr_context` con `rprCreateContext()`. Al crear un contexto, se pueden especificar los dispositivos que se utilizarán para la renderización, así como otros parámetros constantes del contexto. Por último, establecemos el *backend* activo para renderizar en el contexto [70].

Después establecemos los parámetros de renderizado: la máxima recursión del trazado de rayos, el número de iteraciones, si voltear la imagen verticalmente o no, etc. También hemos de establecer el parámetro *gamma* para la *corrección gamma*. Pero, cómo explicaremos más tarde, nos encargamos nosotros de postprocesar la imagen, así que mantenemos un $\gamma = 1$, que equivale a no realizar corrección *gamma*.

Por último, tenemos que crear el *framebuffer* de *RPR*, el búfer que almacena el resultado del renderizado [83]. Para ello hemos de determinar su formato y su resolución. Como formato empleamos 4 canales de color (RGBA), con el tipo de datos de cada canal como `float` de 32 bits. Y necesitamos adjuntar el *framebuffer* al *AOV* de color con `rprContextSetAOV()`, para que el renderizador sepa a qué *framebuffer* enviar sus resultados [83].

Dado que cada uno de los modos de renderizado usa un *backend* distinto, se tendrán que crear dos contextos distintos, uno para cada *backend*.

6.7.2. Proceso de renderizado

El renderizado se ejecuta cuando el usuario llama a `renderPreview()` o a `renderFinal()`.

Renderizado de *RPR*

Para renderizar la escena llamamos a `rprContextRender()`, que realiza la evaluación y acumulación en el *AOV* de color de una única muestra (o número de muestras si se establecen iteraciones). La llamada es bloqueante y el fotograma está listo cuando se devuelve. El contexto acumula las muestras para refinar progresivamente la imagen. Así, cada nueva llamada a render refina el resultado de la imagen con tantas muestras de color como número de iteraciones haya establecido el usuario [108].

Cuando la imagen cambia, para limpiar el *framebuffer* y empezar de cero se llama a `rprFrameBufferClear()`. De lo contrario, podrían producirse contaminaciones entre los fotogramas, debido al carácter acumulativo de la función de renderizado [109].

Una vez tenemos la imagen en el *framebuffer* del *AOV* de color, siguiendo el ejemplo del tutorial oficial de interoperatividad entre *OpenGL* y *RPR* [73], descargamos los datos de la imagen de la GPU a memoria principal para poder realizar el postprocesado.

Para obtener la imagen empleamos la función `rprFrameBufferGetInfo()` [110] para obtener primero el tamaño del *framebuffer* y ver que coincide con la cantidad de espacio que hemos asignado previamente para almacenar los datos de la imagen en memoria. Si coincide, empleamos esa misma función con otro argumento para obtener los datos guardados en el *framebuffer*, es decir, el fotograma renderizado.

Postprocesado mediante *OpenGL*

Ahora hemos de postprocesar la imagen. Podríamos hacer un bucle y recorrer la imagen, multiplicando los datos por los factores adecuados. Pero precisamente para ese tipo de cálculos se diseñaron las GPUs [111], así que haremos uso de ella.

Aprovechando que en el simulador se emplea *OpenGL*, y que cuando se inicialice `PBREnderer` ya habrá un contexto de *OpenGL* configurado, emplearemos esta *API* gráfica para ejecutar un *shader* que postprocesará la imagen.

En la inicialización de la clase se crean los objetos de *OpenGL* necesarios:

- El programa de *shaders* que vamos a ejecutar para postprocesar la imagen.
- La textura donde guardamos la imagen obtenida del *framebuffer* de *RPR*.

- El *framebuffer* de *OpenGL* al que irá el resultado de ejecución del *shader*.
- La textura que adjuntaremos al *framebuffer* de *OpenGL* creado y donde se volcará la imagen postprocesada cuando se termine de ejecutar el *shader*.
- El *quad* que cubre toda la pantalla sobre el que renderizaremos la imagen postprocesada.

Después de obtener la imagen de *RPR*, establecemos el *viewport* a la resolución de la imagen de *RPR*, vinculamos el *framebuffer* de *OpenGL*, el programa de *shaders* y la textura que contendrá el fotograma *RPR*.

Seguidamente, subimos a la GPU la imagen de *RPR* llamando a la función `glTexSubImage2D`. Ahora esta imagen se encontrará en la textura vinculada.

Por último pasamos las variables uniformes al *shader*, vinculamos el *VAO* (*Vertex Array Object*) del *quad* y llamamos a la función de renderizado de *OpenGL*.

El *shader* de vértice que empleamos es un programa simple que se dedica a pasar hacia delante en la *pipeline* las posiciones y las coordenadas de textura del *quad*. No aplicamos ninguna transformación porque los datos de vértices que empleamos ya están en coordenadas del dispositivo normalizado.

En el *shader* de fragmento realizamos el postproceso. Este tiene varias variables uniformes: la textura de la imagen obtenida del *framebuffer* de *RPR*, el factor de exposición *EV* calculado en CPU previamente al ejecutar el *shader* y el parámetro *gamma*. En la función `main()` muestreamos esta textura usando las coordenadas de textura obtenidas del *shader* de vértice, le aplicamos el ajuste de exposición discutido en la [Subsección 6.6](#), y luego aplicamos corrección *gamma*.

Una vez se ha terminado de ejecutar el *shader*, la imagen postprocesada se queda en la textura adjuntada al *framebuffer* de *OpenGL*.

Corrección *gamma*

La corrección *gamma* mencionada anteriormente es un paso fundamental en el procesado de la imagen. Los monitores no muestran los colores de forma lineal, sino que aplican una curva de potencia con un valor conocido como *gamma* γ . Esto provoca que los valores intermedios de brillo de la imagen se oscurezcan. Dado que todos los cálculos de iluminación se han realizado en un espacio lineal para ser físicamente correctos, mostrar el resultado directamente en el monitor produciría una imagen final incorrecta, con un balance de brillos erróneo y una pérdida notable de detalle en las zonas oscuras [\[112\]](#).

Para solucionar esta discrepancia, aplicamos la función inversa a la del monitor sobre el color final, justo antes de su visualización. En nuestro *shader* de fragmento, tomamos el color muestreado de la textura tras el ajuste de exposición, y lo elevamos a una potencia de $1/\gamma$. De este modo, cuando el monitor aplica su propia curva *gamma*, ambos efectos se anulan y la imagen mostrada se corresponde con los colores calculados linealmente.

RPR permite al usuario establecer un valor de γ y aplicarlo al resultado obtenido [\[113\]](#). Pero, dado que buscamos ajustar la exposición de la imagen que obtenemos del *framebuffer*, y la corrección *gamma* ha de ser el último paso antes de mostrar la imagen por el monitor [\[112\]](#), hemos de aplicarla nosotros después de postprocesar el fotograma.

6.7.3. Diferencias entre previsualización e imagen final

Las diferencias entre estos dos modos se deben principalmente a dos factores:

- `renderPreview()` tiene requisitos de tiempo real y está pensado para ser llamado una vez por fotograma del simulador, `renderFinal()` es *offline* y está pensado para ser llamado una vez por cada imagen final a generar a lo largo de una sesión.
- `renderPreview()` emplea el *backend HybridPro*, `renderFinal()` emplea el *backend Northstar*.

Tiempo real VS *offline*

En cuanto al primer factor, para empezar el contexto de *HybridPro* se crea al inicializar la clase y se destruye, junto con todos los objetos adjuntos a él, con el destructor de *PBRRenderer*. Sin embargo, el contexto de *Northstar* se crea y se destruye en la misma función `renderFinal()`.

Una consecuencia de crear el *framebuffer* de *RPR*, el de *OpenGL* y las texturas de *OpenGL* con la inicialización, y no con la función de renderizado, es que cuando el usuario cambia la resolución de la previsualización, se han de borrar y crear de nuevo con la nueva resolución.

También, la función `renderPreview()` usa los valores de parámetros de renderizado (resolución, iteraciones, etc.) establecidos en las variables miembro de la clase, mientras que la función de renderizado *offline* toma esos parámetros como argumentos de la función.

Uno de esos argumentos es una ruta a un archivo, ya que en `renderFinal()` guardamos la imagen resultante del renderizado en un archivo PNG. Esto se lleva a cabo vinculando la textura que contiene la imagen postprocesada y bajándose los datos del fotograma de la GPU a memoria principal haciendo uso de la función `glGetTexImage()`. Después convertimos los valores de `floats` (rango [0, 1]) a bytes (rango [0, 255]) y los escribimos a memoria usando la función `stbi_write_png()` de la librería `stb_image_write` [80].

Por otra parte, a `renderPreview()` el único argumento que se le pasa es un puntero a un entero sin signo, que la función se encarga de llenar con el identificador de la textura que contiene la imagen postprocesada. Después, el usuario puede utilizar ese fotograma en su aplicación de *OpenGL*.

Por último, recordemos que al llamar a `rprContextRender()` se acumulan las muestras en el *framebuffer* de color, y así se va refinando la imagen. Aprovechamos esta característica en el modo de previsualización solo llamando a limpiar el *framebuffer* de *RPR* cuando algún parámetro de la escena cambia, incluyendo la cámara de renderizado. De esta manera, cuando el usuario realiza un cambio en la escena, la imagen se ve ruidosa, pero inmediatamente se estabiliza y se mantiene así mientras el usuario no altere nada. Así, se puede compensar que, para obtener cantidades de fotogramas por segundo interactivas, la previsualización se renderice con un número de iteraciones bajo, comparadas con las normalmente empleadas para renderizar la imagen final.

HybridPro* VS *Northstar

En cuanto al segundo factor, los distintos *backends*, este conlleva que cuando se cree el contexto se registre *HybridPro* para el contexto usado para la previsualización y *Northstar* para el contexto usado para renderizar la imagen final.

A la hora de especificar los dispositivos empleados para cada contexto, en *HybridPro* hacemos uso de la GPU y en *Northstar* idealmente de la GPU y la CPU. Pero, cabe destacar que para emplear la GPU como dispositivo usando *Northstar* en una gráfica *NVIDIA* se ha de tener el *SDK* de CUDA instalado. Por ello, el último argumento que se le pasa a `renderFinal()` es un `bool` que determina si se usa la GPU o no.

Otra diferencia es el material usado para las mallas luminosas, como se ha explicado en la [Subsección 6.5](#).

En último lugar, en el renderizado, en *Northstar* hay que hacer un paso más a parte de llamar a `rprRenderContext()` que no se realiza en *HybridPro*. En *Northstar* el *framebuffer* del *AOV* de color es solo para uso interno, no está pensando para ser mostrado. En su lugar se ha de mostrar el *framebuffer* resuelto, que es otro *framebuffer* que se ha de crear y luego llenar con el resultado de llamar a la función `rprContextResolveFrameBuffer()`. Esta se encarga de tomar el *framebuffer* de color, procesarlo para que se pueda mostrar como renderizado final y volcar el resultado en el *framebuffer* resuelto [114]. Es este el *framebuffer* que luego postprocesamos y escribimos como archivo en el caso del renderizado *offline*.

Conclusión

Sin embargo, pese a todas las diferencias de implementación listadas, la mayor diferencia entre la previsualización y la imagen final es la calidad y el fotorrealismo de la imagen renderizada. Pese a acumular iteraciones en la previsualización, *HybridPro* no puede acercarse al nivel de fidelidad de *Northstar*. Y esto se ve acentuado por el aspecto incorrecto de la iluminación en *HybridPro* debido a las limitaciones de la capa emisiva del *Uber Shader*, como se expuso en la [Subsección 6.5](#), y a la ausencia de normales texturizadas para la capa de recubrimiento (véase la [Subsubsección 6.4.6](#)).

Además, una de las discrepancias más notables y problemáticas encontradas durante el desarrollo es una inconsistencia en la relación de aspecto entre los dos *backends*. Al renderizar una escena idéntica con la misma configuración de cámara y resolución, hay una ligera pero perceptible distorsión geométrica horizontal, como se puede observar en la [Figura 11](#). Hemos determinado que esto es un error de *RPR*.



(a) *HybridPro*



(b) *Northstar*

Figura 11: Imágenes renderizadas a partir de un tutorial oficial de *RPR*. La única diferencia entre las dos imágenes es el *backend* empleado para renderizarlas. Se puede observar como la imagen renderizada con *Northstar* tiene menos ruido, refleja mejor la intensidad de la luz y como el modelo es ligeramente más ancho.

7. Resultados

Como hemos reiterado a lo largo de este trabajo, el paradigma del *PBR* tiene como objetivo fundamental la simulación fiel de la interacción entre la luz y las propiedades físicas de los materiales. El fin último del paradigma es generar imágenes cuyo nivel de fotorrealismo las aproxime de manera indistinguible a la realidad que percibimos. Por consiguiente, la métrica de validación más apropiada para cualquier implementación *PBR* no reside en análisis teóricos, sino en la comparación directa de sus resultados con una escena del mundo real que sirva como referencia [27]. Adoptamos este enfoque empírico para evaluar el éxito de nuestra solución.

Además, no debemos perder de vista que este es un trabajo desarrollado para una empresa que busca utilizar nuestra solución como sustituto a la captura de imágenes reales de carrocerías de vehículos en los túneles de inspección. De aquí también se deriva naturalmente que el método de validación sea la comparación con capturas realizadas por cámaras en el mundo real.

Para esta validación hemos tomado fotografías de superficies en el mundo real mediante un dispositivo que emula una versión reducida y simplificada de un túnel de inspección. Este dispositivo cuenta con los mismos segmentos de luz y cámaras empleadas en los túneles de la empresa.

Después, hemos recreado esta escena del mundo real en *RPR* mediante la clase desarrollada. Se ha intentado replicar con la máxima fidelidad posible la configuración y disposición en la escena de todos los elementos.

Por último, hemos comparado las fotografías tomadas en el mundo real con los fotogramas renderizados mediante nuestra clase para verificar el grado de fotorrealismo de nuestra solución.

7.1. Captura de fotografías

7.1.1. Dispositivo de captura

Con el objetivo de realizar esta validación, los ingenieros de la empresa construyeron un dispositivo para la captura de fotografías en unas condiciones similares a las que se toman en los túneles de inspección de la empresa.

Estos túneles tienen un arco de luz y una agrupación de cámaras posicionadas estratégicamente por el túnel para que el conjunto de sus campos de visión cubra completamente la superficie de la carrocería. Las carrocerías u objetos a inspeccionar se montan en una cinta transportadora y se desplazan a lo largo del túnel. A medida que avanza la superficie a inspeccionar por el túnel, las cámaras captan diferentes secciones de la superficie bajo distintas condiciones de iluminación.

El dispositivo fabricado para la validación intenta emular estos túneles, manteniendo una simplicidad y un tamaño que permitiría su construcción en un tiempo razonable, y facilitara su transporte desde la sede de la empresa en Gandía hasta la UPV, donde se capturaron las fotografías.

Este túnel simplificado está formado por una estructura de aluminio colocada sobre unos ruedines para su transporte, como se puede observar en la [Figura 12](#). Sobre la estructura se soportan dos segmentos de luz y una cámara que produce imágenes en blanco y negro. Estos segmentos y esta cámara son del mismo modelo que los empleados en los túneles reales de la empresa.

El control del dispositivo se realiza conectándolo a un ordenador mediante un cable *Ethernet*. Este ordenador debe tener su dirección IP en rango, tanto con la IP de la cámara, como con la IP del controlador de los segmentos.

Para encender o apagar los segmentos, se desarrolló un programa que permite cambiar la intensidad de ambos segmentos a la vez, o de tan solo uno. De esta manera, se pueden apagar con intensidad igual a 0, o regular la potencia lumínica con intensidades en el rango (0, 100], de una manera similar a nuestro parámetro de intensidad lumínica discutido en la [Subsección 6.5](#).

Las fotografías se toman a través del simulador existente. Este cuenta con una funcionalidad llamada *Tunnel Camera Align*, que permite al usuario configurar una cámara en el simulador con las mismas propiedades que una cámara instalada en un túnel. Sin embargo, nosotros la empleamos porque con esta herramienta podemos acceder al *feed* de la cámara, y guardarnos los fotogramas que captura la cámara como imágenes *PNG*.

El simulador también nos permite establecer el tiempo de exposición de la cámara en microsegundos. Como expondremos más adelante, este es uno de los dos parámetros que variaremos en la escena para poder asegurar la fidelidad de nuestra solución bajo distintas condiciones, y asegurar así su robustez.



Figura 12: Dispositivo de captura empleado para tomar las fotografías. Los segmentos son el elemento de plástico negro que cuelgan de la parte superior del soporte. La cámara se encuentra al final del soporte de metal que sobresale de la parte superior de la estructura.

7.1.2. Elementos capturados

El objetivo final del proyecto es poder simular fielmente las imágenes de carrocerías que captan las cámaras de los túneles de inspección. Pero, dada la dificultad de transportar y seguidamente alojar una carrocería en el edificio del DSIC de la UPV, se optó por, en su lugar, capturar placas donde la empresa había probado distintos tipos de pinturas y acabados.

Estas placas, aunque no tienen la forma de una carrocería, están recubiertas con los mismos acabados de las carrocerías. Por tanto, son igualmente útiles para comprobar si nuestra solución puede simular dichos materiales de manera efectiva. Están recubiertas por una cara de *clearcoat* con pintura de diferentes colores por debajo y por el reverso de *primer* [115]:

- El *clearcoat*, discutido anteriormente, es un revestimiento brillante y transparente que forma la interfaz final con el entorno ([Figura 13](#)).
- El *primer* es la primera capa que se aplica. Entre otras cosas, sirve para igualar la superficie de la carrocería, facilitar la adhesión de las capas de pintura posteriores a la superficie y proteger el metal subyacente ([Figura 14](#)).

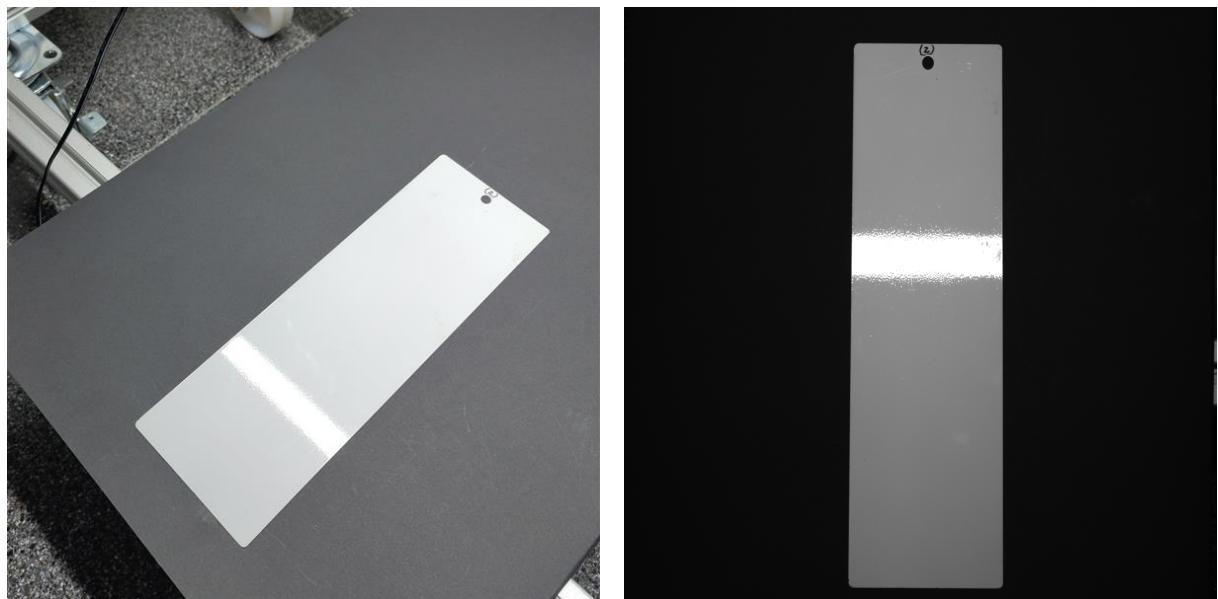


Figura 13: A la izquierda una placa con *clearcoat* blanco fotografiada con una cámara de móvil. A la derecha la misma placa capturada mediante la cámara del dispositivo.

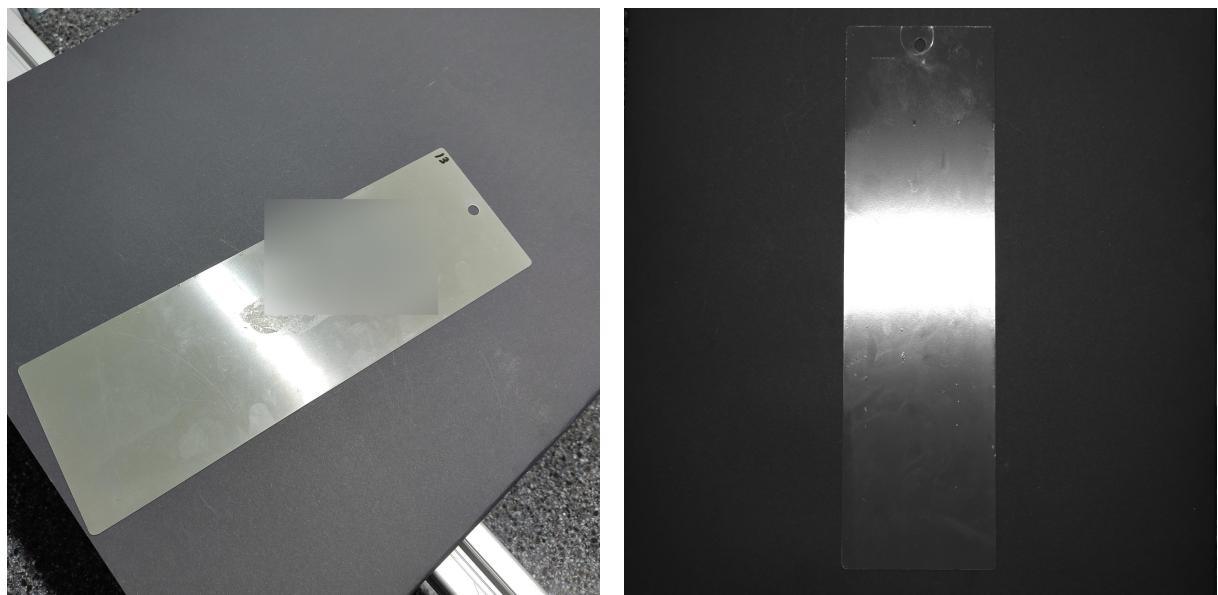


Figura 14: A la izquierda una placa con *primer* fotografiada con una cámara de móvil. A la derecha otra placa también con una capa de *primer* capturada mediante la cámara del dispositivo.

