Escuela Politécnica Superior de Jaén
Máster Universitario en Ingeniería Informática

Universidad de Jaén
Escuela Politécnica Superior
de Jaén

TRABAJO FIN DE MÁSTER

SIMULACIÓN DE ESCANEADOS 3D

Alumno

Alfonso López Ruiz

Tutores

Carlos Javier Ogayar Anguita
(Departamento de Informática)

Francisco Ramón Feito Higueruela
(Departamento de Informática)

Diciembre, 2020

(Página intencionalmente en blanco)

Universidad de Jaén

Departamento de Informática

Don Carlos Javier Ogayar Anguita y Don Francisco Ramón Feito Higueruela, tutores del Trabajo Fin de Máster titulado: '**Simulación de escaneados 3D**', que presenta Don Alfonso López Ruiz, otorgan el visto bueno para su entrega y defensa en la Escuela Politécnica Superior de Jaén.

Jaén, Diciembre de 2020

El alumno: Los tutores:

Alfonso López Ruiz

Carlos Javier Ogayar Anguita

Francisco Ramón Feito Higueruela

(Página intencionalmente en blanco)

Agradecimientos

En primer lugar, me gustaría expresar mi agradecimiento a mis tutores, Carlos Ogayar y Francisco Feito, tanto por su ayuda como por la oportunidad que me han dado de introducirme en un trabajo tan interesante como este.

Igualmente, quiero transmitir mi gratitud a todos aquellos compañeros de máster y de trabajo que han hecho más llevaderos estos meses tan difíciles.

Por último, me gustaría agradecer tanto a mi hermano como a mis padres por su inmenso apoyo durante todo este tiempo.

FICHA DEL TRABAJO FIN DE MÁSTER	
Titulación	Máster en Ingeniería Informática
Modalidad	Trabajo Teórico/Experimental
Especialidad (solo TFG)	Sin especialidad
Mención (solo TFG)	Sin mención
Idioma	Español
Tipo	Específico
TFT en equipo	No
Autor/a	Alfonso López Ruiz
Fecha de asignación	16/11/2020
Descripción corta	Las nubes de puntos 3D procedentes de un sensor LiDAR se utilizan en un número considerable de aplicaciones: desde la preparación y verificación de trabajos de campo, a tareas relacionadas con la inteligencia artificial, como puede ser la conducción autónoma o el entrenamiento de sistemas robóticos. Sin embargo, su obtención representa un coste económico y temporal, y más allá de la adquisición, se observa un número muy reducido de conjuntos de datos etiquetados aplicables a tareas de *machine learning* y de visión por computador. Además, es común encontrar nubes de puntos cuyas etiquetas se reducen a unas pocas clases, que incluso se asignan manualmente, lo que implica que podrían existir errores en el proceso de etiquetado, más allá del limitado nivel de detalle existente. Por tanto, la simulación de un sensor LiDAR sobre un escenario 3D modelado permite obtener nubes de puntos sintéticas, correctamente etiquetadas, con clases ajustadas a un escenario concreto, y con un nivel de detalle personalizado. Por otro lado, la generación de nubes de puntos en gran cantidad puede obtenerse como consecuencia de la introducción de escenarios procedurales. El comportamiento físico del sensor hace que este problema pueda representar una elevada carga de trabajo. Por tanto, la introducción de la computación paralela puede ayudar a reducir el tiempo de respuesta del proceso de escaneo. Además, la simulación del sensor no sólo incluye una generación básica de una nube de puntos, sino también la introducción de aquellos errores más comunes vinculados a un dispositivo LiDAR, con el fin de reproducir de la manera más fiel posible su comportamiento.

Abstract	3D point clouds given by LiDAR sensors have many applications nowadays, from preparing and verifying field works, to tasks related to artificial intelligence, such as autonomous driving or training of robotic systems. Beyond their acquisition cost, there is only a small number of labelled points clouds to be applied to machine learning and computer vision algorithms. Furthermore, labels are commonly reduced to a few classes, which also affect their level of detail. They may even be assigned manually, which suggest there could be errors in the labelling process. Therefore, simulating a LiDAR sensor on a 3D modelled scenario allows to create synthetic points clouds, properly annotated with classes which are defined for an specific scenario and a customized level of detail. On the other hand, generating a large number of points clouds can be achieved with procedural scenarios. An accurate simulation of a LiDAR sensor is a time-consuming task. Thus, parallel computation is also included in this work, in order to reduce the response time of the scanning process. Finally, simulating a LiDAR sensor does not only includes the generation of a simple point cloud, but it also involves simulating those errors which are frequently related to such a sensor. This way, we manage to get an accurate simulation.

NORMAS APLICADAS EN ESTE DOCUMENTO	
LOCALES	
TFT-UJA:2017	Normativa de Trabajos Fin de Grado, Fin de Máster y otros Trabajos Fin de Título de la Universidad de Jaén (Normativa marco UJA aprobada en Consejo de Gobierno)
TFT-EPSJ:2017	Normativa sobre Trabajos Fin de Grado y Fin de Máster en la Escuela Politécnica Superior de Jaén (Normativa EPSJ aprobada en Junta de Escuela)
TFT-EPSJ	Criterios de evaluación y normas de estilo para TFG y TFM de la Escuela Politécnica Superior de Jaén
NACIONALES E INTERNACIONALES	
ISO 2145:1978	Documentación - Numeración de divisiones y subdivisiones en documentos escritos
UNE 50132:1994	Traducción de la ISO 2145
APA 6ª edición	Estilo de referencias y citas de APA (American Psychological Association)

NORMAS UTILIZADAS COMO BASE O REFERENCIA	
NACIONALES	
UNE 157001:2014	Criterios generales para la elaboración formal de los documentos que constituyen un proyecto técnico
UNE 157801:2007	Criterios generales para la elaboración de proyectos de sistemas de información
*Estas normas se han utilizado **como base o referencia** para la inclusión de algunos contenidos y definiciones sobre elaboración de proyectos, entendiendo como **proyecto** la documentación consensuada entre una empresa y un cliente, que da lugar al perfeccionamiento de un contrato para la elaboración de una obra o la prestación de un servicio. Por consiguiente, no debe esperarse la aplicación de estas normas en cuanto a la completitud de los contenidos ni a la organización de los mismos.*	

Contenido

Índice de ilustraciones

Índice de tablas

1 ESPECIFICACIÓN DEL TRABAJO

En este capítulo se presenta la especificación del trabajo, con una estructura y contenidos **inspirados** en los criterios y recomendaciones que establece la norma UNE 157801:2007 - "*Criterios Generales para la elaboración de proyectos de Sistemas de Información*".

A lo largo del documento se utilizarán términos y acrónimos cuya descripción aparece en el apartado de Definiciones y abreviaturas.

1.1 Introducción

La simulación del escaneo 3D de un escenario puede ser una herramienta de gran utilidad, debido al amplio número de aplicaciones en las que se utilizan los resultados obtenidos, así como al coste asociado a su adquisición (económico, temporal, etc). De esta manera, la aplicabilidad de esta simulación se encuentra principalmente en el entorno de la investigación, donde se centra el interés en áreas de la informática como la Inteligencia Artificial, y más concretamente, en aplicaciones espaciales (3D), donde la información semántica del escaneo resultante suele disponer de un gran valor.

De manera general, sin vincular este trabajo a un problema concreto, se pueden observar múltiples ventajas. Simular un escaneo 3D elimina la necesidad de disponer de un dispositivo que realice dicho trabajo, y disminuye el tiempo empleado en la obtención de datos de entrada en un proyecto. Por tanto, el resultado de este trabajo posibilitaría reducir costes y permitiría a un equipo centrarse exclusivamente en la tarea de investigación. Otro factor positivo es la gran variedad de escenarios de los que se puede disponer sin necesidad de llevar a cabo un desplazamiento físico, así como la flexibilidad de dichos escenarios, dado que estos pueden adaptarse para incluir unos modelos u otros en función de las necesidades del algoritmo o el tipo de entorno que se desea plantear (por ejemplo, el escenario de un bosque utilizará modelos completamente diferentes de los que emplearía el escenario de un desierto).

De manera más específica, encontramos áreas de investigación y problemas concretos en los que disponer de esta simulación podría ser de gran interés. Este es el caso de la clasificación de nubes de puntos, cuyo objetivo principal es discernir a

qué tipo de objeto pertenece la información espacial capturada. Mientras que en un escaneo real se desconoce o no existe una certeza completa del tipo de objeto al que pertenece la información espacial, un escaneo sintético dispone de ese conocimiento con total certeza, dado que cada modelo empleado se encuentra etiquetado.

Más allá de las utilidades presentadas, el trabajo desarrollado está orientado principalmente al entorno de la informática gráfica, donde la representación de un escenario y la implementación de algoritmos de diversa complejidad suponen una parte importante de la solución.

Ilustración 1. Nube de 69.752.511 puntos obtenida de un dispositivo LiDAR. Cada punto de la nube almacena información RGB de la escena original. Modelo obtenido de OpenTopology. Imagen renderizada en la aplicación de este trabajo.

Además de la importancia de los algoritmos, se debe considerar una restricción temporal, la cual se debe comprender considerando los tipos de resultados que produce un escaneo. A grandes rasgos, podemos contemplar la salida de un escaneo como millones de puntos en un espacio tridimensional, que además, pueden contener información adicional (color, concepto semántico, etc). Por tanto, la necesidad de hallar millones de respuestas hace necesaria la búsqueda de soluciones que minimicen el tiempo de ejecución. Esta búsqueda no se centra únicamente en paradigmas de programación, como la computación en unidades de procesamiento gráfico (GPGPU, *general-purpose computing on graphics processing units*), sino que también incluye la utilización de estructuras de datos eficientes.

Para completar esta introducción es necesario que el lector sea consciente de la utilización que se ha hecho del concepto de escaneo 3D, en lugar de hacer mención a una técnica concreta. Esta ambigüedad se debe a la existencia de múltiples metodologías para la captura de una escena, tales como SfM (*Structure from Motion*)

o LiDAR (*Light Detection and Ranging*), donde la segunda opción es sin duda la más precisa y extendida, pero también una de las que plantea mayores problemas en su replicación, si consideramos los conceptos físicos que rigen su comportamiento. Un ejemplo de utilización de la tecnología LiDAR se encuentra en los vehículos autónomos, donde la respuesta de estos depende de la información capturada del entorno.

1.1.1 Objetivos del trabajo

Los objetivos de este trabajo, definidos en este punto a un nivel alto de abstracción, se enumeran a continuación de manera exhaustiva:

- **Simular metodologías de escaneo 3D**, y más concretamente, nos centraremos en la tecnología **LiDAR**. La simulación de esta tecnología no sólo implica simular un comportamiento físico básico, donde una emisión láser (la cual puede interpretarse como un rayo) puede impactar en la superficie de un modelo de la escena. También deben introducirse errores comúnmente observados en un dispositivo LiDAR. Por ejemplo, se documentan errores en terrenos de mayor pendiente o superficies similares a un espejo.
- Más allá de un comportamiento genérico, existen **múltiples tipos de dispositivos LiDAR**, sea la distinción más básica un dispositivo orientado a escanear desde la superficie (***terrestrial***), o bien desde un vehículo aéreo (***airbone***). Incluso difieren en la capacidad de penetrar superficies tales como el agua. Por tanto, es necesario desarrollar aquellas herramientas que permitan que el usuario disponga de un escaneo flexible. No se debe orientar este trabajo de tal manera que el usuario pueda seleccionar modelos muy específicos de un sensor LiDAR, sino que estos (o sus características) deberían alcanzarse a través del ajuste de parámetros de un único dispositivo.
- A consecuencia de lo expuesto en el último párrafo, un objetivo claro es obtener una **simulación parametrizada**, la cual debe configurarse de la manera más cómoda posible. Por tanto, la aplicación gráfica debe integrar una interfaz que posibilite esta comunicación.

- Como bien se destacaba en la introducción, existe una restricción temporal importante, la cual no radica principalmente en el tiempo de espera del usuario (una espera de algunos minutos aún conserva todas las ventajas de la simulación respecto de un escaneo real), sino en la obtención de un proyecto donde la depuración sea más sencilla. Por tanto, otro objetivo es **minimizar el tiempo de obtención de los resultados**, sin indicar, de momento, los algoritmos o paradigmas que deben emplearse.

- **Generación y representación de diferentes escenarios en los que se llevará a cabo la simulación**. Partiendo de la distinción básica entre LiDAR terrestre y aéreo, serán necesarios al menos dos escenarios. Se considera un valor añadido la generación procedural de un escenario, de tal manera que es un algoritmo el que se encarga de modelar dicho entorno. Para concluir este objetivo, es necesario destacar que el trabajo que aquí se propone no presenta relación alguna con un posible *framework* donde el usuario pudiera modelar su propia escena. Por tanto, se parte de escenas previamente definidias, o en el mejor de los casos, de una escena generada proceduralmente, de tal manera que podrían obtenerse un número ilimitado de conjuntos de datos (de una misma temática).

  Por tanto, el trabajo desarrollado está meramente relacionado con la investigación y la búsqueda de un algoritmo de escaneo parametrizado, el cual podría extenderse, en un trabajo futuro, con un *framework* donde el usuario pudiera definir sus propias escenas procedurales, de acuerdo con una determinada temática.

- ***Rendering*** (representación) **realista de los escenarios**, considerando las restricciones fijadas por la extensión de este trabajo. Las nubes de puntos resultantes de un escaneo pueden almacenar múltiples tipos de información en cada uno de sus puntos, y en concreto, uno de ellos puede ser un color RGB (*Red, Green y Blue*) procedente del sombreado de la escena. Por tanto, se considera que un objetivo principal es representar una iluminación realista de la escena. Ciertamente, hay técnicas de *rendering* que simulan, de la manera más realista posible, el comportamiento físico de la luz, aunque no son indispensables para obtener un resultado válido.

1.2 Antecedentes y estado del arte

El objetivo de este apartado es exponer la historia y el contexto del problema que se plantea, con el único fin de introducir y justificar las soluciones parciales adoptadas así como la solución final.

La simulación de una actividad o un proceso a través de computador no es un problema de reciente aparición. En primer lugar, existen multitud de trabajos y procesos que necesitan de un entrenamiento previo para considerar que una persona se encuentra completamente capacitada, y en algunos de ellos existe un riesgo bastante evidente cuando no se dispone de experiencia previa. Esta situación se puede dar, por ejemplo, en el ámbito militar o de la medicina, de tal manera que una simulación puede ayudar a adquirir dicha experiencia.

El riesgo no es el único factor que ha propiciado la búsqueda de una simulación, sino también el coste, tanto ecónomico como temporal. Este es el caso de la conducción autónoma, donde la validación del comportamiento del vehículo puede depender de una gran cantidad de pruebas que deben someterse a esquemas muy estrictos (Li et al. 2019; Rosique et al. 2019). De esta manera, mediante una simulación podrían acelerarse estas pruebas y reducir el intervalo de tiempo necesario para validar una solución.

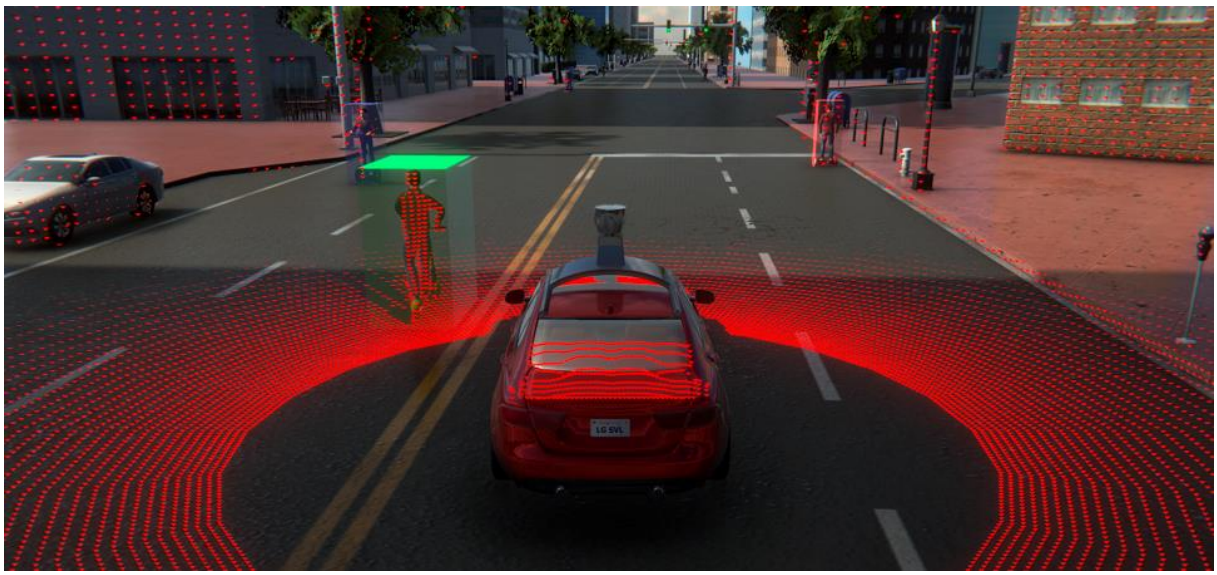


Ilustración 2. Simulación LiDAR perteneciente a una aplicación de soporte para la conducción autónoma, LGSVL Automotive Simulator. Fuente: (Talwar, s.f.).

Además, la simulación con éxito de un dispositivo, o varios de ellos, permite planificar proyectos mediante la comparación de aquellos sensores disponibles y la

selección del más indicado, reduciendo al máximo el coste de la tecnología como consecuencia de una elección adecuada. Por ejemplo, no todos los dispositivos LiDAR producen una misma nube de puntos de un escenario, ni disponen de una misma tecnología emisora y receptora (Royo and Ballesta-Garcia 2019).

Otra capacidad ligada a la simulación es la generación de conjuntos de datos sintéticos, reduciendo de nuevo el coste económico y temporal asociado a la obtención de datos de entrada. Esta capacidad toma especial relevancia en la actualidad debido al auge de los algoritmos de *machine learning*, los cuales suelen depender de grandes cantidades de datos. No obstante, esta necesidad colisiona en ocasiones con la realidad, debido a que los conjuntos de datos existentes en algunas áreas pueden ser muy escasos, de menor tamaño, o pueden contener errores (por ejemplo, cuando la asignación de marcadores es manual) (Xie, Tian, and Zhu 2019).

Por esta misma razón, una tendencia es el uso de datos sintéticos, en lugar de datos reales, durante el proceso de aprendizaje de un algoritmo de *machine learning* (Koch et al. 2019; Ogayar-Anguita et al. 2020).

Una vez comprendida la capacidad y las ventajas que ofrece la simulación, al menos de forma general, es necesario profundizar en el proceso que se propone simular en este trabajo, el cual consiste en el escaneo de un escenario tal y como lo haría un dispositivo LiDAR. El resultado de esta simulación es necesariamente una nube de puntos 3D que representa la estructura de la escena. Por tanto, una de las principales aplicaciones de este tipo de datos es la distinción de aquellos objetos presentes en el escenario capturado. Este problema presenta una gran complejidad, debido a que dicha distinción podría realizarse únicamente en función de la estructura de la superficie intersectada.

Por tanto, se trata de un problema de reconocimiento de objetos 3D, ya sea en forma de clasificación de nubes de puntos (¿qué concepto representa cada punto?) o como segmentación semántica (no es suficiente con hallar el concepto, también debe recuperarse la estructura de dicho concepto/clase). En la actualidad, este conjunto de problemas se resuelve frecuentemente mediante *machine learning*, y más concretamente, mediante *Deep learning* y redes convolucionales (CNN, *Convolutional Neural Network*), las cuales trabajan especialmente bien con información 3D (Karambakhsh et al. 2020; Singh, Mittal, and Bhatia 2019; Wang et al. 2019).

Las aplicaciones del reconocimiento de objetos son muy amplias: desde la conducción autónoma hasta la monitorización de tráfico, labores de vigilancia,

monitorización de zonas naturales (Singh et al. 2019) o agricultura de precisión (detección de maleza, enfermedades, etc) (Milioto, Lottes, and Stachniss 2018).

En definitiva, la simulación de un dispositivo LiDAR permite obtener nubes de puntos sintéticas, donde los marcadores o etiquetas de estos puntos se asignan con total certeza a partir de la clase del objeto capturado, sin necesidad de un etiquetado manual, lo que implica que los datos siempre se encuentran correctamente clasificados, y por tanto, pueden emplearse como entrada de otros algoritmos de inteligencia artificial.

Más allá de esta aplicabilidad potencial, de auge reciente, cabe destacar que este tipo de simulación se documenta en trabajos de investigación de décadas atrás, donde los principales beneficios estaban más bien relacionados con la reducción del coste de adquisición, y la posibilidad de investigar la relación entre valores capturados y condiciones de captura (Goodwin, Coops, and Culvenor 2007; Lovell et al. 2005). Mayoritariamente, se estudiaban entornos forestales mediante modelos de árboles muy simples, donde el principal objetivo era simular el comportamiento del sensor LiDAR frente a la cubierta de vegetación (*canopy*).

Más recientemente, la principal aplicación de la simulación LiDAR es la utilización de datos sintéticos para alimentar algoritmos de inteligencia artificial que sirven como soporte, principalmente, para la conducción autónoma (Lee et al. 2015; Su et al. 2019; Yue et al. 2018). Muchos de los trabajos inmersos en esta temática resuelven el problema del escaneo a través de texturas obtenidas de la captura del entorno mediante una cámara situada en la posición del dispositivo LiDAR, formando lo que se conoce como un *cube map* (Ilustración 3). De esta manera, se puede generar una nube de puntos mediante una transformación que nos permita trasladar puntos 2D en una textura (teóricamente colisionados), al espacio 3D. Por esta misma razón, muchos de los artículos citados describen su solución como un desarrollo en GPU, principalmente debido al *renderizado* de la escena, pero no se utiliza para acelerar otros cálculos.

Ilustración 3. Cube map generado utilizando la primera escena de nuestra aplicación. El primer cube map representa el color de la escena, mientras que el segundo muestra la profundidad en escala de grises.

Este enfoque, aunque eficiente, no simula por completo las físicas de un dispositivo LiDAR, lo que implica que es común introducir errores. Por esta razón, otros trabajos intentan incorporar mejoras sobre esta metodología, como el descarte de colisiones en función de la energía obtenida (Fang et al. 2020). Este mismo artículo propone también otras mejoras tomando como base los resultados obtenidos en el entrenamiento de algoritmos de *deep learning* con nubes de puntos sintéticas, de tal manera que se comprueba que la aplicación en el mundo real de estos algoritmos no funciona tan bien. Por esta razón, propone combinar modelos CAD de gran realismo con entornos escaneados, tales como carreteras.

A diferencia de la solución que se plantea en estos últimos trabajos, algunos de los primeros proyectos hallados en la literatura proponen una simulación basada en *ray-tracing* (Lovell et al. 2005). Ciertamente, esta solución tiene un coste computacional muy alto, un factor mucho más acentuado hace décadas, motivo por el cual los modelos planteados en los entornos forestales eran muy simples. La principal ventaja de una metodología como esta es la precisión de las colisiones obtenidas, dado que no depende de un color (profundidad) en un intervalo muy acotado. Por otro lado, las soluciones que utilizan un *cube map* se adaptan muy bien a un LiDAR empleado en un entorno urbano (por ejemplo, en la conducción autónoma), donde los múltiples retornos de una emisión láser no son tan relevantes. Sin embargo, en el escenario de un bosque, los retornos y la simulación de múltiples propiedades físicas pueden ser muy relevantes.

Este trabajo propone un *framework* de simulación LiDAR, y no únicamente la simulación de un dispositivo muy específico. Además, se adopta un enfoque muy

similar al de las primeras simulaciones. Es decir, el lanzamiento de rayos y la resolución de intersecciones conforman el núcleo de este trabajo. Sobre dicho núcleo se introducirán errores comunes en los dispositivos LiDAR.

Tomando el enfoque descrito, podemos desarrollar el problema del lanzamiento de rayos y la resolución de colisiones. La analogía de este problema con el área de *ray-tracing* es muy evidente, en tanto que es necesario disponer de una estructura de datos muy eficiente que permita resolver millones de intersecciones en un intervalo de tiempo muy reducido. En este caso, el propósito final no es obtener un *rendering* realista, sino generar una nube de puntos a consecuencia de las colisiones identificadas.

La estructura de datos más relevante en el área de *ray-tracing* es el BVH o *Bounding Volume Hierarchy*, un árbol binario donde cada nodo, representado por un AABB (*axis-aligned bounding-box*), puede a su vez contener dos nodos descendientes, que podrán ser hoja (contiene un triángulo, suponiendo que la escena es una composición de mallas de triángulos) o un nodo intermedio (tiene a su vez descendientes) (Anon 2019). En torno a esta estructura podemos encontrar trabajos de investigación recientes cuyo objetivo es, principalmente, la reducción del coste computacional de la técnica de *ray-tracing* (a través de la calidad de la estructura).

Podemos encontrar un gran número de metodologías para su construcción, aunque es posible clasificarlas en tres categorías en función de cómo se construye: desde abajo hacia arriba (mejor calidad), desde arriba hasta abajo (subdivisión), e incremental (inserción). También existen al menos dos criterios que se pueden optimizar en esta fase: tiempo de construcción y calidad. Los primeros algoritmos buscaban reducir el tiempo de construcción (Apetrei 2014; Lauterbach et al. 2009) (siguen siendo los más rápidos), aunque en la actualidad se intentan optimizar ambos parámetros, tiempo y calidad (Hu et al. 2019; Meister and Bittner 2016, 2018).

En la optimización de la calidad del BVH se suele asumir como métrica de coste la heurística SAH (*Surface Area Heuristic*) (MacDonald and Booth 1990), por lo que los esfuerzos de investigación se orientan en otras direcciones. Tampoco es común encontrar modificaciones de la estructura binaria, sea una de las últimas propuestas un árbol dependiente del punto de vista (Gu, 2015). Por tanto, buena parte del esfuerzo se orienta a la generación de la estructura en GPU. En este último aspecto, son comunes los algoritmos relacionados con el *clustering* de primitivas, los cuales se apoyan en ciertas técnicas como la ordenación de primitivas en 3D (Meister and Bittner

2018), heurísticas de densidad para cada primitiva (Hu et al. 2019), estructuras más básicas y de peor calidad, pero también más rápidas (Hendrich et al. 2019), o técnicas tan conocidas como *k-means* (Meister and Bittner 2016) (Ilustración 4).

***Ilustración 4. Representación de clusters mediante diferentes colores. Progresión de la estructura a medida que esta se construye. Fuente:* (Meister and Bittner 2016).**

En relación con el desarrollo de algoritmos en GPU, cabe destacar que es posible acelerar otros muchos algoritmos, más allá del proceso de construcción de un BVH. Para lograr esto, disponemos de múltiples APIs de paralelismo, entre ellas CUDA (Nvidia), OpenCL o *Compute Shaders* (OpenGL). Teniendo en cuenta todas ellas, se selecciona la última citada tomando como referencia las comparativas existentes (Sans and Carmona 2017), pero también su integración con otras soluciones incluidas en el trabajo (OpenGL).

Mediante un BVH es posible realizar una búsqueda muy eficiente para recuperar un triángulo intersectado por un rayo, si lo hubiera. Por tanto, un problema menor que debe igualmente resolverse es la intersección entre entidades tales como un rayo y un triángulo. En este ámbito, no es necesario realizar una extensa revisión bibliográfica, dado que existen métodos estándares definidos y contrastados durante décadas. De esta manera, el algoritmo de Möller-Trumbore (Moller and Trumbore 1998) suele considerarse como un estándar para resolver la intersección rayo-triángulo, aunque es cierto que a partir de esta base es posible incluir mejoras, o desarrollar soluciones adaptadas a otros tipos de modelos, procesos, etc (Baldwin and

Weber 2016). Para resolver la intersección entre un rayo y un AABB (*axis-aligned bounding box*) disponemos de algoritmos como el propuesto por Möller (Tomas Akenine-Möller, 2018), pero también existen otras versiones, como la definida por Martin Eisemann (Eisemann et al. 2007), en la cual se propone una alternativa adecuada para aquellos lenguajes en los que es posible decidir la utilización de un método u otro durante la ejecución.

Por último, para finalizar este estado del arte, se destaca el valor del cálculo de información precisa asociada a la nube, como puede ser el sombreado, en cuyo caso necesitaríamos una técnica de *rendering* realista. En este ámbito, encontramos técnicas que producen resultados bastante realistas, como *ray-tracing* o *ray-marching*. En el primer caso, una complejidad importante radica en el coste computacional, siendo recomendable recurrir a la aceleración de cálculos mediante GPU. En el segundo método, *ray-marching*, no se dispone de una declaración explícita de la estructura de la escena, sino que esta debe definirse mediante funciones matemáticas, lo que dificultaría en gran medida su aplicación a este trabajo. Por ejemplo, la representación de un fractal de Mandelbulb es fácil de llevar a la práctica mediante *ray-marching* (Ilustración 5 e Ilustración 6). Sin embargo, el desarrollo de alguna de estas técnicas no es tan inmediato como para poder resolver un problema de *rendering* y escaneo 3D en el número de horas que se estipula para completar un Trabajo Fin de Máster.

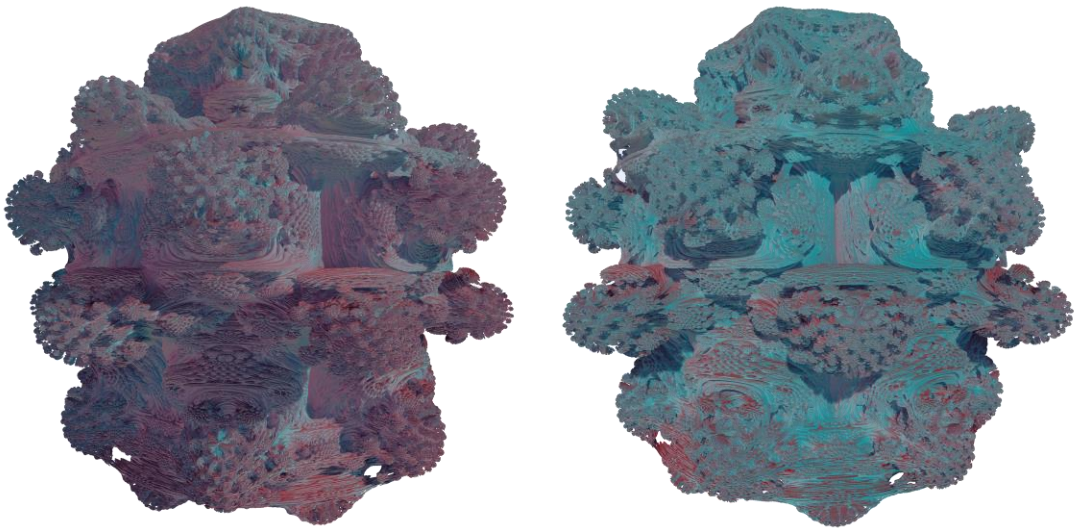


Ilustración 5. Rendering mediante ray-marching de un fractal Mandelbulb. Implementación en la aplicación web Shadertoy[1] e imágenes obtenidas de shadertoy-exporter[2].

[1] https://www.shadertoy.com/
[2] https://github.com/KoltesDigital/shadertoy-exporter

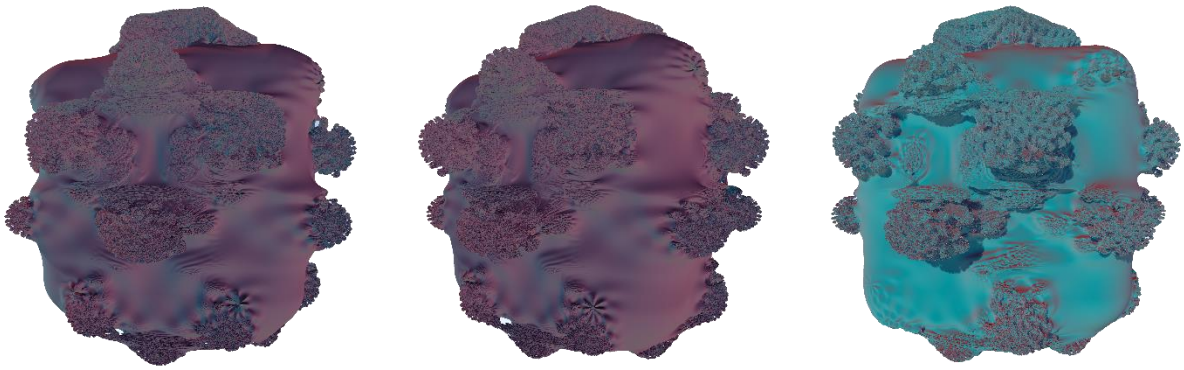


Ilustración 6. Rendering mediante ray-marching de un fractal Mandelbulb combinado con un cubo. La transición suave entre modelos se obtiene empleando SDF (Signed Distance Functions).

1.3 Descripción de la situación de partida

La situación de partida de este trabajo debe expresarse en términos científicos, dado que la solución desarrollada no surge de un sistema previo. Por tanto, esta sección complementa al apartado Antecedentes y estado del arte para permitir al lector de este documento conocer cuáles son las principales aportaciones que se proponen.

1.3.1 Descripción del entorno actual

El entorno científico que podría verse beneficiado de este trabajo viene dado por aquellas áreas de conocimiento donde se identifica la necesidad de utilizar conjuntos de datos tridimensionales con la finalidad de entrenar y comprobar un determinado algoritmo. Se hace esta última distinción dado que existe un amplio número de aplicaciones que emplean datos correspondientes a un espacio 3D, pero no todas ellas toleran la introducción de datos sintéticos. Un claro ejemplo podría ser una aplicación de Realidad Virtual (VR) donde el objetivo es introducir al usuario en un espacio real como si se encontrara en él.

Es necesario aclarar que la situación óptima de estas aplicaciones que podrían tolerar datos sintéticos consistiría en utilizar información espacial de escenarios reales, aunque dentro de este entorno se identifican ciertas carencias que se tratarán en el siguiente apartado.

Dentro de este entorno científico, la aportación más notable se produciría en problemas relacionados con la Inteligencia Artificial, y más concretamente, en problemas tales como la clasificación de nubes de puntos o la segmentación

semántica. Esta aportación se puede representar de manera exhaustiva a través de la siguiente lista:

- Un escaneo simulado permite su ejecución sobre un número infinito de escenarios. La mera inclusión de un escenario generado proceduralmente posibilita obtener un gran número de nubes de puntos diferentes entre ellas. Además, la abstracción de la simulación, para que esta pueda desarrollarse sobre cualquier escena, aumenta el número de posibles escenarios que podrían escanearse. En relación con esto último, cabe destacar que la solución propuesta no se orienta hacia un *framework* que permita introducir un escenario cualquiera, sino que desarrolla la metodología de escaneo, pero claramente podría consistir en un trabajo futuro.
- El propio término “simulación” tiene ciertas implicaciones que se han presentado con anterioridad en la introducción de este trabajo. Simular permite reducir costes, dado que no es necesario un dispositivo físico, y también permite disminuir el tiempo invertido en desplazamientos y capturas de datos.
- A consecuencia de los objetivos propuestos, la simulación del escaneo se debe parametrizar, y por tanto, es posible obtener una nube de puntos flexible, que se puede asemejar en mayor medida a los resultados obtenidos por un determinado tipo de LiDAR. En el estudio genérico de un problema de segmentación semántica, esta flexibilidad puede no representar una ventaja importante, sin embargo, hay otros campos en los que podrían buscarse unas características muy concretas. Este es el caso de la conducción autónoma, donde el dispositivo integrado en el vehículo puede ser conocido, y por tanto, es necesario simular un sensor similar durante la fase de desarrollo o durante las pruebas de un algoritmo de aprendizaje. Por tanto, nos puede interesar una simulación donde pueda controlarse el rango máximo del sensor, o el número de rayos que se lanzan en cada instantánea.
- Obtener un escaneo de una escena modelada (procedural o no) permite conocer qué tipo de objetos se encuentran representados, y por tanto, permite asignar con completa seguridad una determinada clase a un punto de la nube resultante. Más allá de la certeza, una aportación importante es

la capacidad de aumentar el nivel de detalle; actualmente se trabaja con nubes de puntos donde se distinguen algunas clases a un nivel de abstracción alto (vegetación alta, baja, edificio, etc).

- Integración de múltiples tipos de datos en un mismo punto de la nube, tales como intensidad, color RGB o clases semánticas con las que se ha identificado el modelo al que pertenece dicho punto (Ilustración 7). Aunque en este proyecto no se ha contemplado otro tipo de información, sería posible añadir más datos a un mismo punto (por ejemplo, información multiespectral), de tal manera que estos pudieran emplearse para mejorar el proceso de aprendizaje de un algoritmo de Inteligencia Artificial. Nótese como la información añadida debe poder reproducirse en un escenario real, por lo que no deberían incluirse fuentes de datos de las que no dispondremos. El objetivo de las nubes de puntos sintéticas es ayudar en el proceso de aprendizaje de un sistema, pero cuando este se considere suficientemente fiable, se puede emplear en situaciones reales.

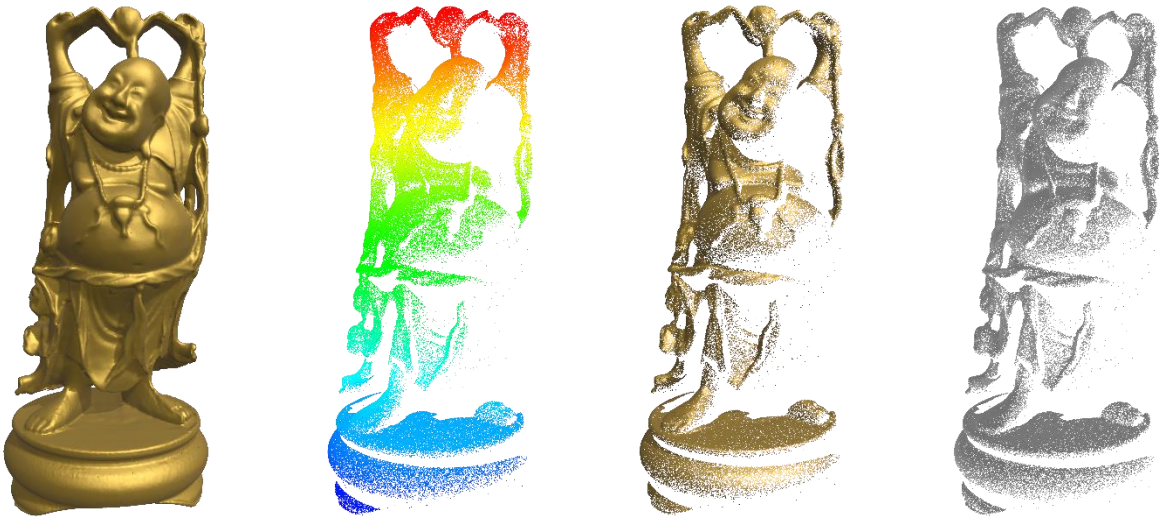


Ilustración 7. Múltiples tipos de datos integrados en una única nube de puntos.

1.3.2 Resumen de las deficiencias y carencias identificadas

Para completar el apartado anterior es necesario conocer qué deficiencias y carencias pretende solventar este trabajo. Para ello debemos trasladarnos a la clasificación semántica de nubes de puntos 3D. Parece evidente que etiquetar una nube de puntos de entrada no es una tarea sencilla: de manera manual puede suponer un trabajo demasiado costoso sin el soporte de herramientas que permitan seleccionar

áreas de puntos, y de manera automática, etiquetar con total precisión no es en absoluto una tarea resuelta (de ser así, no serían necesarias estas nubes de puntos). Por tanto, la primera carencia identificada es la escasez de conjuntos de datos (*datasets*) que pueden servir como soporte para la investigación de esta temática, especialmente cuando se busca una nube de puntos etiquetada de un escenario real. En este contexto, encontramos *datasets* como S3DIS (Armeni et al. 2016), de Stanford, formado por un conjunto de seis escenas de interiores de tres edificios, en las cuales se diferencian hasta trece tipos de elementos (sea uno de ellos *clutter*, para hacer mención al resto de elementos). En este mismo contexto también podemos encontrar nubes de puntos de ciertos objetos, donde el etiquetado es instantáneo, dado que la propia nube sólo engloba puntos de un único objeto. Algunos ejemplos de estos últimos tipos de *datasets* son ShapeNetCore.v2, ShapeNetPart, ModelNet40 o ModelNet10[3].

Ilustración 8. a) Nubes de puntos de los datasets ShapeNetCore.v2, ShapeNetPart, b) nubes de puntos del dataset S3DIS, c) nube de puntos de Semantic3D.net. Imágenes obtenidas de* (Engelmann et al. 2020) *y los pies de página abajo referenciados.

Sin embargo, existen múltiples aplicaciones de la clasificación de nubes de puntos o de la segmentación semántica que se aplican a escenarios exteriores, y donde, por tanto, no son aplicables los *datasets* anteriores. Una de estas aplicaciones puede ser la conducción autónoma, donde se emplean dispositivos tales como LiDAR, de tal modo que la respuesta del vehículo puede depender de la clasificación y la detección de objetos. En este contexto, encontramos *datasets* como Semantic3D.net[4], formado por escenarios urbanos donde se distinguen hasta ocho clases: terreno

[3] https://github.com/AnTao97/PointCloudDatasets
[4] http://www.semantic3d.net/

creado por el hombre, terreno natural, vegetación alta y baja, edificios, *hardscape* (elementos sólidos, no naturales, tales como muros), coches y errores de escaneado.

Como resultado de lo ya expuesto, queda patente la escasez de conjuntos de datos etiquetados existentes, aunque otro problema igualmente importante es la incapacidad de clasificar más allá de las clases identificadas en estos conjuntos de datos, limitando por tanto el ámbito de aplicación.

Más allá de estos problemas, y como se ha destacado en apartados anteriores, la captura de datos (si se decide no partir de *datasets* publicados) puede llegar a ser muy costosa, e incluso pueden existir problemas importantes en los resultados obtenidos. En el caso del sensor LiDAR, la desventaja más destacable es el coste y los errores que pueden obtenerse ante ciertas superficies (*mirror reflection*), pero ninguna de las opciones de captura se encuentra exenta de problemáticas (fotogrametría, RGB-D (RGB + información de profundidad) o SAR (*Interferometric Synthetic Aperture Radar*)) (Xie et al. 2019).

Por tanto, otra deficiencia secundaria en la que puede actuar este trabajo es sobre el coste de adquisición de los datos de entrada, aunque no es tan común esta problemática, dado que se suele acudir a *datasets* publicados (siempre y cuando el escenario sobre el que se desea actuar sea similar a algunos de los ya disponibles).

1.4 Requisitos iniciales

El objetivo de un análisis de requisitos es entender y documentar qué debe hacer un sistema (no cómo). Por tanto, el contenido de este capítulo debe ser una especificación de requisitos enumerados y detallados. Por otro lado, en este punto de documentación del sistema, nos interesa sabér qué se espera de este a un nivel alto de abstracción, debido a que los requisitos específicos se podrían indicar de manera más detallada a lo largo de las iteraciones, mediante historias de usuario.

Un **requisito** se describe como un conjunto de propiedades o restricciones definidas con precisión, pudiendo ser estas abstractas (como las que se identifican este apartado) o más específicas. Para establecer los requisitos es necesario entender qué servicios o utilidades necesita el usuario del sistema, además de las restricciones de funcionamiento asociadas. Se pueden distinguir hasta tres tipos de requisitos:

- **Requisitos funcionales**: describen la funcionalidad del sistema, qué debe hacer o qué se espera que haga.
- **Requisitos no funcionales**: describen aspectos del sistema relacionados con el grado de cumplimiento de los requisitos funcionales.
- **Requisitos de facilidad de uso**: aseguran que hay un buen acoplamiento del sistema con los usuarios y con las tareas que deben realizar.

1.4.1 Requisitos funcionales

Se plantean dos categorías de requisitos funcionales, debido a que el sistema será utilizado tanto por usuarios (clientes) que necesitan obtener un conjunto de nubes de puntos, como por un grupo de desarrolladores al que debe facilitarse su labor. En el desarrollo de este listado se emplean al menos dos expresiones clave: debe, cuando se identifica un requisito principal de la aplicación, y debería, para indicar una sugerencia o propuesta.

- **Requisitos funcionales asociados al usuario**:
  - El sistema debe permitir un movimiento libre tan cómodo como sea posible.
  - La aplicación debe incluir una interfaz donde pueda gestionarse tanto el *rendering* como la configuración de los dispositivos de escaneo.
  - La aplicación debe permitir alternar el modo de *rendering* de la nube de puntos resultante, para poder visualizar todos los datos que pudieran estar contenidos en un punto.
  - El sistema debe disponer de diversos escenarios donde pueda llevarse a cabo una simulación, y siempre que sea posible, se debe desarrollar un escenario procedural en lugar de estático.
  - La aplicación debe permitir modificar los parámetros de todas las superficies representadas en un escenario.
  - El sistema debe permitir exportar una nube de puntos resultante del proceso de simulación.
  - El sistema debería permitir definir rutas cuando el dispositivo de escaneo se encuentra en el medio aéreo.

- **Requisitos funcionales asociados al desarrollador**:
    - La aplicación debe permitir un movimiento libre tan cómodo como sea posible. De hecho, debería seguir una estructura de movimiento similar a la de un *framework* contrastado y conocido por los desarrolladores.
    - La aplicación debe permitir visualizar las estructuras de datos asociada a la escena.
    - El sistema debe disponer de una interfaz donde sea posible modificar las propiedades de las superficies definidas, con el fin de comprobar la corrección de los algoritmos desarrollados.
    - El sistema debe permitir configurar el proceso de simulación, así como iniciar múltiples simulaciones en una misma ejecución.
    - La aplicación debe soportar la generación de capturas de pantalla de tamaño variable que sirvan como soporte para documentar el desarrollo.
    - El sistema debería permitir visualizar los rayos emitidos por el dispositivo, al menos en etapas tempranas de desarrollo.
    - La aplicación debería disponer de un escenario animado que permita comprobar el rendimiento de la simulación.

1.4.2 Requisitos no funcionales

A continuación se describen algunas de las restricciones identificadas como consecuencia de la naturaleza de la solución:

- La aplicación debe minimizar el tiempo de respuesta. Desde el punto de vista del usuario, el tiempo de carga de la aplicación y el tiempo de respuesta de la simulación deben minimizarse. Desde el punto de vista del desarrollador, también se cumple esta necesidad, aunque es más importante el tiempo de carga del sistema, con el fin de facilitar su depuración.
- La aplicación debe utilizar la mínima cantidad de memoria posible, posibilitando así ejecutar la aplicación en un gran número de equipos.

- El sistema, y más concretamente su interfaz, debe ser fácil de interpretar e intuitivo para cualquier usuario, adaptándose a los patrones o modos de utilización de aplicaciones más comunes en la actualidad (por ejemplo, no introducir botones difícilmente visibles).
- Se debe garantizar que la curva de aprendizaje presenta la mayor pendiente posible, y por tanto, que el tiempo necesario para dominar el sistema sea de unos pocos minutos.
- El sistema debe ser tolerante a errores para evitar que la aplicación se cierre bruscamente sin dar información alguna.
- Se debe garantizar que el tamaño de los archivos multimedia, por ejemplo, imágenes o modelos, es óptimo, para que el tiempo de respuesta (carga) no sea excesivo.

1.4.3 Requisitos de facilidad de uso

Los requisitos de facilidad de uso especifican el grado en que usuarios específicos puede conseguir objetivos específicos de forma eficaz, eficiente, confortable y aceptable dentro de un entorno concreto. Para elaborar este listado de requisitos es necesario evaluar en profundidad las características de los usuarios, las tareas, factores situacionales (qué situaciones se pueden dar) y criterios de aceptación del sistema.

El usuario principal de la aplicación queda definido como un investigador cuyo principal estímulo es la obtención de nubes de puntos etiquetadas, por tanto, no se identifica una necesidad especial en la interacción del usuario con la aplicación. Uno de los requisitos funcionales identificados era la introducción de un movimiento libre y cómodo en la escena, por lo que tanto este requisito como la introducción de una interfaz simple e intuitiva pueden ser suficientes para asegurar un aprendizaje eficiente.

En cualquier caso, el sistema que en este proyecto se desarrolla no se concibe como un producto final, sino más bien como un trabajo experimental que pretende definir un algoritmo para simular un proceso real. Por esta razón, los requisitos de facilidad de uso pueden adoptar un matiz de sugerencia.

1.5 Alcance

El alcance del proyecto permite delimitar los resultado del trabajo, es decir, todos los entregables del mismo. En este apartado se contemplan resultados vinculados al proyecto de programación, pero también aquellos solicitados en la guía de este proyecto y vinculados a la entrega de un Trabajo Fin de Máster:

1. Código fuente de la aplicación, donde se puede encontrar la implementación de todos aquellos algoritmos que aquí se describen, tanto en lenguaje de programación C++ como GLSL. Puede suponer un buen punto de partida para trabajos futuros, razón por la cual se documenta mediante el estándar de Doxygen, además de aquellos comentarios que se han considerado convenientes para comprender la solución.
2. Ejecutable del proyecto, el cual permite comprobar la solución desarrollada sin necesidad de compilar el código fuente aportado.
3. Documentación del proyecto, sea esta el mismo documento sobre el que se escriben estas líneas, con el único fin de describir los algoritmos desarrollados, justificar las decisiones realizadas, exponer las tareas de gestión y control del proyecto y dar acceso a manuales de uso e instalación de la aplicación.
4. Recursos de los que depende el fichero ejecutable: *assets*. Dentro de estos recursos se encuentran los modelos 3D o las texturas utilizadas.
5. Vídeo de documentación de la aplicación, donde se representa el funcionamiento de la aplicación y se describen los principales conceptos. Puede enfocarse como un resumen del desarrollo del trabajo.
6. Librerías utilizadas en el proyecto. Al tratarse de algunas carpetas de tamaño reducido, que no necesitan de instalación, se pueden incluir en la entrega del trabajo para evitar invertir tiempo en esta tarea en trabajos futuros.

1.6 Hipótesis y restricciones

En la planificación, la principal restricción aplicable se encuentra en la duración total del trabajo, que será de 300 horas, dado que este se enmarca dentro de un

Trabajo Fin de Máster, al cual le corresponden 12 créditos. Por tanto, a excepción del mantenimiento, todas las etapas del ciclo de vida quedan acotadas por este número de horas.

Dentro del propio desarrollo del trabajo, las principales restricciones se encuentran en la simulación, y más concretamente, en la representación de errores frecuentes en el tipo de escaneo seleccionado. La bibliografía es la única fuente de la cual podemos recuperar dichos errores, y por tanto, la similitud entre el proceso a desarrollar y la realidad depende de la precisión con la que se describen estas situaciones en la literatura hallada. Tampoco se dispone de los dispositivos necesarios para llevar a cabo el escaneo de un escenario, aunque otra fuente válida de datos puede ser la observación de escaneos publicados en la web, con la dificultad de que a partir de estos no se puede conocer la dimensión del error.

Por último, la principal hipótesis que guía este trabajo es el uso predominante de datos de dispositivos LiDAR en los trabajos relacionados con la clasificación de nubes de puntos 3D (Xie et al. 2019), razón por la cual el proceso de escaneo que se simula es el de este dispositivo.

1.7 Estudio de alternativas y viabilidad

Este apartado permite describir qué alternativas se han considerado y justificar la elección realizada, teniendo en cuenta que las alternativas descartadas podrían ser de interés en trabajos futuros. En primer lugar, cabe destacar que la solución que se esboza desde un comienzo se encuentra muy clara, y por tanto, no se plantean demasiadas alternativas, dado que es suficiente con una aplicación donde se *renderice* una escena y una interfaz, y se permita al usuario interaccionar mediante entradas tales como teclado o ratón.

Algunas de las principales alternativas planteadas surgen en torno a la tecnología empleada, aunque esta discusión se establece en el apartado de Tecnologías utilizadas. Además, surgen varias alternativas cuando se plantean los tipos de escenarios a representar.

- Escenario de interior (habitación), siendo este un buen escenario de partida para comenzar una aplicación gráfica por estar compuesto únicamente de modelos 3D importados a los que se aplica una secuencia de

transformaciones geométricas estáticas. Además, supone un escenario perfecto para utilizar un LiDAR terrestre.

- Entorno forestal, el cual tiene un gran valor para simular el funcionamiento de un LiDAR aéreo y sus errores, debido al amplio número de superficies que aporta.
- Entorno urbano, donde se podría simular una escena de conducción autónoma. Además, supondría un buen ejemplo de la utilización de un LiDAR terrestre en una de las aplicaciones de mayor futuro. Otro punto a favor es que se considera necesario incluir una escena animada para comprobar la frecuencia con la que se podría capturar el escenario (experimentación).
- Cualquier otro escenario animado que nos permita igualmente comprobar la frecuencia máxima de captura en la experimentación. Por ejemplo, un sistema de partículas podría ser válido.

Finalmente, se desarrollan todos los escenarios propuestos a excepción del entorno urbano, especialmente considerando las restricciones temporales de este proyecto. Nótese como el modelado de un sistema de partículas es mucho más simple que el de un entorno procedural como el propuesto (más allá del nivel de realismo que se quiera aportar).

Por último, se consideran múltiples alternativas en el algoritmo de resolución del escaneo 3D, aunque por formar parte del desarrollo se ha preferido extender dichas alternativas como una introducción en la primera iteración del apartado de Desarrollo. A grandes rasgos, se puede optar por resolver el problema mediante estructuras de datos y algoritmos de intersección entre formas geométricas, o mediante capturas realizadas por una cámara situada en la posición del dispositivo LiDAR, es decir, mediante texturas.

1.8 Descripción de la solución propuesta

Esta descripción breve de la propuesta realizada permite conocer cuáles son los detalles más significativos de la solución final, mientras que para conocer cómo se ha resuelto esta propuesta es necesario desplazarse al apartado de Desarrollo.

La necesidad planteada inicialmente se puede describir como una aplicación gráfica que permite simular el escaneo 3D, concretamente de un dispositivo LiDAR, con un mínimo tiempo de respuesta, simulando los principales errores descritos en la bibliografía hallada, y permitiendo al usuario controlar cualquier aspecto de este proceso (desde la asignación de un número de rayos hasta la selección de los errores a simular).

Al tratarse de un trabajo experimental, la complejidad recae principalmente en los algoritmos a desarrollar, y por tanto, el producto final se puede describir en muy pocos términos. Así, la solución planteada consiste únicamente en una ventana compuesta por un escenario *renderizado* mediante alguna API y una interfaz que se muestra sobre dicho escenario (Ilustración 9). La API, así como el resto de tecnología a emplear, se describe en el apartado Tecnologías utilizadas.

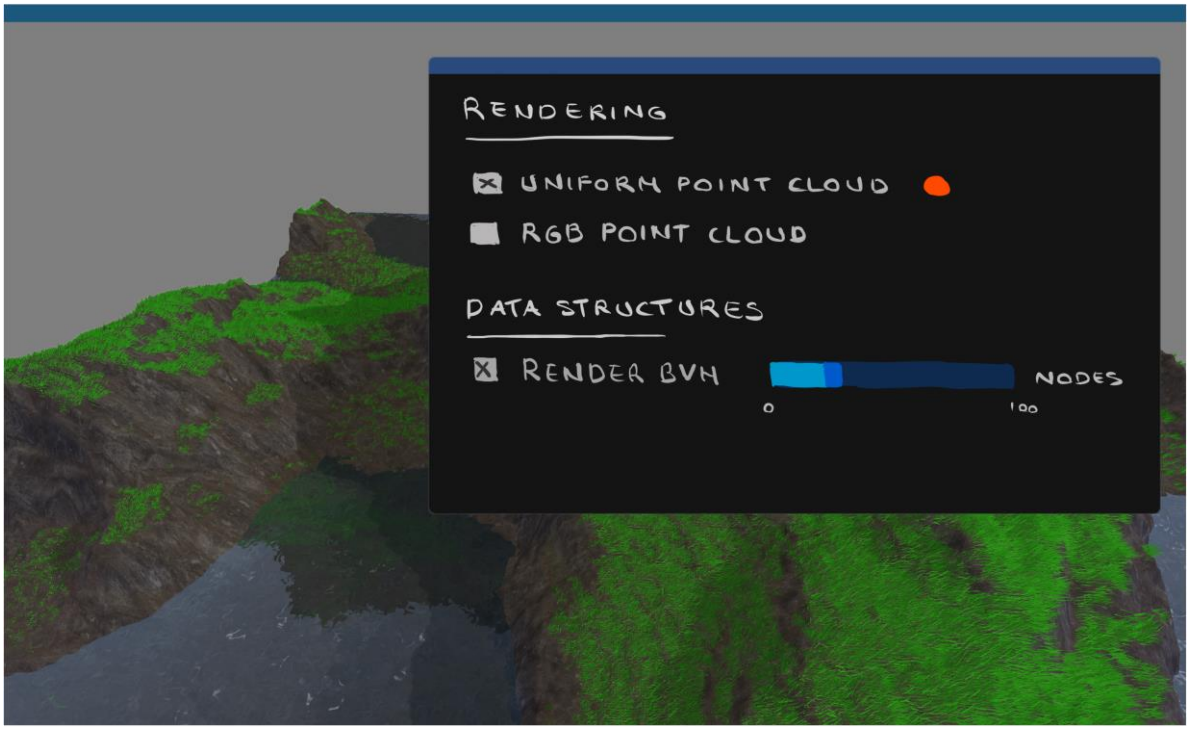


Ilustración 9. Boceto de una ventana de la solución final, donde se muestra un escenario y una interfaz para controlar la visualización o un proceso de la aplicación.

La solución seleccionada se caracteriza, a grandes rasgos, por los siguientes puntos:

- Dispone de al menos dos escenarios, para adaptarse a la captura aérea y terrestre.
- El escenario es capaz de mostrar una nube de puntos adquirida mediante el escaneo 3D en cualquiera de sus formatos, los cuales se rigen por el

color: nube de color uniforme, color dependiente de la clase de objeto impactado, color dependiente de la altura del punto dentro de la caja envolvente de la nube, etc (Ilustración 10).

- Incluye un modelo que permite visualizar la localización del dispositivo de escaneo, con el fin de permitir al usuario planificar correctamente el proceso.
- La aplicación es capaz de mostrar información gráfica del proceso, como las estructuras de datos utilizadas, o los rayos lanzados por el dispositivo.
- La interfaz tiene su núcleo en un menú situado en la parte superior de la ventana, desde donde se permite el acceso a otras ventanas en las cuales se controlan al menos dos aspectos: la visualización (qué y cómo), y los parámetros del proceso de escaneo.
- Además de un escaneo terrestre y aéreo automático, la interfaz permite al usuario dibujar una ruta de vuelo.
- La aplicación permite completar el proceso de escaneo de manera incremental, visualizando la nube a medida que esta se genera. No se trata de una animación, sino que es el usuario quien debe controlar el avance de los incrementos.

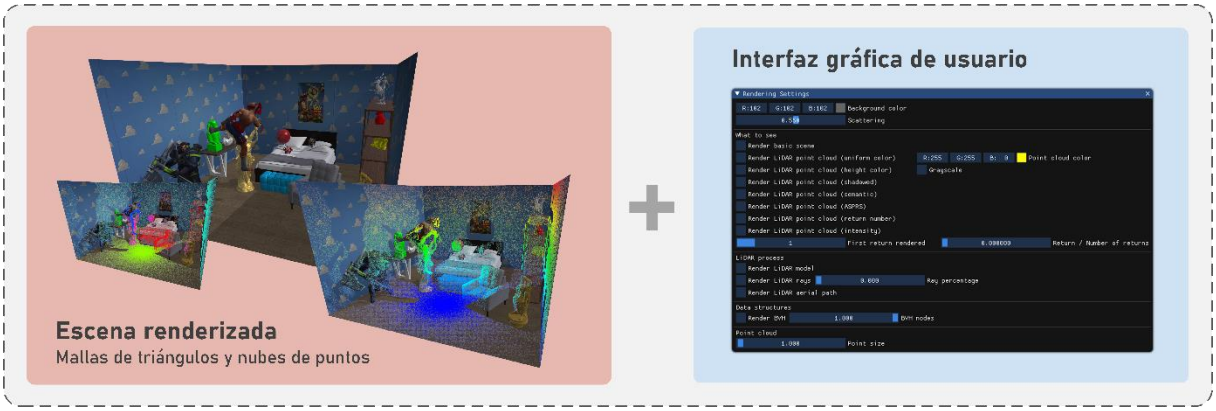


Ilustración 10. Componentes gráficos de la solución final.

Por último, es necesario destacar que la solución descrita se identifica como la única forma de resolver el problema que se plantea, dado que es un escenario muy básico. A medida que profundicemos en los detalles de la aplicación sí podrán surgir puntos discusión acerca de cuál es la mejor alternativa, comenzando por el capítulo de Tecnologías utilizadas.

1.9 Tecnologías utilizadas

Las tecnologías adoptadas en la solución final se citan y justifican a continuación, diviéndose todas ellas en función de la utilidad que aportan a este trabajo.

- Lenguaje de programación: **C++**. Se trata de uno de los principales lenguajes de programación empleados en aplicaciones de Informática Gráfica. Además, la utilización de otras librerías e interfaces de programación citadas en este apartado justifican esta decisión.

- Interfaz de programación para renderizar gráficos 2D y 3D: **OpenGL**[5] (*Open Graphics Library*). Es una API multilenguaje y multiplataforma que permite interactuar con la tarjeta gráfica para obtener un *rendering* con aceleración hardware. La selección de esta solución viene principalmente justificada por la experiencia previa, habiéndose utilizado en múltiples asignaturas del Grado en Ingeniería Informática. Aunque es cierto que existen otras interfaces de programación para gráficos que pueden presentar algunas ventajas respecto a OpenGL, tales como Vulkan o DirectX, se ha considerado que iniciarse en una API diferente excedía en gran medida el esfuerzo que requiere un Trabajo Fin de Máster, especialmente considerando el tiempo de asimilación de conceptos tales como *pipelines*, *shaders* u otras peculiaridades que pudieran asociarse al *framework*.

  La versión de OpenGL utilizada es la 4.5, cuya fecha de lanzamiento se remonta a 2014. Nótese como la versión viene limitada por el hardware empleado; en este caso, una tarjeta gráfica GeForce GTX 1070 no soporta la última versión de OpenGL, 4.6, la cual se remonta a 2017[6].

- Gestor de contexto y ventanas. Se ha empleado **GLFW**[7], una librería *Open Source* y multiplataforma para OpenGL, OpenGL ES (*OpenGL for Embedded Systems*) y Vulkan, siempre en aplicaciones de escritorio. No sólo permite crear la ventana donde se represente una escena, sino que también gestiona las entradas de usuario, así como los eventos que se producen sobre este entorno (por ejemplo, un cambio de tamaño en la

[5] https://www.opengl.org/
[6] https://www.khronos.org/registry/OpenGL/index_gl.php
[7] https://www.glfw.org/

ventana). La elección de esta solución es discutible, pero se debe afrontar dicha discusión conociendo las siguientes herramientas.

- Carga y consulta de extensiones de OpenGL en el equipo: **GLEW**[8] (*OpenGL Extension Wrangler Library*). Se trata de una librería multiplataforma escrita en C++ que permite conocer qué extensiones de OpenGL se encuentran disponibles. Conforma el núcleo de nuestra solución, pero no modificaría ningún aspecto de la misma emplear otra solución.
- Librería de operaciones matemáticas de *rendering*: **GLM**[9] (*OpenGL Mathematics*). La principal de ventaja de esta librería es la similitud con el lenguaje empleado en los *shaders* de OpenGL (GLSL*, OpenGL Shading Language*), dado que se basa en sus especificaciones. Más allá de esa base, incluye otras operaciones con matrices, quaterniones, números aleatorios o ruido. Es sin duda una de las librerías de matemáticas más completas en el entorno de OpenGL, por lo que no es necesario contemplar alternativa alguna.
- Computación paralela: ***compute shader***. Se ha optado por emplear *compute shaders* para tareas de computación paralela, considerando que estos *shaders* se utilizan para operaciones de ámbito general, y no necesariamente relacionadas con *rendering*. El principal factor para emplear esta tecnología y no otra es la utilización de la API de OpenGL (integración automática), aunque ciertamente se podrían haber utilizado otras plataformas de computación paralela, tales como CUDA (*Compute Unified Device Architecture*) u OpenCL (*Open Computing Language*).
- Carga de imágenes PNG: **LodePNG**[10]. Se trata de un codificador/decodificador de imágenes PNG sin dependencia alguna respecto de otra librería. Existen múltiples opciones para realizar esta tarea (SOIL, SAIL, etc), pero la decisión se justifica en este caso por la facilidad de integración y el tamaño mínimo de la librería, teniendo en cuenta que nuestras imágenes seguirán el formato PNG (de no disponer de dicha

[8] http://glew.sourceforge.net/
[9] https://glm.g-truc.net/
[10] https://lodev.org/lodepng/

certeza, se pueden recurrir a alternativas multi-formato, como las ya mencionadas).

- Interfaz gráfica de usuario (GUI, *Graphical User Interface*): **Dear ImGui**[11]. Se trata de una librería desarrollada en C++ que permite integrar puntos de interacción con el usuario dentro del *rendering* de la aplicación. Dos aspectos que se intentan potenciar son la simplicidad y la productividad, permitiendo así obtener iteraciones muy rápidas y acotadas en el tiempo. Se considera una librería de más bajo nivel que otras soluciones, dado que la interfaz se integra dentro del *pipeline* de *rendering*, y por tanto, se representa como cualquier escenario de nuestra aplicación. Este es quizás uno de los principales puntos de discusión.

- Generación de ruido: **FastNoise SIMD**[12]. La principal ventaja de esta librería es la implementación paralela de un gran número de algoritmos de generación de ruido en múltiples dimensiones. Además, dispone de una herramienta donde es posible visualizar la textura que se obtendría ante un elevado número de parámetros, por lo que es posible escoger aquella que mejor se adapte a un escenario (con un conjunto de parámetros asociados).

- Almacenamiento de nubes de puntos: **tinyply**[13]. Se trata de una librería que nos permitirá almacenar los resultados de un escaneo en un fichero de formato PLY (*Polygon File Format*). La estructura que define permite almacenar múltiples elementos, cada uno de ellos caracterizado por un conjunto de atributos. En nuestra aplicación, sólo un elemento será necesario (la nube). La elección de esta solución se justifica a partir de la flexibilidad que ofrece, debido a que no define un conjunto estático de atributos, y por tanto, nos permite introducir información de la nube que no es comúnmente considerada. Otra solución que podría ser de gran utilidad es PDAL[14], una librería que permite almacenar ficheros LAS siguiendo la especificación 1.4, donde se definen los conceptos semánticos que aquí se aplican. Como desventaja, los atributos contenidos son estáticos (a pesar

[11] https://github.com/ocornut/imgui
[12] https://github.com/Auburn/FastNoiseSIMD
[13] https://github.com/ddiakopoulos/tinyply
[14] https://pdal.io/

de que dispone de múltiples formatos de registro), y por tanto, nos impediría guardar información considerada en la aplicación. Por esta razón, se descarta la utilización de esta última solución.

Una vez presentada toda la tecnología anterior, cabe justificar su elección, dado que una alternativa razonable sería el uso de un *framework*, y más concretamente, cabe hacer mención de Qt[15], un *framework* multiplataforma orientado a desarrollar aplicaciones con una interfaz gráfica de usuario. Se establece este punto de discusión dado que un *framework* como el citado puede llegar a concentrar toda la tecnología listada anteriormente en una misma herramienta, debido a que es capaz de gestionar ventanas y entradas de usuario, realizar las mismas funciones de GLEW, integrar *widgets* de OpenGL para el *rendering* e incluye módulos de matemáticas y carga de imágenes. Por todas estas ventajas, las primeras versiones de este proyecto empleaban Qt, aunque finalmente se adoptaron todas las librerías citadas.

La principal razón por la que finalmente se descarta Qt es el control sobre el dibujado, dado que es el *framework* el que gestiona ese evento. Nótese como no todos los *renderizados* tienen por qué producirse dentro de dicho evento. De manera clara, una escena deberá redibujarse cuando se considere necesario refrescar el *widget* dedicado a su representación, pero no sucede esto mismo cuando el objetivo es generar, por ejemplo, una textura en un momento inicial de la aplicación a través de un *renderizado offscreen* y su posterior captura de imagen. Todo esto aquí expuesto no implica que no sea posible lograr tal tarea en un *framework* como Qt, dado que en las primeras versiones de este proyecto se desarrollaron tareas como esta, pero suelen encontrarse más problemas. Más allá de esto, Qt es una herramienta conocida por disponer de una gran documentación, algo que no sucede para sus *widgets* de OpenGL. Por ejemplo, la captura de una ventana emplea clases propias, y no es fácil recuperar información acerca de cómo lograr esto mismo, especialmente cuando la captura es más avanzada (por ejemplo, con *antialiasing*). En definitiva, la principal problemática encontrada en este *framework* es la capacidad de utilizar cualquier función de OpenGL con total seguridad en cualquier punto de la aplicación.

Por otro lado, la comparativa a nivel de interfaz gráfica de usuario entre Dear ImGui y Qt se puede abstraer como una comparativa entre *retained mode* e *inmediate mode*, respectivamente. Mientras que en *inmediate mode* el estado de la aplicación

[15] https://www.qt.io/

se diferencia claramente de la librería gráfica (y por tanto, de la interfaz), en *retained mode* no hay una diferenciación clara (Microsoft, 2018). De esta manera, en *inmediate mode* sólo es necesario *renderizar* cuando se produce un cambio. Por tanto, en este intercambio de tecnología realizado, este es el punto más débil identificado, dado que una interfaz como ImGui se *renderiza* junto al resto de la escena: si el usuario interacciona con un control cualquiera, es necesario redibujar una escena que, en nuestro caso, puede incluir millones de triángulos y puntos.

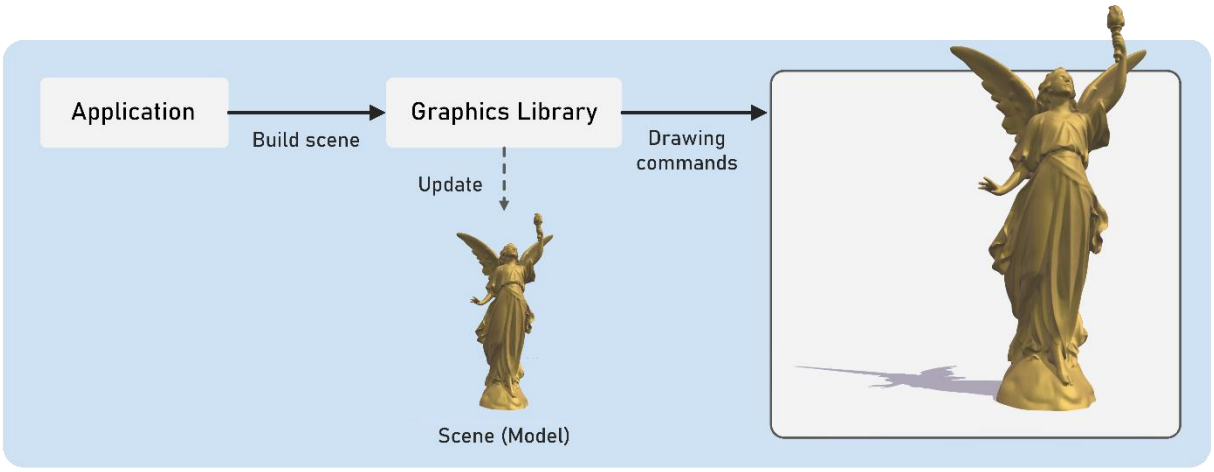


Ilustración 11. Esquema de una interfaz gráfica de tipo retained mode, donde la interfaz se integra en el rendering de un frame.

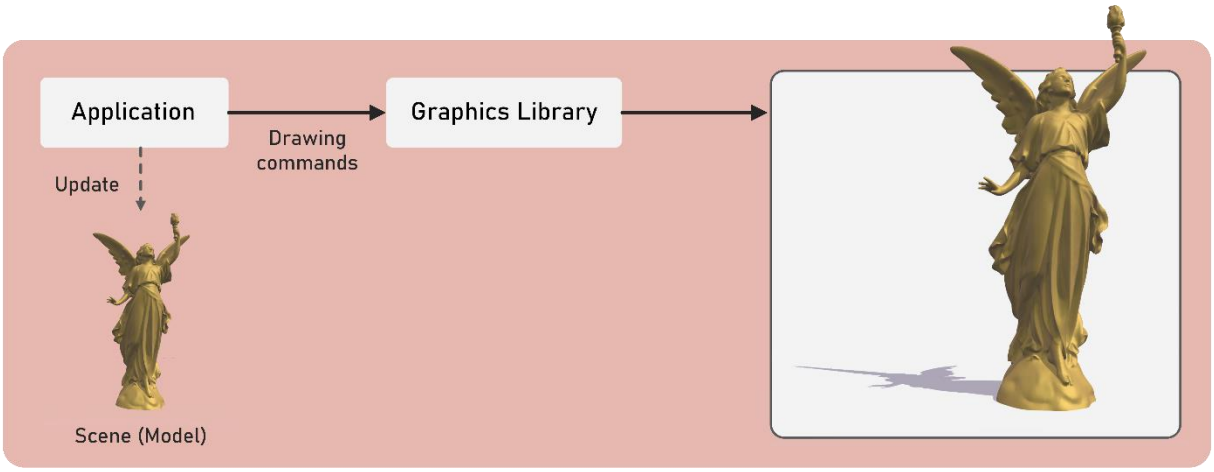


Ilustración 12. Esquema de una interfaz gráfica de tipo inmediate mode, donde el estado de la aplicación se diferencia de la parte gráfica.

Una alternativa a Qt y a todas estas herramientas citadas se encuentra en los motores de videojuegos, como Unity3D[16] o Unreal Engine[17], los cuales incorporan conceptos gráficos básicos que en nuestro trabajo han debido desarrollarse desde

[16] https://unity.com/
[17] https://www.unrealengine.com/

cero (sombras, reflejos, etc). También se trata de *frameworks* que se encuentran lo suficientemente optimizados como para poder representar escenarios muy detallados en tiempo real (al menos, esta es la tendencia). El único factor que nos hace descartar entornos como los mencionados es la flexibilidad que ofrecen; por ejemplo, la creación de las estructuras que van a representarse, como mallas de triángulos, no suele ser tan evidente e inmediata. Por otro lado, enfrentarnos al problema que se plantea únicamente mediante OpenGL nos ofrece flexibilidad máxima para optimizar el problema. Así, es posible almacenar en GPU una estructura que representa un vértice con cualquier tipo de información que consideremos conveniente.

Además de las tecnologías integradas en la solución, también se han empleado otras herramientas que han permitido el desarrollo de la solución final, las cuales se han seleccionado principalmente en base a la experiencia previamente desarrollada:

- Entorno de desarrollo (IDE, *Integrated Development Environment*): Visual Studio 2019.
- Edición y transformación de modelos 3D: Blender y Autodesk Maya. Normalización de modelos para adecuarse a unos requisitos mínimos establecidos para todos los modelos de la aplicación.
- Edición de imágenes: Adobe Photoshop. Manipulación de texturas empleadas en la aplicación gráfica.
- Documentación de proyecto: Microsoft Word, Microsoft Visio, Microsoft Power Point. Software empleado para explicar conceptos que se han desarrollado en la programación, tanto a nivel textual como gráfico.
- Planificación temporal: Gantt Project. Permite crear, asignar y distribuir tareas en un diagrama de Gantt.
- Creación de esquemas de Ingeniería del Software: Visual Paradigm.

Todas estas últimas herramientas externas (no integradas dentro del propio desarrollo de programación) deben contemplarse en la planificación económica de este trabajo, dado que la mayoría de herramientas citadas no son gratuitas.

1.10 Metodología de desarrollo de software

El objetivo de este apartado es presentar la metodología de desarrollo software empleada, considerando que cualquier proyecto, y más concretamente, un proyecto Ingeniería, debe seguir una clara metodología para lograr el resultado esperado. La selección de una metodología influye en la organización de los siguientes apartados de este documento, y por tanto, es vital seleccionar una metodología adecuada desde el comienzo.

En la selección de la metodología se deben considerar las circunstancias en las que desarrolla el proyecto. Al tratarse de un trabajo experimental donde los objetivos no se logran mediante soluciones naïve, es necesario plantear reuniones puntuales con los directores de este trabajo, con el fin de establecer nuevos objetivos y/o revisar el desarrollo obtenido hasta ese momento. Por tanto, parece evidente que este Trabajo Fin de Máster tiene un claro carácter iterativo (e incremental).

De esta manera, una metodología que refleja dichas circunstancias es la metodología ágil, la cual se basa en la construcción de pequeños incrementos de un producto software, que además, deben verificarse. Se trata de una posibilidad que ofrece el software y que no se da en otras áreas relacionadas con la ingeniería, donde no existe tal capacidad de responder a necesidades y requisitos diferentes (Shore & Chromatic, 2007) (Launch School, 2020).

Este tipo de desarrollo se puede sintetizar en un conjunto de conceptos dados por la publicación “Manifesto for Agile Development” (Ken Beck et al., 2001), el cual surge como reacción frente al modelo de desarrollo predominante en la década de los 90 en la Ingeniería del Software: el modelo en cascada. El principal problema de este modelo es el esquema de flujo de tareas que propone, dado que es completamente lineal, con una nula capacidad de retroceder a tareas anteriores y realizar cambios con facilidad. A estos problemas mencionados, es necesario añadir la dificultad de planificar con gran precisión un proyecto en las primeras etapas, así como la lentitud en la obtención de resultados (al menos, de cara al cliente, el cuál sólo accede al producto final).

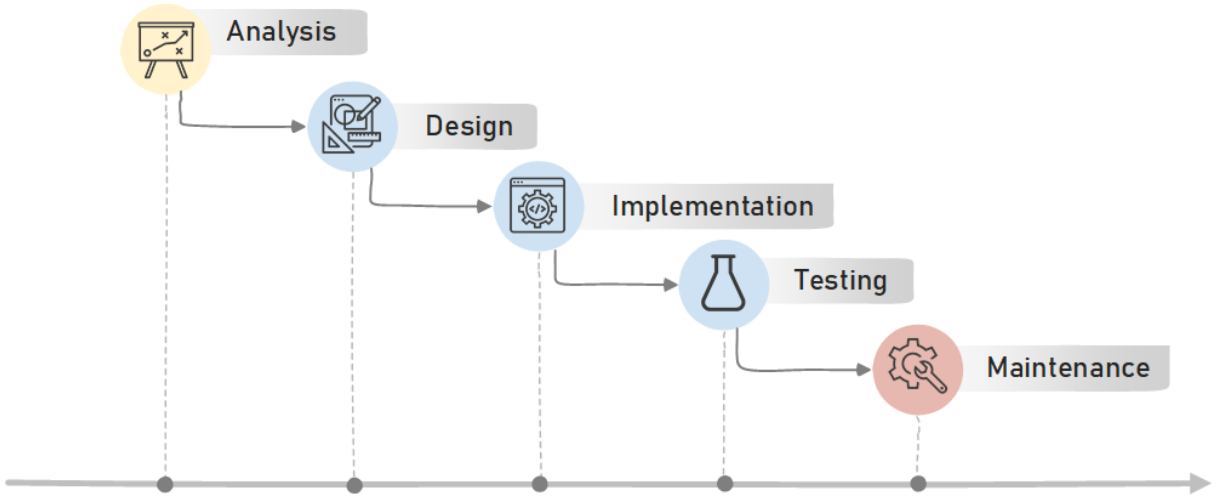


Ilustración 13. Esquema de una metodología de desarrollo en cascada.

En definitiva, el desarrollo ágil se sustenta sobre cuatro principio básicos (Ken Beck et al., 2001):

- Personas e interacciones por encima de procesos y herramientas. Es evidente la importancia de emplear métodos y herramientas adecuadas, pero la interacción y comunicación entre el propio equipo puede llegar a ser más eficiente que cualquier herramienta software.
- Software en funcionamiento por encima de una documentación exhaustiva. Se pueden generar pequeñas soluciones respecto del producto final para obtener el *feedback* del cliente, de tal manera que se minimiza la planificación inicial y el software se auto-documenta a través de su diseño y las pruebas realizadas (por ejemplo, de aceptación).
- Colaboración del cliente por encima de la negociación de un contrato. Aunque es cierto que deben existir contratos firmados por las dos partes, es necesario involucrar al usuario dentro del ciclo de desarrollo, eliminando así la distancia que pueda existir entre ambos. Esta colaboración se puede materializar a través de tests de usabilidad o simplemente reuniones (*stakeholders* o algún representante de la parte del cliente, *product owner*).
- Responder ante cambios es mejor que seguir una planificación. Como se describía en la metodología en cascada, realizar una planificación inicial precisa es muy difícil. Por tanto, se prefiere una situación que pueda asimilar correctamente los cambios que puedan producirse.

Dentro de esta metodología de desarrollo software, existen múltiples marcos

de trabajo, tales como Scrum o XP (*Extreme Programming*), sea el primer marco el adoptado en este trabajo. Se trata de de un marco de trabajo bajo el que se engloban un conjunto de procesos y técnicas, siendo sus principales pilares transparencia (visibilidad para responsables del producto), inspección (para detectar variaciones) y adaptación. Los principales conceptos se agrupan bajo los términos de equipo de Scrum, eventos, roles y artefactos.

Debido a las limitaciones de este Trabajo Fin de Máster en cuanto a tamaño de equipo y restricciones temporales, no todos los conceptos del marco Scrum son aplicables. Por tanto, a continuación se enumeran aquellos conceptos que sí lo son y se han aplicado en el desarrollo de este trabajo (Schwaber & Sutherland, 2017):

- *Sprint*: es el núcleo de Scrum, de tal manera que el carácter incremental vendrá definido por *sprints* de una duración aproximada de un mes, al final de los cuales se debe obtener un producto usable. Lo más común es que los *sprints* tengan una duración constante a lo largo del proyecto, aunque es posible considerar inicialmente una duración variable para ajustarla posteriormente en base a observaciones.

  En nuestro caso, no se ha establecido coherencia alguna en la complejidad de los objetivos propuestos en cada uno de los *sprints*, por tanto, no se asegura una duración constante, aunque la extensión final de cada uno de ellos se ha aproximado al mes de desarrollo antes propuesto.

- Planificación de *sprint*, con el objetivo de plantear qué se va a desarrollar en la siguiente iteración y definir claramente cuál es la aplicación que se espera obtener al final del mismo.

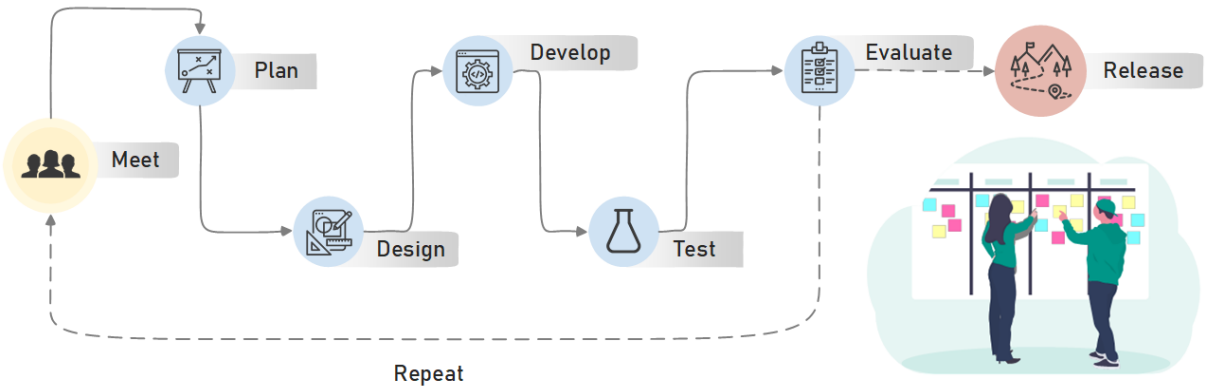


Ilustración 14. Esquema del ciclo de vida de una metodología ágil basada en iteraciones.

- **Roles de un equipo Scrum**: *product owner*, como representante de la parte cliente, equipo de desarrollo, y *Scrum Master*, que no representa la figura de un líder de equipo, dado el equipo debe auto-organizarse, pero sí representa la figura encargada de facilitar la aplicación de esta metodología, gestionar cambios, etc (y por tanto, debe conocer en profundidad este marco de trabajo).

  Es evidente que no todos estos roles se pueden representar en este trabajo, pero sí que se ejercen ciertas tareas de cada uno de ellos. No se puede desechar por completo la figura del *product owner*, dado que en cierto modo son los directores de proyecto quienes pueden evaluar el trabajo y conocen las funcionalidad básicas que deben encontrarse en la solución final. Igualmente, también participan en el equipo de desarrollo, al menos de manera teórica, mientras que la figura del *Scrum Master* no se aplica dado que no es necesario controlar la correcta ejecución de este marco de trabajo, y en cualquier otra circunstancia, sería el desarrollador principal.

- ***Product Backlog***: es una lista ordenada de todos aquellos requisitos que debe reunir el producto final, y por tanto, debe completarse o ser modificada a medida que avanza el proyecto. En este trabajo, el alcance y los objetivos inicialmente definidos son nuestra referencia para comprobar la completitud del proyecto.

1.11 Análisis de riesgos

La identificación de riesgos nos permite conocer qué problemas podemos encontrar durante en la ejecución del proyecto, pudiéndose completar este análisis tanto en una etapa inicial (planteamiento) como a lo largo del desarrollo. No es necesario seguir una metodología concreta para que este apartado sea de suficiente utilidad; puede ser de gran ayuda una simple matriz donde se identifiquen incertidumbres, junto a su probabilidad de ocurrencia y su impacto, y se propongan alternativas.

De hecho, estos son algunos de los principios básicos de una **matriz riesgo**. Dicha matriz permite obtener el nivel de riesgo a partir de la probabilidad de ocurrencia y las consecuencias de ese mismo riesgo. Estadísticamente, el nivel de riesgo se

puede calcular multiplicando la probabilidad de ocurrencia por un factor de gravedad de las consecuencias identificadas. Sin embargo, en la práctica puede ser difícil estimar ambos valores con precisión.

No existe una única matriz estándar que defina unos mismos niveles, probabilidades e impactos, y por tanto, la matriz que aquí se emplea es sólo una posibilidad (Talbot, 2018; Goddard Space Flight Center, 2009). La matriz de referencia que se emplea para clasificar los riesgos identificados es la Tabla 12, donde se indica mediante un código de color el nivel de riesgo correspondiente para una combinación Impacto-Probabilidad. Además, el impacto se acompaña de algunos indicadores que podrían sernos útiles, como el retraso en días que podría provocar un determinado riesgo, o su impacto en la completitud de los requisitos. Otros indicadores, como las pérdidas económicas, se han descartado dado que la principal pérdida procedería de extender el contrato del autor del trabajo, y en cualquier caso, este se encuentra acotado temporalmente. También se han mantenido otros indicadores, a modo de ejemplo, que no son necesariamente aplicables a nuestro proyecto. Este es el caso del impacto en la reputación, dado que no se trabaja para un cliente concreto.

A partir de la matriz de la Tabla 12 podemos valorar el nivel de riesgo que suponen aquellos puntos débiles identificados en este trabajo, con la finalidad de conocer cuáles son más prioritarios. Para cada uno de los riesgos identificados se indica tanto la probabilidad como el impacto del mismo en los términos expresados en la Tabla 12.

Valoración de riesgos			
Riesgo identificado	**Impacto**	**Probabilidad**	**Nivel**
Los miembros del equipo no tienen el conocimiento y/o habilidades requeridas para llevar a cabo el proyecto	Muy alto	Muy baja	
Desarrollo incorrecto de la interfaz de usuario	Bajo	Muy baja	
Se producen cambios significativos en objetivos, alcance, o requisitos planificados, inmediatamente después del inicio del proyecto	Alto	Muy baja	
Baja capacidad del equipo para adaptarse a tecnologías en las que no se posee experiencia	Alto	Baja	

Pérdida o daño de proyecto desarrollado hasta cierto instante de tiempo	Muy alto	Baja
Tiempo de desarrollo insuficiente considerando los objetivos propuestos	Alto	Media
Plazos planificados incompatibles con el cronograma del proyecto	Medio	Media
Rendimiento pobre del sistema.	Alto	Media
Dificultad para hallar material relativo a la documentación de errores del dispositivo de escaneo seleccionado	Medio	Media
Planificación y/o gestión ineficaz.	Alto	Media

Tabla 1. Riesgos identificados, junto a su impacto, probabilidad y nivel de riesgo obtenido.

Nótese como el impacto, y especialmente la probabilidad, no se asignan en función de unos criterios claramente objetivos, y por tanto, ambos factores pueden interpretarse de manera diferente por otro autor. La probabilidad se ha seleccionado en base a nuestra experiencia previa, mientras que el impacto se fundamenta principalmente en las consecuencias que pudiera ocasionar en la calidad del producto y en la planificación temporal.

Para completar el análisis de riesgos no es suficiente con una valoración del nivel de riesgo de cada punto problemático identificado, sino que también es necesario proponer una acción a llevar a cabo si se diera alguno de estos riesgos. De manera previa a la descripción del plan de contingencia, se indicará cuál es la acción a realizar: aceptar, mitigar, evitar o transferir. Por tanto, no todos los riesgos deben disponer de un plan alternativo, dado que entre las acciones a realizar encontramos la aceptación. Otro factor importante es la diferenciación entre evitar y mitigar; sólo se ha considerado la primera acción cuando el plan de contingencia elimina por completo el riesgo, lo cual no siempre sucede.