7.1.3. Condiciones de captura

Para garantizar un entorno de iluminación controlado, las capturas se llevaron a cabo en un seminario. Este es un espacio con ausencia de luz natural. Esta elección se debe a que la iluminación ambiental representa una variable difícil de modelar y replicar con precisión, ya que como hemos explicado en la [Subsección 6.2](#), no disponemos de las herramientas para capturar las condiciones de iluminación de un entorno. Al minimizar la relevancia de dicho factor, se mitiga el impacto de las limitaciones de nuestra aproximación simplificada a la iluminación ambiental, asegurando una mayor fidelidad en la recreación de la escena.

Además, también era clave asegurar que todo lo capturado en las imágenes fuera replicable. Y no disponíamos de la malla de los soportes de aluminio que conforman la estructura del dispositivo. Por tanto, era crucial ocultarlas. Con este fin posicionamos una tabla de madera de 8mm de grosor encima de dicha estructura. Los elementos a capturar se posicionaron sobre esta tabla.

Sobre la tabla adherimos una cartulina negra mate que absorbe la mayor parte de la luz que recibe, con el mismo fin que el del material del suelo discutido en la [Subsección 6.2](#). De hecho, dicho suelo es la recreación de la cartulina. Recordemos, el objetivo es minimizar los reflejos provenientes de la propia tabla y facilitar la distinción del elemento capturado respecto al fondo.

A la hora de tomar las fotografías, con el propósito de validar la robustez de nuestra solución, el proceso de captura se diseñó para evaluar su rendimiento bajo un rango diverso de condiciones. Se introdujeron variaciones sistemáticas en dos parámetros: la intensidad de la iluminación de los segmentos y el tiempo de exposición de la cámara. Este enfoque permite verificar que **PBRRender** mantiene la fidelidad de las imágenes generadas a través de un espectro de escenarios lumínicos y de adquisición de imagen, demostrando así su aplicabilidad general.

Con 3 valores de intensidad luminosa distintos y 3 tiempos de exposición distintos, se tomaron un total de 9 capturas por cada elemento que fotografiábamos. Los 3 valores seleccionados de intensidad luminosa fueron los mismos para todos los objetos. Estos valores fueron 20, 50 y 90. Véase la [Figura 15](#)



Figura 15: La misma placa de *clearcoat* blanca bajo las 3 intensidades luminosas distintas y una exposición constante de 0,004s. De izquierda a derecha, 20, 50 y 90.

Por otro lado, decidimos que fueran los tiempos de exposición escogidos los que variaran según el elemento, ya que la cantidad de luz reflejada varía sustancialmente según el material del objeto. Metales o pinturas blancas eran mucho más brillantes en las mismas condiciones de intensidad lumínica y tiempo de exposición que pinturas negras.

Para cada elemento repetimos el mismo algoritmo. Establecimos la intensidad lumínica a 20 y buscamos el mínimo tiempo exposición que no produjera una imagen en negro. Después, cambiamos la intensidad a 90 y buscamos la máxima exposición donde no hubiera saturación a blanco. Para el tercer valor de tiempo de exposición, seleccionamos el punto medio entre los valores a cada extremo. Se puede observar un ejemplo en la [Figura 16](#).

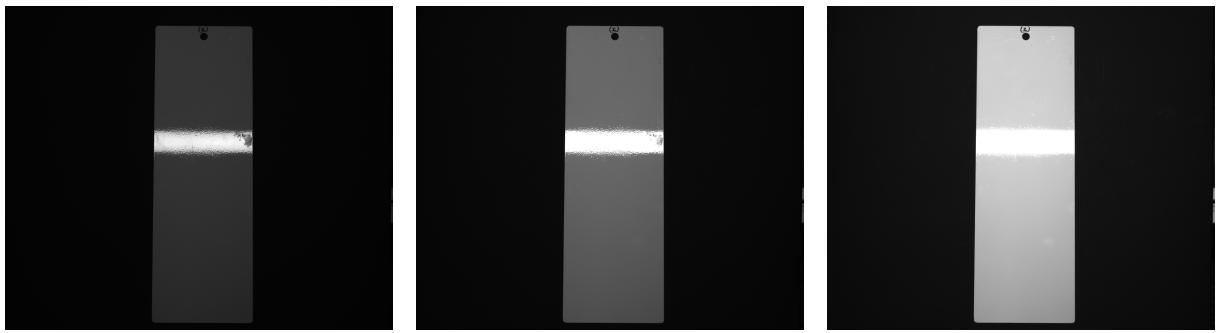


Figura 16: La misma placa de *clearcoat* blanca bajo los 3 tiempos de exposición seleccionados para ella, y una intensidad luminosa constante de 90. De izquierda a derecha, en este caso, 0,001s, 0,002s y 0,004s.

7.2. Recreación mediante PBRRender

El primer paso que tomamos a la hora de recrear la escena consistió en posicionar las luces y la cámara. Con el fin de garantizar una correspondencia 1:1 entre el entorno real y el virtual, la empresa nos facilitó un archivo de escena de su simulador. Dicho archivo replicaba con exactitud la configuración del dispositivo de captura que habían construido, proporcionando las transformaciones precisas de posición y orientación tanto para la cámara como para los segmentos de luz.

En nuestra solución, los dos segmentos de luz fueron definidos como dos *quads* con las dimensiones estipuladas por el archivo de escena. La transformación de la cámara también fue copiada de los valores determinados en el simulador, al igual que los valores de apertura, distancia focal y distancia de enfoque.

Sin embargo, al poner esta configuración en práctica, observamos una discrepancia en la posición del haz especular entre nuestra escena virtual en *RPR* y las fotografías capturadas. Empíricamente, observamos que el vector *up* de la cámara debía ser invertido y la posición de las luces debía ser desplazada un centímetro.

Después, establecimos los parámetros que constituyen la especificación técnica de las fuentes de luz, el flujo lumínico y la eficacia luminosa de la radiación. La empresa nos proporcionó una ficha técnica de los segmentos donde venía listado su flujo, pero no la eficacia. Así que tomamos un valor típico de eficacia luminosa de un LED blanco, 100 [116]

Para recrear las placas también hicimos uso de un *quad* con la anchura y altura que fue medida al tomar las capturas. Para su posicionamiento en la escena, resultaba difícil juzgar las diferencias en posición y rotación empleando el entorno interactivo debido a las diferencias de proporción entre la ventana del entorno y la resolución de la imagen capturada. Entonces, renderizamos una imagen con pocas iteraciones para agilizar el proceso, la comparamos con la posición de la placa en la captura que estábamos recreando y ajustamos la posición de la placa virtual, repitiendo el proceso tantas veces como fuera necesario.

Por último, para la configuración del material, empleamos la máxima intensidad lumínica y tiempo de exposición, y ajustamos sus parámetros en el entorno interactivo. Cuando el resultado se asemejaba al de la captura, para afinar los detalles, seguimos un algoritmo similar al anterior, ya que *HybridPro* no soporta los mapas de normales que hacen posible visualizar la piel de naranja.

7.2.1. Recreación del *clearcoat* blanco

Para esta placa (Figura 17) empleamos los siguientes valores de los parámetros de nuestro material:

- *BaseColor* = (255, 255, 255)
- *Roughness* = 0,628
- *Metallic* = 0,818
- *Clearcoat* = 0,964
- *ClearcoatRoughness* = 0,0

El color blanco y el valor alto de la capa de recubrimiento son autoexplicativos. El valor de *metallic* es alto porque se trata de una placa de metal y porque un valor más alto de este parámetro potencia el componente especular. Y junto con una rugosidad más alta esta potencia se distribuye por la mayor parte de la placa, disminuyendo gradualmente a medida que se aleja del centro del haz. Por otra parte, el valor de la rugosidad del recubrimiento es cero para obtener unos bordes del haz especular lo más definidos posibles.

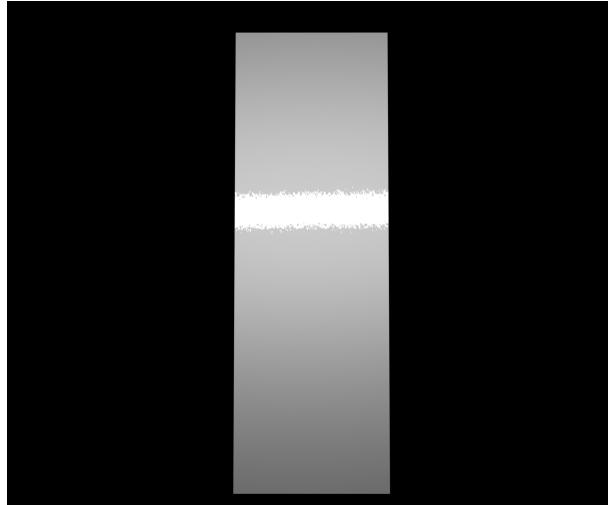


Figura 17: Imagen en blanco y negro de la placa de *clearcoat* blanca simulada en **PBRRenderer** con intensidad lumínica a 0.9 y un tiempo de exposición de 0,004s.

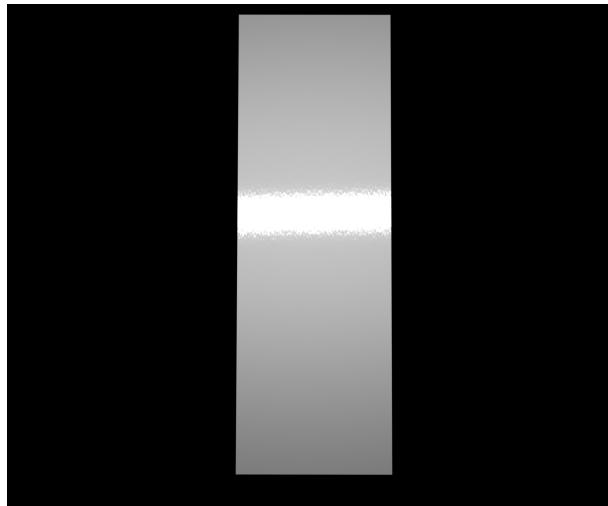


Figura 18: Imagen en blanco y negro de la placa de *clearcoat* gris simulada en **PBRRenderer** con intensidad lumínica a 0.9 y un tiempo de exposición de 0,011s.

7.2.2. Recreación del *clearcoat* gris

Para esta placa (Figura 18) empleamos los siguientes valores de los parámetros de nuestro material:

- $BaseColor = (102, 102, 102)$
- $Roughness = 0,470$
- $Metallic = 0,864$
- $Clearcoat = 1,0$
- $ClearcoatRoughness = 0,055$

Se trata de una placa gris, por tanto el color base también es un gris. Y la rugosidad es ligeramente más baja que en el blanco, porque en aquel ese valor de rugosidad saturaba demasiado a blanco el área alrededor del haz especular. Además, aquí buscábamos un haz especular ligeramente más difuminado en los bordes, por ello la rugosidad de la capa de recubrimiento no es cero. En cuanto al resto de parámetros, se trata de un caso muy similar a la placa de *clearcoat* blanco.

También capturamos y por tanto recreamos esta misma placa con un ángulo distinto para asegurar la robustez de nuestra solución, como se puede observar en la Figura 19.

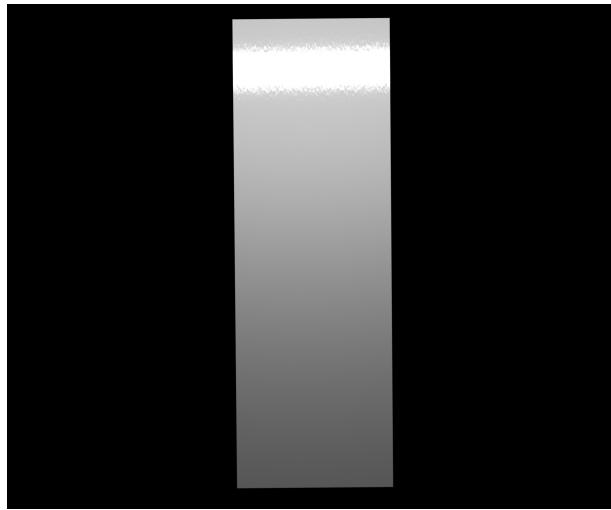


Figura 19: Imagen en blanco y negro de la placa de *clearcoat* gris simulada en **PBRRender** con intensidad lumínica a 0.9 y un tiempo de exposición de 0,011s, pero con una inclinación de 6°.

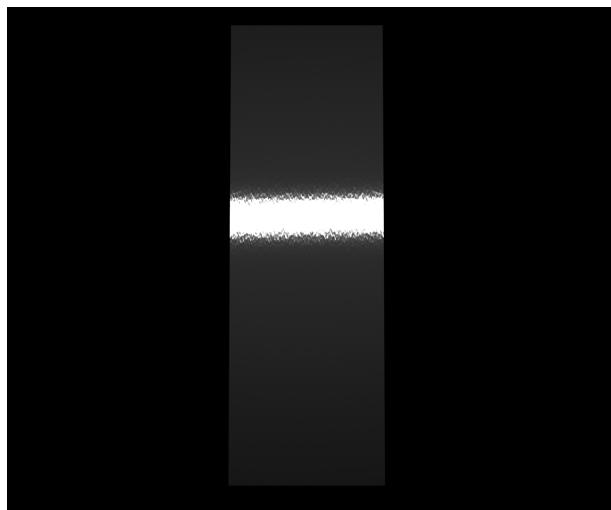


Figura 20: Imagen en blanco y negro de la placa de *clearcoat* negra simulada en **PBRRender** con intensidad lumínica a 0.9 y un tiempo de exposición de 0,016s.

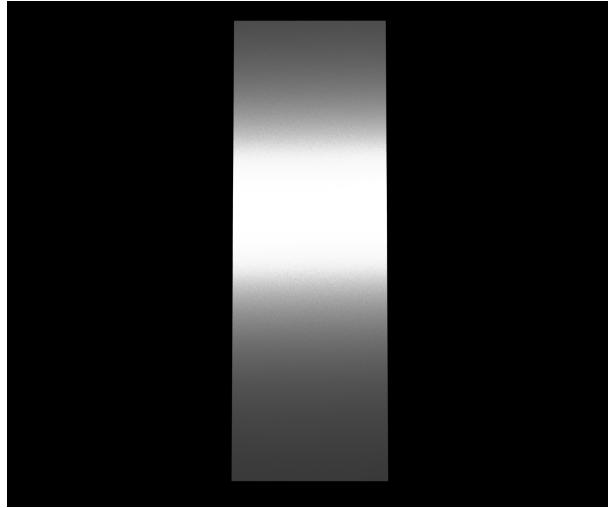


Figura 21: Imagen en blanco y negro de la placa de *primer* simulada en **PBRRender** con intensidad lumínica a 0.9 y un tiempo de exposición de 0,019s.

7.2.3. Recreación del *clearcoat* negro

Para esta placa (Figura 20) empleamos los siguientes valores de los parámetros de nuestro material:

- *BaseColor* = (0, 0, 4)
- *Roughness* = 0,421
- *Metallic* = 0,939
- *Clearcoat* = 0,926
- *ClearcoatRoughness* = 0,043

El ligero azul en el color se debe a que la placa original tenía un leve tono azul. Para el resto de parámetros se repite el mismo razonamiento. Podemos concluir que todos las placas de *clearcoat* se emulan con unos parámetros de material muy similares, como era de esperar.

7.2.4. Recreación del *primer*

Para esta placa (Figura 21) empleamos los siguientes valores de los parámetros de nuestro material:

- *BaseColor* = (26, 30, 10)
- *Roughness* = 0,2
- *Metallic* = 0,942
- *Clearcoat* = 0,987
- *ClearcoatRoughness* = 0,283

A diferencia de los anteriores materiales, este no es *clearcoat*, por lo que no empleamos el mapa de normales para simular la piel de naranja.

El *primer* tiene un color verde oscuro que imitamos en nuestro parámetro color base. Aquí se puede observar como con una rugosidad relativamente baja en ambas capas juntamos la especularidad de la capa base con la de la capa de recubrimiento, además de una metalicidad muy alta, para saturar el haz especular lo máximo posible en el mayor área posible.

7.3. Comparación

Una vez han sido capturadas las muestras y han sido recreadas tan fielmente como ha sido posible en nuestra solución, el siguiente y último paso es comparar ambas imágenes y así validar el grado de fotorrealismo de **PBRRender**.

Por brevedad, en esta sección solo incluiremos las fotografías con mayor luminancia y exposición, y las fotografías con menor luminancia y exposición de cada placa. La comparación entre las 18 fotografías para cada placa se puede encontrar en el [Apéndice B](#).

7.3.1. Comparación del *clearcoat* blanco

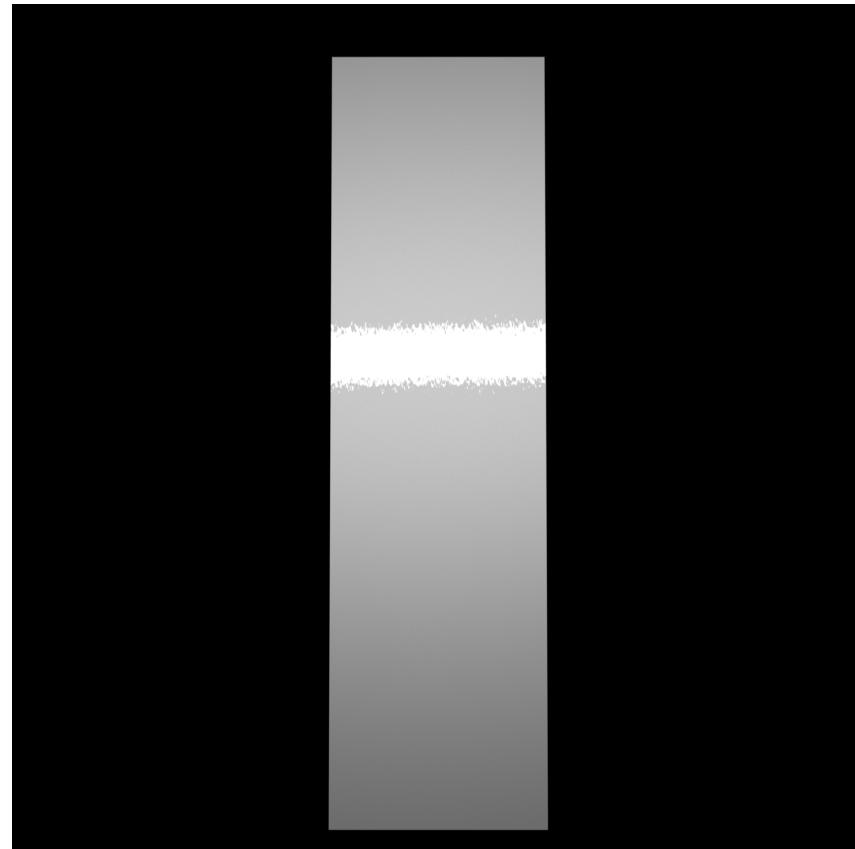
El análisis comparativo de las imágenes de mayor luminancia ([Figura 22](#)) revela una excelente coincidencia en las zonas que no caen dentro del haz especular, donde los colores son casi indistinguibles. Aunque la forma del haz de luz muestra pequeñas diferencias, este es un resultado esperado dada la dificultad de modelar el patrón de imperfecciones único de la placa real. Y podemos observar como el mapa de normales escogido modela más que satisfactoriamente la piel de naranja presente en las placas con *clearcoat*.

Sin embargo, la observación más significativa es que el reflejo en la fotografía real posee una luminosidad marginalmente superior. Aun con esta salvedad, la similitud visual entre la captura y el renderizado es notablemente alta.

No obstante, al observar la comparación en el caso de mínima luminancia ([Figura 23](#)) se aprecia una clara discrepancia, tanto dentro del reflejo como fuera. En la imagen real, la zona fuera del haz tiene un color oscuro constante que prácticamente satura al negro. Por otra parte, en la imagen simulada se puede observar un claro gradiente de brillo desde el haz hacia el exterior, que en ningún momento se aproxima a saturar al negro. Además, el haz de la imagen simulada es, una vez más, ligeramente más brillante que el real.

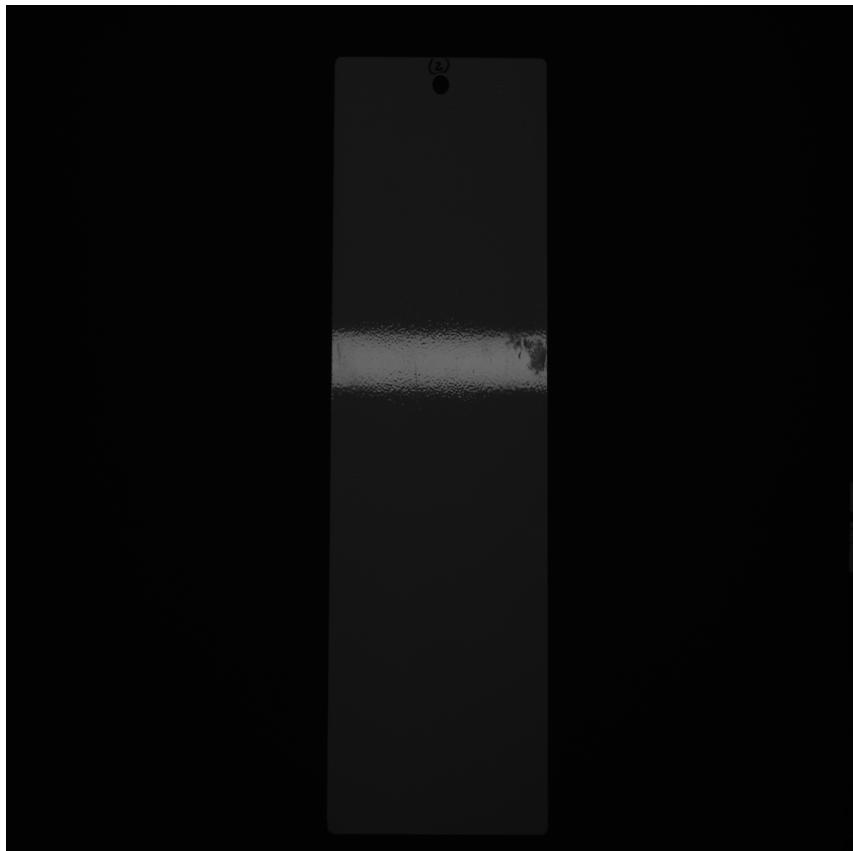


(a) Real

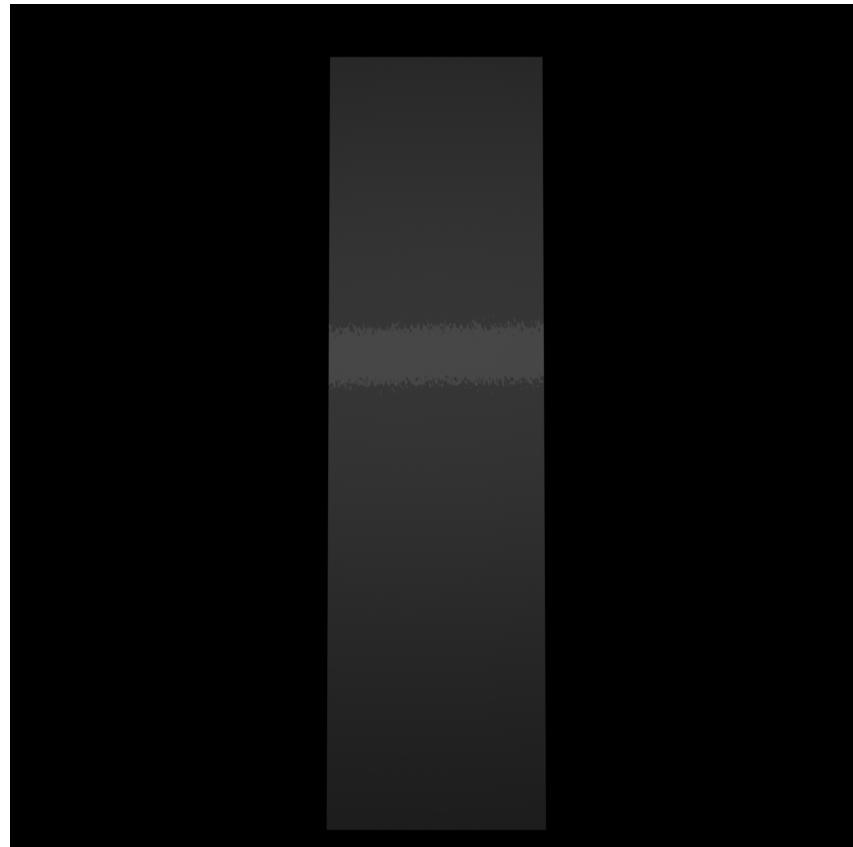


(b) Simulado

Figura 22: Comparación de la placa de *clearcoat* blanca con intensidad lumínica de 0.9 o 90 y un tiempo de exposición de 0,004s



(a) Real



(b) Simulado

Figura 23: Comparación de la placa de *clearcoat* blanca con intensidad lumínica de 0.2 o 20 y un tiempo de exposición de 0,001s

7.3.2. Comparación del *clearcoat* gris

Observamos, al estudiar ambas imágenes de mayor brillo (Figura 24) que ambos colores de la imagen presentan una muy buena coincidencia, tanto el del reflejo como el del *off-specular*. Además, la forma del haz es prácticamente idéntica. El único elemento no replicado son las huellas o manchas que presenta la placa real, pero replicar esos fenómenos no entra dentro de los objetivos de este proyecto por el momento, por lo que este es un resultado excelente.

Pero, de nuevo, si se analizan los fotogramas de menor brillo (Figura 25), se observan las mismas flaquesas en el caso del *clearcoat* blanco. Un gradiente demasiado brillante fuera del haz, cuando en el caso real se trata de un color más próximo al negro puro y constante. Y un haz especular muy poco brillante. Afortunadamente, la forma del haz se mantiene consistentemente precisa.

Al inclinar esta placa, se observan las mismas fortalezas y debilidades (Figura 26). Aunque podemos notar que la perdida de luminosidad a medida que disminuye la elevación es consistente entre la escena real y la virtual.

7.3.3. Comparación del *clearcoat* negro

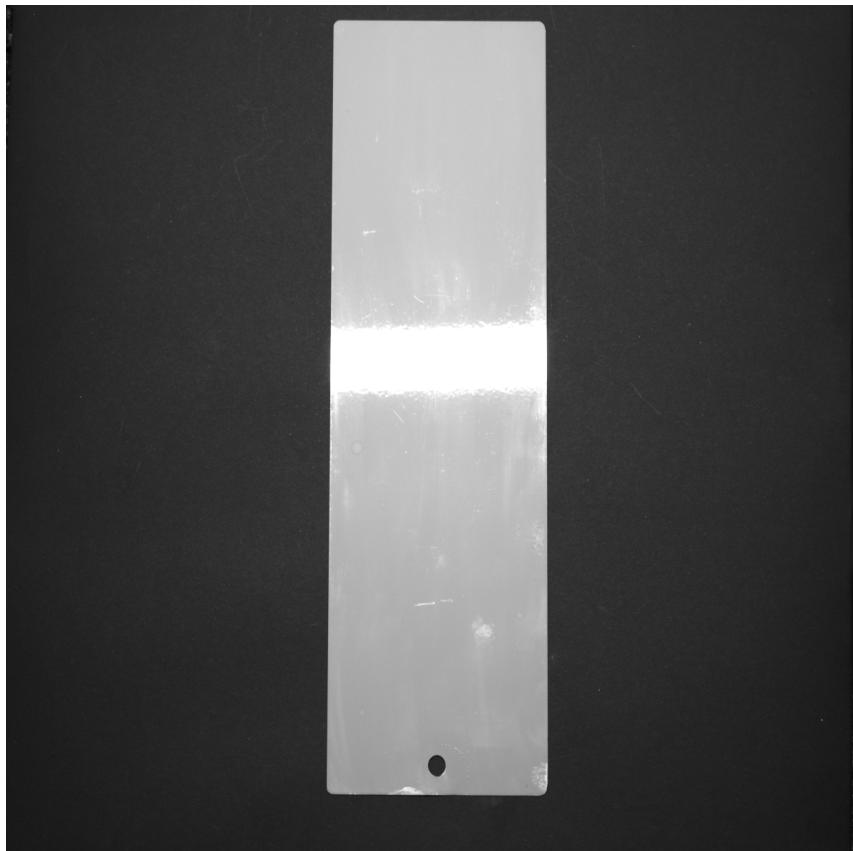
Observamos que en esta ocasión, la disparidad entre la cartulina negra del mundo real y las propiedades perfectas de absorción de luz del suelo de la escena virtual dificultan la tarea de comparar el color de la zona *off-specular*, en el caso de las fotografías con mayor luminancia (Figura 27). En cuanto al reflejo, presenta las mismas fortalezas que el *clearcoat* gris.

Al analizar las cuatro imágenes a la vez, podemos observar como la imagen de máxima luminancia simulada casi se asemeja más a la de mínima luminancia real, principalmente por el suelo de la escena virtual. Nuevamente, el haz especular de la de menor luminancia simulada es demasiado poco brillante, aunque en este caso el color del área que no refleja es muy similar entre la imagen real y la simulada para este caso de mínimo brillo.

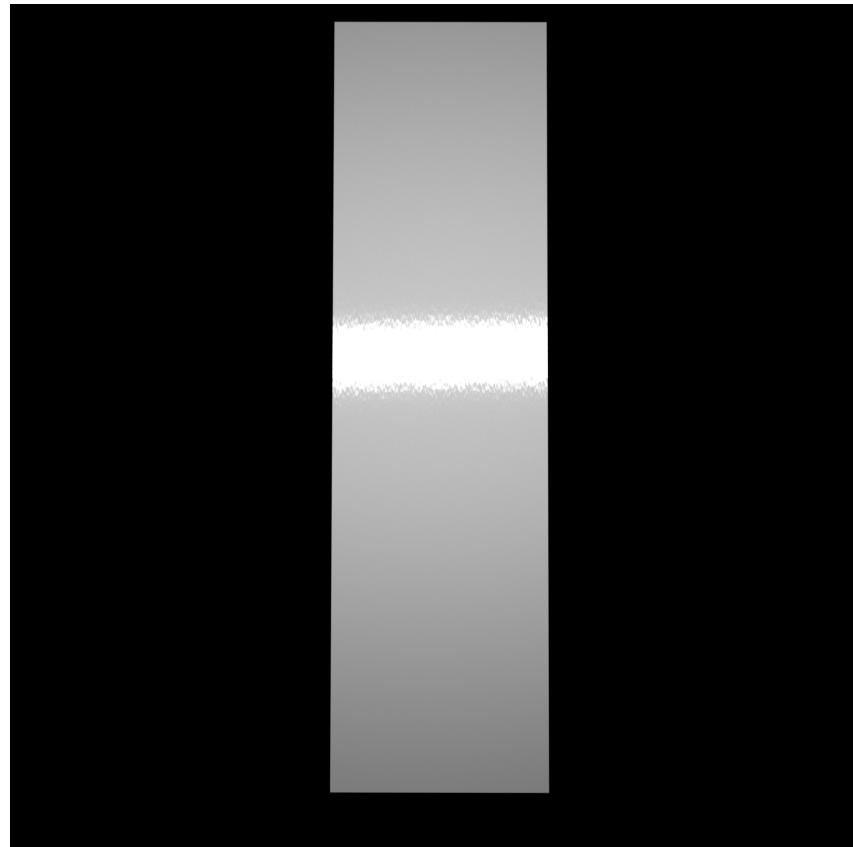
7.3.4. Comparación del *primer*

En el caso de brillo alto (Figura 29), determinamos que la principal diferencia entre la imagen real y la simulada es la ausencia de textura en la segunda. Al tratarse de *primer*, la textura no parece piel de naranja, sino que resulta ser manchas o huellas. Por ello no ha sido simulada. A parte de este hecho, la zona saturada de la imagen real es de ligeramente mayor tamaño que la de la imagen recreada. Y el color de la zona que no se encuentra reflejando es marginalmente demasiado oscuro en los extremos. Por otro lado, encontramos que tanto la imagen simulada como la real presentan extensiones longitudinales del haz especular en forma de gradientes en los laterales.

Como en todos los casos, en las imágenes de brillo menor (Figura 30) el haz especular de la imagen simulada es significativamente menos brillante de lo esperado, aunque se pueda observar que tienen una forma muy similar. En ambos fotogramas, el color del área *off-specular* es un gradiente que se extiende hacia los extremos, pero en el caso de la imagen real, este gradiente llega al negro.

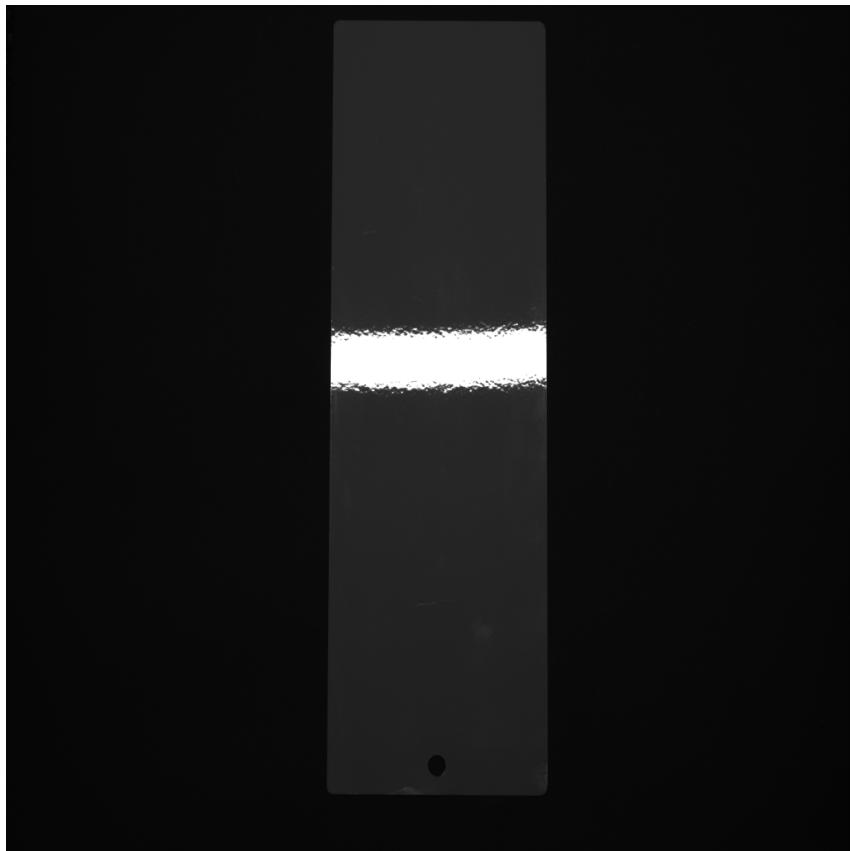


(a) Real

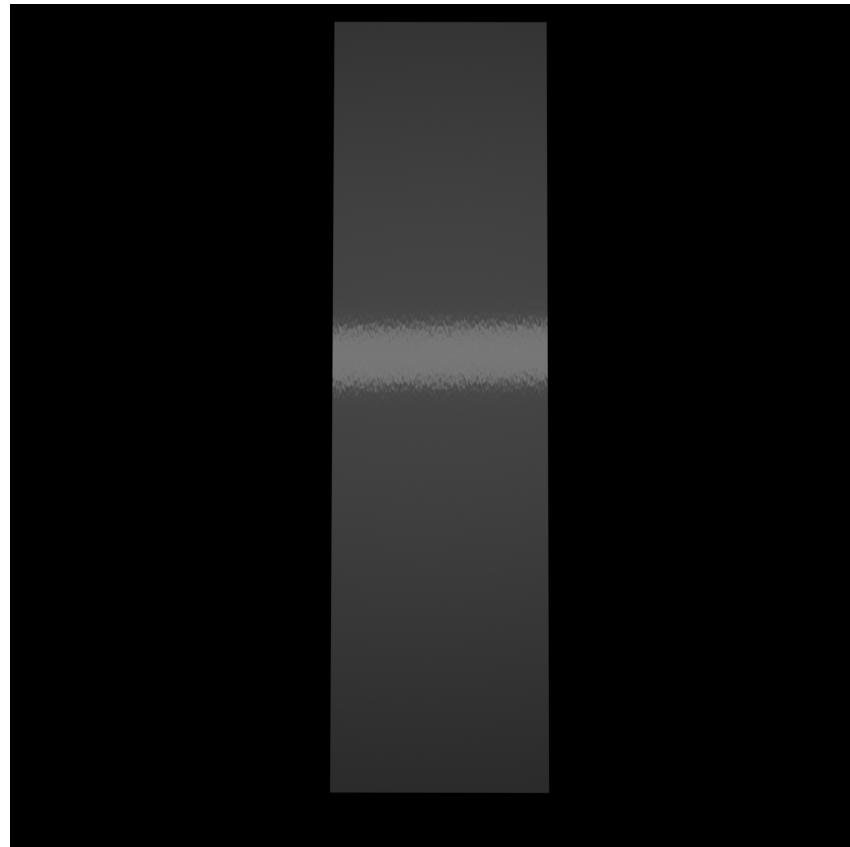


(b) Simulado

Figura 24: Comparación de la placa de *clearcoat* gris con intensidad lumínica de 0.9 o 90 y un tiempo de exposición de 0,011s

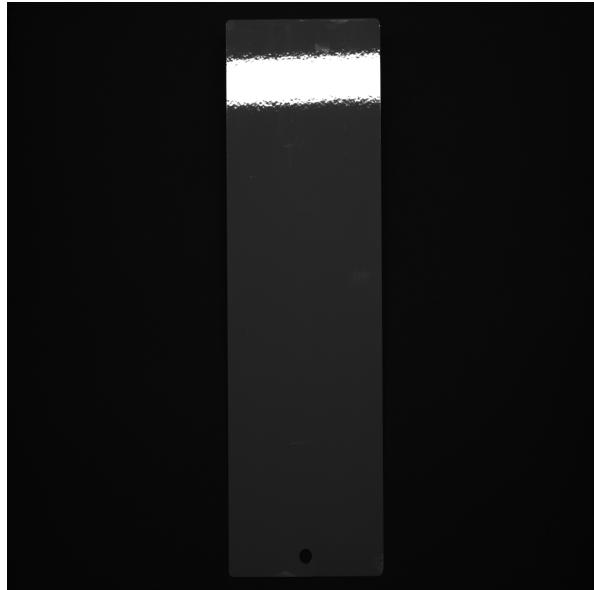


(a) Real

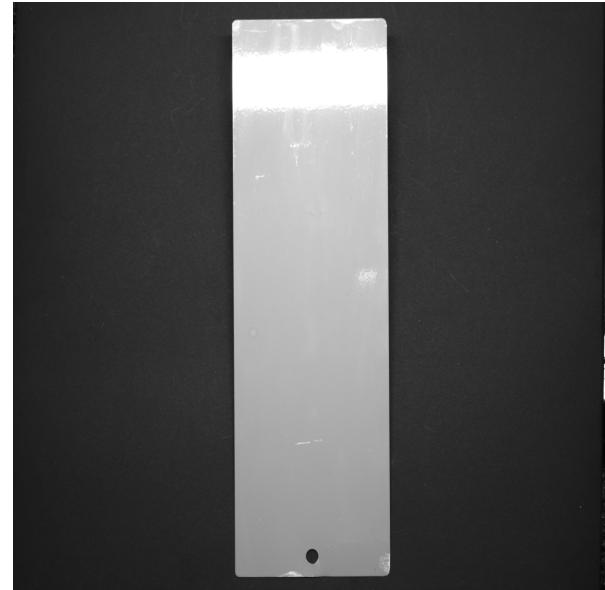


(b) Simulado

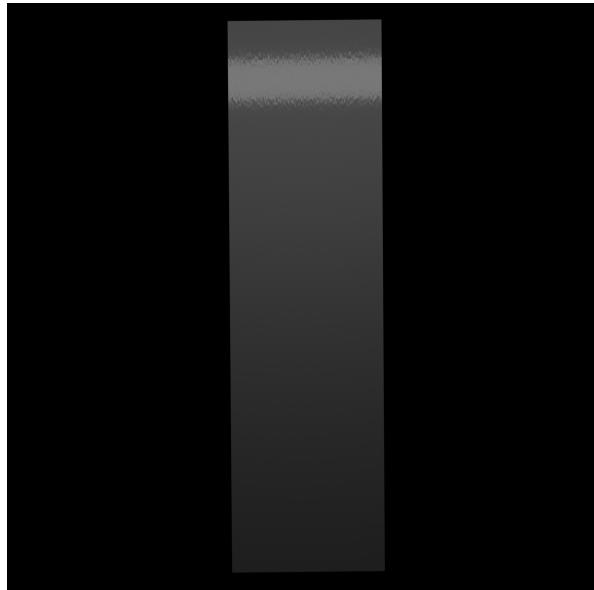
Figura 25: Comparación de la placa de *clearcoat* gris con intensidad lumínica de 0.2 o 20 y un tiempo de exposición de 0,005s



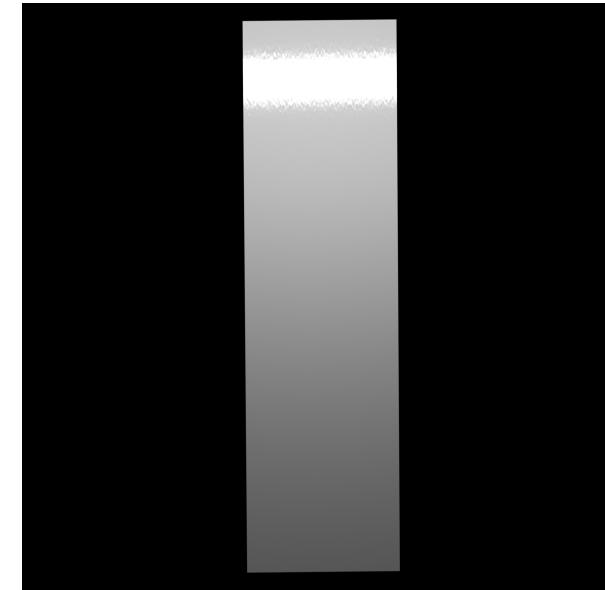
(a) Real



(b) Real

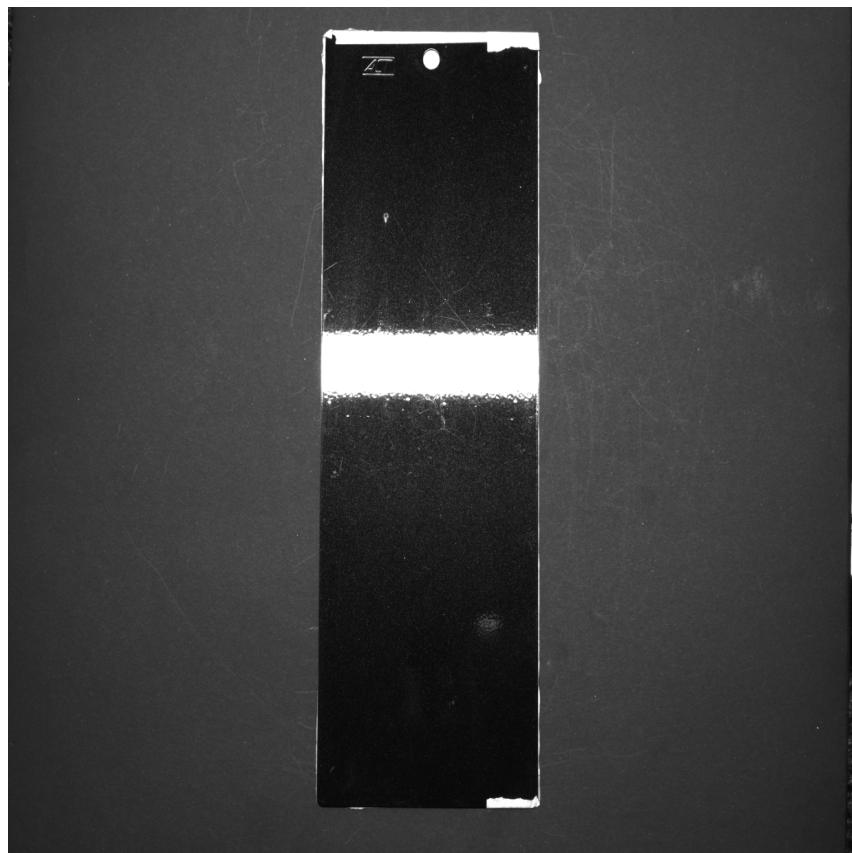


(c) Simulado

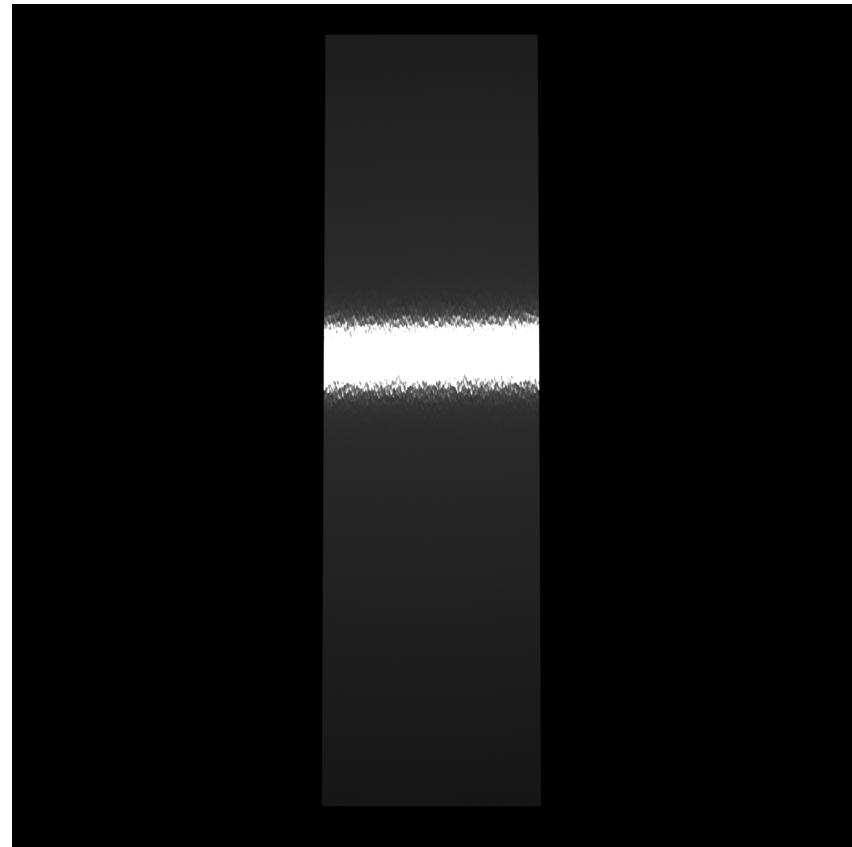


(d) Simulado

Figura 26: Comparación de la placa de *clearcoat* gris inclinada 6° con intensidad lumínica de 0.2 o 20 y un tiempo de exposición de 0,005s a la izquierda. Y intensidad lumínica de 0.9 o 90 y un tiempo de exposición de 0,011s a la derecha.