Estrategia y plan de contingencia	
Riesgo identificado	**Los miembros del equipo no tienen el conocimiento y/o habilidades requeridas para llevar a cabo el proyecto.**
Estrategia	Mitigar
Plan de contingencia	Formación del equipo de manera previa al comienzo del proyecto, si se tratara de un problema de conocimiento. Contratación de personal cualificado cuando el riesgo procede de las habilidades del equipo.
Riesgo identificado	**Desarrollo incorrecto de la interfaz de usuario.**
Estrategia	Mitigar
Plan de contingencia	Realización de esquemas que planteen la resolución de la interfaz en base a los requisitos presentados. Se debe refinar a lo largo del desarrollo del proyecto.
Riesgo identificado	**Se producen cambios significativos en objetivos, alcance, o requisitos planificados, inmediatamente después del inicio del proyecto.**
Estrategia	Mitigar
Plan de contingencia	Documentar objetivos, alcance y requisitos en el inicio del proyecto, y asegurar su aceptación por parte del equipo de trabajo.
Riesgo identificado	**Baja capacidad del equipo para adaptarse a tecnologías en las que no se posee experiencia.**
Estrategia	Mitigar
Plan de contingencia	Comprobar capacidad de equipo de trabajo con las nuevas tecnologías propuestas. En caso de no darse la adaptación, planificar proyecto de acuerdo a tecnologías conocidas. Igualmente, la formación o la contratación de personal cualificado pueden ser buenas medidas.
Riesgo identificado	**Pérdida o daño de proyecto desarrollado hasta cierto instante de tiempo.**

Estrategia	Evitar
Plan de contingencia	Implantar el uso de un sistema de control de versiones desde el inicio del proyecto, así como unas políticas de documentación que permitan encontrar efizcamente cualquier punto del desarrollo.
Riesgo identificado	**Tiempo de desarrollo insuficiente considerando los objetivos propuestos.**
Estrategia	Aceptar (sólo cuando la ejecución del proyecto es para un TFM)
Plan de contingencia	Si bien es recomendable que el proyecto se limite a una planificación inicial y se acote a las horas de dedicación propuestas para un Trabajo Fin de Máster, es posible extender tiempos (excepto en el caso de que el proyecto se ejecute de manera externa y exista un cliente).
Riesgo identificado	**Plazos planificados incompatibles con el cronograma del proyecto.**
Estrategia	Mitigar
Plan de contingencia	Al tratarse de un Trabajo Fin de Máster, existe cierta flexibilidad en los plazos, por lo que el cronograma y los plazos deben refinarse durante el desarrollo y con la aceptación del equipo. Como sucediera en el riesgo anterior, este plan no se ajusta a una ejecución externa del proyecto.
Riesgo identificado	**Rendimiento pobre del sistema.**
Estrategia	Mitigar
Plan de contingencia	En el inicio, identificación de puntos que podrían afectar al rendimiento del sistema, para que el equipo puede prestar especial atención a los mismos. Posteriormente, dirigir esfuerzo de desarrollo y mejora sobre aquellas partes del programa en las que se invierte una mayor cantidad de tiempo o se emplea más memoria.
Riesgo identificado	**Dificultad para hallar material relativo a la documentación de errores del dispositivo de escaneo seleccionado.**

Acción a realizar	Mitigar
Plan de contingencia	Realizar un trabajo de búsqueda de bibliografía al inicio del proyecto, de tal manera que si esta se considera insuficiente se puede recurrir al hardware de escaneo adquirido para comprobar en primera persona qué tipo de errores son frecuentes.
Riesgo identificado	**Planificación y/o gestión ineficaz.**
Estrategia	Mitigar
Plan de contingencia	Planificación/gestión por parte de personal con experiencia suficiente. En cualquier caso, se reconoce que el mayor riesgo es invertir una gran cantidad de tiempo en una planificación ineficaz, por lo que puede definirse un plan que se refine durante el desarrollo del proyecto.

Tabla 2. Estrategia y plan de contingencia para cada uno de los riesgos antes identificados.

1.12 Estimación del tamaño y esfuerzo

La estimación de tamaño y esfuerzo del proyecto tiene como objetivo conocer la extensión del mismo, así como el número de efectivos que son necesarios para su ejecución. Más allá de esta estimación, sabemos que un Trabajo Fin de Máster se encuentra acotado temporalmente (un número aproximado de 300 horas, correspondientes a 12 créditos) y también a nivel de esfuerzo, dado que sólo se dispone de un efectivo, que es el autor del trabajo. Es decir, independientemente de la estimación resultante, habrá que considerar los límites expuestos.

El cálculo de esta estimación se puede llevar a cabo mediante múltiples modelos, sea uno de los más conocidos el modelo **COCOMO** (*COnstructive COst MOdel*). Este modelo fue publicado por Barry Boehm en 1981, utilizando entonces datos de 63 proyectos (que serían 163 en COCOMO II). La finalidad de estos datos no es otra que ajustar una formula de regresión de la cual se derivan los parámetros del modelo (Kemerer 1987).

Dentro de COCOMO coexisten hasta tres modelos (Universidad del País Vasco):

- **Básico**, del cual se obtienen estimaciones poco precisas, aunque puede ser útil cuando no hay demasiada información del proyecto. Se rige por la ecuación más básica:

$$Esfuerzo = a * (Miles\ de\ líneas\ de\ código)^b \quad (1.1)$$

$$Tiempo\ de\ desarrollo = c * Esfuerzo^d \quad (1.2)$$

  sean a, b, c y d cuatro valores dependientes del tipo de proyecto.

- **Intermedio**, para el cual se deben ajustar hasta 15 factores de coste dentro de los 6 niveles existentes (de Muy bajo a Extremadamente alto).
- **Detallado**, el modelo más preciso, debido a que considera cada componente del sistema de manera individual y es capaz de adaptarse a cada etapa del proyecto.

Además, se distinguen hasta tres tipos de proyectos:

- **Orgánico**, de tal modo que el proyecto se desarrolla en un entorno estable, introduce poca innovación técnica y puede contener hasta varias decenas de miles de líneas de código (menos de cincuenta mil, si es necesaria una referencia).
- **Empotrado**, sea este un proyecto con restricciones importantes, volátil, complejo y de gran innovación técnica.
- **Semilibre**, un proyecto que toma características de los dos anteriores.

Para comenzar con la estimación es necesario seleccionar un modelo COCOMO, así como el tipo de proyecto que mejor se ajuste a este trabajo. En primer lugar, el modelo seleccionado es el **intermedio**, debido a que el modelo básico es demasiado impreciso, y el modelo detallado requiere un conocimiento avanzado de etapas en las que no se ha profundizado, como el diseño del producto, dada la restricción temporal que introduce este trabajo. En segundo lugar, el tipo de proyecto seleccionado es el **semilibre**, dado que se ha considerado que el autor del trabajo no dispone de una experiencia previa en algunas de las tecnologías que se van a emplear, aunque el entorno no es especialmente inestable. Ciertamente, también existe un factor de innovación importante, aunque no se establece un nivel de complejidad excesivamente elevado.

El siguiente paso en un modelo intermedio es la valoración de algunos conductores de coste, hasta dieciséis. Es necesario destacar que la omisión de alguno de ellos es posible, dado que se trata de factores entre 0 y 1 cuya finalidad es simplemente escalar un valor inicial de 1, de tal manera que valores mayores de 1 aumentan el esfuerzo.

Valoración de conductores de coste	
Conductor de coste	**Valoración**
Fiabilidad requerida del software	Nominal (1)
Tamaño de la base de datos	Bajo (0,94)
Complejidad del producto	Alto (1,15)
Restricciones del tiempo de ejecución	Muy alto (1,3)
Restricciones del almacenamiento principal	Muy alto (1,21)
Volatilidad de la máquina virtual	Bajo (0,87)
Tiempo de respuesta del ordenador	Bajo (0,87)
Capacidad del analista	Nominal (1)
Experiencia en la aplicación	Alta (0,91)
Capacidad de los programadores	Muy alta (0,7)
Experiencia en S.O. utilizado	Alto (0,9)
Experiencia en el lenguaje de programación	Alto (0,95)
Prácticas de programación modernas	Alto (0,91)
Utilización de herramientas software	Muy alto (0,83)
Limitaciones de planificación del proyecto	Muy alto (1,10)
Factor multiplicador	0,5823

Tabla 3. Valoración de conductores de coste contemplados en los modelos COCOMO intermedio y detallado.

Una vez se obtiene el factor multiplicador a partir de la valoración de los conductores de coste, se expone la fórmula que se aplica sobre los modelos

COCOMO intermedio y detallado. Esta fórmula es muy similar a la anteriormente expuesta, aunque incorpora el factor que acabamos de calcular, *M(x)*, y emplea un subíndice i que, para el modelo detallado, supone el identificador de una etapa concreta (análisis de requisitos, diseño, programación, etc).

$$Esfuerzo_i = M(x) * a_i * (Miles\ de\ líneas\ de\ código)^{b_i} \quad (1.3)$$

El único valor que aún desconocemos es el número de líneas de código, que en el momento en que se escribe este trabajo es de 30.123, de donde 22.655 se corresponden a ficheros en lenguaje C++, y 7.468 se corresponden a líneas de código de *shaders* (GLSL).

Por último, los coeficientes a, b, c y d vinculados a un proyecto de tipo semilibre son 3,00, 1,12, 2,50 y 0,35 respectivamente (Tabla 14). Si aplicamos todos estos valores a las fórmulas anteriormente expuestas, se obtienen los siguientes resultados:

$$Esfuerzo = 0{,}5823 * 3 * 30{,}123^{1{,}12} = 79{,}183\ efectivos/mes$$

$$Tiempo\ de\ desarrollo = 2{,}5 * 72{,}96^{0{,}35} = 11{,}54\ meses$$

$$\frac{72{,}96\ \frac{efectivos}{mes}}{11{,}22\ meses} = 6{,}86\ efectivos \approx 7\ efectivos$$

Un proyecto de las características definidas en la Tabla 3 necesitaría de 7 efectivos al mes durante 11,54 meses, aunque como bien se expuso anteriormente, estas condiciones de ejecución no se pueden dar debido a la existencia de restricciones temporales y de efectivos (un único autor).

1.13 Planificación temporal

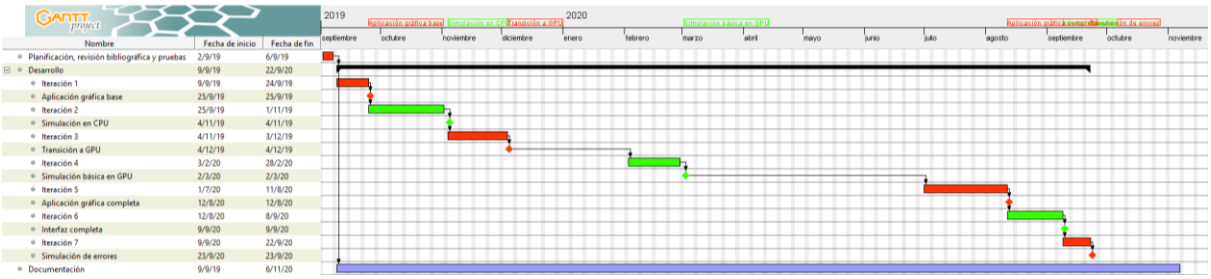
La planificación temporal de un proyecto permite representar el avance del mismo a lo largo de una dimensión temporal. Esta planificación se debe adaptar a la metodología software elegida, la cual es una metodología ágil, y por tanto, no hablaremos de tareas, sino de iteraciones, hitos y resultados.

Un diagrama comúnmente utilizado para representar la distribución de iteraciones en el tiempo (en otro caso, tareas) es el diagrama de Gantt, de tal manera que para cada iteración se muestra su extensión temporal con un nivel de detalle ajustable (días, semanas, meses, etc). En su forma original, las tareas representadas no tienen dependencias entre ellas, pero esto es algo que sí se ha incorporado en

herramientas más recientes. Para la elaboración de este diagrama se ha seleccionado la herramienta de código abierto GanttProject.

Antes de mostrar la planificación en forma de diagrama de Gantt, es necesario destacar que el tiempo de dedicación a este proyecto se ha fraccionado en tres intervalos, debido a que el desarrollo del mismo coincide en tiempo con el máster al que este trabajo pertenece. De esta manera, hay determinados meses donde el desarrollo frena por completo, lo que implica principalmente que en su reanudación es necesario un pequeño intervalo de tiempo para retormar conceptos.

El grado de prioridad de las tareas de cada iteración se muestra mediante el código de color expuesto en la Tabla 4.



Nombre	Fecha de inicio	Fecha de fin
Planificación, revisión bibliográfica y pruebas	2/9/19	6/9/19
Desarrollo	9/9/19	22/9/20
Iteración 1	9/9/19	24/9/19
Aplicación gráfica base	25/9/19	25/9/19
Iteración 2	25/9/19	1/11/19
Simulación en CPU	4/11/19	4/11/19
Iteración 3	4/11/19	3/12/19
Transición a GPU	4/12/19	4/12/19
Iteración 4	3/2/20	28/2/20
Simulación básica en GPU	2/3/20	2/3/20
Iteración 5	1/7/20	11/8/20
Aplicación gráfica completa	12/8/20	12/8/20
Iteración 6	12/8/20	8/9/20
Interfaz completa	9/9/20	9/9/20
Iteración 7	9/9/20	22/9/20
Simulación de errores	23/9/20	23/9/20
Documentación	9/9/19	6/11/20

Ilustración 15. Diagrama de Gantt donde se muestra la ejecución de hasta siete iteraciones.

Prioridad de iteraciones	
Prioridad alta	
Prioridad media	
Prioridad baja	

Tabla 4. Código de color para la prioridad de las iteraciones y tareas.

En el diagrama de la Ilustración 15 se identifican hasta siete iteraciones, más dos tareas, sea una de ellas la documentación del proyecto, la cual se puede desarrollar en paralelo desde el comienzo del mismo. En cualquier caso, esta tarea se podría considerar que está incluida en todas y cada una de las iteraciones consideradas. En la finalización de todas las iteraciones se ha situado un hito, el cual indica el estado de la aplicación en dicho punto.

El escenario ideal permitiría dividir cada una de estas iteraciones en pequeñas tareas, sobre las cuales se podría hacer un seguimiento diario a través de un diagrama *Burndown*, con el fin de indicar el número de tareas completadas frente al número de

tareas idealmente finalizadas en un instante de tiempo concreto. Sin embargo, no se ha hecho un seguimiento diario en los últimos meses de desarrollo.

Para finalizar este apartado se enumeran las iteraciones de la Ilustración 15 con una descripción que indica qué tareas se encuentran comprendidas. También se indica la duración de cada iteración y una medida de esfuerzo, que se halla en el intervalo [0, 10], de tal manera que después es posible representar la dificultad relativa de las iteraciones.

1. **Planificación, revisión bibliográfica y pruebas**. No se debe interpretar como una iteración, sino como una tarea para establecer unos objetivos claros y obtener información de la materia. Por ejemplo, era necesario cerciorarse de que la necesidad que este trabajo pretende superar realmente existe.
    - Duración: 5 días.
    - Esfuerzo: 2.
2. **Iteración 1**. Desarrollo de una aplicación gráfica que conforma la base del proyecto. No implica un desarrollo de gran esfuerzo dado que muchos de los conceptos que se incorporan se consideran asimilados, y por tanto, no se parte de cero. Sobre esta base previa se añaden conceptos observados en la bibliografía de esta área (Akenine-Möller 2018).
    - Duración: 12 días.
    - Esfuerzo: 3.
3. **Iteración 2**. Desarrollo de una primera simulación en CPU. Se construye sobre estructuras de datos y algoritmos conocidos de antemano, aunque en el área de las intersecciones geométricas es necesario acudir a otros algoritmos desconocidos hasta este momento. En algunos de ellos, que se conciben como estándares de la materia, se reconocen fallos en el proceso algorítmico, razón por la cual se prolonga esta iteración más de lo planificado.
    - Duración: 28 días.
    - Esfuerzo: 5.

4. **Iteración 3**. Se produce una transición de los algoritmos en CPU hacia GPU (unidad de procesamiento gráfico) una vez se observa el elevado tiempo de respuesta de la versión anterior. Muchos de los algoritmos desarrollados anteriormente deben descartarse, dado que las posibilidades que se abren son muy diferentes. Se comienza con el desarrollo de algoritmos en GPU para la generación de la geometría y la topología de los modelos 3D.
    - Duración: 22 días.
    - Esfuerzo: 5.
5. **Iteración 4**. Se inicia la creación de un escenario alternativo (terreno), a la vez que se desarrolla un algoritmo base de captura de la escena (no incluye la simulación de errores).
    - Duración: 20 días.
    - Esfuerzo: 7.
6. **Iteración 5**. Con una simulación básica disponible, el esfuerzo se dirige en completar la aplicación gráfica: reflejos y refracciones en superficies, lagos (incorporan igualmente estos dos últimos conceptos), etc. Igualmente, se crea un escenario alternativo con animaciones.
    1. Duración: 30 días.
    2. Esfuerzo: 6.
7. **Iteración 6**. Introducción de la interfaz gráfica de usuario en la aplicación. Desarrollo de las múltiples ventanas que permiten al usuario interaccionar y visualizar datos de la aplicación. La iteración debe comenzar necesariamente con un pequeño estudio de la librería empleada, hasta ese momento desconocida. También se incluyen algunos tipos de *rendering* de nubes de puntos.
    - Duración: 20 días.
    - Esfuerzo: 3.

8. **Iteración 7**. Simulación de errores en el escaneo LiDAR. Buena parte de esta iteración depende de la revisión de bibliografía donde se describen los principales errores observados y la magnitud de los mismos.
    1. Duración: 10 días.
    2. Esfuerzo: 5.
9. **Documentación**: como bien se indica anteriormente, no se trata de una iteración, sino de una tarea que se inicia muy temprano y que se completa cuando se considere que este documento alcanza una calidad mínima.
    1. Duración: 25 días.
    2. Esfuerzo: 5.

Por último, se presenta un esquema que muestra de manera gráfica el esfuerzo relativo que ha supuesto cada iteración y tarea citada:

Ilustración 16. Representación gráfica del esfuerzo de las iteraciones contempladas en el diagrama de Gantt.

1.14 Presupuesto

La elaboración de un presupuesto tiene como objetivo calcular el coste de ejecución del proyecto, así como conocer qué medios, materiales y servicios son necesarios. Por esta misma razón, este apartado se dividirá en tantos capítulos como recursos se reconozcan, habiéndose identificado al menos los siguientes:

- **Hardware**, contemplando en este punto todos aquellos recursos informáticos empleados.
- **Software**. Se incluyen todas las herramientas software que han permitido desarrollar el proyecto.
- **Recursos humanos**, de tal manera que aquí se puede introducir el coste asociado a la contratación de personal.
- **Costes indirectos**. Para concluir el presupuesto, se incluyen gastos derivados del uso de instalaciones, facturas de servicios, etc. Igualmente, ante la dificultad de calcular el coste de ciertos servicios y recursos, se puede incluir un sobrecoste equivalente a cierto porcentaje de la cantidad acumulada, con el fin de sufragar el coste de los recursos restantes.

1.14.1 Coste hardware

El coste hardware vinculado a este proyecto procede, en primer lugar, del equipo de desarrollo, de tal manera que deberían ser necesarios tantos equipos como empleados se incluyen en este mismo presupuesto. Además, se ha considerado necesario incluir el coste de dispositivos de escaneo, como un LiDAR, dado que puede ser un buen punto de partida para conocer los errores más frecuentes. Se puede considerar un gasto prescindible si los conjuntos de datos y artículos publicados son suficientes para comprender los problemas que pudieran surgir en torno al sensor. En cualquier caso, conocer los errores y su dimensión no es una tarea fácil, a pesar de disponer del dispositivo.

El coste del equipo de desarrollo se obtiene a partir de la factura del equipo empleado en este trabajo, adquirido de APP Informática.

Componente hardware	Coste estimado (€)
Bloque 1.1. Equipo de desarrollo	
Intel core i7.6700 3.4GHz 8M LGA 1151 Procesador	324,1
MSI 1151 Z170-A PRO Placa base	117,4
W. Digital 1TB 3.5 WD10EZEX SATA3 7200 64MB Disco duro	54,9

Kingston HyperX Savage DDR4 16GB 2666 CL13 Memoria RAM	109,5
Torre ATX NOX Coolbay SX Carcasa de torre	47,1
Cooler Master Refrigeración. CPU HYPER 212X INTEL Ventiladores	37,4
Gigabyte GeForce NVIDIA GTX 1070 WF2 OC 8GB DDR5 Tarjeta gráfica	532,2
Asus Grabadora 24X Sata Energy DR W24F1ST Grabadora	14,8
Samsung 2.5 256GB SATA3 Serie 850 PRO Disco duro SSD	148
NOX Urano VX 650W ATX Fuente de alimentación	54,6
ASUS VS247NR 23.6" Full HD Negro Reacondicionado Monitor	98
Teclado mecánico CM Storm Teclado	119,9
Keep Out X4 Ratón	29
Total Bloque 1.1. Hardware	1.686,9

Tabla 5. Desglose de componentes hardware del equipo de desarrollo empleado. Para cada componente se indica su coste estimado.

El coste estimado aquí propuesto, extraído de una factura, se corresponde al precio de estos componentes cuatro años atrás. Por tanto, el valor de cada uno de ellos es menor del que aquí se presenta, aunque si se decidiera ejecutar este proyecto debería adquirirse un equipo con componentes más recientes. De esta manera, el presupuesto propuesto debe servir como referencia del coste de un equipo con una capacidad suficiente para completar las tareas de desarrollo.

El hardware restante procede de aquellos dispositivos necesarios para la captura terrestre y aérea de un escenario, con el fin de analizar los errores que se pueden producir. Como bien se aclaraba anteriormente, esta segunda partida de presupuesto puede ser prescindible, pero en ningún caso puede invalidar una de las motivaciones principales de este proyecto, que era la reducción de costes de aquellos trabajos que necesitan nubes de puntos como datos de entrada, dado que es el

proyecto de investigación el que adquiere el material, mientras que investigaciones posteriores podrían emplear simplemente los resultados de la simulación.

En cuanto a la fuente de obtención de estos datos, el coste estimado de los dispositivos relacionados con el escaneo aéreo procede de la web de DJI[18], uno de los principales fabricantes de vehículos aéreos no tripulados, mientras que el coste del dispositivo terrestre procede de Leice Geosystems[19]. El coste estimado del dron se obtiene de la web SphereDrones[20], a pesar de ser un dispositivo de DJI, dado que no se suelen aportar precios hasta contactar con el distribuidor. Por simplicidad, se ha considerado que el precio aportado por estas fuentes no incluye impuestos, y por tanto, se ha considerado un coste adicional del 21% respecto de lo ya acumulado.

Dispositivo	Coste estimado (€)
Bloque 1.2. Dispositivos de escaneo	
DJI Matrice 210 Vehículo aéreo no tripulado	11.343,60
Livox Mid-40 Sensor LiDAR para incorporarlo al vehículo aéreo	599
LEICA BLK360 Sensor LiDAR terrestre	16.000
Total Bloque 1.2. Dispositivos de escaneo	27.942,6 x 1,21 = 33.810,54

Tabla 6. Dispositivos de escaneo que podrían ser necesarios para el desarrollo de la simulación, junto al coste detallado por el fabricante de cada dispositivo.

1.14.2 Coste software

El software utilizado en este trabajo incluye todas aquellas herramientas que han posibilitado su ejecución, desde un entorno de desarrollo hasta el editor de texto que ha posibilitado su documentación. Es necesario considerar que buena parte del software empleado dispone de licencias temporales o permanentes para estudiantes, pero no se puede suponer que dichas licencias estarán operativas en la ejecución del proyecto por parte de un grupo externo. Por tanto, se desglosan los precios suponiendo el peor de los escenarios, donde no se aplica ninguna ventaja.

[18] https://www.dji.com/
[19] https://leica-geosystems.com/
[20] https://spheredrones.com.au/

Todos los costes representados en esta tabla proceden de la página oficial de cada una de las herramientas propuestas, aunque se ha evitado añadir el enlace de dichos precios en el pie de página. También cabe destacar que se ha optado por priorizar el uso de licencias anuales sobre mensuales, excepto en aquellos casos en los que no ha sido posible (por ejemplo, Github Team).

Software	Tipo de pago	Coste estimado (€)
Bloque 2. Herramientas software		
Windows 10 Home Sistema operativo	Único	145
Visual Studio Community 2019 Entorno de desarrollo	No aplicable	0
Autodesk Maya Edición de modelos 3D	Anual	2.136
Alternativa: Blender Edición de modelos 3D	No aplicable	0
Adobe Photoshop Edición de imágenes	Mensual	24,19 x 8 = 193,52
Alternativa: GIMP Edición de imágenes	No aplicable	0
Microsoft 365 Family Documentación y presentación	Anual	99
Github Team Control de versiones	Usuario, Mensual	14 x 8 = 112
Visio Profesional 2019 Creación de diagramas	Único	809
Visual Paradigm Professional Creación de diagramas de Ingeniería Software	Único	999
Gantt Project Planificación temporal	No aplicable	0
	Total Bloque 2. Herramientas software	4.493,52

Tabla 7. Listado de herramientas software necesarias para la ejecución del proyecto, así como el coste estimado correspondiente a las licencias necesarias para su utilización.

1.14.3 Coste de recursos humanos

Para estimar el coste derivado de la contratación de personal se considera que este Trabajo Fin de Máster sólo consta de un autor. Por tanto, sólo nos queda asignar un coste estimado, para lo cual se utiliza el XVII Convenio colectivo estatal de empresas de consultoría y estudios de mercado y de la opinión pública (Ministerio de Empleo y Seguridad Social, 2018). El rol seleccionado es el de Analista Programador (Grupo C, Nivel I, Área 3: Consultoría, Desarrollo y Sistemas), para el cual se recupera el salario total que entra en vigor a partir del 31 de diciembre de 2019. En cuanto al rol, se elige dicha posición para indicar que la labor de planificación y diseño de la aplicación recae sobre el único autor del trabajo.

Categoría	Número	Salario anual (€)	Coste estimado (€)
Bloque 3. Recursos humanos			
Analista programador Único autor del proyecto	1	24.640,37	24.640,37 x 3 / 4 = 16.426,91
Paga extra semestral	1	-	2.053,36
Seguridad social	1	-	18.480,27 x 0.33 = 6.098,49
		Total Bloque 3. Recursos humanos	24.578,76

Tabla 8. Personal identificado y coste estimado correspondiente a su salario.

Para completar el cálculo del coste asociado al personal de desarrollo, se añade una paga semestral adicional (considerando que la duración del proyecto es de 8 meses, de acuerdo con la planificación temporal). También se debe incluir el coste asociado a la Seguridad Social, que se calcula como el 33% del coste acumulado (incluyendo pagas adicionales).

1.14.4 Costes indirectos

Los costes derivados incluyen aquellos gastos que proceden de la contratación de servicios, alquiler de instalaciones, etc. En este caso, nos centraremos principalmente en la identificación de algunos servicios básicos. Sin embargo, dada la dificultad de calcular otros costes, tales como un alquiler o servicio de

electricidad/agua, se incluye un sobrecoste correspondiente al 10% del total acumulado, razón por la cual se considerará en un apartado posterior.

Una parte importante del coste de este apartado también vendrá dado por los dispositivos identificados en hardware, ya sea debido a su montaje o bien para habilitar su manipulación (en el caso del dron, curso de aprendizaje y habilitación, más un seguro de responsabilidad civil).

Descripción	Coste estimado (€)
Bloque 4. Costes indirectos	
600 Mbps de fibra y llamadas ilimitadas (Vodafone) Servicio de Internet	32 x 8 = 256
Curso de piloto avanzado de drones Obtención de permiso para utilización de dron	490
Seguro de dron Seguro necesario para la utilización del dron	250
Total Bloque 4. Costes indirectos	996

Tabla 9. Servicios identificados como costes indirectos y sus respectivos costes estimados.

Para calcular el coste estimado del servicio de Internet se ha acudido a la propia web del ofertante[21]. En el caso del curso de piloto, se ha recurrido a la búsqueda de un título homologado por AESA (Agencia Estatal de Seguridad Aérea) publicado por One Air[22]. Por último, para estimar el coste del seguro se ha utilizado el máximo precio observado en comparadoras de seguros, indicando las características del dron antes propuesto.

1.14.5 Coste total

Una vez justificado el coste estimado del proyecto en cada uno de sus apartados, podemos acumular dicho coste para aportar una cifra final, en la cual se debe considerar un sobrecoste sobre el total acumulado, con el fin de sufragar otros gastos de difícil estimación.

[21] https://www.vodafone.es/
[22] https://www.oneair.es/

Categoría	Coste estimado (€)
Bloque 5. Coste acumulado	
Coste hardware	35.497,44
Coste software	4.493,52
Coste de recursos humanos	24.578,76
Costes indirectos	996
Total Bloque 5. Coste acumulado parcial	65.565,72
Sobrecoste de 10%	6.556,57
Total Bloque 5. Coste acumulado final	72.122,29

Tabla 10. Recopilación de costes de los apartados previos. Cálculo de sobrecoste.

El coste final estimado se eleva a la cifra de 72.122,29 euros (€), aunque existen ciertos módulos, como el hardware de captura, que pueden omitirse para disminuir esta cifra.

1.14.6 Amortización

Para completar este apartado se debe calcular la amortización parcial de bienes y servicios citados para el periodo de ejecución definido en la planificación temporal.

Esta amortización se aplica especialmente sobre el grupo de bienes denominados “Equipos electrónicos e informáticos. Sistemas y programas”, tal y como lo define la Agencia Tributaria. Dentro de este grupo se incluye tanto el hardware como el software especificado. En el caso del hardware, parece evidente que tras un intervalo de tiempo este pueda disminuir su valor, e incluso quedar obsoleto, como es el caso de una tarjeta gráfica. La situación del software no es necesariamente la misma, dado que algunas de las licencias adquiridas implican el acceso a actualizaciones, siempre y cuando el software sea único y no ofrezca nuevas versiones cada cierto tiempo (a las que no se tendría acceso).

En cualquier caso, en aras de la simplicidad se ha decidido considerar el bloque de hardware y software al completo. La amortización a aplicar será una amortización lineal, de tal forma que disponemos de una base amortizable dependiente del valor

inicial y el valor residual. También se considera un coeficiente de amortización anual en lugar de su alternativa, que sería un período máximo de años (vida útil).

El valor del coeficiente anual de amortización o el máximo número de años de amortización se pueden obtener de la Agencia Tributaria[23]. De esta manera, se define que el máximo número de años de amortización del hardware es de 6 años, mientras que para el software serían 10 años. Como alternativa, se plantea un coeficiente lineal máximo de 20% y 33% respectivamente.

Por último, cabe destacar que la base amortizable se calcula como sigue:

$$Base\ amortizable = Valor\ Inicial - Valor\ Residual \tag{1.4}$$

Nótese como el valor inicial será el coste estimado antes propuesto, pero el valor residual es desconocido; sólo conocemos la vida útil que la Agencia Tributaria propone para cada tipo de recurso. De esta manera, se ha estimado oportuno seleccionar un valor residual del 30% para el hardware (6 años de vida útil), y un 10% para el software (10 años de vida útil).

Con estos datos, podemos calcular la amortización anual como sigue:

$$Amortización\ anual = \frac{Base\ AMT}{Vida\ útil} = Base\ AMT * Coeficiente\ AMT \tag{1.5}$$

A partir de estas ecuaciones se puede calcular la amortización lineal para recursos hardware y software:

Categoría	Valor inicial (€)	Valor residual (€)	Vida útil (años)	Amort. anual (€)
Amortización				
Hardware	35.497,44	10.649,23	6	4.969,64
Software	4.493,52	449,352	10	1.334,52
			Total amortizado (anual)	6.304,16

Tabla 11. Amortización anual de bienes hardware y software.

Dado que el proyecto no se extiende durante un año, sino durante 8 meses, la amortización durante este período de tiempo sería de 6.304,16 euros * 2 / 3 = **4.202,77 euros** (€).

[23] Tabla de coeficientes de amortización lineal a partir del 1 de enero de 2015. Fuente: Agencia Tributaria.

1.15 Normas y referencias

A continuación, se presentan las referencias de cualquier tipo que han sido de aplicación en la elaboración del trabajo y/o en la puesta en práctica del mismo.

1.15.1 Métodos, herramientas, modelos, métricas y prototipos

En este apartado se deben contemplar todos aquellos métodos, prototipos, métricas, programas, modelos y herramientas que se han empleado para obtener estimaciones anteriores: riesgos, costes, tiempos, etc. Por tanto, se crea un subapartado para todas aquellas estimaciones que deban completarse.

1.15.1.1 Presupuesto

La elaboración del presupuesto se ha basado únicamente en observaciones de las fuentes antes referenciadas. Para el cálculo de la amortización durante el tiempo de desarrollo del proyecto se han utilizado los períodos de años y coeficientes aportados por la Agencia Tributaria.

1.15.1.2 Análisis de riesgos

Para llevar a cabo el análisis de riesgos se ha empleado una matriz riesgo, de tal manera que para cada riesgo se ha valorado su impacto y probabilidad. De este modo, la Tabla 9 ha sido la principal referencia para asignar el impacto y el nivel de riesgo (código de color).

Impacto en el proyecto				Probabilidad				
	Planificación temporal	**Calidad de producto**	**Reputación**	**Muy baja** (< 0.1%)	**Baja** (0.1%-25%)	**Media** (25%-50%)	**Alta** (50%-80%)	**Muy alta** (> 80%)
Muy alto (5)	> 30	No se alcanzan los requisitos del sistema.	Repercusión internacional. Impacto irreparable en *stakeholders.*					
Alto (4)	14-30	Efecto sustancial en los objetivos de rendimiento.	Repercusión nacional. Impacto sustancial en *stakeholders.*					
Medio (3)	7-14	No mantiene todos los requisitos.	Repercusión regional. Impacto moderado en *stakeholders.*					
Bajo (2)	4-7	Caída de rendimiento menor. Mantiene requisitos.	Repercusión a nivel local. Impacto menor en *stakeholders.*					
Muy bajo (1)	< 4	Impacto muy ligero sobre el rendimiento. Se mantienen requisitos.	Poca repercusión. Impacto muy pequeño en *stakeholders.*					

Riesgo alto
Riesgo medio
Riesgo bajo

Tabla 12. Matriz riesgo junto a algunos indicadores de impacto que permiten clasificar los puntos de incertidumbre del proyecto.

1.15.1.3 Estimación de tamaño y esfuerzo

Esta estimación se ha obtenido mediante el modelo COCOMO, el cual ha sido descrito previamente. Tan sólo se han omitido las constantes correspondientes a los modificadores (Tabla 13) así como al tipo de proyecto (Tabla 14). A partir de ambas tablas y las decisiones justificadas en el apartado Estimación del tamaño y esfuerzo, se podrían rehacer los cálculos para comprobar su corrección.

Modificadores de modelo COCOMO						
Atributos	**Valor**					
	Muy bajo	Bajo	Nominal	Alto	Muy alto	Extr. alto
Fiabilidad	0,75	0,88	1,00	1,15	1,40	-
Tamaño de Base de datos	-	0,94	1,00	1,08	1,16	-
Complejidad	0,70	0,85	1,00	1,15	1,30	1,65
Restricciones de tiempo de ejecución	-	-	1,00	1,11	1,30	1,66
Restricciones de memoria virtual	-	-	1,00	1,06	1,21	1,56
Volatilidad de la máquina virtual	-	0,87	1,00	1,15	1,30	-
Tiempo de respuesta	-	0,87	1,00	1,07	1,15	-
Capacidad de análisis	1,46	1,19	1,00	0,86	0,71	-
Experiencia en la aplicación	1,29	1,13	1,00	0,91	0,82	-
Calidad de los programadores	1,42	1,17	1,00	0,86	0,70	-
Experiencia en la máquina virtual	1,21	1,10	1,00	0,90	-	-

Experiencia en el lenguaje	1,14	1,07	1,00	0,95	-	-
Técnicas actualizadas de programación	1,24	1,10	1,00	0,91	0,82	-
Utilización de herramientas de software	1,24	1,10	1,00	0,91	0,83	-
Restricciones de tiempo de desarrollo	1,22	1,08	1,00	1,04	1,10	-

Tabla 13. Posibles valores de los quince modificadores del modelo de estimación de costes COCOMO.

Constantes asociadas a tipos de proyectos				
Tipo de proyecto	***a***	***b***	***c***	***d***
Orgánico	3,2	1,05	2,5	0,38
Semilibre	3	1,12	2,5	0,35
Empotrado	2,8	1,2	2,5	0,32

Tabla 14. Constantes dependientes del tipo de proyecto que se aplican en las fórmulas de cálculo de esfuerzo y tiempo de desarrollo en el modelo COCOMO.

1.15.2 Mecanismos de control de calidad

El aseguramiento de la calidad en la redacción del trabajo implica la verificación de la completitud (falta de omisiones), la integridad de la documentación, así como una redacción clara, concisa y entendible por todos los participantes e interesados en dicho trabajo.

Al estar el trabajo desarrollado por una sola persona y verificado por un/a tutor/a, no es necesario llevar un documento de control de edición y revisión de la documentación. De esta forma, se han utilizado mecanismos básicos para la verificación de la integridad y completitud, que incluyen un control de versiones a nivel de documento y un control sencillo de la trazabilidad de especificaciones y requisitos.

2 DESARROLLO

La descripción del desarrollo del trabajo se divide en iteraciones, tantas como se indican en la Planificación temporal, debido a que la metodología de desarrollo seleccionada es una metodología ágil. En cada iteración se describe la solución a un problema más específico, de tal manera que al final de cada una de estas etapas podemos comprobar el estado de la solución e identificar nuevas necesidades. Además del desarrollo, en cada iteración se puede incluir cualquier otro tipo de documentación: historias de usuario, diagramas, *storyboards*, juegos de pruebas, etc. Este trabajo no plantea requisitos importantes desde el punto de vista del usuario, al tratarse de un trabajo experimental; por tanto, nos centraremos en aquellos diagramas que permitan comprender la solución implementada, y puntualmente, se puede recurrir a historias de usuario, cuando la iteración que nos ocupe lo justifique.

A lo largo de este desarrollo se mostrarán diversos resultados cuantitativos, dado que algunos de ellos justifican la ejecución de las siguientes iteraciones. Por tanto, el apartado Experimentación, resultados y discusión queda supeditado al desarrollo de pruebas adicionales que muestren las capacidades de la solución final.

2.1 Planificación, revisión bibliográfica y pruebas

La primera etapa del desarrollo tiene como finalidad realizar una revisión bibliográfica, cuyo principal resultado se plasma en Antecedentes y estado del arte. A partir de esta revisión se extraen cuáles son las técnicas de simulación más comunes, las cuales se analizan para decidir cuál es el punto de partida de la primera iteración.

También en el estado del arte se esboza la solución finalmente considerada, la cual se sustenta principalmente sobre el lanzamiento de rayos, más que en la generación de texturas, aunque es cierto que este último enfoque es el más común. La principal razón por la que se escoge este planteamiento es la precisión de la solución, de tal manera que se desea simular de la manera más fiel posible las físicas asociadas a un dispositivo LiDAR. Además, la inclusión de errores comunes y otros comportamientos, como los múltiples retornos, pueden ser más fáciles de simular cuando se conocen todos los parámetros físicos de un pulso láser. Esto no implica que estos resultados no puedan obtenerse mediante la utilización de texturas.

Las principales pruebas que aquí se desarrollan se orientan en torno a la generación de un *cube map*, como el que se muestra en la Ilustración 3, para analizar las posibilidades de todos los enfoques planteados.

Por último, se utiliza este apartado para plantear los principales requisitos de la aplicación en forma de historias de usuario, una vez se han analizado los principales objetivos que debe contemplarse en la solución final. Se considera que esta forma de expresar los requisitos es más idónea que un diagrama de caso de uso, dado que la aplicación no presenta flujos complejos de acciones de usuario, sino más bien necesidades puntuales.

2.1.1 Historias de usuario

El objetivo de una historia de usuario es describir, de manera informal, las funcionalidades que debe tener un sistema. Por tanto, es una descripción muy simple de requisitos, pudiéndose combinar esta forma de documentación con diagramas de casos de uso. Una historia de usuario contiene tres puntos clave (Ilustración 17): el tipo de usuario, qué desea este, y por qué (Visual Paradigm, 2020). Al tratarse de una descripción tan básica, cualquier formato puede darse por válido, siempre y cuando indiquen de manera clara el requisito que se plantea. En nuestro caso, presentaremos estas historias mediante varias tablas, una por actor identificado.

Ilustración 17. Representación del concepto de historias de usuario.

Historias de usuario: Como **usuario/investigador externo**…	
Quiero	**Para**
Visualizar la escena preparada para la captura.	Seleccionar la escena que mejor se adapte a la finalidad de mi trabajo.
Seleccionar la escena preparada para la captura.	Obtener una nube de puntos de un escenario adaptado a la finalidad de mi trabajo.
Desplazarme por la escena.	Observar la nube de puntos resultante del proceso de escaneo.
Manipular un modelo LiDAR parametrizado.	Adaptar los parámetros de escaneo a una situación particular.
Disponer de una interfaz gráfica de usuario	Modificar fácilmente todas aquellas variables presentes en la aplicación.
Manipular la visualización de modelos en la escena	Observar los resultados obtenidos del escaneo 3D.
Modificar los parámetros de todos los objetos de la escena.	Obtener una escena donde todos los objetos se vinculan con un determinado material.
Indicar una ruta aérea de manera manual	Simular la ruta de un vuelo aéreo realizado por un vehículo no tripulado.
Exportar la nube de puntos obtenida.	Introducir la nube de puntos en cualquier otro proceso externo, por ejemplo, relacionado con la inteligencia artificial.

Tabla 15. Historias de usuario donde el actor es un usuario ajeno al desarrollo.

Historias de usuario: Como **desarrollador**…	
Quiero	**Para**
Generar capturas de pantalla con un tamaño flexible.	Documentar el trabajo realizado.
Visualizar la estructura de datos que organiza la escena.	Comprobar el correcto funcionamiento de construcción de la estructura de datos.
Visualizar el lanzamiento de rayos desde el dispositivo.	Comprobar que la generación de rayos se ajusta al tipo de LiDAR y a los parámetros indicados.
Disponer de una interfaz gráfica de usuario	Habilitar o deshabilitar la visualización de cualquier modelo, ya sea una estructura de datos o una nube de puntos.
Disponer de un escenario animado	Medir rendimiento de algoritmos desarrollados.

Tabla 16. Historias de usuario donde el actor es un desarrollador.

Una historia de usuario es una herramienta especialmente útil en metodologías ágiles, dado que estas funcionalidades suelen suponer requisitos de rápido desarrollo. Tanto en la Tabla 15 como en la Tabla 16 se describen funcionalidades a un nivel más alto, considerando que esta es una fase previa al desarrollo. Por tanto, los objetivos, requisitos e historias de usuario descritas en este documento deberían definir claramente qué se espera de la solución final. A medida que avancemos, estas historias de usuario podrán indicar requisitos más específicos.

2.2 Primera Iteración

La primera iteración tiene como objetivo la construcción de una aplicación que incorpore un amplio número de conceptos de la informática gráfica que son necesarios para el proyecto. Por tanto, en esta iteración sólo se describe la implementación de aquellas entidades relacionadas con la aplicación gráfica. Este apartado se dividirá en tantas secciones como categorías de entidades se identifiquen, comenzando por el núcleo de la aplicación.

2.2.1 Entidades de la aplicación gráfica

2.2.1.1 Núcleo de la aplicación

Partiendo de las librerías descritas en Tecnologías utilizadas, incluidas en el proyecto mediantes los pasos que se muestran en Instalación y configuración del sistema, el desarrollo comienza mediante una entidad ***Window***, que representa la ventana donde se *renderiza* una escena.

Nótese como sólo es posible disponer de una ventana a lo largo de la ejecución, razón por la cual esta entidad adopta el patrón Singleton (Eric Freeman, Elisabeth Freeman, Bert Bates 2013), por el cual sólo se dispone de un objeto de tipo *Window*. Además, esta instancia es globalmente accesible, lo que facilita su uso. Otra ventaja de la utilización de este patrón es la inicialización perezosa, dado que la instancia única no se crea hasta que esta se solicite. En nuestro caso, esta ventaja no se aplica, dado que una ventana es necesaria desde el inicio de la aplicación, y por tanto, no estaríamos disminuyendo la carga de trabajo en esta etapa.

La primera funcionalidad que recae sobre una ventana es la inicialización del contexto de GLFW, donde se indica, entre otros valores, cuál es la versión de OpenGL

seleccionada o qué funciones (estáticas) reciben ciertos eventos vinculados a la ventana. Igualmente, desde este punto también se solicita la inicialización de OpenGL o de la interfaz gráfica (Dear ImGui), aunque ambas funciones no recaen sobre la ventana, sino sobre las entidades *Renderer* y *GUI*, respectivamente. Por último, sobre la ventana se ejecutará el bucle principal de la aplicación, en el cual se obtienen nuevos eventos y se organiza el *rendering* de la aplicación.

Otra entidad importante es ***InputManager***, dado que en ella podemos encontrar todos los métodos donde se notifican los eventos de interés que se producen sobre la ventana: modificación de tamaño, entrada de teclado, entrada de ratón (*scroll*, botones y cursor) o la actualización del contenido. En el caso del último evento, cabe destacar que *Window* lo utiliza en cada iteración de su bucle, pero es posible encontrar llamadas adicionales al mismo cuando es necesario redibujar la ventana (por ejemplo, después de recibir la oclusión de otra ventana).

Antes de comenzar a describir cómo se procesan los eventos recibidos es necesario destacar que la interacción con el usuario emula la interacción implementada en el *framework* Unity3D, donde el movimiento libre se produce siempre que el botón derecho se encuentre presionado, y además, la velocidad de movimiento crece a medida que avanza el tiempo en el que se mantiene dicha presión.

La primera entrada de interés es la pulsación de una tecla, donde la respuesta desarrollada es la notificación del evento a la entidad correspondiente. Por ejemplo, presionar la tecla ‘X’ produce una rotación *orbit* en torno a un punto, por lo que se debe notificar a la cámara principal de la aplicación. Más allá de la distinción de la tecla pulsada mediante algunas valores constantes definidos, es necesario indicar que hay ciertos eventos en los que se produce una actualización de los acumuladores que permiten incrementar la velocidad de algunos movimientos; en este caso, *dolly* y *truck* (por desarrollar en la cámara).

Al disponer de múltiples tipos de movimientos, es posible darle nombre a todos ellos e inicializar múltiples estructuras de datos de tipo vector para almacenar los valores vinculados a cada movimiento. Así, se dispone tanto de un conjunto de acumuladores (de valor inicial cero) como de un conjunto de constantes que representan la velocidad inicial de cada movimiento, sea esta diferente para todos ellos. De esta manera, es posible obtener la velocidad de un movimiento en un instante de tiempo concreto mediante el siguiente cálculo:

$$speed = direction * movementSpeed[type] + accumulator[type] * movementSpeed[type] * movementMultiplier \quad (2.1)$$

Sea $direction$ un valor 1 o -1, que aplica el cambio en una dirección u otra, y $movementMultiplier$ un factor multiplicador de velocidad que actúa sobre las acumulaciones.

El valor acumulado de un movimiento vuelve a ser cero cuando el usuario libera la tecla que producía dicho movimiento, razón por la cual es necesario recibir los eventos de liberación de teclas. En cuanto a los movimientos de ratón, se describen hasta tres tipos de eventos:

- **Pulsación de botón**. Se registra la pulsación de botón izquierdo y derecho de ratón, aunque este último es el que más nos interesa por iniciar el movimiento libre del usuario en la escena (*pan* y *tilt*). En ambas situaciones se inicializa la posición del cursor a un valor no válido para comenzar a rastrear los movimientos de dicho cursor.

- **Movimiento de ratón (cursor)**. Permite llevar a cabo los movimientos *pan* y *tilt*, donde la cámara no modifica su posición pero sí lo hace el objetivo de la misma (*tilt* se identifica con movimientos verticales del cursor, y *pan* con movimientos horizontales). De esta manera, la velocidad de movimiento en un instante concreto vendrá dada por el siguiente cálculo:

$$speed_{pan} = -movementSpeed[type] * (position_x - lastPosition_x)$$
$$speed_{tilt} = -movementSpeed[type] * (position_y - lastPosition_y) \quad (2.2)$$

  Nótese como es necesario modificar el signo de la velocidad de movimiento, debido a que una velocidad positiva realiza una rotación en sentido anti-horario (*counterclockwise*). Sin embargo, cuando el usuario desplaza su cursor hacia la derecha busca justamente la rotación opuesta.

- **Evento de *scroll* (rueda)**, cuyo único fin es habilitar el movimiento de *zoom* sobre la escena. El valor de desplazamiento en la Y, dado por GLFW, contiene la dirección, positiva o negativa, del mismo. Por tanto, sólo hay que aplicarle a dicho valor la velocidad de movimiento inicial para el movimiento de *zoom*.

Un último detalle que debe destacarse es la relación que esta entidad mantiene con la interfaz gráfica. Por la naturaleza de esta última, la aplicación debería redibujarse constantemente, sin importar que existan cambios o no. Esto no ocurría antes de incluir la interfaz, dado que conocemos de antemano que las únicas situaciones que desembocan en un redibujado son algunos de los eventos de usuario que recibimos en la entidad *InputManager*. Por tanto, para evitar redibujar constantemente, se realiza un seguimiento de posibles cambios en la interfaz, de tal manera que para cada evento de usuario se agrega un número preestablecido de redibujados. Así, sólo será necesario *renderizar* de nuevo cuando haya solicitudes en la cola. Por ejemplo, presionar un botón del ratón puede implicar la pulsación de un botón de la interfaz, y por esta razón se asignan dos solicitudes de *rendering*, para asegurar que las animaciones de la interfaz puedan visualizarse (la librería Dear ImGui no nos permite conocer exactamente si el usuario ha interactuado con un elemento de la interfaz). Así, el número de renderizados asignados a cada movimiento se muestra en la Tabla 17.

Evento	Número de *renderings*
Pulsación de tecla	1
Liberación de tecla	1
Presión de botón de ratón	2
Movimiento de cursor	1
Scroll (rueda de ratón)	1
Modificación del tamaño de la ventana	2

Tabla 17. Número de llamadas de rendering por evento.

Uno de los eventos integrados en *InputManager* que ha pasado desapercibido hasta ahora es el evento de actualizar (*refresh*) la ventana, donde no sólo se *renderiza* la escena sino también la interfaz, dado que esta se integra en el *pipeline* de *rendering* como un modelo más. Por tanto, el dibujado de la interfaz recae sobre una entidad que ha recibido el nombre de **GUI** (*Graphical User Interface*). Tanto de esta entidad como de la anterior sólo debe existir una instancia, y como tal, se implementan bajo el patrón *Singleton* antes citado.

En iteraciones posteriores se completa la interfaz, pero en esta primera iteración únicamente se define una base donde sólo es posible inicializar Dear ImGui con una versión de OpenGL, así como renderizar una ventana integrada en la librería a modo de demostración, la cual nos servirá posteriormente para desarrollar nuestra interfaz (en lugar de documentación, la librería muestra su funcionamiento mediante este ejemplo).

Para finalizar con el núcleo de la aplicación gráfica, es necesario describir la entidad ***Renderer***, la cual, como su nombre indica, se encarga de *renderizar* la escena (la interfaz queda excluida), aunque su principal funcionalidad es la preparación del entorno OpenGL (habilita el uso de nubes de puntos de diferente tamaño o el *antialiasing*, indica el índice que se emplea para distinguir primitivas en un vector de índices, etc). También se encarga de gestionar las capturas de pantalla, dado que el *framebuffer* que se emplea pertenece a esta instancia, así como los eventos de modificación de tamaño de la ventana (actualiza *viewport*, *aspect ratio* de su *framebuffer* o indica a la escena que haga esto mismo con sus cámaras). Su comportamiento en el *rendering* se limita a notificar a la escena activa que es necesario el *renderizado* de sus modelos.

Todas las entidades aquí citadas, y su relación, se muestran en el diagrama de clases de la Ilustración 18.

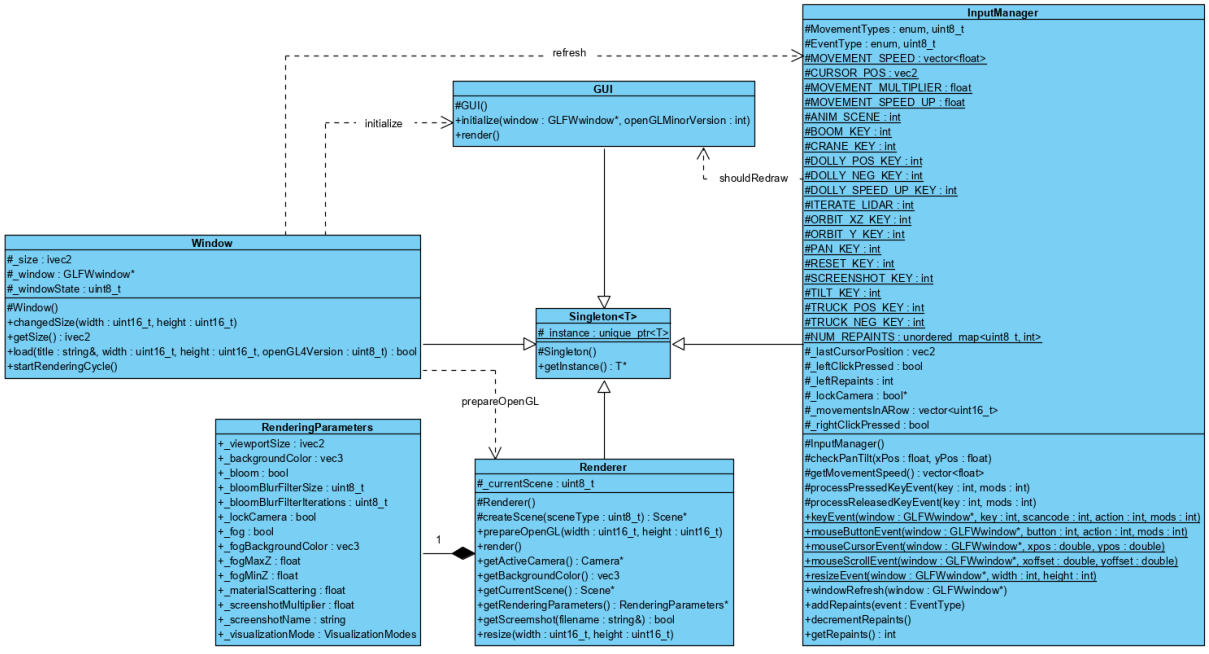


Ilustración 18. Diagrama de clases del núcleo de la aplicación.

Una vez definida la base de la aplicación, podemos detallar el resto de entidades que sí tienen relación con la construcción de la vista de una escena, y no únicamente con su representación al usuario.

2.2.1.2 Texturas y materiales

El color y la interacción de los modelos con el entorno depende en primer lugar de texturas y materiales, sea esta última entidad un contenedor de la primera. Es decir, la aplicación gráfica consta de tantas texturas como se deseen, y lo mismo ocurre para los materiales, los cuales a su vez se enlazan con algunas texturas que desempeñan una determinada función.

Para una textura se plantean hasta cinco constructores:

- Constructor de una textura que se inicializa a partir de un fichero PNG (para lo cual se utiliza la librería LodePNG).
- Constructor de una textura mediante un vector de cuatro componentes, cuyo objetivo es disponer de un color uniforme. Una alternativa para el usuario sería crear una imagen de cualquier tamaño con el color deseado, pero es más eficiente indicar el color y que la aplicación cree una textura de tamaño 1x1.
- Constructor de una textura a partir de un conjunto de números reales. El origen de este constructor se encuentra en la librería FastNoiseSIMD, la cual almacena los valores de ruido en un vector de números reales.
- Constructor de una textura a partir de una instancia de la clase *Image*. Cuando trabajamos con un *framebuffer object* (aún por desarrollar), se extrae la imagen *renderizada* y se encapsula en una instancia de tipo *Image*, la cual nos permitirá, entre otras operaciones, almacenar la imagen en el sistema a través de un hilo independiente respecto del flujo de la aplicación principal. Por tanto, no es más que otra manera de encapsular un vector de valores (en este caso, de *unsigned char*).
- Constructor de una textura a partir de un conjunto de instancias de la clase *Image*, permitiendo así definir un *cube map*.

El objetivo al construir una textura es emplearla en un *shader program*, y por tanto, es necesario proveer de los mecanismos de enlazado, encontrándose al menos cuatro:

- Aplicación de textura indicando *shader program* y el tipo de textura (*AmbientDiffuse* (*Kad*), *specular* (*Ks*), semitransparente, *bump map*, *displacement map*, *bloom* o *cube map*). Se dispone de una tabla hash que relaciona cada tipo de textura con un nombre de variable en el *shader*. Por tanto, esta versión es más cómoda pero también es más estricta (es necesario comprobar el nombre de cada tipo de textura para comenzar a implementar un *shader*).
- Aplicación de textura indicando *shader program*, un identificador y el nombre de la variable que ocupa. Es una versión mucho más flexible que suele emplearse cuando la textura no se integra dentro de un material. El identificador implica cierto riesgo en tanto que no pueden existir dos texturas asignadas a un mismo *slot*.
- Aplicación de textura indicando *shader program*, identificador, nombre de variable, uso de la textura y formato de la misma. Este método extiende la versión anterior para poder integrar la textura en un *compute shader*, dado que en este tipo de *shaders* es posible modificar una textura (aunque se ha especificado siempre como *READ_ONLY*). También existe un listado de tipos de formatos de imágenes que habrá que considerar (Rodríguez, 2013).
- Aplicación de *cube map* indicando el *shader program*. En este caso se asume que el nombre de la variable viene dado por la *tabla hash* antes citada.

Por otra parte, un **material** no es más que un conjunto de texturas que desempeñan algunos de los papeles que se han citado (*Kad*, *Ks*, etc). No es necesario disponer de texturas de todos los tipos, y de hecho, el material aplica *uniforms* y escoge subrutinas en base a las texturas enlazadas. Por ejemplo, un material podrá disponer de un *bump map*, en cuyo caso sólo se modifican las normales, o de un *bump* y un *displacement map* para la técnica de *displacement mapping*. En ambos casos requieren subrutinas diferentes. De forma análoga a la textura, un material también debe aplicarse sobre un *shader program*, aunque en este método se delegan todas

las llamadas necesarias para renderizar un modelo con el material seleccionado. El principal problema asociado a un método tan estático como este es que no siempre se aplica un material para renderizar el modelo de una única manera, lo cual puede implicar que en futuras iteraciones sea necesario incluir otras versiones alternativas respecto de esta base.

Junto al concepto de textura y material encontramos dos bases de datos que se implementan bajo el patrón *Singleton*, ***TextureList*** y ***MaterialList***, de tal manera que se posibilita su acceso global: cualquier modelo, escena o algoritmo puede necesitar una textura o un material. Más allá de dar acceso a estos recursos, la principal funcionalidad de ambas bases de datos es reducir la carga de trabajo de la aplicación en el inicio; las texturas sólo se cargan cuando estas se solicitan, y esto es de vital importancia cuando las texturas se forman a partir de ficheros en el sistema, dado que el tiempo de acceso y carga de las mismas puede ser relativamente elevado.

Nótese como el concepto de inicialización perezosa es incompatible con una base de datos donde se indican las características de las texturas y materiales mediante la creación de sus correspondientes instancias, dado que en nuestra aplicación su mera construcción implica también la carga de la textura. Por tanto, una solución válida sería mantener esta estructura e incluir una inicialización de textura cuando esta se solicite, mediante un método *load*, con la desventaja de que en este punto se desconoce si la textura se debía construir a partir de un fichero, un color, una imagen, etc, al margen de que sea posible almacenar en una variable qué tipo de inicialización debe darse. Por tanto, la solución finalmente propuesta consiste en evitar la creación de texturas y materiales en el comienzo de la aplicación, introduciendo así dos tablas hash para las texturas (nombre de textura y color o nombre de textura y ruta en el sistema), así como una nueva estructura, ***MaterialSpecs***, donde se indican las propiedades de un material.

Para completar la revisión de entidades relacionadas con materiales y texturas en la aplicación, se muestra la Ilustración 19 y la Ilustración 20:

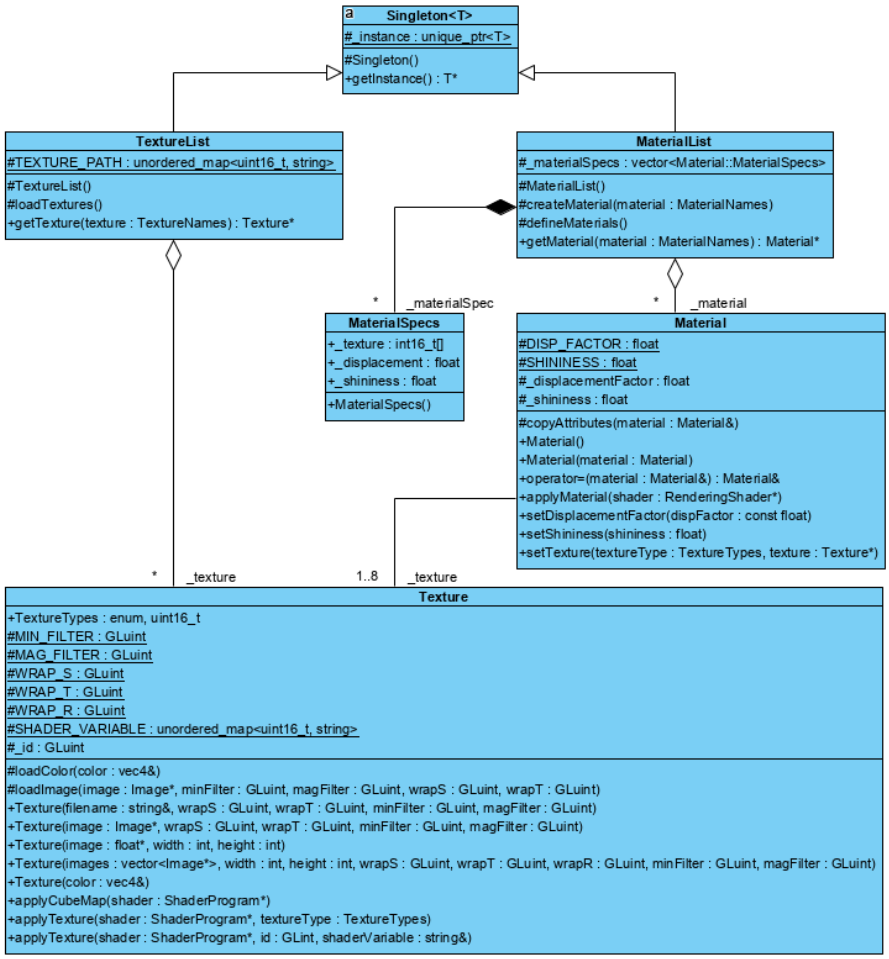


Ilustración 19. Diagrama de clases para entidades relacionadas con materiales y texturas.

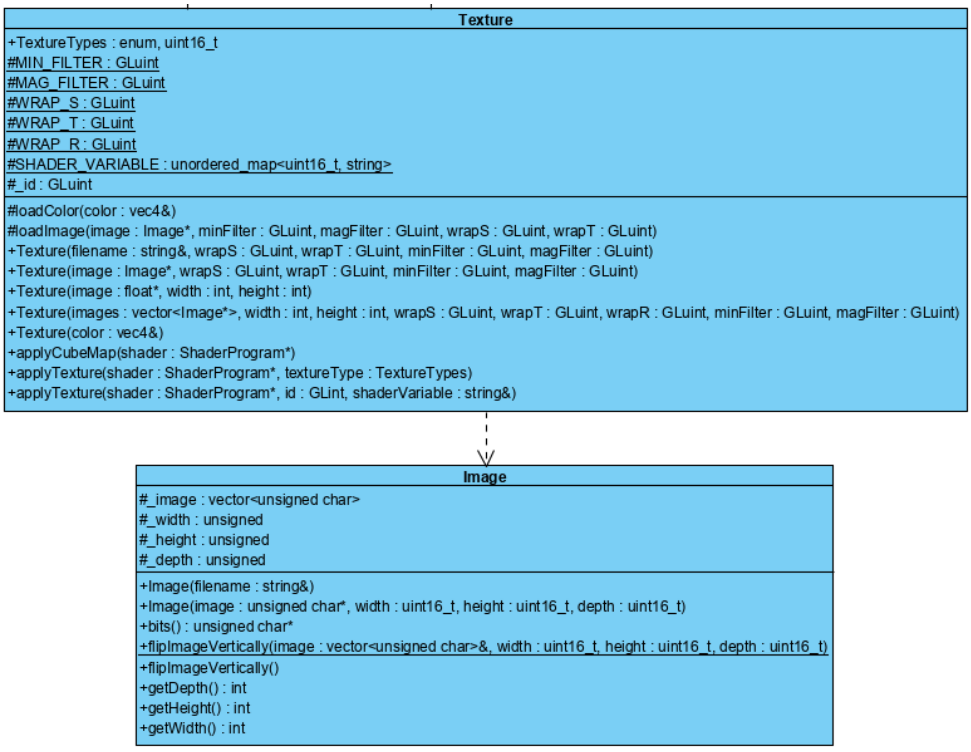


Ilustración 20. Diagrama de clases de la entidad Texture y la clase auxiliar Image.

2.2.1.3 Modelos de luces

Mediante las texturas y materiales que se han descrito anteriormente sólo se podría crear una escena de color uniforme, sin embargo, el objetivo de este trabajo es obtener un resultado realista con técnicas de *rendering* muy básicas. El siguiente paso consiste en la introducción de modelos de luces, hasta cuatro, donde la forma de interactuar con la escena será diferente para todos ellos.

Antes de comenzar a describir cada modelo es necesario reflexionar acerca de su implementación. No todos comparten unos mismos atributos; hay modelos en los que la posición de la fuente de luz es desconocida (muy lejana) o en los que no hay única dirección en la que se emite la luz. Una solución básica consistiría en desarrollar un sistema jerárquico, donde la raíz sería una entidad abstracta *LightSource*, que contendría todos aquellos atributos comunes. Después podríamos implementar modelos de luces más concretos, e incluso subclases que contengan varios modelos con atributos comunes. Evidentemente, este es un enfoque válido, aunque se trata de una solución muy básica y estática. Por ejemplo, modificar el modelo de luz para una instancia no sería sencillo, y entre dichas transiciones podríamos perder datos.

Un enfoque diferente viene dado por el patrón ***Strategy***, donde no se definen los modelos de luz como subclases de un modelo principal, sino como un conjunto de aplicadores que implementan un método donde se especifica qué subrutinas y variables deben aplicarse a un *shader program*. De esta manera, sólo existirá una entidad ***Light***, que incluye todos los atributos posibles, y que también mantiene un identificador con el modelo seleccionado por el usuario, y un vector de aplicadores. Se da la circunstancia de que dichos aplicadores no mantienen atributos, sólo implementan algunos métodos, por lo que el citado vector de aplicadores puede compartirse entre todas las luces, reduciendo así los recursos que emplea la aplicación.

Más allá de la estructura a seguir, los cuatro modelos de los que se hace distinción se describen a continuación:

2.2.1.3.1 Point light

Una luz puntual se define por una posición en un espacio 3D, y como todos los modelos que se han desarrollado, por dos vectores de tres componentes, *Id* e *Is*, que representan los colores que intervienen en la componente difusa y especular (Ilustración 21). Aquellas luces puntuales que mantienen su intensidad (I_L) constante

también se conocen como *omni lights* (Akenine-Möller 2018), y de momento, así será hasta detallar la atenuación, aunque no se modificará la intensidad como tal, sino la contribución a la escena, E_L.

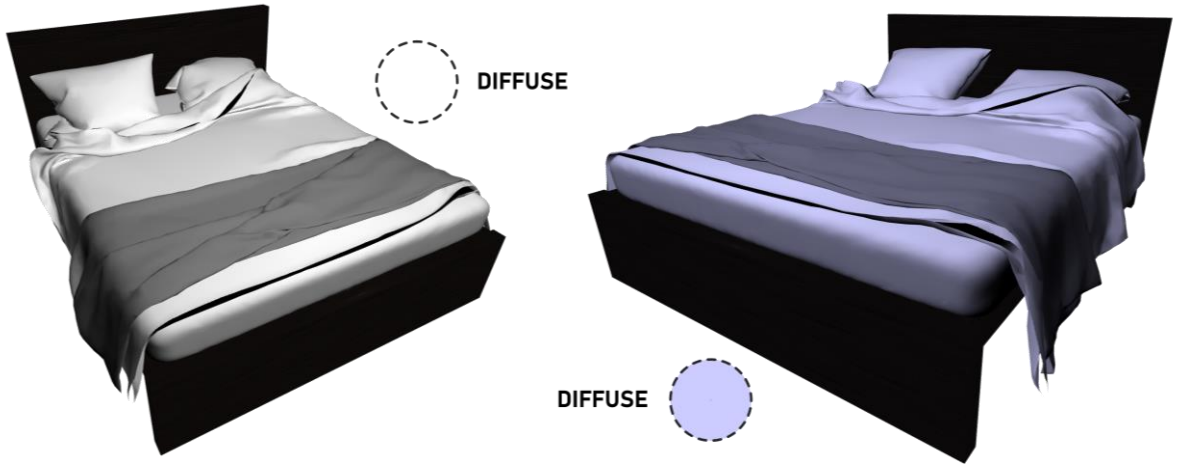


Ilustración 21. Un único modelo renderizado con dos luces puntuales de diferente intensidad en la componente difusa.

Sin un modelo de atenuación, la contribución de una luz puntual se calcula como se muestra a continuación, conociendo de antemano la posición ($p_{surface}$) y la normal de la superficie en dicha posición ($\hat{n}$), así como la posición que ocupa la luz (p_{light}):

$$dotLN = \hat{n} * \left(\widehat{p_{light} - p_{surface}}\right)$$

$$dotHN = \left(-\widehat{p_{surface}} + \left(\widehat{\widehat{p_{light}} - \widehat{p_{surface}}}\right)\right) * \hat{n}$$

$$diffuse = I_d * K_{ad} * \max\left(\frac{(dotLN + scattering)}{1 + scattering}, 0\right) \tag{2.3}$$

$$specular = I_s * K_s * \max(dotHN, 0)^{shininess}$$

$$contribution_{light} = diffuse + specular$$

El factor más evidente de esta formulación es $dotLN$, dado que buena parte de la contribución dependerá del ángulo entre la normal de la superficie y el vector $p_{light} - p_{surface}$. Nótese como la contribución máxima se obtiene cuando el ángulo es cero, y por tanto, $diffuse = I_d * K_{ad}$. Para todos aquellos puntos que forman un ángulo mayor que 90 se obtiene una componente difusa nula, siempre y cuando $scattering$ sea cero. Por tanto, la función de $scattering$ es emular una luz ambiente, aunque sin

constituir una instancia más, lo cual supone una gran ventaja en una aplicación donde la utilización de múltiples fuentes de luz en una escena se resuelve mediante *multipass rendering* (cada luz supone un *rendering* adicional).

En la formulación propuesta podríamos pensar que p_{light} y $p_{surface}$ se hallan en un espacio de coordenadas diferente (mundo y cámara, respectivamente), aunque al darse este cálculo en un *fragment shader*, lo más común es disponer de la posición $p_{surface}$ en el sistema de la cámara, y la posición de la luz se puede trasladar a este mismo sistema desde el exterior (en el instante en que se indica el valor de la variable *uniform*).

Por otro lado, la componente especular depende del punto de visión, por lo que a partir de lo ya expuesto podemos obtener $-p_{surface}$. El cálculo que aquí se desarrolla se corresponde con una versión descrita en (Akenine-Möller 2018), donde el vector *h* empleado se conoce como *halfway vector*, un vector que forma el mismo ángulo tanto con v, $(-p_{surface})$, como con l, $(p_{light} - p_{surface})$. Tanto en la Ilustración 22 como en la Ilustración 23 se muestran las diferencias entre una formulación tradicional y la propuesta. En el caso de estos modelos lo que se obtiene es una influencia mayor de la componente especular a lo largo de la superficie. Es decir, el ángulo entre h y n es menor que el ángulo entre los dos vectores que habitualmente se emplean, $reflect(-l, n)$ y $-p_{surface}$.

Ilustración 22. Modelo de Buddha renderizado utilizando 1) un cálculo tradicional de la componente especular, 2) el vector h descrito (halfway vector).

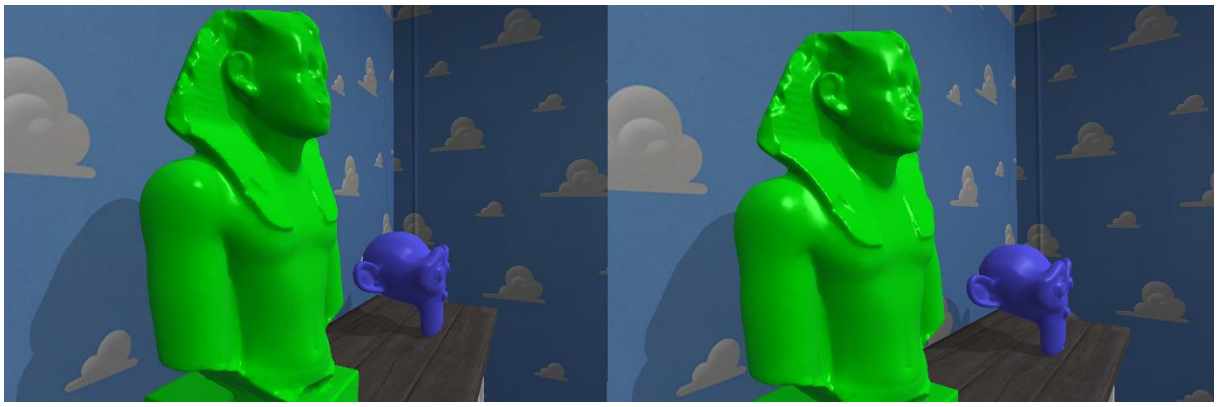


Ilustración 23. Dos modelos renderizados utilizando 1) un cálculo tradicional de la componente especular, 2) el vector h descrito (halfway vector).

En el caso de los planos lo que suele buscarse no es una mayor extensión, sino justamente lo contrario: un brillo mucho más estrecho y reducido.

2.2.1.3.2 Spot light

El modelo de luz *spot light* emite la luz de forma cónica, de tal manera que la máxima contribución se alcanza en el centro de dicho cono. A diferencia del modelo *omni* antes descrito, las fuentes de luz reales tienen variaciones de su intensidad, y en este caso dicha variación vendrá dada por el ángulo entre d, un vector dirección de la luz (define el centro del cono), y $-l$ ($p_{surface} - p_{light}$). Toda la formulación que antes se aplicaba a la luz puntual se mantiene vigente para este modelo de luz, con la diferencia de que ahora es necesario atenuar la contribución final.

$$contribution_{light} = spotAtt * (diffuse + specular) \quad (2.4)$$

El modelo de luz hasta ahora descrito produciría una emisión que se reduce paulatinamente a medida que el ángulo decrece, hasta llegar al límite definido por un ángulo α. Para evitar un descenso lineal que ilumine de manera muy reducida un área, suele elevarse el coeficiente de atenuación, $-l * d$, a un exponente s_{exp}.

Otra manera de plantear este problema es considerar dos ángulos: penumbra (θ_p) y umbra (θ_u). El primero indica dónde comienza la transición, mientras que el segundo indica dónde se alcanza el valor cero. De esta manera, se obtiene un área donde no se aplica el factor de atenuación y por tanto, recibe la máxima irradiancia. El factor de atenuación se puede definir como sigue:

$$f(x) = \begin{cases} 1, & \cos\theta_p \leq \cos\theta_s \\ \left(\dfrac{\cos\alpha - \cos\theta_u}{\cos\theta_p - \cos\theta_u}\right)^{s_{exp}}, & \cos\theta_u < \cos\theta_s < \cos\theta_p \\ 0, & \cos\theta_s \leq \cos\theta_p \end{cases} \quad (2.5)$$

Algunos resultados obtenidos con este modelo de luz se muestran en la Ilustración 24.

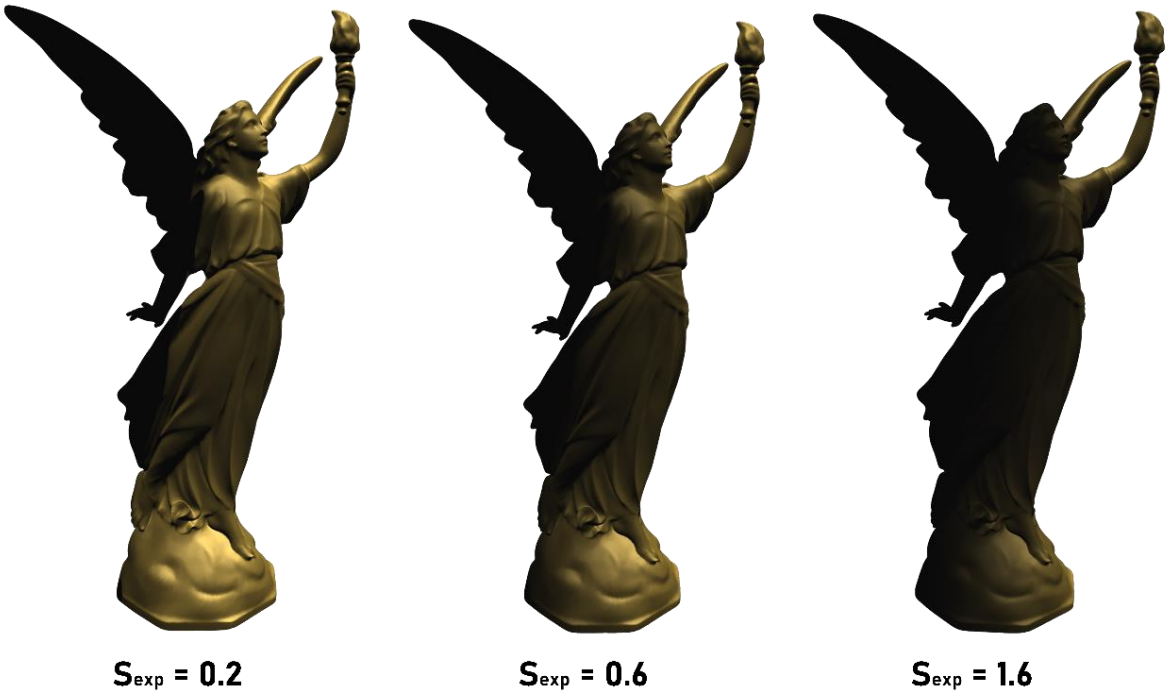


Ilustración 24. Comparativa de tres modelos spot light de diferente exponente s_{exp}.

2.2.1.3.3 Directional light

Un modelo de luz direccional se define principalmente por una dirección d, de tal manera que $l = p_{light} - p_{surface} = -d$. No se conoce su posición, que se considera desconocida o muy lejana (en el infinito), por lo que sólo se dispone de un único vector dirección para cualquier posición. La contribución de la luz dependerá únicamente del ángulo entre dicho vector y la normal de la superficie. Por tanto, toda la formulación expuesta en la luz puntual también se aplica aquí, con la diferencia de que p_{light} es desconocida.

Si aplicamos este tipo de iluminación al modelo empleado en la Ilustración 21 se obtiene la Ilustración 25, donde se pueden observar ciertas zonas de la superficie en las que la irradiancia recibida es mucho menor, al disponer de un vector normal que forma con el vector l constante $(-d)$ un ángulo mayor o cercano a los 90º. También es necesario considerar que la luz puntual de la Ilustración 21 se sitúa sobre

el modelo, razón por la cual gran parte de la superficie se encuentra iluminada. De situarse en un lateral, el resultado no diferiría en gran medida del que aquí se obtiene.

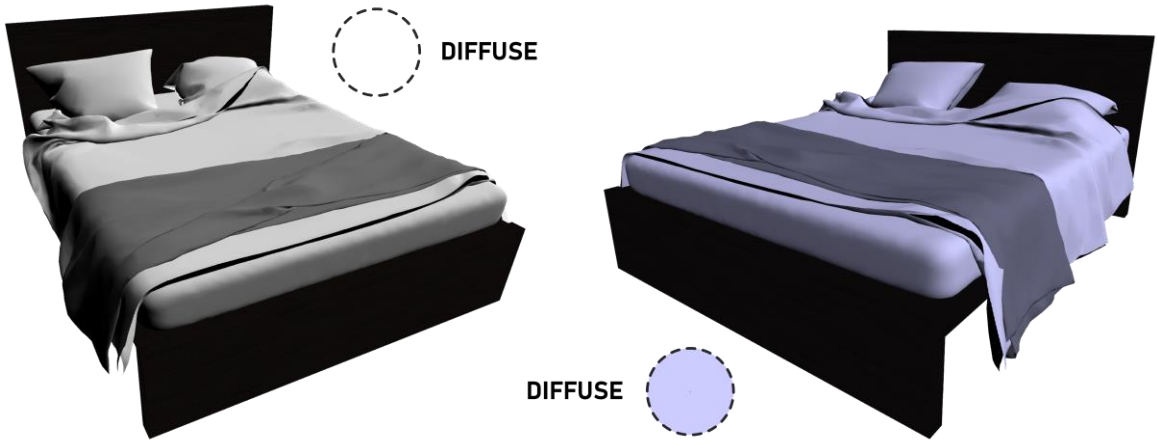


Ilustración 25. Un único modelo renderizado con dos luces direccionales de diferente intensidad en la componente difusa.

2.2.1.3.4 Rim light

En el ámbito de la fotografía se suele emplear el término de *rim light* para hacer mención a una técnica de iluminación que permite resaltar los contornos de las formas representadas. En las aplicaciones gráficas no se utiliza este modelo únicamente para iluminar el contorno y mantener el resto de la escena en sombra, sino como un modelo de iluminación más que permite resaltar las formas representadas respecto de un fondo.

Este efecto se podría obtener con algunos de los modelos antes descritos, aunque es cierto que su configuración puede llegar a ser más compleja, dado que es necesario elegir un modelo y seleccionar una posición (detrás de la escena) que afecte a los modelos de la misma y cree el efecto deseado. Sin embargo, es suficiente con utilizar la normal y el vector de visión en una posición para obtener un modelo directo. Esta versión no representa realmente la influencia de un modelo de luz, sino que más bien se trata de un efecto donde se resaltan los bordes de la superficie, pero en una aplicación gráfica puede ser suficiente y su cálculo es mucho más rápido (Ilustración 26). Por tanto, el modelo de *rim light* que aquí se plantea se expresa como sigue:

$$E_L = color_{rim\ light} * \left(1 - \max\left(p_{\widehat{surface}} * \hat{n}, 0\right)\right) \quad (2.6)$$

Los puntos que debemos iluminar son aquellos que presentan una mayor diferencia de ángulo entre n y $-p_{surface}$, por tanto, la contribución se corresponde a

uno menos el producto escalar de ambos vectores normalizados (el coseno de 90º es cero, mientras que en esta situación lo que se busca es un factor de uno).

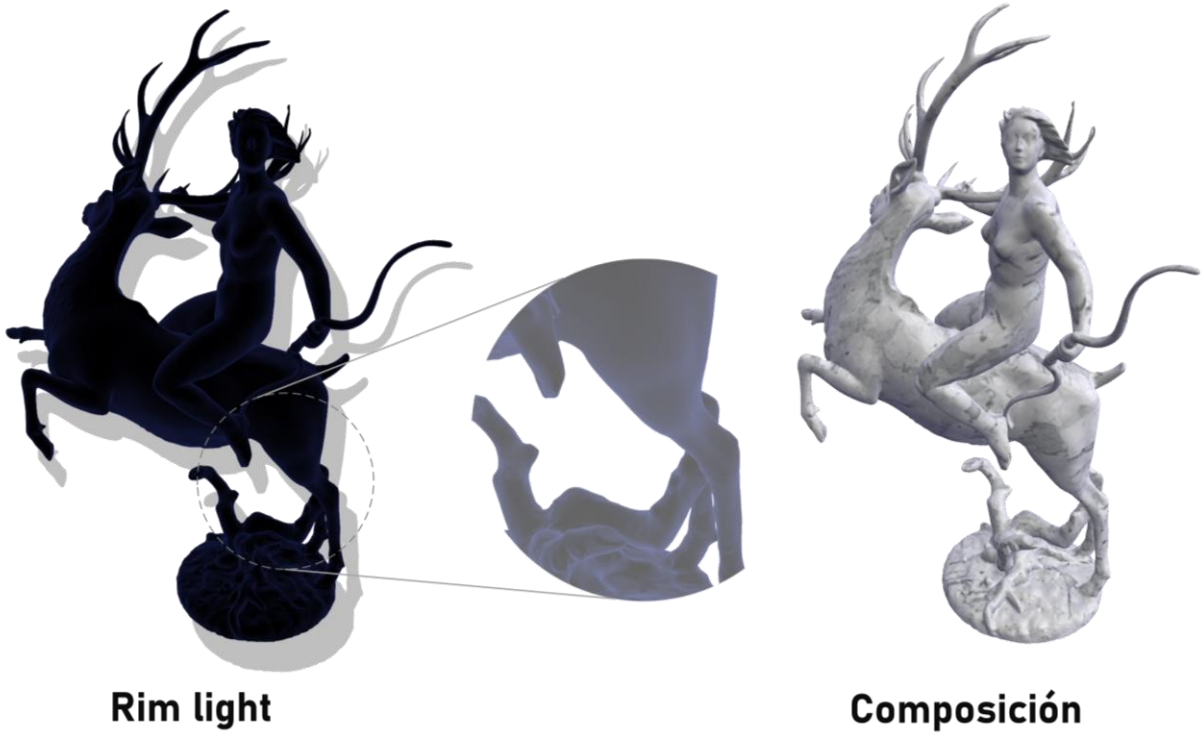


Ilustración 26. Rim light aplicada al modelo de Artemis. Se muestra un detalle del resultado con brillo aumentado para permitir su visualización en papel.

2.2.1.3.5 Atenuación

Algunos de los modelos que antes se plantean mantienen una intensidad constante, al margen del factor de incidencia que suele aplicarse como resultado del producto escalar entre la normal y el vector l de la fuente de luz. Sin embargo, un modelo más realista debería un considerar un factor adicional de atenuación, que vendría dado por la distancia del punto iluminado a la posición de la luz (siempre y cuando esta sea conocida; por ejemplo, esto no sucede en el modelo direccional). Hasta tres modelos se plantean para incluir esta atenuación en la aplicación gráfica, del más básico al más complejo y lento de calcular (Akenine-Möller 2018):

- La atenuación más básica, directa y comúnmente utilizada, donde se indican tres coeficientes que influyen en la atenuación:

$$attenuation = \min\left(\frac{1}{c_1 + c_2 * r + c_3 * r^2}, 1\right) \quad (2.7)$$

- Otra función de atenuación mucho más intuitiva es la siguiente, donde se indican dos parámetros de una luz: máxima y mínima distancia en la que influye la fuente de luz.

$$attenuation = \text{clamp}\left(\frac{\text{r}_{\text{end}} - r}{r_{end} - r_{start}}, 0, 1\right) \tag{2.8}$$

- Una función de atenuación más compleja, utilizada por Pixar, se describe a continuación (Barzel 1997):

$$attenuation = \begin{cases} \text{f}_{\text{max}} \text{e}^{k_0 \left(\frac{r}{r_c}\right)^{-k_1}}, & \text{r} \leq \text{r}_\text{c} \\ \text{f}_\text{c} \left(\frac{r_c}{r}\right)^{s_e}, & \text{r} > \text{r}_\text{c} \end{cases} \tag{2.9}$$

Esta función de atenuación nunca alcanza el valor cero. f_c se alcanza cuando $r = r_c$, mientras que cuando $r < r_c$ la función se aproxima gradualmente a f_{max}. El exponente s_e controla el decrecimiento de la función cuando $r > r_c$.

Al tratarse de una función a trozos es importante que la derivada de la misma sea continua en el punto de transición, $r = r_c$, para generar un modelo de atenuación correcto. Por esta razón, los valores de k_0 y k_1 no suelen ser parámetros variables, sino que se les asignan los siguientes valores:

$$k_0 = \ln\left(\frac{f_c}{f_{max}}\right);\ k_1 = \frac{s_e}{k_0} \tag{2.10}$$

En la Ilustración 27 se muestra una comparativa de los tres modelos de atenuación propuestos; del modelo más básico al modelo utilizado por Pixar.

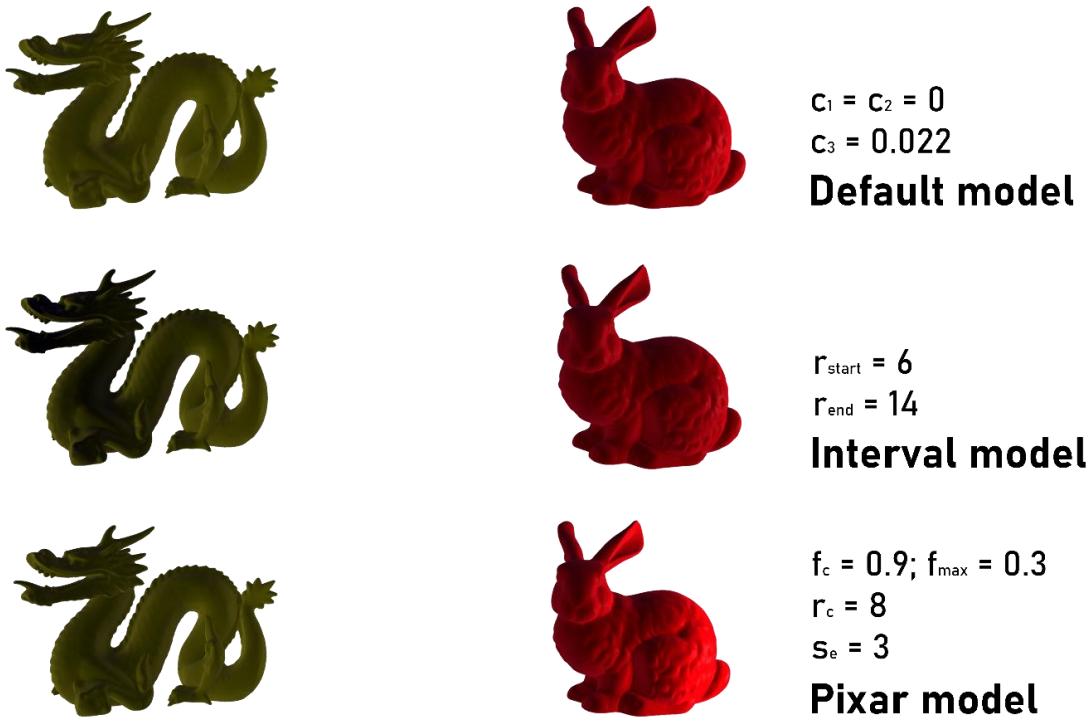


Ilustración 27. Resultados obtenidos tras la aplicación de los tres modelos de atenuación descritos.

Por último, cabe destacar que en el *shader* empleado para el *rendering* de triángulos, todas estas funciones de atenuación se expresan mediante una única subrutina que dispondrá de tres versiones. El modelo de luz deberá seleccionar cuál es el modelo de atenuación que debe aplicarse.

2.2.1.3.6 Sombras

La implementación de las sombras en esta aplicación gráfica se corresponde con la técnica de *shadow mapping*, de tal manera que su generación se vincula a las luces, en las que deberá añadirse una cámara. Dicha cámara servirá para capturar la escena de manera previa al *rendering*, con la diferencia de que no se busca una captura que represente el color de la escena, sino la profundidad, y más concretamente, la mínima profundidad a la que se encuentra un objeto desde esa posición. Dicha captura se engloba en la entidad ***ShadowMap***, un FBO que sólo contiene un *buffer* de profundidad.

De esta manera, dado un punto p en un *fragment shader*, la propia textura de profundidad (*shadow map*) y la matriz de proyección de la cámara empleada en la fuente de luz (más un escalado ($\frac{1}{2}$) y una traslación ($\frac{1}{2}$) para desplazar la coordenada resultante de $[-1, 1]$ a $[0, 1]$, con el fin de acceder a la textura), es posible conocer si

el punto que se está tratando se encuentra en sombra o no. De esta comprobación se obtendrán dos valores, uno para la componente difusa y otro para la componente *specular*. El primero se corresponde con el factor de atenuación de la sombra, mientras que el segundo se encarga de suprimir la componente *specular* cuando el punto se encuentra en sombra.

No todas las luces generarán sombras, y de hecho, se considerará sólo una para tal fin, por lo que en nuestro *shader* habrá que diferenciar mediante subrutinas qué flujo se ha de seguir en función del tipo de luz.

La descripción hasta ahora realizada supone la obtención de sombras con fronteras muy bien definidas, lo cual no produce necesariamente una solución realista, y de hecho, es común ver cierta pixelación en la misma. Para evitar este problema puede utilizarse la técnica de ***percentage-closer filtering*** (PCF) (Reeves, Salesinf, and Cook 1987), que en GLSL se vincula con una herramienta como *textureProjOffset*, la cual nos permite *muestrear* un área cercana a un fragmento, introduciendo así unos bordes más difuminados (Ilustración 28). Al emborronarse la sombra, podríamos hablar de una implementación básica de *soft shadows*, en lugar de *hard shadows*.

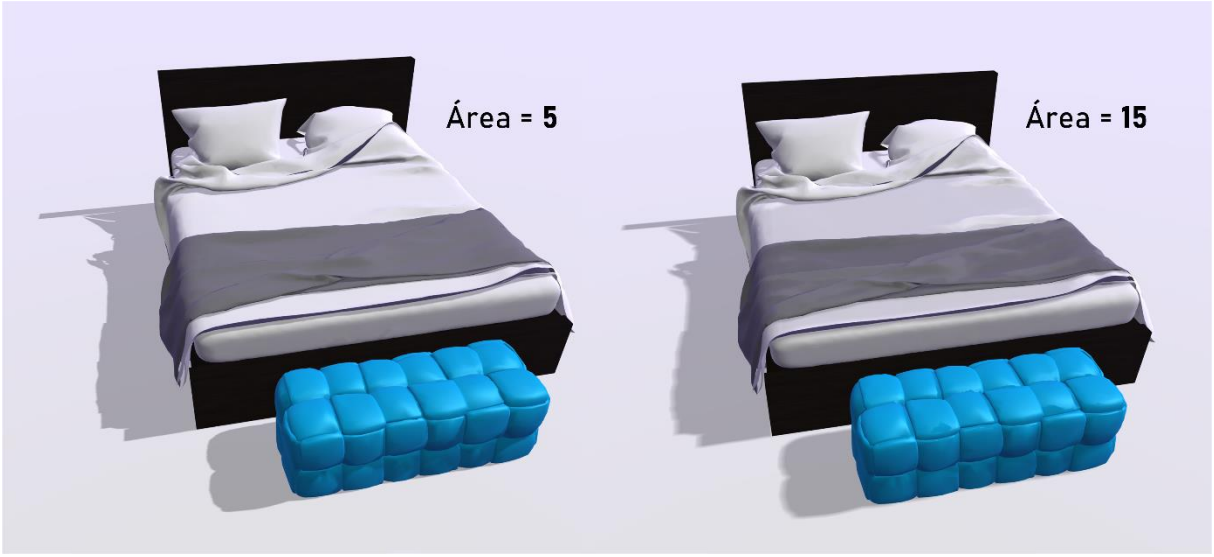


Ilustración 28.Comparativa de sombreado utilizando vecindarios de tamaño 5 y 15.

Sin embargo, si deseamos *soft shadows*, el resultado obtenido no es el más deseable. En primer lugar, la técnica de PCF sólo afecta a la frontera de la sombra, y en segundo lugar, producir un mayor difuminado no es en absoluto escalable, dado que el tiempo de cómputo aumenta drásticamente cuando es necesario muestrear un área mayor. Por ejemplo, el resultado de la Ilustración 30 podría obtenerse con un vecindario mucho más grande y con un tiempo de respuesta muy elevado.

Una solución al problema de la generación de *soft shadows* se propone en la Ilustración 29, donde se parte de un conjunto de vectores de desplazamiento (*offsets*), a los cuales se añade ruido, y finalmente, se trasladan a una malla circular (Wolff 2013). Siempre se parte de una textura estática generada al comienzo de la ejecución que modela el ruido mediante una matriz, no necesariamente de gran tamaño, que almacena en cada celda un conjunto de vectores ($samples = samplesU\ x\ samplesV$) con cuatro valores, donde cada uno de ellos representa dos *offsets*. Por tanto, se origina una textura 3D donde sus píxeles representan información RGBA. Nótese como la información importante se halla en el eje Z; X e Y tan sólo sirven para escoger diferentes *offsets* en fragmentos distintos, para así generar cierta variación.

De esta manera, el proceso que se describe en la Ilustración 29 se representa mediante la siguiente formulación, partiendo de los valores centrales de cada celda:

$$\forall i \in [0, textSize);\ \ \forall j \in [0, textSize);\ \ \forall k \in [0, samples)$$

$$\begin{gathered} x_1 = \frac{k}{samplesU}, y_1 = \frac{samples - 1 - k}{samplesU} \\ x_2 = \frac{k+1}{samplesU}, y_2 = \frac{samples - 2 - k}{samplesU} \\ x_1 = \frac{x_1 + 0.5 + rand}{samplesU}, y_1 = \frac{y_1 + 0.5 + rand}{samplesV} \\ x_2 = \frac{x_2 + 0.5 + rand}{samplesU}, y_2 = \frac{y_2 + 0.5 + rand}{samplesV} \end{gathered} \tag{2.11}$$

$$result[cell] = (\sqrt{y_1} * \cos(2 * \pi * x_1), \sqrt{y_1} * \sin(2 * \pi * x_1), \sqrt{y_2} * \cos(2 * \pi * x_2), \sqrt{y_2} * \sin(2 * \pi * x_2))$$

Una vez construida la textura sólo es necesario trasladarla al *shader* y realizar el muestreo. Dados n samples, se realizarán hasta $n * 2$ muestreos, dado que cada valor final incluye dos *offsets*. Dentro del comportamiento del *shader* se identifican varias ventajas respecto de PCF:

- No siempre se emplean todos los *offsets*, con lo cual es posible reducir carga. Si al muestrear con los cuatro primeros vectores se obtiene un valor de sombra 1 o 0, implica que el fragmento se encuentra totalmente en sombra o alejado de la misma. Por tanto, no es necesario continuar.

- El factor de difuminación se controla con un radio, por lo que el proceso es más escalable.

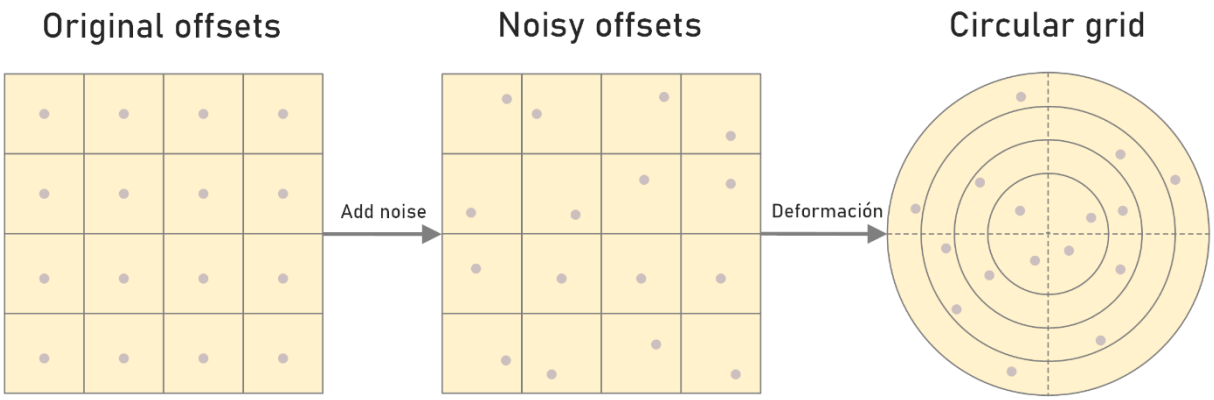


Ilustración 29. Proceso de generación de offsets para la obtención de soft shadows.

Ilustración 30. Representación del modelo anteriormente empleado para ilustrar los resultados de PCF, con el nuevo modelo de sombreado.

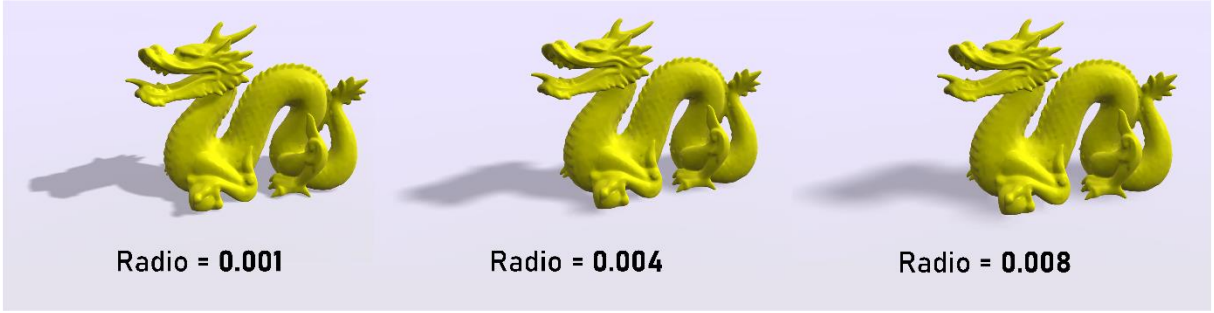


Ilustración 31. Representación de soft shadows con diferentes valores de radio.

Para finalizar con este apartado, es necesario recordar que el *shadow map* se obtenía mediante una cámara, la cual podríamos dar por hecho que utiliza una proyección en perspectiva. Tanto en una *point light* como en una *spot light* esta suposición es correcta, sin embargo, en un modelo direccional se conoce el vector dirección pero no la posición, dado que esta se encuentra demasiado lejos. Por tanto, en este último modelo es más correcto generar las sombras mediante una proyección ortogonal, donde las sombras resultantes son paralelas (en X y Z) a la dirección d que sigue el modelo de luz, como se muestra en la Ilustración 32 y la Ilustración 33.

Tal y como se propone en los modelos de luz, las diferentes proyecciones también se pueden implementar bajo el patrón *Strategy*, especialmente considerando que sólo existen dos diferencias entre ambos sistemas: el cálculo de la matriz de proyección, y el movimiento de *zoom* (se obtiene modificando parámetros distintos en ambos modelos). Esto posibilita de nuevo que una cámara pueda modificar su tipo de proyección durante la ejecución de una manera muy simple.

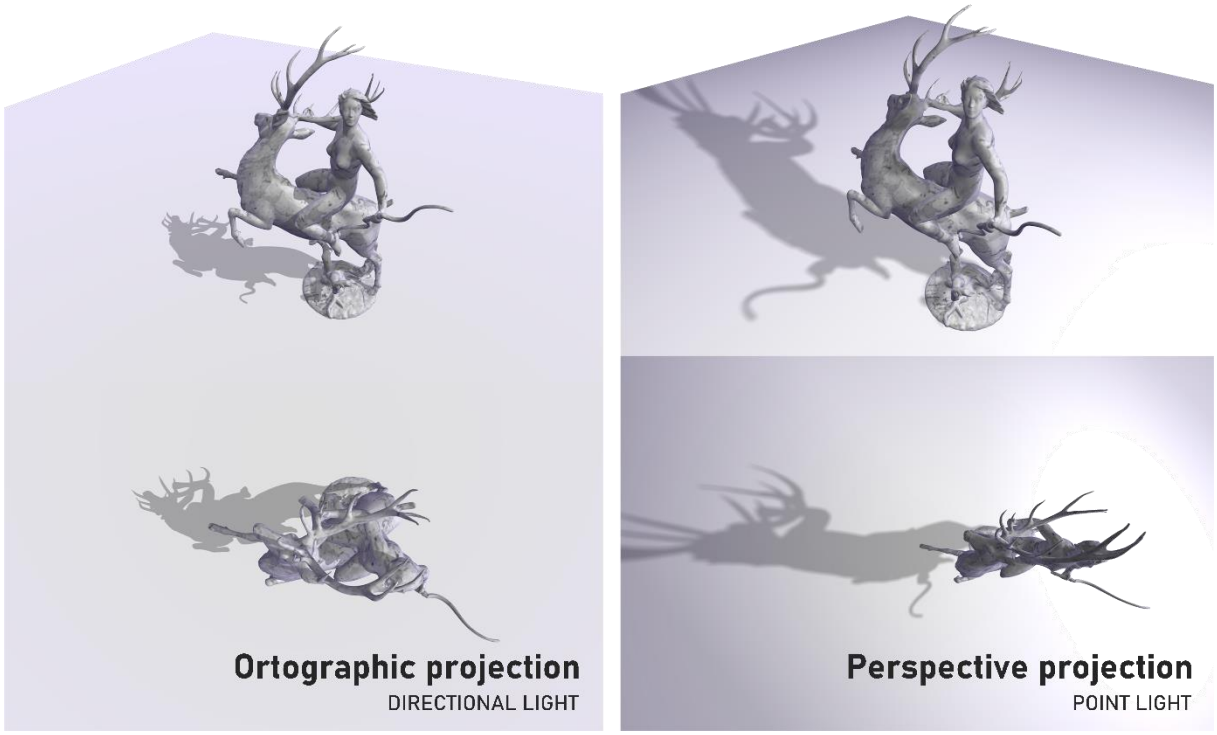


Ilustración 32. Sombras obtenidas con dos modelos de proyección diferentes. 1) Fuente de luz direccional, 2) fuente de luz puntual.

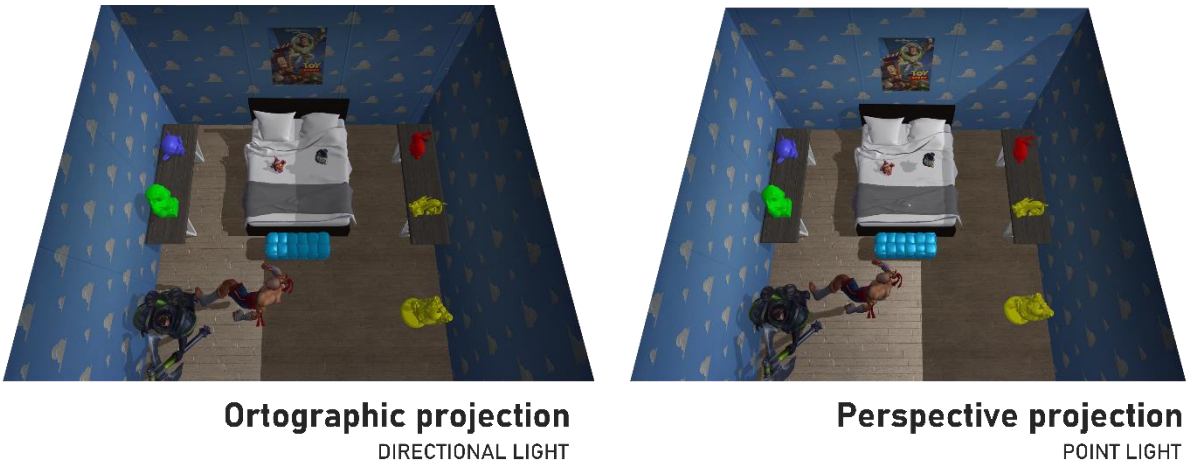


Ilustración 33. Sombras obtenidas con dos modelos de proyección en un escenario alternativo. 1) Fuente de luz direccional, 2) fuente de luz puntual. En la primera imagen, las sombras son paralelas al plano situado al fondo de la habitación. En la segunda, las sombras divergen.

Para completar la descripción de modelos de luz presentes en la aplicación se muestra el diagrama de clases de la Ilustración 34. En dicho diagrama aparece una entidad aún por revisar, *FBO*, que representa un *framebuffer* para llevar a cabo *offscreen renderings.*

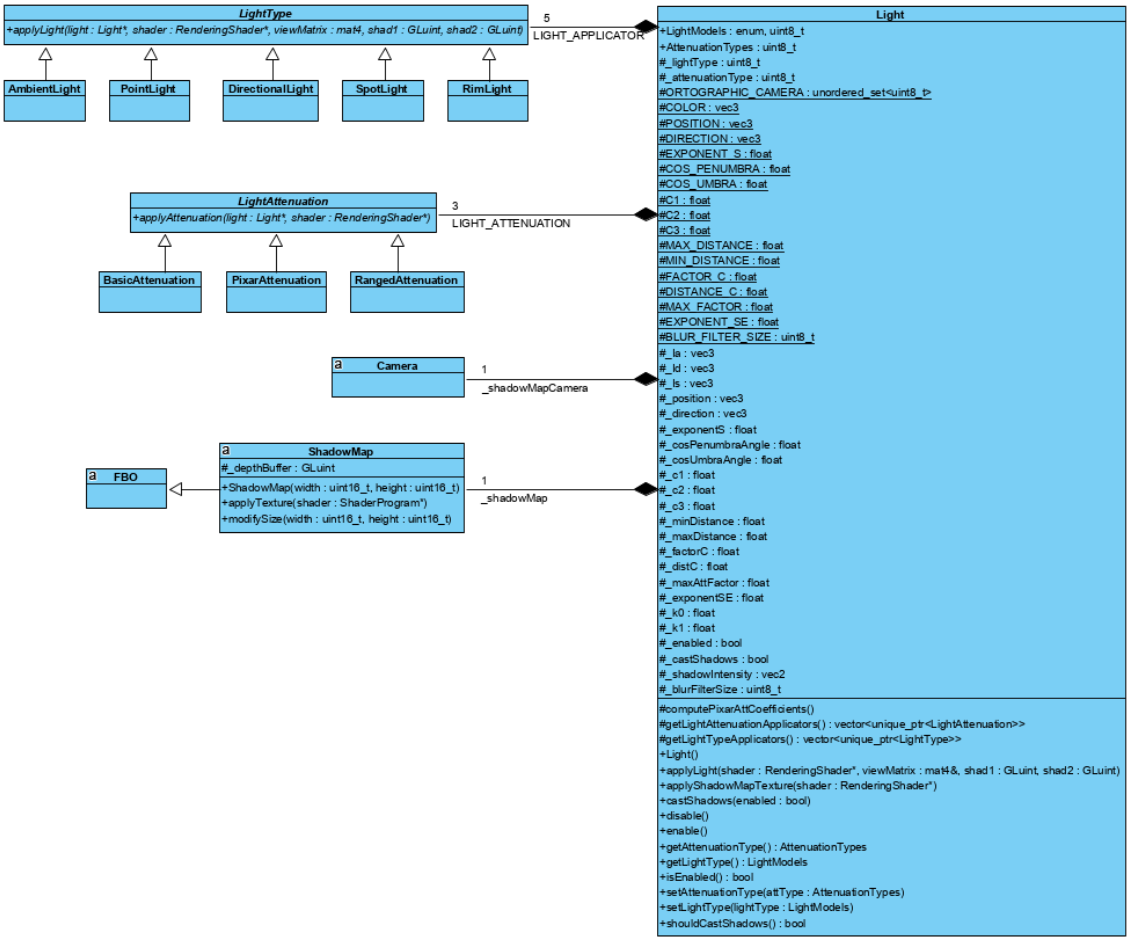


Ilustración 34. Diagrama de clases de todas aquellas entidades relacionadas con la iluminación de la escena.

2.2.1.4 Cámaras

La última herramienta necesaria para visualizar una escena correctamente es una cámara, la cual se define bajo la entidad ***Camera***. La aplicación podrá disponer de múltiples cámaras bajo una instancia de ***CameraManager***, de tal manera que sólo una de ellas se encuentra activa. Los principales aspectos que deben destacarse de la implementación de la cámara son los movimientos que permiten al usuario visualizar la escena con total libertad. Nótese como toda la entrada de usuario se ha descrito con anterioridad, siendo esta tratada por otra entidad, por lo que una cámara sólo recibirá órdenes muy precisas. Normalmente, será tan simple como solicitar un movimiento indicando un valor real, al que hemos llamado velocidad, el cual indica únicamente la fuerza/longitud del movimiento.

Cada movimiento implementado modificará determinados parámetros, los cuales determinarán qué matrices (visión, proyección) deben construirse de nuevo. La situación más común es la reconstrucción únicamente de la matriz de visión, excepto en acciones tales como un *zoom*, que afectan al ángulo de visión de la cámara. Igualmente, sólo se recalculan o modifican aquellas variables que se ven afectadas en cada caso. Por ejemplo, el vector $\overrightarrow{up}$ no suele verse alterado, aunque la inclusión del movimiento *arcball* en la aplicación puede romper este esquema. Los movimientos y acciones desarrolladas se listan a continuación:

- ***Boom***. Traslación vertical en el eje Y de la cámara. Tanto la posición como el objetivo de la cámara sufren la misma traslación.
- ***Crane***. Traslación negativa en el eje Y de la cámara. Se implementa como una llamada al comportamiento de *boom* con una velocidad negativa.

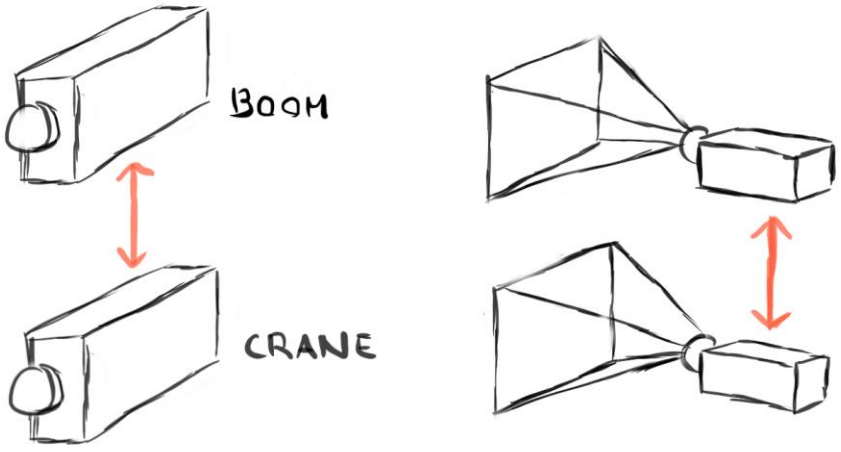


Ilustración 35. Representación gráfica de los movimientos boom y crane.

- ***Dolly***. Traslación en el eje Z de la cámara. La dirección vendrá determinada por el signo de la velocidad especificada. No sólo avanza la posición, sino también el objetivo, lo que indica que se mantiene la distancia entre ambos puntos y no es posible alcanzar o traspasar el punto objetivo. Esto no implica que el comportamiento alternativo al descrito no sea correcto.
- ***Truck***. Movimiento análogo a *dolly*, con la diferencia de que esta traslación se produce en el eje X de la cámara. De nuevo, no se dispone de dos versiones, sino que será el signo de la velocidad el que indique el sentido del movimiento.

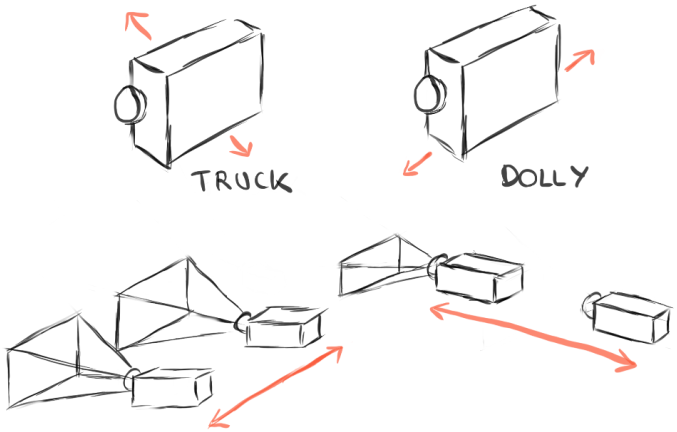


Ilustración 36. Representación gráfica de los movimientos truck y Dolly.

- ***Orbit***. Rotación alrededor del punto objetivo, como si este se tratara del origen del sistema de coordenadas y se produjera una rotación en torno al eje Y.

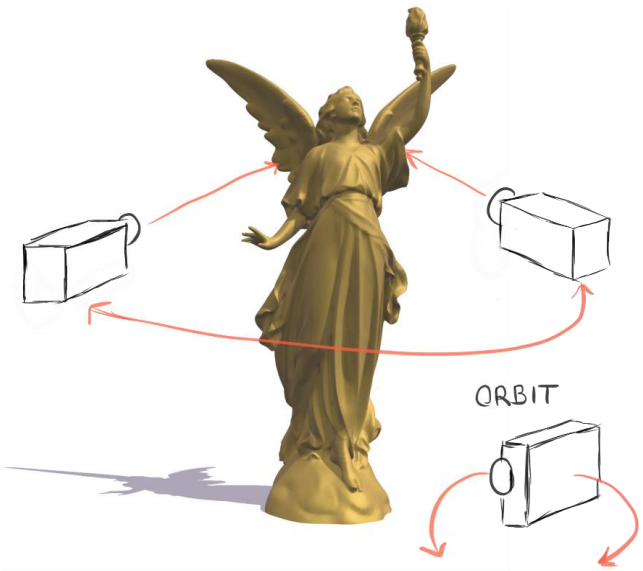


Ilustración 37. Representación gráfica de un movimiento orbit.

- ***Arcball***. Su implementación es tan básica como la del resto de movimientos, pero es sin duda la que más complejidad introduce en términos de control de variables que pueden verse afectadas, dado que es capaz de modificar, por ejemplo, el vector $\overrightarrow{up}$ de nuestra cámara. Este movimiento nos permite rotar alrededor del punto objetivo sin conservar la altura de la cámara. En otras palabras, es posible volar sobre la escena. Su implementación obedece únicamente a una rotación en torno al eje X de la cámara.

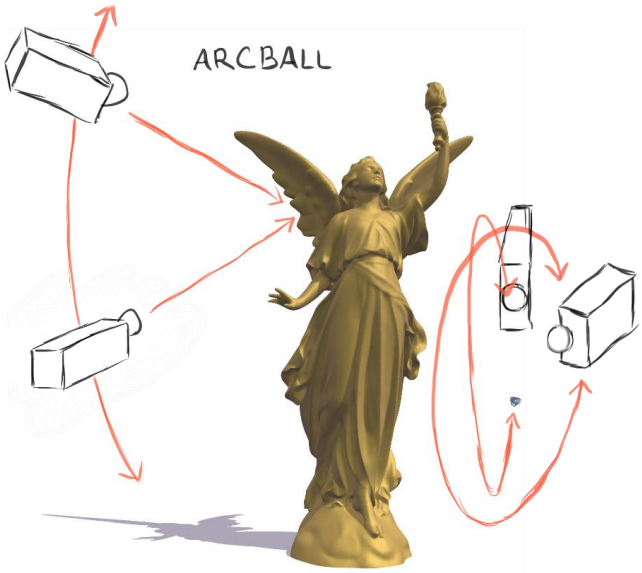


Ilustración 38. Representación gráfica del movimiento arcball.

- ***Pan***. A diferencia de movimientos anteriores, en este caso se mantiene la posición de la cámara y sólo rota el objetivo de la misma en torno a un eje Y situado en dicha posición.

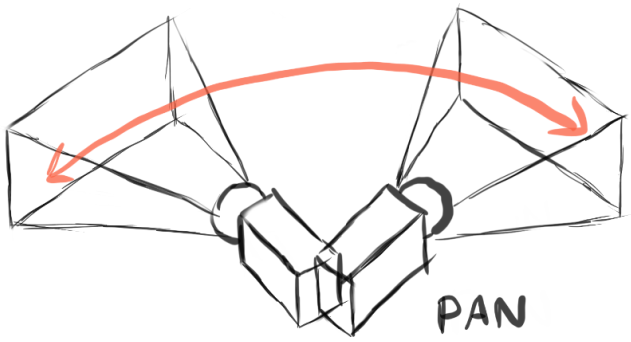


Ilustración 39. Representación gráfica de un movimiento pan.

- ***Tilt***. Como en el caso anterior, con la diferencia de que el eje de rotación será el eje X de la cámara. Además, se limita el ángulo de movimiento a $[-\frac{\pi}{2}, \frac{\pi}{2}]$.

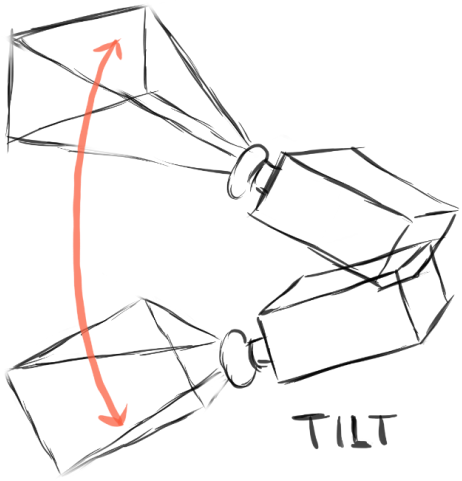


Ilustración 40. Representación gráfica de un movimiento tilt.

- ***Zoom***. Su comportamiento depende la proyección a la que obedece la cámara. En una proyección ortográfica se debe reducir la posición de la esquina inferior izquierda del área de visión, mientras que en una proyección en perspectiva debe reducirse el ángulo de apertura, siempre comprobando que el nuevo ángulo se encuentre en el intervalo $[0, \pi)$.

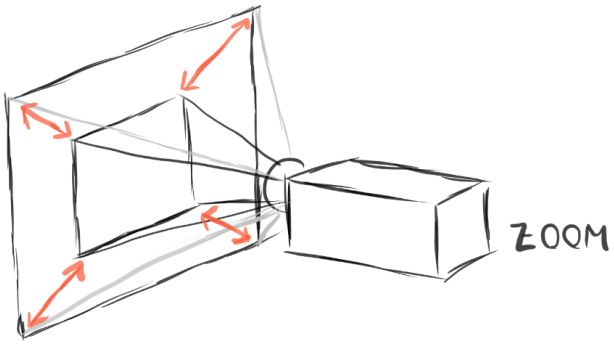


Ilustración 41. Representación gráfica del movimiento de zoom.

- ***Reset*** de cámara. Cuando se construye una cámara, se genera una copia de la misma, de tal manera que si el usuario considera que la configuración actual no es la deseada podrá volver al estado inicial. El comportamiento de este método se reduce a copiar los valores de la cámara de respaldo almacenada. Como la entidad dispone de constructor copia, es suficiente con aislar este comportamiento en un nuevo método que pueda utilizarse tanto en esta acción como en el constructor copia.

Para finalizar, cabe mencionar que la entidad *Camera* se completa con un gran número de métodos que permiten modificar los parámetros de una cámara durante la

ejecución de la aplicación, y que igualmente sirven para construir la instancia (existen demasiados parámetros como para solicitarlos en un constructor). Todos estos métodos *setter* se omiten en el diagrama de clases de la Ilustración 42, con el fin de evitar extender el esquema.

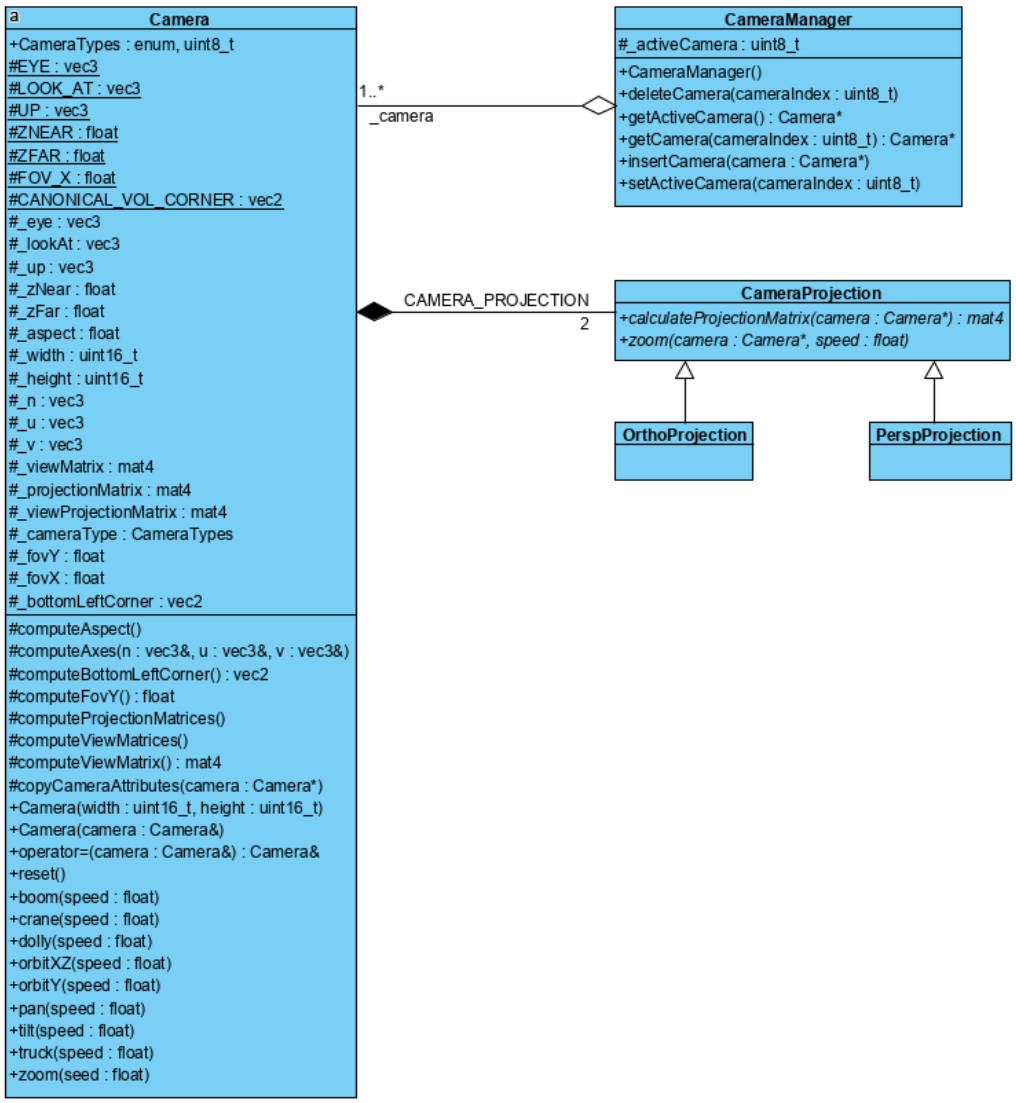


Ilustración 42. Diagrama de clases donde se muestran aquellas entidades relacionadas con una cámara.

2.2.1.5 Shader program

Un *shader* no es más que un programa que define el comportamiento que debe ejecutarse en alguna etapa del *pipeline*. Con el término de *shader program* se hace referencia a un conjunto de *shaders* con un propósito común. A diferencia del código desarrollado en nuestra aplicación principal, el contenido de un *shader* se debe cargar y compilar en tiempo de ejecución a través de OpenGL. Por esta razón, es necesaria una entidad que contenga estas tareas básicas.

Nuestra aplicación no dispone únicamente de un tipo de *shader program*, sino al menos de dos; el primero se encarga del *rendering*, pudiendo incluir tres tipos de shaders (*vertex*, *geometry* y *fragment*), mientras que el segundo se encarga de realizar cálculos en GPU (*compute shader*). Ambos tipos de *shader programs* comparten un amplio conjunto de tareas, las cuales se desarrollan bajo la entidad ***ShaderProgram***. Entre estas tareas se encuentra la lectura, compilación, gestión de subrutinas o el paso de variables *uniform*.

El comportamiento que añade cada tipo de *shader program* es el siguiente:

- ***Shader program* asociado a *rendering*** (***RenderingShader***):
    - Creación del programa, donde se buscan los tres tipos de *shaders* citados. Dado un nombre base, estos se buscan añadiendo algunos de estos sufijos: *-vert.glsl*, *-frag.glsl* o *–geo.glsl*.
    - Activación de subrutinas asociadas a los tres tipos de *shaders* que pueden estar contenidos (de manera previa al *rendering*).
- *Shader program* para computación paralela (no asociada con *rendering*; ***ComputeShader***).
    - Creación del programa, utilizando el prefijo *–comp.glsl* en esta situación.
    - Consulta del máximo tamaño de un grupo en cualquier eje. El tamaño de un grupo está mucho más limitado que el número de grupos, por lo que para ejecutar un *compute shader* se ha empleado el máximo tamaño de grupo y se ha calculado el número de grupos necesarios como $ceil(\frac{tamaño\ requerido}{máximo\ tamaño\ de\ grupo\ en\ el\ eje\ i})$. Otra alternativa sería fijar el número de grupos y seleccionar el tamaño de cada grupo en función del número de elementos con el que operar ($tamaño\ requerido$ en la formulación anterior).
    - Definición de ***Shader Storage Buffer Objects*** (SSBO). Se diferencian al menos dos situaciones: se dispone de un vector inicial que alimenta el SSBO o sólo se conoce el tamaño de dicho vector (y el de tipo de elemento que alberga). La segunda situación planteada está más vinculada a la escritura de un SSBO (cuando no es

importante el valor inicial), mientras que el primer escenario implica al menos la lectura de los datos del SSBO.

- Extracción de datos de un SSBO.
- Ejecución de *shader*. Se añade un *memory barrier* con todos los bits activados, dado que no se reconoce una determinada necesidad.
- Enlazado de *buffers* de entrada/salida (SSBO), indicándose sus identificadores en un vector. Es importante destacar que el orden indicado en el vector debe corresponderse con el identificador indicado en el *compute shader*.

De manera similar a la gestión de texturas y materiales, se define una base de datos de *shaders* (***ShaderList***), desarrollada bajo el patrón *Singleton*, y por tanto, globalmente accesible. De nuevo, se aplica la inicialización perezosa para evitar cargar y compilar *shaders* en la etapa inicial de la aplicación.

Para finalizar este punto, es necesario destacar el esfuerzo realizado para evitar duplicar código en diferentes *shaders*. Para ello se define una sintaxis de inclusión de librerías, tal como `#include <path>`, que consiste en una única línea que puede añadirse en cualquier punto del fichero, aunque resulta evidente que debe posicionarse antes de emplear los métodos o estructuras definidas en la librería. De esta manera, el paso posterior a la carga del contenido de un *shader* es el procesamiento del mismo en busca de esta sintaxis, así como la sustitución de aquellas líneas identificadas por su correspondiente bloque de código. Cuando no es posible acceder a la ruta indicada para la librería, se notifica este error como si se tratara de cualquier otro error de programación en el *shader*.

Nótese como el objetivo es evitar la duplicación de código, lo que implica que una librería puede emplearse en múltiples *shaders*. Por esta misma razón, las librerías ya solicitadas se almacenan en una estructura de datos común a *cualquier shader program*, asociando nombre y su contenido. Este almacenamiento evita únicamente la demora asociada al acceso y la carga del contenido de un fichero, a partir de la segunda solicitud de uso de la librería.

Toda esta relación de *shader programs* se puede observar en el diagrama de la Ilustración 43, donde también se muestra el contenedor globalmente accesible de *shaders*, con dos vectores, uno por cada subclase.

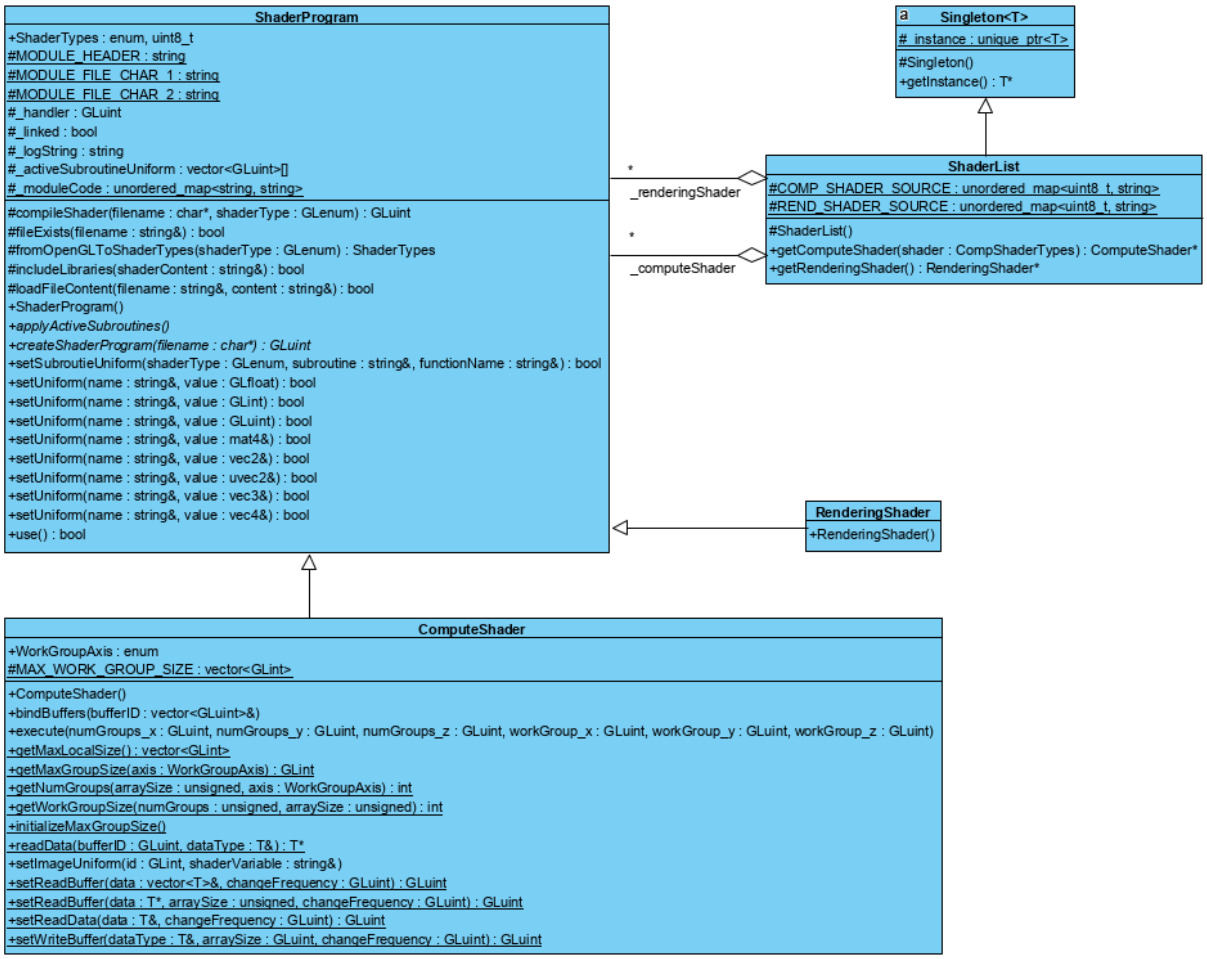


Ilustración 43. Diagrama de clases donde se representan aquellas entidades relacionadas con los shader programs.

2.2.1.6 Modelos y escenas

Una vez se dispone de múltiples modelos de iluminación, materiales y una entidad para visualizar la escena, es posible cargar y generar modelos para su posterior representación. Todos estos modelos se podrán agrupar entre ellos, y será desde la definición de una escena desde donde se *rendericen*. Por tanto, se distinguen al menos tres entidades en esta descripción:

- ***Model3D***. Se trata de un modelo abstracto, donde se incluyen todos los atributos que pueden ser necesarios, y define aquellos métodos que deben implementarse en las subclases, desde comportamientos en la etapa de *rendering* hasta una función de carga que será iniciada por el grupo al que pertenece (o desde una escena si se inicia manualmente). Igualmente, también define aquellos comportamientos básicos que pueden ser comunes a todos los modelos, como el núcleo del *renderizado* de una nube de puntos, de una malla de triángulos o de una malla de alambre, entre otras opciones.

Todas las implementaciones de esta clase se definen mediante herencia: aquí no es necesaria la flexibilidad que se buscaba en los modelos de luces, ni existe diferencia alguna en los atributos que necesita cada tipo de modelo (y si la hubiera, debería ser muy específica).

Cualquier modelo de la aplicación se define a su vez mediante un conjunto de componentes (***ModelComponent***), que no son más que piezas del modelo. En el caso de un modelo importado desde un fichero OBJ es común encontrar este escenario, donde cada pieza tiene unas propiedades de superficie diferentes.

- ***Group3D***. Se define como una implementación de la clase abstracta *Model3D*, aunque un grupo no se *renderiza* como tal, sino que sólo agrupa modelos que mantienen en común una matriz de transformación, y por tanto, un espacio en la escena. Muchos de los comportamientos básicos deben ser redefinidos para que se apliquen sobre todos los modelos contenidos.
- ***Scene***. Se trata principalmente del contenedor de un grupo raíz, que conforma la escena, pero también gestiona el *rendering* de la misma. Nótese como un grupo puede gestionar su *renderizado*, pero existe cierta preparación antes del mismo que debe realizarse en alguna entidad (*Scene*). Igualmente, una escena debe encargarse de definir sus modelos, luces y cámaras. En posteriores iteraciones también será la entidad que controle la simulación de este trabajo, para lo cual podrá definirse una subclase mucho más específica y preparada para tal tarea.

Por tanto, en este punto conocemos las tres entidades encargadas de estructurar un escenario, de tal manera que una escena gestiona uno o varios grupos, y un grupo contiene uno o más modelos. Dichos modelos a su vez pueden representar cualquier objeto, aunque en esta versión inicial de la aplicación se implementan hasta dos tipos:

- Modelo cargado desde un fichero OBJ (***ModelOBJ***). En lugar de utilizar una librería externa, se desarrolla una función muy básica encargada de leer este tipo de ficheros, con el objetivo de reducir el tiempo de carga. La principal desventaja del método desarrollado es que el contenido del fichero

debe seguir una estructura muy concreta, de tal manera que deben indicarse vértices, normales, y coordenadas de textura, y por tanto, los triángulos también deben hacer referencia a toda esta geometría. Cuando no se da esta estructura, es posible modificar la descripción del modelo mediante programas como Blender.

- Plano (***PlanarSurface***). Este modelo se define por una anchura, una profundidad y un número de divisiones verticales y horizontales, donde la esquina superior derecha se vincula con una coordenada textura que también es posible indicar. Por tanto, no es más que una matriz que debe construirse con cierto nivel de detalle. Cuando el plano contiene materiales que aplican la técnica de *displacement mapping* es común dar un gran nivel de detalle, dado que son los vértices los que se desplazan de acuerdo a un *height map*, y no los fragmentos. En cualquier otra situación, es suficiente con la definición más simple de un plano, dada por sólo cuatro vértices.

Más allá de las entidades que engloban un conjunto de componentes con una geometría y topología, y que se encargan de su *renderizado*, es necesario destacar la función de un VAO, la entidad encargada de trasladar dicha geometría y topología a GPU. Un VAO a su vez no es más que una estructura que contiene un conjunto de *buffers* relacionados con la geometría, *Vertex Buffer Object* (VBO), y con la topología, *Index Buffer Object* (IBO). En el caso de este último *buffer*, su definición es tan simple como un conjunto de índices que indican cómo se relacionan los vértices. En función del tipo de topología empleada, será necesario incluir un valor indicador de fin de primitiva cada cierto número de índices. Cuando trabajamos con triángulos, se debe incluir dicho valor cada tres índices, mientras que con una tira de triángulos (*triangle strip)* se desconoce cuál es el período sin conocer el modelo con el que se trabaja.

Para definir un VBO se desarrollan al menos dos enfoques:

- Un *buffer* por tipo de geometría. En el caso de un vértice del que se conoce su vector normal y su coordenada de textura, serán necesarios tres VBOs.

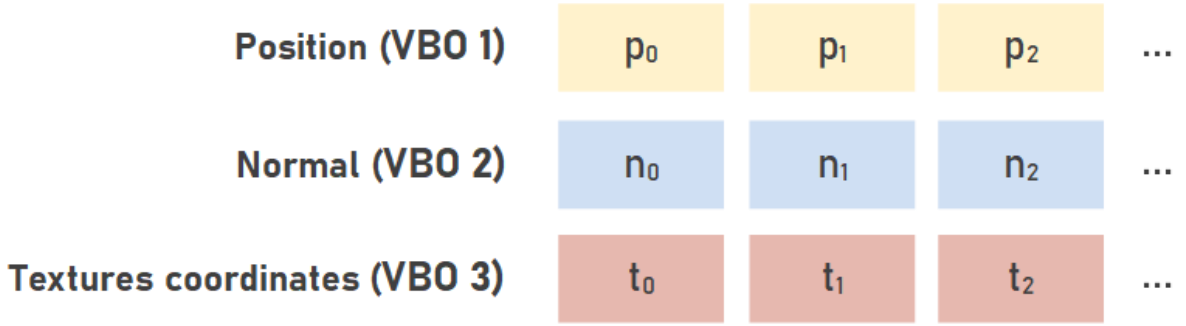


Ilustración 44. Representación de la definición de tres VBOs utilizando el enfoque más básico.

- *Interleaved buffer*. Sólo define un VBO en el que la geometría se intercala. La principal ventaja de esta definición es que la información del modelo se encuentra más compacta, y que al trasladar la misma a GPU es suficiente con una llamada en la que se indica un único vector (por ejemplo, un puntero de valores *float*). La desventaja se encuentra principalmente en escenas animadas, dado que es posible que la posición o la normal de un vértice modifiquen su valor, pero no es frecuente que suceda esto mismo con la coordenada de textura. Es decir, para notificar los cambios habría que indicar un vector con los tres tipos de atributos propuestos, incluso aunque las coordenadas de textura no se han visto alteradas.

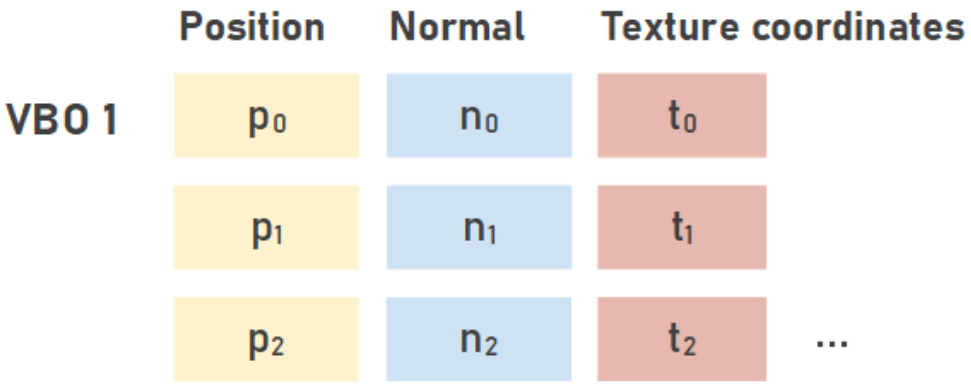


Ilustración 45. Representación de un interleaved VBO.

Por tanto, la solución perfecta no se encuentra en alguno de estos enfoques, sino en una combinación de los mismos, de tal manera que la información dinámica se aisla del resto mediante un VBO dedicado.

Todas y cada una de las entidades citadas, y las asociaciones entre ellas, se muestran en los diagramas de la Ilustración 46 y la Ilustración 47.

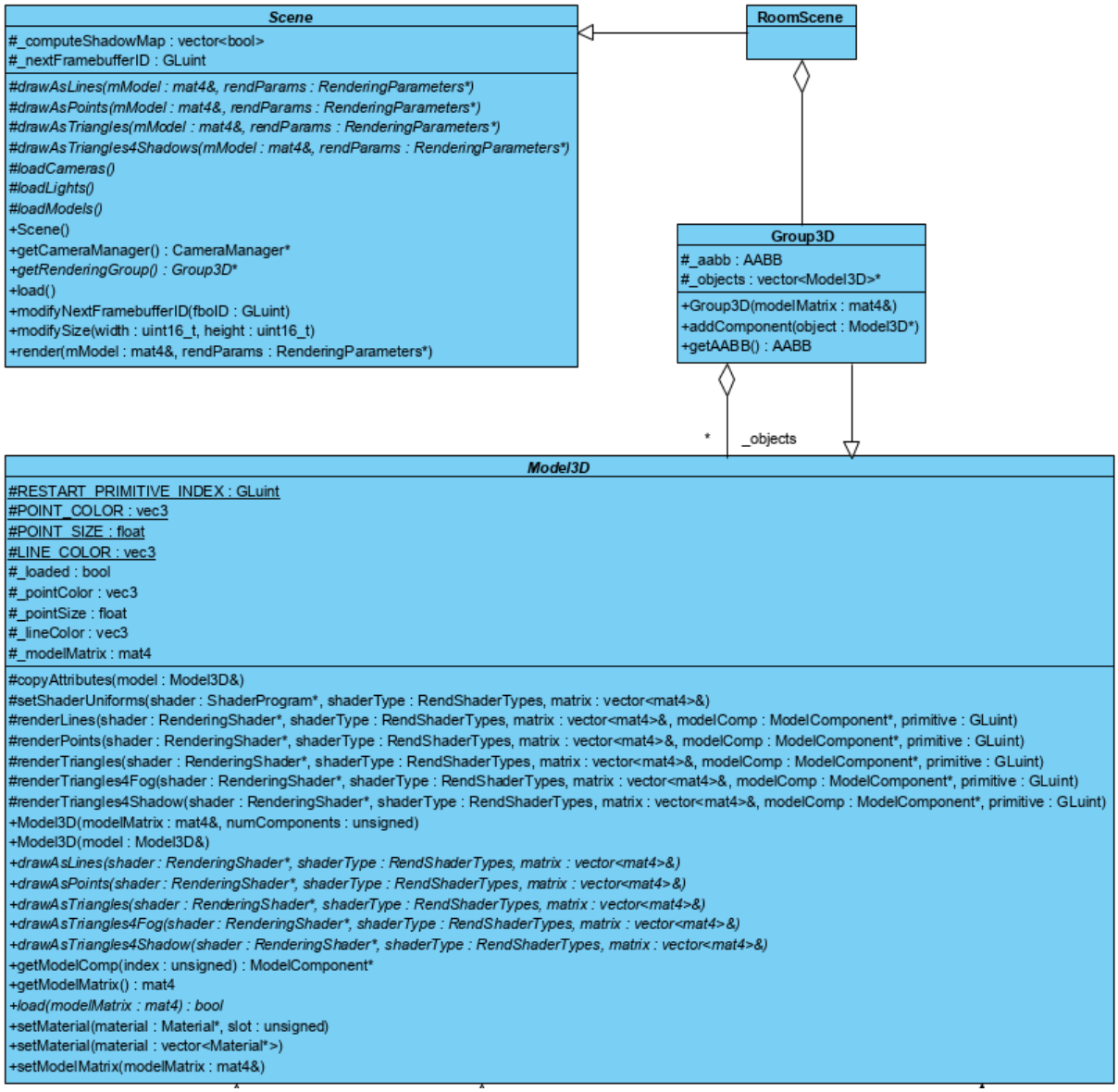


Ilustración 46. Diagrama de clases donde se muestran aquellas entidades relacionadas con los modelos y las escenas de la aplicación.

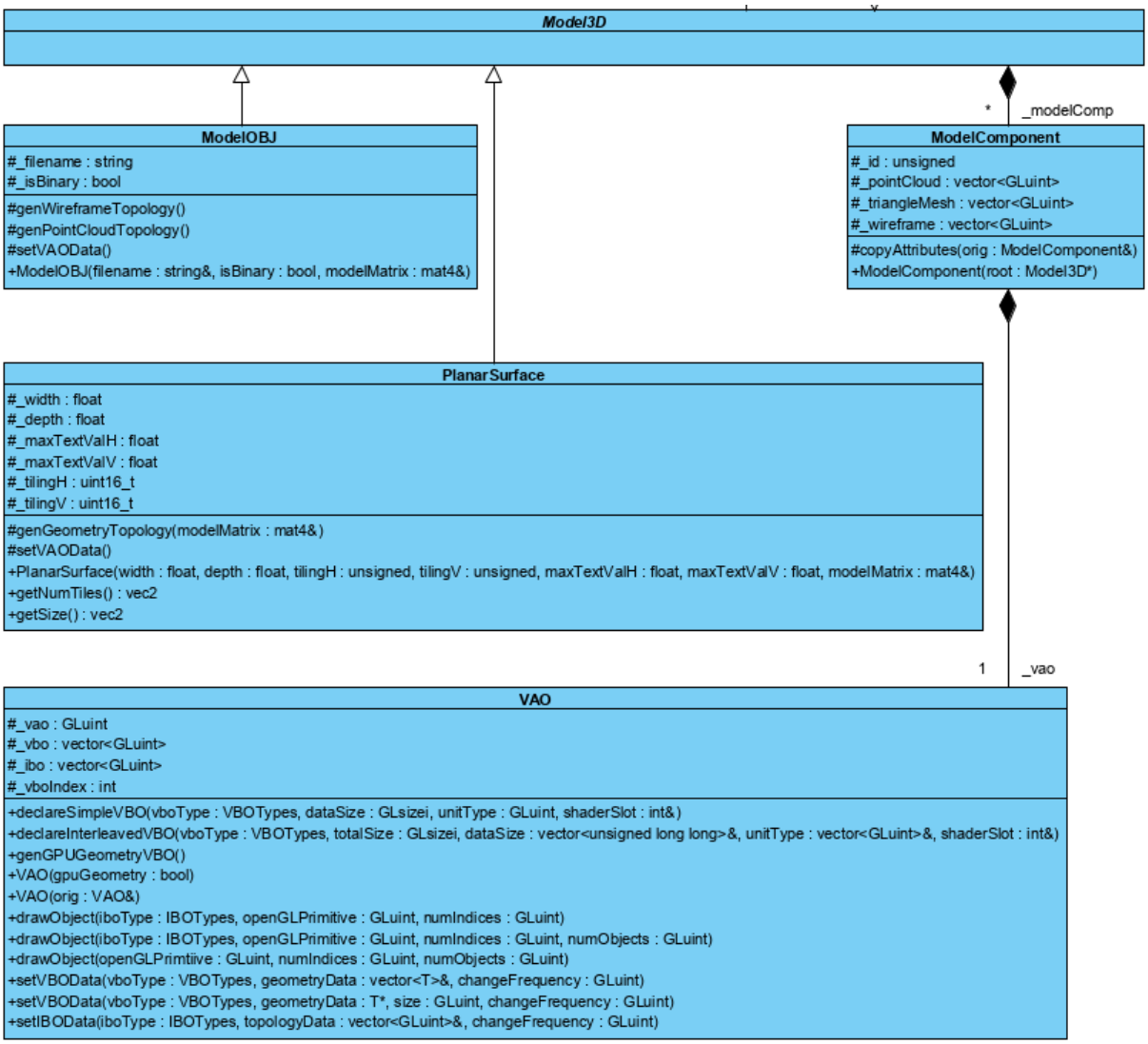


Ilustración 47. Diagrama de clases donde se representan las entidades relacionadas con los modelos de la aplicación.

2.2.1.7 Otras funcionalidades

En este último apartado de la aplicación se describen otras piezas cuya funcionalidad es secundaria, al menos en este punto. Entre estas piezas se encuentran el FBO encargado de capturar la ventana o diversas fuentes de utilidades en la aplicación.

2.2.1.7.1 Captura de pantalla

Un FBO habilitado para capturar la ventana se define en primer lugar mediante la clase abstracta ***FBO***, donde se reúnen atributos básicos, como el identificador del FBO, el tamaño de sus *buffers* (en forma de anchura y altura) y un indicador de éxito en la creación del mismo.

Por tanto, cada implementación de un FBO es responsable de definir sus *buffers*. En el caso de una captura de pantalla son necesarios hasta tres, además de un FBO adicional:

- El primer FBO se vincula únicamente con un *buffer* de color (RGBA), donde se almacenará el resultado final.
- Al segundo FBO se le ha denominado *multisampled* FBO, y este será el *framebuffer* donde se almacene la escena *renderizada*. Por esta razón, son necesarios dos *buffers*, color y profundidad.

El propio contexto de GLFW mantiene un *framebuffer* como el último descrito, sobre el cual también aplica técnicas de *antialiasing*, considerando que se define un número de *samples* al comienzo de la aplicación. Sin embargo, nuestro FBO no dispone de tal funcionalidad, por lo que se debe implementar mediante un *framebuffer* con *multisampling*, que recoge la escena, y una operación de transferencia del resultado obtenido al FBO cuyo identificador se define en la clase base (***glBlitFramebuffer***).

Cuando se crea el *buffer* de color y de profundidad asociado al FBO de *multisampling*, se pueden indicar el número de *samples* deseados. De esta manera, cuando este número es 0 es posible simular el problema de partida (Ilustración 48).

La entidad aquí descrita también gestiona el almacenamiento de la captura en el sistema mediante la librería LodePNG. Para evitar crear una espera innecesaria en la aplicación, la operación de almacenamiento se desarrolla en un hilo diferente.

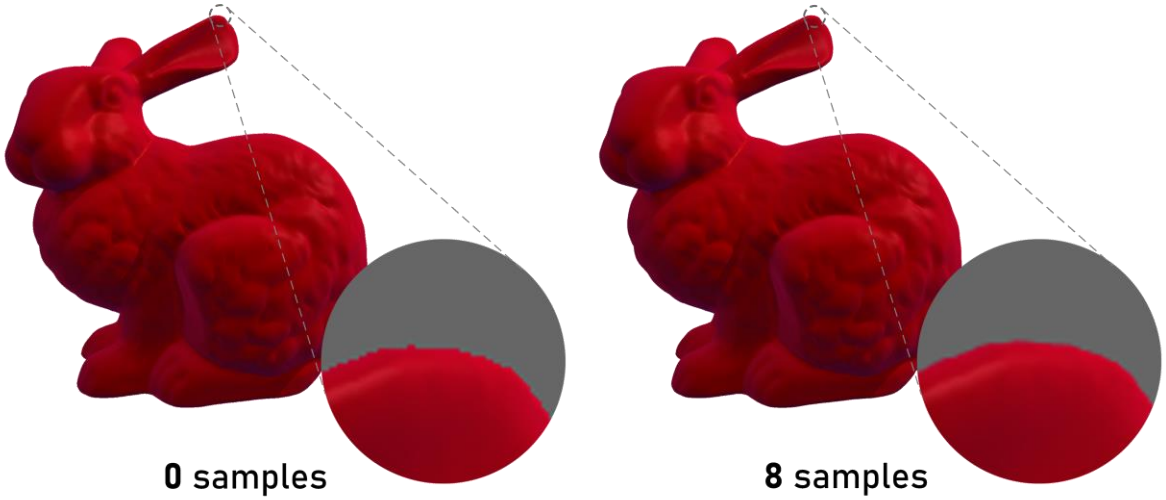


Ilustración 48. Capturas de pantalla con diferente número de samples.

2.2.1.7.2 Bloom effect

Mediante este efecto podemos *renderizar* materiales que emiten brillo, aunque no forman por sí mismos una fuente de luz más. La principal dificultad de este método recae en la representación de dicha emisión, dado que una característica común es la difusión de la emisión en un área cercana. Esta difusión se podría obtener mediante algún filtro de emborronamiento, pero aún quedan otros problemas por resolver. Por esta razón, se introduce este apartado para describir cómo se resuelve este efecto.

En primer lugar, para desarrollar este efecto se considera una entidad, ***BloomFBO***, que gestionará el proceso y todos los *buffers* que se generan (nótese como el emborronamiento no se introduce en el *pipeline* de *rendering* de la escena principal, sino antes de este, dado que el emborronamiento no afecta a todos los modelos). Se procede a describir este proceso mediante algunos pasos (LearnOpenGL, 2017):

1. Enlazado de FBO, el cual está compuesto por dos *buffers* de color y uno de profundidad. Se compone de dos *buffers* dado que este efecto no aplica el mismo *rendering* que el de la escena normal, por tanto, para evitar crear otro *shader*, se incluye una nueva salida en el *shader* que se empleaba hasta ahora. Es decir, nuestros *shaders* (o al menos aquellos que utilizan como primitiva un triángulo) producen dos salidas: escena iluminada (salida de ejecución normal) e iluminación de materiales emisivos (Ilustración 49).
2. *Renderizado* de la escena sobre el FBO antes descrito.
3. Emborronamiento de la segunda salida obtenida. El algoritmo desarrollado es un filtro gaussiano que consta de dos pasos, un filtrado horizontal y otro filtrado vertical. De nuevo, son necesarios dos *buffers* de color, dado que en cada iteración se parte de uno de ellos y se dibuja sobre el siguiente (excepto en la primera iteración, en la que se parte de una textura de color del FBO).

   El *rendering* que se implementa en esta etapa no se ha descrito hasta ahora. Partiendo de una textura (captura de materiales emisivos) podemos renderizarla sobre un plano. En este caso, se utiliza un cuadrado cuya máximo valor de coordenada de textura es 1. De esta manera, se trate o no de un cuadrado, la posición final se puede calcular como sigue (para ocupar el *viewport* completo):

$$gl\_Position = vec4(textCoord * 2 - 1, 0, 1) \tag{2.12}$$

En el propio algoritmo, el máximo tamaño disponible para la matriz del filtrado es 5, dado que los pesos de la misma se indican mediante un vector definido en el *shader*. El comportamiento del filtrado vertical y horizontal podría implementarse bajo una misma formulación, aunque al darse en diferentes ejecuciones también es posible seleccionar el comportamiento mediante una subrutina que ofrece las dos posibilidades. Si intentáramos generalizar la fórmula se obtendría lo siguiente, considerando que $\mathrm{horizontal}$ y $\mathrm{vertical}$ son dos valores *booleanos* que pueden actuar como valores numéricos para suprimir un cálculo cuando estos adoptan el valor cero, y que N es el tamaño del *kernel*.

$$textOffset = \frac{1}{textureSize(texBright)}$$

$$color = texture(texBright, textCoord).rgb * weight[0]$$

$$\sum_{i=1}^{N-1} color += texture(texBright, textCoord + vec2(textOffset.y * i * horizontal, textOffset.y * i * vertical)).rgb * weight[i] \tag{2.13}$$

$$\sum_{i=1}^{N-1} color += texture(texBright, textCoord - vec2(textOffset.y * i * horizontal, textOffset.y * i * vertical)).rgb * weight[i]$$

Nótese como no es posible que $horizontal = vertical$, razón por la cual se establecen únicamente dos sumatorias, en lugar de cuatro. Por otro lado, es posible indicar tantas iteraciones como sean necesarias, teniendo en cuenta que dadas n iteraciones serán necesarias 20 ejecuciones del *shader* que implementa el filtrado.

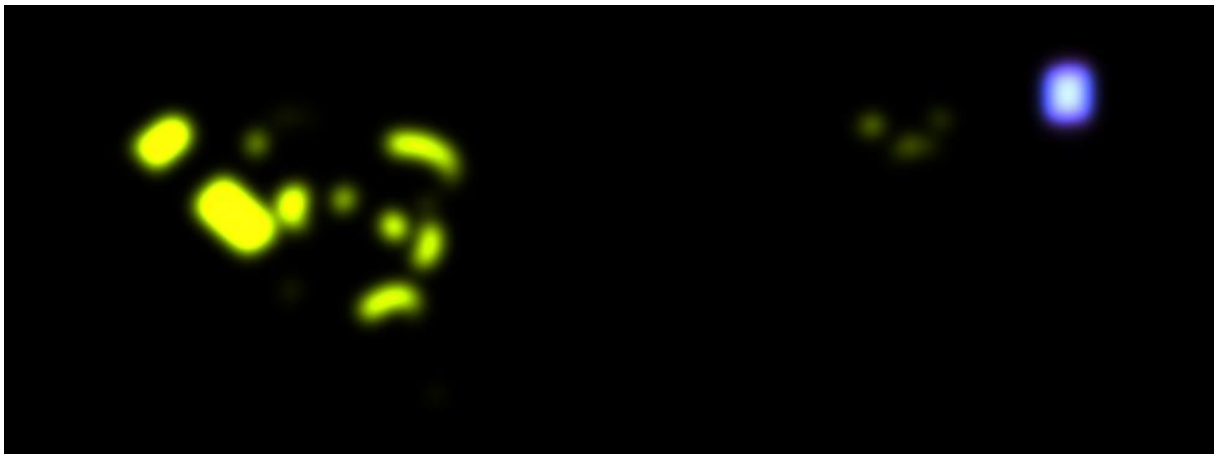


Ilustración 49. Rendering de materiales emisivos en tres objetos diferentes.

Ilustración 50. Rendering de un mismo modelo aplicando sobre el material emisivo un filtro gaussiano con kernel de tamaño 5 y diferente número de iteraciones.

4. Por último, una vez se obtiene una textura como la de la Ilustración 49, junto a la textura almacenada en el primer *buffer* de color del FBO (escena iluminada tal y como se producía hasta ahora), es posible sumar ambas y *renderizar* dicha suma sobre un plano (tal y como hacíamos durante el filtro gaussiano).

Tanto este *framebuffer* como el anterior, *FBOScreenshot*, se presentan en el diagrama de la Ilustración 51, junto al *framebuffer* encargado de capturar el mapa de profundidad de la técnica de *shadow mapping*, ***ShadowMap***.

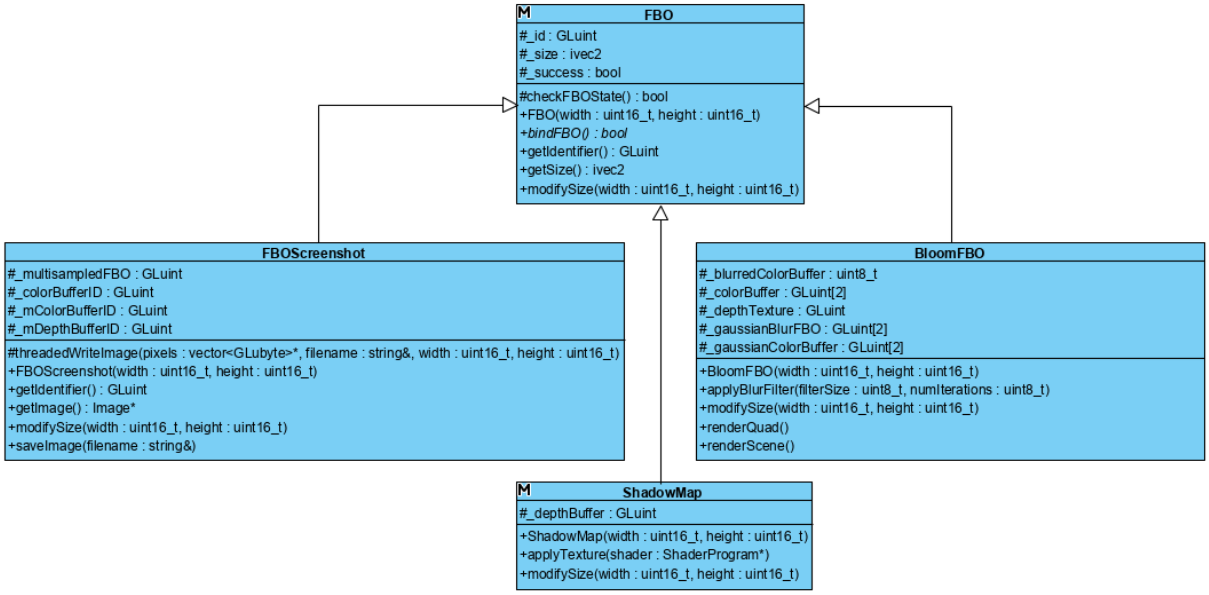


Ilustración 51. Diagrama de clases de framebuffers implementados.

2.2.1.7.3 Utilidades de OpenGL

Por último, cabe destacar el contenedor de utilidades de OpenGL, ***OpenGLUtilities***, donde se da acceso a VAOs ya construidos, listos para su *renderizado*. Por ejemplo, permite obtener la descripción de un cuadrado para un escenario como el que se describe en *Bloom effect*. Las utilidades contenidas en esta clase se pueden observar en la Ilustración 52.

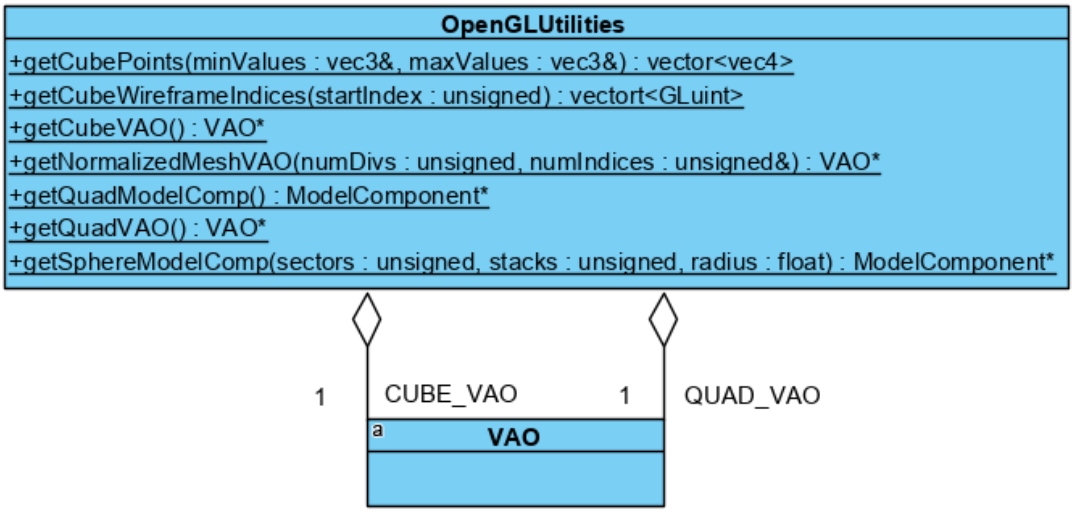


Ilustración 52. Diagrama de clases del contenedor de utilidades de OpenGL.

2.2.2 Diagrama de clases

Una vez desarrolladas las entidades principales que se incluyen en la aplicación gráfica, podemos representar el conjunto de relaciones existentes entre todas ellas.

En particular, es especialmente valioso captar aquellas relaciones que no se han podido representar en los diagramas de clases anteriores, dado que se daban entre entidades de secciones diferentes. Además, cabe destacar que tanto el diagrama que se presenta en esta sección (Ilustración 53) como todos los diagramas anteriores pueden presentar modificaciones en cualquiera de las iteraciones posteriores; aquí sólo se describe todo aquello que posibilita el *rendering* de un escenario. De hecho, la aplicación contiene en este punto otras muchas entidades que han sido omitidas por no estar vinculadas con el dibujado de una escena, y por tanto, se presentarán en las siguientes iteraciones. Este es el caso, por ejemplo, de una entidad dedicada exclusivamente a proveer acceso global a tests de intersección entre muchos de los pares de primitivas que se utilizan.

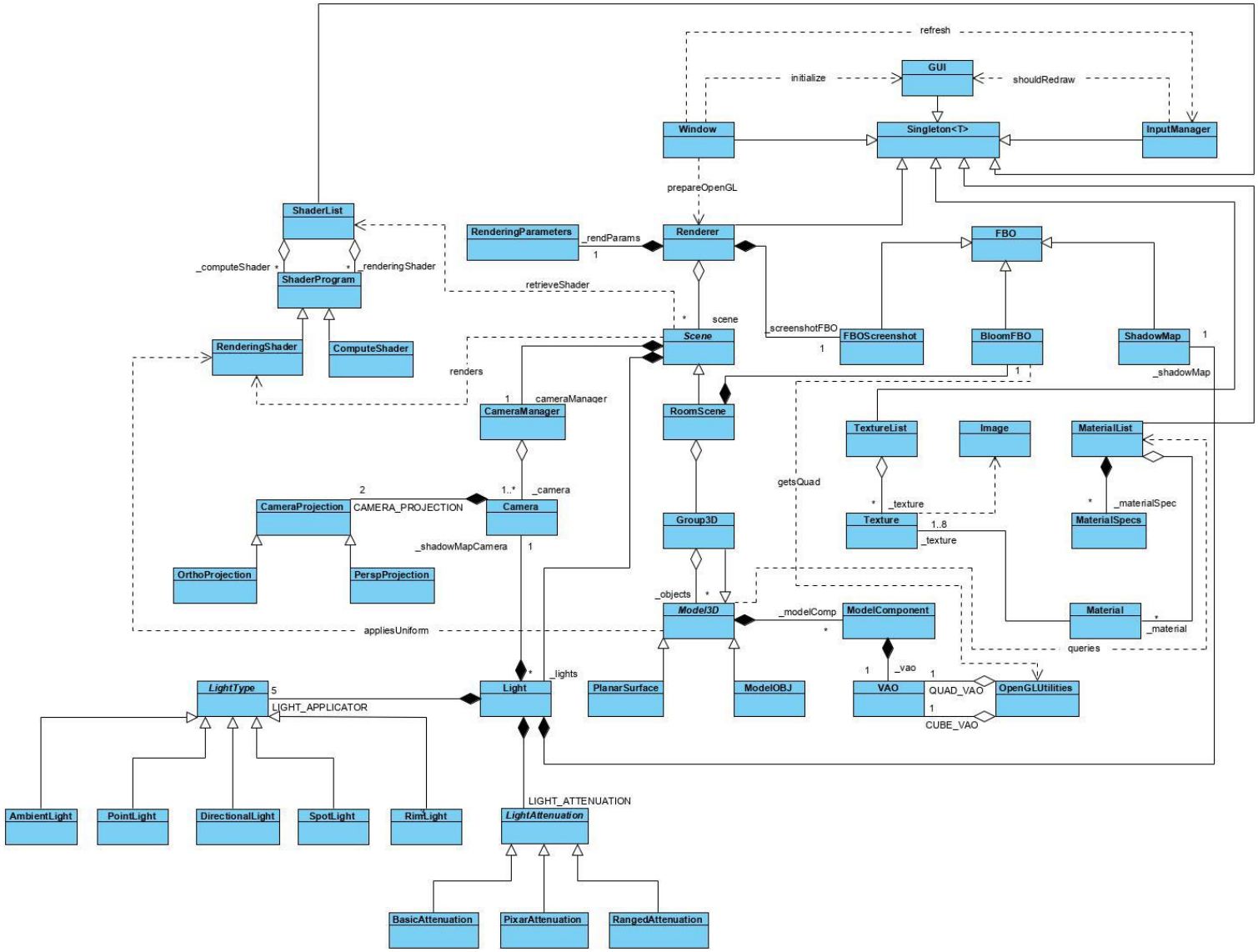


Ilustración 53. Diagrama de clases donde se enfatiza la relación de todas las entidades descritas.

2.3 Segunda iteración

El objetivo de la segunda iteración es construir una simulación básica en CPU, de tal manera que, obviando los requisitos no funcionales propuestos, y más concretamente las restricciones temporales, esta podría ser una solución aceptable (aunque inacabada). Por tanto, primero se describirá y justificará el algoritmo seleccionado, para después mostrar los resultados obtenidos.

2.3.1 Desarrollo de solución

Para elaborar la solución se parte de una escena estática como la que se representa en la Ilustración 54. En este punto se conoce que el dispositivo que se simula es un LiDAR, y por tanto, el problema se puede enfocar como un lanzamiento de rayos. Es decir, el problema se convierte en una elección correcta de estructuras de datos y algoritmos.

Ilustración 54. Escena estática inicial.

En primer lugar, es necesario plantear qué estructura de datos debe emplearse para recorrer la escena de la manera más eficiente posible. Parece evidente que debe ser una estructural espacial 3D, y en dicho caso, podemos encontrar algunos ejemplos, como mallas regulares, octrees, k-d trees o un BVH (*Boundary Hierarchy*

Volume; lo describiremos más tarde). A la hora de evaluar qué estructura de datos emplear no sólo se debe considerar el tiempo de respuesta que ofrece bajo una consulta, sino también la capacidad de recorrer dicha estructura, así como el nivel de adaptación y flexibilidad de la misma.

En el caso de la **malla regular**, se concibe como una matriz cuyo número de celdas se indica a partir de tres valores estáticos: divisiones en X, en Y y en Z. El tiempo de construcción es muy bajo, y el algoritmo para recorrer la estructura es quizás el más sencillo de todas las estructuras planteadas. A diferencia de la malla, el ***octree*** es una estructura adaptativa que depende sólo de algunos parámetros que controlan la complejidad de la estructura, como es el máximo nivel de profundidad o el máximo número de elementos por nodo para subdividir. Comienza con un nodo raíz, que no es más que un cubo que define los límites de la escena, y subdivide dicho cubo (en otros ocho) tantas veces como lo necesite, con el fin de cumplir la restricción de primitivas almacenadas en un mismo nodo (a menos que alcance el máximo nivel indicado) (Ilustración 55). El algoritmo que permite recorrer esta estructura no es trivial, y además, presenta un problema que no se da en otras estructuras. Los nodos de este árbol no son más que cubos, y cuando estos se subdividen no se garantiza que todos los nodos contengan información, sino sólo al menos uno de ellos. Es decir, recorrer un *octree* implica que en el camino encontraremos nodos vacíos que extenderán este proceso en el tiempo.

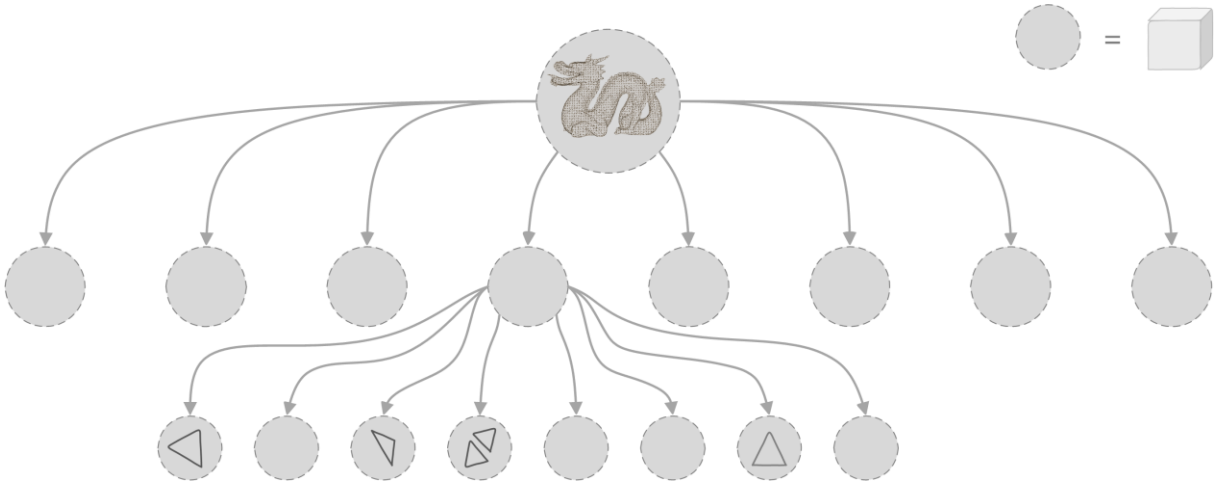


Ilustración 55. Representación de un octree que almacena los triángulos de un modelo. Algunos de los nodos hoja se encuentran vacíos.

Por otro lado, tenemos otras dos estructuras que pueden (y suelen) emplearse en *ray-tracing*, donde el tiempo de resolución de una intersección es vital, y por tanto, el recorrido de la estructura también lo es. Dichas estructuras son los *k-d trees* y el

BVH (Akenine-Möller 2018). El punto de partida de un *k-d tree* es el mismo que el de un *octree*; partimos de un cubo que cubre la escena entera (*axis-aligned bounding box*) y se subdivide, con la diferencia de que no se particiona el cubo en otros cuatro fragmentos, sino en dos mediante un plano paralelo a alguno de los ejes, X, Y o Z (primero se divide en X, después en Y, y por último, en Z, y vuelve a repetirse el ciclo) (Ilustración 56). Por tanto, esta estructura se adapta mejor a los objetos contenidos en la escena, mientras que un *octree* se adapta especialmente bien un espacio cúbico en base a la densidad de primitivas. El problema que se representa en estas últimas líneas es muy simple: podemos considerar una primitiva justo en el centro de la escena, de tal manera que cuando se divida el nodo raíz, dicho objeto formará parte de cuatro nodos descendientes diferentes. Es decir, un *octree* ignora por completo la forma de la primitiva contenida, ya sea una malla o un triángulo. En una nube de puntos, este problema no ocurre dado que no hay posibilidad de que un punto pertenezca a varios nodos (excepto en el caso extremo de situarse justamente en la frontera de dos nodos). Esto que aquí se expone no implica que todas las primitivas de un *k-d tree* pertenezcan a un único nodo; es cierto que puede haber planos separadores que corten una primitiva, pero al menos la estructura se adapta mejor a los objetos contenidos, y en cualquiera de los casos, la subdivisión se realiza en base a una heurística que puede personalizarse. De hecho, también existen versiones que no alinean los planos divisores con los ejes X, Y o Z, sino con los polígonos, aunque no es frecuente su uso hoy en día. Por ejemplo, esta versión alternativa se utilizaba en el videojuego Doom como una alternativa a un *Z-buffer* por hardware, en ese momento inexistente. Una última ventaja del *k-d tree*, como se mencionaba en el párrafo anterior, es que la división en base a primitivas permite que el recorrido de la estructura no contenga nodos vacíos, minimizando así el número de pasos de este recorrido (aún así, todo esto depende de la heurística; una mala división podría generar nodos vacíos).

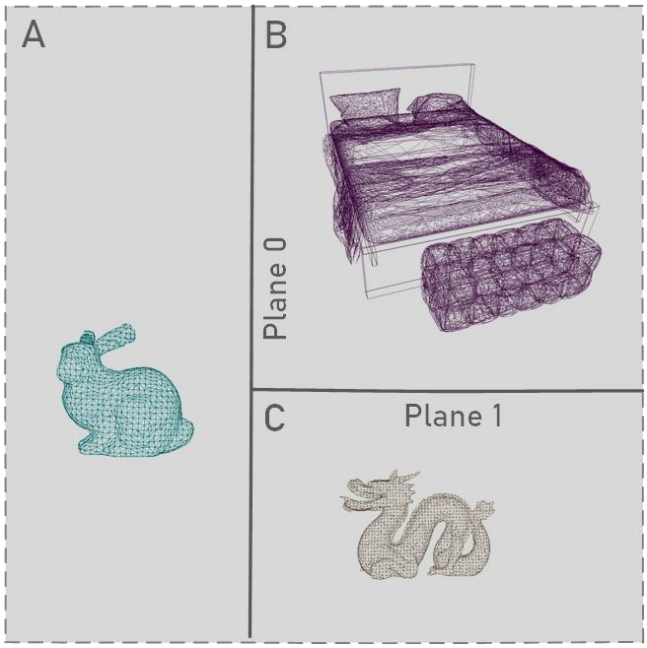


Ilustración 56. Representación de la división de la escena mediante segmentos en un entorno 2D.

Otra estructura que se tratará más tarde, y que sí contempla una primitiva como tal, es el BVH o jerarquía de volúmenes envolventes (Ilustración 57). De nuevo, se parte de un nodo raíz que no es más que un AABB de la escena completa. La manera más fácil de imaginar esta estructura es considerar un enfoque de construcción *down-top*, es decir, se parte de las primitivas y comenzamos a unir nodos hasta alcanzar la raíz. Los nodos de esta estructura de datos se definen como AABBs de un conjunto de primitivas (en el caso de un nodo hoja, el conjunto tendrá tamaño uno). A diferencia de las estructuras de datos anteriores, aquí una primitiva no puede estar contenida en varios nodos de un mismo nivel. De nuevo, la forma en la que se asocian los nodos viene definida por una heurística, que en última instancia repercute en la calidad de la estructura. El recorrido por esta estructura parte del nodo raíz, y comprueba la intersección de un rayo con los descendientes, permitiendo así descartar nodos (en cada paso, la mitad restante) (Karras, Thinking Parallel, Part II: Tree Traversal on the GPU, 2012). Esta misma solución también podría implementarse en un *octree* o un *k-d tree*, pero hay soluciones más eficientes que permiten comenzar en un nodo raíz y hallar vecinos siguiendo la dirección del rayo, así hasta salir de la estructura. Un BVH es una estructura habitual en aplicaciones de *ray-tracing*, por lo que existen artículos más recientes acerca de optimizaciones (en GPU) que podrían reducir el número de nodos visitados (Hendrich et al. 2019), generalmente clasificando rayos y asignando subárboles donde iniciar la búsqueda.