(a) Real

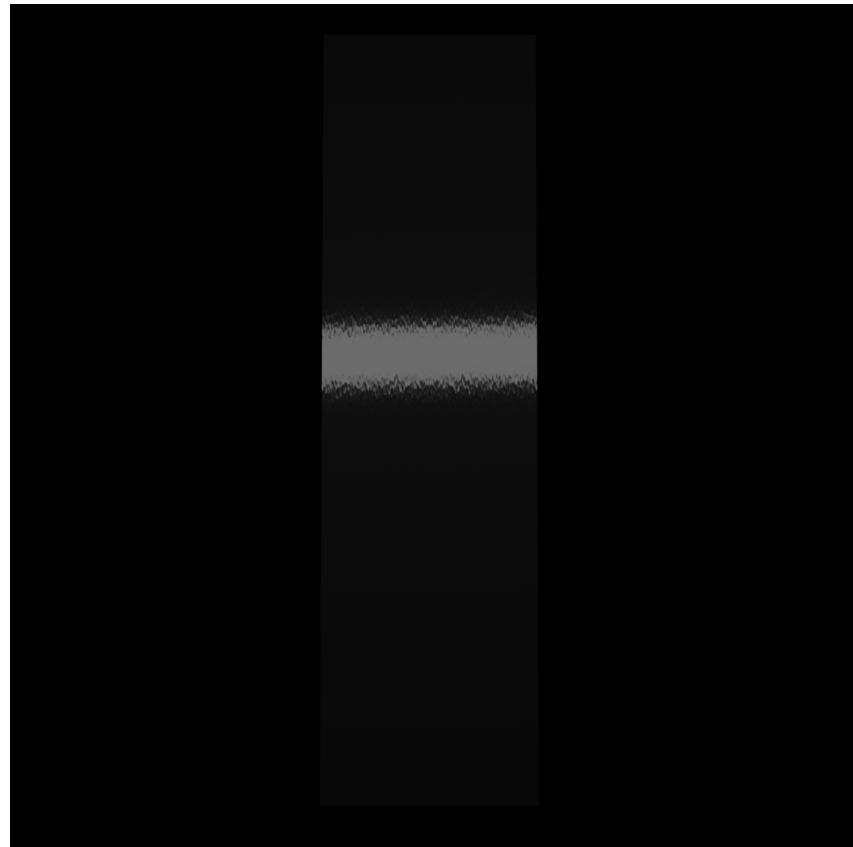


(b) Simulado

Figura 27: Comparación de la placa de *clearcoat* negro con intensidad lumínica de 0.9 o 90 y un tiempo de exposición de 0,016s

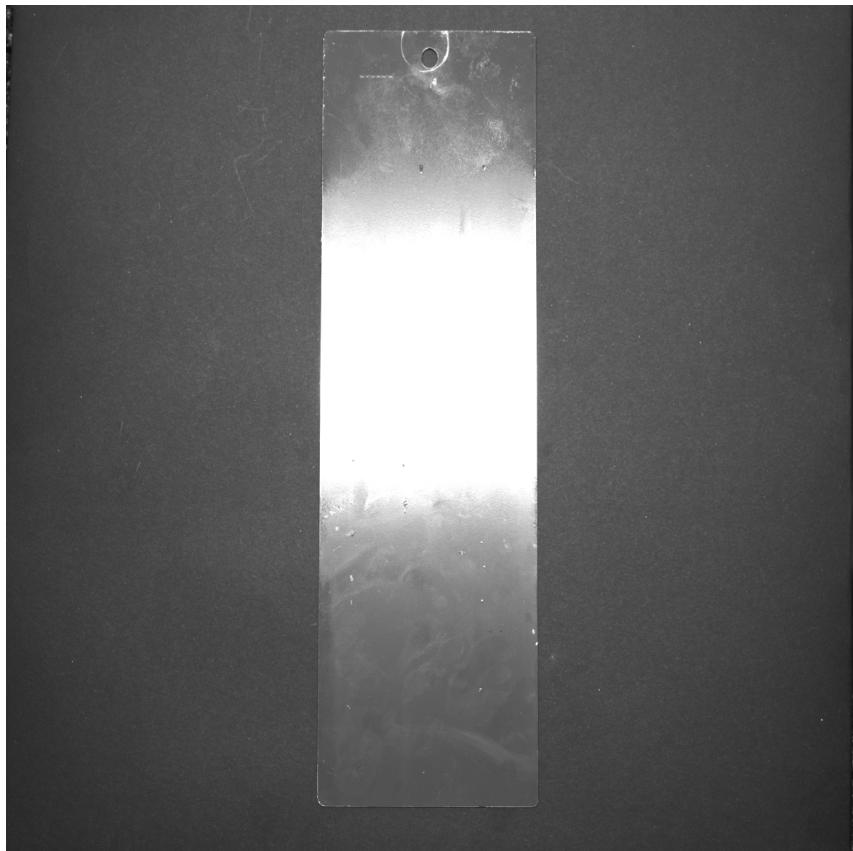


(a) Real

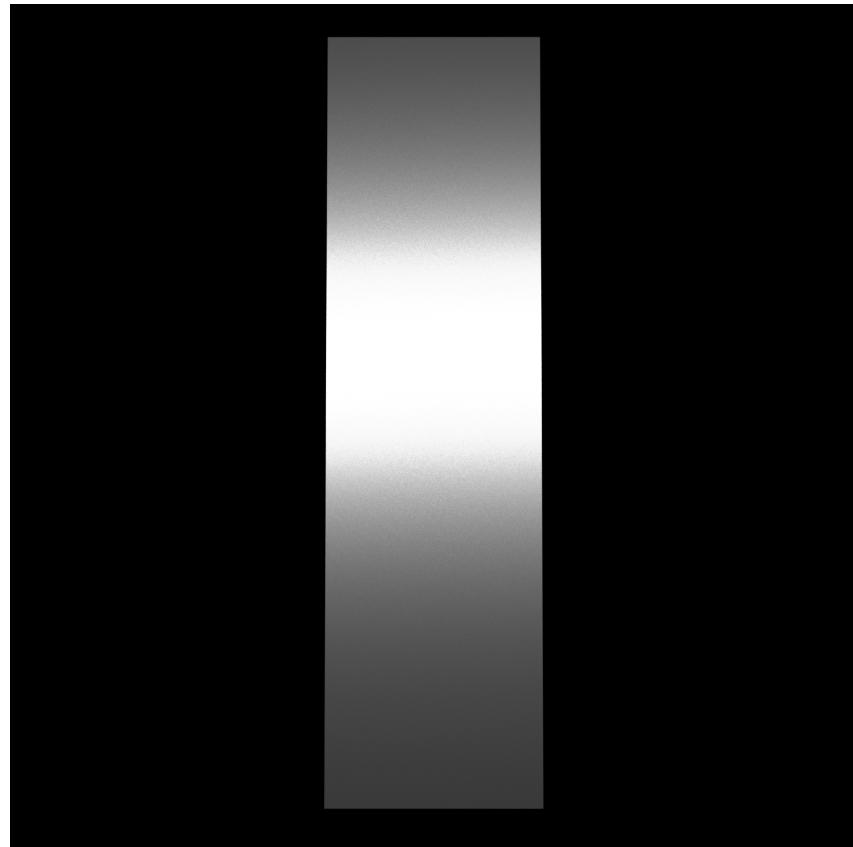


(b) Simulado

Figura 28: Comparación de la placa de *clearcoat* gris con intensidad lumínica de 0.2 o 20 y un tiempo de exposición de 0,006s

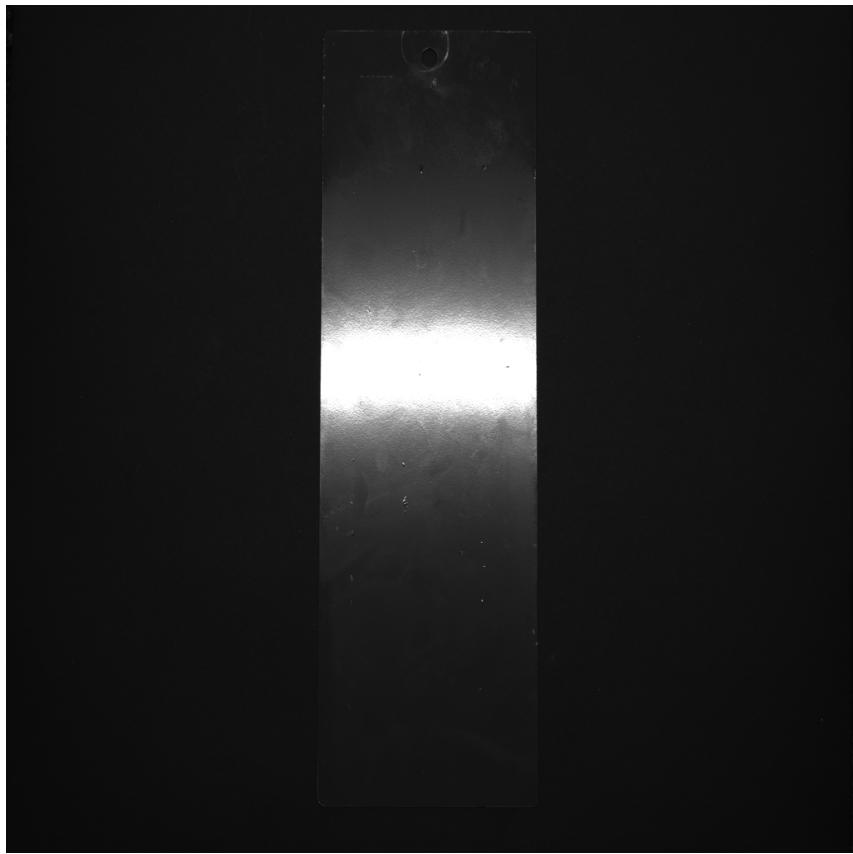


(a) Real

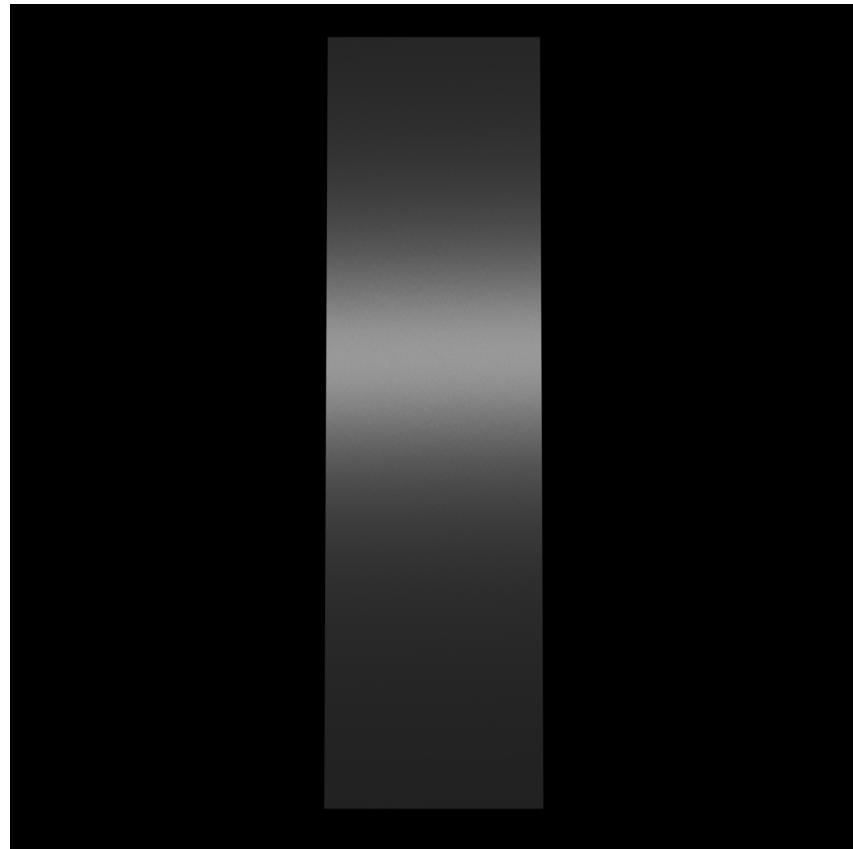


(b) Simulado

Figura 29: Comparación de la placa de *primer* con intensidad lumínica de 0.9 o 90 y un tiempo de exposición de 0,019s



(a) Real



(b) Simulado

Figura 30: Comparación de la placa de *clearcoat* gris con intensidad lumínica de 0.2 o 20 y un tiempo de exposición de 0,01s

7.3.5. Conclusión

Tras analizar detalladamente tanto las imágenes generadas como las capturadas, hemos podido observar que las imágenes de mayor brillo, las usadas para configurar los materiales, presentaban, por lo general, buenos resultados. Pero, al cambiar las condiciones de exposición o intensidad luminosa, el brillo de las generadas no era consistente con el de las capturadas. Las áreas oscuras tendían a ser demasiado brillantes y las brillantes tendían a ser demasiado oscuras. Además, incluso en los fotogramas generados de máxima luminancia, el haz especular era propenso a ser marginalmente menos brillante de lo esperado.

Esta discrepancia se podría atribuir a la implementación del ajuste de exposición o la potencia de las fuentes de luz. Nosotros nos decantamos por la primera por dos razones:

- La metodología para convertir el flujo lumínico de las fuentes de luz en radiancia es robusta, ya que se fundamenta en fuentes fiables y ha sido validada. Por el contrario, el modelo de ajuste de exposición implementado presenta incertidumbres significativas por dos motivos. Primero, está concebido para cámaras analógicas de película, mientras que en nuestro caso empleamos cámaras de sensor digital. Y segundo, depende de una constante de calibración empírica ($K = 2,81$) cuyo origen no pudimos verificar, al no tener acceso a la fuente bibliográfica original citada.
- Como se puede observar en la [Figura 31](#), al decrementar la exposición, la luminosidad no disminuye lo suficiente en las imágenes recreadas en comparación con las capturadas. Como consecuencia, hay inconsistencias tanto en la zona especular como en la *off-specular*. Por otro lado, si se analiza la [Figura 32](#), al reducir la radiancia de los segmentos, tan solo hay inconsistencias en el brillo del reflejo. La luminosidad del área fuera del reflejo disminuye con la misma cadencia tanto en el caso generado como en el real. Esto sugiere una implementación del ajuste de exposición problemática.

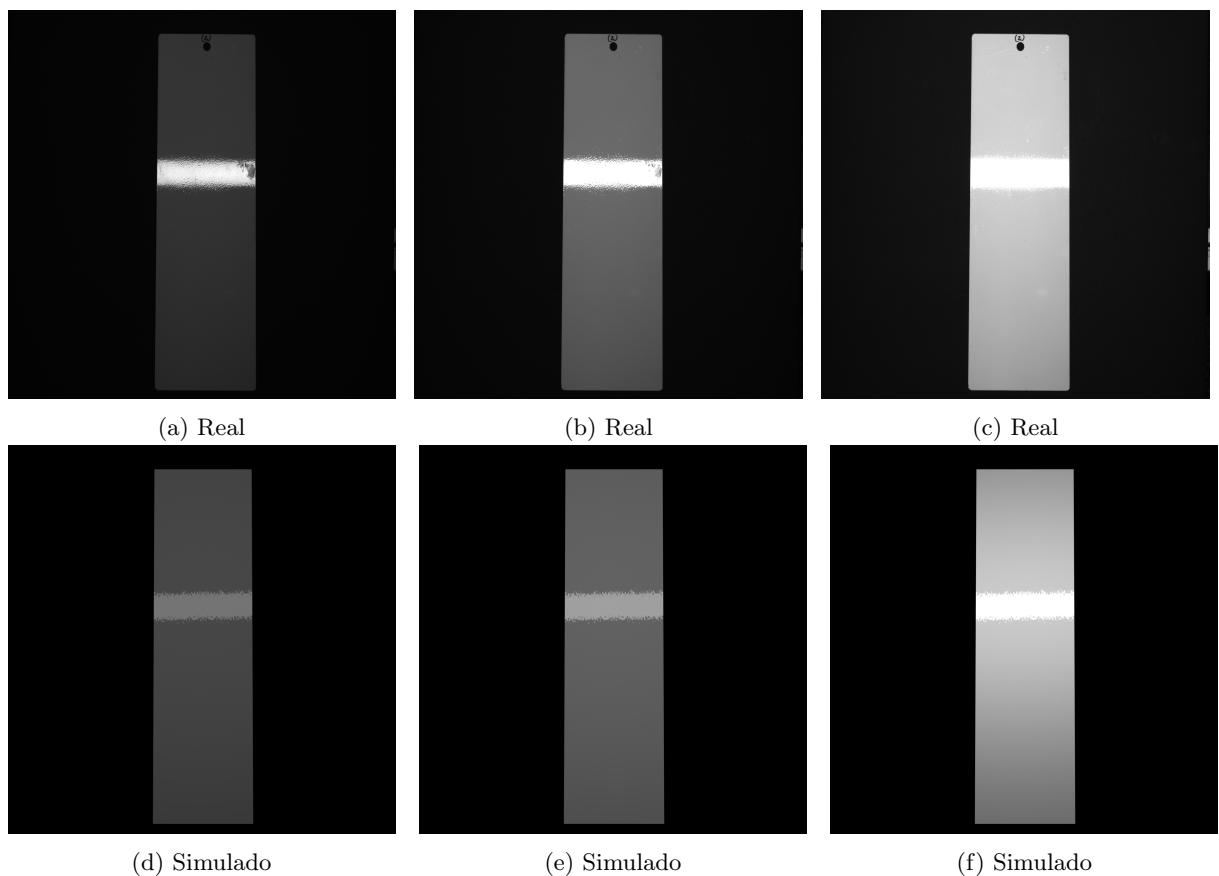


Figura 31: Comparación del efecto de cambiar la exposición con la misma placa de *clearcoat* blanco y con una intensidad luminosa de 0.9 o 90. El tiempo de exposición es, de izquierda a derecha, 0,001s, 0,002s, 0,004s.

Ahora, centrándolo en los logros del proyecto, listaremos los éxitos más destacables de la solución implementada:

- La posición y forma del haz especular en los fotogramas renderizados se ha mantenido fiel al observado en las capturas, en todos los casos.