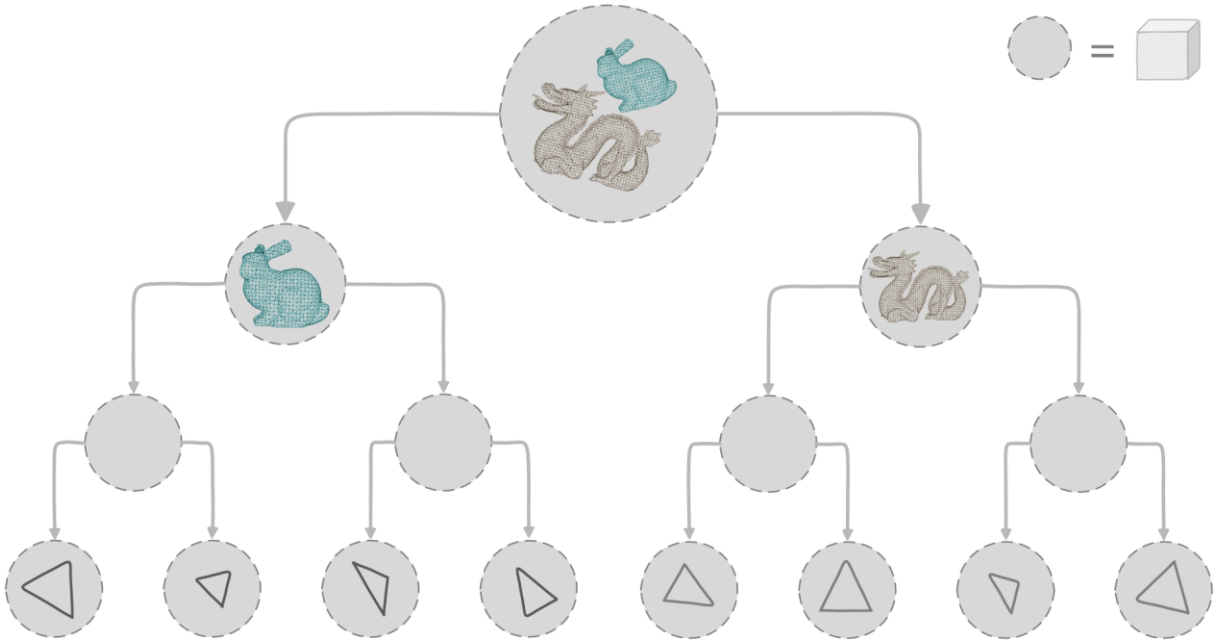


Ilustración 57. Representación de la estructura de un BVH.

En definitiva, en base a la descripción realizada podemos seleccionar una estructura de datos candidata para nuestra primera solución. Aunque es cierto que un BVH es la estructura que parte con más ventaja, también se han considerando los conocimientos previos con otras estructuras, así como la memoria que pueden necesitar o la adaptación a un escenario como el propuesto anteriormente. Las dos principales estructuras candidatas son el BVH, dada la calidad de la estructura y el tiempo de respuesta, y el *octree*, por el conocimiento previo de esta estructura.

En base a las restricciones funcionales de uso de memoria, así como a la experiencia, se selecciona finalmente el *octree*. Nótese como dicha estructura limita su profundidad mediante un valor indicado inicialmente, y por tanto, también limita la memoria utilizada. Además, partimos de un escenario que ya contiene de por sí algunos millones de vértices y triángulos. Por otro lado, el escenario propuesto no presenta grandes desequilibrios de densidad de primitivas en diferentes puntos, por lo que se considera que no debe existir una gran distancia entre los resultados de ambas estructuras (en cuanto a tiempo de respuesta).

Con la perspectiva actual, donde se documenta esta iteración habiendo finalizado el proyecto, puede catalogarse esta elección como incorrecta, pero sin duda, nos servirá para comparar resultados y establecer una base con la que valorar las mejoras halladas.

2.3.1.1 Implementación de estructuras de datos

Considerando que la estructura seleccionada es finalmente un *octree*, se dedica este apartado a su implementación. Dicha estructura recibe el nombre de ***OctreeTriangle***, dado que la escena presentada se compone únicamente de mallas de triángulos, y por tanto, la primitiva es un triángulo. Nótese como esta primitiva podrá pertenecer a más de un nodo, y más concretamente, puede pertenecer hasta a cuatro (tres vértices en un nodo, más un nodo intersectado por el triángulo).

Para comenzar este apartado se debe tratar la estructura de un nodo, *OctreeNode*, el cual se define por los siguientes atributos:

- Un AABB que define la frontera del triángulo.
- Cuatro punteros a las instancias de los nodos descendientes, también pertenecientes a la clase *OctreeNode*, pudiendo ser estos nulos cuando el nodo aún no ha alcanzado la máxima capacidad indicada (siempre y cuando no sea el último nivel).
- Un valor booleano que indica si se han creado instancias descendientes. Nótese como el vector de punteros a nodos descendientes se encuentra inicializado con valores nulos, y por tanto, no es válido comprobar su tamaño, sino que sería necesario iterar por el mismo para comprobar si dichos punteros son nulos o no. En función de la implementación, puede bastar con la comprobación del primer valor.
- Nivel de profundidad en el *octree*, para conocer si se ha alcanzado el límite establecido, aunque también podría calcularse este en el propio método recursivo de inserción, mediante un parámetro en la función.
- Contenido del nodo, el cual se expresa como un conjunto de punteros a triángulos, o en este caso, caras de una malla (`Face*`).

Para esta entidad, quizás los métodos más destacables sean la generación de los descendientes, donde se aplica el pseudocódigo que se muestra en la Ilustración 58 para la división del AABB del nodo, o la búsqueda de aquellos descendientes que intersectan con un triángulo, *getChildrenIndices* (para una inserción, o simplemente una búsqueda). La complejidad de este último método radica principalmente en la implementación de la intersección, por lo que por ahora sólo cabe destacar que se

devuelve un vector de índices de los descendientes donde la función recursiva debe continuar su búsqueda.

Algorithm 1 Divide un AABB N veces

```
1: size ← aabb_size / N
2:
3: for x = 0, 1, ..., N − 1 do
4:     for y = 0, 1, ..., N − 1 do
5:         for z = 0, 1, ..., N − 1 do
6:             Generate new AABB defined by min and max points
7:             min ← aabb_min + size ∗ (x, y, z)
8:             max ← aabb_max + size ∗ (x, y, z)
9:         end for
10:    end for
11: end for
```

Ilustración 58. Pseudocódigo del algoritmo de división de un AABB mediante un parámetro N. En un octree, $N = 2$.

Una vez descrito el nodo del *octree*, podemos desplazarnos a la propia estructura de datos, cuyos atributos son únicamente un nodo raíz, por el cual comienza una búsqueda, un nivel máximo de profundidad, y un valor máximo de capacidad por nodo. Estos dos últimos parámetros influyen en el tamaño mínimo de las cajas envolventes contenidas en la estructura del *octree*, debido a la subdivisión del AABB inicial a medida que se insertan primitivas (Ilustración 59).

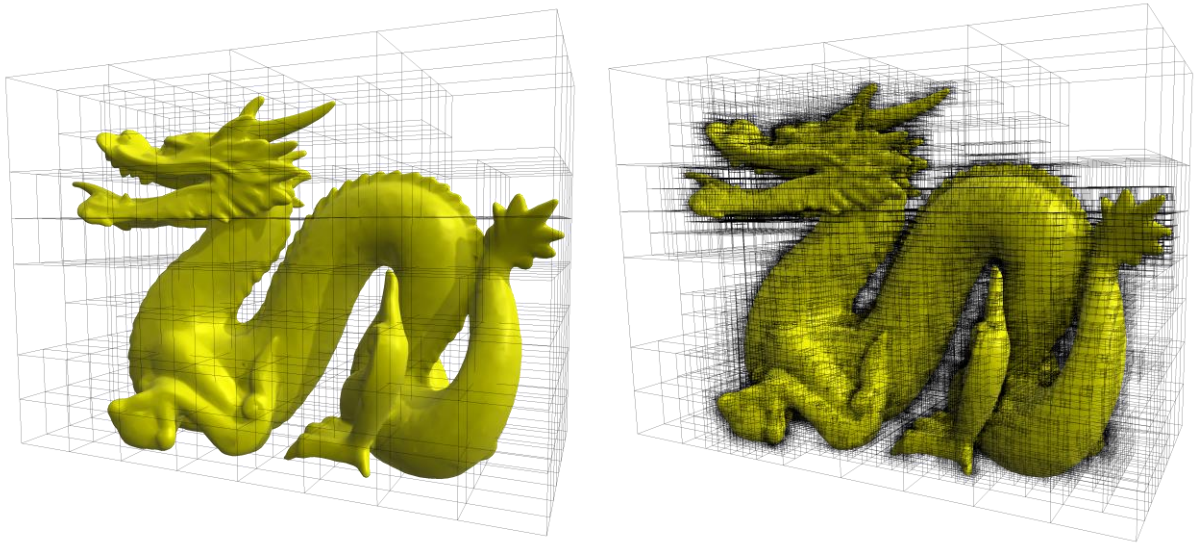


Ilustración 59. Dos octrees con diferentes valores máximos de profundidad para un mismo modelo. 1) Nivel máximo: 3, 2) nivel máximo: 8.

De esta estructura de datos nos interesa especialmente la inserción y la localización de aquellos nodos donde debe encontrarse un triángulo. Si la información

contenida se tratara de una nube de puntos, hablaríamos de un único nodo (incluso en el caso extremo, donde el punto se encuentra en una frontera), pero este no es el caso. Por tanto, en nuestras funciones recursivas debe indicarse un vector de punteros donde se almacenarán aquellos nodos encontrados.

Nótese como el proceso de inserción de un triángulo requiere, en primer lugar, de una búsqueda de nodos hoja. Así, dicho algoritmo de búsqueda se representa en la Ilustración 60, mientras que la inserción se muestra en la Ilustración 61. Cuando se alcanza el máximo nivel de capacidad, se reubican los triángulos en aquellos nodos que los intersectan (pudiendo ser más de uno por triángulo), siempre y cuando no se haya alcanzado el máximo nivel de profundidad.

Algorithm 2 *LocateNodeForInsertion*: Localización de nodos en los que debe insertarse el triángulo T

```
1: if node.isLeaf then
2:     Añadir a foundNodes el nodo actual
3:     Volver
4: end if
5:
6: childIndices ← node.getChildrenIndices(T)
7:
8: for child = 0, 1, ..., childIndices.size − 1 do
9:     Búsqueda recursiva en node.child[childIndices[child]]
10: end for
```

Ilustración 60. Pseudocódigo de la función de localización de nodos para la inserción de un triángulo.

Nótese como esta misma función es válida para comprobar la existencia de un triángulo en el *octree*, aunque en dicho escenario sólo se necesita el tamaño del vector de nodos hallados, `foundNodes`. Si se encuentra vacío, entonces el triángulo no ha sido hallado. En cualquier otro caso, habrá que comprobar si el triángulo realmente existe en alguno de los nodos hallados.

Algorithm 3 *push_back*: Inserción de un triángulo T en un octree

```
1: Inicializar vector de punteros de nodos
2: locateNodeForInsertion(octree.root, T, nodes)
3:
4: for each node ∈ nodes do
5:     node.add(T)
6:     if node.content.size > umbral ∧ node_level < maxLevel then
7:         Inicializa descendientes de node
8:         for each triangle ∈ node.content do
9:             intersNodes ← node.getChildrenIndices(triangle)
10:            for child = 0, 1, ..., intersNodes.size − 1 do
11:                Inserción recursiva
12:                insert(node.child[intersNodes[child]], triangle)
13:            end for
14:        end for
15:    end if
16: end for
```

Ilustración 61. Pseudocódigo del algoritmo de inserción de un triángulo en un octree.

El test de intersección entre un triángulo y un AABB que se efectúa en `getChildrenIndices`, con el objetivo de conocer en qué nodo deberíamos añadir el triángulo, se resuelve mediante un test de eje separador (Ericson 2004). Este test se basa en el teorema del hiperplano separador, de tal manera que si dos conjuntos convexos, A y B, no intersectan, debe existir un plano separador, P, tal que A y B quedan aislados cada uno a un lado del plano. Cuando alguno de los objetos es cóncavo, un plano no es suficiente, sino que es necesario una superficie curva, aunque este claramente no es nuestro caso (AABB y triángulo).

A partir de dicho hiperplano P, podemos extraer un eje separador L, perpendicular a P, donde la proyección ortogonal de A y B produce dos intervalos, que no se solapan cuando A y B no intersectan. Por tanto, un eje separador existe sólo si existe P, aunque comprobar si existe L es menos costoso a nivel de cómputo que determinar si existe P.

Este tipo de test no sólo se aplica a un triángulo y un AABB, sino a otras muchas primitivas; por ejemplo, cubo-plano, esfera-plano, etc. En buena parte de las ocasiones, se tratan de objetos simétricos, por lo que se parte de un punto central C, el cual es fácilmente proyectable, y se considera el radio de la forma (y a partir de aquí, se puede comprobar el solapamiento de dos intervalos proyectados en L). Partiendo de un número infinito de potenciales ejes separadores, se debe limitar el test a unas pocas comprobaciones, que en el caso de los objetos convexos se reduce

a comprobar tres situaciones de contacto: cara-cara, cara-arista (se da por incluida la situación arista-cara) y arista-arista. Nótese como en estas situaciones ya se contempla la casuística cara-vértice y arista-vértice, dado que un vértice forma parte de la arista, y por tanto, podemos evitar cálculos innecesarios.

Para las dos primeras casuísticas, cara-cara y cara-arista, se emplean sus normales para comprobar si estas pueden ser ejes separadores. En el caso de arista-arista, lo que se emplea es el producto vectorial de ambas aristas como posible eje separador. Si las dos aristas colisionaran, se formaría un vector perpendicular que surge de un punto de colisión pertenecientes a ambas.

Por tanto, la intersección de un AABB y un triángulo puede implementarse mediante este mismo test, y a partir de lo expuesto, se puede extraer que es necesario comprobar hasta 13 ejes separadores:

- Tres normales del AABB (X, Y, Z).
- Una normal del triángulo.
- Nueve vectores que surgen del producto vectorial de todas las combinaciones posibles entre tres vectores dirección de las aristas del AABB, y las aristas del triángulo.

La implementación final toma como base el algoritmo desarrollado en (Akenine-Möllser 2001) y descrito en profundidad en (Ericson 2004). Un dato muy importante a la hora de conseguir la máxima eficiencia posible de este método es el orden de los tests que se llevan a cabo. (Akenine-Möllser 2001) sugiere que el orden a seguir es 3-1-2, es decir, primero se realizan los nueve tests relacionados con la intersección arista-arista, y por último, se comprueba la normal del triángulo como posible eje separador. La gran mayoría de tests que se ejecutan son diferentes, y por tanto, la implementación define hasta seis métodos diferentes para tal fin. Existe un gran intercambio de información entre todos estos métodos y la función principal, razón por la cual se ha creado una estructura, ***TriangleAABBIntersData***, que evita indicar un gran número de parámetros en cada llamada a una función. En lugar de ello, sólo se indica la dirección en memoria de la instancia de dicha estructura.

El orden de ejecución de este test de intersección se define a continuación:

1. Cálculo del punto central del AABB y su radio, el cual no se corresponde con el máximo radio de una esfera contenida en el AABB, sino con un vector conocido como *extent*, tal que $radius = length(extent) =$

$length(aabb_{center} - aabb_{min})$. Toda esta información comienza a almacenarse en la instancia de *TriangleAABBIntersData*.

2. Cálculo de vértices v_0, v_1 y v_2 desplazados, de tal manera que el punto central del AABB se convierte en el origen del sistema, evitando así muchos cálculos en los siguientes tests. También se calculan y almacenan los tres ejes del triángulo.

3. Comienzan los nueve tests arista-arista. Tantos estos tests como todos los que se describan a continuación, se tratan de pruebas que descartan la intersección. Es decir, el objetivo es salir de este método de intersección lo antes posible, de tal manera que esta sólo se prueba como cierta cuando ningún test consiga descartar esta situación.

   En estos nueve test disponemos de tres vectores dirección del AABB y tres del triángulo. Aunque hemos hablado anteriormente de aristas, no es relevante el signo de la dirección de dichos vectores, sino que se interpretan como ejes, y por tanto, se especifica $faceEdge$ como $fabsf(faceEdge_i)$. Tras cada conjunto de tres tests, se avanza al siguiente eje del triángulo (será el nuevo centro de las comparaciones).

4. Se comprueba la intersección entre las caras del AABB y el triángulo. Esta comprobación equivale a la intersección entre dos AABBs, sea uno de ellos el mínimo AABB del triángulo, y el segundo nuestro AABB (perteneciente al *octree*). La manera más fácil de implementar esto mismo es obtener el mínimo y máximo valor del triángulo en cada eje, X, Y y Z, lo cual se puede interpretar como en una proyección en dos planos del espacio, XY e YZ. Por ejemplo, en el primer test sólo nos interesa conocer si el mínimo valor en X del triángulo es mayor que el radio (situación 1) o si el máximo valor se encuentra más alejado que el radio de la caja envolvente (situación 2). Esto mismo se repite para Y y Z.

5. La última intersección por probar es la que se produce entre el plano del triángulo y el AABB, de tal manera que la normal del triángulo podría ser un eje separador.

   Un test muy sencillo para esta situación parte de los siguientes cálculos:

$$plane_{normal} = triangle_{edge_0} \times triangle_{edge_1}$$
$$plane_D = plane_{normal} * triangle_{v_0}$$
$$r = aabb_{extent} * abs(plane_{normal}) \quad (2.14)$$
$$s = plane_{normal} * aabb_{center} - plane_D = -plane_D$$

De esta manera, se determina que un AABB y un plano intersectan cuando $abs(s) \leq r$, sea s la distancia del centro del AABB al plano, y r el valor t de la ecuación $L(t) = aabb.center + t * plane.normal$. Si $plane_{normal}$ se encuentra normalizada, entonces t es la distancia del centro del AABB al punto con el que intersecta la normal del plano. Por tanto, AABB y triángulo intersectan si $s \in [-r, r]$.

De esta manera, se resuelve qué nodos podrían incluir un triángulo. Si el nodo intersectado es un nodo hoja, se insertará directamente (aunque haya que reubicar seguidamente), si no, será necesario descender en la jerarquía.

2.3.1.2 Estructuración de la escena

La escena que se muestra al comienzo de esta iteración no es más que un conjunto de triángulos, por lo que conocida la forma de crear un *octree* compuesto por estas primitivas, sería posible dar paso a la resolución del problema de *ray-casting* desde una posición que representa la localización del dispositivo LiDAR.

Aunque es cierto que es posible crear sólo un *octree* con todos los triángulos de la escena, lo que se plantea inicialmente es muy diferente, especialmente considerando la escena propuesta, donde se dispone de millones de triángulos y muy pocos elementos (mallas). Lo que pretende aclarar esta última línea es que el enfoque que a continuación se describe no sería tan eficiente para otras escenas que se han desarrollado más tarde, donde también se presentan millones de triángulos, pero coexisten muchos elementos juntos. En cualquier caso, las escenas con estas características se incluyen más tarde bajo un enfoque completamente diferente.

La estructuración de la escena que se plantea inicialmente es la siguiente:

1. En lugar de que cada rayo del proceso de simulación deba recorrer un *octree* de una profundidad considerable, se propone tener en cuenta primero algún tipo de forma envolvente para cada una de las figuras que se representan en la escena (modelos individuales: suelo, pared, mesa, etc).

A partir de estas formas simples se puede determinar si un rayo impacta sobre las mismas, y en dicho caso, se podrá seguir continuar con el paso 2.

Quedaría por determinar qué tipo de forma envolvente se emplea, para lo cual hay que considerar que uno de los principales factores que se considera en la aplicación es el tiempo de respuesta, tanto en la generación de la envolvente como en la resolución de su impacto con un rayo. Así, dos formas envolventes completamente opuestas en ambos factores son la envolvente convexa y el AABB (Ilustración 62). La primera necesita de un tiempo de generación relativamente elevado en función de la complejidad de la malla, e igualmente, la resolución de su intersección con un rayo depende de un test malla-rayo, volviendo al problema inicial (rayo frente a una malla que podría estructurarse en un *octree*, aunque con muchos menos triángulos). La segunda estructura, un AABB, es posiblemente la más rápida de construir y la que nos permite conocer más rápidamente si un rayo la intersecta.

Por las razones expuestas se opta por un AABB como estructura envolvente de las figuras de la escena.

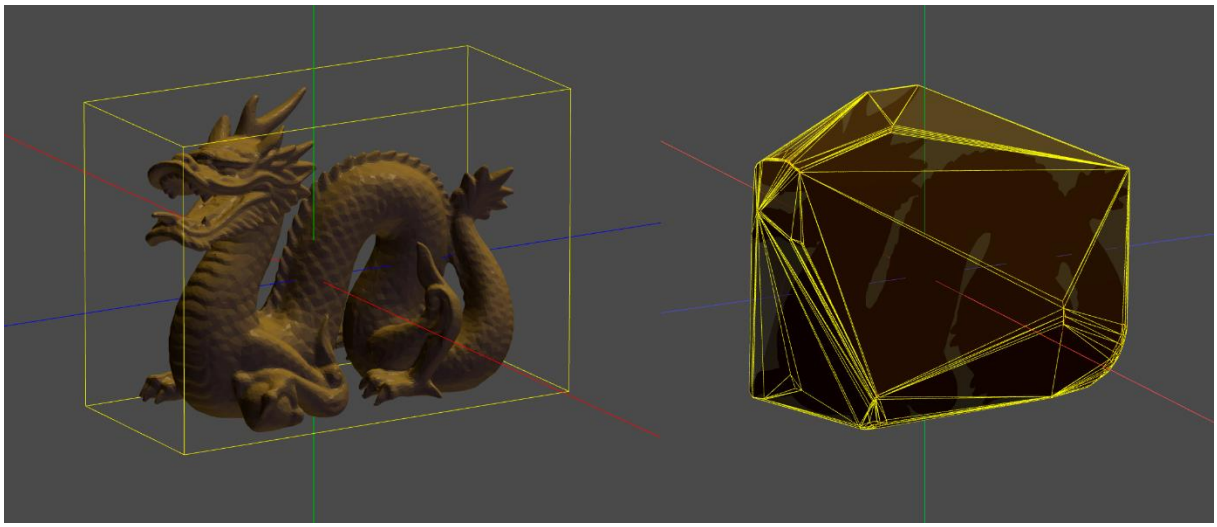


Ilustración 62. Dos estructuras envolventes de un modelo. 1) Axis-aligned bounding-box, 2) envolvente convexa (construida mediante el algoritmo de Gift wrapping).

2. De manera análoga al *octree* antes expuesto, se desarrolla un *octree* cuya primitiva contenida es un AABB, y no un triángulo, planteándose los mismos problemas anteriores: la primitiva puede intersectar con múltiples nodos, y además, habrá que desarrollar un test de intersección entre dos AABBs. En

cualquier caso, nos puede ayudar a enfocar el problema de *ray-casting* en figuras muy concretas, en lugar de la escena completa, aunque es necesario destacar que pueden ser múltiples los modelos intersectados por un rayo. En un enfoque básico de *ray-casting*, nos basta con la primera intersección, que de hecho es conocida cuando se recorre el *octree* mediante el algoritmo que veremos más tarde, pero el dispositivo LiDAR actúa en algunas superficies, como en la vegetación, generando múltiples retornos que no necesariamente proceden de refracciones ni reflejos de la emisión original (no cambia la dirección, y por tanto, sólo es necesario continuar recorriendo el *octree* para capturar más modelos intersectados).

Además de todo lo ya planteado, se añade un problema adicional, y es que un AABB es únicamente una aproximación. Es decir, un rayo podría atravesar el AABB y no intersectar con la malla. No produce falsos negativos, sino falsos positivos. Una de las ventajas de la envolvente convexa es que minimiza el número de falsos positivos al ajustarse mejor al modelo (*better culling*). Por esta misma razón, no es suficiente con extraer únicamente el primer modelo intersectado.

3. Una vez conocemos qué figuras pueden ser intersectadas, es necesario recurrir al *octree* de cada figura, el cual almacena triángulos, y por tanto, se corresponde a la descripción de la sección anterior. De este *octree* sí se obtendrán intersecciones en forma de puntos vinculados a un modelo y a un triángulo de dicho modelo. El mismo orden en que se descubren las intersecciones es el mismo orden en que estas se producen, por lo que no es necesario ningún proceso adicional de ordenación, y esto mismo se aplica al descubrimiento de modelos en el *octree* de AABBs.

Los *octrees* descritos para AABBs y triángulos se muestran en la Ilustración 63, empleando como máximo nivel de profundidad 5 y 6 respectivamente.

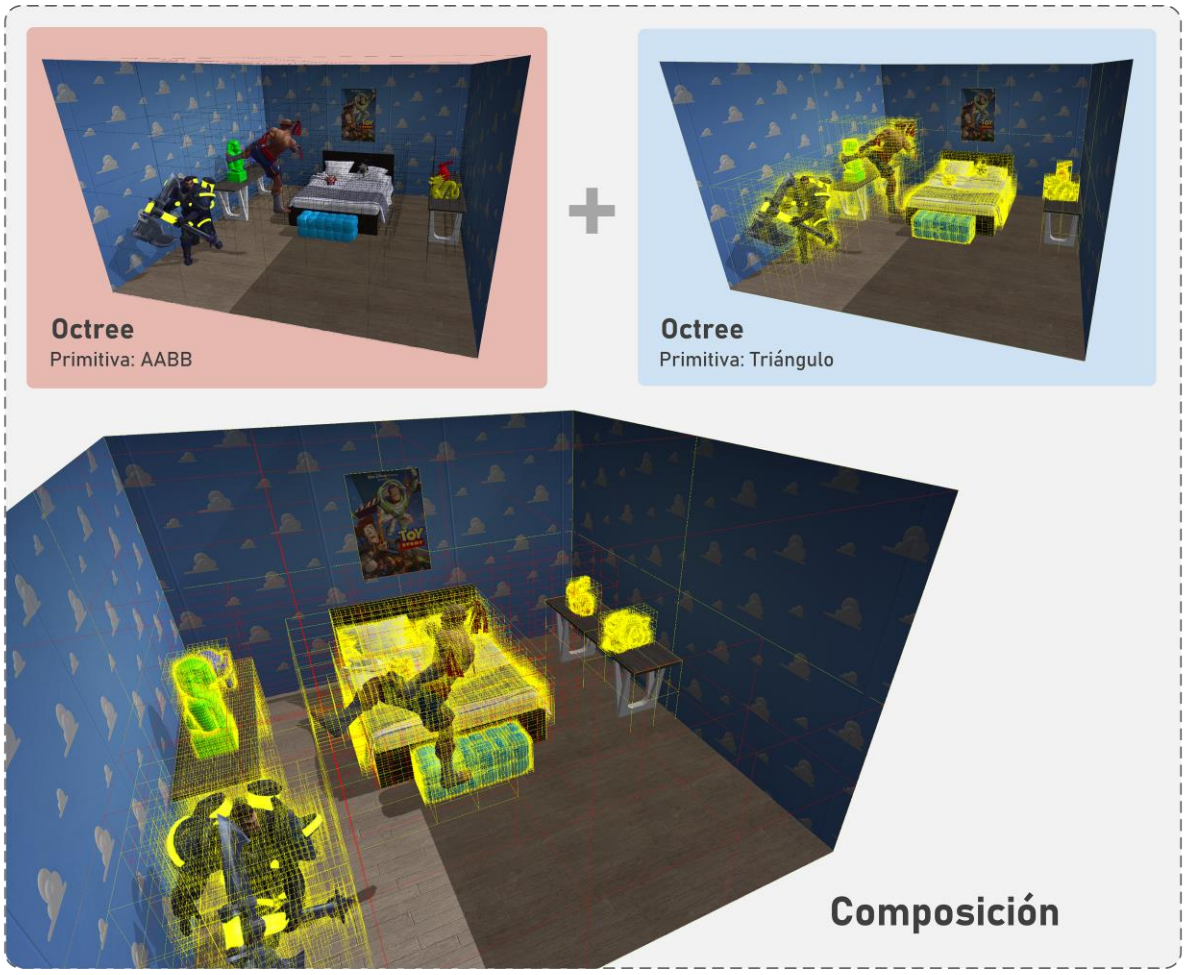


Ilustración 63. Representación de dos octrees que contienen cajas envolventes y triángulos de la escena originalmente propuesta.

A continuación se describe el algoritmo de intersección de dos AABB, dado que este debe sustituir el test previo de intersección entre un AABB y un triángulo. En cuanto al resto del *octree*, su implementación permanece invariable respecto de la versión presentada anteriormente, con la diferencia de que el contenido de un nodo serán un conjunto de cajas envolventes, y no de triángulos.

El algoritmo que permite conocer si dos AABB intersectan se describe igualmente en (Ericson 2004):

$$
\begin{aligned}
&a_{center} = aabb_{1_{center}}; \quad a_{extent} = aabb_{1_{extent}} \\
&b_{center} = aabb_{2_{center}}; \quad b_{extent} = aabb_{2_{extent}} \\
&bool3\ xyz = abs(a_{center} - b_{center}) \leq (a_{extent} + b_{extent}) \\
&bool\ intersection = xyz.x\ \&\ xyz.y\ \&\ xyz.z
\end{aligned}
\tag{2.15}
$$

En cualquier otro caso, una alternativa aceptable es la introducción de un único *octree* de triángulos que incluya la escena completa, simplificando así el proceso. El resultado de esta otra opción deriva en una imagen como la que se muestra en la Ilustración 64.

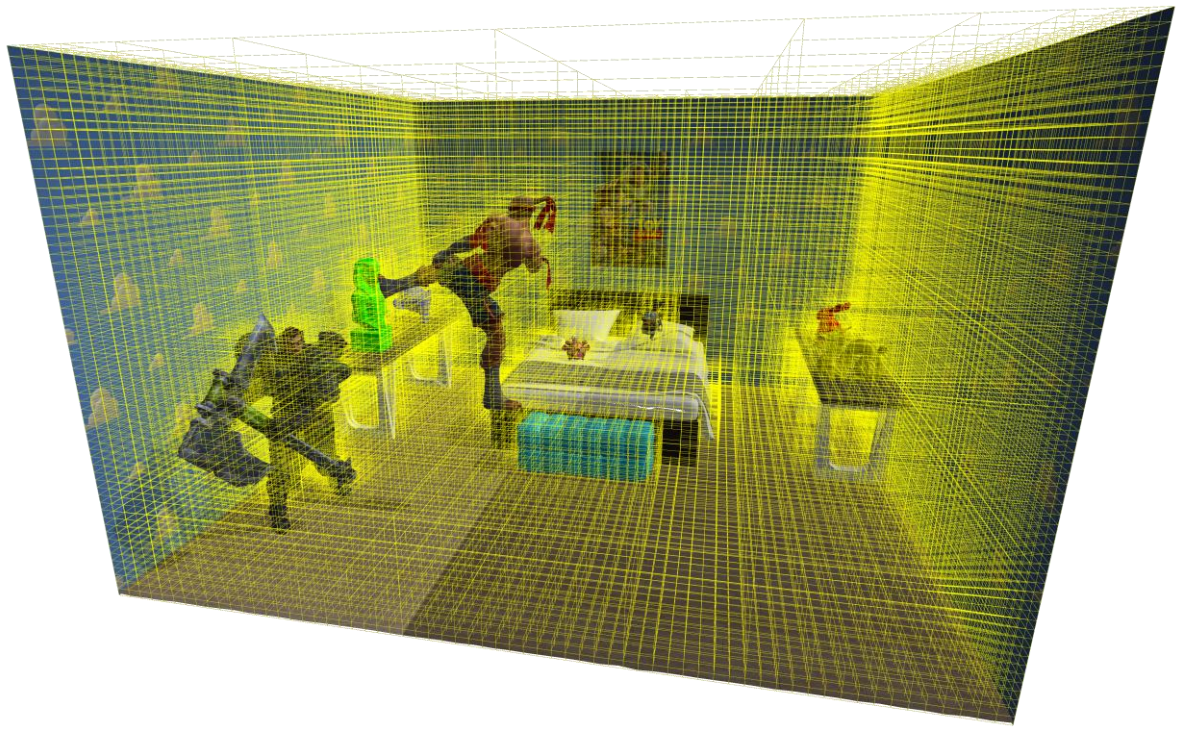


Ilustración 64. Representación de un octree de triángulos sobre la escena originalmente propuesta.

Más allá de las justificaciones expuestas y de cualquier otra suposición, podemos comprobar pragmáticamente qué solución ofrece un mejor rendimiento: un único *octree* que almacene triángulos, o un escenario donde se introduce un filtrado de modelos candidatos, e igualmente, un *octree* compuesto por triángulos. Para ello se ejecutarán un conjunto de pruebas donde sólo se medirá el tiempo de respuesta de la recuperación de triángulos candidatos, lo que incluye en el mismo proceso la generación de rayos. Es decir, se medirá el tiempo de respuesta de un proceso puramente iterativo, que también incluye el algoritmo de *octree traversal* que más tarde se describe (apartado Octree traversal). Para ello, se lanzarán hasta 1.000.000 rayos que cubrirán el espacio completo (*offset* equivalente a cero).

Las pruebas realizadas para cada configuración se muestran en columnas diferentes, de manera que habrá que observar el tiempo medio para comprobar qué versión proporciona mejores resultados (Tabla 18).

Antes de mostrar las pruebas es necesario destacar que en su ejecución, y en la de cualquier otra prueba posterior, se ha empleado el ordenador que se describe en el presupuesto hardware de este mismo documento (apartado Presupuesto).

Tiempo medio de obtención de triángulos candidatos (1.000.000 de rayos)			
Escenario 1. Dos octrees		**Escenario 2. Un octree**	
Profundidad de octree: AABB (5), triángulo (8) **Máx. tamaño de nodo**: AABB (3), triángulo (10)		**Profundidad de octree**: triángulo (10) **Máx. tamaño de nodo**: triángulo (10)	
Ejecución	**Tiempo de respuesta (ms)**	**Ejecución**	**Tiempo de respuesta (ms)**
1	1.452	1	2.443
2	1.445	2	2.361
3	1.386	3	2.630
4	1.463	4	2.600
5	1.434	5	2.636
Tiempo medio de respuesta (ms)	1.436	**Tiempo medio de respuesta (ms)**	2.534
Profundidad de octree: AABB (6), triángulo (10) **Máx. tamaño de nodo**: AABB (1), triángulo (10)		**Profundidad de octree**: triángulo (14) **Máx. tamaño de nodo**: triángulo (10)	
Ejecución	**Tiempo de respuesta (ms)**	**Ejecución**	**Tiempo de respuesta (ms)**
1	1.494	1	2.227
2	1.687	2	2.584
3	1.584	3	2.174
4	1.554	4	2.077
4	1.543	4	2.345
Tiempo medio de respuesta (ms)	1.572,4	**Tiempo medio de respuesta (ms)**	2.281,4
Profundidad de octree: AABB (3), triángulo (8) **Máx. tamaño de nodo**: AABB (5), triángulo (10)		**Profundidad de octree**: triángulo (16) **Máx. tamaño de nodo**: triángulo (10)	
Ejecución	**Tiempo de respuesta (ms)**	**Ejecución**	**Tiempo de respuesta (ms)**
1	1.884	1	2.409
2	1.722	2	2.690
1	1.613	1	2.416
2	1.587	2	2.194

3	1.653		3	2.597	
Tiempo medio de respuesta (ms)	1.691,8		**Tiempo medio de respuesta (ms)**	2.461,2	

Tabla 18. Tiempo medio de respuesta de la recuperación de todos los triángulos que podrían ser intersectados para cada uno de los rayos lanzados.

Cualquiera de las pruebas ejecutadas para el primer escenario obtienen un menor tiempo medio de ejecución respecto del resto de pruebas propuestas en el segundo escenario. Por tanto, los resultados obtenidos se ajustan a lo esperado, dado que se trata de un escenario muy complejo para un único *octree* de triángulos. Con tan pocos modelos (y formas envolventes), el enfoque planteado parece el óptimo. El principal problema del segundo enfoque es que debe minimizar el número de triángulos por nodo, pero para ello se necesita un gran nivel de profundidad, y en dicho caso, recorrer un *octree* se vuelve demasiado complejo, incluso empleando un enfoque tan eficiente como el que veremos después. En definitiva, las pruebas realizadas parecen justificar el enfoque propuesto y descrito en este mismo apartado.

Para finalizar este capítulo, se muestra el tiempo medio de construcción de un *octree* que almacena todos los triángulos de la escena (Tabla 19), dado que este es necesariamente el escenario más complejo. Nótese como el objetivo de esta prueba no es mostrar el tiempo de construcción de todos los *octrees* del escenario previamente seleccionado, sino mostrar la eficiencia de la construcción de un *octree* cuando la geometría y topología de la escena es relativamente elevada. Dicha escena está compuesta por 310.363 vértices y 604.993 triángulos (sean estos últimos los que nos interesan). Para obtener el tiempo medio de respuesta en la construcción se considerarán hasta cinco ejecuciones por cada grupo de parámetros.

Tiempo medio de construcción de un octree (triángulos)	
Máxima profundidad: 8. Máximo número de triángulos por nodo: 5	
Ejecución	**Tiempo de respuesta (ms)**
1	3.273
2	3.319
3	3.240
4	3.309
5	3.216

	Tiempo medio de respuesta (ms)	3.271,4

Máxima profundidad: 8. Máximo número de triángulos por nodo: 10		
Ejecución	**Tiempo de respuesta (ms)**	
1	2.940	
2	2.821	
3	2.946	
4	2.877	
5	2.842	
	Tiempo medio de respuesta (ms)	2.885,2

Máxima profundidad: 5. Máximo número de triángulos por nodo: 10		
Ejecución	**Tiempo de respuesta (ms)**	
1	1.364	
2	1.364	
3	1.310	
4	1.300	
5	1.270	
	Tiempo medio de respuesta (ms)	1.321,6

Máxima profundidad: 10. Máximo número de triángulos por nodo: 10		
Ejecución	**Tiempo de respuesta (ms)**	
1	5.050	
2	5.143	
3	5.108	
4	5.036	
5	4.984	
	Tiempo medio de respuesta (ms)	5.064,2

Tabla 19. Tiempos de construcción de un octree en una escena de 604.993 triángulos bajo diferentes configuraciones.

En las pruebas ejecutadas se muestra como el tiempo de respuesta sigue creciendo a medida que aumenta el nivel de profundidad del *octree*, lo que implica que existe una necesidad de seguir subdiviendo, dada la abundante geometría y topología de la escena. Sin embargo, nos interesa un escenario donde el número de triángulos en los nodos hoja sea muy reducido, de tal manera que tras la ejecución del algoritmo de *octree traversal* deban comprobarse el mínimo número de triángulos. También es necesario considerar que la mayoría de superficies en la primera escena se representan como opacas (por ver algunas páginas más tarde), lo que implica que será suficiente con recuperar los triángulos del primer nodo, debido a que sólo es necesaria la primera intersección. En definitiva, se debe interpretar el máximo número de triángulos por nodo como el número de triángulos que estamos dispuestos a comprobar por cada rayo, aunque también sería deseable comprobar el tamaño medio de los nodos hoja tras construir el *octree*, con el fin de conocer si la complejidad de la escena demanda un mayor nivel de profundidad.

2.3.1.3 Octree traversal

La finalidad de este apartado es introducir la implementación del recorrido de un *octree* partiendo de un rayo. Una versión básica comenzaría por la raíz, y visitaría recursivamente todos aquellos nodos que intersecten con el rayo, de tal manera que en los nodos hoja se encuentran aquellas primitivas susceptibles de ser intersectadas. Sin embargo, esta solución presenta al menos dos problemas. En primer lugar, el algoritmo por sí mismo no es eficiente, especialmente cuando el *octree* dispone de una gran profundidad. No sólo es necesario pensar en un problema de eficiencia ligado al número de nodos visitados, sino también vinculado al gran número de tests de intersección que se ejecutan. Por tanto, es importante destacar la importancia de un algoritmo que nos permita recorrer esta estructura jerárquica de manera eficiente. Por ejemplo, un enfoque *bottom-up* nos permitiría hallar el primer nodo intersectado por el rayo, y a partir de aquí, es posible desplazarse a otros vecinos mediante un proceso de *neighbour finding*.

Uno de los algoritmos que suele emplearse para resolver este problema es (Revelles, Ureña, & Lastra, 2000), elaborado por un grupo de investigadores de la Universidad de Granada. Se trata de un método *Top-Down*, de tal manera que también comienza por la raíz del *octree*, aunque claramente es más eficiente que el enfoque naïve. Se contempla como un algoritmo paramétrico debido al uso de la formulación

paramétrica del rayo, r, definido mediante una tupla (p, d), donde $p = (p_x, p_y, p_z)$ es el origen de dicho rayo, y $d = (d_x, d_y, d_z)$ es la dirección normalizada del mismo. De esta manera, es posible calcular un nuevo punto perteneciente al rayo empleando un valor $t \geq 0$ tal que:

$$rayPoint = p + t * d \tag{2.16}$$

Así, todos los cálculos de este *traversal algorithm* emplean el valor t de la formulación parámetrica anterior. Para plantear la base del problema podemos centrarnos en un único nodo de la estructura, o, definido por dos puntos, máximo ($p_1(o)$) y mínimo ($p_0(o)$). De esta manera, un rayo r y un nodo o intersectan cuando:

$$p_{min}(o) \leq p_r(t) < p_{max}(o) \tag{2.17}$$

De esta manera, los dos valores parámetricos de la intersección se definen como sigue:

$$\begin{gathered} t_i(o,r) = \frac{p_i(o) - p_{orig}}{d} \qquad i \in \{0,1,2\} \\ \Delta t(o,r) = t_1(o,r) - t_0(o,r) = \frac{p_1(o) - p_0(o)}{d} = \frac{s(o)}{d} \end{gathered} \tag{2.18}$$

Nótese como al considerar las coordenadas del vector $\Delta t(o,r)$ de manera individual es posible simplificar aún más la última fórmula, dado que $p_{1_x}(o) - p_{0_x}(o) = p_{1_y}(o) - p_{0_y}(o) = p_{1_z}(o) - p_{0_z}(o) = s(o)$, sea $s(o)$ la longitud de cualquier arista del cubo o. Esta relación se mantiene en los descendientes de dicho nodo, de tal manera que para el nivel inmediatamente inferior se obtiene que $\Delta t(o_i, r) = \frac{\Delta t(o,r)}{2}$. Dicho de otra forma, algunos de estos cálculos sólo se deben dar en un nodo raíz.

De la misma manera, es posible establecer las siguientes ecuaciones para obtener los parámetros de un nodo descendiente a partir de un antecesor.

$$p_{min}(o_i) = p_{min}(o) + s(o_i)\Delta n_i \tag{2.19}$$

Sea n el vector posición del nodo n_i respecto de su antecesor.

$$\begin{aligned} t_0(o,r) = \frac{p_0(o) - p_{orig}}{d} = \frac{p_{min}(o) + s(o_i)\Delta n_i - p_{orig}}{d} = \\ = \frac{p_{min}(o) - p_{orig}}{d} + \frac{s(o_i)\Delta n_i}{d} = t_0(o,r) + \Delta t(o_i,r)\Delta n_i \end{aligned} \tag{2.20}$$

En definitiva, a partir de la información de un nodo raíz es posible calcular los parámetros de los descendientes de manera recursiva. Por otro lado, partiendo de la fórmula del comienzo, donde se establece que $p_{min}(o) \leq p_r(t) < p_{max}(o)$ se verifica cuando existe una intersección, se determina la siguiente relación:

$$t_{min}(o,r) \leq t < t_{max}(o,r)$$

$$N\acute{o}tese\ como\ p_{min}(o) \leq p_r(t) < p_{max}(o) = p_{min}\big(t_0(o,r)\big) \leq p_r(t) < p_r\big(t_1(o,r)\big) \quad (2.21)$$

Cuando se cumple la anterior condición, todos los valores de t mayores que 0 y situados en el intervalo $[t_{min}, t_{max})$, sea $t_{min} = \max\left(t_{0_x}(o,r), t_{0_y}(o,r), t_{0_z}(o,r)\right)$ y $t_{max} = \min\left(t_{1_x}(o,r), t_{1_y}(o,r), t_{1_z}(o,r)\right)$, producen su correspondiente punto $p_r(t)$ perteneciente al nodo.

Por tanto, la implementación de este algoritmo es simple:

1. Se parte de un nodo raíz, donde se verifica que se cumple $t_{min}(o,r) < t_{max}(o,r)$. Nótese como, por ejemplo, t_{x_0} podría calcularse como $(p_{min}(o) - ray.orig)\ /\ d.x$. En nuestra implementación, se ha modificado $t_{min}(o,r) < t_{max}(o,r)$ para convertirlo en $t_{min}(o,r) \leq t_{max}(o,r)$, con el fin de dar soporte a casos muy extremos, pero observados en la aplicación.

2. Si no se verifica la condición, el rayo no intersecta con el *octree*. En otro caso, comienza un proceso recursivo que necesita los valores $t_0(o,r)$ y $t_1(o,r)$. En este proceso deberán subdividirse los nodos, y por tanto, también será de utilidad el cálculo del valor t intermedio, t_m, el cual se puede obtener como si se tratara de un punto intermedio: $\frac{(t_0+t_1)}{2}$ (es necesario recordar que $\Delta t(o_i,r) = \frac{\Delta t(o,r)}{2}$).

3. En cada uno de los niveles el objetivo es obtener, como primer paso, el nodo que antes es intersectado (entrada), y tras este, el resto de nodos intersectados que comparten un mismo antecesor. Cuando se recorran todos estos nodos, el algoritmo simplemente finalizará con el nodo padre actual.

Sobre la base ya desarrollada es necesario añadir algunos conceptos más, los cuales se pueden representar como escenarios muy acotados que deben

implementarse mediante simples estructuras condicionales. Por ejemplo, se puede representar mediante una tabla qué condiciones deben darse para seleccionar un nodo u otro como punto de partida. Al tratarse de conclusiones obtenidas mediante la observación, no se profundiza en aquellos puntos que aún nos quedan por tratar. Si se desea obtener más información se recomienda acudir al artículo en el que se basa esta descripción, aunque la nomenclatura puede variar al explicar sus conceptos bajo la estructura de un *quadtree*.

Se reconocen al menos dos situaciones donde el flujo del algoritmo queda definido por un conjunto de escenarios predefinidos:

- Búsqueda del primer nodo intersectado. A partir del máximo valor de t_0 (X, Y o Z) es posible determinar cuál es el plano de entrada (Tabla 20). Tomando este plano en consideración, se acotan las condiciones que deben examinarse para obtener el nodo de entrada, el cual vendrá dado por un identificador que se obtiene modificando ciertos bits de un valor inicial 0 (Tabla 21).

  Para conocer a qué nodo se corresponde el identificador es necesario considerar la Ilustración 65, donde se puede observar como el eje X se representa en el bit más significativo, y justo lo contrario sucede para el eje Z.

Plano de entrada	
Máximo valor de t_0	**Plano**
t_{0_x}	YZ
t_{0_y}	XZ
t_{0_z}	XY

Tabla 20. Plano de entrada para cada máximo valor de t_0.

Selección de nodo inicial		
Plano de entrada	**Condiciones**	**Bit afectado**
XY	$t_{m_x}(o) < t_{o_z}(o)$	0
	$t_{m_y}(o) < t_{o_z}(o)$	1
XZ	$t_{m_x}(o) < t_{o_y}(o)$	0
	$t_{m_z}(o) < t_{o_y}(o)$	2
YZ	$t_{m_y}(o) < t_{o_x}(o)$	1
	$t_{m_z}(o) < t_{o_x}(o)$	2

Tabla 21. Condiciones de selección del nodo inicial.

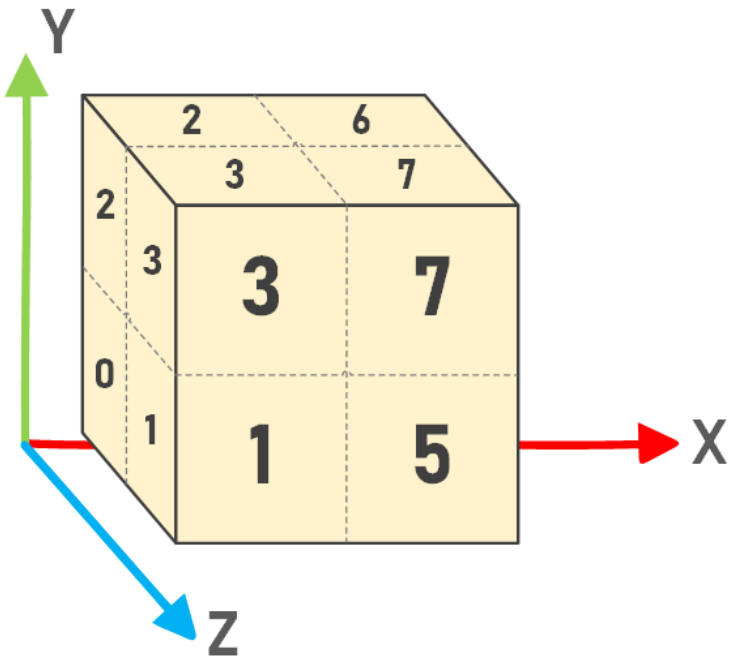


Ilustración 65. Identificadores de subnodos. Desplazamiento de 2^2 en X, 2^1 en Y, y 2^0 en Z.

- A partir del nodo inicial, es necesario hallar el siguiente nodo que debe visitarse. Para ello podemos utilizar la Tabla 22, donde el plano de salida vendrá dado por t_{max}, que bien como sabemos, se corresponde al mínimo valor de $t_{1_x}(o,r), t_{1_y}(o,r)$ y $t_{1_z}(o,r)$. Como disponemos de ocho escenarios diferentes, se filtra el nodo actual mediante una estructura condicional *switch*, en la cual, además de explorar dicho nodo, debe hallarse el siguiente nodo que debe recorrerse. Para ello, se emplea una función que en el artículo se define como `new_node`, donde además de introducirse los

valores t_{m_x}, t_{m_y} y t_{m_z}, también se indica cuál es el siguiente nodo que debería visitarse en X, Y y Z. De esta forma, cuando se escoja cierto plano de salida, se puede devolver el identificador del nuevo vecino que debe visitarse. Nótese como en la Tabla 22 se indican algunos valores nulos que hacen referencia a la finalización de la búsqueda de nodos vecinos en un mismo nodo padre. Con este objetivo, lo que se hace es indicar un identificador de valor 8 (los nodos se numeran de 0 a 7), de tal manera que el bucle sólo continúa cuando el identificador del vecino es menor que 8.

Transición hacia el siguiente nodo			
Nodo actual	**Plano de salida: YZ**	**Plano de salida: XZ**	**Plano de salida: XY**
0	4	2	1
1	5	3	-
2	6	-	3
3	7	-	-
4	-	6	5
5	-	7	-
6	-	-	7
7	-	-	-

Tabla 22. Siguiente nodo que debe recorrerse cuando se cumplen las condiciones expuestas (nodo actual y plano de salida).

Las dos cuestiones que quedan por resolver son el control de rayos paralelos a uno de los ejes X, Y o Z, y la generalización para rayos de componente negativa. La primera de las cuestiones posee una solución sencilla en nuestra implementación, dado que para evitar los valores infinitos es suficiente con modificar aquellos componentes de valor cero en el vector dirección, d, añadiendo un valor ε muy cercano al cero. De esta manera, el rayo no es completamente paralelo y no produce valores infinitos.

El segundo problema presenta una solución más compleja, pero igualmente, es posible resolverlo de manera eficiente. El algoritmo de *traversal* que aquí se describe sólo funciona correctamente cuando la dirección del rayo es positiva, por lo que una alternativa sería generar una nueva tabla para los 8 casos antes expuestos y para cada sub-nodo, lo cual es necesariamente ineficiente y además genera una gran cantidad de código innecesario. En lugar de esto, es posible reflejar un rayo de

dirección d y punto de origen p, de tal manera que si una componente es negativa se propone hacer lo siguiente:

$$d_i' = -d_i$$
$$p_i' = s(octree)_i - p_i \tag{2.22}$$

Sea s el tamaño del octree (raíz) en algún eje. Sin embargo, la formulación que propone el artículo para p_i' es incorrecta, a menos que el mínimo punto del AABB de la escena se encuentre en el origen de coordenadas. Lo que debe proponerse es reflejar el rayo respecto de las fronteras del AABB de la escena, y esta formulación no es suficiente para un AABB situado en cualquier punto del espacio. Es decir, el problema se halla en la definición de p_i', dado que que d_i' no representa una posición específica en el espacio. La formulación que verdaderamente solventa este problema es la siguiente:

$$p_i' = aabb_{center_i} - \left(p_i - aabb_{center_i}\right) = 2aabb_{center_i} - p_i \tag{2.23}$$

Es evidente que esta transformación no es una solución *per se*, sino un primer paso para que el algoritmo funcione correctamente. Nótese como la selección de nodos sería aún incorrecta, dado que no se ajusta a r, sino a r'. Para ello podríamos establecer una función f que devuelva para cada nodo su imagen. Es decir, cuando el nodo actual es i no se accede a este, sino a $f(i)$. Se da la situación de que la función que corrige este comportamiento se define como:

$$f(i) = i \oplus a = i \oplus (4s_x + 2s_y + s_z) \tag{2.24}$$

Donde s_i será 1 cuando la componente d_i del rayo sea menor que 0. En cualquiera de los casos, siempre hemos necesitado tres estructuras condicionales para comprobar d_x, d_y y d_z, por lo que simplemente se aprovechan estos bloques para definir el valor de a, que en la implementación se corresponde con un tipo de dato `char` (en realidad, sólo necesitaríamos 3 bits).

En la Ilustración 66 se muestran algunas imágenes en la que se representan aquellos nodos intersectados por un rayo cualquiera, con el fin de demostrar la corrección del algoritmo implementado. Para simplificar la visualización de la ruta escogida se emplea un único *octree* de triángulos. En aquellos puntos donde existe una mayor densidad de primitivas, los nodos son de menor tamaño.

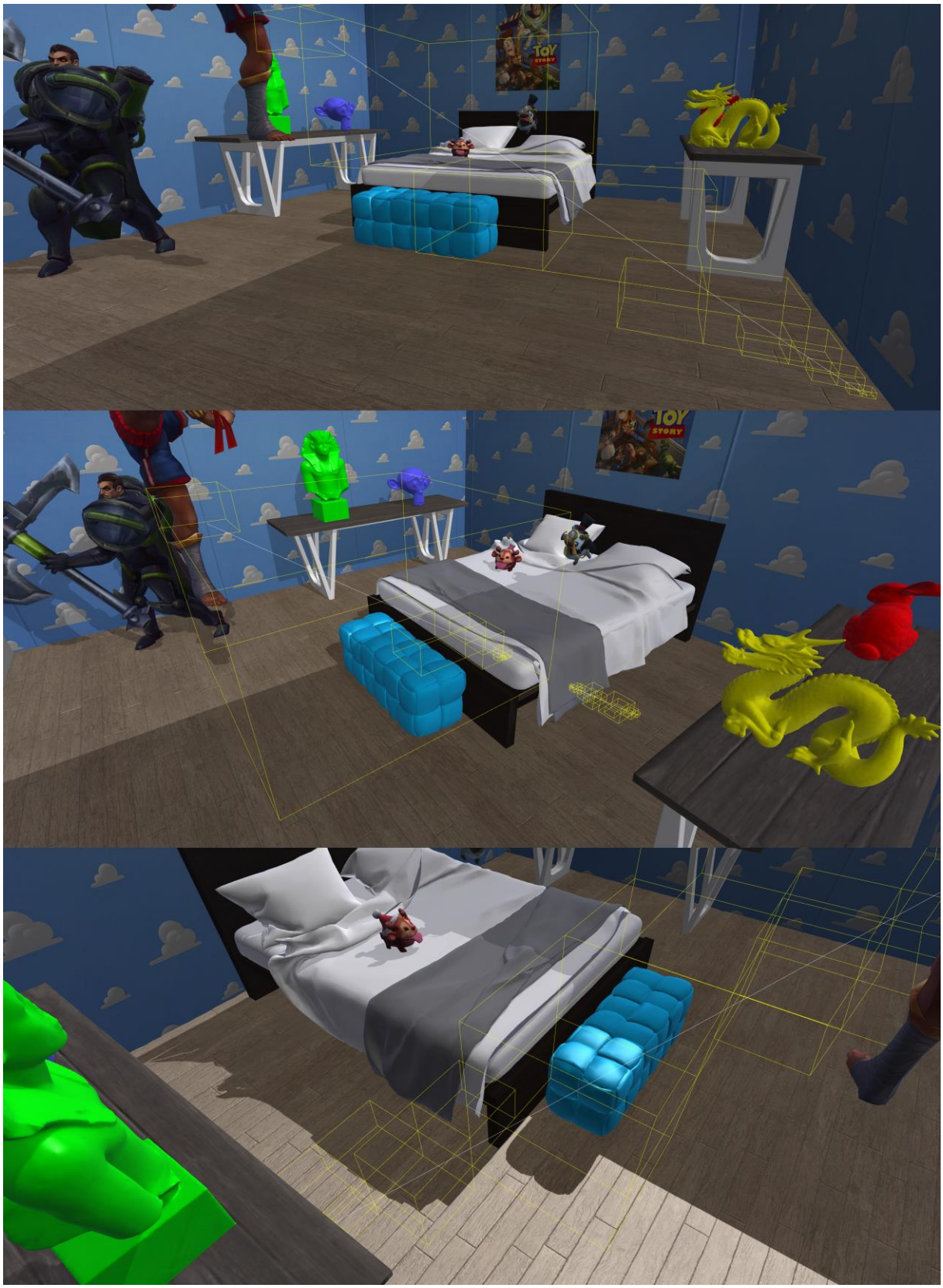


Ilustración 66. Representación de AABBs de un octree intersectados por tres rayos diferentes.

Se da así por finalizada la implementación del algoritmo de *octree traversal*, que en este punto nos permite conocer qué nodos de un *octree* podrían contener

primitivas de interés a partir de la definición de un rayo. Para completar el recorrido y la recuperación de primitivas, tanto en un *octree* que almacena AABBs como triángulos, es necesario describir algunos algoritmos de intersección que nos permiten discernir qué primitivas son realmente intersectadas por un rayo. De nuevo, un AABB no es más que un volumen aproximado que nos indica si la primitiva podría ser intersectada, pero no existe una certeza completa, incluso en el caso del *octree* que contiene AABBs, dado que estas cajas envolventes no tienen por qué coincidir exactamente con la subdivisión de nodos del *octree* (y de hecho, no lo hace).

Una vez descrito el algoritmo que se emplea para recorrer un *octree*, se muestra la descripción de todas las clases involucradas en este apartado mediante un diagrama de clases, donde se indican cuáles son los métodos encargados de resolver la construcción, así como la intersección de un rayo con estas estructuras de datos (Ilustración 67).

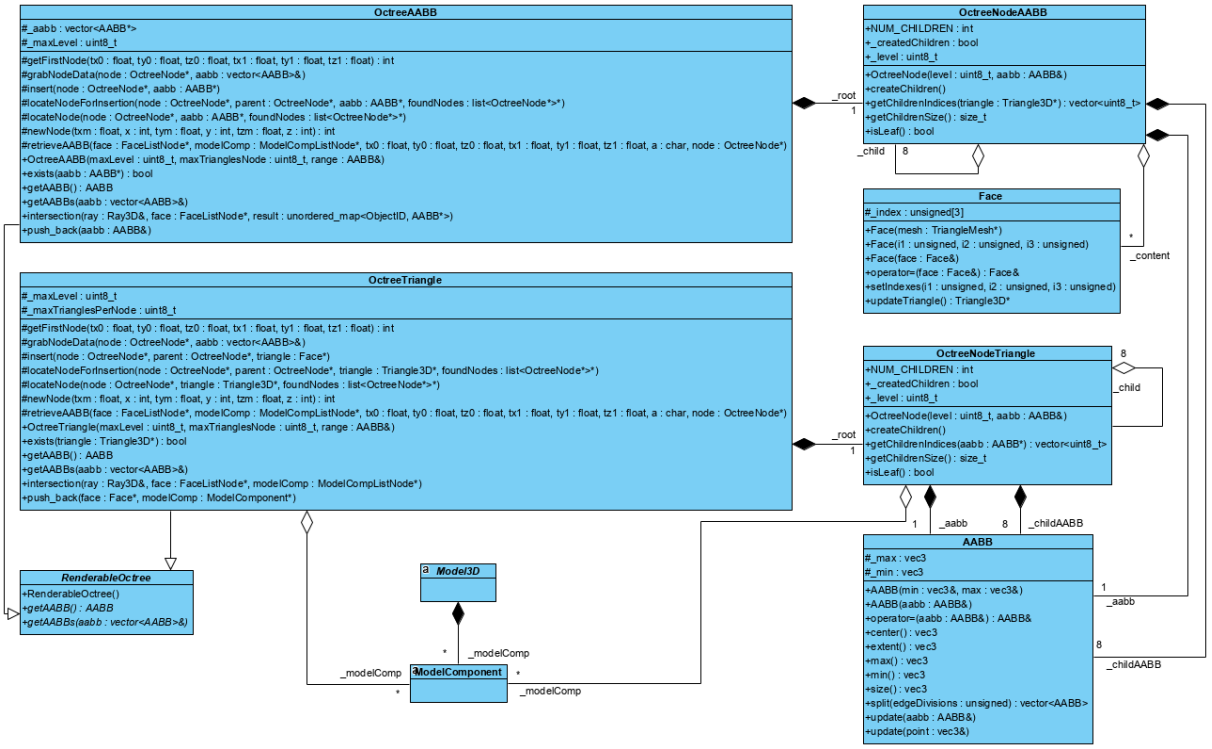


Ilustración 67. Diagrama de clases donde se muestran las entidades relacionadas con octrees.

2.3.1.4 Otros algoritmos de intersección

En apartados anteriores se definían tests de intersección entre un AABB y dos primitivas tales como un triángulo o un AABB, donde el test descrito para este segundo caso era mucho más simple. En esta sección se introducen algoritmos de intersección donde la primera primitiva no es un AABB, sino un rayo.

En el caso de la intersección rayo-triángulo, una de las soluciones más rápidas se propone en (Moller and Trumbore 1998), y se describe igualmente en (Akenine-Möller 2018). Además de este algoritmo, existen otras tantas soluciones que no necesariamente parten con tanta desventaja respecto de la primera solución citada (Badouel 1990; Jiménez et al. 2014).

La solución implementada en este trabajo se basa en el primer trabajo que se ha propuesto, el cual parte de la formulación explícita de un punto de un triángulo, donde este se define a partir de dos coordenadas baricéntricas, (u, v), sea la tercera, w, el resultado de $1 - u - v$, donde $u, v \geq 0$ y $u + v \leq 1$. Además, ya conocemos la ecuación de un segmento, que en el caso del rayo implica que $t \geq 0$, por lo que es posible hallar t para un punto del triángulo, $f(u, v)$.

$$f(u,v) = (1-u-v) * p_0 + u * p_1 + v * p_2 = o + t * d$$

$$(-d \quad p_1 - p_0 \quad p_2 - p_0)\begin{pmatrix} t \\ u \\ v \end{pmatrix} = M\begin{pmatrix} t \\ u \\ v \end{pmatrix} = (-d \;\; e_0 \;\; e_1)\begin{pmatrix} t \\ u \\ v \end{pmatrix} = o - p_0 \tag{2.25}$$

Lo que se obtiene no es más que un sistema de ecuaciones lineales que puede resolverse mediante la regla de Cramer, habiendo sustituido previamente $o - p_0$ por s para simplificar la expresión que sigue:

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{\det(-d, e_1, e_2)}\begin{pmatrix} \det(s, e_1, e_2) \\ \det(-d, s, e_2) \\ \det(-d, e_1, s) \end{pmatrix} \tag{2.26}$$

Dicha expresión se puede simplificar aún más, de tal manera que al final se obtiene que t, u y v pueden calcularse como se muestra a continuación:

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{-\mathrm{n} * \mathrm{d}}\begin{pmatrix} \mathrm{n} * \mathrm{s} \\ \mathrm{m} * \mathrm{e}_2 \\ -\mathrm{m} * \mathrm{e}_1 \end{pmatrix} \tag{2.27}$$

En esta formulación final, sabemos que $s = o - p_0$, $m = s \times d$ y que n es la normal del triángulo. Una vez conocemos cómo calcular los valores paramétricos $\{t, u, v\}$, podemos presentar la implementación de este test de intersección en forma de pseudocódigo (Ilustración 68). Una de las grandes ventajas de este test es que al basarse en un enfoque parámetrico, disponemos de los valores de $\{t, u\ v\}$, y por tanto, el test no sólo devuelve un valor booleano indicando si existe o no la intersección, sino también el punto concreto donde esta se produce.

Mediante este test de intersección podríamos conocer qué triángulos, dentro de una lista de candidatos, intersectan realmente con el rayo. Sin embargo, en secciones anteriores se propone un enfoque donde comenzamos descartando aquellos modelos con los que no intersecta el rayo. Por tanto, aún no disponemos de la capacidad de resolver el escenario completo que se plantea.

Algorithm 4 *RayTriangleIntersect*: Test de intersección de un triángulo T y un rayo r

$e_1 \leftarrow T.p_1 - T.p_0$
$e_2 \leftarrow T.p_2 - T.p_0$
$q \leftarrow r.d \times e_2$
$detM \leftarrow e_1 * q$

if $abs(detM) < \epsilon$ **then**
    No intersección
end if

$f \leftarrow 1/detM$
$s \leftarrow r.orig - p_0$
$u \leftarrow f(sq)$

if $u < 0$ **then**
    No intersección
end if

$r \leftarrow s \times e_1$
$v \leftarrow f(dr)$

if $v < 0 \vee u + v > 1$ **then**
    No intersección
end if

$t \leftarrow f(e_2 r)$

if $t \geq 0$ **then**
    Intersección
end if

No intersección

***Ilustración 68. Test de intersección de un triángulo y un rayo mediante* (Moller and Trumbore 1998).**

El siguiente test que debemos plantear es el que comprueba la intersección entre un rayo y un AABB. En esta área existe un número elevado de algoritmos que resuelven este problema, aunque el objetivo siempre es el mismo: obtener el algoritmo más eficiente que resuelva la intersección en el mínimo tiempo posible. Uno de estos

algoritmos se describe en (Eisemann et al. 2007), y su principal ventaja se debe comprender cuando se muestre la implementación del mismo.

El enfoque que toma el algoritmo nos recuerda al de otros trabajos relacionados con estructuras, como el BVH, que más tarde veremos (Hendrich et al. 2019), dado que se basa en la clasificación de un rayo. Al tratarse de una primitiva definida por un punto de origen, p, y un vector dirección d, se identifican hasta veintisiete tipos de rayos en función de todas las combinaciones de signos posibles en sus tres componentes (positivo, cero y negativo). Al considerar esta clasificación, no es necesario ejecutar un test de intersección con las seis caras del AABB, sino que es posible acotar este número a tres. Por ejemplo, un rayo de tipo MMM (*minus, minus, minus*), donde cualquier componente de d es negativa, sólo podría colisionar con la cara superior, frontal y derecha. Al fin y al cabo, se trata de indicar manualmente qué caras se corresponden con cada tipo de rayo. En contraposición con MMM, PPP sería un rayo de dirección positiva en todos sus componentes, mientras que O sería el indicador para el valor 0.

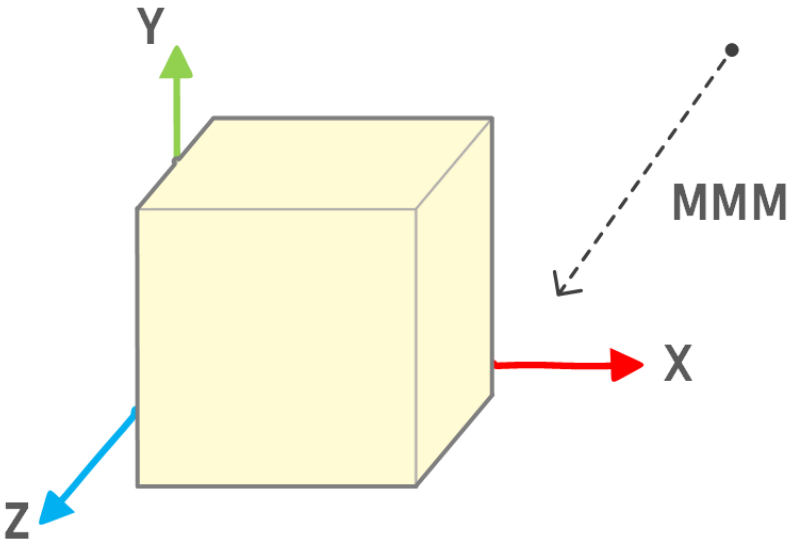


Ilustración 69. Representación de ejes (frontera del cubo) que deben compararse con un rayo de dirección negativa en todos sus componentes.

Para determinar si el rayo intersecta con el AABB es necesario hallar relación de este y aquellos ejes que conforman los límites de la superficie establecida por los tres planos seleccionados. En la Ilustración 69 se enfatizan los ejes que deberían comprobarse para un rayo de tipo MMM. Para hallar esta relación se emplean lo que se conocen como coordenadas de Plücker, de tal manera que nos permite expresar una línea L, que pasa entre los puntos A y B en un espacio tridimensional, mediante

seis valores. La línea L se puede expresar en términos de A y B, o bien en términos de una posición de origen, O, y una dirección, D.

$$\begin{aligned} L_0 &= A_x B_y - B_x A_y = O_x D_y - D_x O_y \\ L_1 &= A_x B_z - B_x A_z = O_x D_z - D_x O_z \\ L_2 &= A_x - B_x = -D_x \\ L_3 &= A_y B_z - B_y A_z = O_y D_k - D_y O_z \\ L_4 &= A_z - B_z = -D_z \\ L_5 &= B_y - A_y = D_y \end{aligned} \tag{2.28}$$

Nótese como la primera notación, expresada en términos de A y B, nos puede ayudar a describir una arista del cubo, mientras que la segunda notación permite describir el rayo. Sin embargo, estos seis valores no suponen información relevante de la relación entre ambas primitivas, al menos cuando se consideran de manera aislada. Necesitamos una función que el artículo ha acertado en llamar $side(line_1, line_2)$, que nuestro caso será $side(L, R)$. Esta función se define como el producto escalar permutado de las coordenadas de L y R, donde se dan los siguientes escenarios:

$$side(L,R) = \begin{cases} > 0, & Clockwise\ relative\ orientation \\ 0, & Intersection \\ < 0, & Counterclockwise\ relative\ orientation \end{cases} \tag{2.29}$$

La función $side(L, R)$ se define finalmente mediante la expresión que se muestra a continuación, la cual se desarrolla a partir de una serie de transformaciones que se sustentan sobre la formulación anteriormente indicada de las coordenadas de Plücker.

$$side(L,R) = -(D * N) = -D * \big((A - O) \times (B - O)\big) = -D_x L_3 + D_y L_1 - D_z L_0 + R_1 L_5 + R_0 L_4 + R_3 L_2 \tag{2.30}$$

Sea D la dirección del rayo y N la normal del plano que contiene a A, B y O. Además, cabe mencionar que se puede utilizar indistintamente $side(L, R)$ o $side(R, L)$, aunque el signo del resultado será justamente el opuesto.

La última comprobación que define el algoritmo intenta determinar si el AABB se encuentra en la parte positiva del rayo, dado que los tests de Plücker asimilan que ambas partes son líneas infinitas (recordemos que se definían mediante dos puntos

únicamente). Este último test es fácilmente implementable dado que conocemos el origen del rayo, O, y la mínima posición del AABB, p_0. Por tanto, el AABB se encuentra a la derecha del rayo si $O_x < p_{0_x}$, $O_y < p_{0_y}$y $O_z < p_{0_z}$.

A partir de aquí, se muestran los principales aspectos de nuestra implementación. A diferencia de los tests anteriores, el número de atributos y métodos ligados a este test es elevado, por lo que se crea una clase adicional, ***EisemannRay***, igualmente integrada en los ficheros *Ray.h* y *Ray.c*pp (como su versión básica, *Ray3D*).

- Una de las ventajas de este método se encuentra vinculada al lenguaje de programación y las posibilidades que este ofrece. De esta manera, disponemos de 26 tipos rayos, por lo que una estructura condicional con tantas comprobaciones reduciría la eficiencia del algoritmo. Una alternativa a esta solución es la construcción de una tabla *hash* que relacione cada posible identificador de un rayo con una función de estructura previamente definida (`std::function<bool(const EisemannRay&, const vec3&, const vec3&)`). De esta manera, dado un rayo y el identificador de su tipo, es posible recuperar un objeto de la tabla *hash*, que no será más que una función, e invocarla con la instancia del rayo y los límites del AABB.

- Es evidente que dicha estructura de datos no se encuentra ligada a un rayo o un AABB en concreto, y por tanto, es suficiente con inicializarla al comienzo de la aplicación a través de un método estático. Habrá que insertar hasta 26 funciones, todas ellas diferentes y definidas en el código fuente que acompaña al artículo. Por ejemplo, el comportamiento de la función que acompaña al tipo MMM se define como sigue:

$$\begin{aligned} hit = !\,(O < p_{min} \vee \left(r_{\cdot jbyi} * p_{min_x} - p_{max_y} + ray_{cxy} > 0\right) \\ \vee \left(r_{\cdot jbyj} * p_{min_y} - p_{max_x} + ray_{cyx} > 0\right) \\ \vee \left(r_{\cdot jbyk} * p_{min_z} - p_{max_y} + ray_{czy} > 0\right) \\ \vee \left(r_{\cdot kbyj} * p_{min_y} - p_{max_z} + ray_{cyz} > 0\right) \\ \vee \left(r_{\cdot kbyi} * p_{min_x} - p_{max_z} + ray_{cxz} > 0\right) \\ \vee \left(r_{\cdot ibyk} * p_{min_z} - p_{max_x} + ray_{czx} > 0\right) \end{aligned} \tag{2.31}$$

- Dado un rayo R, es posible precalcular todos aquellos parámetros que serán necesarios en el proceso. Igualmente, se deberá clasificar correctamente, asignándole un identificador a partir del signo de las tres coordenadas de su

dirección. El signo de cada componente de la dirección d de un rayo se puede extraer fácilmente mediante la siguiente operación:

$$sign = (0 < value) - (value < 0) \tag{2.32}$$

En C++ es posible implementar esta misma función como una plantilla, donde 0 pasa a convertirse en $T(0)$, de tal manera que el tipo de dato con el que se trabaja debe definir un constructor que reciba un valor numérico, así como sobrecargar el operador $<$.

El principal problema que encontramos en este punto es hallar la función $f(x, y, z)$, sean $\{x, y, z\}$ tres valores que representan el signo en X, Y y Z. No es posible una representación en forma de bit en tanto que no hay sólo dos valores posibles por coordenada, sino tres. Lo que se propone es definir tres vectores de tamaño 3, uno por coordenada, que contengan potencias de dos tal que la suma de cualquier combinación no colisione con otra. Una solución, aunque no la única, es la siguiente:

$$\begin{aligned} vector_x &= \{2, 4, 8\} \\ vector_y &= \{16, 32, 64\} \\ vector_z &= \{128, 256, 512\} \\ r_{type} &= vector_x[sign_x] + vector_y[sign_y] + vector_z[sign_z] \end{aligned} \tag{2.33}$$

Por una mera cuestión de comodidad, el único punto por resolver es la asignación de identificadores a cada valor del `enum RayType`, dado que una enumeración que comience por 0 no es válida para *castear* directamente el identificador obtenido hacia el tipo de dato que representa el *enum*. De esta manera, ha sido necesario establecer manualmente un identificador para cada uno de los 26 tipos de rayos, que se corresponde con la suma que se ha definido previamente.

La definición de la última clase aquí empleada, *EisenmannRay*, se muestra en la Ilustración 70 mediante un diagrama de clases. En dicho diagrama también se muestra la entidad encargada de *renderizar* un rayo, de manera genérica, lo que nos servirá más tarde para mostrar el resultado de la generación de rayos.

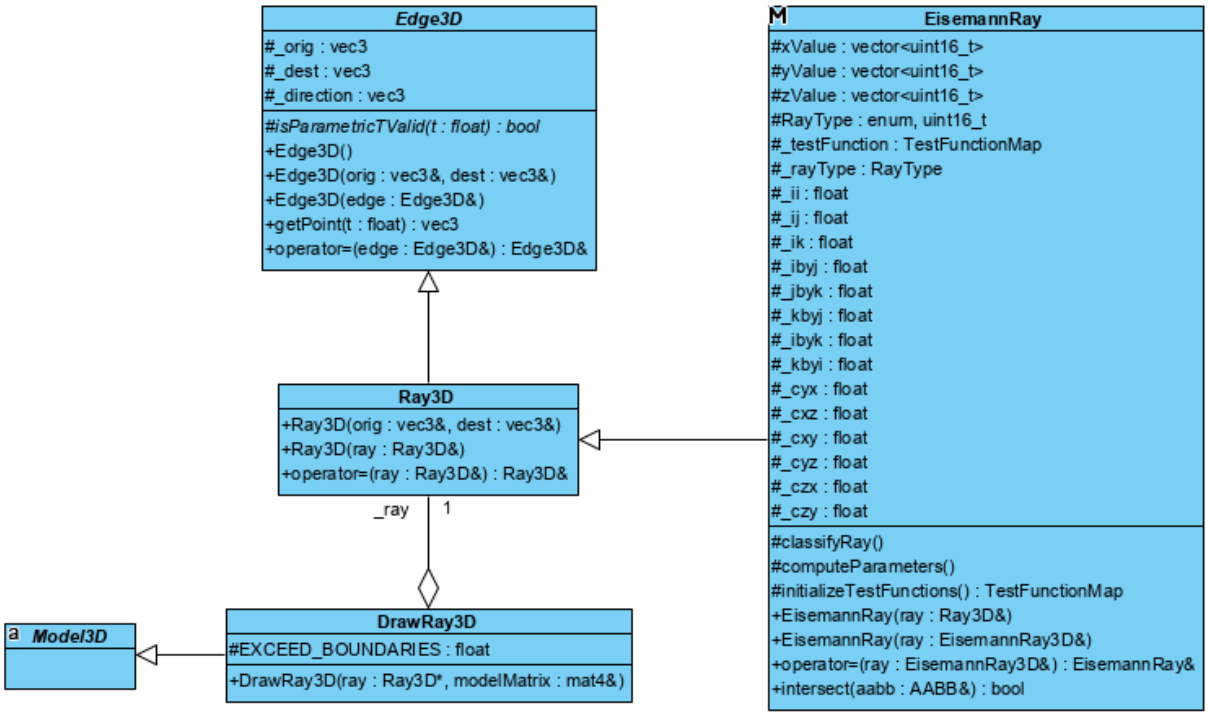


Ilustración 70. Diagrama de clases que muestra las entidades de las que depende un rayo perteneciente al algoritmo de Eisemann.

2.3.1.5 Simulación

En este punto del desarrollo es posible crear un escenario básico, estructurarlo en un *octree*, y resolver las intersecciones de un conjunto de rayos con dicha estructura y sus primitivas almacenadas. En definitiva, sólo faltaría organizar el proceso de la simulación bajo una entidad a la que llamaremos ***LiDARSimulation***, y que de hecho, conservará este nombre hasta el final de proyecto. Esta entidad se puede concebir como el propio sensor LiDAR, y se encargará tanto de la generación de los rayos como de resolver todas las intersecciones, y por tanto, también generará las nubes de puntos resultantes.

Una instancia de *LiDARSimulation* recibe un grupo de modelos 3D en su construcción, dado que debe conocer el escenario sobre el que actuar. A partir de este grupo, inicializará un *octree* con un nivel máximo de profundidad y un tamaño límite para cada nodo, ambos indicados previamente en el constructor. Nótese como sólo disponemos de un grupo, mientras que el *octree* debe contener los triángulos de cualquier modelo de la escena, por lo que a partir de aquí se inicia un proceso recursivo controlado por la entidad *Group3D*, para que cualquier instancia de un modelo 3D itere sobre sus primitivas y las introduzca en la estructura de datos.

El siguiente paso relevante, es la generación de todos aquellos rayos que el dispositivo emite sobre la escena y producen intersecciones, generando en última instancia una nube de puntos. Aunque se adapta la entidad *LiDARSimulation* para un escenario de captura aérea y terrestre, en este punto sólo se pone en marcha el segundo método para la generación de rayos. Los parámetros del proceso que pueden controlarse en este punto son los siguientes:

- Número de rayos que se generan en el espacio de 360 grados alrededor del eje Y. Este valor sirve para definir la densidad de rayos en el plano horizontal.
- Número de rayos en el espacio vertical, de 180 grados. Es un parámetro análogo al anterior, y sólo sirve para definir la densidad de rayos en el espacio vertical. El número total de rayos generados se calcula como $rays_x * rays_y$.
- Un *offset*, en forma de ángulo, que sólo influye en el espacio vertical. Debería tenerse en cuenta en un LiDAR terrestre debido a que este no es capaz de captar todo el espacio completo, y de hecho, el mayor problema se presenta en el espacio vertical, dado que suele existir un límite de apertura. El usuario indica el ángulo en grados, y este se transforma tal que $\alpha = clamp(\beta, 0, 45) * \frac{\pi}{180}$, de tal manera que se limita el *offset* a 45 grados, y se almacena en radianes.

La generación de rayos se implementa como un proceso iterativo, considerando tanto $rays_x$ como $rays_y$, de tal manera que es posible subdividir el espacio de 360 grados tantas veces como $rays_x$, obteniéndose así θ. Para un valor de θ es posible calcular la posición que le corresponde en la frontera de una esfera normalizada, donde sólo podrán obtenerse valores distintos de 0 en X y Z. Así, se obtiene una posición $(\cos\theta, 0, -\sin\theta)$. A partir de dicha posición se podrá calcular un eje de rotación, que servirá para rotar el punto generado y desplazarlo a lo largo de lo que hemos llamado espacio vertical (los 180 grados que cubre la esfera normalizada). Este eje de rotación siempre tendrá una componente $Y = 0$, de tal manera que se corresponde con una rotación de 90 grados en sentido horario de la posición previamente calculada. Es decir, se calcula como $(-p_z, 0, p_x)$.

Después de esto, es posible subdividir el espacio de 180 grados tantas veces como indique $rays_y$, obteniéndose γ. Sin embargo, el espacio que abarca no tiene por qué ser de 180 grados, dado que se ha definido un *offset* en la base del dispositivo (Ilustración 71). Por tanto, $\gamma = \frac{\pi - offset}{rays_y}$. Sólo quedaría rotar p en función del eje e antes calculado, considerando el valor de γ. El valor de partida para realizar esta segunda rotación se define como $-\frac{\pi}{2} - offset$, es decir, -90 grados más el *offset* indicado, que sólo se aplica en la parte inferior del dispositivo, donde se encuentra el soporte de este y se identifica un límite de apertura.

Un ejemplo de la generación de rayos en la escena inicial se muestra en la Ilustración 73. Dentro de este proceso iterativo, también se comprobarán intersecciones y se obtendrán los puntos resultantes, como bien se define en el siguiente bloque de pseudocódigo (Ilustración 72).

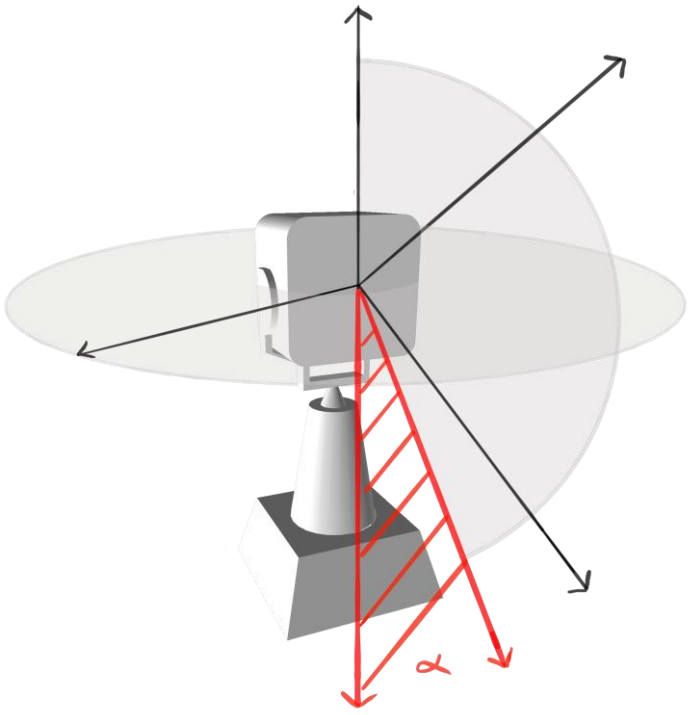


Ilustración 71. Representación del rango de captura del dispositivo LiDAR.

Algorithm 5 *TerrestrialSimulation*: Generación de rayos de LiDAR terrestre

```
1: x_angle ← π
2: x_spacing ← 2π/rays_x; y_spacing = (π − offset)/rays_y
3:
4: for x = 0, 1, ..., rays_x − 1 do
5:     pos ← (cos(x_angle), 0, −sin(x_angle))
6:     axis_rot ← (−pos_z, 0, pos_x)
7:     y_angle ← 0
8:
9:     for y = 0, 1, ..., rays_y − 1 do
10:        pos_sphere ← rotate(angle_y, axis_rot) ∗ pos
11:        ray ← Ray(lidar_pos, lidar_pos + pos_sphere)
12:
13:        octree.intersection(ray, faceList, modelCompList)
14:        p, mod ← findIntersection(ray, faceList, modelCompList)
15:
16:        Añadir puntos a nube uniforme
17:        Guardar componentes en un unordered_set
18:
19:        y_angle += y_spacing
20:    end for
21:
22:    x_angle += x_spacing
23: end for
24:
25: for each model ∈ modelCompMap do
26:    model.loadPoints()
27: end for
```

Ilustración 72. Pseudocódigo de generación de rayos y recuperación de intersecciones.

En primer lugar, se puede observar como el resultado de una intersección es un conjunto de puntos, y no sólo uno. En este punto del desarrollo se incorpora la noción de opacidad de un modelo, lo que no implica que el material tenga esta propiedad, sino que un rayo podrá traspasarlo. Por ejemplo, esta propiedad debería aplicarse a la vegetación o a las hojas de un árbol. La implementación realizada propone la utilización de rayos de opacidad inicial uno, que se irá decrementando a medida que se colisione con superficies. Al tratarse de una etapa inicial, la versión descrita es muy simple y tendría el mismo comportamiento para cualquier rayo, algo que no necesariamente es realista.

En segundo lugar, de una intersección no sólo nos interesa el punto resultante, sino también el componente al que pertenece dicho modelo. Aunque es cierto que una nube de puntos apta como resultado podría adoptar un color uniforme, no todos los tipos de resultados toman este enfoque. El más básico, y que justifica la ejecución de este proyecto, es un *rendering* donde los puntos de cada modelo adoptan un color que

indica el concepto semántico de dichos modelos. Otro tipo de nube de puntos que justifica la distinción de puntos por modelos, aunque menos intuitiva, es el *rendering* RGB, dado que cada modelo deberá renderizar únicamente sus puntos, indicando previamente el material o cualquier otra propiedad que sólo se aplique a dicha instancia. Es necesario indicar que en este último ejemplo el problema presenta una mayor dificultad, dado que el *rendering* RGB supone un sombreado, por lo que es necesario conocer otros datos de la geometría en esa posición, como la normal o la coordenada de textura.

Por tanto, es necesaria una nube de puntos formada por todas las intersecciones halladas, pero cada modelo también debe contener un conjunto de intersecciones que sólo le afecten a él. Ambas nubes de puntos deben cargarse al final del proceso de la simulación, lo cual se corresponde únicamente con una generación de las estructuras de dibujo en GPU.

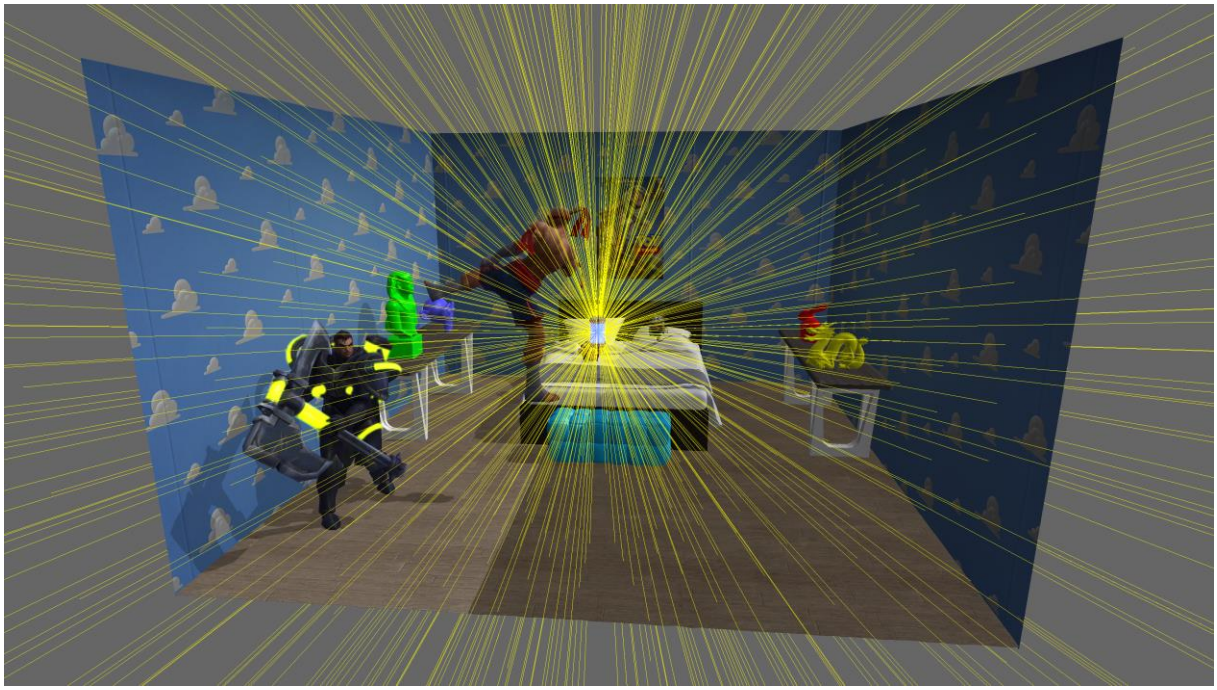


Ilustración 73. Representación de 400 rayos emitidos en una simulación.