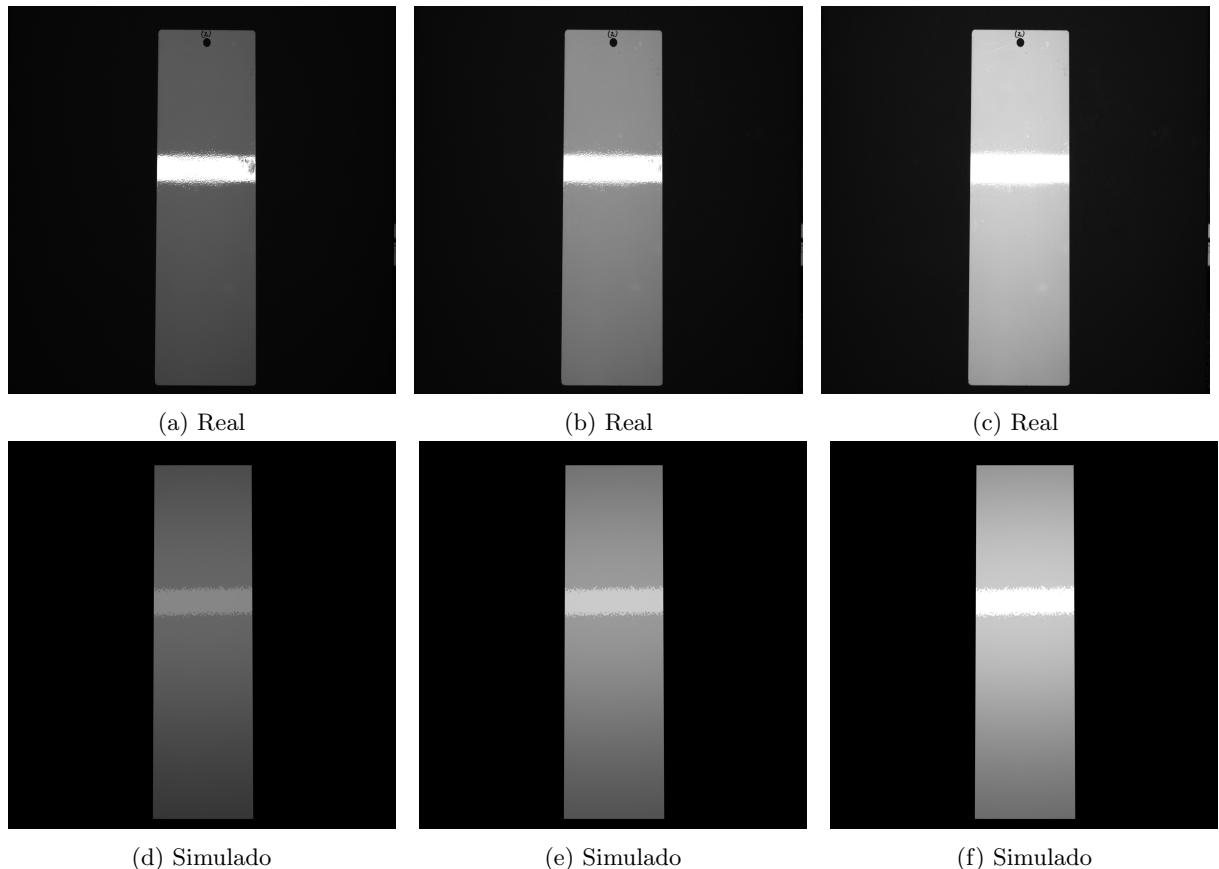


Figura 32: Comparación del efecto de cambiar la intensidad de la luz con la misma placa de *clearcoat* blanco y con un tiempo de exposición de 0,004s. La intensidad luminosa es, de izquierda a derecha, 0.2 o 20, 0.5 o 50, 0.9 o 90.

- El sistema de materiales ha demostrado una gran versatilidad, logrando al menos un buen resultado para cada una de las placas simuladas, lo que confirma la flexibilidad de la solución propuesta.
- La solución ha probado a ser robusta ante transformaciones del modelo de la superficie a inspeccionar, manteniendo la coherencia y fidelidad de la simulación frente a cambios en la posición y orientación de las placas de prueba.
- Se ha logrado replicar con éxito fenómenos visuales complejos y específicos, como la particular forma del reflejo especular observado en la placa de *primer* o la piel de naranja característica de los recubrimientos de carrocerías.

8. Conclusión

A lo largo de este Trabajo de Fin de Grado, se ha abordado el desafío industrial de generar datos sintéticos fotorrealistas para la automatización del control de calidad. La respuesta a esta necesidad se ha materializado en el diseño y la implementación de **PBRRender**, una librería de renderizado basada en física construida sobre el SDK de *Radeon ProRender*, concebida desde su inicio para una futura integración en el *software* de simulación de la empresa.

La validación del trabajo realizado, mediante la comparación de capturas del mundo real con imágenes sintéticas que trataban de replicarlas, ha resultado mixta. Por un lado se han conseguido replicar con éxito muchos de los fenómenos observados en las fotografías. Pero, dado un fallo crítico en la simulación de las cámaras, muchas de las imágenes generadas tienen niveles de brillo que no se asemejan al real.

Seguidamente procederemos a analizar cada uno de los objetivos propuestos al inicio de este Trabajo de Fin de Grado para determinar su grado de consecución.

1. **Sintetizar imágenes generadas mediante gráficos por computador que reproduzcan fielmente fotografías tomadas en el mundo real.** Pese haber obtenido múltiples resultados excelentes en la etapa de validación, hay suficientes imágenes renderizadas cuyo grado de fotorrealismo no es satisfactorio. Específicamente, al compararlas con las fotografías que tratan de replicar, observamos que el nivel de luminosidad de estas imágenes no es el adecuado, como ya se ha discutido extensivamente en la sección anterior. En vista de los resultados, este objetivo requiere trabajo adicional y debe considerarse, por tanto, como **formalmente incumplido en su estado actual**.
 - a) **Generar imágenes que simulen con exactitud capturas de carrocerías bajo los túneles de inspección.** La validación empírica de este objetivo no pudo realizarse, ya que limitaciones logísticas y técnicas impidieron el acceso a los túneles de inspección y carrocerías reales. Por lo tanto, no es posible emitir un veredicto formal sobre su cumplimiento basado en pruebas directas. No obstante, dado que este es un caso de aplicación específico del objetivo principal, el cual hemos declarado como formalmente incumplido, es lógico inferir que este objetivo tampoco se habría satisfecho. Por extensión, **se considera teóricamente incumplido**, a falta de una validación empírica que lo corrobore.
2. **Diseñar una solución fácilmente integrable en el simulador existente.** Desde el primer momento hemos abordado esta solución como una interfaz creada para ser empleada en el *software* de simulación de la empresa.
 - El lenguaje con el que está desarrollada e implementada nuestra solución es *C++*, el mismo en el que está construido el simulador. De esta manera evitamos la complejidad y la sobrecarga de rendimiento asociadas a la creación de puentes entre lenguajes. La integración se realiza a nivel nativo lo que simplifica tanto el proceso de compilación como la depuración.
 - **PBRRender** reutiliza muchas de las librerías ya utilizadas en el *software* de *AUTIS*, como *GLM*. De esta manera, la única dependencia externa que añade nuestra solución es el *SDK* de *Radeon ProRender*.
 - Empleamos *OpenGL* para postprocesar la imagen y con *renderPreview()* devolvemos el fotograma generado como una textura de *OpenGL*. El simulador emplea esta misma *API* gráfica, por lo que puede manejar directamente esta textura sin tener que resubirla a la GPU bajo otra *API*. Este es un enfoque eficiente que mejora la tasa de fotogramas por segundo de la previsualización en tiempo real, donde el rendimiento es clave.
 - En el simulador los arcos de luz de los túneles se definen como un conjunto de mallas idénticas. La solución desarrollada define también sus fuentes de luz como mallas con materiales emisivos, por lo que se puede emplear la misma definición para **PBRRender** sin necesidad de traducción. Además, **PBRRender** emplea instancias de *RPR* para optimizar el consumo de memoria y el rendimiento cuando se hace uso de una misma malla múltiples veces en una escena, como es el caso de los arcos de luz del simulador.
 - Las cámaras de nuestra interfaz se definen mayormente con los mismos parámetros que las cámaras del *software* de simulación. Los únicos parámetros que añade nuestra solución que no se encuentran en las cámaras del simulador son la sensibilidad y el tamaño del sensor.

Dada todas estas razones, **damos este objetivo como cumplido, al menos teóricamente**. Matizamos con teóricamente porque esta interfaz aún no ha sido integrada en el simulador. En consecuencia, hasta que no se realice esta tarea no se le podrá otorgar un veredicto firme a este objetivo.

3. **Soportar renderizado en tiempo real y *offline*.** Ofrecemos a los usuarios una previsualización preparada para el tiempo real mediante `renderPreview()` que emplea *HybridPro*. Así, se les permite modificar la escena y recibir retroalimentación instantánea. Una vez ha terminado el ajuste, se puede llamar a `renderFinal()`, que renderizará una imagen final de un grado superlativo de fotorrealismo con *Northstar*.

Por consiguiente, **concluimos que este objetivo ha sido alcanzado en mayor medida**. El único matiz en nuestra solución se debe a las limitaciones de *HybridPro*, como los artefactos en la iluminación o la ausencia de mapas de normales, que en ocasiones impiden que la previsualización sea un predictor exacto de la imagen final.

4. **Permitir al usuario definir la escena virtual de manera robusta e intuitiva.** Para evaluar el estado de este objetivo analizaremos si se han cumplido los sub-objetivos que lo componen.

- a) **Exponer parámetros intuitivos y versátiles de control del material que compone la superficie.** Nuestro sistema de materiales cuenta la posibilidad de simular fenómenos característicos de la pintura de automóviles, como la piel de naranja, mediante mapas de normales. También dispone de un conjunto de parámetros de configuración que, durante nuestras pruebas de validación, nos han otorgado la capacidad de recrear los materiales de todas las muestras tanto rápidamente como correctamente. De tal manera, se ha demostrado que nuestra solución es intuitiva a la par que versátil, y por consiguiente, **declaramos este objetivo como cumplido**.
- b) **Adaptar la configuración de las fuentes de luz de la escena virtual a las condiciones específicas de los túneles.** Como hemos discutido también al analizar el segundo objetivo, hemos empleado mallas emisoras como fuentes de luz. De esta manera, se pueden definir luces con exactamente la misma forma y superficie que los arcos de luz de los túneles. Además, hemos diseñado un algoritmo para poder hacer uso del valor real de potencia lumínica de los segmentos de luz de la empresa en nuestras simulaciones. Una vez más, como ha quedado demostrado mediante la precisión del haz espectral discutida en la [Subsección 7.3](#), **hemos cumplido este objetivo**.
- c) **Simular fielmente las cámaras empleadas en los túneles.** La simulación de las cámaras supone nuestra principal fuente de error (véase [Subsección 7.3](#)). Una implementación pobre de la exposición ha terminado desencadenando en la inhabilidad de cumplir nuestro principal objetivo de simular fielmente capturas del mundo real. Este objetivo se **declara categóricamente como incumplido**.

Una vez más, las cámaras son el eslabón débil en la cadena. Dado que no se han cumplido todos los sub-objetivos, **declaramos este objetivo como incumplido**.

8.1. Trabajos futuros

Si bien la consecución del objetivo principal de fotorrealismo absoluto no se ha alcanzado en su totalidad, los resultados obtenidos en este Trabajo de Fin de Grado son de gran valor. Los numerosos casos en los que se ha logrado una correspondencia visual casi perfecta entre la captura real y la imagen sintética demuestran que la arquitectura propuesta y las tecnologías seleccionadas son las adecuadas. El proyecto ha sentado con éxito una base funcional y robusta.

De aún mayor importancia es el hecho de que hemos logrado diagnosticar el principal impedimento a la recreación fiel de las imágenes capturadas, la errónea implementación del ajuste de exposición de la cámara. Este hallazgo nos permite trazar una hoja de ruta clara para la evolución del proyecto hacia el cumplimiento de todos los objetivos propuestos.

Por tanto, el trabajo realizado no debe considerarse un fin, sino el primer paso fundamental de un proceso iterativo. Las siguientes líneas de trabajo construyen sobre estos sólidos cimientos con el objetivo de perfeccionar el sistema, ampliar las capacidades de nuestra solución y enfrentarla a pruebas más exigentes y específicas. Todo ello con el fin último de cumplir con las necesidades del caso de uso industrial. A continuación, se detallan las tareas prioritarias a realizar.

- El principal trabajo a realizar consiste en corregir la incorrecta implementación de la exposición, que ha sido ya mencionada extensivamente. Con este fin, la mejor opción disponible es emplear otro *SDK* de *AMD* y *GPUOpen*, *Radeon Image Filters*. Se trata de una librería que implementa una gran cantidad de filtros de postprocesado populares, entre los que se incluyen el *Photographic Tone Mapper* [117].

Este filtro toma como entrada parámetros como los *f-stops*, sensibilidad, tiempo de exposición, distancia focal y gamma. Con ellos emula la funcionalidad de una cámara con esos mismos valores, realizando el ajuste de exposición y la corrección gamma [118]. Este ajuste de exposición desarrollado por una empresa líder en la industria representa una alternativa robusta y validada frente a nuestra implementación plagada de incertidumbres.

La única incertidumbre que surge con esta librería es la viabilidad de emplearla en un entorno de tiempo real, ya que por su diseño basado en colas [119] parece estar orientada a procesado de imagen *offline*.

- Otra adición prioritaria debería ser implementar *bloom*, un efecto de postprocesado que produce franjas (o plumas) de luz que se extienden desde los bordes de las zonas brillantes de una imagen, contribuyendo a la ilusión de una luz extremadamente brillante que inunda la cámara o el ojo que capta la escena [120]. Como se puede discernir por su descripción, con este filtro resolveríamos el problema de que incluso en las imágenes con máximo brillo, el haz especular no se observe con el mismo brillo superlativo que el captado por la cámara. Por suerte, *Radeon Image Filters* también incluye soporte para este efecto de postprocesado [120].
- Eventualmente, se deberían modelar los *flakes* para emular fielmente los recubrimientos metálicos empleados en automóviles. Estos son copos metálicos que se mezclan con colorantes transparentes y se suspenden en un aglutinante para formar la capa de pintura. A nivel microscópico y bajo iluminación directa, los *flakes* individuales generan un efecto de destello similar a la purpurina, cuya intensidad depende directamente de su tamaño y distribución. *Flakes* más grandes y gruesos producen un destello intenso, mientras que *flakes* muy finos dan lugar a un acabado más uniforme y similar a un color sólido [92].
Simular este efecto es muy complejo [121] y se debería de estudiar si *Radeon ProRender* está siquiera capacitado ello.
- Las pruebas de validación realizadas fueron con muestras de pintura de coche en forma de placas, en un dispositivo de captura especializado. Para cumplir con uno de los objetivos propuestos, se deberían de repetir estas pruebas con carrocerías reales en túneles reales.
- Recordemos que el motor de renderizado basado en la física es tan solo el primer paso de un sistema más grande de generación de muestras para entrenar un modelo de visión por computador. La siguiente tarea a desarrollar después del motor es simular anomalías o imperfecciones en las superficies de las carrocerías, posiblemente empleando mapas de normales.

Referencias

- [1] Matt Pharr, Wenzel Jakob y Greg Humphreys. «Introduction». En: *Physically based rendering: From theory to implementation*. The MIT Press, 2023. URL: <https://pbr-book.org>.
- [2] Harolf Frederic Walton y Jorge Reyes. *Analisis Quimico e Instrumental Moderno*. Reverte, 1983.
- [3] Tomas Akenine-Möller, Eric Haines y Naty Hoffman. «Physically Based Shading». En: *Real-time rendering*. CRC Press, 2018.
- [4] Matt Pharr, Wenzel Jakob y Greg Humphreys. «Radiometry». En: *Physically based rendering: From theory to implementation*. The MIT Press, 2023. URL: <https://pbr-book.org>.
- [5] Tomas Akenine-Möller, Eric Haines y Naty Hoffman. «Light and Color». En: *Real-time rendering*. CRC Press, 2018.
- [6] Matt Pharr, Wenzel Jakob y Greg Humphreys. «Surface Reflection». En: *Physically based rendering: From theory to implementation*. The MIT Press, 2023. URL: <https://pbr-book.org>.
- [7] Matt Pharr, Wenzel Jakob y Greg Humphreys. «Photorealistic Rendering and the Ray-Tracing Algorithm». En: *Physically based rendering: From theory to implementation*. The MIT Press, 2023. URL: <https://pbr-book.org>.
- [8] Newcastle University. *Dot Product*. URL: <https://www.ncl.ac.uk/webtemplate/ask-assets/external/mathematics-resources/core-mathematics/vectors/dot-product.html>.
- [9] Jakub Boksansky. *Crash Course in BRDF Implementation*. <https://boksajak.github.io/blog/BRDF>. Feb. de 2021.
- [10] Natty Hoffman. «Fresnel Equations Considered Harmful». En: *Eurographics* (2019).
- [11] Matt Pharr, Wenzel Jakob y Greg Humphreys. «Roughness Using Microfacet Theory». En: *Physically based rendering: From theory to implementation*. The MIT Press, 2023. URL: <https://pbr-book.org>.
- [12] Tomas Akenine-Möller, Eric Haines y Naty Hoffman. «Environment Light». En: *Real-time rendering*. CRC Press, 2018.
- [13] Wolfram Alpha. *Physically Based Rendering*. URL: <https://reference.wolfram.com/language/tutorial/PhysicallyBasedRendering.html#659862423>.
- [14] Tomas Akenine-Möller, Eric Haines y Naty Hoffman. «Shading Basics». En: *Real-time rendering*. CRC Press, 2018.
- [15] Joey De Vries. «Basic Lighting». En: *Learn opengl: Learn modern opengl graphics programming in a step-by-step fashion*. Kendall & Welling, 2020. URL: <https://learnopengl.com/>.
- [16] Tomas Akenine-Möller, Eric Haines y Naty Hoffman. «Global Illumination». En: *Real-time rendering*. CRC Press, 2018.
- [17] Steve Seitz. *Ray Tracing in 5 minutes!* 2022. URL: https://youtu.be/H5TB217zq6s?si=fBKqnca9tewaDa_4.
- [18] Contribuyentes de Wikipedia. *Path tracing*. URL: https://en.wikipedia.org/wiki/Path_tracing.
- [19] Contribuyentes de Wikipedia. *Cámara Fotográfica*. URL: https://es.wikipedia.org/wiki/C%C3%A1mara_fotogr%C3%A1fica.
- [20] José L. Fernández. *La cámara Fotográfica*. URL: <https://www.fisicalab.com/apartado/camara-fotos>.
- [21] Romain Guy y Mathias Agopian. *Physically Based Rendering in Filament*. URL: <https://github.io/filament/Filament.md.html>.
- [22] NVIDIA. *Renderizado en la GPU NVIDIA Iray*. URL: <https://www.nvidia.com/es-es/design-visualization/iray/>.
- [23] NVIDIA. *NVIDIA Iray Executive Overview*. 2015. URL: https://d29g4g2dyqv443.cloudfront.net/sites/default/files/akamai/designworks/docs/nvidia_iray_overview.150810.A4.pdf.
- [24] NVIDIA. *Iray Rendering Features*. URL: <https://www.nvidia.com/content/dam/en-za/Solutions/design-visualization/documents/provis-iray-features-pdf-1148708-final.pdf>.
- [25] Contribuyentes de Wikipedia. *Spectral Rendering*. URL: https://en.wikipedia.org/wiki/Spectral_rendering.

- [26] Dau Design and Consulting Inc. *Validation of NVIDIA Iray against CIE 171:2006*. 2016. URL: https://d29g4g2dyqv443.cloudfront.net/sites/default/files/akamai/designworks/docs/Validation-of-NVIDIA's-Iray-against-CIE-171_20160217.pdf.
- [27] Comission Internationale de l'Eclairage. «CIE 171:2006 test cases to assess the accuracy of lighting computer programs». En: *CIE* (2006). DOI: 10.25039/tr.171.2006.
- [28] NVIDIA. *Lenguaje de Definición de Materiales de NVIDIA*. URL: <https://www.nvidia.com/es-es/design-visualization/technologies/material-definition-language/>.
- [29] Contribuyentes de Wikipedia. *Unreal Engine*. URL: https://en.wikipedia.org/wiki/Unreal_Engine.
- [30] Epic Games. *Unreal Engine 5*. URL: <https://www.unrealengine.com/en-US/unreal-engine-5>.
- [31] Epic Games. *Lumen Global Illumination and Reflections*. URL: <https://dev.epicgames.com/documentation/en-us/unreal-engine/lumen-global-illumination-and-reflections-in-unreal-engine>.
- [32] Epic Games. *Geometría virtualizada de Nanite*. URL: <https://dev.epicgames.com/documentation/es-es/unreal-engine/nanite-virtualized-geometry-in-unreal-engine>.
- [33] Epic Games. *Using Physical Units in Unreal Engine*. URL: <https://dev.epicgames.com/documentation/en-us/unreal-engine/using-physical-lighting-units-in-unreal-engine>.
- [34] Epic Games. *Cine Camera Actor*. URL: <https://dev.epicgames.com/documentation/en-us/unreal-engine/cinematic-cameras-in-unreal-engine>.
- [35] Epic Games. *Physically Based Materials*. URL: <https://dev.epicgames.com/documentation/en-us/unreal-engine/physically-based-materials-in-unreal-engine>.
- [36] Epic Games. *Layering Materials in Unreal Engine*. URL: <https://dev.epicgames.com/documentation/en-us/unreal-engine/layering-materials-in-unreal-engine>.
- [37] Chaos Group. *Chaos V-Ray*. URL: <https://www.chaos.com/es/vray>.
- [38] Contribuyentes de Wikipedia. *V-Ray*. URL: <https://en.wikipedia.org/wiki/V-Ray>.
- [39] Chaos Group. *Working with V-Ray Scenes*. URL: <https://docs.chaos.com/display/APPSDK/Working+with+V-Ray+Scenes>.
- [40] Chaos Group. *Light Cache GI*. URL: <https://docs.chaos.com/display/THEORY/Light+Cache+GI>.
- [41] Chaos Group. *Brute Force GI*. URL: <https://docs.chaos.com/display/THEORY/Brute+Force+GI>.
- [42] Chaos Group. *Caustics*. URL: <https://docs.chaos.com/display/APPSDK/Caustics>.
- [43] Chaos Group. *VRayMtl*. URL: <https://docs.chaos.com/display/VMAX/VRayMtl>.
- [44] Chaos Group. *VRayPhysicalCamera*. URL: <https://docs.chaos.com/display/VMAX/VRayPhysicalCamera>.
- [45] Chaos Group. *Mesh Light*. URL: <https://docs.chaos.com/display/VMAX/Mesh+Light>.
- [46] Chaos Group. *V-Ray wins Engineering Emmy Award*. URL: <https://www.chaos.com/blog/v-ray-wins-engineering-emmy-award?srsltid=AfmB0ooY4E0L9h2n-oq4HQm-wflfo41YKBWwLnkmzd1mt9RT1L13ixApxU>.
- [47] Chaos Group. *Chaos Group's V-Ray Wins Academy Award*. URL: <https://www.chaos.com/blog/chaos-groups-v-ray-wins-academy-award?srsltid=AfmB0opLceKeEbQ4WjQAEYGFm7d0hATknoCV1EWwrG6nM2LxApU>.
- [48] Advanced Micro Devices Inc. *AMD Radeon ProRender*. URL: <https://www.amd.com/es/products/graphics/software/radeon-prorender.html>.
- [49] Contribuyentes de Wikipedia. *GPUOpen*. URL: <https://en.wikipedia.org/wiki/GPUOpen>.
- [50] Advanced Micro Devices Inc. *rpr_material_node_type*. URL: https://radeon-pro.github.io/RadeonProRenderDocs/en/sdk/info_setting_types/rpr_material_node_type.html.
- [51] Advanced Micro Devices Inc. *About uber shader*. 2022. URL: <https://radeon-pro.github.io/RadeonProRenderDocs/en/uber/about.html>.
- [52] Advanced Micro Devices Inc. *December AMD Radeon ProRender updates – USD, MaterialX, new Materials library, plug-in updates*. URL: <https://gpuopen.com/prorender-usd-materialx-matlib/>.
- [53] Contribuyentes de MaterialX. *Material X*. URL: <https://materialx.org/>.