El proceso de resolución de la intersección para un rayo vendrá definido por la función `findIntersection`, citada en el último bloque de pseudocódigo. La entrada de la misma será el propio rayo, así como una lista de listas de triángulos, y una estructura análoga que permitirá vincular cada triángulo con su correspondiente modelo. Cada uno de estos vectores se corresponde con las intersecciones halladas en un único nodo del *octree*, de tal manera que si es necesario reordenar las

intersecciones en función de la distancia, sólo debe hacerse para las intersecciones halladas en un único nodo, dado que su tamaño es necesariamente menor. Nótese como en el algoritmo de *traversal* sí se recorrían los nodos ordenadamente, dado que se calcula en todo momento cuál es el nodo de entrada, pero dentro de un nodo se almacenan los triángulos sin ningún tipo de orden, y en cualquiera de los casos, dependería de la dirección del rayo.

Como parámetros de entrada y salida también encontramos un vector de puntos hallados y otro vector con el componente al que pertenecen. El algoritmo, también descrito en el pseudocódigo de la Ilustración 74, se implementa como sigue:

1. Se itera por cada una de las listas de triángulos (y de componentes) mientras no se alcance el final o la propiedad de opacidad no haya alcanzado el cero. Frente a la mayoría de materiales, el rayo no podrá continuar, y por tanto, será suficiente con la primera intersección hallada.

2. Se recorre la lista actual de triángulos y se comprueba la intersección del rayo con cada uno de ellos. Recordemos que hasta ahora se ha verificado que el rayo colisiona con el AABB que los contiene, pero no con las primitivas. Se debe comprobar la intersección con todos los triángulos dado que no hay una ordenación previa por distancia. Todas las colisiones se almacenarán en un multimapa que permitirá ordenarlas en función de la distancia a la que se produce la colisión. Para cada colisión se almacenará en una tabla *hash* el triángulo asociado y el componente al que pertenece.

Algorithm 6 *FindIntersection*: Procesamiento de posibles intersecciones de un único rayo

```
1:  ray_opacity ← 1
2:  while ray_opacity > 0 ∧ Nodos sin procesar > 0 do
3:      for each triangle ∈ nodeList do
4:          if Existe intersección rayo-triángulo then
5:              Introducir intersección con distancia en multimap
6:          end if
7:      end for
8:
9:      while ray_opacity > 0 ∧ Intersecciones sin procesar > 0 do
10:         Calcular resto de parámetros de intersección
11:         Incluir intersección en nubes de puntos
12:         ray_opacity −= model_opacity
13:     end while
14: end while
```

Ilustración 74. Pseudocódigo de la resolución de intersecciones de un único rayo.

3. Se itera sobre el conjunto de colisiones halladas, a pesar de que estas se encuentran ordenadas, dado que es posible que exista más de una intersección cuando la opacidad del material donde se produce la primera colisión no es uno, y además, la "opacidad" asociada al rayo no ha alcanzado el cero. En cualquiera de los casos, se trata de una iteración sobre el multimapa antes citado.

A medida que se observa qué colisiones finalmente se consideran, es posible calcular más parámetros asociados a las mismas, y no antes, dado que así evitamos ciertos cálculos vinculados a colisiones que no se van a incluir en la solución final.

La normal se puede recuperar automáticamente a partir del triángulo al que pertenece el punto (se obtiene de una tabla *hash*), pero esto no sucede para la coordenada de textura. Para calcular dicha coordenada habrá que recurrir de nuevo a la regla de Cramer, dado que el punto de intersección, p, se puede expresar como un conjunto de coordenadas baricéntricas aplicadas a la fórmula del triángulo (Ericson 2004). De hecho, esta fórmula se describió previamente:

$$f(u,v) = P = u * p_0 + v * p_1 + w * p_2 = u * p_0 + v * p_1 + (1 - u - v) * p_2 \quad (2.34)$$

Esta misma ecuación se puede reformular como continúa, teniendo en cuenta que realmente se puede interpretar como la parametrización de un plano:

$$P = p_o + v(B - A) + w(C - A)$$

$$P - A = v(B - A) + w(C - A) \quad (2.35)$$

$$v_2 = u * v_0 + w * v_1; \quad v_2 = P_A,\ v_0 = B - A,\ v_1 = C - A$$

Habiéndose escogido u y w para aclarar nomenclatura respecto de v.

Mediante un producto escalar es posible formar un sistema de ecuaciones lineales que podrá resolverse fácilmente (Ilustración 75):

$$v_2 * v_0 = v_0(u * v_0 + w * v_1)$$

$$v_2 * v_1 = v_1(u * v_0 + w * v_1)$$

$$\begin{pmatrix} v_0v_0 & v_0v_1 \\ v_1v_0 & v_1v_1 \end{pmatrix}\begin{pmatrix} u \\ w \end{pmatrix} = \begin{pmatrix} v_2v_0 \\ v_2v_1 \end{pmatrix} \quad (2.36)$$

$$u = \frac{\begin{bmatrix} v_2v_0 & v_0v_1 \\ v_2v_1 & v_1v_1 \end{bmatrix}}{\begin{bmatrix} v_0v_0 & v_0v_1 \\ v_1v_0 & v_1v_1 \end{bmatrix}}; \quad w = \frac{\begin{bmatrix} v_0v_0 & v_2v_0 \\ v_1v_0 & v_2v_1 \end{bmatrix}}{\begin{bmatrix} v_0v_0 & v_0v_1 \\ v_1v_0 & v_1v_1 \end{bmatrix}}; \quad v = 1 - u - w$$

Algorithm 7 *ComputeTextCoord*: Cálculo de coordenadas de textura para un punto p de un triángulo

```
1:  v0 ← t.p1 − t.p0
2:  v1 ← t.p2 − t.p0
3:  v2 ← p − t.p0
4:
5:  d00 ← dot(v0, v0)
6:  d01 ← dot(v0, v1)
7:  d11 ← dot(v1, v1)
8:  d20 ← dot(v2, v0)
9:  d21 ← dot(v2, v1)
10: denom ← d00 ∗ d11 − d01 ∗ d01
11: v ← (d11 ∗ d20 − d01 ∗ d21)/denom
12: v ← (d00 ∗ d21 − d01 ∗ d20)/denom
13: u ← 1 − v − w
14:
15: tc_result ← u ∗ tc0 + v ∗ tc1 + w ∗ tc2
```

Ilustración 75. Pseudocódigo del cálculo de coordenadas de textura de un punto p perteneciente a un triángulo.

En este punto del desarrollo disponemos tanto de una nube de puntos con todas las colisiones observadas, bajo la entidad ***PointCloud3D***, sea ***DrawPointCloud3D*** la encargada de *renderizarla*, como de múltiples nubes de puntos dispersas bajo cada componente de un modelo 3D, con el fin de lograr un tipo de *rendering* completamente diferente.

Todo el comportamiento aquí descrito se integra en la clase *LiDARSimulation*, la cual se define como se muestra en el diagrama de clases de la Ilustración 76.

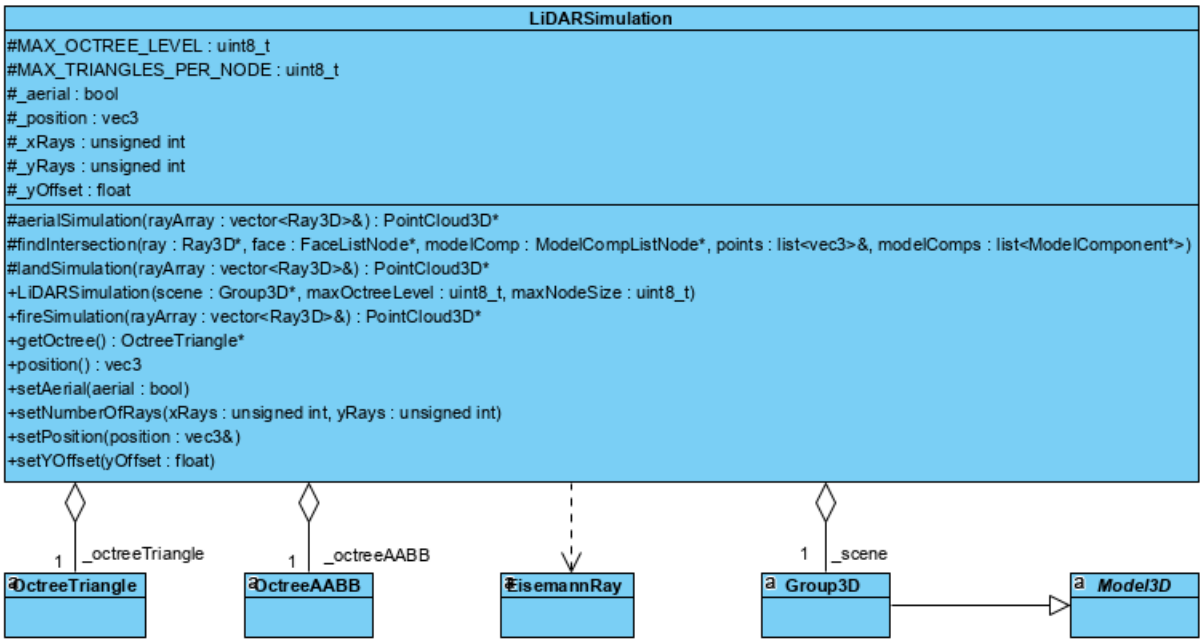


Ilustración 76. Diagrama de clases donde se representan aquellas entidades relacionadas con la simulación.

Por tanto, finalizamos este apartado mostrando los resultados obtenidos al renderizar la nube de puntos resultante, tanto de manera uniforme (Ilustración 77) como en RGB (Ilustración 78). Para lograr ambas capturas se ha ejecutado una simulación de 250.000 rayos, donde el *offset* del dispositivo es de $\frac{\pi}{9}$.

Ilustración 77. Nube de puntos renderizada de manera uniforme.

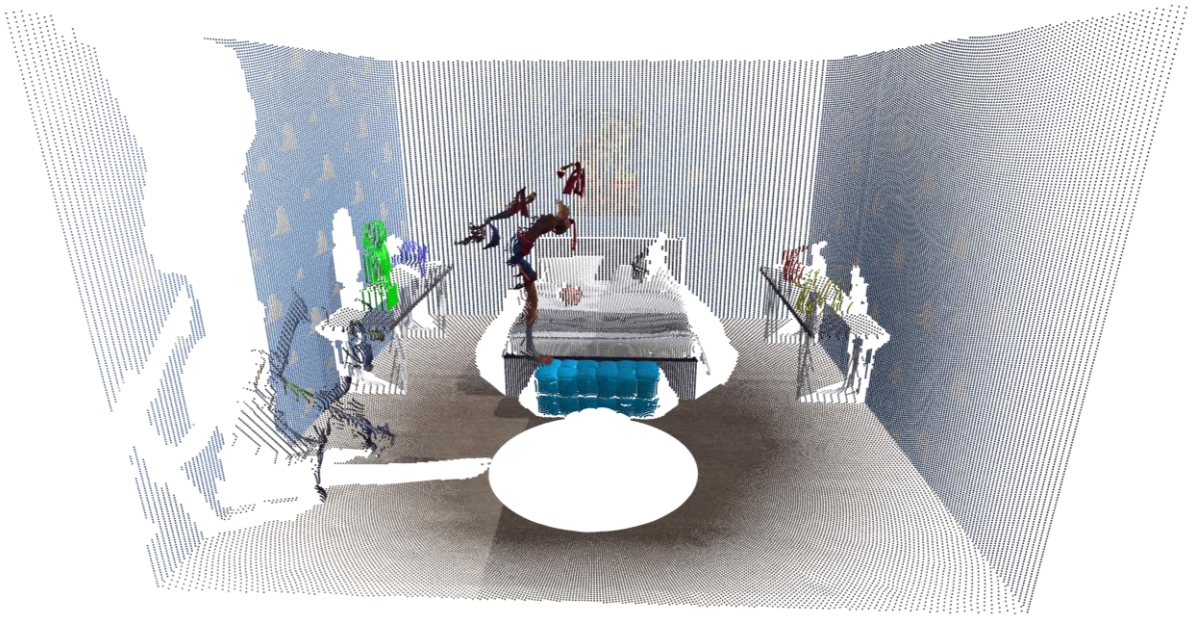


Ilustración 78. Nube de puntos RGB compuesta por los puntos de intersección de cada uno de los modelos.

Nótese como en ambas capturas se puede apreciar cierto patrón en el *renderizado* de las nubes de puntos, que se asemeja a un patrón de Moiré, un efecto de distorsión que se genera cuando interfieren varios patrones de líneas, el cual puede producirse en la generación de imágenes por ordenador. Por ejemplo, este patrón también puede darse en la implementación de la técnica de *shadow mapping* (Ilustración 79), aunque en esa situación se trata de un problema de resolución y su solución se encuentra documentada. En nuestro caso, este patrón se puede observar cuando el número de puntos de la nube es muy elevado. En posteriores iteraciones se considerará este artefacto y se solventará a través de una solución a la que podríamos llamar *jittering*, basada en una idea de (Akenine-Möller 2018) para aliviar el problema del *aliasing*.

Ilustración 79. Representación de una escena donde la técnica de shadow mapping genera un patrón de Moiré (shadow acne). Se modifica el brillo y el contraste de una zona afectada para permitir su visualización.

2.3.1.6 Diagrama de clases

Al igual que en la última iteración, se introduce este apartado para conocer principalmente las relaciones entre aquellas clases necesarias en la implementación de todo lo que en este apartado se describe. Asimismo, se utiliza este diagrama (Ilustración 80) para mostrar el contenido de otras clases que no se han mencionado a lo largo de este apartado, debido a que son entidades especialmente evidentes o análogas a algunas de las ya ilustradas.

Todos aquellos métodos y estructuras necesarias para comprobar intersecciones se agrupan bajo el paquete *Intersections*. Sin embargo, en la implementación, las funciones definidas bajo este *namespace* no pertenecen a una clase, tal y como se muestra en el diagrama, aunque esta entidad es la única que nos permite definir el conjunto de intersecciones disponibles en un diagrama de este tipo.

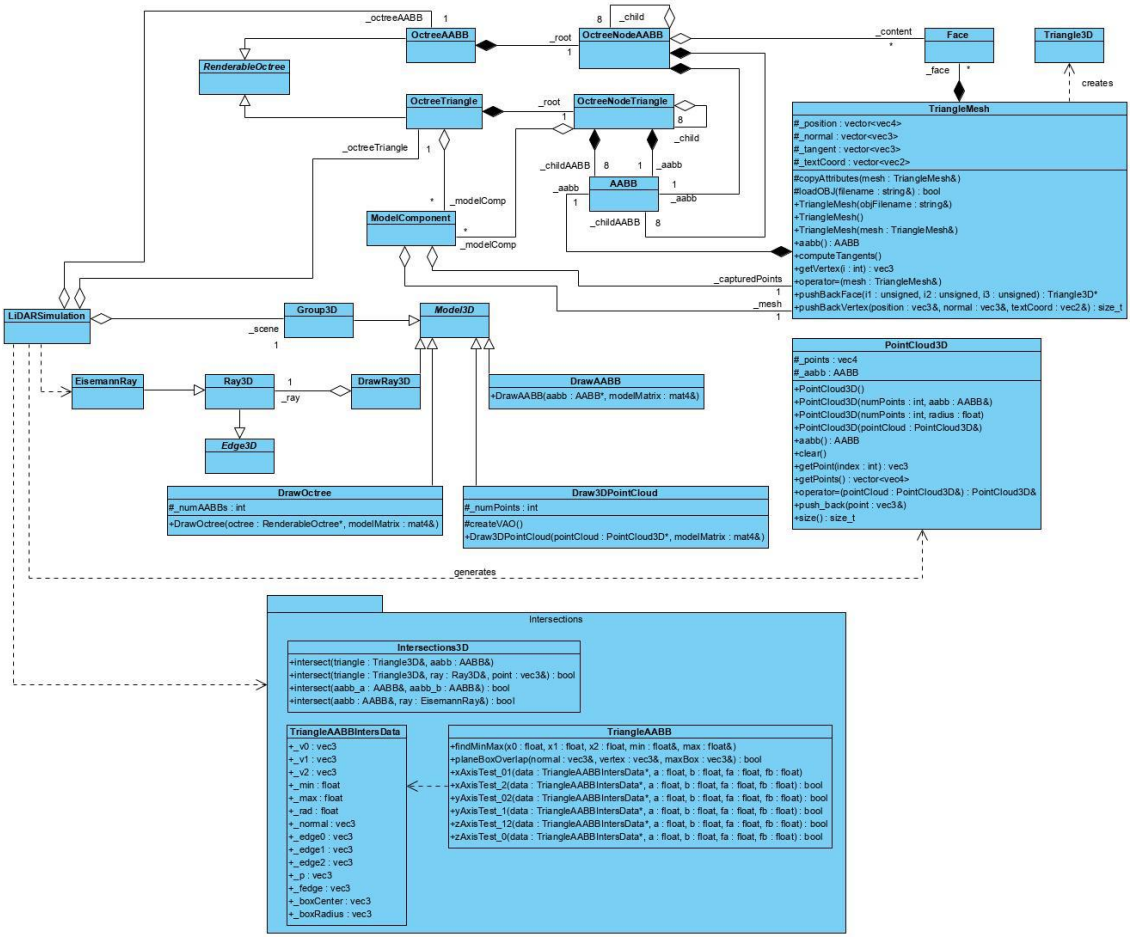


Ilustración 80. Diagrama de clases que ofrece una visión global de las entidades utilizadas durante esta iteración.

2.4 Tercera iteración

En este punto, la aplicación implementa una simulación muy básica, inacabada y no optimizada, pero esta versión nos permite observar qué rendimiento podríamos esperar de una posible aplicación final. Los tiempos de ejecución que se han expuesto en la segunda iteración sólo representan una pequeña parte del tiempo de respuesta observado, dado que además de inicializar y recorrer las estructuras de datos también es necesario cargar el escenario en el inicio de la aplicación, calcular datos derivados de dicho escenario, comprobar qué triángulos hallados son realmente intersecciones, etc.

Por tanto, se observa un tiempo de respuesta que no se ajusta a las restricciones que se exponían en Requisitos no funcionales. Más allá del cumplimiento de estos requisitos, obtener un menor tiempo de respuesta presenta beneficios para todas las partes involucradas en el sistema. Para un desarrollador representa la

capacidad de utilizar escenarios complejos, sin necesidad de acudir a configuraciones reducidas para evitar largas esperas. Para el usuario final, reducir el tiempo de respuesta en la inicialización de recursos (carga) y en la simulación implica obtener una mayor valoración de la aplicación. Por último, también debe considerarse la motivación de obtener un sistema más eficiente que cualquiera de los *frameworks* observados, en los que el número de rayos generados en cada instante de tiempo se encuentra muy acotado, con el único fin de obtener un simulador en tiempo real (frecuentemente, en un escenario de conducción autónoma).

La finalidad de esta iteración es trasladar la implementación ya realizada de CPU a GPU, lo que nos permitirá ejecutar cálculos en paralelo y en última instancia, reducir el tiempo de cómputo. El principal problema de esta transición es la necesidad de descartar algunos de los algoritmos implementados, ya sea por la complejidad o la imposibilidad de trasladarlos a un *shader*, como es el caso de una intersección rayo-AABB mediante el algoritmo de (Eisemann et al. 2007). También pueden existir otras alternativas más fácilmente paralelizables.

2.4.1 Tiempos de respuesta

Para comenzar este capítulo se muestran los tiempos de respuesta observados en la versión alcanzada tras finalizar la segunda iteración. A partir de ellos se justifica la ejecución de la transición de CPU a GPU, y además, suponen una base de tiempo de ejecución con la que comparar futuras implementaciones. La demora de la aplicación no sólo se identifica en la etapa de simulación, sino también en la inicialización de los recursos, debido a que habrá que acceder a ficheros para cargar e interpretar su contenido.

En primer lugar, el tiempo medio de respuesta obtenido en la carga de los modelos se muestra en la Tabla 23, habiendo empleado 5 ejecuciones para obtener el valor final. Como bien sabemos, la escena se compone de 310.363 vértices y 604.993 triángulos, y a diferencia de la inicialización del *octree*, ambos tipos de datos implican mayor carga de trabajo. No todos los modelos de la escena proceden de ficheros OBJ, por lo que sólo deben considerarse 300.973 vértices y 587.351 triángulos en el cálculo del tiempo medio de lectura de ficheros.

Además de la carga del fichero, la inicialización de un modelo implica más cálculos, como el cálculo de tangentes para poder aplicar la técnica de *displacement mapping*, e incluso hay otros modelos que se generan íntegramente en la aplicación,

tales como las superficies planas. Sin embargo, hay otras operaciones, como la generación del VAO para *renderizar* un modelo, que no deberían incluirse aquí, dado que no es posible optimizarlas (sin reducir la información del modelo). También se excluye la generación de la topología de nube de puntos y malla de alambre (*wireframe*), dado que son completamente innecesarias para nuestro propósito.

Tiempo medio de respuesta de la carga de modelos		
Ejecución	**Tiempo de lectura de ficheros (ms)**	**Tiempo total de carga (ms)**
1	1.665,95	1.711,21
2	1.723,23	1.767,52
3	1.691,25	1.737,96
4	1.691,82	1.741,41
5	1.708,39	1.756,35
6	1.690,7	1.737,52
	Tiempo medio (ms) 1.695,22	**Tiempo medio (ms)** 1.741,99

Tabla 23. Tiempo medio de respuesta correspondiente a la carga de modelos de la escena.

Con una escena de algo más de medio millón de triángulos existe una demora de más de un segundo y medio (1,741 de media), aunque es necesario considerar que el propósito de este trabajo experimental es comprobar la simulación de un dispositivo LiDAR en una escena de mayor complejidad y utilizando el máximo número posible de pulsos láser. Por tanto, este tiempo de respuesta no es tolerable cuando se prevee un crecimiento considerable de la geometría y topología de la escena.

El segundo punto donde se detecta una demora importante es la simulación de un sensor LiDAR, de manera que a medida que aumenta el número de emisiones, también se detecta un tiempo de respuesta muy elevado. En la iteración anterior se comprobaba únicamente el tiempo requerido para hallar posibles triángulos intersectados por un rayo, obteniéndose un tiempo cercano a un segundo y medio con la mejor configuración. Para ello se empleaba un millón de rayos, y el tiempo de la simulación completa apenas aumenta unos segundos más ante tal escenario. Sin embargo, cuando el número de rayos aumenta también se obtiene un tiempo de respuesta más elevado. En la Tabla 24 se compara el tiempo medio obtenido para

tres cantidades de rayos diferentes, considerando la simulación completa (*traversal*, intersección rayo-triángulo, ordenaciones, etc).

Tiempo medio de respuesta de la simulación LiDAR			
Millones de rayos	1	4	9
Ejecución	**Tiempo de respuesta (ms)**		
1	5.262	19.916	45.188
2	5.212	20.766	48.304
3	5.317	19.764	45.300
4	5.144	20.909	51.337
5	5.829	20.413	48.890
Tiempo medio (ms)	5.352,8	20.353,6	47.803,8

Tabla 24. Tiempo medio respuesta de una simulación LiDAR utlizando distintas cantidades de rayos.

Nótese como el tiempo obtenido pertenece a una versión cuya simulación se encuentra incompleta; es decir, el tiempo que aquí se muestra probablemente aumente al incluir más características en la simulación. Por tanto, el tiempo base representado no es más que una referencia, dado que la mejora de rendimiento de la solución final probablemente sea mayor que la pueda calcularse mediante esta tabla.

En definitiva, el tiempo medio obtenido en ambos procesos no representa un problema considerable en la versión actual del proyecto, pero sí lo será posteriormente cuando se alcance la complejidad requerida tanto en la escena como en la propia simulación. Por esta razón, se describen a continuación todas aquellas modificaciones realizadas para reducir el tiempo de respuesta.

2.4.2 Optimización de carga de modelos OBJ

La carga de un modelo OBJ se implementa hasta este punto leyendo cada una de las líneas del fichero mediante funcionalidades muy básicas de C++. Cada línea se compara con hasta cuatro formatos (vértice, normal, coordenada de textura y triángulo), pudiendo ser uno de ellos el correcto. La lectura de un fichero OBJ se mantiene simple para acelerar en la medida de lo posible este proceso, a pesar de que ciertos formatos no puedan ser interpretados correctamente.

Una solución alternativa que podría reducir el tiempo de carga es la utilización de una librería optimizada para este propósito, como *tinyobjloader*[24], donde se documenta la carga de una escena de seis millones de triángulos en un segundo y medio, utilizando para ello un ordenador no necesariamente mejor que el que se emplea en este trabajo. A la carga también es necesario añadir el tiempo derivado de transferir las estructuras de datos de la librería a las estructuras de nuestra aplicación, además del cálculo de cualquier otro tipo de dato asociado a geometría y/o topología, como es el caso de las tangentes. Por tanto, volveríamos al punto de partida, aunque es posible que se obtuviera cierta mejora.

Otra solución es la utilización de **ficheros binarios** junto a la carga de modelos OBJ ya descrita. De esta manera, la primera vez que se emplea un modelo será necesario cargarlo mediante el método hasta ahora empleado. Sin embargo, una vez se dispone del modelo cargado es posible almacenarlo en un fichero, y más concretamente, un fichero binario al que se ha añadido la terminación “-bin” para diferenciarlo. Lo que se almacena no es exactamente la misma información que se encuentra en el fichero OBJ (de ser así, no habría ventaja alguna), sino que se almacenan las estructuras de datos como un conjunto de caracteres (`char*`) mediante algunas funciones básicas de escritura de C++. Para ello se indica tanto la dirección de memoria de la variable que se va a almacenar como el tamaño de la misma. En el caso de un vector será la dirección del primer elemento contenido, y el tamaño se definirá como `sizeof(T) * vectorSize`, sea `T` el tipo de elemento contenido. El fichero en el que se almacena toda esta nueva información no es en absoluto legible.

Al igual que en el almacenamiento, es posible realizar la acción de lectura indicando la dirección de la variable destino y el tamaño de los datos que se esperan, que deben calcularse mediante la multiplicación antes expuesta. Esto implica que en la escritura no sólo deben almacenarse las estructuras de datos del modelo, sino también el tamaño de las mismas, y además, dicho tamaño debe posicionarse antes que el conjunto de caracteres de la estructura. El valor de una variable que indica la longitud de una estructura se calcula como `sizeof(size_t)`.

En la lectura, deberá recuperarse primero el tamaño de una estructura, ajustar el tamaño de la misma (en el caso de un vector, `resize`) e introducir un conjunto de caracteres en la misma, sin que sea necesario *casting* alguno para transformar el tipo

[24] https://github.com/tinyobjloader/tinyobjloader

de dato. Al final, esta solución actúa a muy bajo nivel introduciendo unos valores binarios en cada posición.

Esta solución, que es la finalmente implementada, presenta una serie de ventajas y desventajas que deben analizarse:

- A diferencia de la utilización de una librería o de una carga a un nivel muy bajo, no es necesario transferir los datos hacia las estructuras de datos de la aplicación (salvo una primera vez).
- La carga y escritura se implementa mediante apenas unas pocas líneas de código, y el proceso es muy eficiente al trabajar a tan bajo nivel.
- Almacenar la información del modelo después de la carga implica también que no es necesario calcular datos derivados tales como tangentes o topología.
- Una desventaja es que almacena directamente la estructura de datos, por lo que un cambio en la misma invalidaría la información ya almacenada (aunque sería suficiente con indicar que cargue el modelo desde cero en la siguiente ejecución).

Por tanto, sólo nos quedaría documentar el tiempo de respuesta obtenido mediante esta solución. La escena empleada es la misma que se utilizó al hallar el tiempo medio de respuesta con la función de carga original (300.973 vértices y 587.351 triángulos). En este caso, sólo se contempla el tiempo asociado a la carga del fichero, dado que cualquier cómputo posterior no ha sido optimizado aún. En cualquier caso, la utilización de un fichero binario evita buena parte de las operaciones posteriores (si no todas).

Tiempo medio de respuesta de la carga de modelos			
Ejecución	**Tiempo de lectura de ficheros (ms)**	**Tiempo previo de lectura de ficheros (ms)**	
1	67,764	1.665,95	
2	65,243	1.723,23	
3	70,225	1.691,25	
4	66,754	1.691,82	
5	64,686	1.708,39	
6	70,308	1.690,7	
	Tiempo medio (ms) 67,49	**Tiempo medio (ms)**	1.695,22

Tabla 25. Comparativa del tiempo medio de respuesta de la carga de modelos para dos soluciones diferentes.

La comparativa de la Tabla 25 se establece frente al tiempo aplicado a la lectura e interpretación de los ficheros OBJ. Sin embargo, para ser completamente exactos, habría que comparar el tiempo medio obtenido con nuestro nuevo método frente al tiempo recuperado de la lectura de ficheros OBJ y cualquier otro cálculo derivado, pero en la Tabla 23 no sólo se incluían modelos OBJ en este segundo tiempo, sino también superficies planas. En cualquier caso, la mejora es bastante significativa.

Igualmente, esta prueba se puede ejecutar sobre la primera escena de la aplicación final, donde el número de vértices de los modelos OBJ crece hasta 3.338.346 y el de triángulos a 6.660.500 (Tabla 26).

Tiempo medio de respuesta de la carga de modelos	
Ejecución	**Tiempo de lectura de ficheros (ms)**
1	1.037,67
2	1.085,38
3	1.067,86
4	1.061,39
5	1.046,56
6	1.059,37
Tiempo medio (ms)	1.059,705

Tabla 26. Tiempo medio de respuesta de la carga de ficheros binarios en la aplicación final.

El resto de optimizaciones consideradas no afectan directamente a la carga de modelos, y como tal, se desarrolla en una sección diferente.

2.4.3 Introducción a los *compute shaders*

Buena parte de la documentación que resta de esta iteración se encuentra dedicada a la implementación de algoritmos paralelos en *compute shaders*, y esto mismo se aplica a iteraciones posteriores. Así, el objetivo de este apartado es introducir algunos conceptos básicos que puedan servir para el resto de la documentación.

Un *compute shader*, al igual que el resto de *shaders* aquí desarrollados, se describe mediante el lenguaje GLSL. A nivel de comportamiento, no existe diferencia alguna con un *vertex* o un *fragment shader*, más allá de su propósito. Ciertamente, hay variables *built-in* que difieren con estos dos últimos *shaders* y que son características de un contexto paralelo.

Los hilos que ejecutan el comportamiento programado se agrupan en lo que se conocen como *work groups*, los cuales a su vez forman un *dispatch*, que podrá ser 1D, 2D o 3D. Cada *work group* estará formado por un conjunto de hilos, teniendo en cuenta que este número se encuentra limitado, y comúnmente, el número máximo de *work groups* es mayor que el número máximo permitido de hebras por *work group*. Es decir, para asegurarnos de que se cubre un *buffer* de cierto tamaño es más seguro fijar el máximo número de hebras por *work group*, y a partir de este número calcular la cantidad de grupos que serán necesarios.

Un hilo no sólo se interpreta de manera local, sino que también existe un registro global. Por tanto, una variable *built-in* básica es **`gl_GlobalInvocation.x{yz}`**, lo que nos proporciona el identificador global. Por ejemplo, si el identificador de un hilo en un *dispatch* 1D excede el tamaño del *buffer*, podemos simplemente finalizar el *shader* (para este hilo). Otras variables comunes, aunque no tan empleadas en nuestros *shaders*, son `gl_LocalInvocationIndex`, `gl_LocalInvocationID.{x,y,z}` o `gl_WorkGroupSize.{x,y,z}`, sea la primera de ellas el resultado de la siguiente operación.

$$\begin{aligned} gl\_LocalInvocationIndex \\ = gl\_LocalInvocationID.z * gl\_WorkGroupSize.z \\ * gl\_WorkGroupSize.y + gl\_LocalInvocationID.y \\ * gl\_WorkGroupSize.x + gl\_LocalInvocationID.x \end{aligned} \tag{2.37}$$

A diferencia de un *vertex* y un *fragment shader*, no existen *input* y *output buffers* como tal, sino SSBOs, imágenes, y como siempre, *uniforms* y *samplers*. Sobre los SSBOs y las imágenes podremos leer y escribir, por lo que se debe interpretar como un *buffer* en GPU que podemos manipular libremente. En nuestra aplicación, se distinguen varias maneras de crear un SSBO, y todas ellas reciben el nombre de *read/write*, no necesariamente porque los *buffers* generados se limiten a alguna de estas operaciones, sino porque la forma de crearlos no es exactamente la misma. Cuando generamos un *buffer* de lectura, conocemos de antemano su contenido, y por tanto, es posible transferir a GPU el contenido del *buffer* indicando el tamaño del mismo. Cuando generamos un *buffer* que utilizaremos para escribir, no disponemos de una estructura en memoria, sino que sólo sabemos el tipo de dato y el tamaño del *buffer*. Una situación muy diferente es la vinculación de un *buffer*, en el que se escribe, pero necesita de un valor por defecto, en cuyo caso es posible introducirlo como si se tratara de un *buffer* de lectura. En cualquier caso, dentro de un *shader* podemos leer o escribir indistintamente de la manera en que se haya definido el *buffer*.

Por otro lado, la definición del tamaño de un *work group* suele darse en el propio *compute shader*, aunque es posible evitar esto e introducir dicho tamaño desde el exterior, tal y como sigue:

```
#extension GL_ARB_compute_variable_group_size: enable
layout (local_size_variable) in;
```

Por otro lado, la definición de un SSBO sigue la estructura de la siguiente línea:

```
layout (std430, binding = 0) buffer BufferName { Type t[]; };
```

donde *Type* podrá ser un tipo de variable definido en GLSL, tal como `vec3`, `vec4`, etc, o una estructura a medida. Además, para evitar definir múltiples veces una estructura compartida en varios *shaders*, podemos hacer uso de lo que conocíamos en el primer capítulo del desarrollo (Shader program) como librerías, líneas con una sintaxis muy específica que definían la ruta del fichero que debía integrarse.

A la hora de ejecutar un *shader*, donde se debe indicar el número de grupos en las tres dimensiones, y el tamaño de los mismos, se calculan todos estos valores como sigue:

$$numGroups = ceil\left(\frac{size_{array}}{MAX\ SIZE_{workGroup[axis]}}\right) \tag{2.38}$$

$$groupSize = MAX\ SIZE_{workGroup[axis]}$$

Como el límite de tamaño es mucho más estricto dentro de un grupo que en el *dispatch*, se fija el máximo número de hilos por grupo y se calcula cuántos grupos son necesarios. En el peor de los casos, habrá $MAX\ SIZE_{workGroup[axis]} - 1$ hilos que deberán finalizar sin realizar ningún cálculo (en 1D). En un *buffer* de millones de elementos, esta cantidad resulta insignificante. En nuestra tarjeta gráfica, el máximo tamaño de un grupo es 1536 (1D), 1024 (2D) y 64 (3D).

Por último, también cabe destacar la estructura que debe seguir un *buffer*, dado que el tamaño de cada elemento debe ser un múltiplo de 16 *bytes*, y por tanto, no existe gran libertad a la hora de elegir el orden o el tipo de atributos contenidos en la estructura. En el caso de que no se definan elementos de tal tamaño, el *shader* seguiría funcionando correctamente, pero al leer el *buffer* encontraríamos valores que no concuerdan necesariamente con lo que esperábamos, dado que *bytes* de diferentes elementos pueden interpretarse como si pertenecieran a un mismo elemento. Como referencia: 16 *bytes* son equivalentes a un vector de cuatro valores de 32 bits, por lo que deberíamos agrupar valores de 4 en 4 (en principio, se considera que *int*, *uint* o *float* tienen un mismo tamaño).

Cuando existen ciertos huecos que imposibilitan obtener un tamaño múltiplo de 16 *bytes*, lo que suele hacerse es incluir un *padding*, o bien expandir el tamaño de algún elemento. Por ejemplo, podríamos incluir un vector de cuatro componentes en lugar de uno de tres, siempre y cuando nos sea indiferente el uso de un tipo u otro.

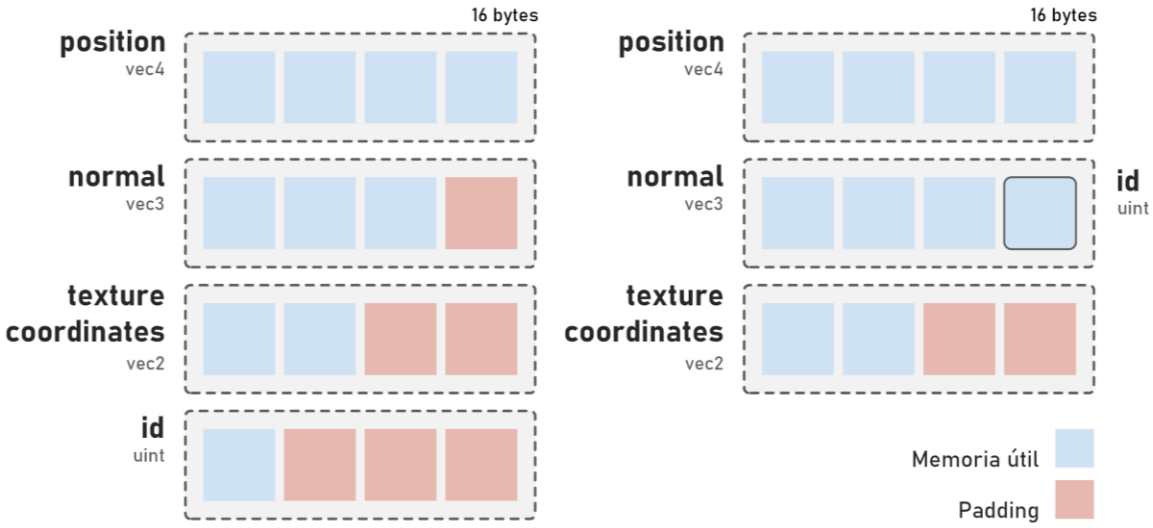


Ilustración 81. Representación del esquema que debe seguir una estructura para adaptar su tamaño al solicitado por la GPU. Propuesta de reubicación de un valor numérico para reducir la memoria empleada.

2.4.4 Optimización de cálculos sobre modelos OBJ

A partir del apartado previo se puede deducir que la optimización que se implementa sobre modelos OBJ sólo se aplica en aquellos casos en los que el fichero aún no se encuentra almacenado en formato binario. Por tanto, la optimización que aquí pueda desarrollarse no supone una mejora constante de la aplicación y además, es difícil de captar, en tanto que la generación de datos derivados suponía una cantidad ínfima de tiempo respecto de la carga del modelo OBJ.

Los procesos de cálculo que pueden optimizarse son los que se detallan a continuación.

2.4.4.1 Aplicación de matriz del modelo

Este primer cálculo se puede describir como la transformación de los vértices de un modelo mediante su matriz de modelado y la aplicación de la técnica de *displacement mapping*. Este proceso se puede implementar mediante un *computer shader* genérico para cualquier tipo de modelo, en tanto que sólo recibe como entrada la información de los vértices, la matriz de modelado, y la selección de una ruta de una subrutina dedicada a *displacement mapping*.

El porqué de esta etapa de cálculo se fundamenta en la eficiencia de todos los procesos que vienen después, dado que es normal corregir la situación de un modelo mediante una matriz de transformación. En este caso, resultaría muy costoso multiplicar constantemente una matriz y una posición cada vez que sea necesario obtener la posición real, o la normal del vértice. Con la introducción de la técnica de *displacement mapping* esta situación es aún menos factible, dado que además de la multiplicación sería necesario consultar una textura, y vincular la propia textura a un amplio número de *compute shaders*.

Crear varios SSBOs para cada tipo de dato de la geometría no es factible, por lo que toda esta información se agrupa bajo la estructura ***VertexGPUData***, la cual se describirá más tarde con detenimiento.

El comportamiento definido en este *shader* se muestra a continuación:

$$\begin{aligned} vData[tIndex]_{pos} &= vec3(matrix_{model} * vec4(vData[tIndex]_{pos}, 1)) \\ vData[tIndex]_{norm} &= vec3(matrix_{model} * vec4(vData[tIndex]_{norm}, 0)) \\ vData[tIndex]_{pos} &= displacementUniform(tIndex) \end{aligned} \tag{2.39}$$

A su vez, la subrutina contiene dos posibles rutas, sea una de ellas el retorno de la posición sin ningún tipo de modificación, $vData[tIndex]_{pos}$. La segunda ruta permite aplicar la técnica de *displacement mapping* tal y como se describe a continuación:

$$disp = texture(texDispSampler, vData[tIndex].textCoord)_r$$
$$result = vData[tIndex]_{pos} + vData[tIndex]_{normal} * disp * dispLength \quad (2.40)$$

Nótese como $disp$ se trata de un desplazamiento en el intervalo $[0,1]$, lo que permite controlar la longitud del desplazamiento mediante un parámetro $displacementLength$, aquí abreviado como $dispLength$.

Una vez ejecutado el *shader*, es posible recuperar la información resultante utilizando el identificador del *buffer* antes generado y el tipo de dato que contiene dicho *buffer*, lo que nos devolverá un puntero. La información referenciada mediante este puntero se podrá transferir a un vector para facilitar su visualización.

2.4.4.2 Cálculo de tangentes

La introducción de *displacement mapping* y *bump mapping* en la aplicación implica que es necesario obtener la tangente de cada vértice respecto de la superficie, dado que buena parte de la información de iluminación se transfiere al sistema de coordenadas formado por los vectores tangente, binormal y normal. Para calcular las tangentes es necesario inicializar tres SSBOs, dos de ellos para la geometría y la topología del modelo, y un tercer *buffer* que servirá de soporte. El SSBO de topología sólo contendrá en este punto la información básica, como el índice de aquellos vértices que conforman el triángulo, mientras que el tercer *buffer* se utiliza para realizar sumatorias afectadas por varios hilos.

Uno de los principales inconvenientes de un *compute shader*, pero también de otros *frameworks* de paralelismo en GPU, es la ausencia de operaciones atómicas de para valores flotantes. Una operación atómica es necesaria cuando varios hilos acceden y modifican un mismo elemento, y aunque se trata de operaciones más lentas, no siempre se pueden evitar. Como GLSL permite realizar operaciones atómicas sobre valores enteros, una solución que aquí se propone es la transformación de los valores flotantes en valores enteros utilizando un factor multiplicador, por ejemplo, 10.000. Este mismo factor se podrá utilizar en una segunda etapa para obtener el verdadero valor flotante. Evidentemente, en todo este proceso

puede perderse cierta precisión, pero este escenario es más deseable que la reserva de mucha más memoria para evitar colisiones.

En CPU, el cálculo de tangentes es muy fácil de llevar a la práctica (Lengyel, 2001), dado que la influencia de un triángulo en cada uno de sus vértices se calcula de manera iterativa. En GPU, necesitamos dos etapas y el vector de soporte antes comentado. Dicho vector será de tipo `uvec4`, donde los tres primeros componentes serán la tangente transformada, y el cuarto componente indica cuántos triángulos han colaborado en el cálculo. La propuesta de dos etapas es necesaria debido a que todos los triángulos deben calcular su contribución sobre sus vértices para poder iterar, posteriormente, por cada uno de los vértices de la malla y calcular su tangente. GLSL proporciona una herramienta, *barrier*, que permite llevar a cabo una espera hasta que todos los hilos alcancen cierto punto del programa, pero esta función sólo afecta a los hilos de un grupo (*work group*), mientras que lo que nosotros buscamos es una pausa global. Por esta razón, este algoritmo está compuesto por dos etapas, y la espera se realiza en CPU mediante `glMemoryBarrier`.

De manera muy breve, para no extender este apartado, habría que considerar T, la tangente, B, la bitangente, Q, un punto cualquiera dentro del triángulo, P_0, P_1, P_2, los tres vértices del triángulo, y $(u_0, v_0), (u_1, v_1), (u_2, v_2)$, las tres coordenadas de textura de P_0, P_1 y P_2. Se puede reformular la ecuación del triángulo como se muestra a continuación:

$$Q - P_0 = (u - u_0) * T + (v - v_0) * B \tag{2.41}$$

Se consideran además dos puntos, Q_1 y Q_2, que podrán calcularse como $P_1 - P_0$ y $P_2 - P_0$, respectivamente. Además, sus coordenadas de textura, (s_1, t_1) y (s_2, t_2), equivalen a $(u_1 - u_0, v_1 - v_0)$ y $(u_2 - u_0, v_2 - v_0)$. Por tanto, el sistema de ecuaciones que habrá que resolver es el siguiente:

$$\begin{bmatrix} Q_{1_x} & Q_{1_y} & Q_{1_z} \\ Q_{2_x} & Q_{2_y} & Q_{2_z} \end{bmatrix} = \begin{bmatrix} s_1 & t_1 \\ s_2 & t_2 \end{bmatrix} \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}$$
$$\begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix} = \frac{1}{s_1 t_2 - s_2 t_1} \begin{bmatrix} t_2 & -t_1 \\ -s_2 & s_1 \end{bmatrix} \begin{bmatrix} Q_{1_x} & Q_{1_y} & Q_{1_z} \\ Q_{2_x} & Q_{2_y} & Q_{2_z} \end{bmatrix} \tag{2.42}$$

La primera etapa del cálculo de tangentes consiste únicamente en la resolución de dicho sistema de ecuaciones, aunque no se almacena T_{j_i} ($i \in \{x, y, z\}$), sino $T_{j_i} * N_j$, sea N_j el número de triángulos que influyen en la tangente T_j. Nótese como el

sistema de ecuaciones también resuelve B_j, pero este vector no es en absoluto necesario para nuestros propósitos, y tampoco colabora en la segunda etapa del cálculo de tangentes.

En la Ilustración 82 se representa el pseudocódigo que permite resolver esta primera etapa en un *compute shader*. En esta imagen, las operaciones atómicas se resuelven mediante dos líneas que actualizan tanto la tangente como el número de triángulos considerados, aunque en la implementación estas dos líneas se convierten en doce (4 x 3), dado que este tipo de operaciones tampoco pueden aplicarse sobre vectores de valores enteros.

Algorithm 8 $ComputeTangents_{Stage_1}$: Primera etapa del cálculo de tangentes de una malla

Hilo de identificador $tIndex$
$index_i$ será el índice del vértice i de $triangle_{tIndex}$ en el buffer de geometría, $i \in 0, 1, 2$
v_i será la posición del vértice i
w_i será la coordenada de textura del vértice i

$Q_1 \leftarrow v_2 - v_1$
$Q_2 \leftarrow v_3 - v_1$
$s_1 \leftarrow w_{2x} - w_{1x}$
$t_1 \leftarrow w_{2y} - w_{1y}$
$s_2 \leftarrow w_{3x} - w_{1x}$
$t_2 \leftarrow w_{3y} - w_{1y}$

$r \leftarrow 1/(s_1 * t_2 - s_2 * t_1)$
$s \leftarrow vec3((t_2 * x_1 - t_1 * x_2) * r, (t_2 * y_1 - t_1 * y_2) * r, (t_2 * z_1 - t_1 * z_2) * r)$
$result \leftarrow (s + M) * M$

Operaciones atómicas
$atomicAdd(vTangent[index_i]_i, result)$
$atomicAdd(vTangent[index_i]_w, 1)$

Ilustración 82. Pseudocódigo de la primera etapa del cálculo de tangentes en un compute shader.

Una vez completada la primera etapa, disponemos de la tangente en un punto (y en teoría, también dispondríamos de la bitangente). Por tanto, lo que nos queda realmente es construir una base ortogonal a partir de tres vectores linealmente independientes (a la tangente y la bitangente se añade la normal del punto, conocida

de antemano). Para ello se puede utilizar el algoritmo de ortogonalización de Grand-Schmidt (EMS Press, 2020), que de manera genérica se expresa como se muestra a continuación:

$$u_k = v_k - \sum_{j=1}^{k-1} \frac{\langle v_k, u_j \rangle}{\langle u_j, u_j \rangle} u_j \tag{2.43}$$

Nos basta con recuperar los dos primeros vectores, dado que en nuestro caso no es necesario obtener el vector bitangente, como bien se puede observar en la siguiente formulación.

$$u_1 = v_1 = tangente$$

$$u_2 = v_2 - \frac{v_2 u_1}{u_1 u_1} u_1 = tangente - n * \frac{\langle tangente, n \rangle}{\langle n, n \rangle} \tag{2.44}$$

$$u_2 = tangente_{ortogonal} = tangente - n * dot(tangente, n)$$

En esta formulación, $\langle v_1, v_2 \rangle$ es la proyección de v_1 en v_2, equivalente a $dot(v_1, v_2) * v_2 * magnitud_{v_2}$, y n es un vector normalizado, razón por la cual buena parte de esta formulación equivale a 1. De manera previa al cálculo de una nueva tangente, es necesario deshacer la transformación de la primera etapa, por la cual los valores flotantes pasaban a convertirse en valores enteros sin signo. Para comprender cómo deshacer esta transformación se expone un pequeño ejemplo:

$$sum_{int} = \left(t_{1_i} + M\right) * M + \left(t_{2_i} + M\right) * M = M\left(t_{1_i} + t_{2_i} + 2M\right)$$

$$sum_{float} = t_{1_i} + t_{2_i}$$

$$sum_{int} = M\left(sum_{float} + 2M\right) \tag{2.45}$$

$$sum_{float} = \frac{sum_{int}}{M} - 2M$$

En este caso, N_j es 2, pero puede identificarse como el valor que se encuentra en la cuarta coordenada del vector donde se guarda la transformación. Esta misma formulación se representa en el pseudocódigo de la Ilustración 83. A diferencia de antes, en esta segunda etapa el número de hilos necesarios equivale al número de vértices, mientras que en la primera etapa se trabajaba con cada uno de los triángulos de la malla.

Algorithm 9 $ComputeTangents_{Stage_2}$: Segunda etapa del cálculo de tangentes de una malla

1: Hilo de identificador $tIndex$
2: $n_{triangles} \leftarrow vertexTangent[index].w$
3: $n \leftarrow vertexData[tIndex].normal$
4: $t \leftarrow vTangent[tIndex].xyz/M - M * n_{triangles}$
5: $vertexData[tIndex].tangent = normalize(t - n * dot(n, t))$

Ilustración 83. Pseudocódigo correspondiente a la segunda etapa del cálculo de tangentes en un compute shader.

2.4.4.3 Cálculo de información de la topología

A medida que se lee un fichero OBJ, se construyen instancias de una estructura relacionada con la topología, a la que hemos llamado ***FaceGPUData***, y que también se describe más tarde. Uno de los atributos de esta estructura es un vector de tres valores enteros sin signo, que almacenan los índices de los vértices que componen el triángulo. De hecho, así es como se define el fichero OBJ (rectificando el primer índice posible, que es uno en lugar de cero). Sin embargo, hay más datos relevantes acerca de la cara de una malla, como la normal, y por cuestiones que más tarde enteremos, su AABB.

Además, este *buffer* de instancias de *FaceGPUData* no se trata del *buffer* que permitirá *renderizar* un modelo, dado que este se compone únicamente de índices y de un valor numérico muy elevado que se utilizará para distinguir primitivas. Es cierto que podría haberse generado durante la carga del modelo OBJ, pero conocedores de la necesidad de ejecutar un *compute shader* para calcular más datos de la topología, se reduce al máximo el número de operaciones en CPU, y se desplazan el resto a GPU.

El *compute shader* que aquí se utiliza no es más que una combinación de operaciones por completar relacionadas con la propia malla, más que con los vértices.

Se definen así tres SSBOs: geometría (*VertexGPUData*; sólo lectura), topología (*FaceGPUData*; lectura y escritura) y un *buffer* de valores enteros sin signo (sólo escritura). El cálculo de la normal de una cara de la malla, así como el cálculo del AABB, se define mediante las siguientes líneas:

$$
\begin{aligned}
p_1 &= vertex_{data}[mesh_{data}[index].vertices_x].position \\
p_2 &= vertex_{data}[mesh_{data}[index].vertices_y].position \\
p_3 &= vertex_{data}[mesh_{data}[index].vertices_z].position \\
normal &= normalize\big((p_2 - p_1) \times (p_3 - p_1)\big) \\
point_{min} &= \min(p_1, \min(p_2, p_3)) \\
point_{max} &= \max(p_1, \max(p_2, p_3))
\end{aligned}
\tag{2.46}
$$

Por otro lado, el único punto de interés en la generación del *buffer* de índices de la malla es la asignación de su tamaño, dado que este debe ser cuatro veces mayor que el número de triángulos ($index_1, index_2, index_3, index_{restart}$). Por tanto, para un hilo de identificador global $index_{thread}$, habrá que acceder a las posiciones situadas en el intervalo $[index_{thread} * 4, index_{thread} * 4 + 3]$.

2.4.4.4 Tiempos de respuesta

Para finalizar este apartado centrado únicamente en la inicialización de modelos procedentes de ficheros OBJ, se muestra el tiempo de respuesta vinculado a todos aquellos cálculos derivados de la información que se recupera desde el fichero. Se excluye el tiempo de cálculo empleado en la definición de otros tipos de topologías que podían encontrarse en la versión final de la iteración dos. Partimos nuevamente del mismo escenario de modelos OBJ, formado por 300.973 vértices y 587.351 triángulos.

En este caso, el tiempo obtenido en la versión que emplea cálculos en GPU no mejora en absoluto la versión CPU, especialmente debido a la definición de la escena, dado que esta no está compuesta por un único modelo, sino por varios de ellos. Por tanto, para cada modelo es necesario inicializar sus correspondientes SSBOs, recuperar su contenido o destruirlos, lo que hace que la versión a priori optimizada no mejore el tiempo de la versión básica. Una de las posibles modificaciones del proyecto podría ser la inclusión de una escena elaborada en una herramienta externa, reduciendo así la carga de trabajo y evitando cambios de contexto constantes.

Para comprobar esto es posible hallar estos mismos tiempos para una única escena, compuesta por un modelo que conste de millones de triángulos. Para ello podemos acudir a cualquier escena empleada en trabajos de investigación, como San

Miguel. Para medir los tiempos de la Tabla 28 se emplea una versión simplificada de esta escena (Ilustración 84), que consta de 3.738.829 vértices y 4.963.199 triángulos.

En cualquier caso, esta transición de algoritmos de CPU a GPU para modelos OBJ se podría haber evitado, dado que no representa ventaja o desventaja alguna en el rendimiento de esta aplicación, al menos si la escena se define tal y como se hace hasta este punto.

Tiempo medio de respuesta del cálculo de información derivada (escena inicial)				
Ejecución	**Tiempo de respuesta en CPU (ms)**		**Tiempo de respuesta en GPU (ms)**	
1	36,165		39,768	
2	33,827		35,191	
3	35,024		39,156	
4	32,232		39,282	
5	33,114		37,216	
	Tiempo medio (ms)	34,07	**Tiempo medio (ms)**	38,12

Tabla 27. Comparativa, empleando dos versiones diferentes, del tiempo medio de respuesta obtenido en el cálculo de información derivada de modelos OBJ.

Ilustración 84. Rendering de la escena San Miguel en la aplicación Visor3D de Microsoft Windows.

Tiempo medio de respuesta del cálculo de información derivada (San Miguel)			
Ejecución	**Tiempo de respuesta en CPU (ms)**	**Tiempo de respuesta en GPU (ms)**	
1	391,434	145,135	
2	393,979	164,258	
3	412,992	187,755	
4	393,828	191,622	
5	402,152	154,458	
	Tiempo medio (ms) 398,877	**Tiempo medio (ms)**	168,64

Tabla 28. Tiempo medio de respuesta del cálculo de información derivada de modelos OBJ. En este caso se emplea un único modelo (San Miguel).

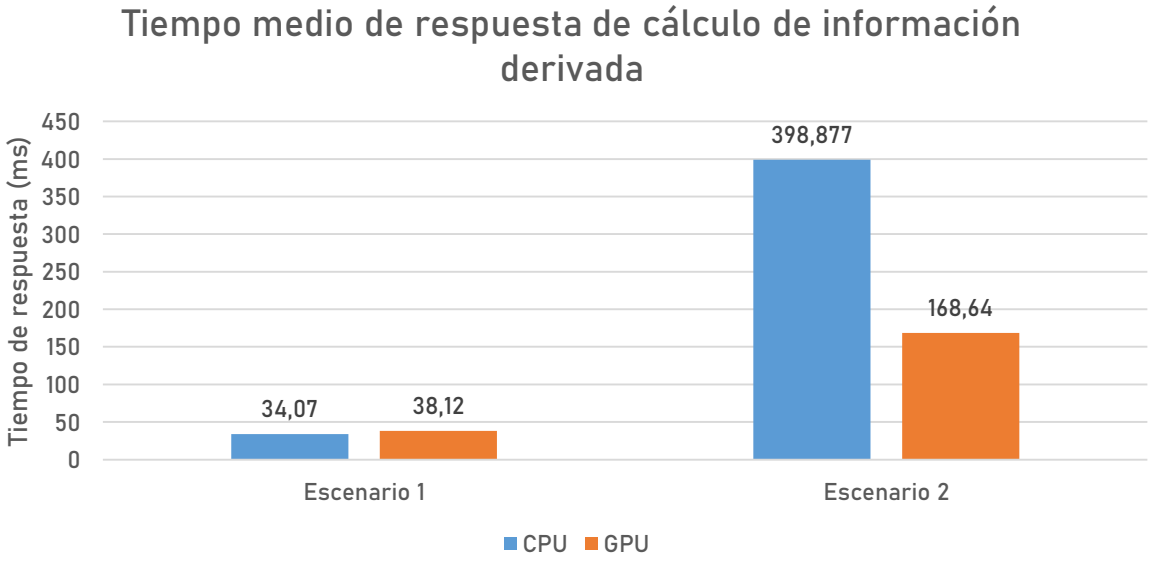


Ilustración 85. Representación mediante un diagrama de barras del tiempo medio obtenido en el cálculo de información derivada de modelos OBJ, tanto en la versión CPU como en GPU.

2.4.5 Optimización de cálculos en superficies planas

La segunda subclase de un modelo 3D es una superficie plana, donde toma mucho más sentido la optimización de su generación, dado que este tipo de objeto no se almacena en un fichero, sino que se construye siempre que se inicia la aplicación. Su proceso de construcción se divide en dos fases: generación de geometría y parte de la topología, y generación del resto de topología.

2.4.5.1 Generación de geometría (y topología)

El primer *compute shader* que se implementa aúna la construcción de una malla que define la superficie plana, y el comportamiento que antes se describió para transformar los vértices en una posición final. Un modelo OBJ suele modificarse únicamente mediante transformaciones básicas (traslación, rotación y escalado), mientras que la aplicación de *displacement mapping* no es tan frecuente. En una superficie plana ocurre exactamente lo contrario; la flexibilidad de su construcción es elevada, y no suele ser frecuente modificar el tamaño o la posición del plano (salvo una traslación en el eje Y), pero la aplicación de *displacement mapping* supone un amplio número de oportunidades. Por ejemplo, sería posible crear un terreno partiendo de una superficie como esta, y un mapa de altura.

Antes de comenzar, cabe mencionar los parámetros por los que se rige la generación de un plano: 1) anchura y profundidad en el espacio, 2) número de divisiones horizontales y verticales, y 3) coordenadas de textura aplicables a la esquina superior derecha del plano. Además, este se construye centrado en el origen del sistema de coordenadas.

A diferencia del modelo OBJ, el problema que aquí se plantea es idóneo para utilizar una malla 2D de grupos de hilos. Por tanto, en la ejecución habrá que indicar el número de grupos e hilos deseados en X e Y, mientras que Z permanecerá como 1. Antes de iniciar la ejecución se precalcula el tamaño de cada celda de la matriz ($tile_{width}, tile_{depth}$), y el tamaño de la mitad exacta del plano, ($width_2, depth_2$), para trasladar el plano al origen de coordenadas. Al final de la ejecución de este primer *shader* se espera obtener un *buffer* de geometría (*VertexGPUData*), y un *buffer* de índices que indican cómo renderizar el plano. Por tanto, son necesarios dos SSBOs que no necesitan de una inicialización (se define el espacio y más tarde se leen). También habrá que considerar que el número de divisiones en ambos ejes viene dado por $tiling_h$ y $tiling_v$, pero son necesarios más hilos, concretamente, $(tiling_h + 1) * (tiling_v + 1)$ (es necesario cubrir el último punto de cada fila de la malla, el cual no precede a ninguna celda).

El cálculo de la geometría de la malla para un hilo de identificador 2D $(index_x, index_y)$ se define mediante la siguiente formulación:

$$
\begin{aligned}
position &= vec3\left(index_x * tile_{width} - width_2, 0, index_y * tile_{depth} - depth_2\right) \\
normal &= vec3\left(matrix_{model} * vec4(0,1,0,0)\right) \\
tangent &= vec3\left(matrix_{model} * vec4(1,0,0,0)\right) \\
textCoord &= vec2(\frac{index_x}{tiling_h} * maxTextCoord_x, maxTextCoord_y - \frac{index_y}{tiling_v} \\
&\quad * maxTextCoord_y) \\
position &= vec3(matrix_{model} * vec4(displacementUniform(id_{global}), 1))
\end{aligned}
\tag{2.47}
$$

Nótese como el *dispatch* de la ejecución es 2D, pero los SSBOs donde se almacenan los datos no lo son. Por tanto, es necesario calcular id_{global} como $(grid_x + 1) * index_y + index_x$ para poder almacenar toda la información relacionada con la geometría de un punto. La técnica de *displacement mapping* se aplica en último lugar debido a que es necesario utilizar una posición y unas coordenadas de textura que deben calcularse previamente.

Para generar la topología, y más concretamente, la información vinculada a la estructura *FaceGPUData*, es necesario disponer de la información de todos los vértices, dado que debe calcularse la normal o el AABB de una cara (de momento no es posible). Sin embargo, la construcción de un *buffer* de índices para el *rendering* no necesita ningún tipo de conocimiento acerca de los vértices, más allá de sus posiciones en el *buffer* de geometría. En lugar de definir el plano como un conjunto de triángulos, es posible construirlo como un conjunto de tiras de triángulos (*triangle strip*) (Ilustración 86). Más concretamente, las tiras serán las columnas del plano. Esto produce un ahorro significativo de memoria debido a que los valores numéricos que indican el fin de una primitiva ahora son muchos más escasos, teniendo en cuenta que una tira estará formada por $(tiling_v + 1) * 2$ triángulos. El ahorro de memoria será más significativo a medida que aumenta la subdivisión del plano.

La implementación de este fragmento del *compute shader* se muestra en el pseudocódigo de la Ilustración 87. El cálculo más importante es la posición donde se almacenarán estos índices, $rawMeshPosition$, que se obtiene a partir de $\left(grid_y + 1\right) * 2 + 1$ (número de índices en una tira, donde el último 1 pertenece al índice de reinicio de la primitiva), $index_x$ (desplazamiento horizontal; número de tiras antes definidas), e $index_y * 2$ (*offset* en la tira actual).

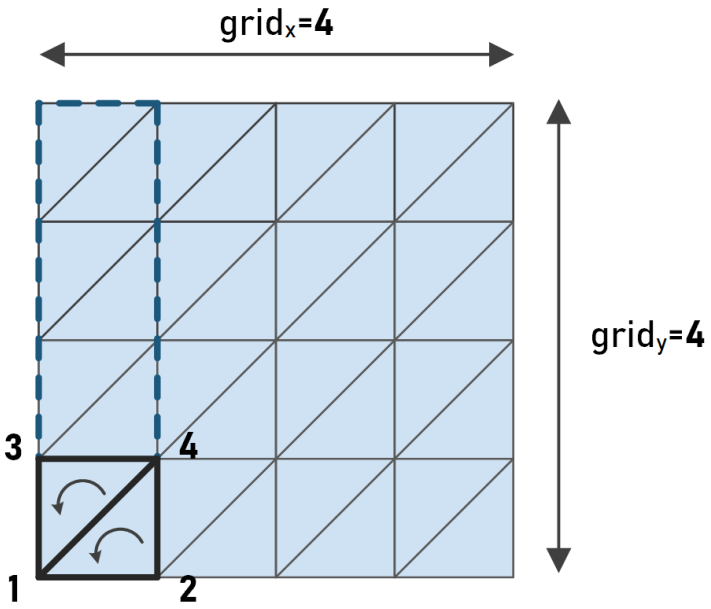


Ilustración 86. Representación de un plano de 4 x 4 subdivisiones, donde se destaca el concepto de triangle strip y la definición de dos triángulos en cada misma celda.

Algorithm 10 $ComputePlanarSurfaceIndices$: Definición de índices de la topología de un plano

$index_1 \leftarrow index_y * (grid_x + 1) + index_x$
$index_2 \leftarrow index_y * (grid_x + 1) + index_x + 1$

$rawMeshPosition \leftarrow index_x * ((grid_y + 1) * 2 + 1) + index_y * 2)$
$rawMeshData[rawMeshPosition] \leftarrow index_2$
$rawMeshData[rawMeshPosition + 1] \leftarrow index_1$

if $index_y == grid_y - 1$ **then**
    $index_3 \leftarrow (index_y + 1) * (grid_x + 1) + index_x$
    $index_4 \leftarrow (index_y + 1) * (grid_x + 1) + index_x + 1$

    $rawMeshData[rawMeshPosition + 2] \leftarrow index_4$
    $rawMeshData[rawMeshPosition + 3] \leftarrow index_3$
    $rawMeshData[rawMeshPosition + 4] \leftarrow restartPrimitive$
end if

Ilustración 87. Pseudocódigo de la definición de los índices de un plano empleando como primitiva una tira de triángulos.

2.4.5.2 Cálculo de información de la topología

El siguiente *compute shader* implementado tiene como finalidad completar la información restante de la topología. A diferencia del *shader* anterior, se sustituye un

dispatch 2D por 1D, de tal manera que sólo tenemos conocimiento del número de triángulos que deben generarse así como del número de divisiones de la malla. También podría haberse empleado un *dispatch* 2D, aunque si nos fijamos en el último *shader*, no se aprovechan las características de dicha estructura, e igualmente, es necesario acceder a un SSBO definido en una única dimensión. Por tanto, nos es indiferente la estructura del *dispatch* en cuanto a eficiencia del programa. La fila y la columna asociada a un hilo de identificador $tIndex$ se puede hallar como $floor(\frac{tIndex}{grid_y})$ y $tIndex \,\%\, grid_x$.

Los vértices que conforman los dos triángulos de los que se ocupa un hilo cualquiera se definen por los siguientes índices:

$$\begin{aligned} index_1 &= row * (grid_x + 1) + column \\ index_2 &= row * (grid_x + 1) + column + 1 \\ index_3 &= (row + 1) * (grid_x + 1) + column \\ index_4 &= (row + 1) * (grid_x + 1) + column + 1 \end{aligned} \tag{2.48}$$

Una vez calculados los índices que conforman los triángulos se puede calcular la normal y el AABB, tal y como se mostró para el modelo OBJ. Igualmente, es necesario rectificar las tangentes mediante el mismo algoritmo que antes se describía, razón por la cual se implementa bajo *Model3D* y no sobre cualquier subclase.

2.4.5.3 Tiempo de respuesta

Se propone lanzar un conjunto de pruebas para comprobar la mejora obtenida mediante la generación de un plano en GPU. El objetivo será construir varios planos de diferente complejidad. Nos desprendemos momentáneamente de la escena que veníamos usando, y nos centraremos en una única superficie plana. El tiempo de construcción en CPU se recupera de la aplicación que se obtiene al final de la segunda iteración.

En la medición del tiempo no sólo se incluye el tiempo de cálculo en GPU, sino también la definición y recuperación de *buffers*. En cualquier otro caso, el tiempo obtenido no sería realista; cuando elegimos el algoritmo paralelo debemos saber que existe cierta carga en CPU derivada de las dos operaciones citadas.

Tiempo medio de respuesta de la construcción de un plano (100 x 100)				
Ejecución	**Tiempo de construcción en CPU (ms)**		**Tiempo de construcción en GPU (ms)**	
1	4,605		5,596	
2	4,109		5,783	
3	4,251		5,471	
4	4,732		5,466	
5	4,389		5,184	
	Tiempo medio (ms)	4,4172	**Tiempo medio (ms)**	5,5

Tabla 29. Comparativa de tiempo medio de respuesta para la generación de un plano con 10.000 subdivisiones tanto en CPU como en GPU.

Tiempo medio de respuesta de la construcción de un plano (500 x 500)				
Ejecución	**Tiempo de construcción en CPU (ms)**		**Tiempo de construcción en GPU (ms)**	
1	161,548		45,550	
2	160,980		43,565	
3	159,617		43,062	
4	159,111		43,051	
5	167,899		42,623	
	Tiempo medio (ms)	161,831	**Tiempo medio (ms)**	43,57

Tabla 30. Comparativa de tiempo medio de respuesta para la generación de un plano con 250.000 subdivisiones tanto en CPU como en GPU.

Tiempo medio de respuesta de la construcción de un plano (1000 x 1000)				
Ejecución	**Tiempo de construcción en CPU (ms)**		**Tiempo de construcción en GPU (ms)**	
1	652,109		155,079	
2	610,039		153,152	
3	639,589		159,371	
4	620,732		152,784	
5	622,019		153,802	
	Tiempo medio (ms)	628,89	**Tiempo medio (ms)**	154,755

Tabla 31. Comparativa de tiempo medio de respuesta para la generación de un plano con 1.000.000 subdivisiones tanto en CPU como en GPU.

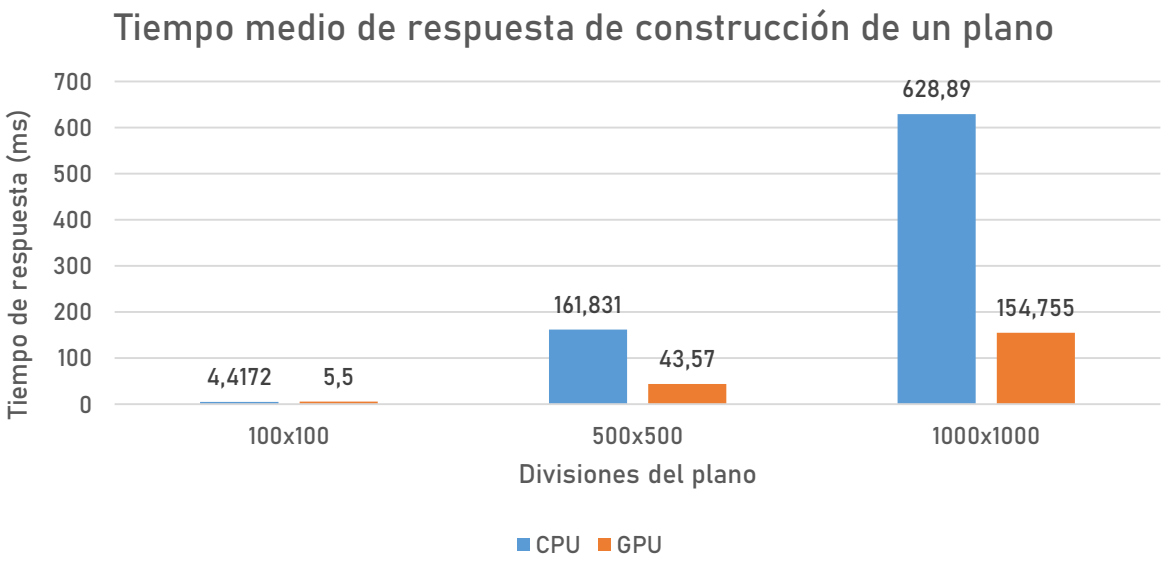


Ilustración 88. Diagrama de barras donde se representa el tiempo medio obtenido en la construcción de un plano, tanto en la versión CPU como en GPU.

El tiempo medio obtenido en cada prueba permite observar algo muy interesante, y es que la solución paralela no siempre es la más eficiente en todos los escenarios posibles. En nuestra aplicación, la ejecución de un *compute shader* requiere cierta preparación (definición de SSBOs, cálculo de número de grupos necesarios, etc) así como una etapa de recuperación de datos una vez finalizada la ejecución. Por tanto, hay escenarios en los no se calcula una cantidad tan elevada de datos, y en dichos casos, es difícil hallar una mejora en la versión paralela. En nuestra aplicación sólo se conservan los algoritmos paralelos de generación de un plano, debido a que se ha considerado que la distancia existente entre la versión CPU y GPU no es tan elevada con subdivisiones más pequeñas, pero también podría haberse establecido un umbral debajo del cual se prefiere la versión CPU.

Para finalizar, cabe mencionar que el tiempo medido para el algoritmo de GPU también incluye operaciones en CPU, de tal manera que los resultados mostrados representan el verdadero tiempo de ejecución de la versión alternativa completa. Aún así, es posible recuperar el tiempo empleado en la ejecución del *compute shader* para todos y cada uno de los hilos lanzados, a lo cual hemos llamado tiempo de ejecución en el *kernel*. Para medir este tiempo es posible recurrir a órdenes de OpenGL que inician o finalizan una consulta, sea una de ellas la consulta de tiempo. El comienzo de la consulta se define mediante las siguientes líneas:

```
GLuint query;
GLuint64 elapsedTime;
int done = 0;
glGenQueries(1, &query);
glBeginQuery(GL_TIME_ELAPSED, query);
```

Igualmente, es posible finalizar la consulta y recuperar el tiempo medido en nanosegundos, como bien se muestra a continuación, aunque esta no sería la única manera de resolver la necesidad que aquí se plantea (Lighthouse3d, 2011):

```
glEndQuery(GL_TIME_ELAPSED);
while (!done) {
      glGetQueryObjectiv(query, GL_QUERY_RESULT_AVAILABLE, &done);
}

glGetQueryObjectui64v(query, GL_QUERY_RESULT, &elapsedTime);
```

Midiendo únicamente el tiempo de ejecución en el *kernel*, se obtienen los resultados de la Tabla 32 y la Tabla 33. Los tiempos obtenidos son muchos más reducidos que los antes mostrados, y por tanto, se puede extraer que buena parte de la carga (inevitable) sigue perteneciendo a operaciones ejecutadas en CPU.

Tiempo medio de respuesta en el kernel (100 x 100)	
Ejecución	**Tiempo de ejecución en kernel (ms)**
1	0,1781
2	0,1966
3	0,1976
4	0,1587
5	0,17
Tiempo medio (ms)	0,1802

Tabla 32. Tiempo medio de respuesta para 10.000 subdivisiones, considerando únicamente el tiempo empleado en el compute shader.

Tiempo medio de respuesta en el kernel (1000 x 1000)	
Ejecución	**Tiempo de ejecución en kernel (ms)**
1	4,18
2	4,184
3	4,2874
4	4,1257
5	4,1472
Tiempo medio (ms)	4,184

Tabla 33. Tiempo medio de respuesta para 1.000.000 subdivisiones, considerando únicamente el tiempo empleado en el compute shader.

2.4.6 Generación de estructura de datos en GPU

El siguiente punto relevante que nos ocupa es la inicialización de una estructura de datos en GPU, y para seleccionar una podemos hacer extensiva a este apartado la descripción de estructuras de datos que se hizo en Desarrollo de solución. A partir de dicha descripción se extraía que un BVH era quizás la estructura más apropiada para nuestro problema, a pesar de que finalmente fue otra estructura la seleccionada. Sin embargo, cuando nos trasladamos a GPU no existen dudas acerca la estructura que debe emplearse, dado que un BVH es la estructura más utilizada en la actualidad en aplicaciones de *ray-tracing*, dado que permite resolver las intersecciones en un tiempo muy reducido. Este mismo problema es análogo a nuestra aplicación, donde podríamos hablar más bien de *ray-casting*. La ventaja de utilizar una estructura muy eficiente para resolver un problema tan ampliamente tratado como el *ray-tracing*, es la existencia de una investigación activa en torno a la generación de la estructura de datos en GPU, así como la optimización de otras operaciones, tales como un algoritmo de *traversal*. Por tanto, disponemos de algoritmos actualizados recientemente y con los que podríamos comparar el tiempo de respuesta obtenido.

Uno de los algoritmos más recientes acerca de la construcción de un BVH se describe en (Meister and Bittner 2018), aunque durante la ejecución del trabajo surgen otros algoritmos muy similares y más recientes (Hu et al. 2019). En cualquier caso, será el primer artículo citado el que finalmente se implemente. Como bien se destacó en el estado del arte de este trabajo, lo que se busca con este tipo de estructuras no es el mínimo tiempo de construcción (para esto siempre podemos recurrir a (Apetrei

2014)), sino generar una estructura de suficiente calidad para minimizar el tiempo de búsqueda de la intersección de un rayo con la escena. Dicha calidad se mide mediante una métrica, SAH (*Surface Area Heuristic*), que también se considerará en la construcción de la estructura.

El algoritmo completo consta de múltiples etapas que iremos desarrollando en este apartado, además de todas las funciones que se hayan considerado necesarias. En muy pocas palabras, el funcionamiento del algoritmo radica en ordenar espacialmente todas las primitivas de la escena, que serán triángulos, en función de lo que se conocen como *Morton codes*, y a partir de dicha ordenación podemos agrupar primitivas en base a una heurística hasta alcanzar un único AABB, que será la raíz del BVH.

2.4.6.1 Definición de estructuras en GPU

La primera sección de este apartado se dedica a describir aquellas estructuras que se definen tanto en CPU como en GPU para almacenar geometría y topología de modelos, así como cualquier otro tipo de dato. Este punto de la documentación quizás no es el idóneo para tratar todas las estructuras, debido a que no hemos visto cómo trabaja en profundidad el algoritmo de construcción, pero puede merecerar la pena este esfuerzo para que todos los conceptos queden claros en un único apartado.

Así, la primera estructura definida engloba la geometría de la escena. Un punto de la escena se contiene en ***VertexGPUData***, donde sus atributos son una posición 3D en el espacio, su normal, tangente y coordenada de textura. Todos estos atributos necesitan de *paddings* con los que alinear una instancia de esta estructura con el espacio requerido (múltiplo de 16 *bytes*). Así, a la posición se le añade un valor flotante, y a la coordenada de textura un vector con dos valores numéricos.

Más allá de cómputos en GPU, esta estructura tiene una gran influencia en el resto de la aplicación, dado que es la forma más fácil de definir los vértices de una malla, y por tanto, también se puede utilizar en GPU para *renderizar* un modelo. Se puede inicializar un VAO indicando que se utilizará la estructura *VertexGPUData*, lo que implica el uso de un *interleaved* VBO, aunque también es posible acudir al esquema anterior, donde hasta 4 VBOs eran necesarios.

Cada triángulo de la escena se define por una instancia de la estructura ***FaceGPUData***, donde se almacena la posición de los vértices del triángulo en el *buffer* de geometría, el mínimo y máximo punto que definen su AABB, y una normal del

triángulo. En realidad, salvo el primer atributo (índices de vértices), el resto son calculables a partir del primero. Si se almacenan es por una cuestión de reducicción del número de cálculos. Igualmente, también se almacena el identificador del componente al que pertenece el triángulo, donde el componente a su vez pertenece a un modelo. Algunos de los atributos que se almacenan son vectores de 3 valores, mientras que el identificador representa un único valor, y por tanto, es posible utilizar este en la posición que ocuparía un *padding*, y así reducir la memoria necesaria.

Un triángulo pertenece a su vez a una malla, que se engloba bajo ***MeshGPUData***. Para comprender esta clase es necesario tener en cuenta que en la construcción de la estructura de datos de la escena no disponemos de un conjunto de modelos o componentes, sino de millones de triángulos. Inicialmente, todos los triángulos se encuentran ordenados por modelo (después de una simple operación de *insert* de todos los triángulos de todos los componentes en un mismo *buffer*), pero tras ordenar las primitivas espacialmente, esta ordenación se rompe. Aún así, un triángulo almacena el identificador de su componente. Esta ordenación no se rompe en el caso del *buffer* de geometría, pero dicho *buffer* está formado por millones de vértices de los que no sabemos a qué modelo pertenecen. Por tanto, *MeshGPUData* almacena una posición de inicio y un número de vértices, que indican dónde comenzar a leer en el *buffer* de geometría y cuántos elementos deben leerse.

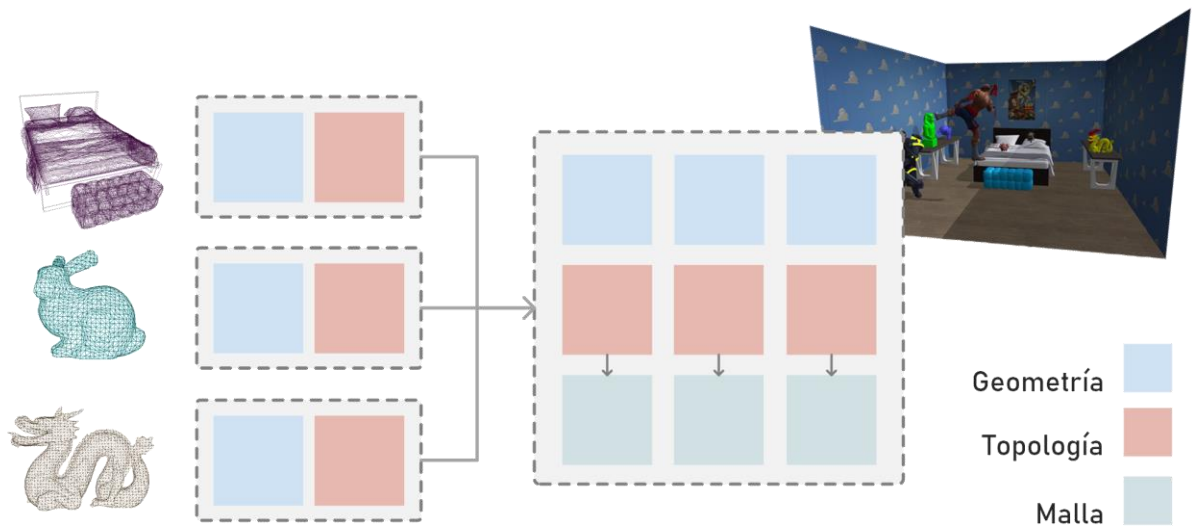


Ilustración 89. Esquema de compactación de estructuras de datos de los modelos de una escena.

Además de estos dos atributos, una malla también almacena otras propiedades vinculadas a la superficie del componente: opacidad, brillo, reflectancia, tipo de superficie, etc.

Un BVH se define como un conjunto de AABBs, los cuales se almacenan en un ***BVHCluster***, definido por una posición mínima y máxima, y la posición del triángulo que almacena. Nótese como el *buffer* de triángulos sólo ve modifica la posición de las primitivas contenidas durante la ordenación. Por tanto, es posible almacenar la posición que ocupa un triángulo durante la construcción de la estructura (las modificaciones ocurren de manera previa).

Hasta dos estructuras más son necesarias en la aplicación, rayo y colisión con triángulo, pero por ahora es suficiente con las que hemos visto.

2.4.6.2 Preparación para la construcción de un BVH

La entidad que reúne todo el proceso de construcción es ***Group3D***, dado que la raíz de la escena es el nodo que tiene conocimiento de todos y cada uno de los modelos contenidos. Por tanto, aquí se contienen todos los métodos y atributos relacionados con el proceso de construcción.

El primer paso previo a la construcción es la asignación de un identificador global a cada componente de un modelo, el cual sólo indica la posición en la que se ha registrado. Es decir, el identificador se mantiene invariante a lo largo de las ejecuciones, y aunque simple, cumple con su función de identificar de manera única a un componente.

Tras registrar cada componente, se debe cargar recursivamente la escena, comenzando por el nodo raíz, lo que implica la inicialización de cualquier modelo OBJ, plano o figura contenida. Durante el proceso de carga, un componente debe inicializar los *buffers* locales de geometría y topología (*VertexGPUData* y *FaceGPUData*). También es importante vincular cada una de las caras con el identificador previamente asignado al componente al que pertenece. Esto último se desarrolla en CPU, aunque bien podría trasladarse a GPU en escenas muy complejas.

Una vez construida toda la escena, es necesario agruparla en un único *buffer* que representa el grupo completo. Mediante el registro global de componentes es posible acceder a cada uno de ellos, consultar el número de vértices y triángulos, crear vectores con el tamaño recuperado previamente, e iterar por cada modelo para insertar su geometría y topología al final de nuestros vectores. A medida que se itera

por dichos modelos es posible construir el resumen de una malla, compuesta como bien sabemos por la posición donde comienza su geometría, y el número de vértices que forman parte de ella.

Tras iterar por cada componente, se lleva a cabo una operación de liberación de memoria que nos permitirá reducir en gran medida la cantidad de memoria asignada al proceso, a cambio de desprendernos de muchos de los datos que contenía un componente. Esto enfoque no es válido cuando el escenario propuesto presenta animaciones. Si es estático, sólo es necesario iniciar la construcción del BVH una única vez.

La liberación de memoria no se produce mediante mecanismos tan básicos como la eliminación de todos los elementos de un vector (`clear`), sino que una posible manera de solventar esto es intercambiar memoria (`swap`) entre un vector actual del componente y un *vector r-value* que no se almacena en ningún punto de la aplicación. De esta manera, el contenido que queremos liberar se mueve hacia un vector que será destruido inmediatamente.

2.4.6.3 Construcción de BVH

2.4.6.3.1 Cálculo de información de entrada

En este punto de la implementación podemos suponer que disponemos de al menos tres vectores que describen toda la escena: geometría, *VertexGPUData*, topología, *FaceGPUData*, y descripciones de mallas de triángulos, *MeshGPUData*.

Los conceptos de la construcción de un BVH mediante un algoritmo *down-top* como (Meister and Bittner 2018) son sencillos de comprender. El proceso comienza con un conjunto de AABBs, donde están contenidos los triángulos de la escena, y a partir de aquí se deben agrupar AABBs, de dos en dos, hasta alcanzar la raíz. A la hora de agrupar dos nodos es interesante que estos se encuentren muy cerca en el espacio, y dependiendo del algoritmo, que sus tamaños sean similares. Si se encuentran muy distantes, las posibilidades de que un rayo intersecte con la caja envolvente es mayor, y por tanto, se extiende en el tiempo el recorrido de la estructura (Karras, Thinking Parallel, Part III: Tree Construction on the GPU, 2012).

El primer objetivo de este apartado será reordenar las primitivas, de tal manera que aquellas que se encuentren muy cerca en el espacio 3D, también deben situarse en posiciones muy cercanas en el vector. Aunque sería posible trabajar con la posición

3D de un punto representativo del triángulo, es mucho más fácil representar dicha posición con un único valor, conocido como *Morton code*. Estos puntos representativos del espacio, que se deben comprender más bien como un conjunto de valores binarios, se pueden ordenar, formando lo que se conoce como *Morton curve*. A lo largo de esta curva, encontraremos triángulos muy cercanos en el espacio, logrando justamente lo que se buscaba.

La entrada de una función que calcula el código de Morton es una posición normalizada en $[0,1]$ (en X, Y y Z). Aunque es cierto que podría generarse una posición normalizada en base a unas fronteras manualmente definidas que engloben la escena con más o menos precisión, la solución idónea consiste en calcular el AABB de la escena, algo que ya se hacía en CPU, pero que igualmente podría calcularse en GPU. Un algoritmo para calcular el AABB podría seguir una estructura arbórea, de manera similar al algoritmo de construcción del BVH (salvando las diferencias), dado que podríamos combinar dos AABBs hasta alcanzar una única envolvente, la raíz. Al no disponer de un mecanismo de sincronización global, cada nivel del árbol representa una nueva ejecución de un *compute shader* dedicado a esta tarea.

La implementación de este algoritmo se sustenta sobre el funcionamiento del algoritmo de *prefix scan* que más tarde veremos, lo que nos permite reutilizar parte de la implementación. Nótese como no es necesaria ninguna ordenación previa de las primitivas, dado que al final se alcanza el mismo resultado. La preparación de este cálculo consiste en la inicialización de un SSBO que integra todos los triángulos de la escena (*FaceGPUData*). Además, es necesario conocer cuántas iteraciones son necesarias, las cuales se reducen al cálculo de $ceil(\log_2 N)$, sea N el número de triángulos, dado que en cada paso nos desprendemos de la mitad de AABBs que aún quedaban por agrupar. En cada iteración, el número de hilos necesarios es $ceil(\frac{K}{2})$, sea K el número de AABBs restantes. Es por ello que este último valor no se calcula desde un inicio, sino dentro de cada iteración. Para evitar reservar aún más memoria es posible modificar el SSBO inicial de triángulos, de tal modo que se actualizan sus AABBs, y en cualquier caso, disponemos de los triángulos con su verdadero AABB en memoria, por lo que podríamos generar otro SSBO.

Teniendo en cuenta todas las variables que se han mencionado, y que un hilo conoce el identificador de la iteración actual, $iteration$, podemos actualizar el AABB del segundo triángulo de cada pareja, tal y como se propone en la Ilustración 90.

Algorithm 11 *ComputeAABB*: Cálculo de AABB de dos nodos cualesquiera en un hilo de GPU

1: Hilo de identificador $tIndex$
2:
3: $powCurrentIt \leftarrow 2^{iteration}$
4: $powNextIt \leftarrow 2^{iteration+1}$
5: $index_1 \leftarrow tIndex * powNextIt + powCurrentIt - 1$
6: $index_1 \leftarrow clamp(tIndex * powNextIt + powNextIt - 1, 0, N - 1)$
7:
8: **if** $index_1 == index_2$ **then**
9: Finalizar hilo
10: **end if**
11:
12: Calcular máximo y mínimo punto de $faceData[index_1]$ y $face[index_2]$
13: Asignar nuevo máximo y mínimo punto a $face[index_2]$
14:
15: $min_{global} \leftarrow face[index_2].min * (N_{threads} == 1)$
16: $max_{global} \leftarrow face[index_2].max * (N_{threads} == 1)$

Ilustración 90. Pseudocódigo del cálculo de un AABB para un hilo cualquiera en GPU.

A partir de este bloque de código se puede deducir que el *buffer* de triángulos inicial no se compacta en cada iteración, sino que conserva el mismo tamaño, y por tanto, es necesario calcular los índices a los que debe acceder un hilo para una iteración concreta. Así, en la primera ejecución de la Ilustración 91 se lanzan tres hilos ($ceil\left(\frac{5}{2}\right) = 3$), de tal manera que el último de ellos encontrará que $index_1 = index_2$, finalizando su ejecución. En la primera iteración, los saltos tendrán longitud uno (2^0), mientras que en la última iteración será cuatro (2^2). Además, se juega con la función *clamp* para evitar ciertas comprobaciones cuando el tamaño del vector no es una potencia de dos.

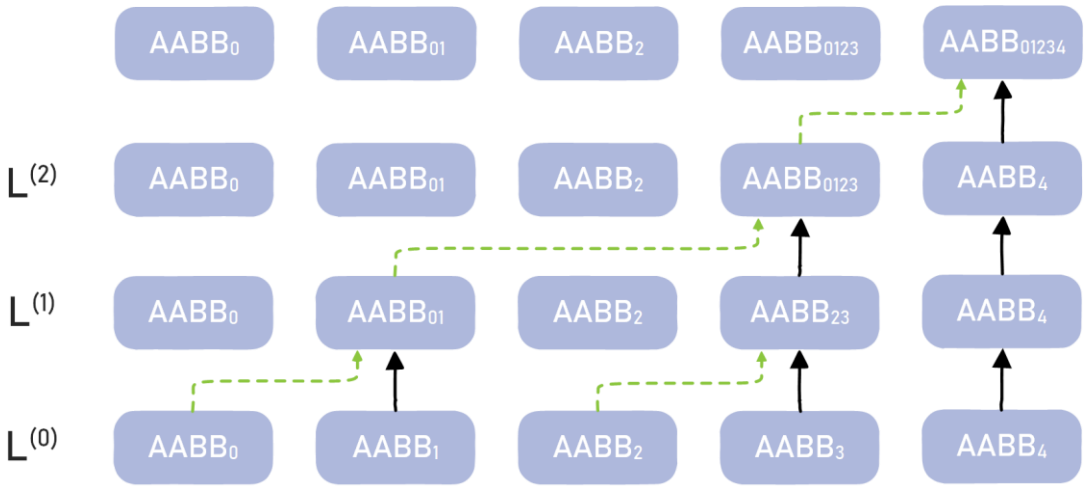


Ilustración 91. Flujo de índices consultados en el cálculo del AABB de una escena.

Aunque es cierto que el AABB de la escena se encuentra en la última iteración y en la última posición del *buffer*, no es factible recuperar este al completo desde GPU, dado que sólo nos interesa la última posición. Por esta misma razón, se incluyen max_{global} y $\min_{\text{global}}$, dos vectores de cuatro componentes que podemos recuperar de manera aislada. También es cierto que el acceso a ambos puntos no es atómico, por lo que su valor depende del orden de acceso, pero en la última iteración sólo existe un hilo, y por tanto, no tendrá que competir con ningún otro por acceder a ambas variables.

El tiempo medio de respuesta en el cálculo del AABB se muestra en la Tabla 34, teniendo en cuenta que no sólo se mide el tiempo de *kernel*, sino que también se incluyen todos los cálculos previos que permiten la ejecución del *shader*. Para obtener estos tiempos se emplea la primera escena de la aplicación final, compuesta, como bien sabemos, por 3.350.433 vértices y 6.683.902 triángulos.

Tiempo medio de respuesta de cálculo del AABB de una escena			
Ejecución	**Tiempo de respuesta en CPU (ms)**	**Tiempo de respuesta en kernel (ms)**	
1	39,828	12,0658	
2	36,914	11,1985	
3	37,756	12,1713	
4	38,665	11,5599	
5	36,496	11,1718	
	Tiempo medio (ms) 37,93	**Tiempo medio (ms)**	11,63

Tabla 34. Tiempo medio empleado en el cálculo del AABB de una escena compleja (6.683.902 triángulos).

Una vez se conoce el AABB de la escena, podríamos normalizar cualquier punto incluido en la misma de la manera más elegante posible. Por tanto, nos desplazamos al cálculo de los *Morton codes* que debemos ordenar más tarde. Este tipo de cálculo también se lleva a cabo en un *compute shader* por la mera razón de que disponemos de un gran número de triángulos, y para todos ellos habrá que calcular su código a partir de la posición en el espacio 3D. Partimos de un *buffer*, definido de manera externa, que traslada toda la información de triángulos en la escena a GPU. Además, es necesario un *buffer* adicional, de valores enteros sin signo (`uint`), donde se almacenará el *Morton code*.

La complejidad de este algoritmo no radica en la implementación, sino en la formulación, y como tal, se pretende describir a partir de las siguientes definiciones:

$$
\begin{aligned}
point_{norm} &= \frac{\frac{face.point_{min} + face.point_{max}}{2} - scene.point_{min}}{scene.point_{max} - scene.point_{min}} \\
point_{expanded} &= \text{clamp}(\text{point}_{\text{norm}} * 1024, 0, 1024) \\
point_{expanded_i} &= expandBits\left(point_{expanded_i}\right), \quad \forall i \in \{0, 1, 2\} \\
code_{morton} &= point_{expanded_x} * 4 + point_{expanded_y} * 2 + point_{expanded}
\end{aligned}
\tag{2.49}
$$

Nótese como la primera definición de $point_{expanded}$ es simplemente una expansión de un valor normalizado al intervalo $[0, 2^{10}]$, de tal manera que X, Y y Z ocuparían de manera individual 10 bits, y de manera conjunta, 30 bits (nos sobrarían 2 bits de un `uint`). Mediante la función `expandBits`, expresada a continuación, se pretende expandir, de nuevo, un valor representado en 10 bits para que ocupe los 30 bits, dado que cada coordenada se expresa mediante 10 bits, pero siempre se encuentran en las posiciones menos significativas. Para obtener esta última expansión se insertan dos ceros tras cada bit ($(1 + 2) * 10$). El objetivo final no es otro que intercalar los bits de cada coordenada (X, Y, Z) en un valor de 30 bits (Ilustración 92).

$$
\begin{aligned}
point_{expanded_i} &= (point_{expanded_i} * 0x00010001u) \,\&\, 0xFF0000FFu \\
point_{expanded_i} &= (point_{expanded_i} * 0x00000101u) \,\&\, 0x0F00F00Fu \\
point_{expanded_i} &= (point_{expanded_i} * 0x00000011u) \,\&\, 0xC30C30C3u \\
point_{expanded_i} &= (point_{expanded_i} * 0x00000005u) \,\&\, 0x49249249u
\end{aligned}
\tag{2.50}
$$

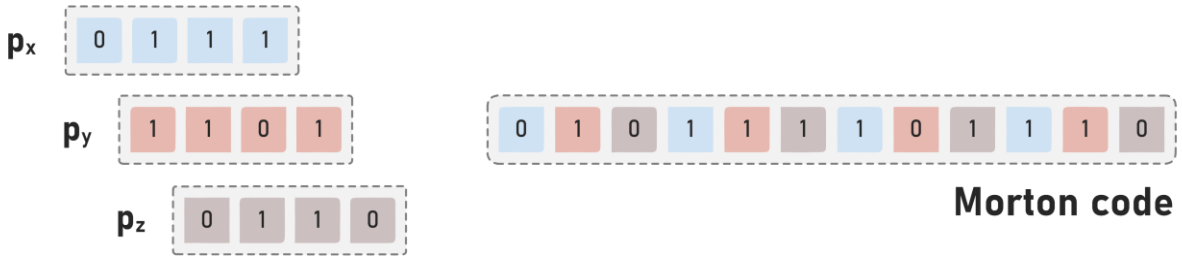


Ilustración 92. Esquema de intercalado de valores binarios de una posición cualquiera, con el fin de generar un Morton code.

El tiempo empleado en hallar esta codificación para todos los triángulos de la escena se muestra en la Tabla 35, tanto para la implementación en CPU como en GPU. Como bien sabemos, la versión GPU también incluye tiempo de CPU de algunas

operaciones básicas, y la implementación en CPU debe igualmente definir el resultado como un *buffer* en GPU. De esta forma, es posible comparar de manera justa los tiempos de ejecución de ambas versiones.

Tiempo medio de respuesta de cálculo de Morton *codes*			
Ejecución	**Tiempo de respuesta en GPU (ms)**	**Tiempo de respuesta en CPU (ms)**	
1	30,254	117,681	
2	29,063	125,304	
3	28,232	114,843	
4	27,759	118,223	
5	29,881	118,335	
	Tiempo medio (ms) 29,03	**Tiempo medio (ms)**	118,87

Tabla 35. Tiempo medio de respuesta para hallar el código de Morton para cada triángulo de la escena (6.683.902 triángulos).

Una vez definida la codificación de los triángulos de la escena, es posible proceder a su ordenación. Al fin y al cabo, el objetivo no es otro que obtener un *buffer* donde triángulos en posiciones cercanas también se encuentran cerca en el espacio 3D. Disponemos de un código representativo de la posición de cada triángulo, sólo nos falta la generación de la *Morton curve*.

Un algoritmo de ordenación que encaja muy bien con la estructura de un *Morton code* es *Radix sort*, el cual se aplica sobre valores enteros, e itera dígito por dígito. En nuestra aplicación, estos dígitos serán bits, pero también se puede comprender fácilmente el algoritmo mediante cadenas que expresen números de cierta longitud. Por ejemplo, dado el conjunto $\{19, 43, 12\}$, es posible comenzar la ordenación considerando el dígito menos significativo, $\{1\mathbf{9}, 4\mathbf{3}, 1\mathbf{2}\}$, lo que produciría que se obtuviera el conjunto ordenado $\{12, 43, 19\}$. Mediante la última iteración se obtendría la ordenación completa, $\{\{12, 19\}, 43\}$. Por tanto, es fácil observar que la complejidad algorítmica es $O(kn)$, sea k el número de dígitos (en el ejemplo propuesto, dos). Como en nuestra aplicación disponemos de 30 bits, la complejidad se reduce a $O(30n)$, sea n el número de triángulos de la escena.

El algoritmo de ordenación *Radix sort* requiere a su vez la implementación de uno de los problemas más comunes de la informática, conocido como *prefix sum*, o *prefix scan*. Dada una posición i del vector, ¿cuál es la sumatoria de todos los

elementos que preceden dicha posición? En función del problema, se podrá añadir a la sumatoria el valor almacenado en i o no (sólo sumatoria de valores anteriores). Este ultimo problema ha sido tratado de manera bastante extensa en la literatura de la computación paralela, por lo que no es difícil encontrar una descripción teórica, e incluso práctica (CUDA) del mismo (Harris, Sengupta, and Owens 2007; Homepage et al. 2009; Hwu 2012).

Por tanto, procedemos a describir cada uno de los cinco pasos que se distinguen en todo este proceso, teniendo en cuenta que, de manera similar al cálculo del AABB, existen ciertas operaciones que requieren de $\log_2 N$ ejecuciones, y que el número inicial de hilos vendrá determinado por $\frac{N}{2}$.

1. En primer lugar, es necesario aplicar una máscara a cada uno de los *Morton codes* obtenidos, lo que nos permitirá recuperar únicamente un bit determinado por la iteración. Por tanto, $mask_{bit} = 1 \ll currentBit$, teniendo en cuenta que el valor inicial de $currentBit$ es 0 (se incrementa su valor en cada iteración de k). Por tanto, necesitamos hasta cuatro *buffers* en el *compute shader* de este paso:

   - *Buffer* de *Morton codes*. Sólo lectura.
   - Un *buffer* de índices que muestra la ordenación de los *Morton codes* originales. Es decir, a partir de este buffer ($indices_1$), y el *buffer* de *Morton codes*, se puede obtener el código que ocupa la posición i, $i \in \mathbb{N} \mid 0 \leq i < N$, a partir de la sintaxis $code_{morton}[indices[i]]$.
   - Un *buffer* que indica si el resultado de la operación $mask_{bit}$ & $code_{morton}$ contiene un bit con valor uno (para cada uno de los códigos disponibles). Es decir, si el índice es mayor o igual que uno, guardaremos uno; de no ser así, es necesario almacenar un cero. Una función de GLSL que permite implementar esto mismo sin una estructura condicional explícita es *step*, donde se indica el umbral (1), y el valor con el que se comparará dicho umbral, devolviendo cero o uno en función de si se encuentra por debajo o por encima (o igual). De ahora en adelante hablaremos de $pBits$.
   - Un *buffer* que almacena la negación de los valores previamente almacenados ($pBits_i \oplus 1$ (XOR)). De ahora en adelante, $nBits$.

2. A partir de este punto, comienza el proceso que nos permite calcular la nueva posición de un *Morton code*, de acuerdo con la ordenación de la iteración actual, que afecta al bit $currentBit$. Las siguientes tres etapas (comenzando por la actual) pertenecen al algoritmo de *prefix scan*. En la descripción anterior se visualizaba como una sumatoria, algo que se mantiene intacto en esta aplicación, pero con un objetivo subyacente distinto. Podemos imaginarnos una ordenación en función del bit menos significativo, de tal manera que los primeros códigos son aquellos que poseen un cero en el último bit, mientras que los últimos códigos tendrán el último bit activo. En definitiva, se ordena de menor a mayor. Por tanto, es necesario contar cuántos códigos existen con el bit actual ($currentBit$) a cero antes de una posición i, con el fin de llevar a cabo un cambio de posición de los códigos.

   El flujo seguido es exactamente igual que en el cálculo del AABB. Son necesarias $ceil(\log_2 N)$ ejecuciones, hasta conseguir ascender a la raíz del árbol que se esboza en todo este proceso. En la raíz hallaremos la sumatoria de todo el *buffer* (Ilustración 93), que en nuestro caso indica cuántos códigos existen, en total, con el bit actual a cero, lo cual nos será igualmente útil. También es fácil observar que, en este paso, la sumatoria calculada en muchos de los nodos se encuentra incompleta. Sólo se encuentran completos aquellos nodos cuya sumatoria ha sido calculada por el primer hilo de cada ejecución.

   Todos los cálculos aquí realizados se almacenan sobre $nBits$ por una mera cuestión de ahorro de memoria.

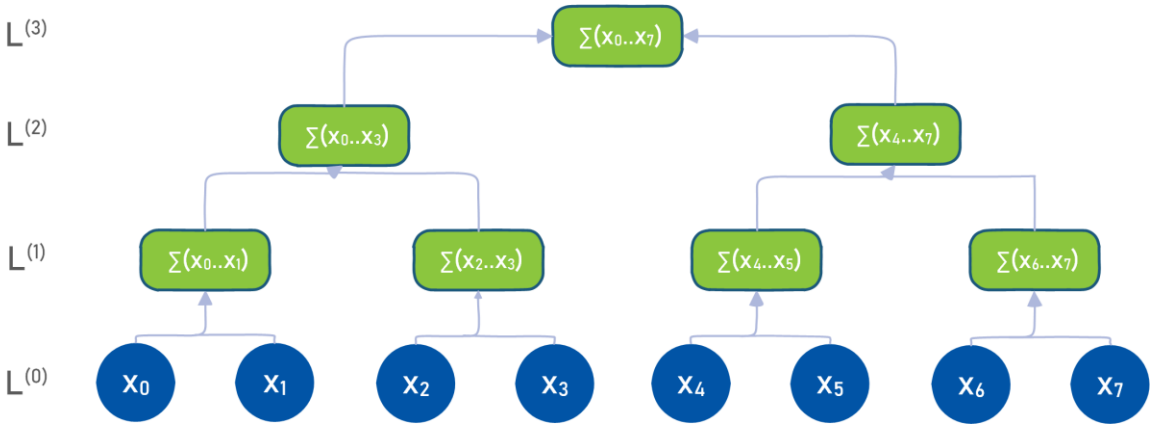


Ilustración 93. Representación del primer paso del algoritmo prefix scan. Etapa reduce.

3. Si bien es cierto que la sumatoria hallada en el nodo raíz es relevante, se debe suprimir su valor para dar paso a la siguiente etapa, donde comenzaremos a intercambiar y sumar de nuevo, pero en la dirección inversa. Por tanto, el valor de $nBits[N-1]$ pasaría a ser 0. En cualquier caso, la sumatoria de un nodo cualquiera se calcula como la sumatoria hasta el índice anterior, por lo que la suma hallada en la raíz no se ajusta a este principio.

   Aunque se trata de una operación muy sencilla, se debe llevar a cabo en GPU, dado que en cualquier otro caso habría que recuperar el *buffer*, modificarlo y trasladarlo de nuevo a GPU. Por tanto, se prefiere lanzar un *compute shader* muy simple antes que llevar a cabo tal flujo de operaciones.

4. Al contrario que la segunda etapa, conocida como *reduce*, esta cuarta etapa se denomina *down-sweep*, dado que se realiza el camino en la dirección inversa: de la raíz a los nodos hoja (Ilustración 94). A diferencia de antes, comenzamos con un hilo y finalizamos con $ceil\left(\frac{N}{2}\right)$. Cada hilo se encarga de un intercambio, y una sumatoria. Partiendo de dos índices, $index_1$ e $index_2$, podemos desplazar la sumatoria localizada en $index_2$ a la posición $index_1$, y almacenar en $index_2$ la sumatoria de los valores almacenados en $index_1$ e $index_2$.

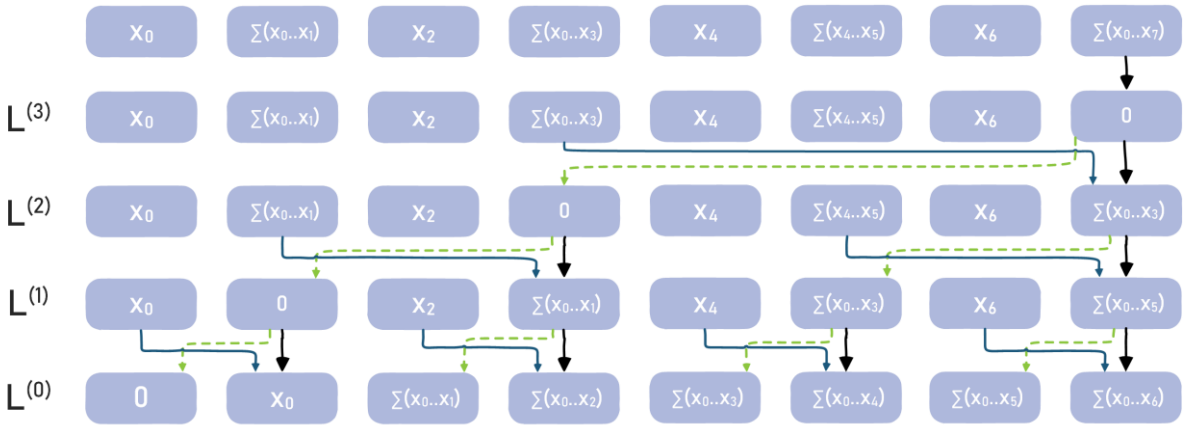


Ilustración 94. Representación de las etapas de reset y down-sweep de prefix scan.

5. La última etapa de nuestro algoritmo de ordenación es la reasignación de los *Morton codes* a sus nuevas posiciones. En esta situación emplearemos todos aquellos *buffers* previamente calculados que no han sido de utilidad

hasta ahora, como $pBits$. Se distinguen principalmente dos situaciones en función del valor de $pBits_i$:

- $pBits_i$ es 0, y por tanto, el bit es 0. Esta es la situación más fácil, dado que la nueva posición vendrá dada por el resultado del proceso de *prefix scan*, $nBits_i$.
- $pBits_i$ es 1, lo que complica en cierto modo el cálculo de la nueva posición, dado que esta no se ha calculado de manera explícita previamente. La nueva posición se define como sigue, sea $tIndex$ el identificador global del hilo:

$$position = tIndex - nBits[tIndex] + nBits[N-1] + (pBits[N-1] \oplus 1) \tag{2.51}$$

La formulación $nBits[N-1] + (pBits[N-1] \oplus 1)$ permite obtener el número total de códigos con el bit $currentBit$ a cero. Nótese como la sumatoria hallada en $i = 7$ se definía como $\sum_{j=0}^{i-1} nBits[j]$, por lo que nos faltaría incrementar la sumatoria si el último código contiene un valor 0 en el bit actual. Mientras tanto, $tIndex - nBits[tIndex]$ nos devuelve la posición que ocuparía un código de bit k activo dentro del conjunto de códigos que se encuentran en esta situación. Por tanto, se conserva siempre el orden previo, aunque este no sea relevante para el algoritmo (Ilustración 95).

Nótese como la asignación de la nueva posición de un código i se debe almacenar en un *buffer* como $indices_1$. Sin embargo, si se hiciera aquí mismo, podríamos encontrar hilos que no pudieran leer el índice adecuado porque este ya se hubiera sobreescrito, generando así una reordenación incorrecta. Por tanto, disponemos de dos *buffers* de índices, $indices_1$ e $indices_2$, que deberán intercambiarse en cada nueva iteración (nuevo bit). Para el *shader* será transparente; recibe dos vectores de índices y no se encarga en absoluto de este intercambio. Será en CPU donde se intercambie el contenido de dos variables que almacenan el identificador de ambos *buffers*.

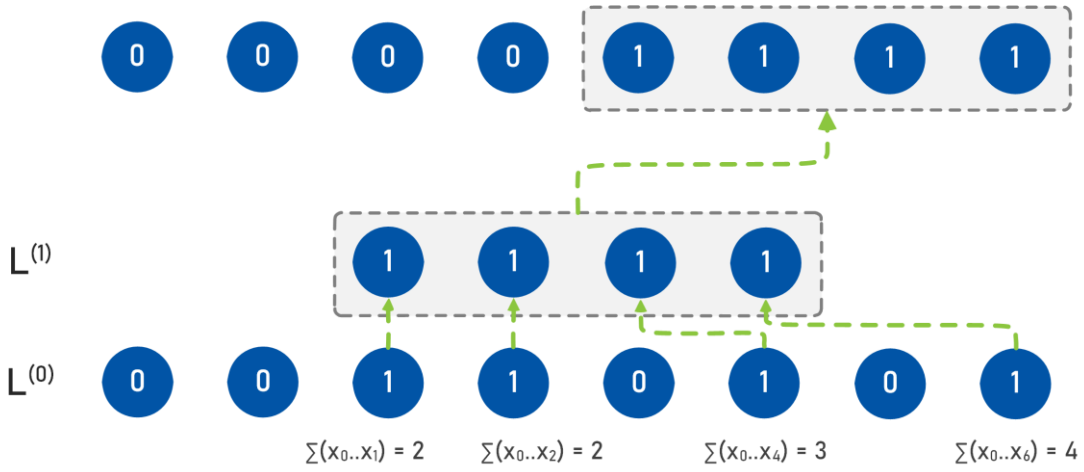


Ilustración 95. Representación de la selección de la nueva posición de aquellos códigos cuyo bit k es 1.

Una vez ordenado el vector de *Morton codes*, es necesario liberar los *buffers* empleados y comenzar con el proceso de construcción del BVH. Igualmente, en el grupo es posible vaciar los vectores que almacenaban la geometría, topología o la información de todas las mallas de la escena, debido a que ya se encuentran en GPU, y esto nos permitirá reducir de nuevo la memoria asociada al proceso.

El tiempo medio empleado en la ordenación se muestra en la Tabla 36 y la Tabla 37 para dos escenarios de diferente tamaño (2.288.011 y 6.683.902 triángulos). En este caso, se compara el tiempo obtenido de la implementación aquí propuesta y uno de los métodos de ordenación que provee C++, *std::sort*, el cual implementa un algoritmo híbrido y adaptativo de ordenación, *IntroSort*, que a su vez se compone de otros tres algoritmos: *Quicksort*, *Heapsort* e *Insertion Sort*. Por tanto, se considera este como uno de los mejores algoritmos de ordenación.

Tiempo medio de ordenación de *Morton codes* (2,2 millones de triángulos)			
Ejecución	**Tiempo de respuesta (ms)**	**Tiempo de respuesta de *std::sort* (ms)**	
1	23,814	85,639	
2	24,052	89,992	
3	26,326	85,456	
4	24,726	88,861	
5	23,857	89,214	
	Tiempo medio (ms) 24,55	**Tiempo medio (ms)**	87,83

Tabla 36. Tiempo medio de ordenación de códigos Morton para más de dos millones de triángulos.

Tiempo medio de ordenación de *Morton codes (6,6 millones de triángulos)*			
Ejecución	**Tiempo de respuesta (ms)**	**Tiempo de respuesta de *std::sort* (ms)**	
1	51,775	269,734	
2	49,921	263,038	
3	47,187	275,940	
4	52,327	254,573	
5	49,196	253,965	
	Tiempo medio (ms) 50,08	**Tiempo medio (ms)**	256,45

Tabla 37. Tiempo medio de ordenación de códigos Morton para más de seis millones de triángulos.

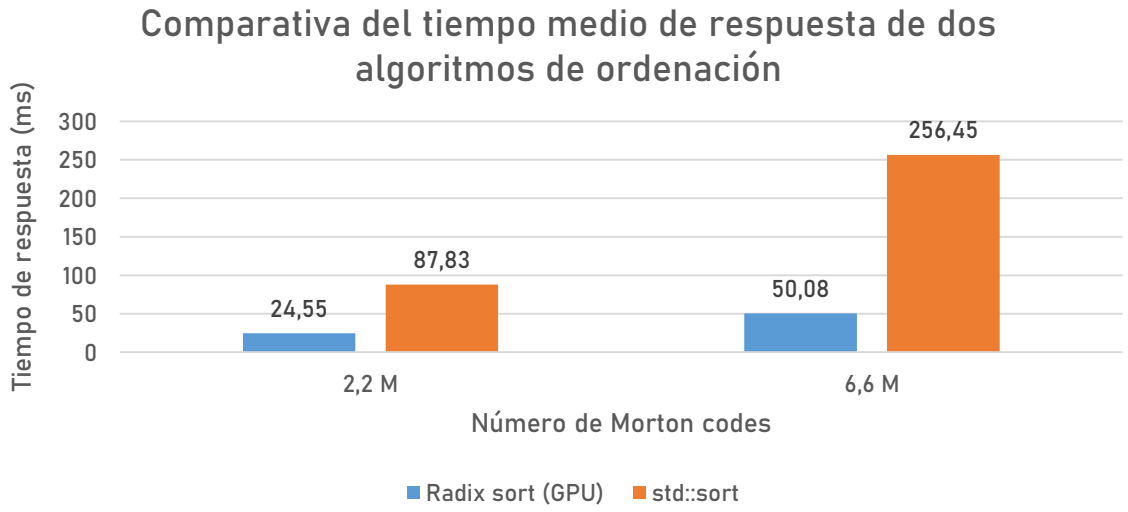


Ilustración 96. Representación mediante un diagrama de barras del tiempo medio obtenido en la ordenación de los códigos Morton, tanto en la versión GPU descrita como en la implementación dada por la librería de algoritmos de C++.

2.4.6.3.2 Núcleo de construcción del BVH

Para comenzar la construcción del BVH es necesario inicializar un *buffer* que integre los nodos hoja del árbol. Para ello sólo necesitamos el *buffer* de índices de los *Morton codes* ordenados, construido en el último algoritmo descrito, así como el *buffer* con la topología de la escena. De esta manera, un clúster queda definido por:

- El índice (posición) de un triángulo en el *buffer* de la topología. Dicho índice es accedido como $indices[tIndex]$, sea $indices$ un *buffer* obtenido con anterioridad en la ordenación, y $tIndex$ el identificador global del hilo.

- Una caja envolvente, formada por el punto mínimo y máximo. Dicha caja se obtiene a su vez de la caja envolvente del triángulo que debería situarse en este clúster, $indices[tIndex]$. Por tanto, se obtiene como $face\big[indices[tIndex]\big].\{point_{min}, point_{max}\}$.

- Dos índices de nodos precedentes. Al tratarse de nodos hoja, no existe ningún nodo anterior, y como tal, se establece que ambos índices almacenen un valor que indica la ausencia de un nodo anterior (`0xFFFFFFF`).

La composición de este *buffer* inicial también se produce mediante un *compute shader*, aprovechando así su definición en GPU, y más tarde, la paralelización de los cálculos sobre este mismo *buffer*. En este último caso, obtendríamos un escenario compuesto de 1) preparación de nodos hoja de manera secuencial, y 2) definición de vector resultante en GPU, mientras que la alternativa implementada considera 1) definición de *buffer*, 2) preparación de nodos hoja en paralelo. Ambos escenarios son factibles, y muy probablemente el cálculo secuencial en CPU apenas alcanza unos microsegundos, pero la definición del *buffer* en GPU es igualmente costosa.

A la hora de inicializar el SSBO con los nodos del árbol es necesario tener en cuenta que no se deben definir únicamente N nodos, sea N el número de triángulos en la escena, sino $2N-1$, que será el número de nodos que componen el árbol completo. Con N nodos sólo podríamos representar los nodos hoja, aunque la inicialización descrita sólo se emplea los N primeros nodos. Es fácil comprobar que $2N-1$ es el número apropiado de nodos a partir de imágenes como la Ilustración 93.

En todo este proceso de inicialización, no sólo se define un *buffer* con todos los nodos que tendrá nuestro BVH, sino también un *buffer* auxiliar de tamaño N que utilizaremos en la construcción, el cual también se inicializa con los nodos hoja (N es el tamaño máximo de un nivel, y por tanto, es suficiente).

Por tanto, podemos iniciar la construcción de nuestra nueva estructura de datos. Se diferencian tres claras etapas:

- **Búsqueda del mejor vecino**. Dado un nodo i y un radio r, es necesario hallar aquel nodo en $\langle i-r, i+r\rangle$ que minimiza una función objetivo dada por la heurística SAH.

- **Comprobación del mejor vecino**. Se debe verificar que $vecino\big[vecino[i]\big] = i$ para poder combinar dos *clusters*, aunque sólo uno de

ellos se actualiza (más concretamente, aquel que verifica que $i < N[i]$, es decir, el nodo de menor identificador (pero igualmente, podría ser el segundo)). El nodo que no se actualiza se invalida.

Podríamos pensar que es posible alcanzar una situación de bucle infinito si ningún par de vecinos verifica la condición de unión. Sin embargo, esta selección de vecinos puede definirse como un grafo dirigido, a partir del cual se demuestra que siempre habrá al menos una unión de nodos. Evidentemente, la situación ideal es que muchos más nodos puedan combinarse, dado que sólo así es posible reducir el tiempo de ejecución.

- **Etapa de *prefix scan* para compactar el *buffer* de nodos**. Tras la segunda etapa se obtiene un conjunto heterogéneo de nodos, algunos de ellos actualizados (compactados), otros invalidados (exactamente el mismo número que de compactados), y otros clústers que no se han modificado al no verificarse las condiciones de unión. A efectos prácticos, el primer y último estado de un nodo son equivalentes, dado que son nodos válidos. El objetivo de esta etapa es diferenciar claramente nodos válidos de los nulos, y el algoritmo de *prefix scan* nos permite hallar la nueva posición de aquellos nodos válidos.

Este mismo comportamiento aquí descrito se representa en el pseudocódigo de la Ilustración 97.

Algorithm 12 $BuildBVH$: Construcción iterativa del BVH

1: $C_{in}, C_{out} \leftarrow [C_0, C_1, ..., C_{n-1}]$
2: $N \leftarrow scan \leftarrow [0, 1, ..., n-1]$
3: $clusters_{current} \leftarrow n$
4:
5: **while** $clusters_{current} > 1$ **do**
6: Calcular vecino más cercano a cada nodo $C_{in}[i]$
7: Barrier()
8:
9: Combinar clusters que verifican $N[N[i]] = i$, sólo en $i < N[i]$
10: Invalidar el nodo que no verifica $i < N[i]$ cuando $N[N[i]] = i$
11: Barrier()
12:
13: Compactar clusters para evitar nodos nulos intermedios (Prefix scan)
14: Barrier()
15:
16: $clusters_{current} \leftarrow scan[clusters_{current} - 1]$
17: **if** $C_{in}[clusters_{current} - 1]$ is null **then**
18: $clusters_{current}$ += 1
19: **end if**
20:
21: $swap(C_{in}, C_{out})$
22: **end while**

Ilustración 97. Pseudocódigo de la construcción de un BVH.

El algoritmo necesita de un gran número de *buffers* y parámetros, por lo que comenzaremos explicando cada uno de ellos con el fin de dar paso a las etapas posteriores.

- $buffer_{cout}$ y $buffer_{cin}$, dos vectores temporales de tamaño N, sea este el número de triángulos de la escena y también la máxima longitud de un nivel de la estructura. Ambos se irán intercambiando, de tal manera que en una iteración sólo uno de ellos contiene información válida.
- $position_{in}$ y $position_{out}$, dos vectores temporales que almacenan posiciones de un *buffer* global del BVH (antes generado) que nos permitirá referenciar un nodo.
- $index_{neighbor}$, el vector que se empleará para seleccionar el mejor vecino.
- $prefixScan$, el *buffer* donde se almacenará el resultado del algoritmo *prefix scan*, con el fin de compactar un conjunto de nodos.
- $cluster_{valid}$, un *buffer* que indica para cada posición si este es válido o nulo.

- $cluster_{merged}$, un vector que indica si se ha producido una unión de clústeres en la posición i.
- $num_{nodes}\ y\ size_{array}$, dos *buffers* de tamaño uno que indican, respectivamente, el número de nodos por combinar, y el número de nodos hasta ahora generados.

Además, en cada iteración debemos calcular num_{groups}, el número de grupos necesarios para un número N de nodos aún por combinar; $num_{threads}$, el número inicial de nodos en ciertas etapas de la construcción, como en la reducción de *prefix scan*; num_{exec}, el número de ejecuciones necesarias en aquellas etapas iterativas (como la antes citada), o $num_{groupsLog}$, el número de grupos necesarios para $num_{threads}$.

A partir de este punto a enumeraremos las diferentes etapas de las que se compone el algoritmo de construcción del BVH. De nuevo, la división en etapas se debe a la necesidad de esperar a que todos los hilos finalicen, algo que se representa en la Ilustración 97 mediante *Barrier()*.

1. Búsqueda del vecino que minimiza una función objetivo dada por la heurística SAH, la cual se define como sigue:

$$
\begin{aligned}
&point_{min} = \min(cluster_1.point_{min}, cluster_2.point_{min})\\
&point_{max} = \min(cluster_1.point_{max}, cluster_2.point_{max})\\
&length = point_{max} - point_{min}\\
&SAH = 2 * length_x * length_y + 2 * length_z * length_y + 2 * length_x * length_z
\end{aligned}
\tag{2.52}
$$

Dicha heurística se puede incorporar a un proceso de búsqueda de un valor mínimo, como el que se muestra en la Ilustración 98. Este mismo comportamiento podría simplificarse aún más, a cambio de un menor rendimiento, dado que es posible comprobar si $index_{current} = tIndex$.

Algorithm 13 *FindBestNeighbor*: Búsqueda del mejor vecino de cada nodo

```
1: index_lower ← index_current ← min(0, tIndex − radius)
2: index_upper ← max(tIndex + radius, N − 1)
3: distance_min ← ∞
4:
5: while index_current < tIndex do
6:     distance ← SAH(tIndex, index_current)
7:     if distance < distance_min then
8:         neighbor[tIndex] ← index_current
9:         distance_min ← distance
10:    end if
11:
12:    index_current += 1
13: end while
14:
15: index_current += 1
16:
17: while index_current < index_upper do
18:    distance ← SAH(tIndex, index_current)
19:    if distance < distance_min then
20:        neighbor[tIndex] ← index_current
21:        distance_min ← distance
22:    end if
23:
24:    index_current += 1
25: end while
```

Ilustración 98. Pseudocódigo de la búsqueda del mejor vecino para un nodo cualquiera del BVH.

2. Una vez hallado el mejor vecino de cada nodo i es posible comenzar a combinar clústeres, para lo cual simplemente habrá que tener en cuenta varias casuísticas: 1) el mejor vecino de i también ha determinado que i es su mejor vecino, y 2) el identificador i es menor que $neighbor[i]$. Se abren así tres escenarios posibles (es indiferente el segundo cuando no se cumple el primero). $cluster_{merged}$ sólo será 1 cuando $i < neighbor[i]$, y $cluster_{valid}$ será 1 en esa misma situación o cuando se verifique $neighbor\big[neighbor[i]\big] \neq i$. Nótese como $cluster_{valid} = prefixScan$, con la diferencia de que este segundo *buffer* sufrirá cambios en la siguiente etapa ($cluster_{valid}$ se empleará más tarde). El pseudocódigo de esta distinción de estados se muestra en la Ilustración 99, donde se representa mediante un comentario la combinación de dos clústeres. A la hora de combinar dos nodos habrá que tener en cuenta ciertos aspectos:

- Es necesario calcular de nuevo el AABB, a partir del mínimo y máximo punto.
- El índice del triángulo contenido es nulo; es un nodo intermedio, y no un nodo hoja.
- Es necesario referenciar las posiciones de los clústeres originales, con el fin de establecer una jerarquía. Dichas posiciones vendrán dadas por $position_{in}[index_1]$ y $position_{in}[index_2]$.
- La posición del nuevo nodo vendrá dada por el resultado de una suma atómica llevada a cabo sobre un contador global. Por tanto, $position_{in}[index_1]$ pasa a almacenar dicho resultado.

Algorithm 14 *ClusterMerging*: Combinación de aquellos clústeres que minimizan mutuamente una función objetivo

```
1:  if neighbor[neighbor[tIndex]] == tIndex then
2:      if tIndex < neighbor[tIndex] then
3:          Construye nuevo clúster con tIndex y neighbor[tIndex]
4:          cluster_valid[tIndex] ← prefixScan[tIndex] ← 1
5:          cluster_valid[neighbor[tIndex]] ← 0
6:          prefixScan[neighbor[tIndex]] ← 0
7:          cluster_merged[tIndex] ← 1
8:      else
9:          cluster_merged[tIndex] ← 0
10:         Return
11:     end if
12: else
13:     cluster_valid[tIndex] ← prefixScan[tIndex] ← 1
14:     cluster_merged[tIndex] ← 0
15: end if
```

Ilustración 99. Pseudocódigo de combinación de clústeres en la segunda etapa de la construcción de un BVH.

3. Etapa de *prefix scan*. Su implementación se ha desarrollado en un apartado previo, por lo que sólo nos interesa conocer la información de entrada y salida del algoritmo para este escenario. La entrada vendrá dada por un *buffer* de ceros y unos, sea uno un indicador de validez del nodo, ya sea porque no ha encontrado un vecino, o porque ha podido combinarse con otro nodo. Por tanto, lo que hará *prefix scan* para una posición i es llevar a cabo un recuento de nodos válidos anteriores a esa posición.

4. **Actualización de nodos válidos**. Se parte de $buffer_{in}$, donde se almacenan los clústeres compactados en la iteración actual, $prefixScan$, que indica la nueva posición de un nodo i válido, y $position_{in}$, donde se almacena la posición del nodo i en el buffer global del BVH. Por tanto, para cada nodo válido es necesario llevar a cabo las siguientes actualizaciones:

$$buffer_{out}\left[prefixScan[tIndex]\right] = buffer_{in}[tIndex]$$
$$position_{out}\left[prefixScan[tIndex]\right] = position_{in}[tIndex] \quad (2.53)$$

5. **Cálculo del número de clústeres válidos aún por combinar**. Será un único hilo el encargado de hallar tal valor, pero es recomendable hacerlo en un *compute shader* para evitar leer desde GPU hasta dos *buffers* de millones de elementos. Después de este cálculo, sólo es necesario leer un *buffer* de tamaño uno, reduciendo así la carga de lectura. El número de nodos válidos se calcule como sigue:

$$num_{valid} = prefixScan[tIndex - 1] + validCluster[tIndex - 1] \quad (2.54)$$

Nótese como se ha simplificado la descripción del algoritmo ignorando el intercambio de *buffers* que se produce tras cada iteración, de tal manera que $buffer_{in}$, $buffer_{out}$, $position_{in}$ y $position_{out}$ se representan mediante identificadores diferentes en iteraciones consecutivas (aunque recuperan su identificador origina cada dos ciclos).

Para finalizar este apartado de construcción de un BVH se muestra el tiempo de respuesta observado (Tabla 38), considerando para ello tres radios diferentes, 10, 25 y 100, como se propone en (Meister and Bittner 2018). Se documenta el tiempo de respuesta obtenido en el núcleo del algoritmo (no necesariamente incluye únicamente operaciones GPU) y el tiempo derivado de la construcción de aquellos *buffers* y parámetros que son necesarios para ejecutar el algoritmo. Por tanto, se debe considerar que el tiempo total de este nuevo algoritmo es la suma de ambos. En cualquier caso, el tiempo medio obtenido es mucho menor que el que se obtenía en la construcción de un *octree* para una escena con muchos menos triángulos. La misma información documentada mediante tablas se representa de manera gráfica en la Ilustración 100.

En cualquier caso, el tiempo aquí obtenido es mayor que el documentado en el artículo donde se describe este proceso de construcción, si bien es cierto que las pruebas se ejecutan sobre una tarjeta gráfica más avanzada, y el *framework* de paralelismo es CUDA, y no un *compute shader*.

Tiempo medio de construcción de un BVH **Escena: 2,2 millones de triángulos**			
Radio: 10			
Ejecución	**Tiempo de respuesta (ms)**		**Tiempo de construcción de buffers (ms)**
1	70,771		148,513
2	64,667		156,977
3	67,861		155,854
4	68,635		144,421
5	68,344		144,422
	Tiempo medio (ms)	68,05	**Tiempo medio (ms)** 150,03
Radio: 25			
Ejecución	**Tiempo de respuesta (ms)**		**Tiempo de construcción de buffers (ms)**
1	103,144		143,496
2	96,711		148,986
3	97,049		146,370
4	96,759		148,837
5	104,117		155,017
	Tiempo medio (ms)	99,55	**Tiempo medio (ms)** 148,54
Radio: 100			
Ejecución	**Tiempo de respuesta (ms)**		**Tiempo de construcción de buffers (ms)**
1	239,492		148,649
2	255,312		151,560

3	280,357		149,545		
4	244,057		150,663		
5	246,076		146,597		
	Tiempo medio (ms)	256,05	**Tiempo medio (ms)**	149,4	

Escena: 6,6 millones de triángulos					
Radio: 10					
Ejecución	**Tiempo de respuesta (ms)**		**Tiempo de construcción de buffers (ms)**		
1	115,066		450,916		
2	115,092		483,856		
3	118,503		465,772		
4	115,826		481,769		
5	113,735		480,973		
	Tiempo medio (ms)	115,64	**Tiempo medio (ms)**	472,65	
Radio: 25					
Ejecución	**Tiempo de respuesta (ms)**		**Tiempo de construcción de buffers (ms)**		
1	192,743		462,385		
2	176,454		437,154		
3	177,392		519,417		
4	186,276		508,072		
5	183,730		532,680		
	Tiempo medio (ms)	183,31	**Tiempo medio (ms)**	491,94	
Radio: 100					
Ejecución	**Tiempo de respuesta (ms)**		**Tiempo de construcción de buffers (ms)**		
1	568,589		494,069		
2	569,517		505,207		

3	566,03	451,214	
4	569,428	482,023	
5	555,029	496,116	
	Tiempo medio (ms) 565,71	**Tiempo medio (ms)**	485,72

Tabla 38. Tiempo medio de respuesta del algoritmo completo de construcción de un BVH para diferentes tamaños de escena y radio.

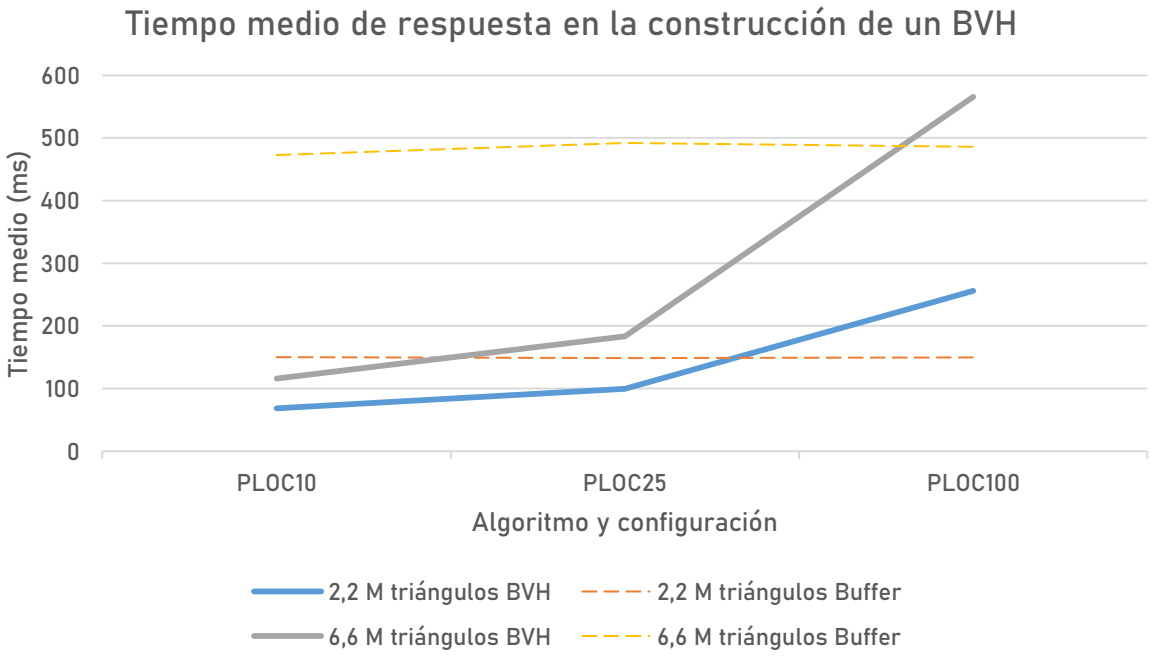


Ilustración 100. Diagrama de líneas para mostrar el tiempo medio de respuesta de la inicialización de recursos y la construcción de un BVH.

Para finalizar este apartado se muestran algunas imágenes de este proceso de construcción. En la Ilustración 101 se muestran los nodos hoja del BVH, de tal manera que su representación parcial nos permite visualizar cómo se completa la *Morton curve* (en qué dirección avanza, ejes predominantes, etc). Nótese como esta representación nos permite indicar el número de nodos que deben representarse; no existe una diferenciación de niveles por sí misma. Igualmente, se muestra el BVH construido sobre la escena de 6,6 millones de triángulos en la Ilustración 102. Finalmente, se muestra la construcción de un BVH sobre una única figura en la Ilustración 103.

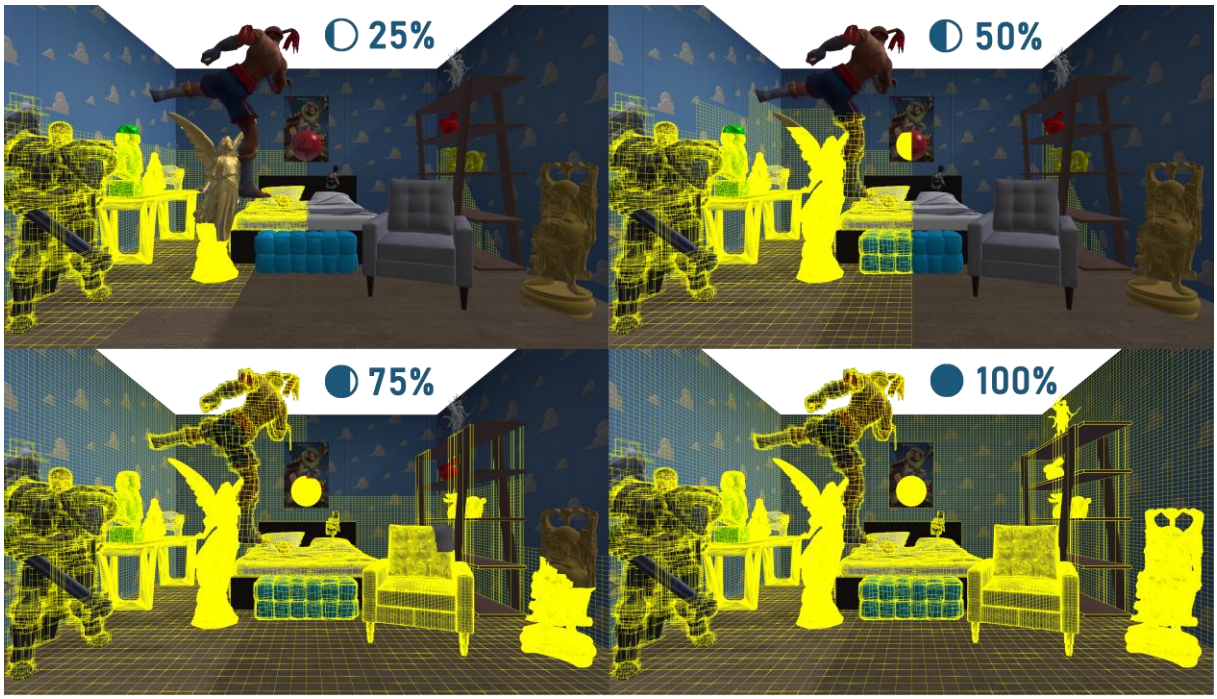


Ilustración 101. Representación de nodos hoja de BVH para mostrar el avance de una Morton Curve.

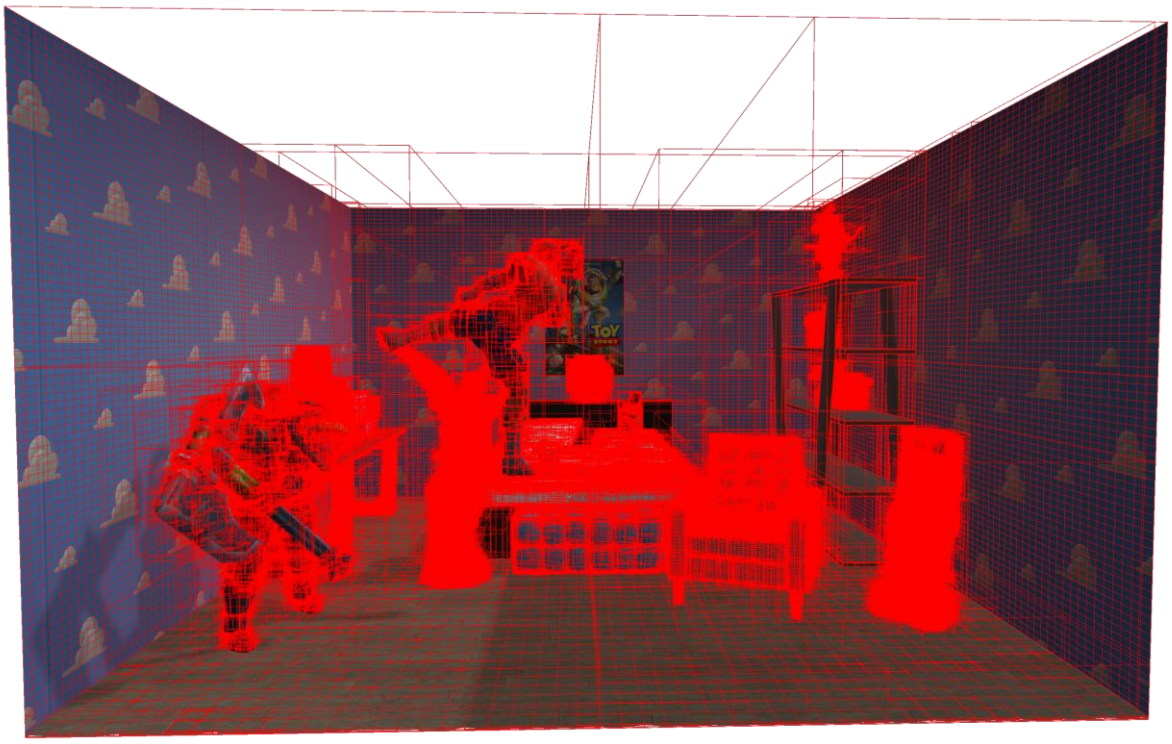


Ilustración 102. Representación de la estructura de un BVH completo.

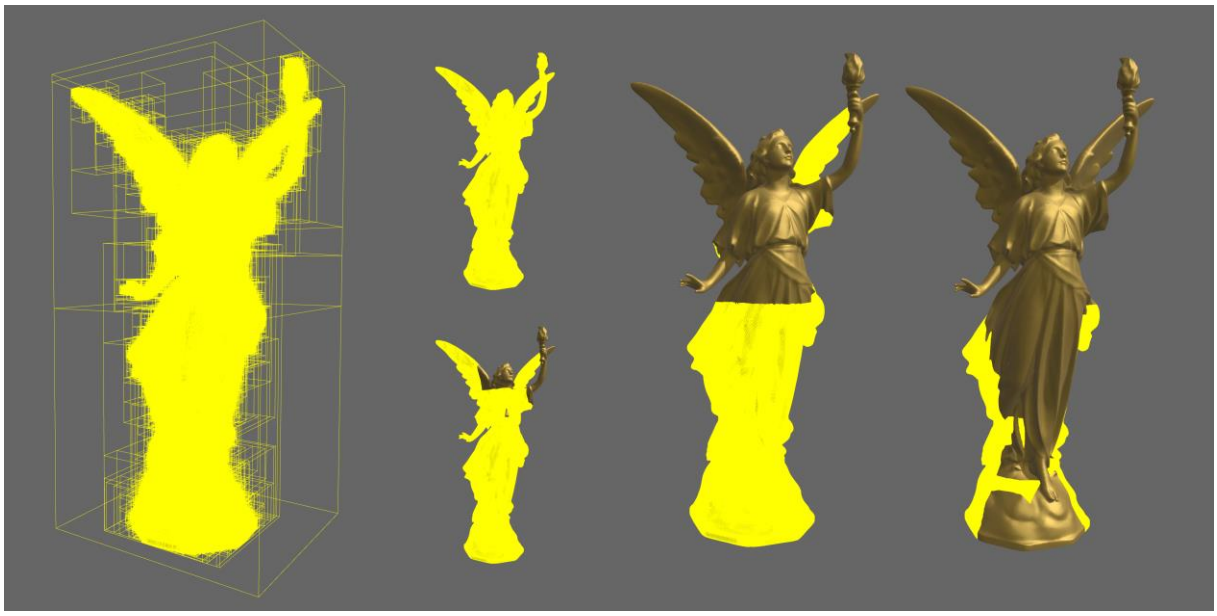


Ilustración 103. Representación de la construcción de un BVH para una única figura.

2.4.7 Diagrama de clases

Como en iteraciones anteriores, se muestra un diagrama de clases actualizado con aquellas entidades que se han visto modificadas durante esta iteración (Ilustración 104). Principalmente, se añaden nuevas estructuras que servirán de nexo con los procesos en GPU, todas ellas con atributos y métodos públicos. El grupo se modifica en gran medida para adaptarse a la construcción de un BVH. Cada componente se asocia con algunas de estas nuevas estructuras, y lo mismo sucede para el grupo, aunque su vinculación es temporal (mientras se construye la estructura de datos). Igualmente, se incluyen otras tantas instancias de un VAO, desde un grupo o la simulación, con el fin de renderizar nuevos elementos de la escena, como el BVH o los rayos emitidos.

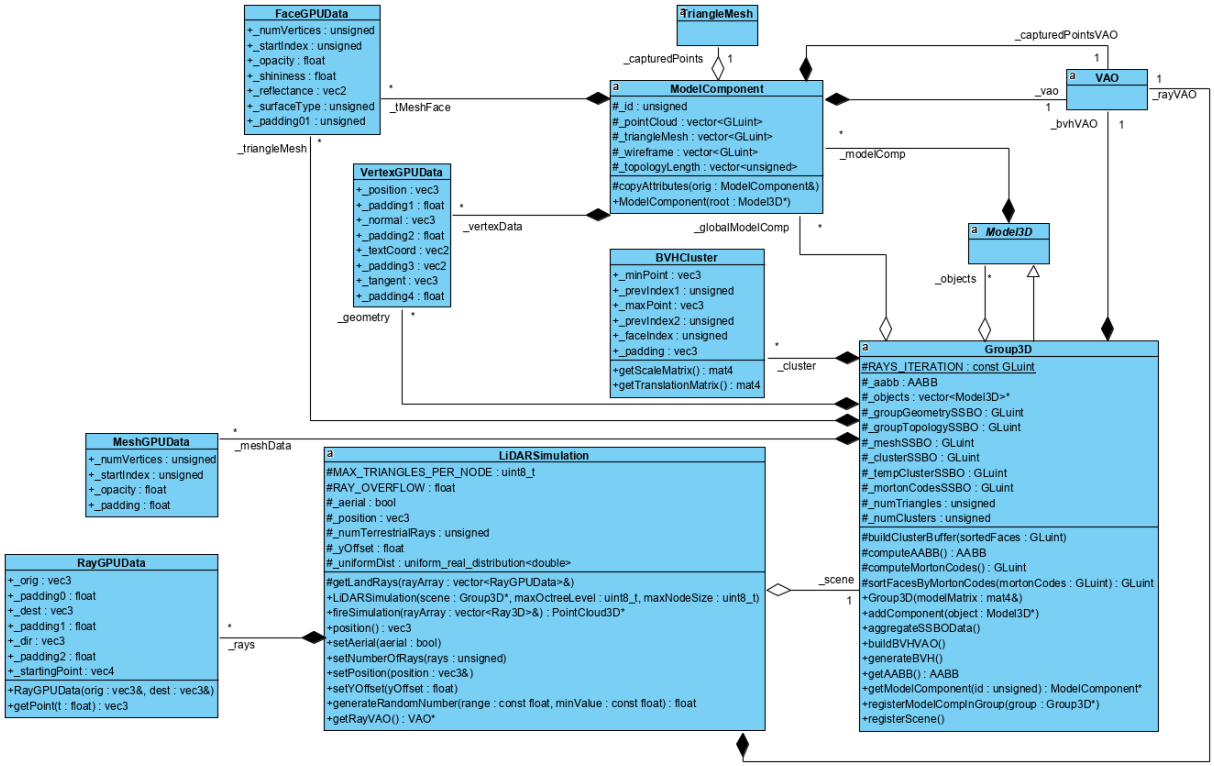


Ilustración 104. Diagrama de clases con aquellas entidades tratadas en la tercera iteración.

2.5 Cuarta iteración

La aplicación hasta ahora descrita únicamente es capaz de representar un escenario estático, a partir del cual construye una estructura de datos en GPU, conocida como BVH. Recordemos que en cierto punto de la documentación se describía un algoritmo de simulación básico capaz de generar una nube de puntos, aunque dicha versión realizaba los cálculos en CPU, mientras que la tendencia adoptada a partir de la tercera iteración es desplazar los cálculos a GPU. Esta decisión se justificaba a partir de los tiempos de respuesta obtenidos en la segunda y tercera iteración.

El objetivo de la iteración que nos ocupa es reproducir en GPU la simulación básica que antes se implementaba en CPU. Igualmente, se inicia la creación de un segundo escenario, que por ahora se describe únicamente como una superficie procedural. Comenzaremos describiendo el primero de los objetivos de acuerdo con el orden de implementación seguido.

2.5.1 Implementación de simulación

La entidad se encarga de controlar la simulación continúa siendo *LiDARSimulation*, donde ya se almacenaba el grupo raíz de la escena, se creaban los rayos o se intersectaban estos con la geometría de la misma. En esta nueva iteración, esta entidad sigue controlando el proceso, pero pierde cierto protagonismo en la etapa de intersección, dado que es el grupo que contiene la escena el que almacena todos aquellos *buffers* útiles en este proceso.

La simulación se define en el método **`fireSimulation`**, donde se indica desde el exterior si es necesario inicializar un VAO que represente todos los rayos. Seguidamente, se debe diferenciar si la simulación requerida es aérea o terrestre. De momento, sólo trataremos esta segunda. En esta diferenciación se introducen en un vector todos aquellos rayos que deberían originarse en un dispositivo LiDAR y colisionar con la escena. Momentáneamente, consideraremos también que la simulación no es incremental (animada), sino completa. Es decir, el vector de rayos cubrirá todo el espacio posible en una única iteración.

La generación de rayos constituye por sí misma un punto de interés en la aplicación. Se parte de un número de rayos, R, indicado por el usuario, y también de un *offset* de apertura, $offset_{terrestrial}$, que sabíamos que se aplicaba en los pies del dispositivo. A partir de estos parámetros es posible calcular los siguientes valores:

$$rays_x = rays_y = \sqrt{R}$$

$$x_{spacing} = \frac{2\pi}{rays_x}; \quad y_{range} = \pi - offset_{terrestrial} \tag{2.55}$$

$$offset_y = -\frac{\pi}{2} + offset_{terrestrial}$$

Nótese como se calcula $x_{spacing}$, pero no $y_{spacing}$. Uno de los problemas que hallábamos en los resultados de la simulación básica en CPU era la presencia de *artifacts* (patrón de Moiré) a consecuencia del patrón repetitivo que describían los puntos, así como la gran densidad de estos. Una solución hallada a este problema es la utilización de una distribución uniforme que nos permita obtener puntos aleatorios. Nótese como con un número suficientemente grandes de puntos, estos se distribuirán uniformemente a lo largo del espacio, y por tanto, se obtiene una sensación de completitud sin ningún tipo de *artifact*. De hecho, para una simulación terrestre es suficiente con emplear una distribución uniforme en $[-1, 1]$ y obtener tres valores

aleatorios para conformar un vector dirección. El principal problema de esta solución es que no existe control alguno para asegurar la aplicación de $offset_{terrestrial}$.

Por tanto, es posible iterar sobre $rays_x, rays_y$, de tal manera que volvemos al escenario inicial. Para cada ray_x se genera su correspondiente punto en una esfera de radio uno, $(\cos(angle_x), 0, -\sin(angle_x))$, y un eje de rotación, $(\sin(angle_x), 0, \cos(angle_x))$. Para dicho valor de $angle_x$ deben generarse $rays_y$ rayos. La aplicación de la distribución uniforme se encuentra precisamente en la selección de un valor dado por $random_{uniform} * y_{range} + offset_y$, definido en el método **`generateRandomNumber`**. A partir de aquí, es posible rotar el punto fijado en $angle_x$ mediante el eje de rotación definido, empleando el valor aleatorio recuperado. Nótese como, incluso en esta situación, existe cierto patrón de repetición en los ejes X y Z. Por esta misma razón, a la posición obtenida tras la rotación se le podría añadir otro valor de ruido, que se aplica no sólo a X y Z, sino también a Y, con el único objetivo de conservar la apertura indicada por $offset_{terrestrial}$. El factor *K* representado en la Ilustración 105 es únicamente un factor de reducción del efecto de *jittering*, de tal modo que queda acotada la alteración de la dirección de un rayo. La mejora obtenida mediante esta alteración se representa en la Ilustración 106, donde igualmente se muestra un fragmento de escena en el que se puede observar el patrón de Moiré.

Algorithm 15 *TerrestrialRays*: Generación de rayos de LiDAR terrestre

$y_{range} = \pi - offset$
$offset_y \leftarrow \pi/2 + offset_{terrestrial}$
$x_{angle} \leftarrow \pi$
$x_{spacing} \leftarrow 2\pi / rays_x$

for $x = 0, 1, \ldots, rays_x - 1$ **do**
    $pos \leftarrow (\cos(x_{angle}), 0, -\sin(x_angle))$
    $axis_{rot} \leftarrow (-pos_z, 0, pos_x)$
    for $y = 0, 1, \ldots, rays_y - 1$ **do**
        $rand_y \leftarrow generateRandomNumber(y_{range}, offset_y)$
        $pos_{sphere} \leftarrow rotate(angle_y, axis_{rot}) * pos$
        $ray \leftarrow Ray(pos_{LiDAR}, pos_{LiDAR} + pos_{sphere} * (uniformRand_{[0,1]} * 2 - 1)/K)$
        $rays.push(ray)$
    end for

    $x_{angle} \mathrel{+}= x_{spacing}$
end for

Ilustración 105. Generación de rayos de un LiDAR terrestre mediante una distribución uniforme.

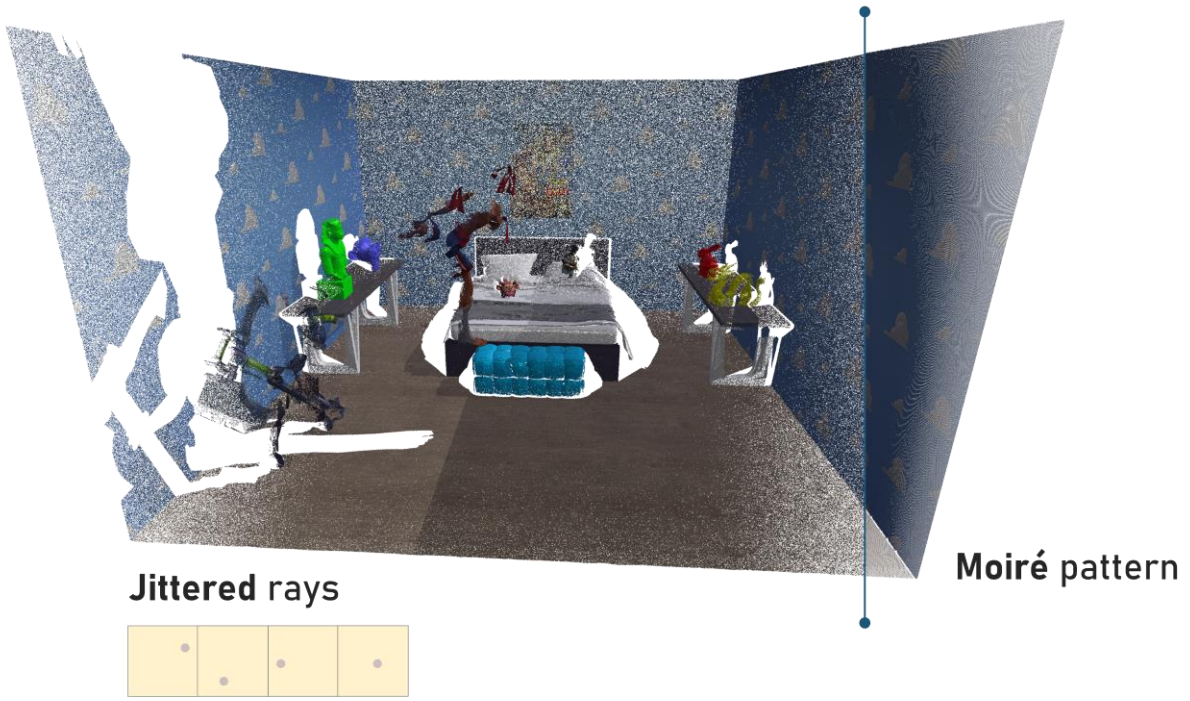


Ilustración 106. Comparativa de dos nubes de puntos: 1) obtenida con la alteración de los rayos, 2) obtenida con una distribución equitativa de rayos en el espacio.

Una vez definidos los rayos que se emiten, se debe indicar al grupo raíz de la escena que intersecte dichos rayos con la misma. La primera tarea que debe llevar a cabo el grupo es la definición de aquellos SSBOs que se introducen en la resolución de la intersección. El más evidente es un *buffer* que engloba los rayos, pero también es necesario definir un SSBO donde se almacenen los resultados de las intersecciones, a partir del cual podremos definir una nube de puntos.

Ambos SSBO contienen estructuras hasta ahora no citadas:

- ***RayGPUData***. Engloba las propiedades de un rayo, principalmente definido por una posición de origen y una dirección. Una de las características de una emisión láser es que pueden existir ciertos comportamientos, tales como refracciones, en los que la dirección se modifica, y en función de los tipos de operaciones que se realicen puede ser conveniente actualizar también el punto de origen (que no sería tal, sino el punto en el que se produce un cambio en la emisión láser). Sin embargo, hay cálculos que requieren conocer la posición desde donde comienza su recorrido el rayo. Por tanto, la alternativa más eficiente en estos casos sería indicar dicha posición mediante un *uniform* (menor gasto de memoria), pero también podría incluirse una propiedad en la entidad *RayGPUData*, a la que podríamos denominar *startingPoint*. Esta segunda opción implica mayor

gasto de memoria, pero también repercute en la flexibilidad del algoritmo, dado que nos permitiría llevar a cabo una simulación donde el dispositivo no permanece en una posición constante. Este es el caso, por ejemplo, de una simulación aérea.

- ***TriangleGPUCollisionData***. Una colisión tendrá un gran número de propiedades, dado que no es suficiente con indicar la posición resultante. Es necesario considerar que, a partir de toda la información aquí almacenada, se podrán obtener múltiples tipos de *renderings*; algunos de ellos emplean la normal, e incluso la coordenada de textura (RGB *rendering*), otros utilizan el retorno en que se halló el punto, etc. Por tanto, se listan a continuación todos estos valores:
    - Punto de intersección.
    - Identificador del triángulo colisionado.
    - Normal de la superficie intersectada.
    - Distancia de colisión. En función de la distancia, es posible que se reduzca la intensidad captada del punto.
    - Coordenadas de textura en el punto de colisión.
    - Identificador del componente al que pertenece. Puede ser un buen punto de partida para el *rendering* semántico.
    - Retorno en el que se captó la colisión.
    - Número total de retornos del rayo al que pertenece la colisión.
    - Ángulo de incidencia. De nuevo, podría afectar la intensidad captada.
    - Intensidad captada en el punto de colisión.
    - Colisión previa. Nos permite anidar intersecciones para rectificar el número total de colisiones de todos los puntos captados por un mismo rayo.

En la definición de los rayos emitidos en un SSBO es necesario considerar que existe cierta restricción dada por el sistema. Por ejemplo, en uno de los equipos de desarrollo se identifica como umbral de memoria la cantidad aproximada de diez

millones de rayos. Por tanto, lo que se propone, dada la limitación hallada, no es una resolución de intersecciones única, sino iterativa, aunque lo ideal es minimizar este número. De hecho, dos iteraciones suelen ser suficientes para hallar una nube de puntos relativamente densa.

Así, se define un umbral, $rays_{iteration}$, que influye en el tamaño de los SSBOs definidos, dado que este vendrá dado por $\min(num_{rays}, rays_{iteration})$. En cualquier caso, no es necesario dividir el vector de rayos tantas veces como indique $num_{rays}/rays_{iteration}$, dado que es posible introducir los datos en un SSBO mediante una dirección de inicio y un tamaño. Por tanto, si $num_{rays} > rays_{iteration}$, es suficiente con emplear $\&rays.at(iteration * rays_{iteration})$, o lo que es lo mismo, $\&rays.at(rays_{size} - rays_{left})$.

El comportamiento de este proceso se muestra en la Ilustración 107, donde se alude únicamente a la ejecución de un *compute shader* que trataremos a continuación.

Algorithm 16 *SolveRaysIntersection*: Resolución de lanzamiento de rayos e intersección con la escena

```
1: SSBO_size ← min(rays_size, rays_iteration)
2: Definir SSBO de colisiones con tamaño SSBO_size
3: Inicializar SSBOs de rayos y número de colisiones
4: rays_left ← rays_size
5:
6: while rays_left > 0 do
7:     Actualizar SSBO de rayos
8:     Reiniciar SSBO de número de colisiones
9:     Inicio: rays_size − rays_left
10:    Tamaño (rays_current): min(rays_left, rays_iteration)
11:
12:    Ejecución de compute shader
13:
14:    Obtención de número de colisiones y vector de colisiones
15:    rays_left −= rays_current
16: end while
17:
18: Liberar buffers
```

Ilustración 107. Pseudocódigo de resolución iterativa de intersecciones de rayos con la escena.

El comportamiento de colisión de un rayo con la escena únicamente requiere un impacto en esta iteración. Por tanto, comenzando en la posición de partida de un rayo, es posible descender en la jerarquía de un BVH, descartando un nodo (y todos sus descendientes) cuando no existe una intersección entre un rayo y un AABB, o almacenando los dos descendientes más inmediatos para ser procesados

posteriormente. Una vez alcanzado un nodo hoja, habrá que verificar si existe una intersección del rayo con el triángulo almacenado. Por tanto, se identifican al menos dos algoritmos de intersección que deben desarrollarse en un *compute shader*:

- Intersección entre un rayo y un AABB. El algoritmo más eficiente se describe en (Majercik et al. 2018) (*slab test*), el cual se reduce únicamente a cuatro líneas:

$$t_0 = \frac{aabb_{min} - ray_{orig}}{ray_{dir}};\ \ t_1 = \frac{aabb_{max} - ray_{orig}}{ray_{dir}}$$

$$t_{min} = \min(t_0, t_1);\ \ t_{max} = \max(t_0, t_1)$$

$$t_{near} = \max\left(t_{0_x}, t_{0_y}, t_{0_z}\right);\ \ t_{far} = \min\left(t_{1_x}, t_{1_y}, t_{1_z}\right) \tag{2.56}$$

$$collision = t_{far} \geq t_{near}$$

  En nuestra aplicación, es posible acotar aún más la comprobación de una colisión, dado que sólo es necesaria la intersección más cercana. Así, el hilo $tIndex$ podrá almacenar en su colisión la mínima distancia de colisión registrada, permitiéndonos utilizar la siguiente línea:

$$collision = t_{near} < collision_{minDistance} \ \wedge\ t_{far} \geq t_{near} \tag{2.57}$$

  Esta operación es únicamente posible cuando la dirección ray_{dir} se encuentra normalizada, en cuyo caso el valor parámetrico t es la distancia de la colisión al punto de origen del rayo. La verificación de esta operación lógica implica únicamente que no se descarta una posible colisión, dado que al menos el mínimo valor t de entrada se encuentra más cerca a ray_{orig} que $collison_{minDistance}$.

- Intersección entre un rayo y un triángulo. Un test clásico es (Moller and Trumbore 1998), el cual ya hemos descrito en una iteración anterior. Su implementación en GLSL no difiere en gran medida de la versión en CPU, en gran parte debido a la utilización de GLM como librería para resolver operaciones matemáticas (su propósito es obtener una sintaxis similar a la de GLSL).

  Se observan otras variantes, como la propuesta por (Quilez, s.f.), aplicada a *ray-marching*, pero una vez implementada se comprueban claros

errores en la resolución de intersecciones de la escena. De no ser así, este sería un algoritmo muy apropiado para nuestra aplicación, dado que se puede resolver igualmente mediante la regla de Cramer, lo que nos permitiría reducir el número de operaciones para hallar la coordenada de textura del punto de colisión (recordemos que necesitábamos resolver un sistema de ecuaciones mediante el mismo método).

Otra problemática que surge en el recorrido de un BVH es la manera en que este se ejecuta. Sabemos cómo intersectar un rayo con sus AABBs, lo que nos permitiría considerar sus dos descendientes como candidatos (o no). La manera de almacenar todos los nodos candidatos es un problema de gran interés en todo este proceso. Se considera que los nuevos nodos añadidos tienen una prioridad mayor en la exploración que aquellos volúmenes que se introdujeron previamente. Por tanto, la estructura de datos que mejor representa este comportamiento es una pila.

La justificación de este comportamiento radica principalmente en la incapacidad de obtener dinámicamente un *buffer* de un tamaño acorde a cierto valor calculado. Por tanto, la pila se debe implementar sobre un *buffer* de tamaño constante. Se podrán lanzar hasta diez millones de rayos en una misma ejecución (si el umbral se mantiene en este valor), por lo que dicho *buffer* no pueder disponer de un gran número de elementos. Por ejemplo, en nuestra aplicación se especifica un tamaño de cien. Una vez conocida la situación, es necesario comprender lo que sucede cuando se implementa una cola y no una pila. Con una cola se exploran nodos en el orden en que se introducen, y es necesario considerar que siempre se introducen los dos descendientes del nodo raíz cuando el rayo colisiona con la escena. Por tanto, una cola supone la exploración de muchas ramas a la vez, lo que implica que es muy posible que se sobrepase rápidamente el tamaño de buffer definido (disponemos de un BVH con millones de nodos hoja). En cualquier caso, una de las principales ventajas de esta estructura de datos es la capacidad de descartar buena parte de la escena en cada paso, por lo que la cola podría no suponer siempre grandes problemas, y aún así, se prefiere prevenir posibles situaciones de acceso a un valor fuera de rango.

Además de todos estos métodos de gran interés, parece evidente que los datos necesarios para llevar a cabo todas estas comprobaciones son 1) el conjunto de nodos del BVH, 2) la geometría completa de la escena, 3) todos los triángulos referenciados

desde los nodos hoja del BVH, 4) información de mallas de triángulos, para acceder a la geometría adecuada (recordemos que la geometría de cierta malla comienza en un determinado índice), 5) un vector de salida de colisiones, y 6) información de los rayos emitidos.

Una vez expuesto el comportamiento y los datos necesarios, se muestra el pseudocódigo que permite explorar la estructura y hallar los puntos de colisión (Ilustración 108). Al buscar únicamente una colisión, el comportamiento aquí descrito sólo se reproduce una vez. La línea $index_{current} += 1$ permite contrarrestar el efecto de $index_{current} -= 1$, lo cual también sería equivalente, de manera conjunta, a $index_{current} -= !faceNull$, si una variable booleana $faceNull$ únicamente fuera verdadera cuando el rayo intersecta un nodo no hoja ($cluster.face = null$).

Algorithm 17 *RayBVHCollision*: Resolución de la colisión rayo-BVH

```
1:  Sea tIndex el identificador del hilo
2:
3:  collision[tIndex].face ← null
4:  collision[tIndex].distance ← ∞
5:
6:  index_current ← 0
7:  heap ← unsigned[100]
8:  heap[index_current] = num_clusters − 1
9:
10: while index_current >= 0 do
11:     cluster ← bvh[heap[index_current]]
12:
13:     if intersectAABBRay(cluster, ray) then
14:         if cluster.face! = null then
15:             if intersectTriangleRay(tIndex, ray) then
16:                 collision[tIndex].face ← cluster.face
17:                 collision[tIndex].modelComp_id ← face[cluster.face].modelComp_id
18:             end if
19:         else
20:             heap[index_current] ← cluster.index_1
21:             heap[index_current + 1] ← cluster.index_2
22:             index_current += 1
23:         end if
24:     end if
25:
26:     index_current −= 1
27: end while
```

Ilustración 108. Pseudocódigo de BVH traversal y comprobación de colisiones con la escena.

Algunos ejemplos de nubes de puntos obtenidas mediante esta simulación se muestran en la Ilustración 106. En esta iteración se reconocen al menos dos modos de *rendering* de dichas nubes de puntos: color RGB de la escena y color uniforme. Mientras que el primer nodo emplea la nube de puntos asociada a cada modelo, el segundo emplea una única nube de puntos general.

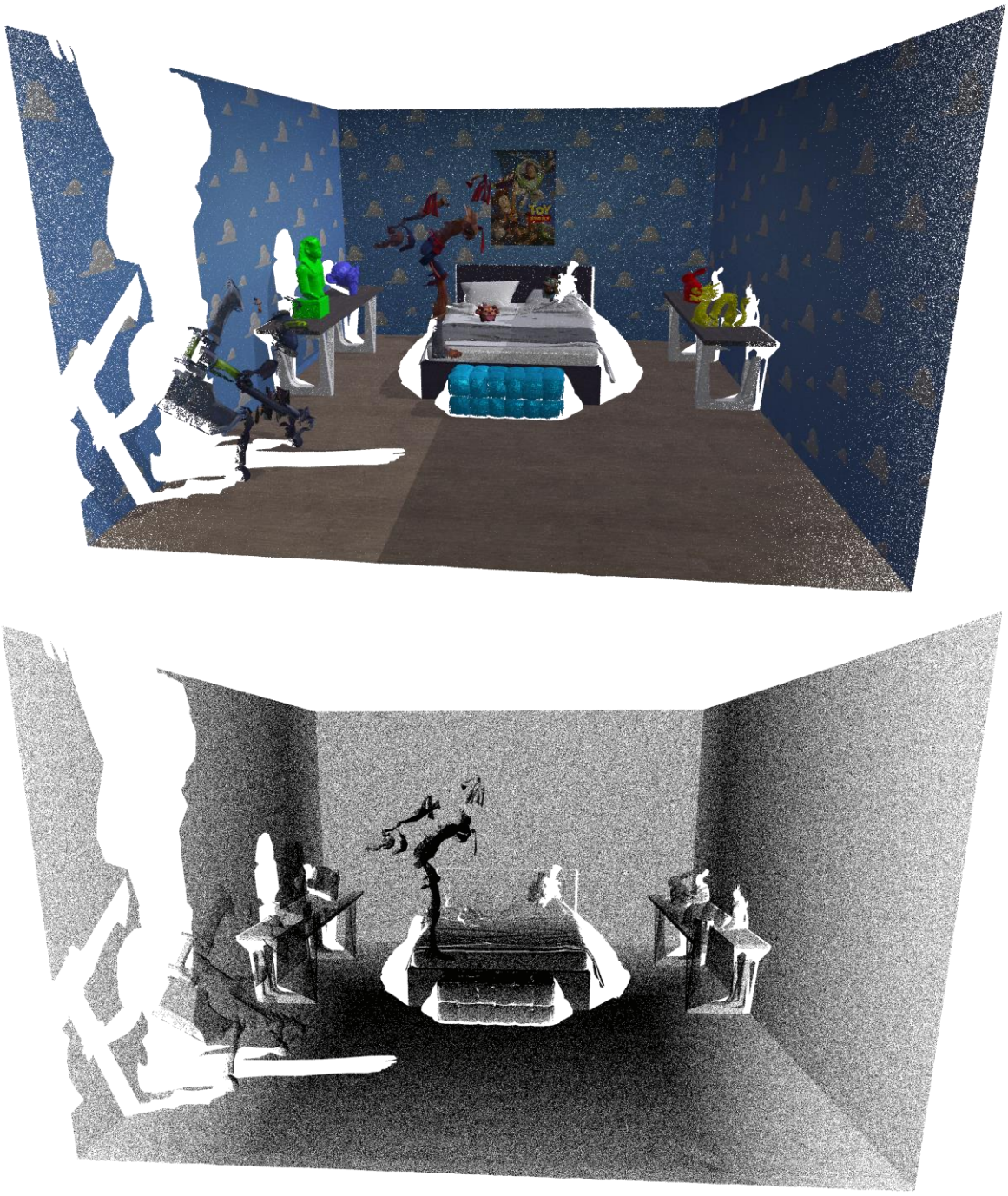


Ilustración 109. Nubes de puntos obtenidas a partir de la simulación de esta iteración. 1) Nube de puntos RGB, 2) nube de puntos de color uniforme.

Otro punto de gran interés en este apartado es la obtención del tiempo de respuesta necesario para completar una simulación (Tabla 39) (Ilustración 110). Recordemos que en iteraciones previas se documentaba el tiempo necesario para resolver este mismo problema con una escena de 600 mil triángulos, y emitiendo hasta 9 millones de rayos, si bien la estructura de datos era un *octree*, y no un BVH. Para el escenario más complejo que se planteaba, se obtenía un tiempo medio de respuesta cercano a los 50 segundos.

Para llevar a cabo las pruebas con esta nueva versión, se consideran dos escenas, las cuales parten de la primera definida. Se añade algún modelo más complejo para alcanzar fácilmente el millón de triángulos, y el resto de primitivas, que permiten alcanzar 2,1 y 6,5 millones, se obtienen a partir de la subdivisión (en mayor o menor medida) de los planos. Además, se consideran tres cantidades de rayos diferentes. Para cada prueba se considera el tiempo de respuesta general, que incluye la definición y lectura de *buffers*, y el tiempo de respuesta en el *kernel*, que obedece al tiempo de ejecución del *compute shader*. Uno de los aspectos más interesantes que se documenta en estas pruebas es la baja repercusión de la complejidad de la escena, lo que nos permite integrar una escena tan compleja como deseemos sin preocuparnos por un gran aumento del tiempo de respuesta. El número de rayos será el principal factor que influye en el tiempo de ejecución final.

Tiempo medio de resolución de intersecciones con la escena **Escena: 2,1 millones de triángulos**			
5 millones de rayos			
Ejecución	**Tiempo de respuesta (ms)**	**Tiempo de respuesta de kernel (ms)**	
1	3.828	2.598	
2	3.841	2.624	
3	3.806	2.560	
4	3.787	2.626	
5	3.821	2.563	
	Tiempo medio (ms) 3.816,6	**Tiempo medio (ms)**	2.594,2
10 millones de rayos			
Ejecución	**Tiempo de respuesta (ms)**	**Tiempo de respuesta de kernel (ms)**	
1	6.647	5.166	
2	6.570	5.214	
3	6.689	5.252	
4	6.541	5.195	

5	6.599		5.232	
	Tiempo medio (ms)	6.609,2	**Tiempo medio (ms)**	5.211,8

20 millones de rayos		
Ejecución	**Tiempo de respuesta (ms)**	**Tiempo de respuesta de kernel (ms)**
1	13.469	11.741
2	13.597	11.728
3	13.498	11.667
4	13.521	11.809
5	13.549	11.789
	Tiempo medio (ms) 13.526,8	**Tiempo medio (ms)** 11.746,8

Escena: 6,5 millones de triángulos		
5 millones de rayos		
Ejecución	**Tiempo de respuesta (ms)**	**Tiempo de respuesta de kernel (ms)**
1	3.822	2.633
2	3.804	2.597
3	3.851	2.650
4	3.902	2.657
5	3.883	2.635
	Tiempo medio (ms) 3.852,4	**Tiempo medio (ms)** 2.634,4

10 millones de rayos		
Ejecución	**Tiempo de respuesta (ms)**	**Tiempo de respuesta de kernel (ms)**
1	6.664	5.289
2	6.635	5.173
3	6.687	5.252
4	6.650	5.243

5	6.660		5.277	
	Tiempo medio (ms)	6.659,2	**Tiempo medio (ms)**	5.246,8

20 millones de rayos		
Ejecución	**Tiempo de respuesta (ms)**	**Tiempo de respuesta de kernel (ms)**
1	14.117	11.893
2	13.771	12.052
3	13.812	11.965
4	13.898	12.018
5	13.885	12.006
	Tiempo medio (ms) 13.896,6	**Tiempo medio (ms)** 11.986,8

Tabla 39. Tiempo medio de respuesta de la simulación LiDAR. Se comprueban escenarios de diversa complejidad y distintas cantidades de rayos. Igualmente, se mide el tiempo de respuesta del método completo (CPU) y el tiempo de resolución únicamente en GPU.

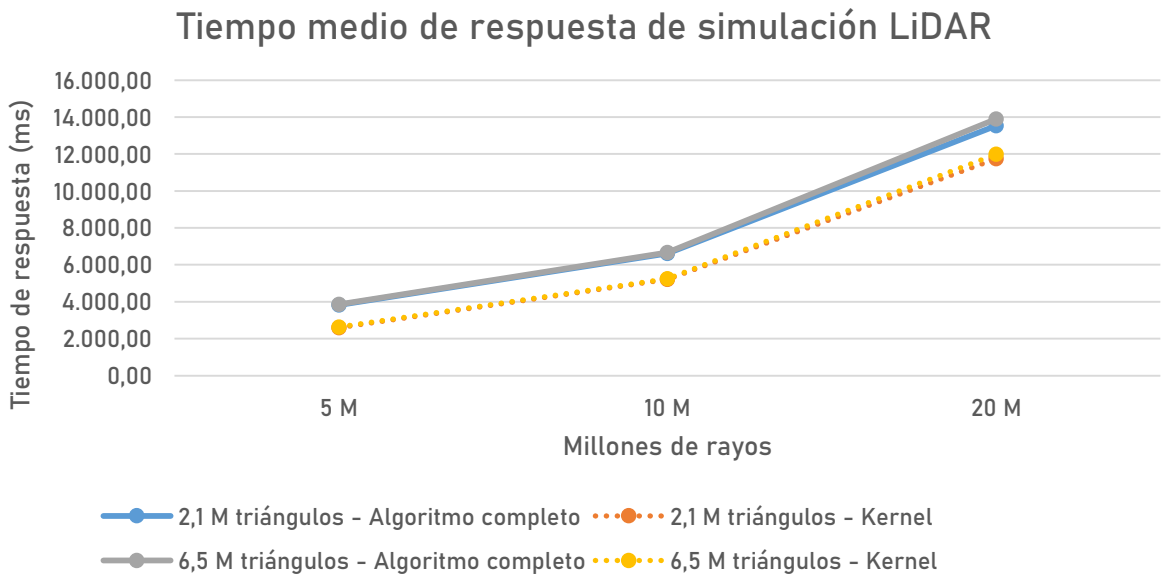


Ilustración 110. Tiempo de respuesta de la simulación LiDAR frente a diversos escenarios.

2.5.2 Generación de un terreno procedural

El segundo escenario contemplado en la aplicación es un terreno procedural que, en esta iteración, se define únicamente como un plano que puede verse alterado en ciertos puntos de la malla, con el único fin de producir las elevaciones de un terreno. Por tanto, parece evidente que disponemos de todas las herramientas para obtener

esto mismo: 1) una malla de nivel de detalle flexible que se implementa bajo la entidad *PlanarSurface*, y 2), implementación de la técnica de *displacement mapping*. Debido a la necesidad de generar una estructura de datos con la escena (y por tanto, con su posición final), la técnica de *displacement mapping* no se aplica en un *vertex shader*, sino en un *compute shader* que no sólo implementa dicha técnica, sino que también aplica una matriz de modelado que pudiera haber indicado el usuario (en cualquier otro caso, será la identidad). Parece evidente que, aunque disponemos de una parte de los mecanismos para desarrollar un terreno, existen otras cuestiones aún por explorar. Por esta razón, se utiliza una entidad que toma la base de *PlanarSurface*, y que engloba todo este nuevo comportamiento (***Terrain***).