- [54] AMD GPUOpen. *MaterialX Library*. URL: <https://matlib.gpuopen.com/main/materials/all>.
- [55] Advanced Micro Devices Inc. *Radeon ProRender SDK*. URL: <https://radeon-pro.github.io/RadeonProRenderDocs/en/sdk/about.html>.
- [56] Advanced Micro Devices Inc. *rpr_camera_info*. URL: https://radeon-pro.github.io/RadeonProRenderDocs/en/sdk/info_setting_types/rpr_camera_info.html#rpr-camera-info.
- [57] Advanced Micro Devices Inc. *rprIESLightSetRadiantPower3f*. URL: <https://radeon-pro.github.io/RadeonProRenderDocs/en/sdk/api/rprieslightsetradiantpower3f.html>.
- [58] NVIDIA. *NVIDIA Iray SDK*. URL: <https://developer.nvidia.com/iray-sdk>.
- [59] NVIDIA. *5 - Iray Interactive*. URL: https://raytracing-docs.nvidia.com/iray/manual/index.html#iray_interactive_renderer_mode#iray-interactive.
- [60] Epic Games. *Rendering High Quality Frames with Movie Render Queue*. URL: <https://dev.epicgames.com/documentation/en-us/unreal-engine/rendering-high-quality-frames-with-movie-render-queue-in-unreal-engine>.
- [61] Epic Games. *Epic Games Github*. URL: <https://github.com/EpicGames>.
- [62] Epic Games. *Documentación de Unreal Engine 5*. URL: <https://dev.epicgames.com/documentation/es-es/unreal-engine/unreal-engine-5-6-documentation>.
- [63] Epic Games. *Biblioteca de formación Unreal Engine*. URL: <https://dev.epicgames.com/community/unreal-engine/learning>.
- [64] Joe Foley. *Unreal Engine dominates as the most successful game engine, Data reveals*. Feb. de 2025. URL: <https://www.creativebloq.com/3d/video-game-design/unreal-engine-dominates-as-the-most-successful-game-engine-data-reveals>.
- [65] Unreal Sensei. *Unreal Sensei*. URL: <https://www.youtube.com/@UnrealSensei>.
- [66] Epic Games. *Epic Games Community*. URL: <https://dev.epicgames.com/community/>.
- [67] Epic Games. *Unreal Engine Licesing*. URL: <https://www.unrealengine.com/en-US/license>.
- [68] Chaos Group. *Getting Started*. URL: <https://docs.chaos.com/display/APPSDK/Getting+Started>.
- [69] Chaos Group. *V-Ray Catalogue*. URL: <https://academy.chaos.com/pages/v-ray-catalogue>.
- [70] Advanced Micro Devices Inc. *Context Creation Tutorial*. URL: https://radeon-pro.github.io/RadeonProRenderDocs/en/sdk/tutorials/context_creation.html.
- [71] Advanced Micro Devices Inc. *Hybrid Tutorial*. URL: <https://radeon-pro.github.io/RadeonProRenderDocs/en/sdk/tutorials/hybrid.html>.
- [72] Advanced Micro Devices Inc. *Downloading and getting started*. URL: https://radeon-pro.github.io/RadeonProRenderDocs/en/sdk/getting_started.html.
- [73] GPUOpen-LibrariesAndSDKs. *32_gl_interop: main.cpp*. https://github.com/GPUOpen-LibrariesAndSDKs/RadeonProRenderSDK/tree/master/tutorials/32_gl_interop. Accessed: 2025-06-15. 2025.
- [74] Apache Software Foundation. *Apache License, Version 2.0*. URL: <https://www.apache.org/licenses/LICENSE-2.0>.
- [75] Contribuyentes de Wikipedia. *C++*. URL: <https://en.wikipedia.org/wiki/C%2B%2B>.
- [76] Contribuyentes de Wikipedia. *Microsoft Visual Studio*. URL: https://es.wikipedia.org/wiki/Microsoft_Visual_Studio.
- [77] Contribuyentes de Wikipedia. *OpenGL*. URL: <https://en.wikipedia.org/wiki/OpenGL>.
- [78] Joey De Vries. «Creating a window». En: *Learn opengl: Learn modern opengl graphics programming in a step-by-step fashion*. Kendall & Welling, 2020. URL: <https://learnopengl.com/>.
- [79] Contribuyentes de GLM. *GLM SDK*. URL: <https://www.opengl.org/sdk/libs/GLM/>.
- [80] Sean T. Barret. *stb_image_write*. https://github.com/nothings/stb/blob/master/stb_image_write.h. Accessed: 2025-06-15. 2025.
- [81] Omar Cornut. *Dear ImGui*. <https://github.com/ocornut/imgui?tab=readme-ov-file>. Accessed: 2025-06-15. 2025.
- [82] Contribuyentes de cppreference.com. *PImpl*. URL: <https://en.cppreference.com/w/cpp/language/pimpl.html>.
- [83] Advanced Micro Devices Inc. *Basic Scene Demo Tutorial*. URL: https://radeon-pro.github.io/RadeonProRenderDocs/en/sdk/tutorials/basic_scene_demo.html.

- [84] Advanced Micro Devices Inc. *rpr_status*. URL: https://radeon-pro.github.io/RadeonProRenderDocs/en/sdk/info_setting_types/rpr_status.html.
- [85] Advanced Micro Devices Inc. *rprCreateContext*. URL: <https://radeon-pro.github.io/RadeonProRenderDocs/en/sdk/api/rprcreatecontext.html>.
- [86] Advanced Micro Devices Inc. *rprContextCreateEnvironmentLight*. URL: <https://radeon-pro.github.io/RadeonProRenderDocs/en/sdk/api/rprcontextcreateenvironmentlight.html>.
- [87] Joey De Vries. «Diffuse Irradiance». En: *Learn opengl: Learn modern opengl graphics programming in a step-by-step fashion*. Kendall & Welling, 2020. URL: <https://learnopengl.com/>.
- [88] Advanced Micro Devices Inc. *rprContextCreateImage*. URL: <https://radeon-pro.github.io/RadeonProRenderDocs/en/sdk/api/rprcontextcreateimage.html>.
- [89] Advanced Micro Devices Inc. *rprContextCreateMesh*. URL: <https://radeon-pro.github.io/RadeonProRenderDocs/en/sdk/api/rprcontextcreatemesh.html>.
- [90] Brent Burley. «Physically Based Shading at Disney». En: *SIGGRAPH* (2012).
- [91] Advanced Micro Devices Inc. *Textured Material Creation Tutorial*. URL: https://radeon-pro.github.io/RadeonProRenderDocs/en/sdk/tutorials/textured_material_creation.html.
- [92] Jayant Silva. Oct. de 2010. URL: <https://rucore.libraries.rutgers.edu/rutgers-lib/30437/PDF/1/play/>.
- [93] Joey De Vries. «Normal Mapping». En: *Learn opengl: Learn modern opengl graphics programming in a step-by-step fashion*. Kendall & Welling, 2020. URL: <https://learnopengl.com/>.
- [94] GPUOpen-LibrariesAndSDKs. *22_material_uber: main.cpp*. https://github.com/GPUOpen-LibrariesAndSDKs/RadeonProRenderSDK/blob/master/tutorials/22_material_uber/main.cpp. Accessed: 2025-06-15. 2025.
- [95] Matt Pharr, Wenzel Jakob y Greg Humphreys. «Distant Lights». En: *Physically based rendering: From theory to implementation*. The MIT Press, 2023. URL: <https://pbr-book.org>.
- [96] Advanced Micro Devices Inc. *Emissive Shape Creation Tutorial*. URL: https://radeon-pro.github.io/RadeonProRenderDocs/en/sdk/tutorials/emissive_shape_creation.html.
- [97] Advanced Micro Devices Inc. *RPR_MATERIAL_NODE_EMISSIVE*. URL: https://radeon-pro.github.io/RadeonProRenderDocs/en/sdk/shaders/rpr_material_node_emissive.html.
- [98] Contribuyentes de Wikipedia. *Luminous efficacy*. URL: https://en.wikipedia.org/wiki/Luminous_efficiency.
- [99] Simon Crone. *Radiance User's Manual, Volume Two*. Nov. de 1992. URL: <https://radsite.lbl.gov/radiance/refer/usman2.pdf>.
- [100] Advanced Micro Devices Inc. *rprContextCreateInstance*. URL: <https://radeon-pro.github.io/RadeonProRenderDocs/en/sdk/api/rprcontextcreateinstance.html>.
- [101] Advanced Micro Devices Inc. *rprCameraLookAt*. URL: <https://radeon-pro.github.io/RadeonProRenderDocs/en/sdk/api/rprcameralookat.html>.
- [102] Joey De Vries. «Coordinate Systems». En: *Learn opengl: Learn modern opengl graphics programming in a step-by-step fashion*. Kendall & Welling, 2020. URL: <https://learnopengl.com/>.
- [103] GPUOpen-LibrariesAndSDKs. *13_deformation_motion_blur: main.cpp*. https://github.com/GPUOpen-LibrariesAndSDKs/RadeonProRenderSDK/tree/master/tutorials/13_deformation_motion_blur/main.cpp. Accessed: 2025-06-15. 2025.
- [104] Lawrence Berkley National Laboratory. *Radsite*. URL: <https://www.radiance-online.org/>.
- [105] Lawrence Berkley National Laboratory. *Filmspeed*. URL: <https://radsite.lbl.gov/radiance/refer/Notes/filmspeed.html>.
- [106] Contribuyentes de Wikipedia. *Film Speed*. URL: https://en.wikipedia.org/wiki/Film_speed#Determining_film_speed.
- [107] Lawrence Berkley National Laboratory. *PFILT*. URL: https://radsite.lbl.gov/radiance/man_html/pfilt.1.html.
- [108] Advanced Micro Devices Inc. *rprContextRender*. URL: <https://radeon-pro.github.io/RadeonProRenderDocs/en/sdk/api/rprcontextrender.html>.
- [109] Advanced Micro Devices Inc. *rprFrameBufferClear*. URL: <https://radeon-pro.github.io/RadeonProRenderDocs/en/sdk/api/rprframebufferclear.html>.

- [110] Advanced Micro Devices Inc. *rprFrameBufferGetInfo*. URL: <https://radeon-pro.github.io/RadeonProRenderDocs/en/sdk/api/rprframebuffergetinfo.html>.
- [111] Tomas Akenine-Möller, Eric Haines y Naty Hoffman. «The Graphics Processing Unit». En: *Real-time rendering*. CRC Press, 2018.
- [112] Joey De Vries. «Gamma Correction». En: *Learn opengl: Learn modern opengl graphics programming in a step-by-step fashion*. Kendall & Welling, 2020. URL: <https://learnopengl.com/>.
- [113] Advanced Micro Devices Inc. *rpr_context_info*. URL: https://radeon-pro.github.io/RadeonProRenderDocs/en/sdk/info_setting_types/rpr_context_info.html.
- [114] GPUOpen-LibrariesAndSDKs. *05_basic_scene: main.cpp*. https://github.com/GPUOpen-LibrariesAndSDKs/RadeonProRenderSDK/blob/master/tutorials/05_basic_scene/main.cpp. Accessed: 2025-06-15. 2025.
- [115] Contribuyentes de Wikipedia. *Automotive Paint*. URL: https://en.wikipedia.org/wiki/Automotive_paint.
- [116] Advanced Micro Devices Inc. *Common Light Properties*. URL: https://radeon-pro.github.io/RadeonProRenderDocs/en/plugins/blender/common_light_properties.html.
- [117] Advanced Micro Devices Inc. *Radeon Image Filtering Library*. URL: <https://gpuopen.com/archived/radeon-image-filtering-library/>.
- [118] Advanced Micro Devices Inc. *Photographic Tone Mapper*. URL: https://radeon-pro.github.io/RadeonProRenderDocs/en/rif/filters/photographic_tone_mapper.html.
- [119] Advanced Micro Devices Inc. *Basic Example*. URL: https://radeon-pro.github.io/RadeonProRenderDocs/en/rif/basic_example.html.
- [120] Advanced Micro Devices Inc. *Bloom*. URL: <https://radeon-pro.github.io/RadeonProRenderDocs/en/rif/filters/bloom.html>.
- [121] Mike Seymour. *V-Ray's practical stochastic rendering of spec-y things*. URL: <https://www.fxguide.com/quicktakes/v-rays-practical-stochastic-rendering-of-spec-y-things/>.

A. Entorno de pruebas interactivo

El entorno de pruebas interactivo es una aplicación en C++ independiente basada en *OpenGL* que permite controlar todos los aspectos de nuestra solución en tiempo real. A través de una interfaz, es posible cargar modelos 3D en formato *.obj*, ajustar dinámicamente sus transformaciones, modificar los parámetros del material de la superficie a inspeccionar, y configurar las propiedades de las luces y cámaras de la escena.

El programa invoca la función `renderPreview()` en cada fotograma para obtener una previsualización interactiva y muestra la textura resultante en pantalla, y también ofrece la funcionalidad de generar y guardar un renderizado final de alta calidad mediante la llamada a `renderFinal()`.

Esta aplicación se desarrolló debido a que la integración de la interfaz no fue simultánea a su implementación. Por tanto, en ausencia del software de simulación que invocara las funciones de nuestra interfaz y mostrara los resultados en pantalla, fue necesario crear un entorno de pruebas por las siguientes razones:

- Facilitar la depuración de la clase `PBRRenderer` durante su desarrollo.
- Simplificar enormemente la tarea de recreación de las imágenes capturadas para la validación de la solución.
- Permitir hacer una demostración de la solución implementada durante la defensa de este Trabajo de Fin de Grado.

Como se ha discutido en la [Subsección 5.1](#), hacemos uso en esta aplicación de *GLFW* para facilitar la creación del contexto de *OpenGL* y la ventana, y para gestionar la entrada por teclado que controla la cámara orbital implementada. Igualmente, empleamos *Dear ImGui* para mostrar una interfaz de usuario gráfica que permite ajustar todos los aspectos de la escena.

A.1. Cámara orbital

Para facilitar la inspección interactiva de la escena en el entorno de pruebas, se ha implementado una cámara orbital. Este tipo de cámara mira siempre a un punto de interés fijo, que en nuestro caso se corresponde con el origen de coordenadas, mientras que su posición orbita alrededor de este punto sobre una esfera imaginaria de radio configurable. De esta manera se puede examinar el modelo de la superficie a inspeccionar desde cualquier ángulo, ofreciendo así una comprensión más completa de su geometría y sus relaciones espaciales.

Las teclas W y S desplazan la cámara verticalmente a lo largo de su órbita, modificando su ángulo polar. Por otro lado, las teclas A y D gestionan el movimiento horizontal, alterando su ángulo azimutal.

Internamente, la clase que gestiona la cámara recalcula su posición tridimensional en cada fotograma basándose en los ángulos polar y azimutal actuales y el radio de la órbita, utilizando una conversión de coordenadas esféricas a cartesianas. Para garantizar evitar el *gimbal lock*, implementamos una restricción que impide que la cámara alcance los polos superior e inferior de la esfera sobre la cual la cámara orbita.

En cada ciclo del bucle de dibujo de la aplicación, el entorno de pruebas consulta la posición, el punto de interés y el vector de orientación vertical (*up*) de la cámara orbital. Estos tres parámetros se pasan directamente a la interfaz `PBRRenderer` para actualizar la cámara de la escena.

A.2. Cargador de *.obj*

Con el fin de facilitar la tarea de recrear los objetos fotografiados durante la validación de la solución, implementamos un cargador para modelos en formato *Wavefront* (*.obj*). Este formato fue elegido por su simplicidad y su amplio soporte en la industria, que facilitó la adquisición de los modelos 3D necesarios.

El cargador se ha diseñado para extraer únicamente la información geométrica relevante para nuestra solución: posiciones de los vértices, normales y coordenadas de textura. Por consiguiente, se ignoran deliberadamente los materiales (ficheros *.mtl*) y las texturas asociadas, ya que la clase `PBRRenderer` maneja las texturas para los mapas de normales por separado.

El proceso de carga se realiza leyendo el fichero *.obj* línea por línea. El cargador *parsea* las palabras clave `v` (vértice), `vn` (normal de vértice) y `vt` (coordenada de textura), almacenando estos datos en listas temporales. Cuando encuentra una directiva `f` (cara), procesa cada uno de sus vértices.

Un único vértice físico puede tener diferentes normales o coordenadas de textura dependiendo de la cara a la que pertenezca. Para evitar la duplicación innecesaria de datos, el cargador implementa un mecanismo de indexación. Utiliza un `map` para asociar cada combinación única de índices de posición, normal y coordenada de textura con un nuevo índice de vértice unificado en la malla final. Si una

combinación de índices ya ha sido procesada, se reutiliza su índice existente. Por otro lado, si es nueva, se crea un nuevo vértice y se añade al `map`.

Además, el cargador es capaz de manejar ficheros `.obj` que contienen múltiples objetos o grupos (indicado por las palabras clave `o` o `g`), devolviendo un vector de mallas, donde cada malla corresponde a un objeto distinto dentro del fichero.

B. Colección completa de capturas y recreaciones

B.1. *Clearcoat* blanco

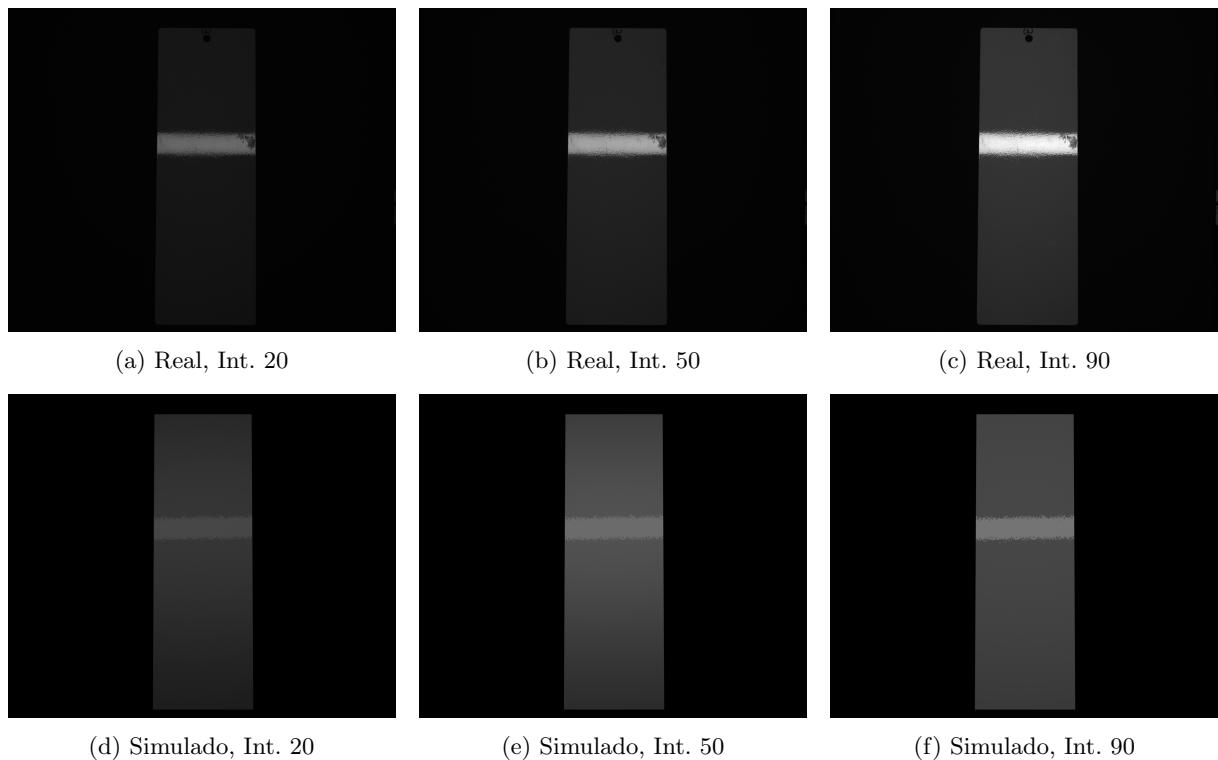


Figura 33: Comparación del efecto de variar la intensidad lumínica (columnas) para un tiempo de exposición constante de $0,001s$. Fila superior: capturas reales. Fila inferior: imágenes simuladas.