La obtención de un terreno procedural implica su generación mediante algoritmos, lo que implica que este terreno puede variar a lo largo de las ejecuciones. Una manera sencilla de resolver este problema es la utilización de una función de ruido, que partiendo de una semilla, pueda generar un resultado dependiente de esta, lo que garantiza poder obtener un mismo escenario múltiples veces (conocida la semilla). Una función de ruido ampliamente utilizada es el ruido de Perlin (Perlin 1985), posteriormente corregida en (Perlin 2002). No se pretende desarrollar este algoritmo en este documento, aunque su lectura es interesante para conocer cómo funciona. Podemos definir este tipo de ruido como una función parámetrica que combina a su vez dos funciones, sea una pseudo-aleatoria y otra continua. No se debe interpretar únicamente como una función 2D, sino que se puede extender a n dimensiones, pudiendo ser una de ellas el tiempo, y por tanto, puede ser una función apropiada para generar animaciones (sea n_N la dimensión tiempo, nos interesa obtener un resultado de $N-1$ dimensiones, donde los incrementos de t determinan la continuidad de la animación). Por ejemplo, en una animación 2D, el eje X vendría determinado por el tiempo, mientras que el resultado de la función de ruido se representaría en Y. En una animación 3D, el problema se puede representar mediante una textura 2D diferente en cada instante de tiempo (una manera interesante de ilustrar esto mismo es mediante un océano en movimiento) (Ilustración 111).

Esta es quizás una de las funciones de ruido que producen resultados más realistas para nuestro campo de aplicación, pero aún así, no suele ser suficiente la utilización de una textura 2D de Perlin como mapa de altura, siempre que queramos un mayor grado de realismo. Por tanto, es sólo una función inicial, y como tal, cualquier otro ruido podría ser válido, aunque ciertamente la situación de partida influye en gran

medida en el resultado final. Por ejemplo, sería difícil alterar un *white noise* (completamente aleatorio) para generar un terreno realista (no hay ningún tipo de correlación entre los valores).

El resultado de la función de ruido de Perlin depende de un conjunto de parámetros que podemos modificar en función de una necesidad específica. En nuestra aplicación, haremos especial uso de la frecuencia, la lacunaridad o el número de octavas. Como bien se describe en el apartado Tecnologías utilizadas, este trabajo emplea la librería FastNoiseSIMD para generar texturas (2D, 3D, etc) de ruido. No sólo incluye un gran número de funciones, sino que además aprovecha la arquitectura del sistema para acelerar los algoritmos. También dispone de una aplicación donde es posible visualizar los resultados que se obtendrían al modelar un gran número de parámetros.

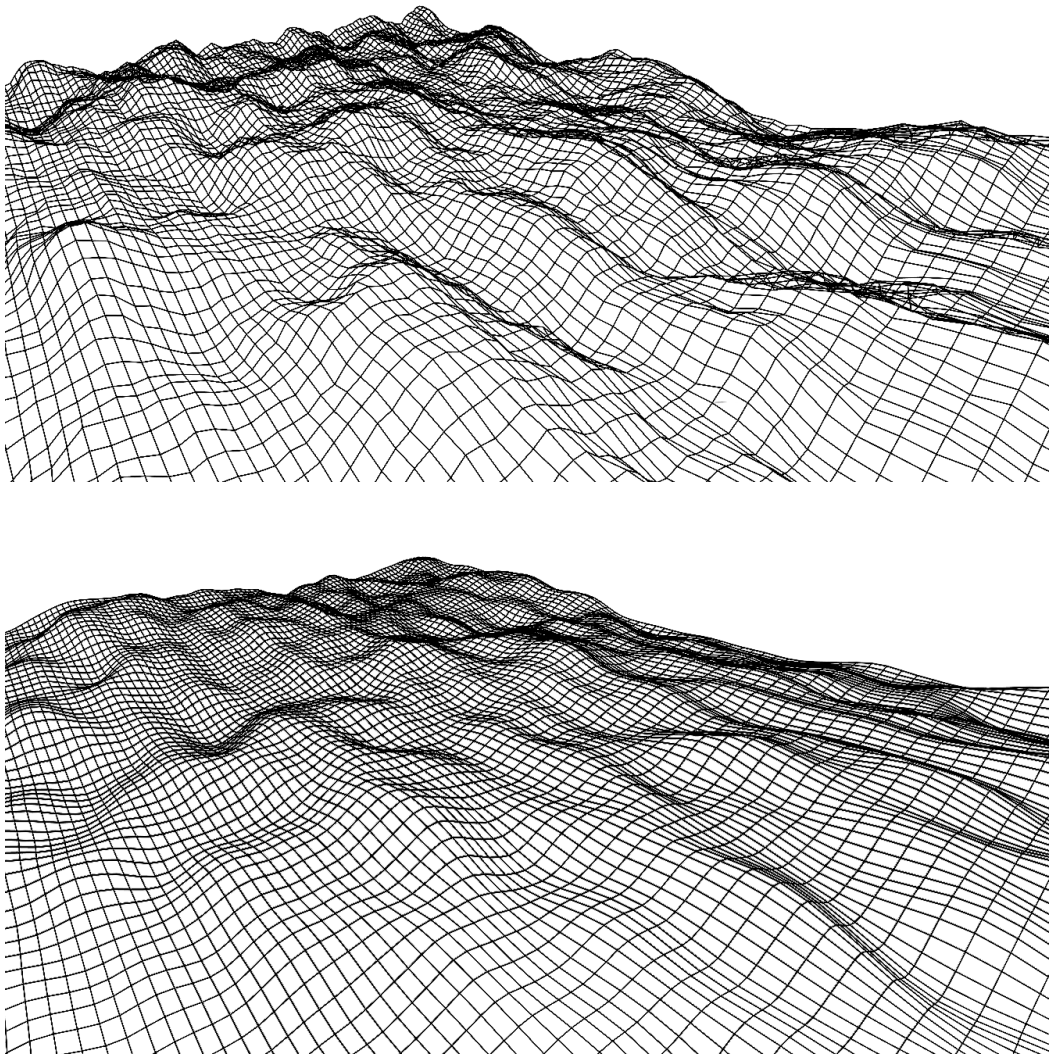


Ilustración 111. Ejemplos de planos modificados utilizando mapas de altura obtenidos mediante ruido de Perlin.

Los parámetros citados se pueden describir como siguen, aunque la mejor manera de comprender esto mismo es de manera visual (Ilustración 112):

- Frecuencia. Influye en la frecuencia con la que se producen cambios importantes de intensidad en una textura. Una mayor frecuencia se puede describir como la captura de un espacio de mayores dimensiones en una imagen de tamaño constante (mayor espacio, más detalles contenidos). Menor frecuencia supone la representación de un espacio más reducido, y por tanto, los cambios son menores.
- Octavas. La función de ruido de Perlin surge de la combinación de varias funciones, y esto mismo es lo que indica este parámetro. Nótese como no todas las octavas tienen un mismo peso en el resultado; lo más común es que se considere un peso decreciente para cada octava inferior. Permite introducir un poco más de ruido para evitar así fronteras bien definidas.
- Lacunaridad. No sólo disminuye el peso de cada octava inferior, sino que también es común aumentar sucesivamente la frecuencia de cada una. Al fin y al cabo, las octavas introducen algo de ruido sobre una función continua, por lo que la última octava será la que introduzca, a priori, más ruido en la función, pero también es cierto que su peso es muy reducido.

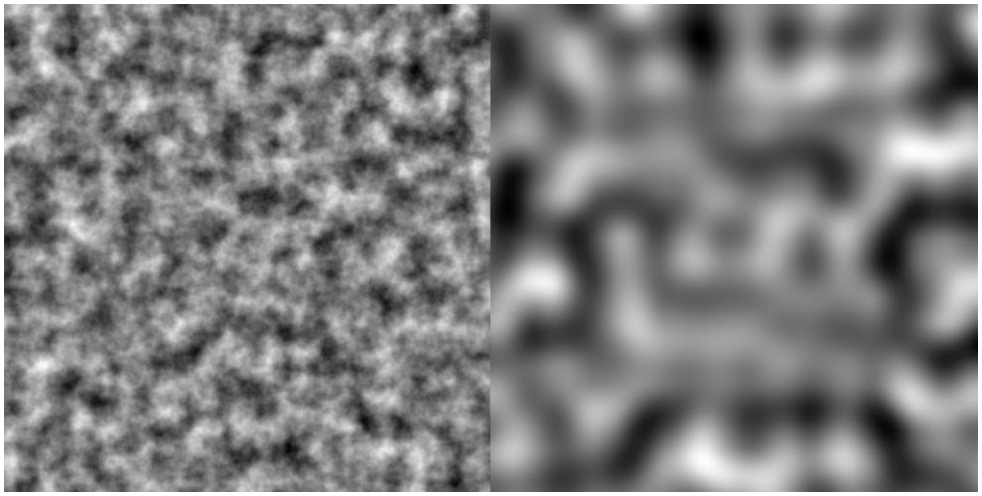


Ilustración 112. Texturas de ruido de Perlin obtenidas mediante diferentes configuraciones. 1) Frecuencia: 0.02, lacunaridad: 2, octavas: 5, 2) frecuencia: 0.01, lacunaridad: 1, octavas: 5.

La primera figura de la Ilustración 112 podría llegar a producir un terreno relativamente adecuado para nuestras exigencias, no así la segunda figura. La obtención de un terreno realista se justifica desde la aplicabilidad de este sistema, el cual pretende producir nubes de puntos que puedan ser empleadas en cualquier otra

solución o tarea de investigación. Trabajar sobre un terreno poco realista supone poca ventaja para un sistema que necesita aprender a partir resultados obtenidos en entornos reales (recordemos que el fin último de un sistema como este debe ser su aplicación en el mundo real).

Para comprender desarrollos posteriores, se muestra el resultado obtenido para un escenario que emplea una textura 2D obtenida mediante ruido de Perlin (frecuencia: 0.01, lacunaridad: 2, octavas: 6).

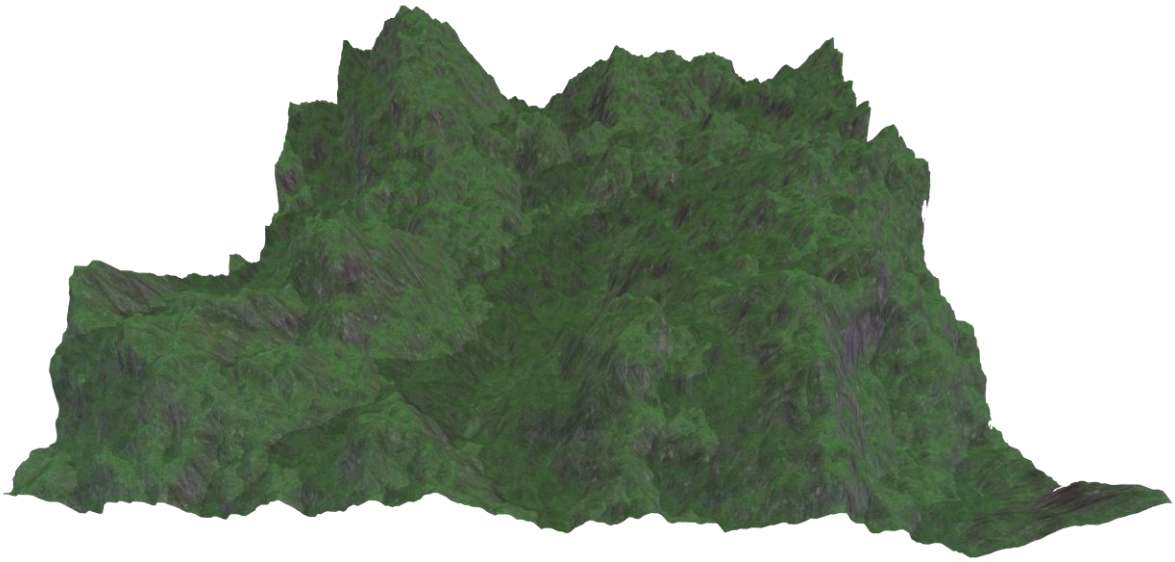


Ilustración 113. Rendering de un terreno cuyo desplazamiento vertical procede de una función de ruido de Perlin.

La generación de terrenos procedurales ha sido uno de los principales problemas de la informática gráfica, y como tal, existen algoritmos desde hace décadas que pretenden lograr esto mismo, sea uno de los primeros (Mandelbrot and Wheeler 1983). Una palabra clave en la generación de terrenos procedurales es **erosión**, dado que en las primeras soluciones (como la citada) se destacan ciertos puntos débiles que serían imposibles en la naturaleza, precisamente debido a la erosión. Indagar en todos los trabajos previos de esta temática supondría un apartado bastante extenso de estado del arte, por tanto, sólo mencionaremos qué métodos y posibilidades suelen emplearse.

Una erosión bastante fácil de intuir es la que produce el agua (erosión hidraúlica), pero también es frecuente encontrar métodos que simulan una erosión térmica (la erosión de *permafrost* con hielo por la acción térmica y mecánica del agua) o una erosión fluvial, como resultado del transporte de rocas a lo largo de una pendiente (Paris, 2019). No son métodos excluyentes, por lo que es común encontrar

algoritmos que integran varias de estas erosiones. De hecho, la erosión térmica no se considera realista por sí misma.

Como en la dinámica de fluidos, se distinguen al menos dos enfoques en la erosión hidraúlica, que será la erosión en la que nos centremos en este trabajo. Podríamos partir de N partículas que se desplazan a lo largo del terreno, causando la erosión que buscamos (método lagrangiano), pero también podríamos considerar una subdivisión del terreno en celdas, que a su vez interactúan con otras celdas vecinas (método euleriano).

El método que se implementa en este trabajo se describe en (Bayer, 2015), y se trata de un algoritmo basado únicamente en partículas que transportan y acumulan sedimento en un terreno en función de un conjunto de parámetros iniciales. La información de entrada del algoritmo, además de las variables que citaremos, es un mapa de altura, con valores en $[0,1]$, que será el resultado de una función de ruido de Perlin. Un sistema de partículas se suele prestar a la definición de un enfoque preciso desde el punto de vista físico, algo que no sucede con este algoritmo en algunos de sus puntos.

Comenzaremos describiendo el comportamiento de una sola partícula para describir el resto de detalles del algoritmo. Nótese como todas las ecuaciones que aquí se exponen se deben trasladar, sin grandes variaciones, al *compute shader* que erosiona el terreno inicial. Por tanto, se prefiere mostrar toda esta formulación frente a la representación de pseudocódigo.

Una partícula se define por su posición, p_{old}, que no es necesariamente un valor entero, situada en $[0, mapSize]$, una velocidad, inicialmente designada como p_{speed}, y una cantidad de agua, p_{water}. Tanto el sedimento como la dirección inicial se inicializan a cero. Para conocer hacia dónde se mueve una partícula es necesario conocer su gradiente, $g(x,y)$, sea (x,y) el píxel del mapa de altura donde se encuentra la partícula (a pesar de que $pos \in \mathbb{R}$). Por tanto, (x,y) se calcula como $uint(pos)$. Para calcular el gradiente se consideran cuatro vecinos, $(x,y),(x+1,y),(x,y+1)$ y $(x+1,y+1)$. Por tanto, este se puede calcular mediante un filtro bilineal que considera los factores $u,v \in [0,1)$, donde $(u,v) = pos - uint(pos)$.

$$g(pos) = \begin{pmatrix} (P_{x+1,y} - P_{x,y}) * (1-v) + (P_{x+1,y+1} - P_{x,y+1}) * v \\ (P_{x,y+1} - P_{x,y}) * (1-u) + (P_{x+1,y+1} - P_{x+1,y}) * u \end{pmatrix} \quad (2.58)$$

A partir del nuevo gradiente es posible calcular la nueva dirección, y a consecuencia de esto, la nueva posición:

$$dir_{new} = dir * p_{inertia} - g(pos) * (1 - p_{inertia})$$
$$pos_{new} = pos_{old} + dir_{new} \quad (2.59)$$

Parece evidente que $dir_{new} = 0$ conduce a una situación en la que la partícula no es capaz de seguir operando, lo cual sucede, principalmente, cuando esta comienza en un punto plano del terreno. Tras el comienzo, es más difícil alcanzar el valor cero absoluto en la dirección normalizada. Si se diera esta situación, es posible finalizar el flujo de la partícula afectada, o bien podría escogerse una dirección aleatoria. Debido a que la implementación se encuentra en un *compute shader*, se ha preferido la primera de las soluciones para evitar incluir una estructura condicional en el proceso, más aún cuando se trata de una situación difícil de encontrar.

Conocida pos_{new}y pos_{old} es posible calcular $\Delta_H = h_{new} - h_{old}$, la diferencia de altura entre ambas posiciones. No se puede descartar que $\Delta_H > 0$ (debido al filtro bilineal), a pesar de que es un comportamiento difícilmente asumible desde el punto de vista físico (sin considerar colisiones). A partir de este incremento, es posible calcular la nueva capacidad de sedimentación de la partícula:

$$sediment_{capacity} = \max(-\Delta_H * speed * water * p_{capacity}, p_{minSlope}) \quad (2.60)$$

Sea $p_{minSlope}$ un valor mínimo que evita que $sediment_{capacity}$ alcance cero. A partir de aquí se abren dos escenarios:

- El sedimento que transporta la partícula es mayor que su capacidad, o bien Δ_H es mayor que 0 (movimiento ascendente). En el primer caso, debe deshacerse de una cantidad de sedimento equivalente a $(sediment - sediment_{capacity}) * d_{deposition}$. Cuando la partícula asciende, debería depositar todo su sedimento, teóricamente, pero en la práctica se propone que deposite $\min(\Delta_H, sediment)$ para evitar que alcance 0. El sedimento no se acumula en pos_{new}, sino en pos_{old}, y más concretamente, se repartirá entre los cuatro vecinos considerados en el gradiente, también en función de (u, v).
- Cualquier otro escenario. No se acumula sedimento en una posición, sino que la partícula erosiona, cuya cantidad vendrá dada por:

$$erode_{amount} = \min(\left(sediment_{capacity} - sediment\right) * p_{erode}, -\Delta_H) \quad (2.61)$$

Mediante el mínimo se pretende evitar que una partícula pueda crear huecos en la superficie. En el caso de la erosión, esta no se aplica mediante un filtro bilineal, sino que se considera un radio en el que tendrá lugar este proceso. Para una posición dentro del radio definido, el peso de su erosión vendrá dado por:

$$w_i = \frac{\max(0, p_{radius} - (|P_i - pos|))}{\sum_{k=0}^{n} \max(0, p_{radius} - (|P_k - pos|))} \quad (2.62)$$

Este peso no es en absoluto dependiente de una posición específica, sino que más bien se debe interpretar como un *kernel* que pudiera aplicarse en un algoritmo de procesamiento de una imagen, que en este caso será el mapa de altura del terreno. Por tanto, es posible precalcular cualquier w_i y todos los desplazamientos necesarios para acceder a P_i. Si consideramos que pos es el centro de coordenadas, entonces es posible simplifcar la fórmula anterior:

$$w_i = \frac{\max(0, 1 - \frac{\sqrt{P_{i_x}^2 + P_{i_y}^2}}{p_{radius}})}{\sum_{k=0}^{n} \max(0, p_{radius} - \frac{\sqrt{P_{k_x}^2 + P_{k_y}^2}}{p_{radius}})} \quad (2.63)$$

A partir de w_i es posible calcular la erosión como $erode_{amount} * w_i$. Además de los pesos, es posible precalcular los índices de las celdas a las que debemos acceder. Dado $x \geq -p_{radius}, y \leq p_{radius}$ y $x^2 + y^2 < p_{radius}^2$, entonces el índice resultante se define como $y * heightMap_{size} + x$. Dicho valor se podrá añadir a $uint\left(pos_y\right) * heightMap_{size} + uint(pos_x)$, el centro del sistema. Nótese como $heightMap$ se debe definir como un *buffer* 1D (en cualquier caso, $y = \frac{index}{heightMap_{size}}, x = mod(index, heightMap_{size})$).

Una vez erosionado o sedimentado el terreno, es posible actualizar la partícula y continuar con la siguiente iteración, así hasta alcanzar el máximo tiempo vida indicado.

$$vel = \sqrt{vel^2 + \Delta_H * p_{gravity}}$$
$$water = water * (1 - p_{evaporation}) \quad (2.64)$$

Además del comportamiento aquí descrito, existen otros tantos puntos de interés que deben tenerse en cuenta, más allá del *compute shader*:

- Sea N el tamaño deseado para el mapa de altura, es necesario considerar $N + p_{radius} * 2$ para las operaciones que precedan e incluyan la erosión (Ilustración 114). De esta manera, es posible evitar el acceso a posiciones no definidas (fuera de mapa) comprobando en nuestro *compute shader* si la posición de la partícula es menor que p_{radius} o mayor que $heightMap_{size} - p_{radius}$ (tanto en X como en Y). En este caso, así como en la situación de $dir = 0$, la partícula finalizará su proceso de erosión.

  Además, el algoritmo presenta algunos problemas con ciertos valores de $p_{inertia}$, lo que puede ocasionar valles muy profundos en los extremos. Por tanto, lo que se propone es recuperar únicamente el centro del mapa tras el proceso de erosión, descartando así bordes de tamaño p_{radius}.

- Uno de los principales problemas de este algoritmo es la actualización del mapa de altura cuando su implementación no es iterativa a nivel de partícula (CPU), dado que pueden existir accesos concurrentes de varias partículas a una misma posición. Además, cualquier celda de este mapa almacena valores flotantes en el intervalo $[0, 1]$, mientras que GLSL sólo permite realizar operaciones atómicas con valores enteros (con, o sin signo). Esto supone la modificación del tipo de dato contenido en el mapa de altura, que deberá ser un entero (con signo, para prevenir *overflows* de *unsigned*). Así, el intervalo en que se encuentran estos valores pasaría a ser $[0, M]$, sea M un factor multiplicador que ya describimos en el cálculo de tangentes en GPU. Un valor se consulta como $\frac{map[i]}{M}$, y se escribe como $map[i] * M$.

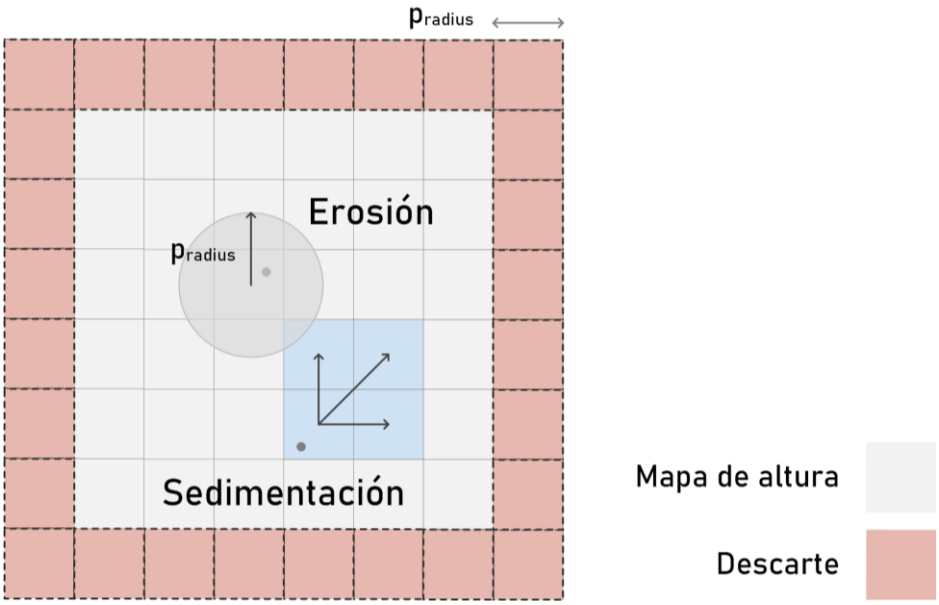


Ilustración 114. Representación de un mapa de altura donde se ilustran las áreas de influencia de los procesos de erosión y sedimentación.

- La posición inicial de todas las partículas se obtiene de una distribución uniforme que produce valores en $[p_{radius}, map_{size} + p_{radius}]$. Nótese como el tamaño del mapa es $map_{size} + p_{radius} * 2$, por lo que este intervalo evita generar partículas en posiciones cuyo radio implica el acceso a posiciones fuera del mapa. Igualmente, se utiliza una distribución uniforme porque se considera que todo el terreno debe erosionarse, pero cualquier otro enfoque es también válido. Por ejemplo, podríamos utilizar la función de ruido de Perlin para extraer una textura 2D y lanzar particulas únicamente donde se supera un umbral, o podríamos guiarnos exclusivamente por la instanciación en las zonas más altas del mapa.

Algunos ejemplos de terrenos obtenidos mediante el proceso que aquí se describe se muestran en la Ilustración 115.

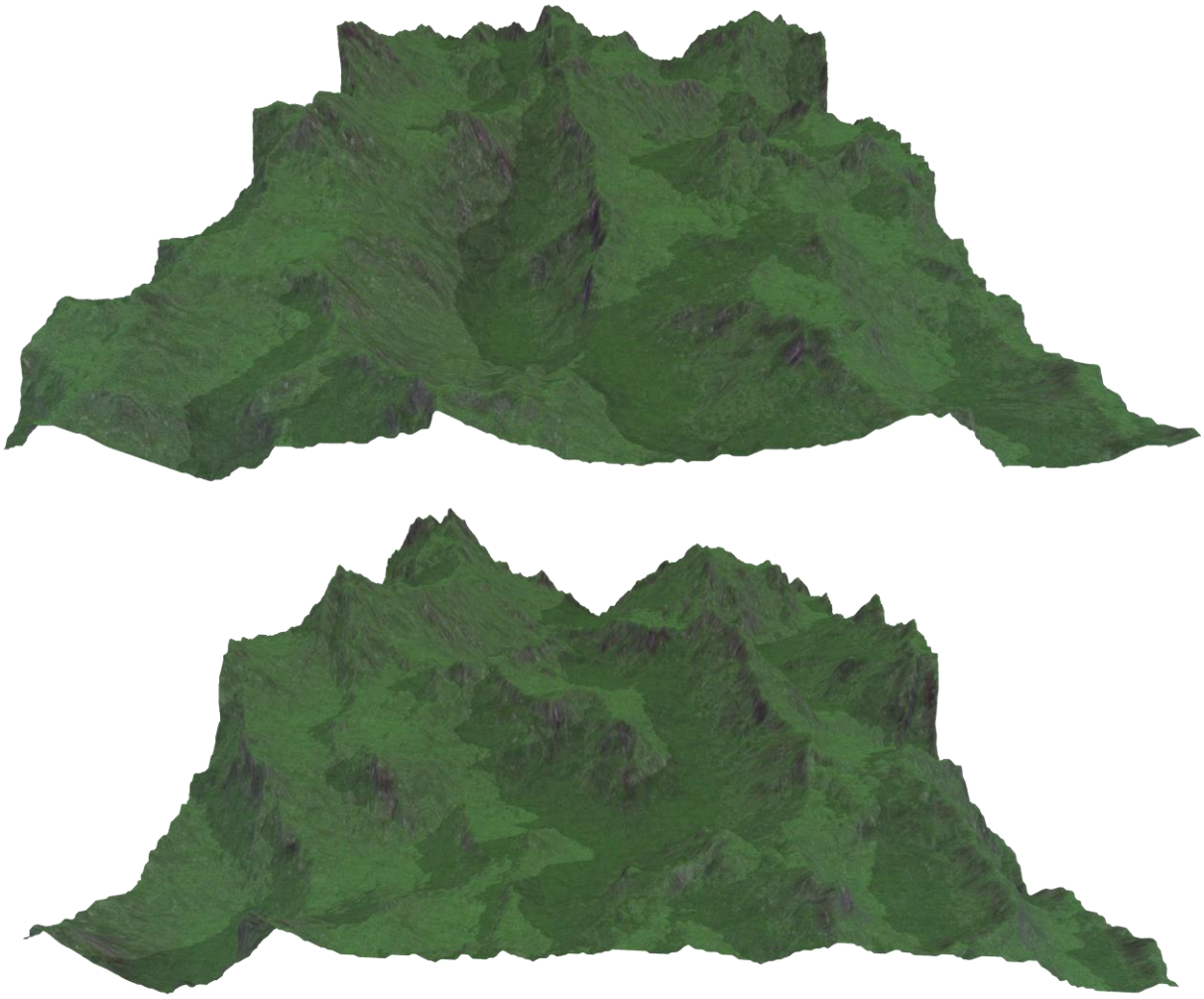


Ilustración 115. Terrenos obtenidos tras aplicar el algoritmo de erosión.

A continuación, se documentan los valores empleados en la erosión de todos los mapas que se muestran en ilustraciones posteriores, especialmente pensando en trabajos futuros (Tabla 40).

Parámetros de erosión			
Parámetro	**Valor**	**Parámetro**	**Valor**
$height$	14	$p_{particles}$	200.000
map_{size}	320	$p_{evaporate}$	0.01
$perlin_{freq}$	0.01	$p_{gravity}$	4
$perlin_{gain}$	0.5	$p_{inertia}$	0.1
$perlin_{lacunarity}$	2	$p_{maxLifetime}$	30

$perlin_{octaves}$	6	$p_{minSlope}$	0.01
$p_{deposition}$	1	$sediment_{capacity}$	4
p_{erode}	0.3	$water_{start}$	1
p_{radius}	3	$speed_{start}$	1

Tabla 40. Valores empleados en los parámetros del algoritmo de erosión.

A partir del proceso de erosión se extrae un nuevo mapa de altura, que no es más que una alteración del primer mapa, el cual se obtuvo a partir de una función de ruido de Perlin. En cualquier caso, cuando el mapa de altura se aplica sobre el plano inicial mediante la técnica de *displacement mapping*, se trasladan las posiciones a lo largo de la normal del plano, pero las normales no se vuelven a calcular. Esto no sucede para los triángulos del plano: una vez modificados los vértices, es posible calcular fácilmente su nueva normal. Sin embargo, si recibimos un *buffer* de geometría esto no es tan sencillo, dado que habría que comprobar la normal de todos los triángulos en los que participa un vértice, y ponderarlas de alguna manera para obtener una normal final. En definitiva, la correcta iluminación del terreno, que depende en última instancia de las normales de sus vértices, dependerá por completo de un mapa de normales, que se integra dentro de la técnica de *bump mapping*.

No es relevante esta técnica en este apartado, sino en el desarrollo de la aplicación gráfica, y por tanto, nos centraremos en la generación del mapa de normales. Dada la textura de un mapa de alturas, elaborada tras el proceso de erosión, y de tamaño map^2_{size}, es posible calcular las normales del terreno mediante un filtro de Sobel. Para ello no se emplea un *compute shader*, sino que al involucrar una textura es preferible un *shader program* que *renderice*, sobre un cuadrado que cubre la ventana completa, el resultado de dicho filtro. Este resultado se podrá capturar y recuperar a través de un FBO cuyo comportamiento se engloba bajo la entidad *FBOScreenshot*, la cual nos permite obtener la escena capturada mediante un objeto *Image*, que a su vez podrá integrarse en una textura (no es más que un vector de bytes).

Por otro lado, el operador de Sobel (Sobel and Feldman 2015) es un algoritmo clásico en la detección de bordes, permitiendo, en su versión básica, producir una imagen en escala de grises donde las superficies más planas toman el valor cero. El algoritmo consiste únicamente en la aplicación de dos matrices de tamaño 3x3 sobre

cada punto de una imagen, de tal manera que cada valor central de una matriz se aplica sobre el punto de referencia, y el resto de valores se aplican sobre sus vecinos. Las dos matrices que deben aplicarse se definen como sigue:

$$M_x = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}, \quad M_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} \tag{2.65}$$

Dado este tamaño de matriz, será necesario consultar el valor del mapa de altura hasta en ocho posiciones. Nótese como no es necesario extraer el valor en el punto (x, y) actual, dado que siempre se multiplicaría por 0, tanto en M_x como en M_y. La aplicación de ambas matrices no se corresponde con una mutiplicación, sino con una convolución (Ilustración 116), de tal manera que se multiplican valores que coinciden en un mismo punto. Lo que se obtiene finalmente no son más que dos sumas, sum_x, sum_y, que no pueden formar aún el mapa de normales.

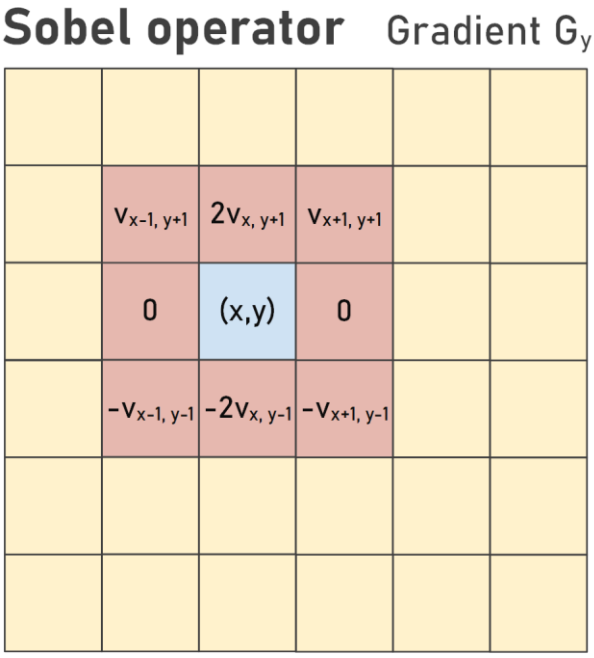


Ilustración 116. Representación del concepto de convolución mediante un gradiente del operador de Sobel.

Además de sum_x, sum_y, que harán las veces de (x, y), se considera un tercer valor de profundidad, $depth$, calculado como $\frac{1}{extrusion}$, sea este un valor indicado desde el exterior del *shader*. Mayor extrusión representará normales más acentuadas en la textura. Podemos considerar entonces un punto 3D, $(sum_x, sum_y, depth)$, a partir del cual es posible extraer un factor de peso en la formulación final. Dicho factor se ha denominado $scale$.

$$scale = \frac{0.5}{\sqrt{sum_x^2 + sum_y^2 + depth^2}} \tag{2.66}$$

Así, el valor final del mapa de normales en un determinado punto se calcula como sigue:

$$rgb = (0.5 + sum_x * scale, 0.5 + sum_y * scale, 0.5 + depth * scale) \tag{2.67}$$

Esto último que aquí se define no es más que una traslación de valores de un espacio $[-r, r]$ hacia $[0, \frac{r}{r-(-r)}]$, o lo que es lo mismo, $[0, \frac{r}{2r}]$. Por tanto, cada valor del punto 3D obtenido se desplaza como sigue:

$$\{x, y, z\} = \frac{\{x, y, z\} + r}{2r} \tag{2.68}$$

Un ejemplo de mapa de normales obtenido a partir de un mapa de altura, y el flujo que aquí se describe, se muestra en la Ilustración 117. Dicha imagen también representa el resultado de un operador de Sobel, el cual se obtiene prescindiendo de todas aquellas fórmulas posteriores a la convolución, y calculando G como $\sqrt{G_x^2 + G_y^2}$, sea G_x o G_y el resultado de una convolución.

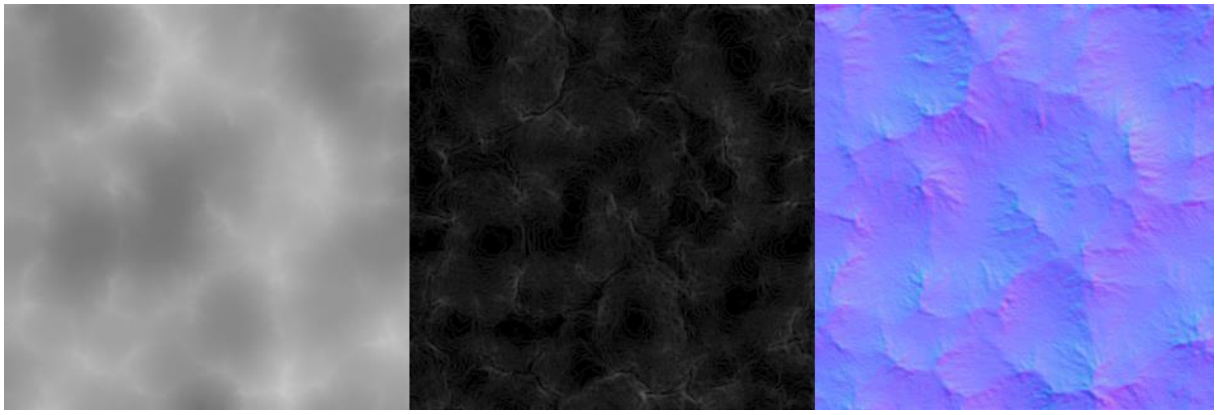


Ilustración 117. Mapa de normales y su respectivo de mapa de altura (izquierda). Resultado de operador de Sobel (medio).

Otro aspecto importante, más allá del algoritmo de erosión, es el renderizado del propio terreno, para lo cual se debe alterar el *shader* clásico de iluminación de triángulos, expuesto en el Anexo, con el fin de integrar nuevas texturas. Un terreno se ha definido como una combinación de dos texturas, roca y césped. En primer lugar, cada textura podrá tener una frecuencia de repetición, evitando así que ambas dependan de una misma coordenada de textura, calculada en la generación del plano a partir de un valor $textCoord_{max}$. La combinación de ambas texturas vendrá además

dada por una operación de *blending*. Un factor importante es la predominancia de una textura u otra será la pendiente del terreno en una posición concreta, pero además es posible modificar dicho factor en función de un peso de la textura de roca, $rock_{weight}$. Es necesario considerar que la normal introducida en el *fragment shader* procede del mapa de normales antes descrito, y por tanto, no es tan fácil de comprender como en el caso de que la normal se hallara en el mundo.

Un punto de máxima pendiente en el mapa de normales podrá tener cuatro normales: $(\mathbf{0},\mathbf{0},\mathbf{1}),(\mathbf{0},\mathbf{0},-\mathbf{1}),(\mathbf{1},\mathbf{0},\mathbf{0})$ y $(-\mathbf{1},\mathbf{0},\mathbf{0})$. En la práctica, dos operaciones de producto escalar son suficientes: $|\boldsymbol{normal} * (\mathbf{0},\mathbf{0},\mathbf{1})|$ y $|\boldsymbol{normal} * (\mathbf{1},\mathbf{0},\mathbf{0})|$ (Ilustración 118). La suma de ambos valores nos da una idea aproximada de cómo de inclinado está el terreno, y dicha suma podrá ser multiplicada por $rock_{weight}$ para ponderar aún más el peso de la roca en una operación de combinación de ambas texturas (Ilustración 119):

$$mix\left(grass_{texture}(rgb), rock_{texture}(rgb), rock_{factor}\right) = \\ = (1 - rock_{factor}) * grass_{text}\,(rgb) + rock_{factor} * rock_{text}\,(rgb) \tag{2.69}$$

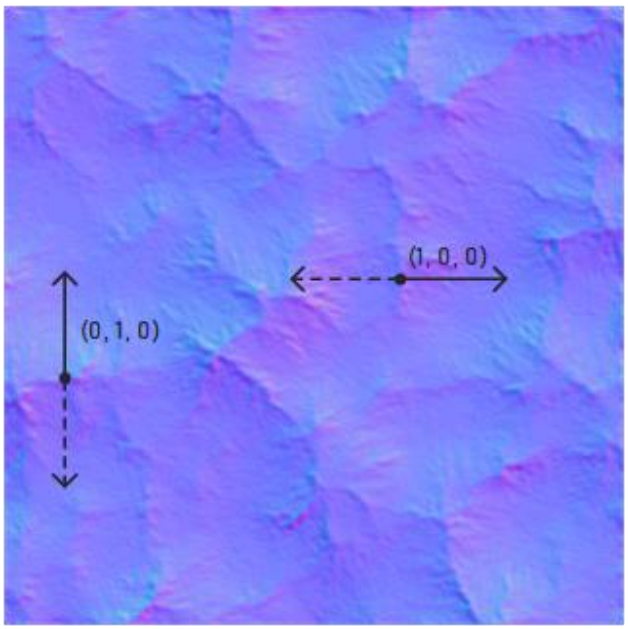


Ilustración 118. Representación de los vectores indicadores de máxima pendiente en un mapa de normales.

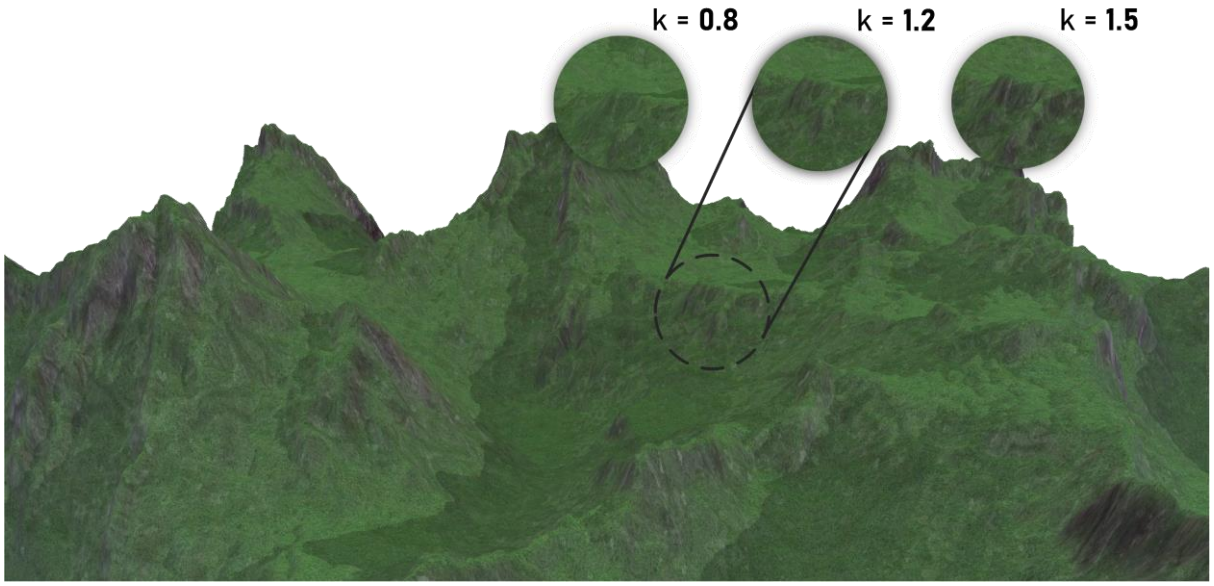


Ilustración 119. Detalles de una misma escena donde se aplican diferentes pesos asociados a la textura de roca.

El sistema desarrollado se ofrece, con relativa facilidad, a producir una animación que nos permita comprobar la posición de las particulas tras k iteraciones (Ilustración 121). También es cierto que el algoritmo no es fácilmente reanudable, dado que una partícula posee velocidad, agua, dirección, etc, aunque ciertamente podrían almacenarse dichos valores. En cualquier caso, si partimos siempre de un mismo estado inicial (por ejemplo, mismas posiciones iniciales de partículas), es posible ejecutar K iteraciones, en la siguiente petición $K + 1$, y así sucesivamente, aunque siempre se parte de cero. En la representación de esta animación sucede algo interesante, debido a que las posiciones iniciales se generan en $[p_{radius}, map_{size} + p_{radius}]$, por lo que se debería observar una parte del terreno donde no alcanzan las partículas (los bordes que más tarde se descartan), aunque la posición calculada en el algoritmo se puede recalcular como $pos * \frac{map_{size_{border}}}{map_{size}}$. Otra solución, menos elegante, es la reducción del tamaño del terreno para ajustarse a map_{size}, dado que hasta ahora lo descrito no es más que 1) un recorte del mapa, y 2) la aplicación de dicho mapa reducido al terreno original.

Dado el gran número de partículas que se emplean, en este caso 200.000, es necesario incluir algún mecanismo que nos permita su *renderizado* (en cualquier otro caso, sería bastante difícil producir una animación relativamente continua). Para ello se emplea lo que se denomina ***instancing***: se indica la geometría y la topología de una única instancia, pero se puede *renderizar* esta misma tantas veces como

deseemos. Parece evidente que será necesario un VBO donde se indique la posición de cada partícula, la cual será recuperada en el *vertex shader*, permitiendo así la actualización de la posición. En este caso, a la posición también se añade un valor, constante, que indica la altura inicial de la partícula, lo que nos permitirá visualizar en iteraciones más tardías como buena parte de las partículas iniciales se concentran en valles. El algoritmo no se ajusta por completo a un comportamiento físico realista, y por esta razón, es común ver partículas que ascienden, o que, tras muchas iteraciones, siguen en puntos del mapa que no son depresiones.

Para finalizar este apartado, se mide el tiempo de respuesta del algoritmo de erosión frente a diferentes cantidades de partículas e iteraciones (Tabla 41). El resto de parámetros se mantienen tal y como antes se describían. Aunque es cierto que estos escenarios nos pueden ayudar a conocer el rendimiento del algoritmo, también es necesario considerar que no es habitual, ni útil, emplear un gran número de partículas, dado que un terreno demasiado erosionado tampoco es un resultado válido en muchos casos. Por otro lado, existen ciertas situaciones en las cuales una partícula (y un hilo) podrían finalizar su proceso de erosión, y por tanto, es posible que exista una mayor varianza en los tiempos de respuesta (especialmente en GPU), en función de las partículas que finalizan prematuramente o de la iteración en que lo hacen. Como en pruebas anteriores, se mide el tiempo necesario para completar el algoritmo en CPU, incluyendo definición y lectura de *buffers*, y tiempo de *kernel*, que se ajusta únicamente al tiempo empleado en GPU.

Tiempo medio de resolución del proceso de erosión				
50.000 partículas. 30 iteraciones				
Ejecución	**Tiempo de respuesta (ms)**		**Tiempo de respuesta de kernel (ms)**	
1	6,371		2,846	
2	4,798		2,240	
3	6,532		2,054	
4	3,787		2,442	
5	5,200		2,236	
	Tiempo medio (ms)	5,337	**Tiempo medio (ms)**	2,363

200.000 partículas. 30 iteraciones				
Ejecución	**Tiempo de respuesta (ms)**		**Tiempo de respuesta de kernel (ms)**	
1	15,616		8,385	
2	23,619		7,141	
3	19,059		6,621	
4	16,560		6,780	
5	20,920		8,365	
	Tiempo medio (ms)	19,154	**Tiempo medio (ms)**	7,458

500.000 partículas. 30 iteraciones				
Ejecución	**Tiempo de respuesta (ms)**		**Tiempo de respuesta de kernel (ms)**	
1	33,646		22,942	
2	35,477		18,832	
3	35,009		18,417	
4	36,229		23,086	
5	32,844		23,003	
	Tiempo medio (ms)	34,641	**Tiempo medio (ms)**	21,256

500.000 partículas. 64 iteraciones				
Ejecución	**Tiempo de respuesta (ms)**		**Tiempo de respuesta de kernel (ms)**	
1	48,523		36,027	
2	50,586		38,560	
3	44,606		41,048	
4	47,194		40,748	
5	52,511		36,943	
	Tiempo medio (ms)	48,684	**Tiempo medio (ms)**	38,665

Tabla 41. Tiempo medio de respuesta para resolver el problema de la erosión.

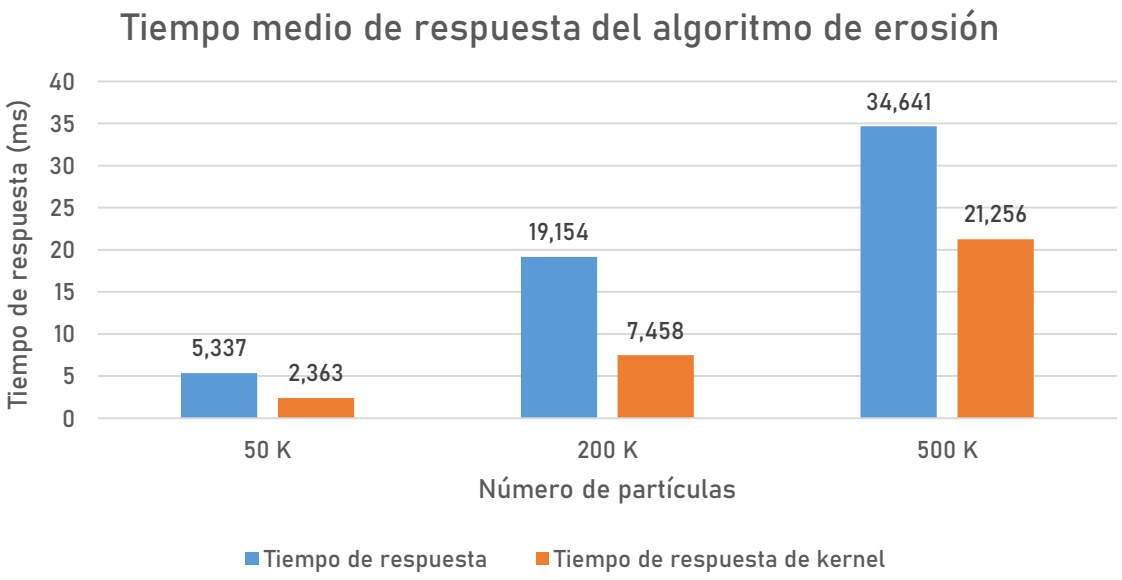


Ilustración 120. Diagrama de barras donde se muestra el tiempo medio de respuesta obtenido para varias configuraciones con diferente número de partículas y un mismo número de iteraciones, 30.

El artículo que propone este mismo algoritmo (Bayer, 2015) contiene algunos detalles sobre el tiempo de ejecución en CPU, permitiéndonos conocer así si el resultado aquí obtenido es realmente bueno. Los tiempos documentados en el artículo no se extraen mediante un ordenador de grandes características, y por tanto, habría que considerar que el tiempo de ejecución en CPU podría verse reducido notablemente hoy en día. En cualquier caso, puede ser una buena referencia. Para un terreno de tamaño 256x256, se documentan algo más de 30 segundos (32.02) para llevar a cabo una erosión con 100.000 partículas y 64 iteraciones. En nuestro trabajo, somos capaces de resolver la erosión de 500.000 partículas, en 64 iteraciones, y en un mapa de tamaño 320x320 en algo más de 48 milisegundos (si consideramos todas las operaciones en CPU).

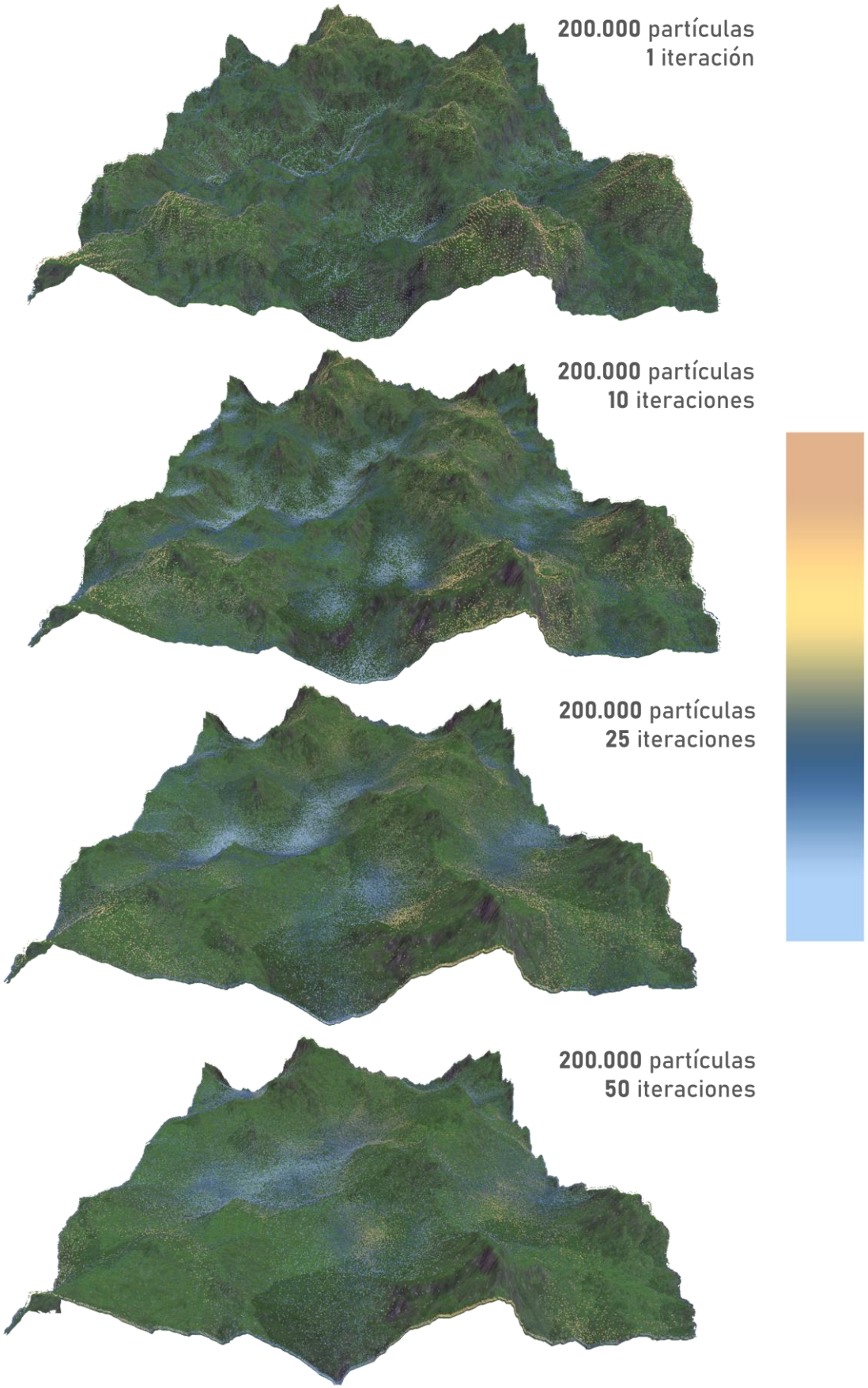


Ilustración 121. Sistema de partículas que indica la posición de estas tras cada iteración del algoritmo de erosión. El color de las partículas se corresponde con la altura inicial.

2.5.3 Diagrama de clases

En esta sección, como en todos los apartados anteriores, se introduce un diagrama de clases que contempla únicamente aquellas entidades afectadas por los cambios descritos en esta iteración (Ilustración 122). Dados los dos temas principales (e inconexos) de esta iteración, podríamos pensar que las entidades no presentan relación alguna, algo que no es necesariamente verdad, y de hecho, en una vista general de la aplicación podría hallarse la relación entre una simulación y un terreno a través de una escena que aúne ambas entidades. Se omiten algunos de los atributos de un terreno, los cuales almacenan valores por defecto que se asignan inicialmente a muchas de las variables que se representan en el diagrama.

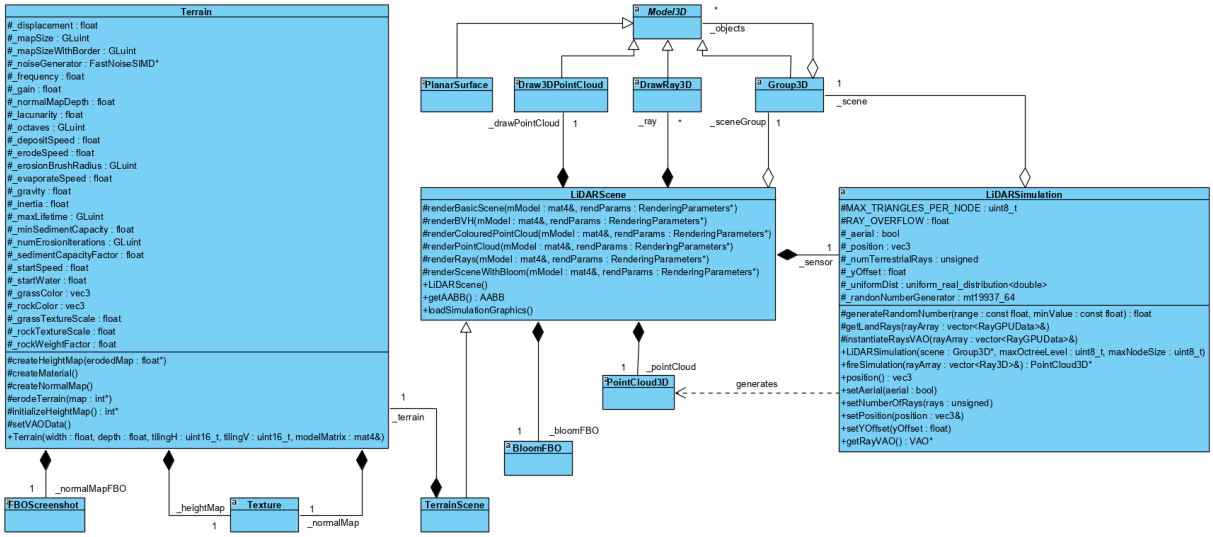


Ilustración 122. Diagrama de clases donde se muestran las entidades modificadas en la cuarta iteración.

2.6 Quinta iteración

El objetivo de la quinta iteración, como bien se describe en la planificación temporal de este trabajo, consiste en completar la aplicación gráfica hasta ahora desarrollada. Por tanto, son muchos los frentes que aquí se abren y que deben detallarse en diferentes secciones. Sin duda, esta es la iteración que introduce un mayor número de conceptos, y como tal, será una de las que más esfuerzo conlleve. Se procede a dividir este apartado en tantos puntos como objetivos se distingan.

2.6.1 Reflejos y refracciones

La introducción de materiales y superficies con reflejos y refracciones se justifica desde una posible interacción especial con las emisiones del dispositivo LIDAR. De hecho, buena parte de las funcionalidades que se implementan en esta iteración presentan cierto interés en la simulación LiDAR. Dicho interés no sólo se debe interpretar como una perturbación en la captura de la nube de puntos, sino que también se puede encontrar en cierto tipo de *rendering*.

La reflexión es la propiedad de una superficie que permite que esta refleje su entorno en función del ángulo de visión de un observador (LearnOpenGL, 2014). El ejemplo más clásico es un espejo, o el agua (veremos esta superficie en detalle). El concepto es muy simple: dada la posición p del espectador, y la posición de una superficie (de normal N), es posible calcular el reflejo del vecto de visión, V, respecto de N. El vector resultante, R, puede "impactar" con el punto de otra superficie, que será el que finalmente se refleje en la posición de nuestra superficie reflectiva (Ilustración 123). Mediante esta descripción se da por hecho que el punto impactado, si lo hubiera, pertenecería a otro modelo, lo cual no es necesariamente cierto, aunque esta situación se excede completamente del objetivo de esta sección.

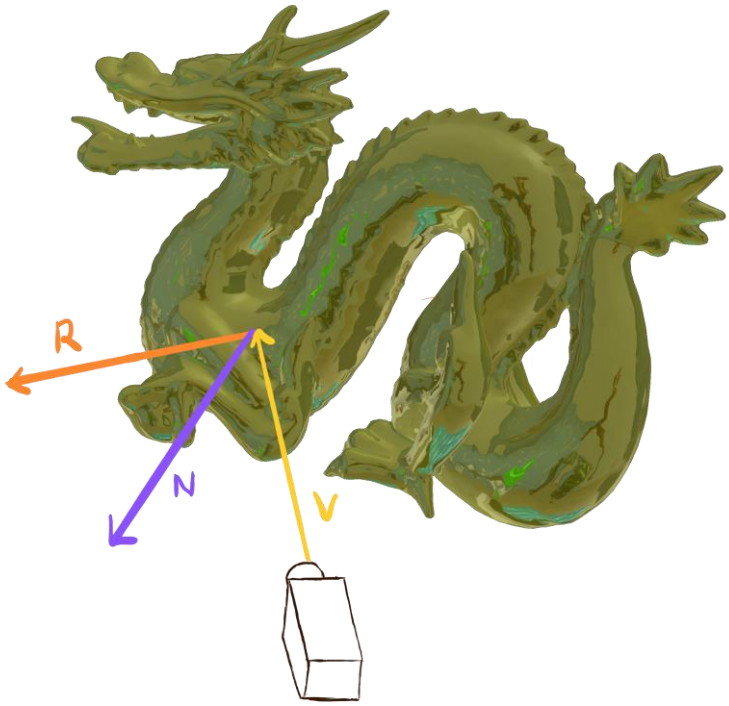


Ilustración 123. Representación de vectores empleados en el problema de la reflexión.

Un concepto comúnmente asociada a la reflexión es un *cubemap*, un cubo cuyas seis caras reflejan el entorno de una escena desde un determinado punto

(Ilustración 124). Podría definir el entorno de una escena (*environment mapping*), pero también podría ilustrar qué se observa desde un modelo de la escena. Tal y como se describe el problema de la reflexión en el párrafo anterior, esta se reduce a una intersección de un rayo con una superficie de la escena, pero esto es demasiado costoso (ya hemos visto a lo largo de este trabajo el coste de la resolución de intersecciones de un gran número de rayos con la escena). Una solución menos costosa, y análoga al problema de la simulación, es la captura de un *cubemap* desde el punto que ocupa el modelo reflectivo. Aunque produce buenos resultados, no es la manera más precisa y realista de calcular un reflejo, y prueba de ello es el caso extremo antes expuesto, que sí podría ser resuelto mediante un enfoque de resolución de intersecciones. En cualquier caso, la solución desarrollada radica precisamente en la obtención de un *cubemap*, para lo cual será necesario crear un nuevo constructor en la entidad *Texture*.

La definición de un *cubemap* no varía en gran medida respecto del resto de texturas hasta ahora empleadas. La principal diferencia radica en el uso del identificador `GL_TEXTURE_CUBE_MAP` en buena parte de las operaciones de OpenGL relacionadas con este punto, no sólo en la construcción de la textura, sino también en su aplicación. Además, no se define una única textura , sino seis, donde el identificador de la primera vendrá dado por `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, pudiendo acceder al resto mediante este mismo identificador más un valor i, sea este el índice de un bucle que permite definir las seis imágenes.

El resto de componentes necesarios para resolver la reflexión se encuentran disponibles en la aplicación:

- Una **cámara**, que debería hallarse en la posición del modelo. Por defecto, se encuentra en la posición $(0, 0, 0)$, y podrá modificarse mediante la matriz de transformación del modelo. El propósito de la cámara será tomar instantáneas del escenario que rodea al modelo (hasta 6), para lo cual será necesario rotar la cámara. Nótese como este efecto se podría conseguir mediante algunos de los movimientos implementados, pero es más sencillo y preciso modificar la posición de *lookAt*. Se puede llevar fácilmente a la práctica dado que es posible definir un vector con 6 direcciones, una por cara, de tal manera que $lookAt = eye + dir$. Además, se podrá iterar por cada dirección, generalizando así el proceso. Nótese como no sólo es

necesario modificar $lookAt$, dado que algunas de las caras requieren la modificación del vector up (más concretamente, la cara superior e inferior).

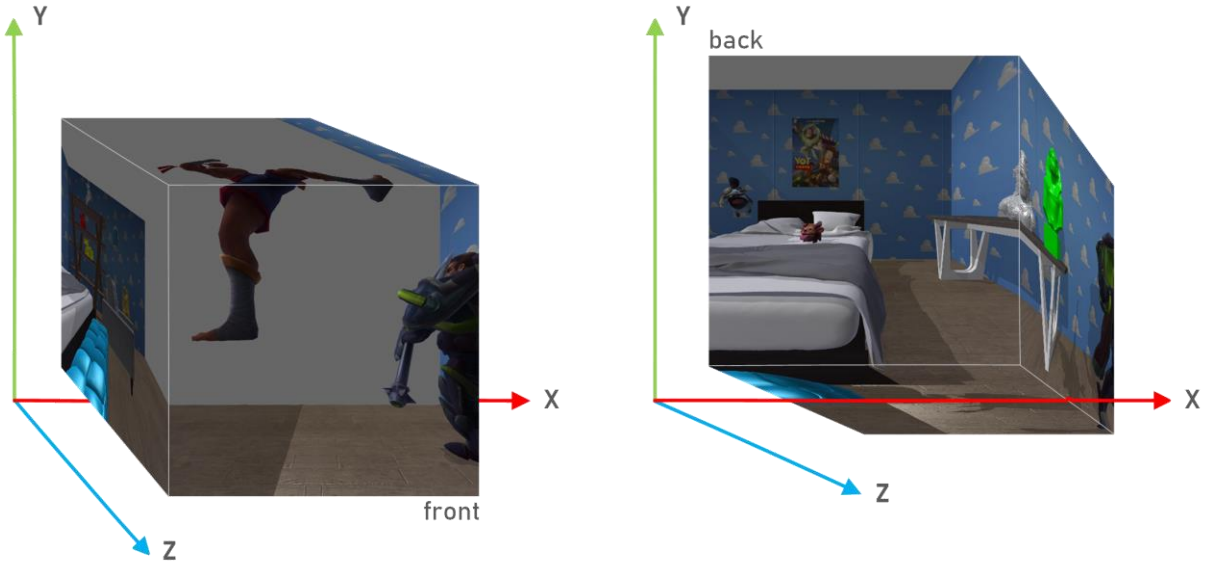


Ilustración 124. Representación de un cubemap donde se pueden observar sus seis caras.

- ***FBO***. Se debe combinar la cámara antes descrita con un *FBO* que capture los seis *renderizados* de la escena. Dicho *FBO* produce un objeto *Image**, bajo el que se engloban funciones como `flipVertically`, que permite voltear verticalmente la imagen resultante (será necesario). El conjunto de imágenes generadas se puede utilizar para inicializar una textura de tipo *cubemap*, de tal manera que el puntero de cada definición de `GL_TEXTURE_CUBE_MAP_POSITIVE_X + i` se vincula a cada una de dichas imágenes.

La inclusión de este tipo de superficie tiene otras tantas implicaciones en la aplicación:

- Su comportamiento se engloba en una entidad ***ReflectiveObject***, que recibe en su constructor la escena que le rodea. Nótese como esta entidad no representa un nuevo tipo de objeto por sí mismo, razón por la cual se construye a partir de cualquier otro modelo o componente (*ModelComponent*). La carga de un objeto reflectivo dependerá a su vez de la carga de cualquier otro objeto, del cual podemos extraer (*std::move*) la definición de su geometría y topología. En cualquier caso, puede existir una matriz de modelo por encima de dicho objeto, asociada a la superficie reflectiva, que debe igualmente aplicarse.

- La escena debe adaptar su comportamiento a esta nueva entidad. Más concretamente, en el *rendering* de sus triángulos recibe un parámetro adicional: una cámara. Dicha cámara podrá pertenecer a la escena en circunstancias normales, o al objeto reflectivo en ciertos casos.
- Un modelo define un método de *rendering* adicional, adaptado a los objetos reflectivos, los cuales deben ser los únicos que lo sobrecarguen. En dicho método se comprueba si se debe *renderizar* cierto modelo o no. Si un objeto reflectivo se encuentra *renderizando* la escena para hallar su *cubemap*, no deberá renderizarse este mismo objeto. Esto no sucede para la generación del mapa de sombras, dado que la propia sombra del objeto debería reflejarse en él.

Las escenas hasta ahora representadas son estáticas, lo cual sugiere que la obtención de un *environment mapping* podría llevarse a cabo únicamente en el inicio de la aplicación. Por tanto, la entidad *ReflectiveObject* deberá almacenar un valor booleano que indica si se ha obtenido el *cubemap* o no, en cuyo caso habrá que *renderizar* un escenario (Ilustración 125). La implementación realizada incluye en cualquier escena un vector de objetos reflectivos, de tal manera que antes de comenzar el *rendering* podrá comprobarse si todos los objetos disponen de su *cubemap*. Podría trasladarse esta comprobación al punto de carga, pero en dicho caso no se adaptaría a un posible trabajo futuro en el que se incluyeran animaciones.

Algorithm 18 *CaptureEnvironment*: Renderizado de imágenes de un cubemap

```
1: Definir vector de direcciones
2:
3: for each dir ∈ directions do
4:     camera.lookAt(camera.eye + dir)
5:     if dir.y < 0 then
6:         camera.up(0, 0, 1)
7:     else if dir.y > 0 then
8:         camera.up(0, 0, −1)
9:     else
10:        camera.up(0, 1, 0)
11:    end if
12:
13:    Bind FBO
14:    Limpiar buffers de color y profundidad
15:    Adaptar tamaño de viewport
16:
17:    Renderizar escena
18:
19:    images.push(fbo.getImage().flipImageVertically())
20: end for
21:
22: Bind FBO por defecto y restaurar tamaño de viewport
23: Construir cube map con images
24: Eliminar imágenes
```

Ilustración 125. Rendering de escena desde una instancia de ReflectiveObject.

Para *renderizar* la reflexión del modelo es necesario acudir a un *shader program*, *reflectiveTriangleMesh*, que se desarrolla a partir del programa de *renderizado* de triángulos que se documenta en el Anexo. Las principales modificaciones introducidas son las siguientes:

- *Vertex shader*. A partir de la posición y la normal del vértice en el mundo, así como la posición de la cámara, es posible calcular R, que se almacenará como una variable de salida, y por tanto, pasará por el *rasterizador*.
- *Fragment shader.* Dado R y un *cubemap* en una variable *uniform*, es posible consultar este *sampler* mediante una coordenada de textura 3D, $(-R_x, R_y, R_z)$. El uso de R_x o $-R_x$ dependerá de la definición dada de las imágenes.

Otro problema de este *rendering* es la asignación final de un color (¿cómo podemos combinar el reflejo y el color del modelo?). La solución más sencilla radica en una operación de *blending* tal que $rgb = mix(reflection_{rgb}, diffuse_{rgb}, diffuse_{weight}) = (1 - diffuse_{weight}) * reflection_{rgb} + diffuse_{weight} * diffuse_{rgb}$. Esta solución podría llegar a producir buenos resultados si se calculara (y añadiera a la imagen final) una única vez. El problema al que nos enfrentamos en una aplicación que emplea *multipass rendering* es la acumulación del mismo valor múltiples veces, a pesar de que $reflection_{rgb}$ suele presentar un peso muy reducido (este es el valor constante que se puede identificar con claridad en la ecuación de *blending*). Por esta misma razón, se definen al menos dos comportamientos mediante una subrutina:

- *mixColor*. *Blending* de ambos colores tal y como se ha descrito.
- *baseColor*, donde el valor RGB de la reflexión toma un peso muy reducido.

A continuación se muestran algunos ejemplos de superficies reflectivas empleadas en el primer escenario (Ilustración 126, Ilustración 127 e Ilustración 128):

Ilustración 126. Modelo de Lucy junto a su cubemap.

Ilustración 127. Rendering de superficie reflectiva. Modelo de Lucy.

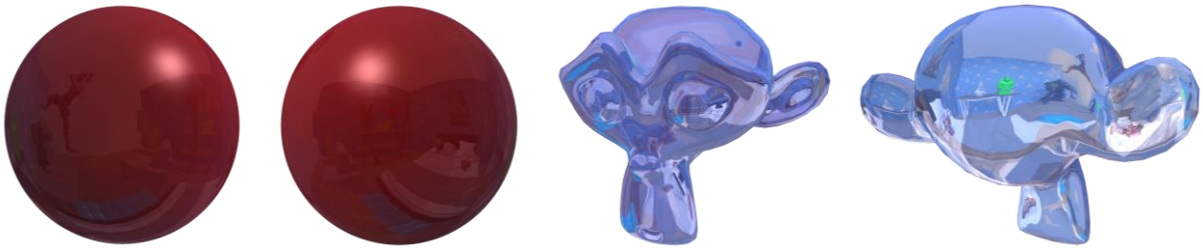


Ilustración 128. Rendering de dos superficies reflectivas diferentes.

Más allá de la reflexión, otro tipo de *environment mapping* es la refracción, un cambio de dirección de la luz al alcanzar un material diferente. Un buen ejemplo de esto mismo es el agua, donde todo aquello que se encuentra debajo de la misma suele verse, en cierto modo, distorsionado. En este caso, el vector R' no se representa como un rebote en la superficie, sino como un vector que traspasa la superficie y modifica su dirección. De nuevo, GLSL dispone de una operación `refract`, que recibe como antes V y N, aunque añade un nuevo valor flotante, eta, que representa el ratio de los

índices de refracción del medio. No es difícil encontrar listados de índices para diversos materiales. Por ejemplo, (LearnOpenGL, 2014) incluye los siguientes:

Índices de refracción de materiales	
Material	**Índice de refracción**
Aire	1
Agua	1,33
Hielo	1,309
Cristal	1,52
Diamante	2,42

Tabla 42. Índices de refracción de múltiples materiales. Fuente: (LearnOpenGL, 2014).

Podemos encontrar listados mucho más extensos en otras fuentes, pero la Tabla 42 puede ser suficiente. A partir de aquí, podríamos calcular lo que OpenGL denomina eta, el ratio entre los índices de los dos materiales por los que pasa el rayo. En nuestra implementación siempre se considera que el primer material es el aire (índice de refracción 1), por lo que sólo debe indicarse el índice de un material que deberá especificar el usuario. Un objeto reflectivo define un *enum* con los 5 materiales identificados en la Tabla 42, así como una tabla *hash* que relaciona cada material con su índice.

Nótese como una superficie reflectiva y refractiva no son excluyentes, y como tal, se incluyen en un mismo *shader*. Fuera de este, la inclusión del comportamiento de refracción no supone ningún cambio más allá de permitir al usuario indicar el material del objeto. Dentro del *shader*, se debe incluir el cálculo del vector refracción R', que pasará por el *rasterizador*:

$$R' = refract(V, N, \frac{1}{refractive_{index}}) \tag{2.70}$$

No todos los objetos reflectivos presentan refracción, y como tal, se incluye un peso, $reflective_{weight}$, que pondera dicha relación mediante la fórmula de *blending* que hemos visto anteriormente. Por tanto, a la mezcla de la refracción y la reflexión se le denomina $environment_{rgb}$, que será el color que pase a combinarse con $diffuse_{rgb}$ como antes se describía.

Algunos ejemplos de superficies refractivas se muestran en la Ilustración 129:

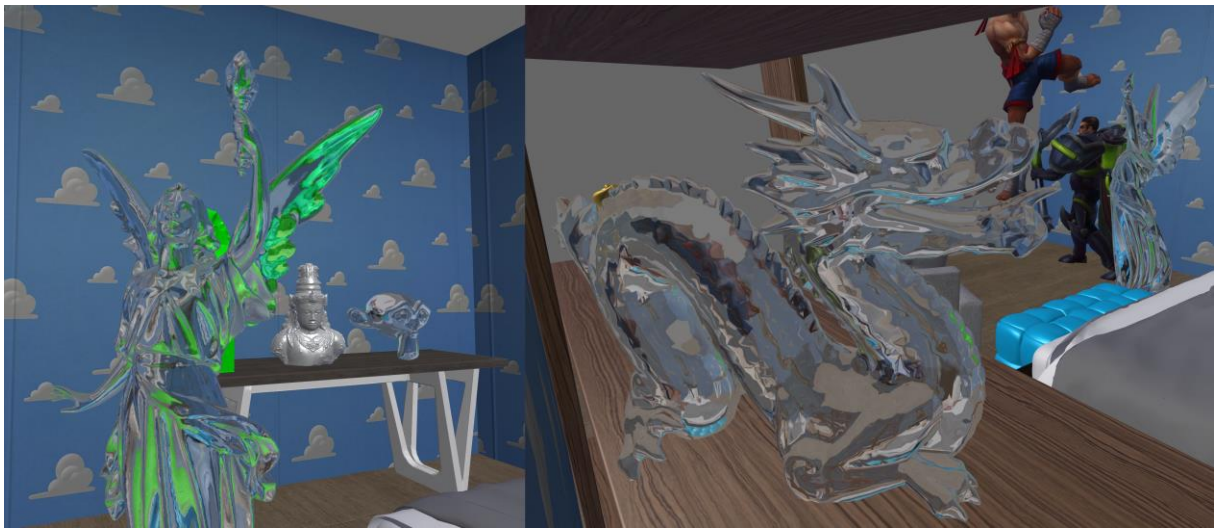


Ilustración 129. Rendering de tres modelos con superficies reflectivas y refractivas.

2.6.2 Generación de agua

El agua es probablemente una de las superficies más relevantes en un escaneo LiDAR, y como tal, se incluye en el segundo escenario planteado, el terreno. No parte de unos principios muy distintos a los ya expuestos en Reflejos y refracciones, dado que el agua se puede plantear como una superficie plana que distorsiona el *renderizado* del terreno inferior para generar una falsa sensación de movimiento.

Más allá de las distorsiones que puedan aplicarse, es necesario volver a los conceptos de reflexión y refracción. A diferencia de antes, ahora es fácil interpretar dónde podrían intersectar los rayos que surgen de ambos conceptos. Por ejemplo, la refracción de V respecto de la normal del plano, N, en un punto de nuestro lago, impactará con la superficie del terreno que se encuentra debajo de dicho lago. En el caso de la reflexión, el rayo resultante podrá impactar, pero no necesariamente es cierto, con algún punto de la superficie del terreno, aunque parece evidente que dicho punto se encuentra por encima del agua.

En definitiva, se podrían distinguir dos texturas diferentes que deberían aplicarse sobre el agua para crear los efectos de reflexión y refracción. OpenGL dispone de un mecanismo que facilita la distinción de ambas texturas, teniendo en cuenta que es posible diferenciarlas mediante un plano de corte. A partir de dicho plano es posible calcular la distancia de un punto a su superficie, de tal manera que si esta es negativa, el punto se descarta. Esto es lo que se conoce como *clip distance*, y es posible aplicarlo en un *vertex shader*. Además, se puede incorporar al *shader* del terreno de manera natural, de tal manera que mediante una subrutina es posible

escoger la aplicación o no del *clipping*. En cualquier caso, es necesaria la ecuación del plano que representa el agua, teniendo en cuenta que este se inicializa en $Y = 0$ y que podrá recibir una matriz de transformación. Por tanto, la ecuación de un plano (paralelo al eje X) se describe como sigue:

$$Ax + By + Cz + D = reflection * y - refraction * y + \left(M * (0,0,0,1)\right)_y = 0 \qquad (2.71)$$

$reflection$ y $refraction$ serán dos variables que contendrán 0 o 1 en función del escenario, de tal manera que ambas no podrán tener el mismo valor. Mediante la matriz M y el origen de coordenadas es posible hallar el desplazamiento horizontal, un valor constante. Por tanto, la distancia de un punto al plano se puede hallar mediante una simple multiplicación de valores en la coordenada Y y la suma de una constante. Cuando es necesario hallar la reflexión, la normal del plano es $(0,1,0)$, y para hallar la refracción debería considerarse $(0,-1,0)$, de tal manera que en este segundo caso los puntos debajo del plano obtendrían distancias positivas.

Parece evidente que, como antes, será necesario un FBO para capturar ambos *renderings*, y como tal, también será necesaria una entidad, ***Water***, que englobe todo este comportamiento. La principal diferencia con el escenario antes empleado radica en la centralización del proceso de captura *offscreen*. Una escena no conoce en absoluto cuántos o qué objetos presentan superficies reflectivas, a priori, pero esto sí sucede en la escena del terreno, dado que el agua se considerará como un elemento estático que debe presentarse siempre. Por tanto, será la entidad *TerrainScene* la que tome la iniciativa para obtener ambas texturas en tiempo real, aunque igualmente será necesario que *Water* enlace su FBO o aplique todas las variables necesarias antes de ejecutar el *shader* de *rendering*.

Algunos ejemplos de texturas que se obtienen mediante el proceso descrito se muestran en la Ilustración 130, donde las zonas ocluidas de la textura de refracción se muestran mediante líneas discontinuas en la composición final:

Ilustración 130. Texturas de reflexión y refracción para el rendering de agua.

El ratio de aspecto que se observa en las texturas no repercute en el resultado final, teniendo en cuenta que el acceso se produce mediante coordenadas de textura en el intervalo $[0, 1]$. Podría ajustarse el tamaño del FBO para adaptarlo al tamaño actual del *viewport*, pero en la práctica esto no es necesario, e involucra más operaciones en un nuevo proceso de captura (por simples que estas sean). También cabe mencionar que el *buffer* de color se limpia con un valor completamente diferente al que se suele emplear, con el objetivo de representar el cielo, que es lo que debería visualizarse en nuestra escena en lugar de un color estático de fondo. Se ha mantenido este último proceso muy simple, pero podría ser interesante añadir cualquier forma, por ejemplo, similar a las nubes, sobre el fondo de color constante (una textura de ruido de Perlin es suficiente, y más cuando el agua va a distorsionar su apariencia).

Sin embargo, se da la situación de que pueden existir ciertos puntos del agua, especialmente en los bordes de colisión con el terreno, donde pueden generarse *artefacts* debido a que la distancia al plano es cero, y por tanto, se representa el color por defecto del *buffer*. También es necesario indicar que esta técnica funciona especialmente bien en visualizaciones a pie de terreno, como podría suceder en un

videojuego, pero no en visualizaciones desde un punto arbitrario, como se hará en este trabajo. Lo que sucede en estos casos es que la textura reflexión contiene claramente zonas de cielo que producirán un resultado no deseado. Una solución, aunque no se trata de la más precisa y elegante, es extender el *rendering* de la reflexión, de tal manera que se puede considerar que el plano de agua se encuentra varias unidades por debajo de donde realmente se encuentra, e incluso puede *renderizarse* por completo, siendo esta última la opción la mejor. Es necesario observar que en la textura de refracción se representan zonas ocluidas del terreno que nunca van a aplicarse en el *rendering*. Por tanto, eliminar las zonas ocluidas implica que podría llevarse a cabo un único *rendering* donde se combinaran ambas texturas. Además, a diferencia del capítulo anterior, estas texturas deben *renderizarse* en cada *frame*, dado que dependen del punto de visión, por lo que nunca está de más reducir la carga de trabajo *offscreen*.

En este punto, en el que no se aplica distorsión alguna, podríamos al menos acceder a la textura generada para obtener la reflexión y refracción en cierto punto de la superficie del agua. Dada la matriz de proyección, y el punto p de la superficie, entonces se obtiene:

$$clipSpace = M_{viewProj} * vec4(p, 1)$$

$$ndc = \frac{\frac{clipSpace_{xy}}{clipSpace_{w}}}{2} + 0.5 \tag{2.72}$$

$$reflection_{textCoord} = vec2\left(ndc_x, 1 - ndc_y\right)$$

$$refraction_{textCoord} = vec2\left(ndc_x, ndc_y\right)$$

Tras consultar la textura mediante ambas coordenadas de textura, es posible combinarlas de cualquier manera posible, teniendo en cuenta que también debe introducirse la componente difusa del agua en la ecuación.

Una herramienta frecuente cuando se desea distorsionar una textura es lo que se conoce como *DuDv map*, una textura que toma un color característico debido a que sólo los canales Red y Green toman valores diferentes de cero. Esta se puede construir fácilmente a partir de un mapa de normales (por ejemplo, en Adobe Photoshop), e igualmente, el mapa de normales se puede construir a partir de una función de ruido. Esto implica que un *DuDv map* se podría generar proceduralmente,

aunque en nuestro trabajo se escogen dos texturas estáticas, que representan una misma distorsión a lo largo de las diferentes ejecuciones de la aplicación (Ilustración 131).

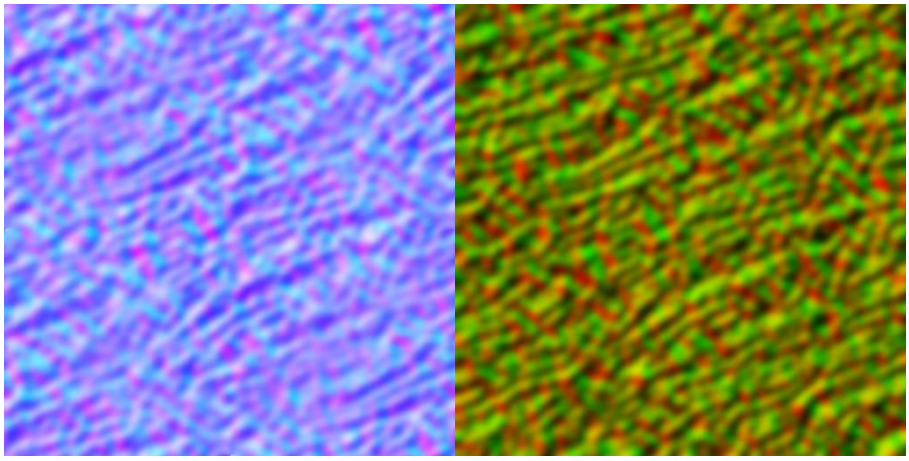


Ilustración 131. Mapa de normales (izquierda) y DuDv map (derecha).

El objetivo de esta textura de ruido es alterar las dos coordenadas de textura antes calculadas, aunque siempre debe existir cierta continuidad, justo lo contrario que sucedería con un *white noise*. Nótese como la distorsión se puede llevar más allá de una representación 3D, dado que podemos incluir la dimensión tiempo para variar dicha distorsión a lo largo de esta nueva dimensión. Para ello, consideraremos un valor t, que comienza en 0, e irá aumentando su valor mediante pequeños incrementos. La cuestión se centra ahora en qué coordenadas se debe añadir dicho valor. Es cierto que se podría añadir tanto al valor X como Y de las coordenadas de textura del punto p, lo cual produce una traslación (y un patrón) en diagonal. Es fácil observar que cada cierto tiempo se volverá a repetir el patrón, aunque el incremento de t sea muy pequeño (se modifica en cada *frame*).

Una solución que permite evitar dicho patrón consiste en tomar un valor (*Red*, *Green*) de la textura como una nueva coordenada de textura, que a su vez podemos trasladar considerando t (de nuevo), y ya sí, utilizaríamos esa misma coordenada para recuperar un vector de distorsión que se sumará a las coordenadas de textura antes calculadas. La distorsión obtenida es igualmente aplicable a las coordenadas de textura de *shadow mapping*, de tal manera que la distorsión también afecta a la sombra que se proyecta sobre el agua.

$$
\begin{aligned}
&textCoord_{in} = textCoord * TILING \\
&textCoord_{distorted} = texture\left(noise, \left(textCoord_{in_x} + t, textCoord_{in_y}\right)\right)_{xy} * W \\
&textCoord_{distorted} += \left(textCoord_{distorted_x}, textCoord_{distorted_y} + t\right) \\
&textCoord_{distorted} = texture(noise, textCoord_{distorted}) * 2 - 1 \\
&reflection_{textCoord} = clamp(reflection_{textCoord} + textCoord_{distorted}, \varepsilon, 1 - \varepsilon) \\
&refraction_{textCoord} = clamp(refraction_{textCoord} + textCoord_{distorted}, \varepsilon, 1 - \varepsilon)
\end{aligned}
\tag{2.73}
$$

Nótese como t se introduce tanto en X como en Y, aunque se produce en líneas diferentes para evitar una traslación exactamente diagonal. Igualmente, es posible reducir la variación entre *frames* a través de lo que se ha denominado como $TILING$, algo que también podría controlarse en el constructor de una superficie plana ($textCoord_{max}$). El peso de la distorsión vendrá a su vez determinado por un valor W, que en nuestro trabajo se establece como 0.1. La Ilustración 132 muestra un posible escenario que aplica la distorsión descrita sobre el agua.

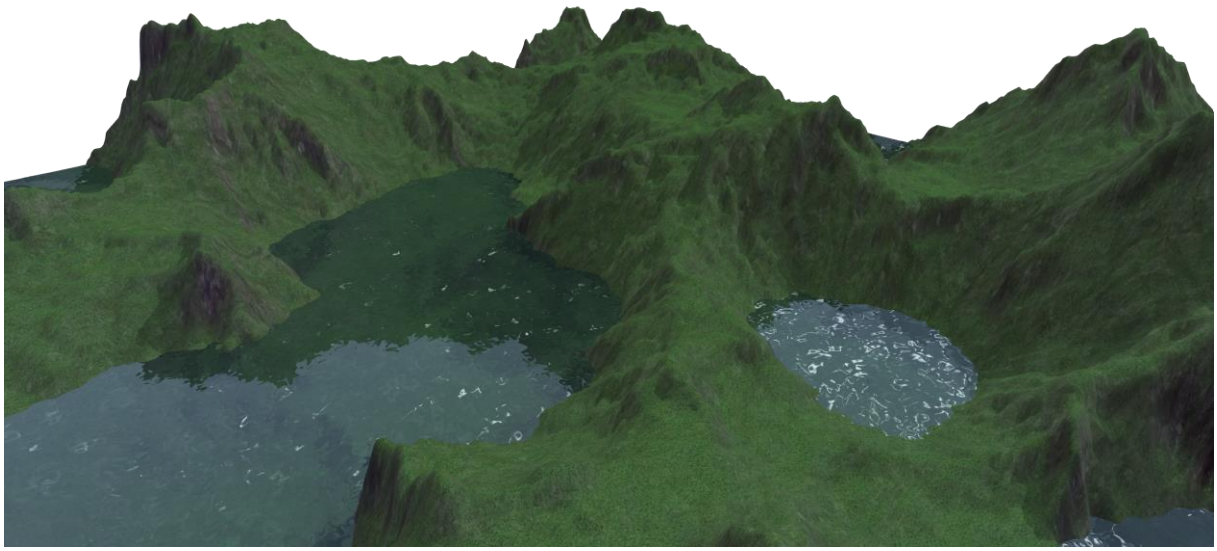


Ilustración 132. Rendering de agua con distorsión.