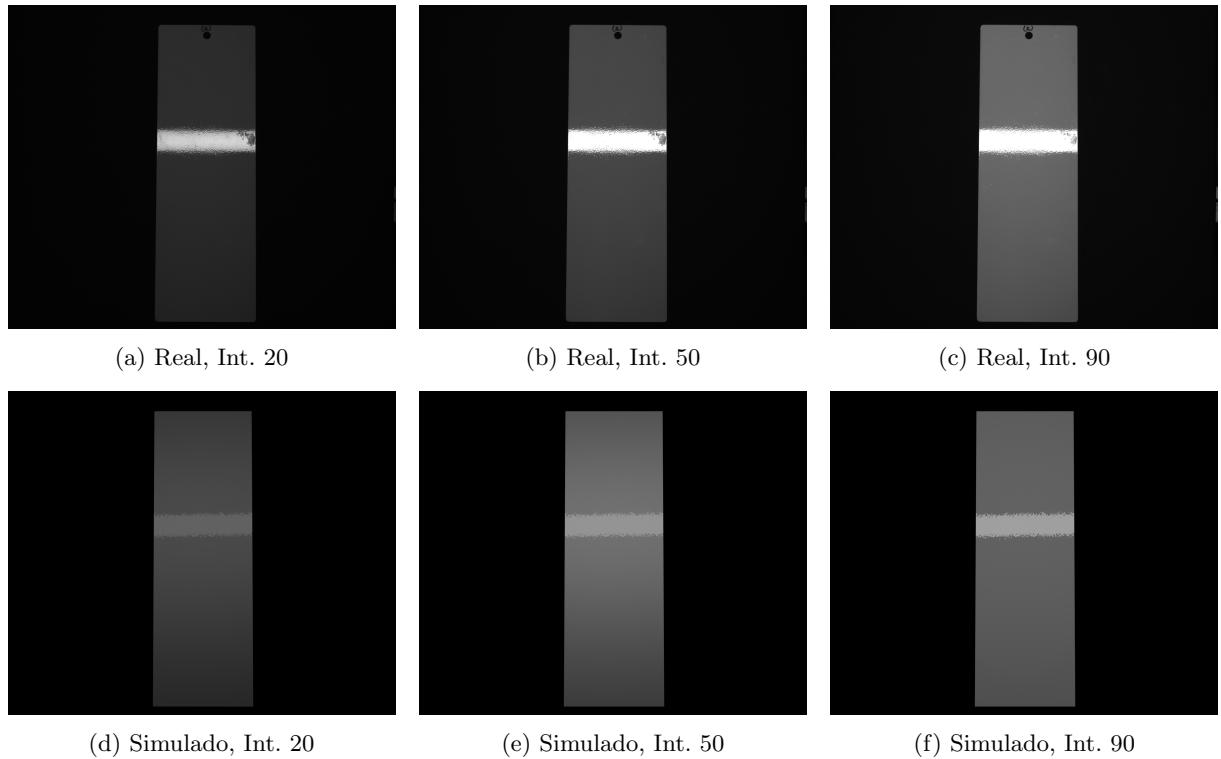


Figura 34: Comparación del efecto de variar la intensidad lumínica (columnas) para un tiempo de exposición constante de 0,002s. Se mantiene la misma estructura de la figura anterior.

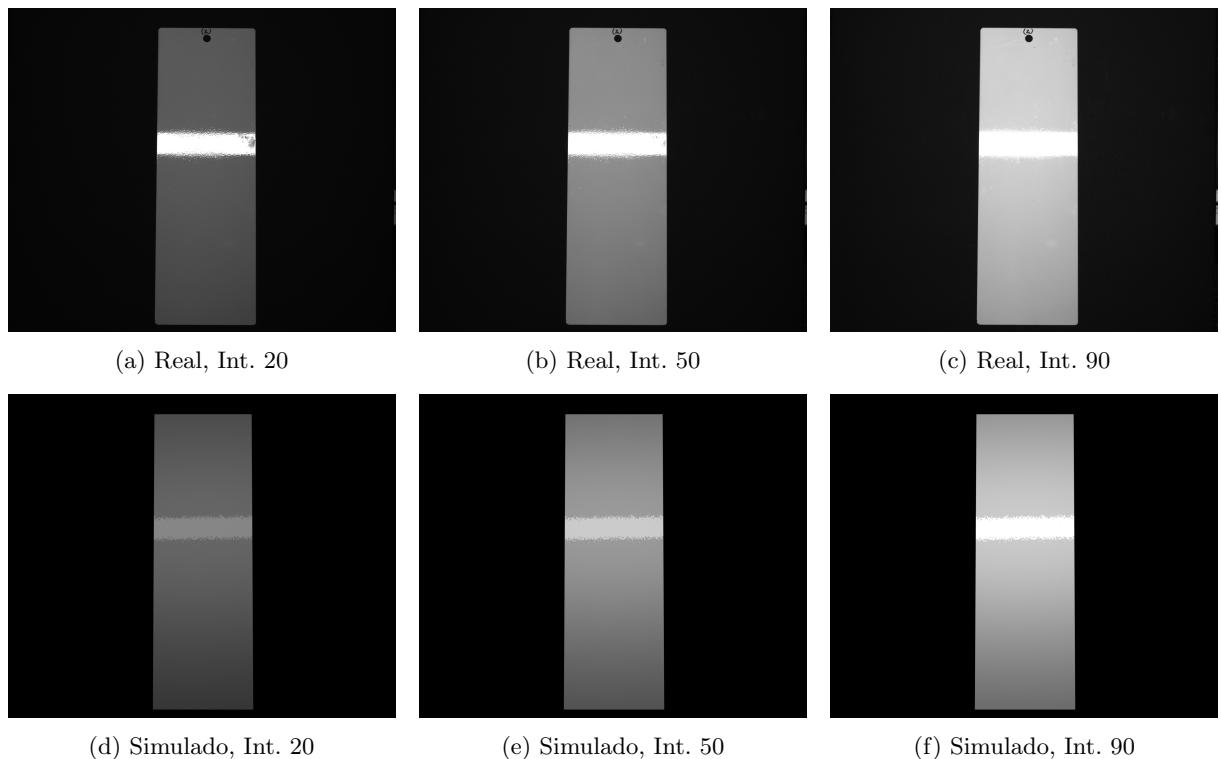


Figura 35: Comparación del efecto de variar la intensidad lumínica (columnas) para un tiempo de exposición constante de 0,004s. La estructura es idéntica a las figuras anteriores.

B.2. *Clearcoat gris*

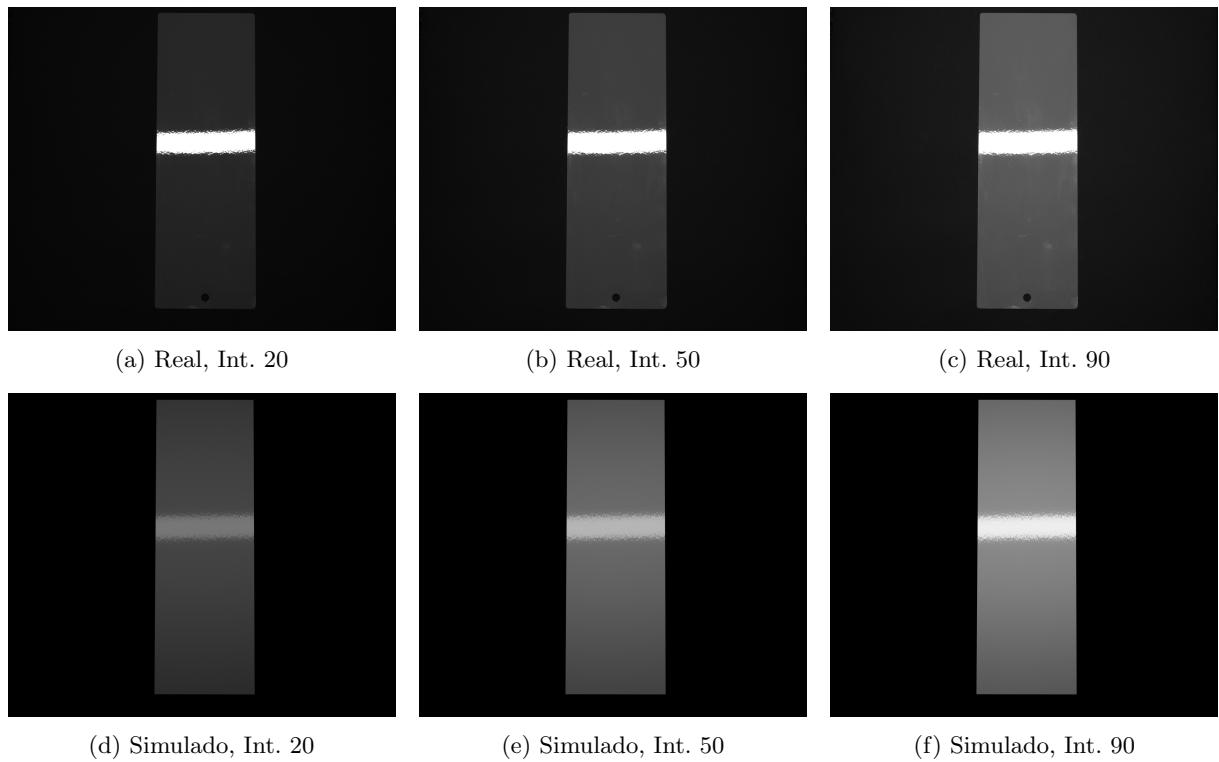


Figura 36: Comparación para la placa de *clearcoat gris* con un tiempo de exposición constante de $0,005s$.

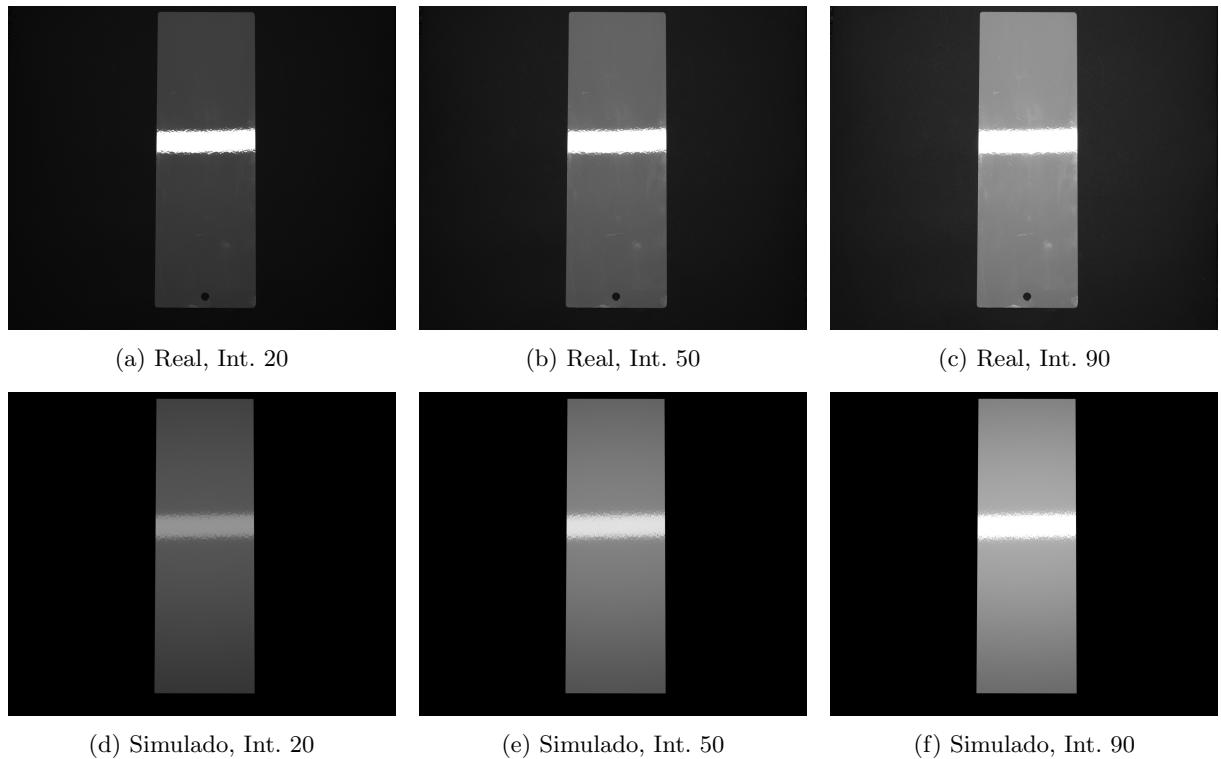


Figura 37: Comparación para la placa de *clearcoat* gris con un tiempo de exposición constante de 0,008s. La estructura es idéntica a la figura anterior.

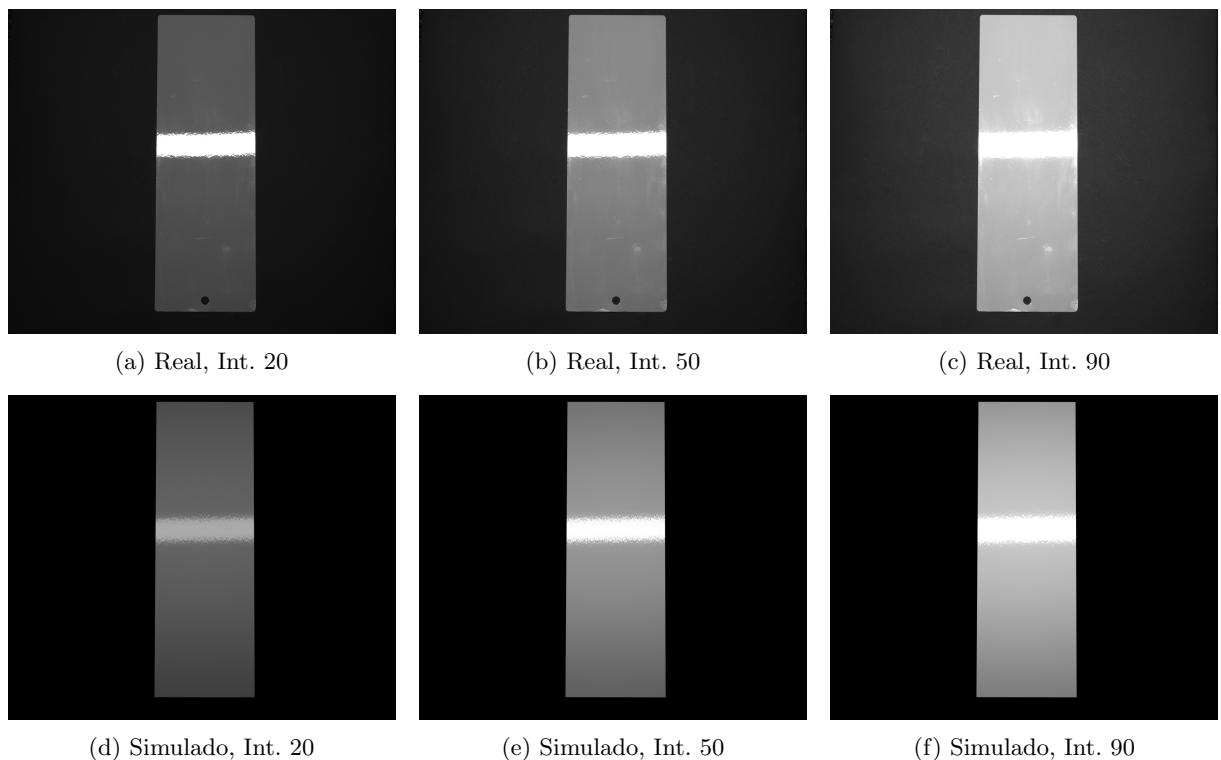


Figura 38: Comparación del efecto de variar la intensidad lumínica (columnas) sobre la placa de *clearcoat* gris, con un tiempo de exposición constante de 0,011s. Fila superior: capturas reales. Fila inferior: imágenes simuladas.

B.3. *Clearcoat gris inclinado*

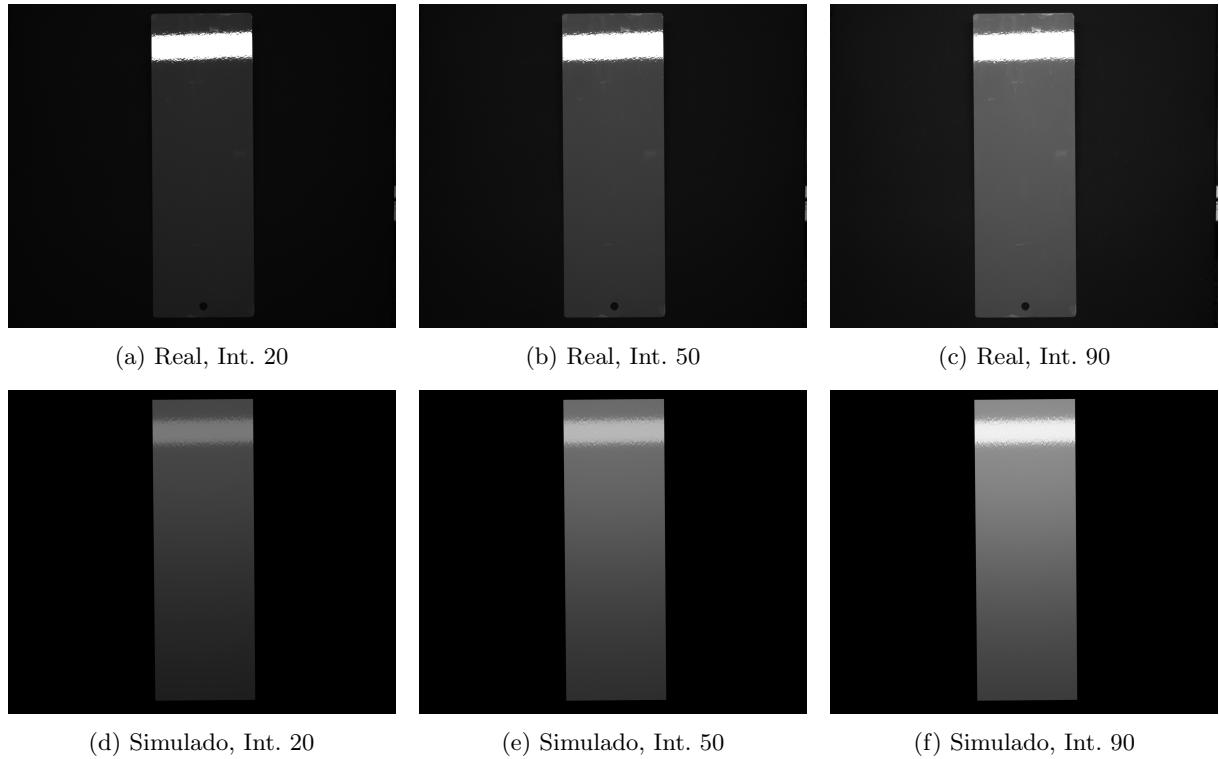


Figura 39: Comparación para la placa de *clearcoat gris inclinada* con un tiempo de exposición constante de 0,005s.

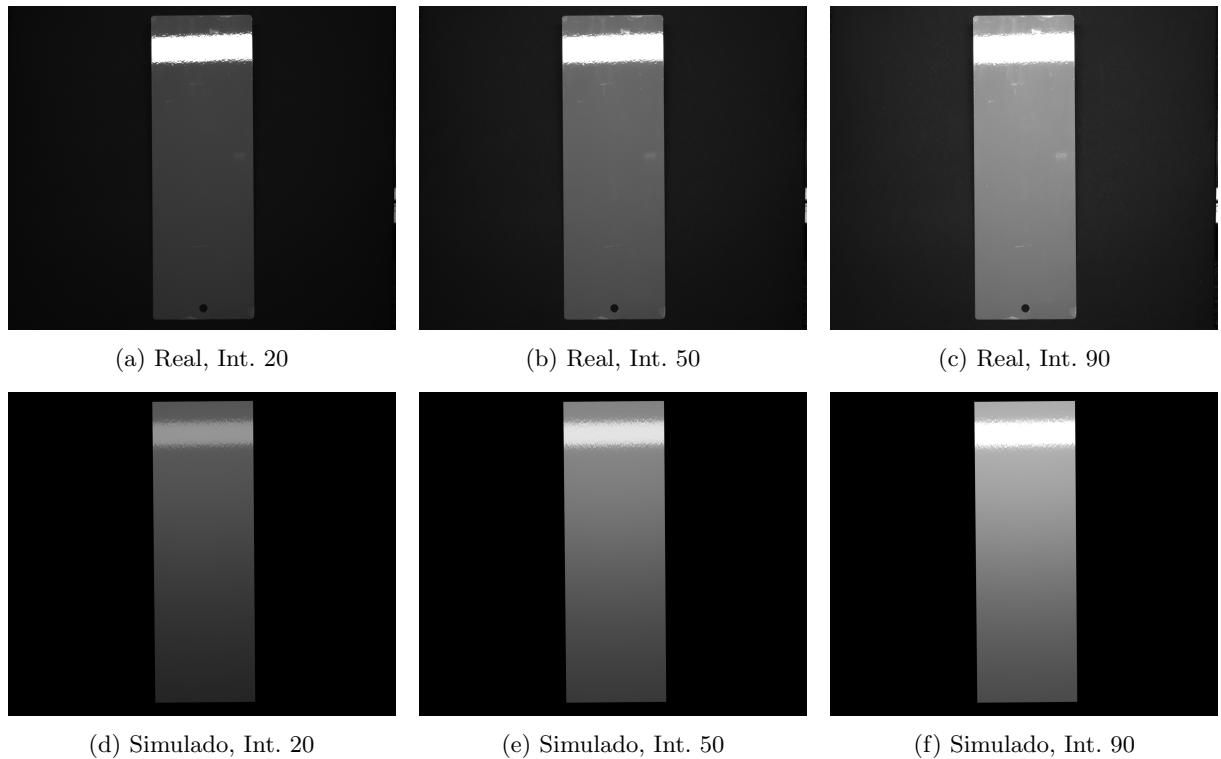


Figura 40: Comparación para la placa de *clearcoat* gris inclinada con un tiempo de exposición constante de $0,008s$.

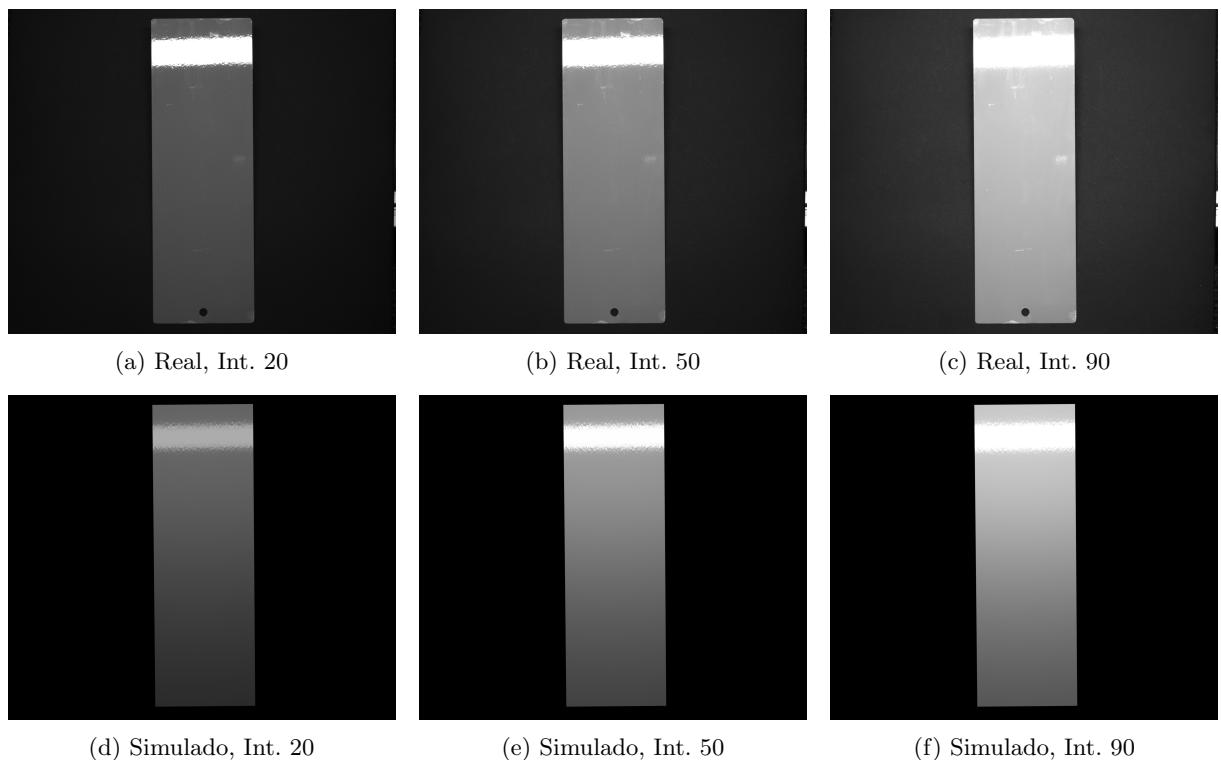


Figura 41: Comparación del efecto de variar la intensidad lumínica (columnas) sobre la placa de *clearcoat* gris inclinada, con un tiempo de exposición constante de $0,011s$. Fila superior: capturas reales. Fila inferior: imágenes simuladas.

B.4. *Clearcoat* negro

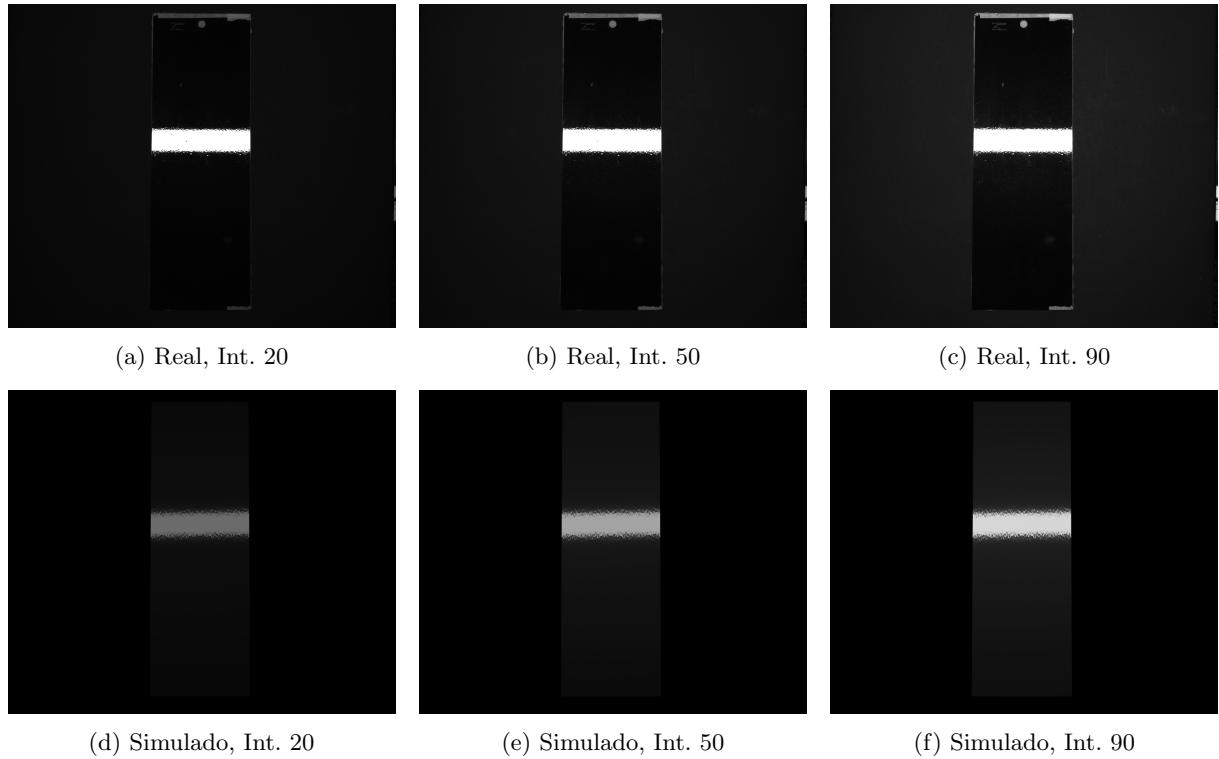


Figura 42: Comparación para la placa de *clearcoat* negro con un tiempo de exposición constante de 0,006s.

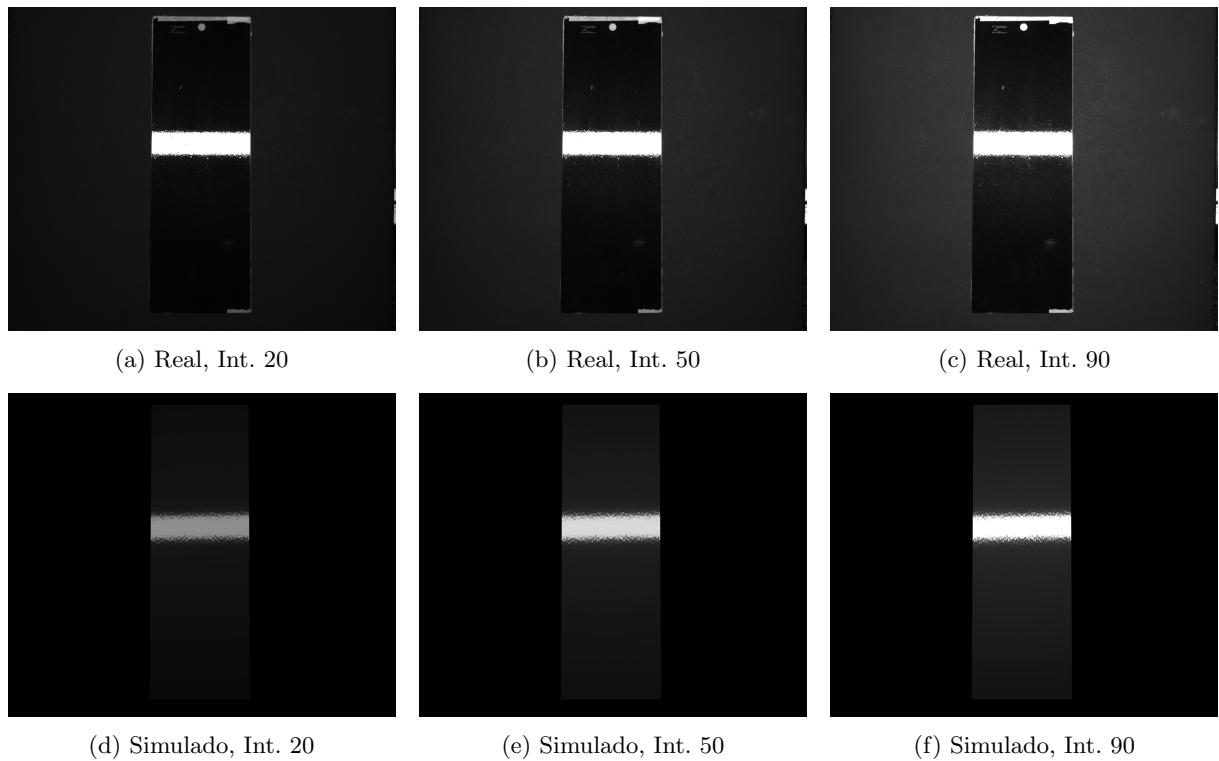


Figura 43: Comparación para la placa de *clearcoat* negro con un tiempo de exposición constante de $0,011s$.

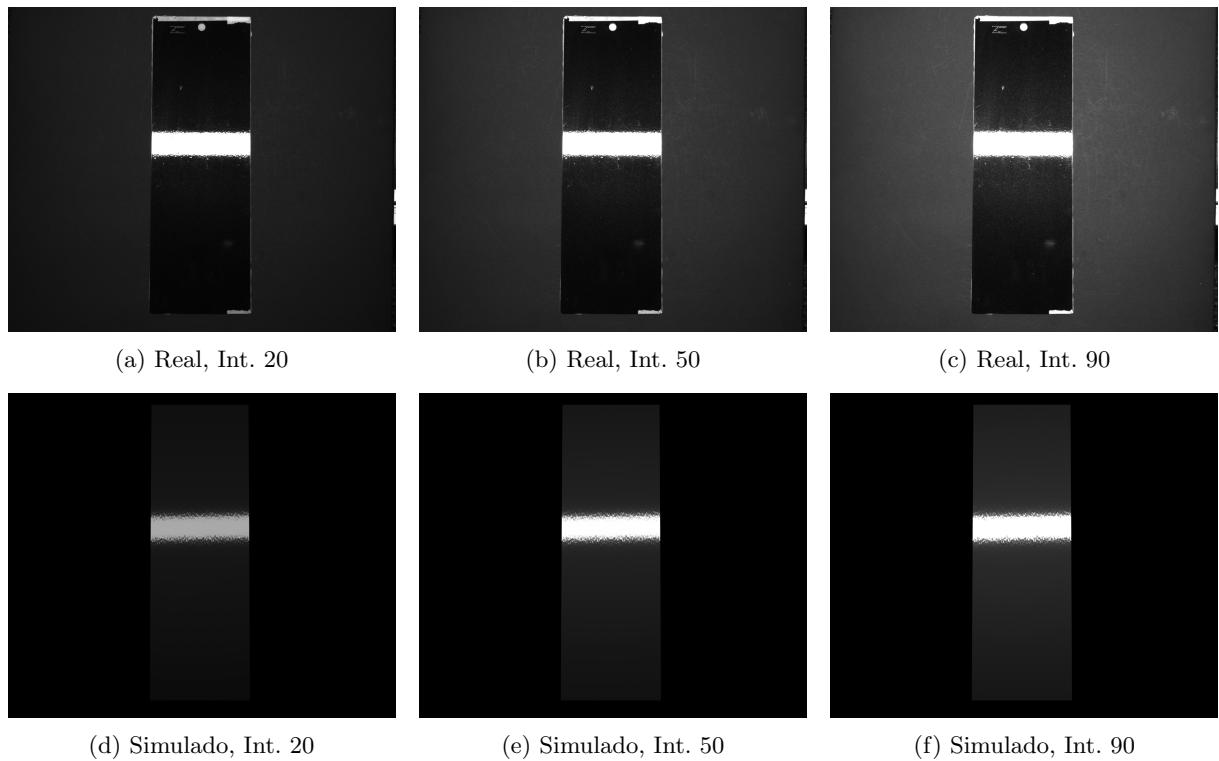


Figura 44: Comparación del efecto de variar la intensidad lumínica (columnas) sobre la placa de *clearcoat* negro, con un tiempo de exposición constante de $0,016s$. Fila superior: capturas reales. Fila inferior: imágenes simuladas.

B.5. *Primer*

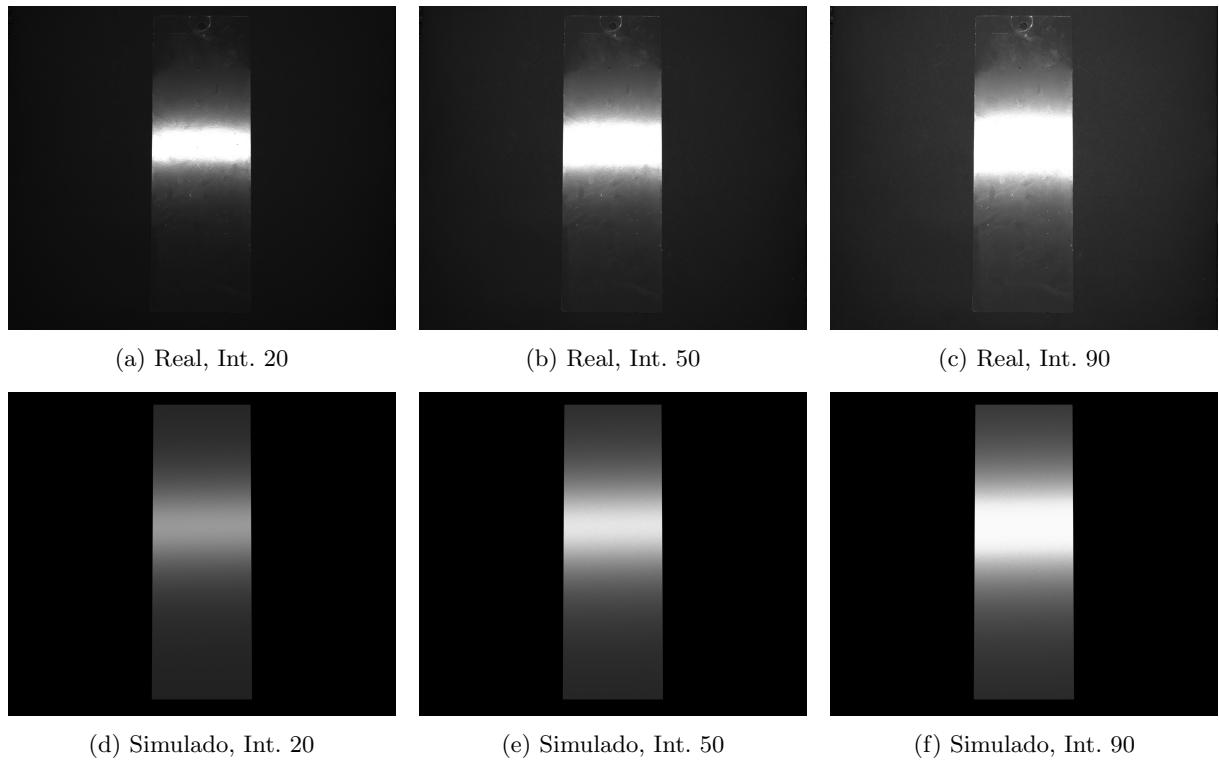


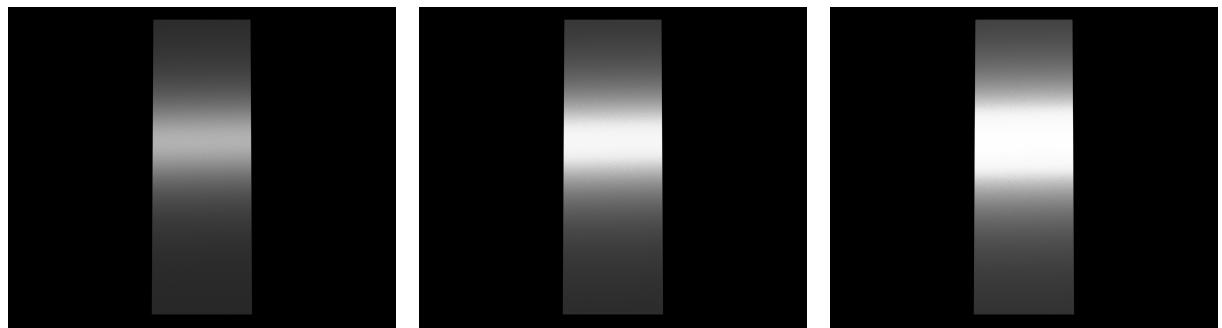
Figura 45: Comparación del efecto de variar la intensidad lumínica (columnas) sobre la placa de *primer*, con un tiempo de exposición constante de 0,010s. Fila superior: capturas reales. Fila inferior: imágenes simuladas.



(a) Real, Int. 20

(b) Real, Int. 50

(c) Real, Int. 90



(d) Simulado, Int. 20

(e) Simulado, Int. 50

(f) Simulado, Int. 90

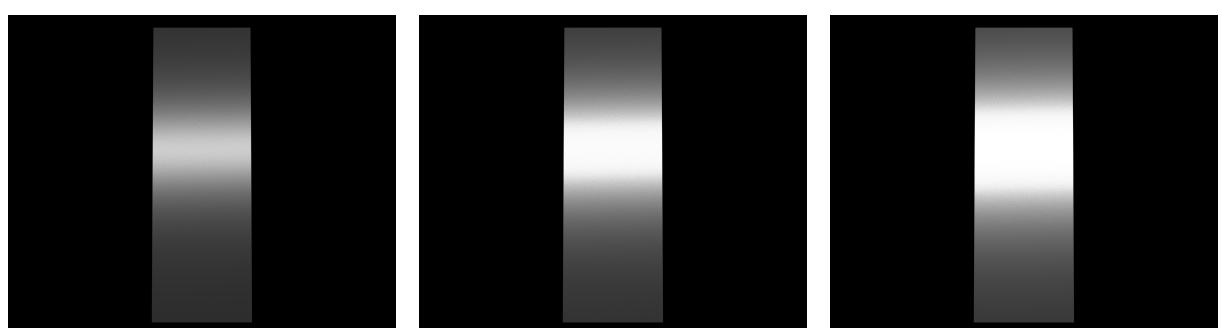
Figura 46: Comparación para la placa de *primer* con un tiempo de exposición constante de 0,014s.



(a) Real, Int. 20

(b) Real, Int. 50

(c) Real, Int. 90



(d) Simulado, Int. 20

(e) Simulado, Int. 50

(f) Simulado, Int. 90

Figura 47: Comparación para la placa de *primer* con un tiempo de exposición constante de 0,019s.

C. Objetivos de Desarrollo Sostenible

Aunque la naturaleza de este proyecto es puramente técnica, su aplicación industrial y la metodología empleada para resolver el problema planteado por *AUTIS* se alinean directamente con varios Objetivos de Desarrollo Sostenible de las Naciones Unidas, como se puede observar en la [Tabla 1](#). La contribución principal recae en la capacidad de la solución para sustituir un proceso físico costoso y generador de residuos (el daño deliberado de vehículos) por una alternativa digital eficiente y limpia. A continuación, se detalla la contribución a cada ODS identificado.

C.1. ODS 9: Industria, Innovación e Infraestructura

Este objetivo busca construir infraestructuras resilientes, promover la industrialización sostenible y fomentar la innovación. El proyecto contribuye directamente a las siguientes metas:

- **Fomento de la innovación (Meta 9.5):** Se desarrolla una solución tecnológica de vanguardia (renderizado PBR para la generación de datos sintéticos) que moderniza y optimiza un proceso industrial crítico como es el control de calidad.
- **Modernización industrial (Meta 9.4):** La automatización del control de calidad, facilitada por los datos que este sistema puede generar, impulsa la modernización de la industria.

C.2. ODS 12: Producción y Consumo Responsables

Este es el objetivo con la conexión más directa y significativa, ya que busca garantizar modalidades de consumo y producción sostenibles.

- **Prevención de residuos (Meta 12.5):** La motivación central del proyecto es evitar la necesidad de dañar físicamente un gran volumen de carrocerías. Cada imagen sintética generada previene la creación de un residuo industrial complejo, ahorrando metales, plásticos, pinturas y la energía asociada a su fabricación y tratamiento.
- **Eficiencia en el uso de recursos (Meta 12.2):** Se reemplaza un proceso de alto coste material por uno de coste computacional. La eficiencia en el uso de recursos es drásticamente superior.
- **Reducción del uso de productos químicos (Meta 12.4):** Como consecuencia directa de lo anterior, se evita el uso de disolventes, pigmentos, barnices y otros productos químicos necesarios para el pintado y acabado de las carrocerías que habrían sido descartadas.

C.3. ODS 8: Trabajo Decente y Crecimiento Económico

Este objetivo promueve el crecimiento económico sostenido e inclusivo. El proyecto contribuye de forma indirecta al:

- **Aumento de la productividad (Meta 8.2):** La automatización del control de calidad permite a la industria ser más eficiente, reducir los costes asociados a defectos y mejorar la calidad del producto final, lo que impulsa un crecimiento económico sostenible.
- **Desvinculación del crecimiento y la degradación ambiental (Meta 8.4):** Este TFG es un ejemplo práctico de cómo la tecnología puede permitir un avance industrial (crecimiento económico) mientras se reduce su impacto medioambiental (degradación), al eliminar una fuente de residuos materiales del proceso.

C.4. ODS 4: Educación de Calidad

El propio desarrollo del Trabajo de Fin de Grado es una manifestación de este objetivo.

- **Desarrollo de competencias técnicas (Meta 4.4):** El proyecto me ha permitido la adquisición de competencias avanzadas y altamente especializadas en campos de alta demanda como es el renderizado basado, preparándome para el mercado laboral tecnológico.

Cuadro 1: Evaluación del impacto del TFG según los Objetivos de Desarrollo Sostenible.

Objetivos de Desarrollo Sostenible	Alto	Medio	Bajo	No procede
ODS 1. Fin de la pobreza.				X
ODS 2. Hambre cero.				X
ODS 3. Salud y bienestar.				X
ODS 4. Educación de calidad.	X			
ODS 5. Igualdad de género.				X
ODS 6. Agua limpia y saneamiento.			X	
ODS 7. Energía asequible y no contaminante.				X
ODS 8. Trabajo decente y crecimiento económico.		X		
ODS 9. Industria, innovación e infraestructuras.	X			
ODS 10. Reducción de las desigualdades.				X
ODS 11. Ciudades y comunidades sostenibles.				X
ODS 12. Producción y consumo responsables.	X			
ODS 13. Acción por el clima.			X	
ODS 14. Vida submarina.				X
ODS 15. Vida de ecosistemas terrestres.				X
ODS 16. Paz, justicia e instituciones sólidas.				X
ODS 17. Alianzas para lograr objetivos.				X