Los valores RGB obtenidos de las coordenas de textura de reflexión y refracción se podrían combinar mediante una simple operación de *blending*, donde $\propto$ puede representar el peso de cualquiera de las componentes (y $1-\propto$ para la componente restante). Sin embargo, es posible determinar con mayor precisión el valor de $\propto$, mediante lo que se denomina como *fresnel*. En un escenario real donde se sitúa una superficie plana de agua (por simplificación), la reflexión toma más peso

a medida que nuestro ángulo de visión con dicha superficie decrece. Sin embargo, cuando visualizamos el agua formando un ángulo de 90 grados con la superficie (es el caso más extremo), se puede llegar a visualizar más allá del agua. No es más que un pequeño detalle de *rendering* que puede implementarse en una única línea, sea $M_{modelView}$ la matriz de visión de la cámara (no se aplica la matriz del modelo en el *rendering*), $vPosition$ la posición del vértice en el mundo, y R_k un factor para enfatizar la refracción (por defecto toma valor uno):

$$refractive_{factor} = \left(-\left(M_{modelView} * \widehat{(vPosition, 1)}\right)_{xyz} * \left(M_{modelView} * (0,1,0,0)\right)_{xyz}\right)^{R_k} \quad (2.74)$$

Nótese como $-\left(M_{modelView} * (vPosition, 1)\right)$ se traduce en un *fragment shader* como $-position$, habiéndose calculado previamente $M_{modelView} * (vPosition, 1)$ en el *vertex shader*. Recordemos que $position$ representa la posición que ocupa un vértice respecto del sistema que define la cámara. Por tanto, $-position$ es el vector que iría desde el vértice hasta la posición de la cámara, y así, podemos compararlo con el vector up, igualmente definido respecto del sistema de la cámara. Una comparativa de dos ángulos de visión diferentes se muestra en la Ilustración 133.

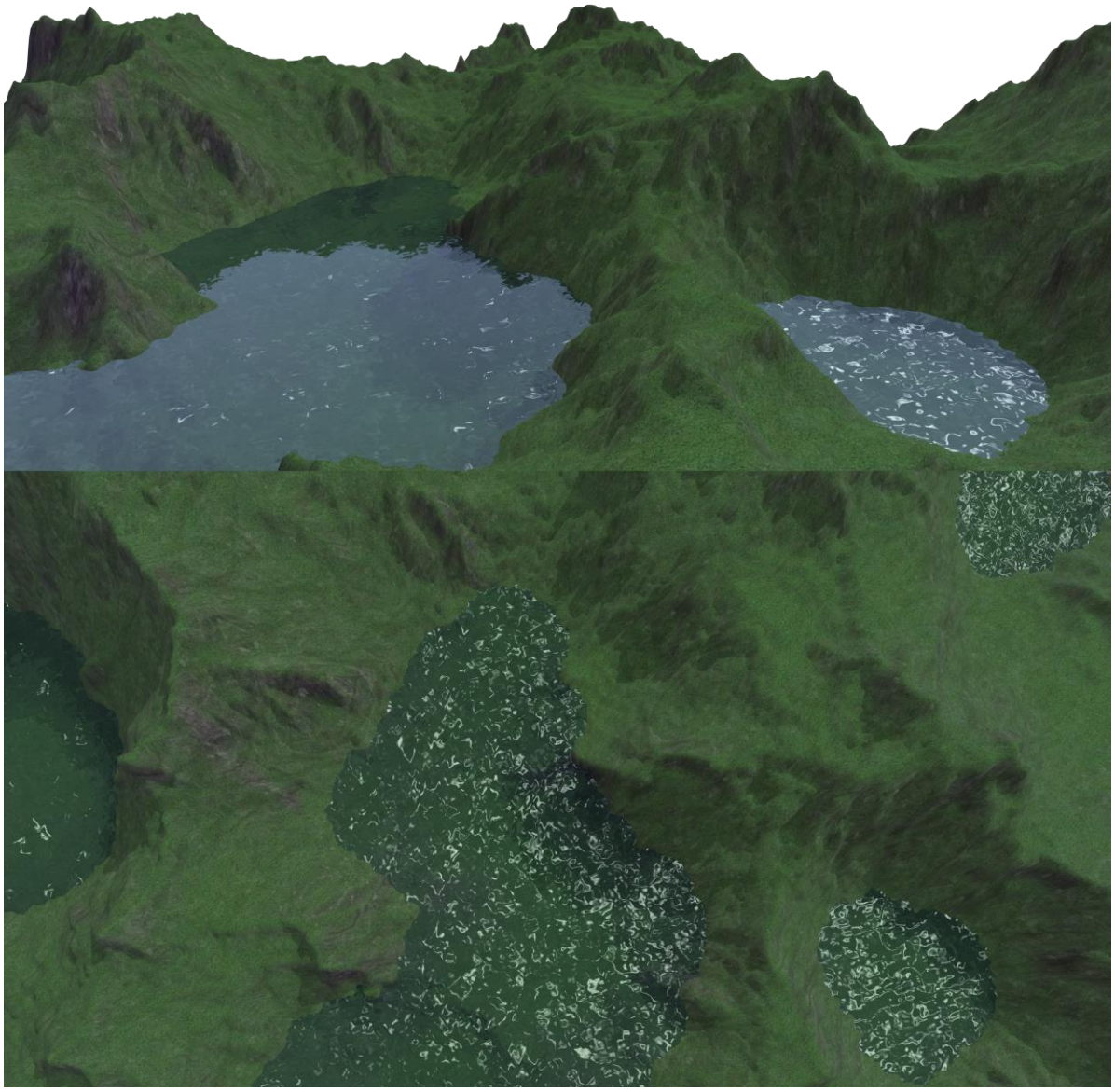


Ilustración 133. Comparativa de rendering de agua empleando diferentes ángulos de visión.

2.6.3 Generación procedural de vegetación

La generación procedural de vegetación involucra la instanciación múltiple de modelos de vegetación, que en este caso no representa más que una capa de césped sobre el terreno. De nuevo, la vegetación es una superficie de gran interés en la simulación LiDAR. Generar una capa de vegetación involucra un elevado número de instanciaciones, y por esta misma razón, emplear un modelo de geometría muy compleja no es práctico, no ya sólo para *rendering*, sino también para el proceso de simulación descrito, dado que necesitaríamos un *buffer* de geometría y topología de un tamaño muy elevado, y evidentemente, existen ciertos límite de almacenamiento en GPU.

Por esta misma razón, es necesario elaborar una o varias primitivas, de complejidad reducida, que permitan simular esta vegetación, e incluso así, habrá que evaluar cuántas instancias somos capaces de añadir a la escena. Los modelos que se plantean son los que se muestran en la Ilustración 134:

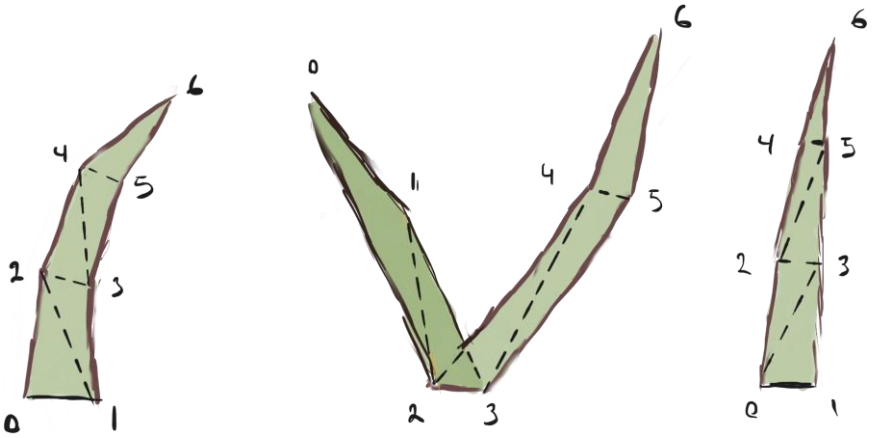


Ilustración 134. Modelos de vegetación planteados en la aplicación.

Estos modelos no son estáticos sobre el terreno, sino que además de la posición, podrán variar su escala y rotación. Antes de continuar con la instanciación, es necesario considerar la generación de un mapa de vegetación, es decir, un mapa de probabilidad para la instanciación de vegetación. Se debe tener en cuenta que, a medida que decrece la altura y el terreno se acerca a la superficie de agua, la probabilidad decrece, y esto sucede igualmente en puntos con demasiada pendiente. Además, es posible elaborar dicho mapa en GPU mediante un *dispatch* 2D. Es cierto que el mapa construido no posee un tamaño excesivamente elevado (se trata del mismo tamaño del mapa de altura), y por tanto, no sería necesario hacerlo de esta manera. Sin embargo, dicho mapa será útil en etapas posteriores que sí necesitan de computación paralela, por lo que podemos definir el *buffer* en este punto y aprovecharlo en etapas posteriores.

A la hora de calcular el mapa, se consideran hasta 6 parámetros:

- Mapa de altura y de normales del terreno. Tanto la altura como la pendiente del terreno son relevantes en esta etapa.
- Mínima y máxima altura, de tal manera que la función de probabilidad se puede describir en términos de una función de interpolación algo más suave, no necesariamente lineal. En nuestro caso, dicha función se corresponde

con una función de Hermite (*smoothstep* en GLSL). Por debajo del mínimo, la probabilidad es cero; por encima del máximo, la probabilidad es del 100%.

- Peso de la altura y la pendiente en la función de probabilidad, de tal manera que lo más común es que la pendiente tome mayor peso; se prefiere eliminar vegetación en función de la pendiente que en función de la altura.

A continuación se exponen todos aquellos valores que deben ser calculados para hallar la función de probabilidad final:

$$
\begin{aligned}
&textCoord = \frac{pos}{map_{size}} \\
&height = texture(heightSampler, textCoord)_x \\
&normal = texture(normalSampler, textCoord)_{xyz} \\
&height_{prob} = \big(1 - hermite(height_{min}, height_{max}, height)\big) * weight_{height} \\
&slope_{prob} = (1 - dot\big(normal, (0,0,1)\big) * weight_{slope} \\
&probability = 1 - height_{prob} - slope_{prob}
\end{aligned}
\tag{2.75}
$$

La pendiente en un punto no se determina mediante el vector $(0,1,0)$, sino a través de los vectores $(1,0,0)$ y $(0,0,1)$, aunque es más sencillo realizar el producto escalar con $(0,1,0)$ y obtener $1-\propto$ con el resultado obtenido. En un mapa de normales, $(0,1,0)$ se presenta como $(0,0,1)$, razón por la cual aparece este vector en las fórmulas definidas. De esta manera, las paredes completamente verticales alcanzarían $slope_{prob} = weight_{slope}$. Además de $height_{prob}$ y $slope_{prob}$, es posible incluir un valor adicional en $probability$, que podría proceder de una textura de ruido. Dadas las restricciones del número de modelos instanciables, se ha preferido no reducir aún más la función de probabilidad, pero sería factible crear pequeñas áreas de terreno con mayor cantidad de vegetación (por ejemplo, mediante una textura de ruido de Perlin).

En la Ilustración 135 se muestran dos ejemplos de mapas de probabilidad obtenidos a partir de diferentes semillas y configuraciones de erosión. Parece evidente que el resultado depende en gran medida de la generación del mapa de altura. En el ejemplo superior se establece una configuración que pretende crear un mapa mucho más suavizado, lo que produce unas fronteras muy bien definidas en el mapa de probabilidad, mientras que el segundo ejemplo se acerca más a cualquiera de los

terrenos antes ilustrados, dado que se centra en la generación de cambios mucho más visibles (se suele hacer uso del término *ravines* para hacer referencia a esos "hilos" pronunciados que se forman en el mapa de normales). Aunque también se generan fronteras en este segundo mapa de probabilidad, no están tan bien definidas y son mucho más bruscas. Esto sugiere que será necesario ajustar la generación del mapa de probabilidad en función de la configuración de la erosión.

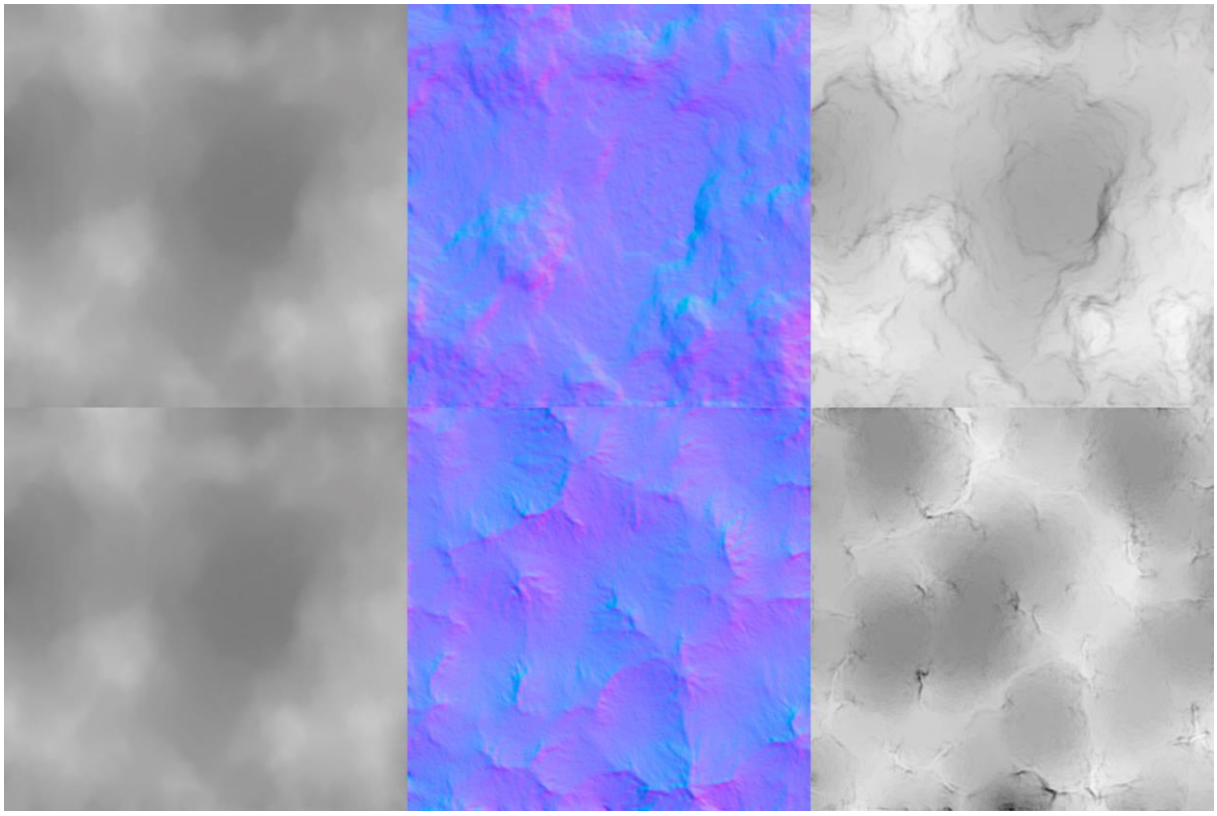


Ilustración 135. Mapas de probabilidad (derecha) junto a su correspondiente mapa de altura (izquierda) y mapa de normales (centro).

Cabe destacar que todo el comportamiento descrito hasta ahora en torno a la vegetación se desarrolla en una entidad ***Vegetation***, dependiente del terreno para su instanciación, dado que debe elaborar el mapa de vegetación a partir de la información que aporta el terreno.

Por otro lado, los modelos de la Ilustración 134 se pueden representar mediante algunos vectores estáticos que podrían definirse en un *compute shader*. No sólo se debe definir la geometría (7 vértices en todos los casos), sino también la topología (3 vectores que contienen 5 valores de tipo `uvec3`). Todos los modelos describen su forma mediante un mismo número de vértices y triángulos con el fin de simplificar el problema, aunque bien podría incluirse un contador que pudiera actualizarse

atómicamente. Igualmente, la descripción realizada de los modelos se ofrece a generar una versión más reducida de los mismos, un nivel de detalle diferente, que pudiera ser relevante si quisiéramos reducir la cantidad de memoria empleada o la geometría que debería *renderizarse*.

En la Ilustración 136 se describe, mediante pseudocódigo, cómo se instancian los diversos modelos de vegetación en un *compute shader*, teniendo en cuenta que debe existir cierta variabilidad entre las instancias. En este caso, la información de entrada será un vector de posiciones 2D con valores situados en el intervalo $[0, 1]$. El resto de *buffers* empleados almacenan información de salida, tal como datos de vértices, triángulos, índices de la topología o el número de modelos instanciados (será relevante para recuperar la información de los *buffers* tras la ejecución del *compute shader*). Nótese como las posiciones indicadas inicialmente no son más que propuestas, dado que su instanciación depende del mapa de probabilidad y del umbral que más tarde se especifique. En este trabajo, las posiciones se generan mediante una distribución uniforme en $[0, 1]$, con el único fin de repartir equitativamente la vegetación a lo largo del mapa.

El primer bloque de este algoritmo se dedica únicamente al descarte de algunas de las posiciones obtenidas, sean un umbral y el valor recuperado del mapa de probabilidad los valores que determinan si se produce un descarte. Además, se calcula la base del nuevo modelo, dada por una posición en el terreno y la normal en dicho punto. Como disponemos de tres modelos diferentes, son necesarios dos umbrales para seleccionar la geometría y topología que se utiliza como referencia en la función `generateGrass`.

Algorithm 19 $GenerateVegetation_0$: Descarte de posiciones de instanciación y selección de modelo que debe generarse

$pos_{random} \leftarrow randomData[tIndex]_{xy}$
$textCoord \leftarrow pos_{random}$
$probability \leftarrow texture(map_{vegetation}, textCoord)$

if $probability > threshold$ **then**
    $plant_{ID} \leftarrow atomicAdd(plants_{num}, 1)$
    $normal \leftarrow 2 * texture(map_{normals}, textCoord) - 1$
    $normal \leftarrow (normal_x, normal_z, normal_y)$
    $position \leftarrow (pos_{random_x} * width - width/2, 0, pos_{random_y} * depth - depth/2, 1)$
    $position \leftarrow M_{terrain} * position$
    $position \leftarrow position + (0, 1, 0, 0) * texture(map_{height}, textCoord)_r * displacement_{terrain}$

    $randType \leftarrow rand(position_{xz})$
    if $randType < threshold_0$ **then**
        $generateGrass(plant_{ID}, position_{xyz}, normal, textCoord, 0)$
    else if $randType < threshold_1$ **then**
        $generateGrass(plant_{ID}, position_{xyz}, normal, textCoord, 1)$
    else
        $generateGrass(plant_{ID}, position_{xyz}, normal, textCoord, 2)$
    end if
end if

Ilustración 136. Pseudocódigo de instanciación de modelos de vegetación en un compute shader.

La generación de un modelo se especifica en el pseudocódigo de la Ilustración 137. Una vez se calculan los índices base en cada vector para la instancia actual, es posible calcular su nueva normal, una vez se rota en torno al eje Y (con un ángulo aleatorio), u obtener su escala (también aleatoria) dentro de un intervalo previamente definido. Aún permanece en este bloque de código un vestigio de un sistema de vegetación con animación, dado que permite alterar el modelo mediante algunos parámetros básicos de viento, como la fuerza o la dirección, que en última instancia permite calcular un gradiente que indica la dirección de movimiento del modelo. Dicho gradiente deberá aplicarse sobre cada uno de los vértices predefinidos para el modelo i. Nótese como, inicialmente, todos los triángulos de cualquiera de los modelos se definen en $Z = 0$, por lo que es posible calcular una misma normal para todos los vértices y triángulos.

Se hace referencia en múltiples ocasiones a una función `rand`, que no viene dada por GLSL, sino que debe implementarse. No pretende generar un ruido con mayor continuidad que un *white noise*, por lo que se emplea una función de ruido

hallada en la web que simplemente transforma dos valores (`vec2`) en un valor pseudo-aleatorio en $[0, 1]$, a través de operaciones como un producto escalar o un seno en el que intervienen algunas constantes. No es más que una función que siempre produce un mismo resultado para el mismo valor de entrada, y en cierto modo, puede servir para nuestro propósito (sólo necesitamos incluir variaciones entre diferentes instancias).

Algorithm 20 $GenerateVegetation_1$: Definición de geometría y topología de una instancia de vegetación

$index_{basePos_0} \leftarrow plant_{ID} * NUM\_VERTICES\_GRASS$
$index_{basePos_1} \leftarrow plant_{ID} * (NUM\_VERTICES\_GRASS + 1)$
$index_{baseFace} \leftarrow plant_{ID} * NUM\_FACES\_GRASS$
$rad \leftarrow rand(position_{xy}) * 2 - 1$
$rotation_y \leftarrow mat4((0, 1, 0), rand(position_{xy}) * 2 - 1)$
$gradient \leftarrow wind_{strength} * 0.5 * wind_{direction}$
$gradient \mathrel{-}= -0.8 * (0, 1, 0) * dot((0, 1, 0), gradient)$
$instance_{normal} \leftarrow norm((rotation_y * (0, 0, 1, 0))_{xyz})$
$instance_{tangent} \leftarrow cross(-instance_{normal}, normal)$
$instance_{tangent} \leftarrow norm((rotation_y * instance_{tangent}, 0)_{xyz})$
$scale \leftarrow rand(textCoord) * (scale_{max} - scale_{min}) + scale_{min}$

for $i = 0, 1, \ldots, NUM\_VERTICES\_GRASS$ **do**
    $y \leftarrow scale * normal * GRASS_y[i]$
    $xz \leftarrow (rotation_y * (GRASS\_SCALE * GRASS_x[i] * gradient), 1)_{xyz}$
    $instance_{pos} \leftarrow position + xz + y$

    $vertex[index_{basePos_0} + i].position = instance_{pos}$
    $vertex[index_{basePos_0} + i].normal = instance_{normal}$
    $vertex[index_{basePos_0} + i].textCoord = (0, 0)$
    $vertex[index_{basePos_0} + i].tangent = instance_{tangent}$

    $rawMesh[index_{basePos_1} + i] \leftarrow index_{basePos_0} + i$
end for

$rawMesh[index_{basePos_1} + NUM\_VERTICES\_GRASS] \leftarrow index_{restart}$

for $i = 0, 1, \ldots, NUM\_FACES\_GRASS$ **do**
    $face[index_{baseFace} + i].vertices \leftarrow index_{basePos_0} + triangleIndices[i]$
    $face[index_{baseFace} + i].normal \leftarrow instance_{normal}$
    $computeAABB(index_{baseFace} + i)$
end for

Ilustración 137. Pseudocódigo de generación de geometría y topología de un modelo de vegetación.

La vegetación especifidada se puede *renderizar* fácilmente mediante un *shader program* que reciba un color difuso constante, el cual pertenece al color de la vegetación. Por tanto, no es necesario elaborar un nuevo *shader program* o llevar a cabo alguna modificación sobre la base planteada en el Anexo, dado que es suficiente con definir un material adecuado. El resultado final de todo este proceso se muestra en la Ilustración 138.

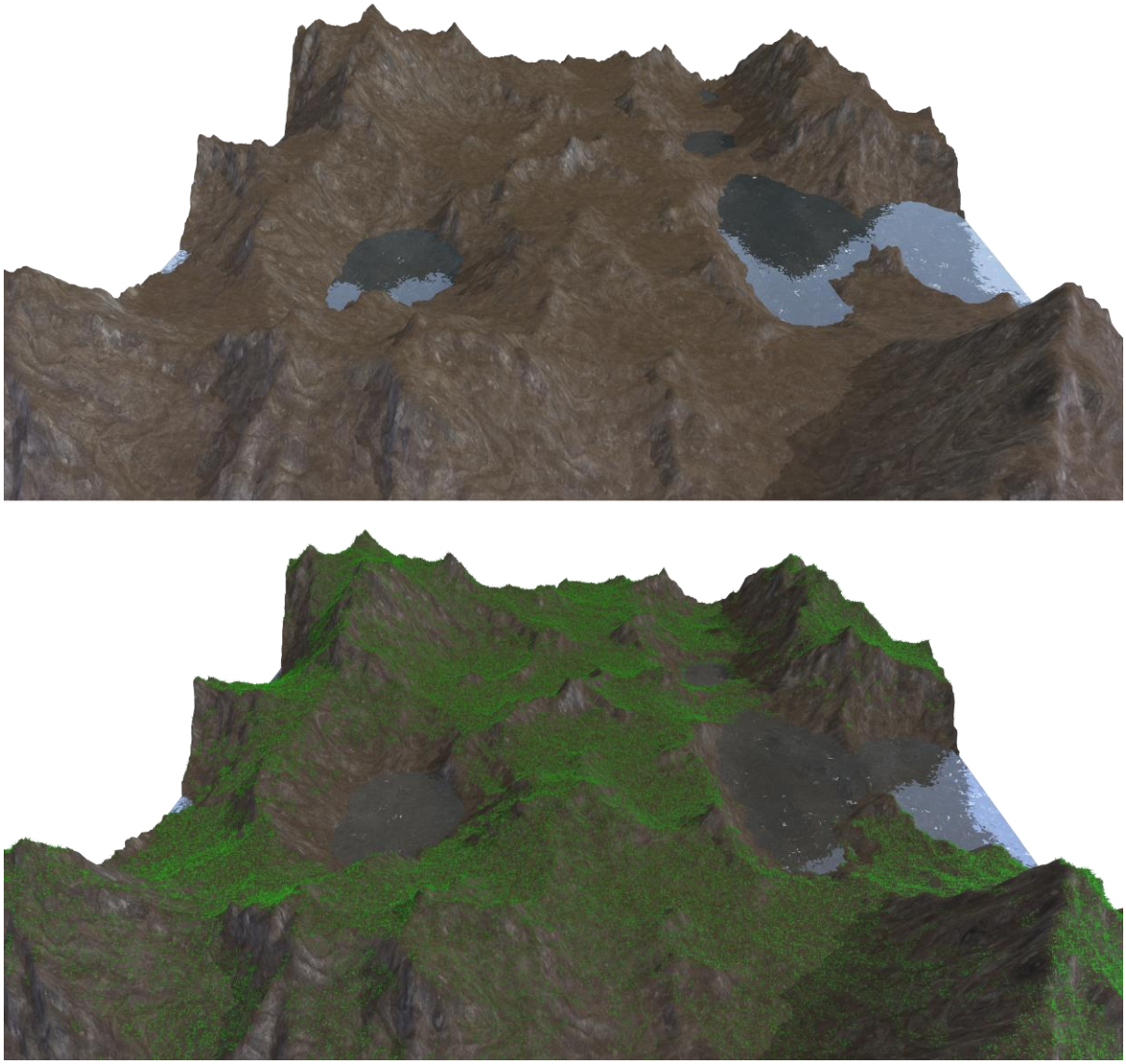


Ilustración 138. Comparativa de terreno sin y con vegetación.

Sin embargo, sería de interés elaborar un *shader program* diferente para representar una posible alternativa a la generación procedural de la vegetación, en la cual se emplea lo que se denomina como *geometry shader*. Esta alternativa no puede sustituir completamente al planteamiento descrito, sino que sólo tendría aplicación en la etapa de *rendering*. En cualquier caso, en este apartado no vamos a encontrar una descripción extensa de esta otra solución, debido a que no se pretende extender aún más esta documentación con un enfoque alternativo. Ciertamente, se trata de un enfoque muy interesante si únicamente nos centramos en el *rendering*, en tanto que permite instanciar mucha más vegetación en comparación con la capacidad hasta ahora documentada. Sin embargo, se debe descartar su aplicación dado que no disponemos de la capacidad para representar tal cantidad de vegetación mediante

buffers, lo que supondría que la simulación LiDAR debería efectuarse sobre un subconjunto, dificultando la verificación del resultado sobre la escena.

Sobre esta capa de vegetación generada en el *geometry shader* se puede añadir una función de ruido que determine el desplazamiento de la misma, lo que nos permite simular el viento. Esto es posible debido a que los modelos de vegetación se emiten de nuevo en cada *frame*, por lo que es bastante sencillo realizar una transformación acorde a una textura de ruido y el valor obtenido para un instante de tiempo t. Una de las exigencias de este enfoque es también la necesidad de definir un terreno con un número notable de subdivisiones, dado que la generación de nuevas primitivas depende a su vez de las primitivas iniciales. Por ejemplo, un plano con 100x100 subdivisiones no produciría buenos resultados si sólo se genera un modelo por triángulo del terreno. Como solución, podríamos subdividir más (es el enfoque más directo y el que aquí se emplea), o generar más de una primitiva en cada triángulo (dada la ecuación del triángulo, es posible generar más de una posición aleatoria dentro del mismo).

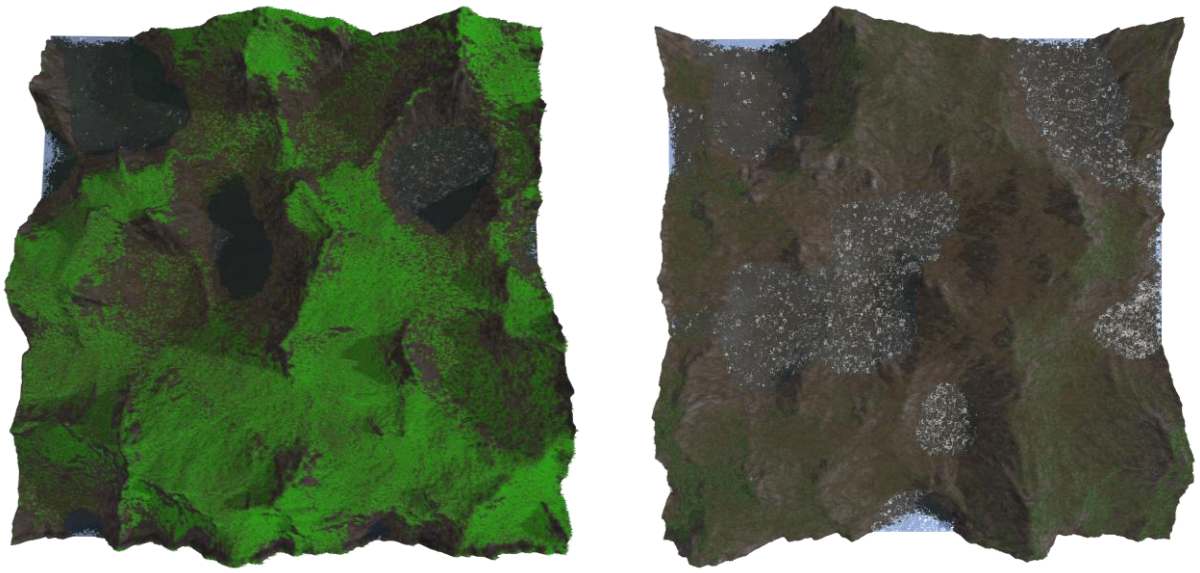


Ilustración 139. Comparativa de un enfoque alternativo de generación de vegetación y la versión final.

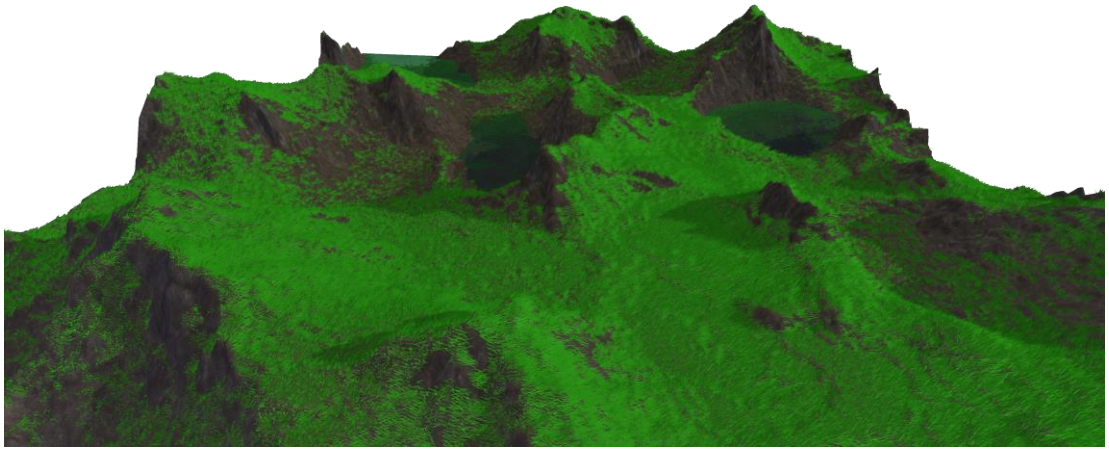


Ilustración 140. Resultado de vegetación generada en el geometry shader.

Para finalizar con este apartado, se muestra el tiempo medio de respuesta de la generación de vegetación en GPU (Tabla 44) así como la generación del mapa de probabilidad (Tabla 43). El tiempo de respuesta se comprueba para un número variable de posiciones, teniendo en cuenta que este número no representa el número final de modelos. Además, cada tiempo se ha medido con una semilla diferente para el terreno, por lo que podemos encontrar ciertas ejecuciones donde el terreno permita añadir más o menos vegetación. En el caso de la construcción del mapa de probabilidad sólo se comprueba un tamaño, que será el valor dado en el proceso de erosión (320). En todos los tiempos obtenidos se incluye la definición de *buffers* en GPU, la generación de posiciones aleatorias en CPU o la recuperación de *buffers*, siendo esta última operación la que mayor tiempo representa dentro del total.

Tiempo medio de generación de un mapa de probabilidad		
Tamaño de mapa: 320 x 320		
Ejecución	**Tiempo de respuesta (ms)**	**Tiempo de kernel (ms)**
1	14,670	0,0266
2	14,327	0,0256
3	13,178	0,0266
4	13,397	0,0256
5	14,425	0,0266
	Tiempo medio (ms) 13,999	**Tiempo medio (ms)** 0,026

Tabla 43. Tiempo medio de respuesta de la generación del mapa de probabilidad para el tamaño propuesto en el proceso de erosión.

Tiempo medio de generación de vegetación				
250.000 posiciones candidatas				
Ejecución	**Tiempo de respuesta (ms)**		**Tiempo de kernel (ms)**	
1	43,354		1,9015	
2	43,071		2,2384	
3	45,324		1,6302	
4	43,471		2,2476	
5	46,936		1,4028	
	Tiempo medio (ms)	44,431	**Tiempo medio (ms)**	1,884
800.000 posiciones candidatas				
Ejecución	**Tiempo de respuesta (ms)**		**Tiempo de kernel (ms)**	
1	147,507		4,8865	
2	137,246		9,2815	
3	141,333		7,81	
4	160,306		4,2752	
5	126,331		4,8261	
	Tiempo medio (ms)	142,54	**Tiempo medio (ms)**	6,215
1.200.000 posiciones candidatas				
Ejecución	**Tiempo de respuesta (ms)**		**Tiempo de kernel (ms)**	
1	220,260		8,7602	
2	212,954		10,6762	
3	251,317		12,0637	
4	196,628		9,2139	
5	197,203		9,7781	
	Tiempo medio (ms)	215,672	**Tiempo medio (ms)**	10,098

Tabla 44. Tiempo medio de respuesta de la instanciación de modelos de vegetación.

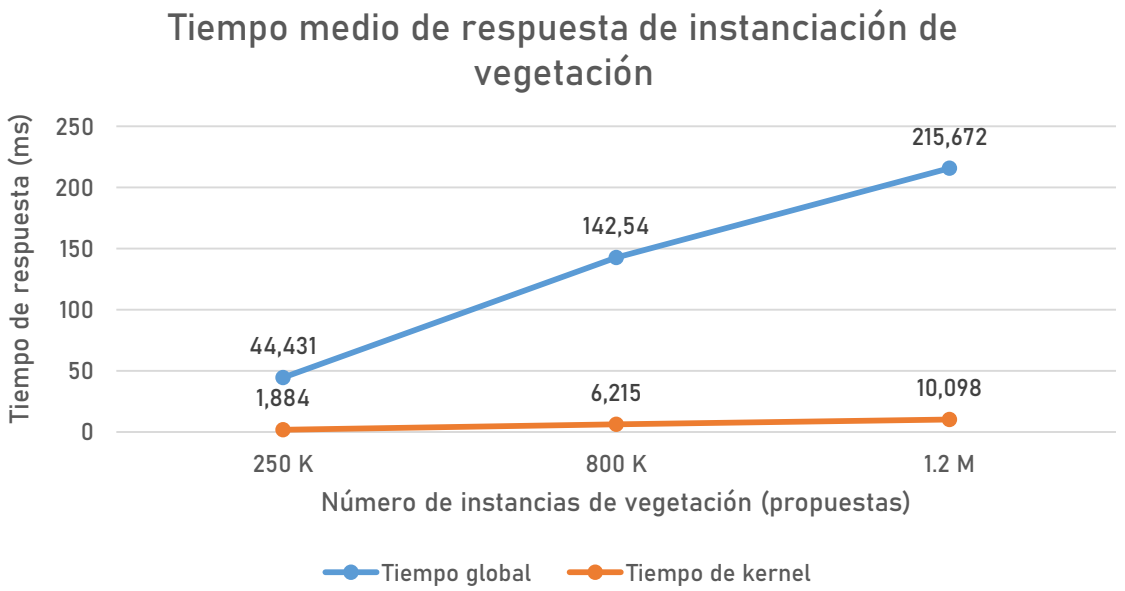


Ilustración 141. Representación mediante un diagrama de líneas del tiempo medio obtenido en la instanciación de los modelos de vegetación.

2.6.4 Generación procedural de bosque

La completitud del entorno propuesto se alcanza con la introducción de un bosque, es decir, módelos de árboles instanciados proceduralmente. Si bien el problema es similar a la generación de vegetación, existen diferencias muy evidentes:

- El modelo o modelos empleados no se generan a bajo nivel, es decir, se parte de un modelo externo y se replica cuantas veces deseemos.
- La complejidad de los modelos utilizados es necesariamente mayor, lo cual plantea restricciones en número de instancias (se muestran en la Ilustración 142).
- Los modelos introducidos, a diferencia de la vegetación, poseen mayores dimensiones. Por tanto, una colisión entre ellos es mucho más visible que cualquier colisión que pudiera producirse entre los modelos de vegetación, los cuales eran notablemente más pequeños.
- El *rendering* de todos los modelos instanciados puede suponer un gran problema, dado que pueden añadir a la escena millones de triángulos y vértices. Por tanto, habrá que separar en este caso *rendering* y definición de *buffers* de árboles, en tanto que no es posible obtener una escena fluida sin hacer uso de *Instance Rendering*.

Ilustración 142. Modelos de árboles empleados en la generación del bosque.

La entidad encargada de controlar esta generación procedural se denomina ***Forest***, y el proceso que sigue se describe a continuación:

1. Carga de modelos de árboles. Serán tres los modelos considerados (Ilustración 142), que a su vez se subdividen en tronco y hojas. En un *rendering* nos es indiferente la subdivisión, pero no sucede así en el escaneo LiDAR, dado que cada modelo presenta un comportamiento diferente. Todos ellos comparten los mismos materiales, por lo que se simplifica en gran medida la inicialización, y a pesar de esto, se generaliza para poder incluir cualquier otro tipo de árbol. Disponemos de una carpeta raíz, un vector de nombres de árboles (que a su vez redirigen a carpetas donde encontraremos dos modelos, *Trunk* y *Canopy*), y dos vectores de materiales, uno para cada parte del árbol (aunque los tres materiales de cada vector serán los mismos, dado que sólo disponemos de uno para cada parte del árbol).

   A pesar de que los modelos seleccionados no son especialmente complejos, al menos en comparación con otras opciones, su *rendering* supone un desafío, dado que introducen una carga notable. Por tanto, podemos acudir de nuevo a lo que se denomina *Instance Rendering*; se define cada modelo una única vez, pero se representa múltiples veces. Este

concepto no es apto para nuestro enfoque de construcción de la escena, por lo que habrá que plantear la existencia de múltiples componentes en la entidad *Forest*, cada uno con una finalidad.

Además, la utilización de ***Instance Rendering*** supone también añadir información adicional al VAO donde se almacenan los árboles. Más concretamente, son necesarios tres VBOs que almacenan información de traslación, rotación y escalado. Bien podría definirse un único VBO para todas estas extensiones, pero el VBO de traslación (y sólo este) se reutilizará en la siguiente escena, razón por la cual se ha decidido definir cada tipo de dato de manera individual.

2. **Designación de posibles posiciones candidatas**. Es cierto que podrían generarse posiciones distribuidas uniformemente a lo largo del mapa, pero un problema que deseamos evitar es la colisión de árboles. Comprobar la intersección de modelos parece una operación fuera de lugar para una propuesta (y no un requisito) de generación de un entorno algo más realista. El enfoque que se plantea es la utilización de una estructura de datos que subdivida el terreno, permitiéndonos determinar si en cierta zona del espacio existe una carga superior a la deseada. Dicha estructura será una malla regular personalizada para el problema que se trata, de tal manera que cada celda no almacena una lista de elementos contenidos (no es nuestro propósito realizar búsquedas), sino sólo el número de ellos. Ciertamente, no existen búsquedas, pero sí consultas: 1) ¿la celda donde se encuentra un punto está ocupada?, 2) ¿dicha celda supera un umbral de densidad?, 3) ¿la celda donde se encuentra un punto, y todas aquellas contenidas en un radio, superan un umbral de densidad?

   De la misma manera que se permite consultar, también es posible insertar de dos maneras diferentes: introducción de un elemento en la celda donde se encuentra la posición indicada, o introducción del elemento tanto en dicha celda como en todas aquellas que se encuentran dentro de un radio. Además, se ofrece la posibilidad de añadir más de un elemento; aunque hemos hablado de que se almacena el número de elementos contenidos en una celda, se debe observar más bien como un valor de densidad, de tal manera que una celda se podrá descartar si supera cierto

umbral. Hay modelos más flexibles, como un árbol, en el que nos basta con que los troncos no colisionen, en cuyo caso no es necesario restringir su instanciación únicamente a celdas vacías (si están son suficientemente grandes), pero hay otros modelos (los veremos más tarde), que tienen restricciones más grandes. Por esto último, un bosque se instancia después de aquellos modelos menos flexibles, dado que será más fácil encontrar una posición para estos últimos si se hace en este orden. Para evitar que la zona ocupada por un modelo de mayores restricciones sea ocupada por árboles es necesario saturar un radio, y más concretamente, se debe hacer con un valor mayor que el umbral que el que estos emplean.

3. **Definición de posiciones instanciables y sus características**. Dado el vector de posiciones anteriormente obtenido, es posible descartar aquellas posiciones que no se ajustan a unos requisitos. Dichos requisitos son los mismos que se aplicaban a la vegetación: el mapa de probabilidad debe superar un umbral en la posición propuesta, lo que indica que la pendiente o la altura del punto son adecuadas para una nueva instanciación. Además, también se definen valores aleatorios para la escala o la rotación (los valores de X, Y, Z implican rotaciones en dichos ejes, sea el valor más potenciado la rotación en Y, mientras que el resto se atenúa).

   Sabemos que no existe una generación nativa de valores aleatorios en GLSL, sino que se trata de una función que solventa este problema. Para generar valores aleatorios diferentes en cada árbol se utiliza como entrada de dicha función los valores X, Z de las posiciones, de tal manera que es posible emplear $position.xy$, $position.yx$, $position_{[0,1]}.xy$, etc. Nótese como $position_{[0,1]}$ es la posición originalmente introducida, que a través de diversas transformaciones se adaptará al tamaño del terreno.

4. **Generación de la geometría y topología de cada uno de los árboles mediante los 3 SSBOs extraídos anteriormente:** posición, escala y rotación. Para cada tipo de árbol, se lanzan tantos hilos como instancias se hayan extraído mediante el paso 3, de tal manera que a partir de dos SSBOs de geometría y topología de un modelo de árbol, es posible completar otros dos SSBOs que almacenan esta misma información para todas las instancias. Por tanto, el único trabajo de un hilo en este programa es el

cálculo de una matriz de transformación que deberá aplicarse a todos y cada uno de los puntos del modelo base, mientras que en el caso de la topología lo más complejo es el cálculo del índice base necesario para referenciar los puntos de su instancia.

Respecto del último paso descrito, cabe destacar que el resultado final no consiste en 12 SSBOs diferentes (vértices y triángulos por los tres modelos disponibles y por los dos componentes de un modelo), sino únicamente dos. Se da la situación de que en el escaneo LiDAR basta con distinguir tronco y hojas, no es necesario clasificar la información resultante por tipo de árbol (al menos, si no existen diferencias entre ellos en el comportamiento que representan frente al dispositivo LiDAR). Esto implica que en el cálculo del índice base antes citado se incorporan dos *offsets*: uno a nivel global (derivado de ejecuciones previas de otros modelos de árboles), y otro *offset* a nivel local (dentro de una misma ejecución).

El comportamiento de los *shaders* no parece relevante dada su escasa complejidad, por lo que únicamente mostraremos algunos resultados de este apartado. En la Ilustración 143 se explota la técnica de *Instance Rendering*, aunque no es posible representar tal cantidad de geometría en una escena que ya de por sí es compleja. La Ilustración 144 muestra una comparativa de la vista cenital de dos escenarios con diferente densidad de árboles, siendo la segunda (menor densidad) la que representa el escenario de la solución final. La Ilustración 145 muestra el estado de una malla regular, de 180 subdivisiones en X y Z, tras obtener un vector de posiciones candidatas. No todas las posiciones donde se insertan puntos acaban siendo ocupadas, dado que depende en última instancia del mapa de probabilidad y de un umbral, razón por la cual es posible encontrar celdas ocupadas sin un árbol sobre ellas.

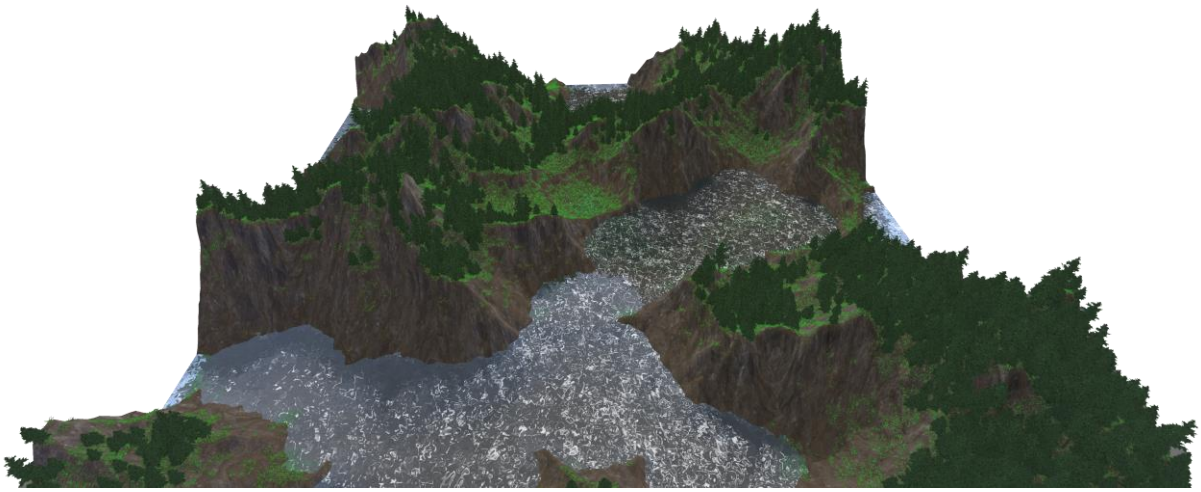


Ilustración 143. Instance rendering de árboles en la escena.

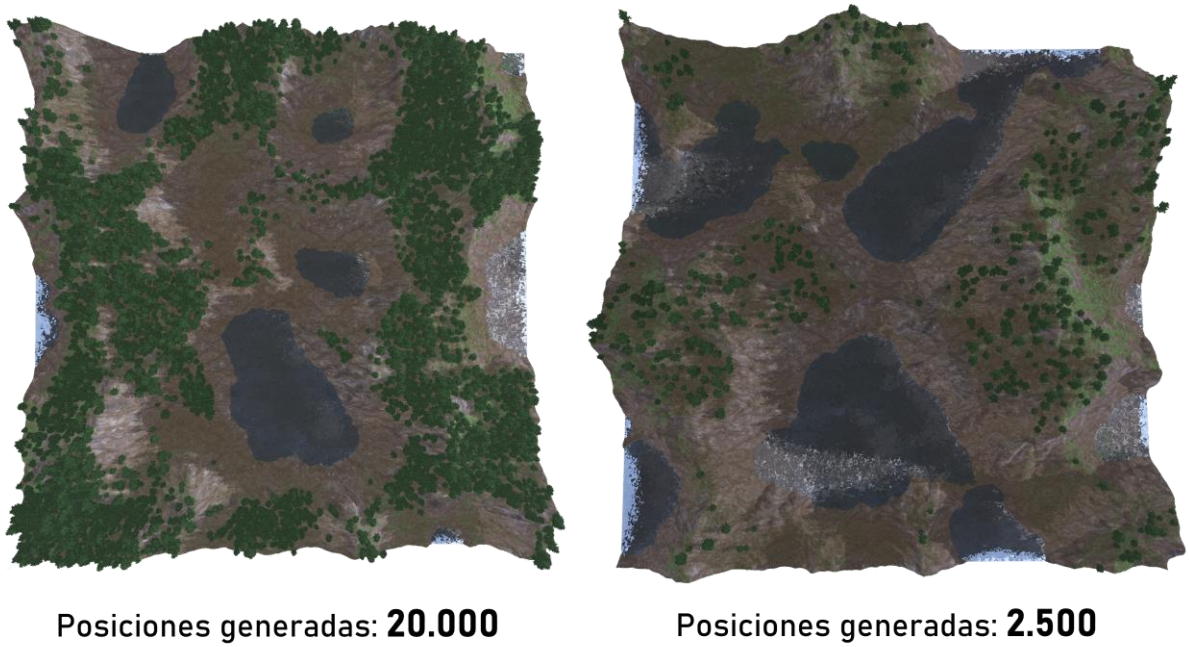


Ilustración 144. Comparativa de dos mapas con diferente densidad de árboles.

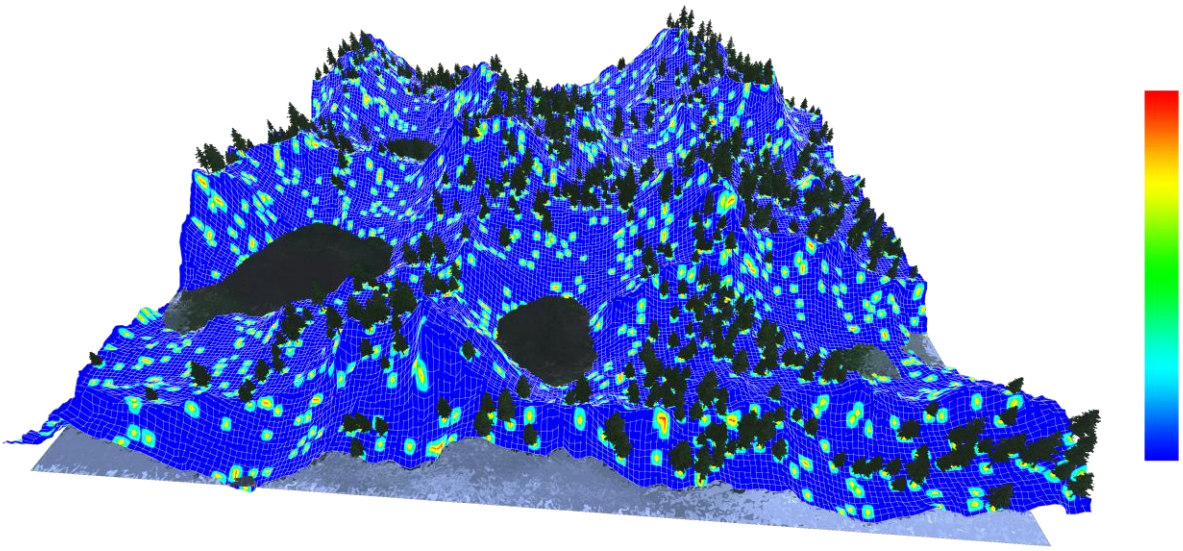


Ilustración 145. Rendering de malla regular. El color representa el número de puntos contenidos respecto del máximo identificado.

A nivel de *rendering*, el principal desafío de la representación de árboles es la aplicación de lo que se conoce como *cutoff*, un valor *alpha* que representa el umbral para descartar fragmentos. La reducida complejidad de los árboles procede de la definición de su vegetación como planos, que deben modelarse mediante una textura de transparencia. Evidentemente, esto es posible en el *rendering* dado que podemos descartar fragmentos, pero no sucede lo mismo en la definición de los *buffers* que se aplicarán en el escaneo LiDAR. Por tanto, es evidente que se podrán obtener puntos de intersección que no se encuentran exactamente sobre la superficie *renderizada*, dado que pertenecen a posiciones descartadas del plano, pero es una desventaja que se asume con la única finalidad de representar un número mayor de árboles.

Dado un valor a de opacidad, recuperado de una textura $alpha$, un fragmento se descarta si $r - cutoff$ es menor que 0. También es posible modificar el color del árbol, de tal manera que adapte su color en función de dos valores, que hemos denominado $healthy_{color}$ y dry_{color}, los cuales se combinarán mediante una simple operación de *blending* y se multiplicarán por el color original del árbol.

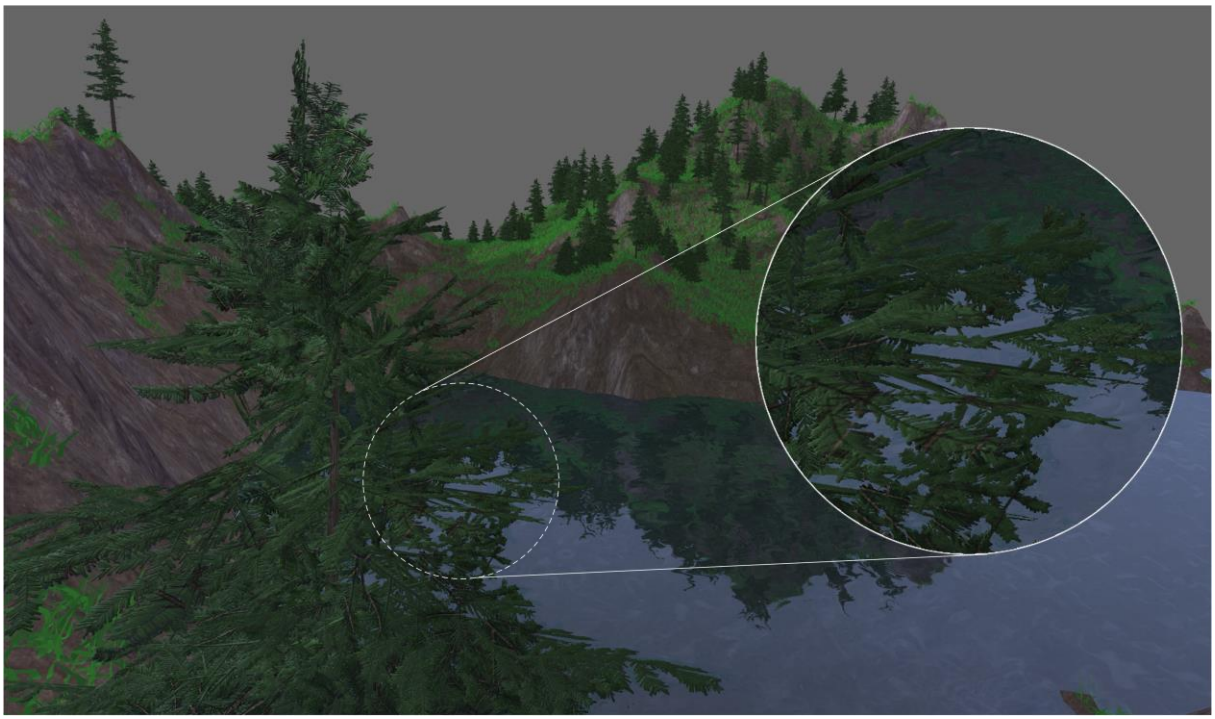


Ilustración 146. Detalle de alpha cutoff en un árbol.

Por último, se representan los tiempos obtenidos en todo el proceso de generación de geometría y topología de árboles (Tabla 45). No se hace una diferenciación entre los dos *compute shaders* dado que no es relevante el rendimiento de cada uno, sino del proceso completo, y de hecho, se incluyen todas aquellas operaciones de definición y recuperación de *buffers*. El tiempo de respuesta de *kernel* representa la suma de los tiempos obtenidos en ambos programas, teniendo en cuenta que son necesarias hasta 12 mediciones de tiempo (*dos compute shaders* para tres modelos formados por dos componentes). Buena parte del tiempo observado pertenece a operaciones con *buffers* en CPU. Igualmente, existe una gran variabilidad en los tiempos documentados, dado que estos dependen del número de modelos finalmente instanciados.

Tiempo medio de definición de buffers de árboles				
1.000 posiciones candidatas				
Ejecución	**Tiempo de respuesta (ms)**		**Tiempo de kernel (ms)**	
1	337,296		0,037	
2	320,594		0,036	
3	330,072		0,026	
4	245,993		0,032	
5	323,825		0,047	
	Tiempo medio (ms)	311,55	**Tiempo medio (ms)**	0,035
2.500 posiciones candidatas				
Ejecución	**Tiempo de respuesta (ms)**		**Tiempo de kernel (ms)**	
1	871,600		0,051	
2	885,841		0,063	
3	772,571		0,049	
4	555,975		0,057	
5	809,833		0,063	
	Tiempo medio (ms)	779,16	**Tiempo medio (ms)**	0,056
5.000 posiciones candidatas				
Ejecución	**Tiempo de respuesta (ms)**		**Tiempo de kernel (ms)**	
1	1.242,631		0,073	
2	1.109,871		0,071	
3	1.502,893		0,059	
4	1.561,751		0,072	
5	1.199,109		0,06	
	Tiempo medio (ms)	1.323,25	**Tiempo medio (ms)**	0,067

Tabla 45. Tiempo medio de respuesta obtenido en la instanciación de árboles en la escena.

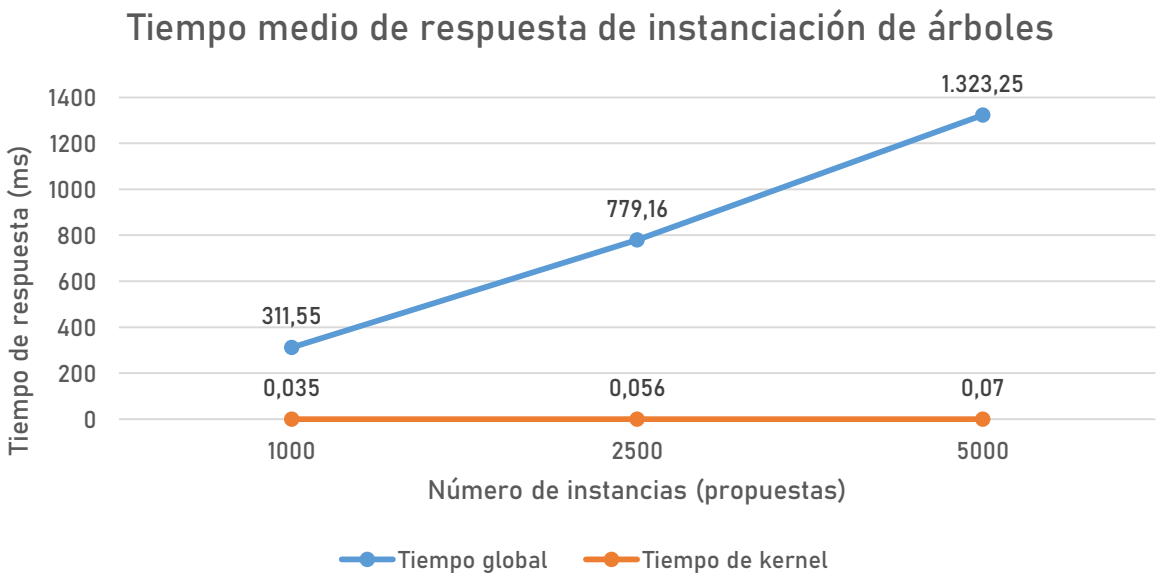


Ilustración 147. Diagrama de líneas donde se muestra el tiempo medio obtenido en la instanciación de árboles en la escena.

2.6.5 Posicionamiento procedural de otras entidades en un terreno

La última propuesta sobre el escenario del terreno es la inclusión de cualquier otro modelo que pueda utilizarse durante el escaneo para simular algunas de las construcciones que se representan en las clases del estándar ASPRS, como torres de transmisión o cualquier otra construcción. Además, desde el punto de vista de la instanciación procedural, la inclusión de modelos con mayores restricciones parece especialmente interesante. Con este fin, se desarrolla una entidad, ***TerrainModel***, encargada de gestionar la instanciación de aquellos modelos que no pertenecen a la vegetación, y que igualmente deben representarse sobre el terreno. Estos modelos serán los primeros en instanciarse, dado que además de restricciones relacionadas con la pendiente del terreno o con la altura, también existen restricciones de saturación de la malla regular, dado que no deben colisionar con un árbol en la escena.

Los modelos incluidos en esta última etapa son los que se muestran en la Ilustración 148, si bien sólo las dos construcciones se instancian mediante *TerrainModel*.

Ilustración 148. Resto de modelos empleados en la elaboración del escenario.

Los pasos que deben completarse para instanciar alguna de estas construcciones son los siguientes:

1. **Definición de posiciones candidatas**, recuperadas mediante una distribución uniforme en el intervalo $[0, 1]$. Sólo se incluyen aquellos puntos que no se encuentran ya ocupados, aunque al tratarse de los primeros modelos instanciados, la única situación en la que se podría descartar un punto es ante posibles colisiones entre los propios modelos. Nótese como dicha comprobación se debe producir dentro de un radio (cada instancia de *TerrainModel* define su propio radio), teniendo en cuenta que la saturación de la malla también se produce dentro de un radio, y no únicamente en la posición de instanciación.

2. **Cálculo de valor de bondad de cada una de las posiciones**. Se justifica el desarrollo de este cálculo en un *compute shader* debido al acceso a dos texturas, altura y normales del terreno. Dado un umbral de altura, el primer factor de bondad relevante se calcula como sigue:

$$step(threshold, height) = \begin{cases} 0, & height < threshold \\ 1, & height \geq threshold \end{cases} \quad (2.76)$$

También se buscan posiciones cuya normal se encuentre lo más cercana posible al vector $(0, 1, 0)$, por tanto, el producto escalar de dicho vector y la

normal hallada puede constituir otro factor de bondad. No sólo se muestrea la posición introducida, sino también un radio alrededor de este. A diferencia de una malla regular, las texturas introducidas no representan una estructura discretizada como tal, por lo que el radio se comprueba como si se tratara de una máscara. Se definen nueve posiciones, $(-1,1) * radio$, $(0,1) * radio$, …, $(1,-1) * radio$, que serán las que finalmente se muestreen para hallar la bondad de un punto.

3. **Obtención del punto con el valor de bondad más alto**. Podría resolverse en GPU mediante alguna de las implementaciones descritas anteriormente, aunque dado el escaso número de posiciones no es necesario (ni más eficiente) resolverlo en un *compute shader*. Dicha posición, así como un área alrededor de la misma, debe "saturarse" para evitar que cualquier modelo posterior pueda intersectar con esta instancia. La saturación se puede llevar a cabo en un radio, ya sea empleando un área circular o cuadrada. La primera no es más que una extensión de la segunda, donde sólo se insertan puntos en aquellas posiciones que verifican $dir.x * dir.x + dir.y * dir.y < radius * radius$, o lo que es lo mismo, $\sqrt{dir_x^2 + dir_y^2} < radius$. Ambos tipos de áreas se representan en la Ilustración 149 y la Ilustración 150.

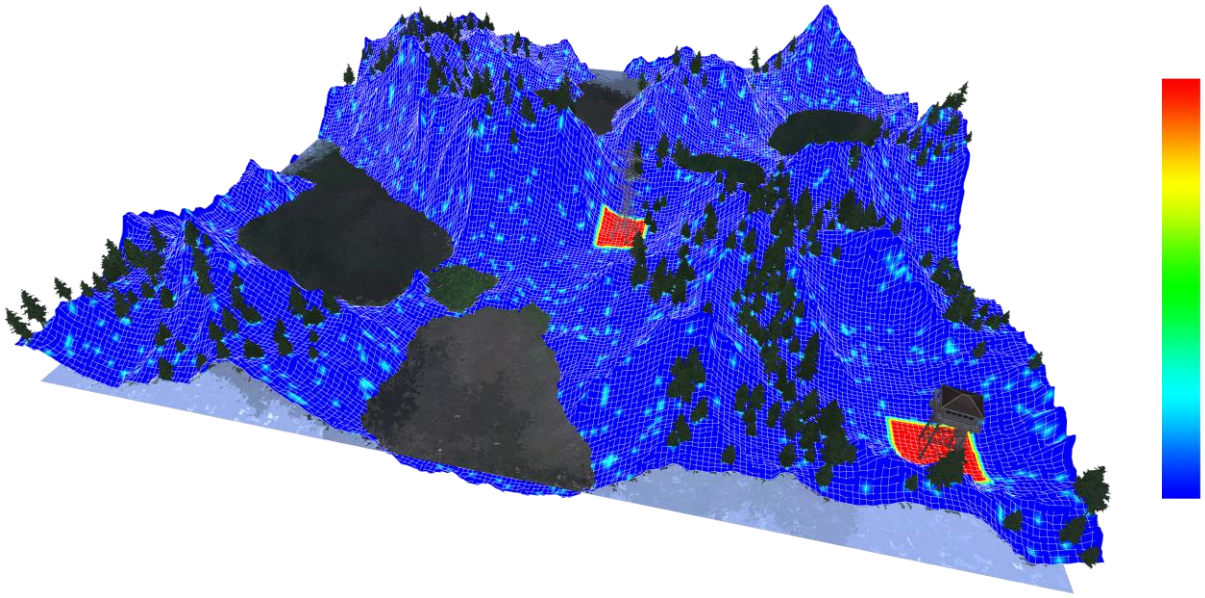


Ilustración 149. Malla regular con textura de saturación del terreno. Área cuadrada.

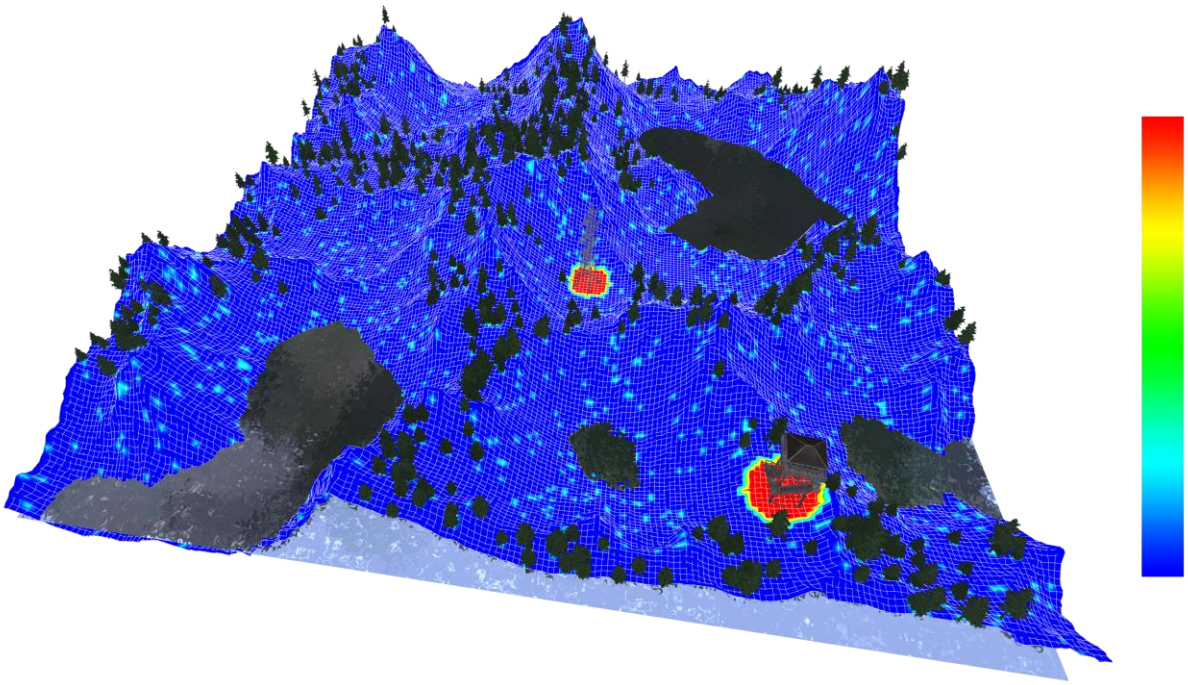


Ilustración 150. Malla regular con textura de saturación del terreno. Área circular.

En esta etapa también se incluyen otros modelos, controlados por la entidad *Vegetation*, cuya instanciación se somete a las mismas restricciones de la vegetación, con la única diferencia de que estos modelos no se definen a bajo nivel, sino que se cargan desde un fichero. No parece relevante describir más en detalle este tipo de instanciación, por lo que simplemente se muestra su resultado (junto a modelos previos) en la Ilustración 151.

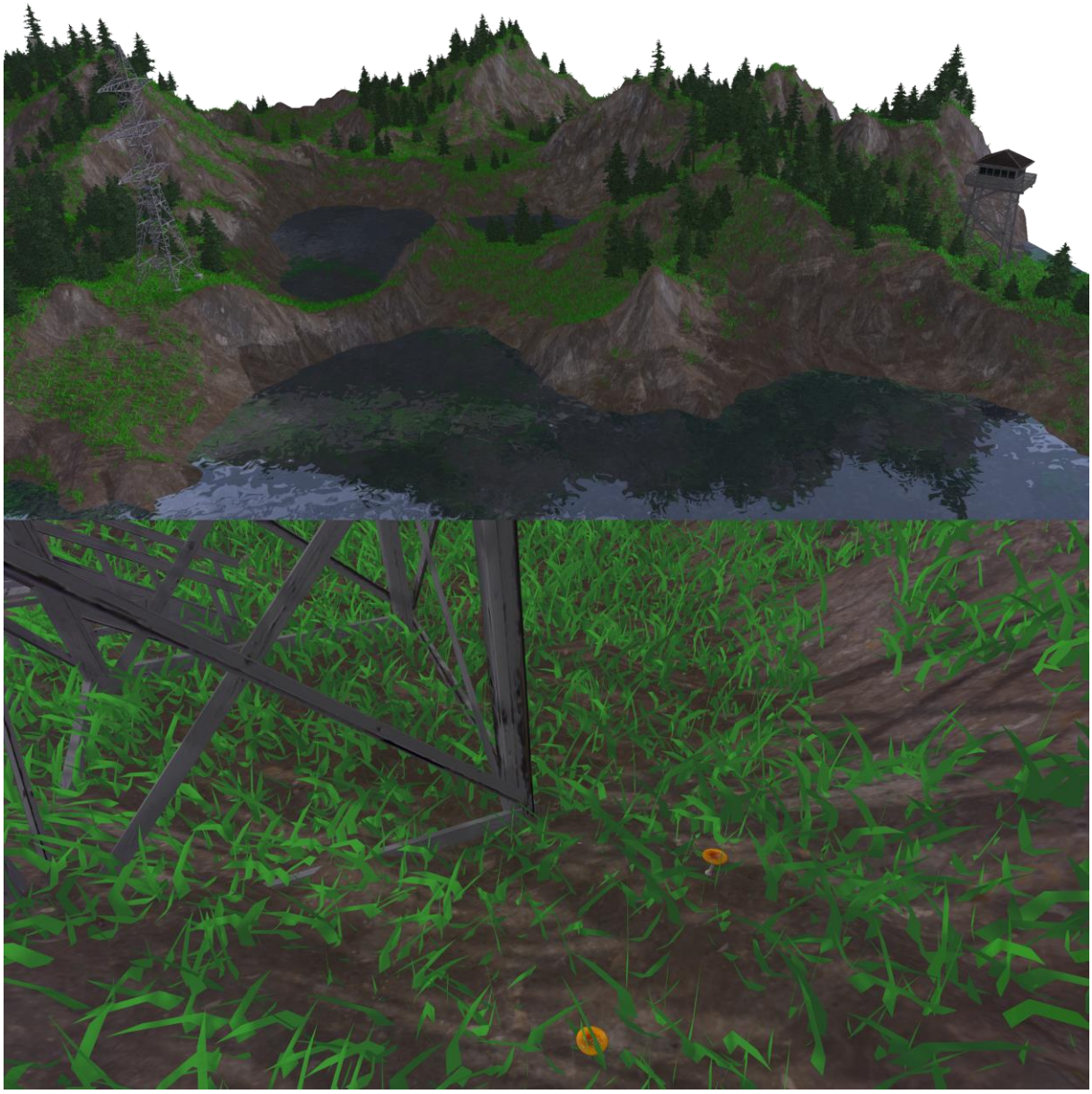


Ilustración 151. Rendering de una torre de transmisión (TerrainModel), una torre de vigilancia (TerrainModel) y setas (Vegetation).

2.6.6 Tercer escenario: Material Point Method

El tercer escenario desarrollado en la aplicación se trata de una animación de un sistema de partículas. Más concretamente, dicho sistema se basa en ***Material Point Method***, una de las técnicas más conocidas hoy en día para la simulación de un gran número de materiales, desde nieve (Stomakhin et al. 2013) hasta cualquier otro tipo de fluido (Ram et al. 2015). Se trata de un método situado entre un enfoque Lagrangiano y Euleriano; se acerca más al primero, pero emplea una malla fija (*Eulerian grid*) que nos permite deshacernos de la malla deformable del enfoque Lagrangiano, la cual se complica excesivamente con materiales altamente

deformables. Por tanto, parece evidente que este método presenta algunas ventajas respecto de los dos enfoques sobre los que se sustenta, aunque en este trabajo podemos observar principalmente la facilidad con la que se establecen límites en el escenario de la animación (lo podríamos representar como una colisión con un muro), la capacidad de detectar colisiones automáticamente mediante la información de un nodo de la malla, o la introducción de varios materiales (que incluso podrían derivar en un proceso multifase simplemente modificando las propiedades de cada material en cierto instante). Al tratarse de simples partículas, también es fácil obtener un comportamiento de *split-merge* dentro del material representado.

La introducción de este escenario en la aplicación se justifica desde la observación del rendimiento del sistema en un escenario con animaciones, donde la estructura de datos que representa la escena debe actualizarse tras cada derivación del movimiento. Además, el tamaño de las partículas nos permite valorar también la precisión del escaneo.

En este trabajo será suficiente con obtener un sistema de partículas que simule un fluido de movimiento restringido dentro de unas fronteras previamente definidas. Debido al nivel de simplicidad buscado, el desarrollo aquí realizado se centra en una de las implementaciones más básicas de MPM con integración de tiempo, basada en el método semi-implícito de Euler. De aquí en adelante, se debe considerar que el artículo de referencia es (Jiang et al. 2016), y que este método se describe en el apartado 10 de dicho artículo.

Este método es, posiblemente, uno de los mejores ejemplos de aplicación de la computación paralela, no sólo por los principios del algoritmo, donde disponemos de un gran número de partículas, que a priori podemos considerar independientes, sino también por la manera con la que se trabaja con *buffers* en GPU. A diferencia de algunas de las aplicaciones anteriores, donde veíamos que la mera definición de un *buffer* suponía cierta demora, en este caso se definen los *buffers* una única vez (al crear la instancia de MPM), y sólo se recuperan las posiciones de las partículas para renderizar cada *frame*.

Para describir el algoritmo, y su implementación, se enumerarán sus pasos, apoyándonos en imágenes que han sido elaboradas tomando como referencia el artículo propuesto. Todas las fases que se describen se implementan sobre un *compute shader*. Antes de comenzar, es necesario conocer que cada partícula tendrá unos atributos básicos, como posición, x_p, masa, m_p, volumen, V_p, velocidad, v_p,

matriz afín, B_p, una matriz de impulso, M_p, así como cualquier otro parámetro del material, aunque estos pueden almacenarse en la entidad MPM, en lugar de la partícula. Por otro lado, una celda de la malla también almacenará una velocidad y una masa asociada a las partículas contenidas. Ambas entidades se estructuran bajo *Particle* y *Cell*, respectivamente, que serán representadas en CPU y GPU.

1. Esta etapa no se define dentro del algoritmo MPM, sino que pertenece a la implementación, dado que tiene como objetivo precalcular datos de etapas posteriores. En este caso, se reinicia el volumen a 0, y se calculan los pesos que permitirán a la partícula transferir sus propiedades a la malla. Más concretamente, dada la posición x_p, es posible calcular $uint(x_p)$, así como $cell_{diff} = x_p - uint(x_p)$, de tal manera que la matriz de pesos se define como sigue:

$$\begin{pmatrix} \left(0.5 - cell_{diff_x}\right)^2 * 0.5 & \left(0.5 - cell_{diff_y}\right)^2 * 0.5 & \left(0.5 - cell_{diff_z}\right)^2 * 0.5 \\ 0.75 - cell_{diff_x}^2 & 0.75 - cell_{diff_y}^2 & 0.75 - cell_{diff_z}^2 \\ \left(0.5 + cell_{diff_x}\right)^2 * 0.5 & \left(0.5 + cell_{diff_y}\right)^2 * 0.5 & \left(0.5 + cell_{diff_z}\right)^2 * 0.5 \end{pmatrix} \quad (2.77)$$

   Por cuestiones de almacenamiento antes citadas, se representa esta matriz como una matriz 3x4, donde el último valor de un `vec4` será el *padding*.

2. Se restauran los valores por defecto de la malla; $v_{x,y}$ y $m_{x,y}$ pasan a ser 0. De nuevo, se trata de una etapa más relacionada con la implementación que con la técnica de MPM.

3. Transferencia de la partícula a la malla, y más concretamente, se transfiere masa y velocidad. Un problema recurrente en este trabajo es el acceso concurrente a una misma variable, la cual deben actualizar las partículas. De nuevo, es necesario una multiplicación de los términos de actualización con un factor K. Habrá que deshacer dicha operación cuando se proceda a su consulta. Además, en esta etapa no se trabaja aún con la malla, si no con un vector de `vec4`, cuyo tamaño vendrá dado por $grid^3_{resolution}$, sea este el tamaño considerado de la malla. Los tres primeros valores del `vec4` almacenan la velocidad, y el valor w la masa.

El método *Affine-Particle-In-Cell* (APIC) (Jiang et al. 2015) define esta transferencia a través los siguientes términos:

$$m_i = \sum_p w_{pi} m_p$$
$$momentum_i = \sum_p w_{pi} m_p (v_p + M_p(x_i - x_p)) \quad (2.78)$$

Donde $w_{pi} = W_p[grid_x]_x + W_p[grid_y]_y + W_p[grid_z]_z$, sea W_p la matriz de pesos anteriormente calculada, teniendo en cuenta que $grid_x, grid_y, grid_z \in [0,2]$. De esta manera, una partícula no sólo influye sobre su posición actual, sino también sobre su entorno. Esto también implica que son necesarias hasta 9 ejecuciones de un mismo *compute shader* (vecindario 3x3). $x_i - x_p$ se puede resolver fácilmente como:

$$cell_{dist} = \left(uint\left(x_{p_x} + grid_x - 1\right) - x_{p_x},\ uint\left(x_{p_y} + grid_y - 1\right) - x_{p_y}, uint\left(x_{p_z} + grid_z - 1\right) - x_{p_z}\right) + 0.5 \quad (2.79)$$

4. Actualización del volumen de las partículas, de acuerdo con las etapas que se muestran en la Ilustración 152. Dicho volumen depende del vecindario de la partícula, por lo que de nuevo, se considera un vecindario de tamaño 3x3 y serán necesarias hasta 9 ejecuciones. El volumen se define como sigue:

$$V_p = \frac{m_p}{\sum_i \frac{m_i}{\Delta x^d} W_i(x_p)} = \frac{m_p \Delta x^d}{\sum_i m_i W_i(x_p)} = \frac{m_p}{\sum_i m_i * w_{pi}} \quad (2.80)$$

La última formulación expone que $\frac{W_i(x_p)}{\Delta x^d} = w_{pi}$, dado que esta es la nomenclatura previamente empleada para hacer referencia al peso de p en una celda vecina i (si establecemos una representación lineal, en otro caso podríamos hablar de (x, y)). Parece evidente que en un *compute shader* no es posible resolver V_p por completo, sino que habrá que esperar a que todas las ejecuciones finalicen. Podría incluirse una estructura condicional que calcule V_p cuando $grid_x = grid_y = grid_z = 2$, pero esto no es necesario, debido a que el siguiente *compute shader* se ejecutará cuando se haya completado el cálculo de $\sum_i m_i * w_{pi}$, instante en el cual podemos obtener

V_p. Además, el siguiente *compute shader* está igualmente relacionado con este proceso, por lo que desde el punto de vista conceptual no se está realizando esta operación en una etapa diferente.

5. Aún seguimos en el proceso de transferencia de la partícula a la malla. En este paso, las celdas de la malla se verán nuevamente afectadas por las partículas, debido al impulso de las mismas, pero para ello es necesario conocer cómo se comporta el material que describen dichas partículas. En nuestro trabajo, deseamos que el comportamiento sea similar al de un fluido, mientras que (Jiang et al. 2016) describe principalmente las ecuaciones que gobiernan un material hiperelástico (*Neo-Hookean solid*). En este apartado, (Niall, 2019) nos puede guiar hacia las ecuaciones correctas. En primer lugar, es posible calcular la presión mediante la ecuación de estado de Murnaghan-Tait:

$$P = \frac{K_0}{n}\left(\left(\frac{density}{density_0}\right)^n - 1\right) + P_0 = \frac{K_0}{n}\left(\left(\frac{\sum_i m_i * w_{pi}}{density_0}\right)^n - 1\right) + P_0 \tag{2.81}$$

K_0, P_0, $density_0$ y n son constantes, mientras que la densidad es conocida del apartado previo. A partir de la presión es posible determinar la tensión a través de las ecuaciones que describen un fluido newtoniano:

$$stress = -P * I + \mu(\nabla v + \nabla v^T) \tag{2.82}$$

Donde I es la matriz identidad, y μ es una constante de viscosidad. El gradiente de velocidad vendrá determinado por la matriz de impulso M_p, de tal manera que es posible resolver $stress$ como sigue:

$$stress = \begin{pmatrix} -P & 0 & 0 \\ 0 & -P & 0 \\ 0 & 0 & -P \end{pmatrix} + \mu \begin{pmatrix} M_{p_{00}} + M_{p_{00}} & M_{p_{01}} + M_{p_{10}} & M_{p_{02}} + M_{p_{20}} \\ M_{p_{10}} + M_{p_{01}} & M_{p_{11}} + M_{p_{11}} & M_{p_{12}} + M_{p_{21}} \\ M_{p_{20}} + M_{p_{02}} & M_{p_{21}} + M_{p_{12}} & M_{p_{22}} + M_{p_{22}} \end{pmatrix} \tag{2.83}$$

De (Hu et al. 2018) se obtiene que el impulso final que debe añadirse a la celda equivale a $-4V_p * stress * dt * w_{pi} * cell_{dist}$, sea dt el incremento de tiempo en cada *frame*. De nuevo, se debe manipular una celda mediante operaciones atómicas. En este punto se sigue trabajando con el vector de `vec4`, en lugar de las celdas.

6. Actualización de velocidad de la malla. Una vez finalizado el proceso denominado *Particle to grid*, es posible calcular la velocidad de cada celda a partir del vector que hemos venido utilizando, `cellVelocityMass`. No sólo debe calcularse la velocidad a partir del impulso y la masa, sino que también debe considerarse la gravedad en esta actualización.

$$v_i = \frac{momentum_i}{m_i} + dt * (0, g, 0) = \frac{cellVelocityMass_{xyz}}{cellVelocityMass.w} + dt * (0, g, 0) \tag{2.84}$$

Sólo será posible calcular v_i cuando $m_i \neq 0$, en cuyo caso ninguna partícula se encuentra en la celda (o en el entorno), y por tanto, no es relevante que no se actualicen sus valores. También habrá que considerar las fronteras de la malla para evitar que el vector velocidad progrese hacia el exterior del volumen previamente definido. No es necesario comprobar explícitamente si el vector velocidad en una celda fronteriza contiene una determinada dirección, sino que es suficiente con actualizar la velocidad a 0 (al colisionar con las fronteras se anula la velocidad por completo).

7. *Grid to particle*. Esta transferencia de datos se produce en la dirección contraria a todo el proceso *Particle to grid*. En la práctica, este proceso se implementa mediante tres *compute shaders*, donde el primero de ellos sólo se encarga de establecer el valor cero en la velocidad y la matriz B de todas las partículas. Este paso es necesario debido a que la etapa G2P se repetirá de nuevo para todo un vecindario, mientras que el valor cero de la velocidad se debe asignar antes de comenzar.

El núcleo de esta etapa consiste en el cálculo de la nueva velocidad de la partícula, así como de su nueva matriz afín B, para lo cual es necesario que nos desplacemos de nuevo a las ecuaciones que define el método APIC:

$$\begin{aligned} v_p &= \sum_p w_{pi} v_i \\ B_p &= \sum_i v_i \left(\frac{\partial W_i}{\partial x}(x_p)\right)^T = \sum_i v_i w_{pi} * cell_{dist} \end{aligned} \tag{2.85}$$

La matriz B no representa ningún término importante en este algoritmo, sino que no es más que una variable que almacena temporalmente un valor necesario para calcular M_p en el siguiente paso, donde también se actualizará, finalmente, la posición x_p de la partícula.

$$M_p = 4 * B_p$$
$$x_p = clamp(x_p + v_p * \ dt, 1, gridRes_{xyz} - 2) \tag{2.86}$$

Donde 4 es una constante que procede de la utilización de interpolaciones cuadráticas como las definidas en w_{pi}. Igualmente, es posible modificar la velocidad de la partícula cuando esta se encuentra en zonas fronterizas del volumen definido en la escena. Nótese como no se aplica a la actualización realizada en esta iteración, sino que se comprueba si en la siguiente iteración podría salir de dichos límites. Por ejemplo:

$$x_{p_{t+1}} = x_p + v_p$$
$$v_{p\{x,y,z\}} = \begin{cases} v_{p\{x,y,z\}} + wall_{min} - x_{p_{t+1}}, & x_{p_{t+1_{\{x,y,z\}}}} < wall_{min} \\ v_{p\{x,y,z\}} + wall_{max} - x_{p_{t+1}}, & x_{p_{t+1_{\{x,y,z\}}}} > wall_{max} \\ v_{p\{x,y,z\}}, & x_{p_{t+1_{\{x,y,z\}}}} \geq wall_{min} \wedge x_{p_{t+1_{\{x,y,z\}}}} \leq wall_{max} \end{cases} \tag{2.87}$$

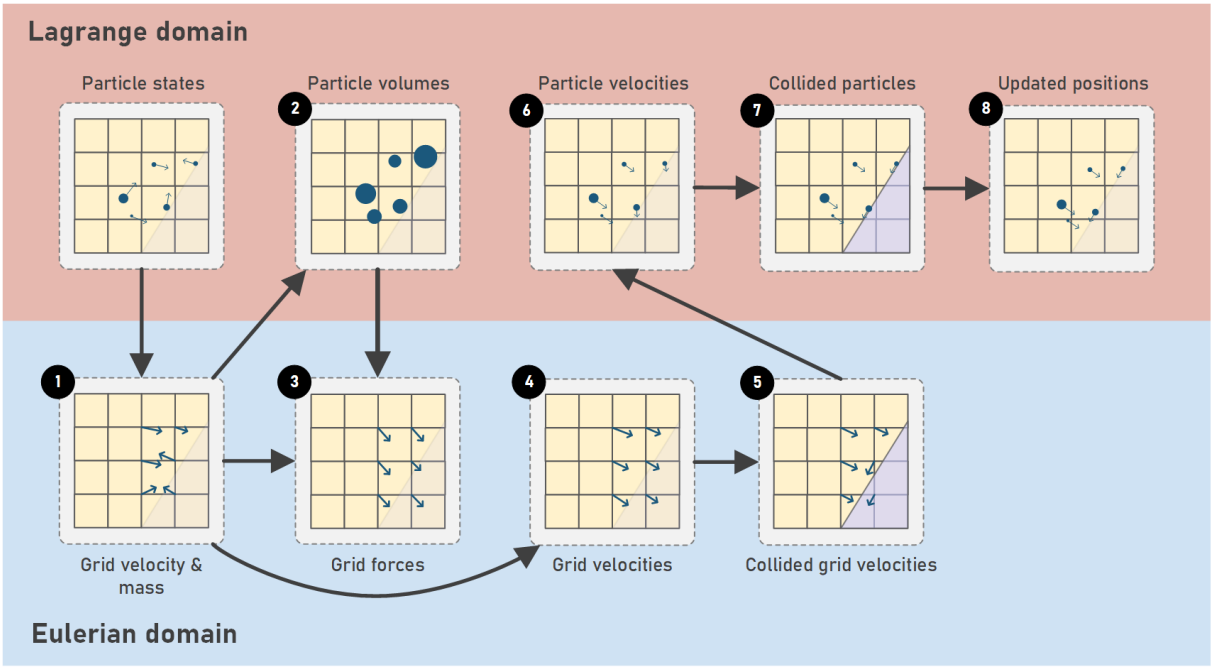


Ilustración 152. Diagrama de pasos que conforman una iteración de MPM. La numeración del diagrama no se corresponde con la numeración mostrada en la descripción del problema.

Partiendo de un elevado número de partículas, y siguiendo el proceso descrito, se puede obtener una implementación muy básica de MPM para un fluido. En el instance inicial, las partículas se pueden ordenar de cualquier modo; por ejemplo, podríamos organizarlas en un cubo, una esfera, etc, o incluso podrían formarse mediante los vértices de algún modelo. La diferencia entre las primeras propuestas y esta última es que las primeras pueden seleccionar, dentro de la aplicación, el número de partículas que deben instanciarse en el sistema. En el caso del modelo, sería necesario recurrir a una herramienta externa, con el fin de subdividir o simplificar la malla (Ilustración 154). Una alternativa a esto último es la instanciación de puntos dentro la malla del modelo, lo cual implica la construcción de una estructura de datos para acelerar dicha consulta. Por ejemplo, podríamos emplear un *octree* donde los nodos se asociaran con un color en función de su posición respecto de la malla (fuera, dentro, o frontera, y en este último caso habría que realizar alguna comprobación adicional) (Ilustración 153). En cualquier caso, este planteamiento es igualmente lento, y además, no se identifica la instanciación puntos a partir de un modelo como una necesidad de la aplicación, razón por la cual se descarta añadir esta opción en la solución final.

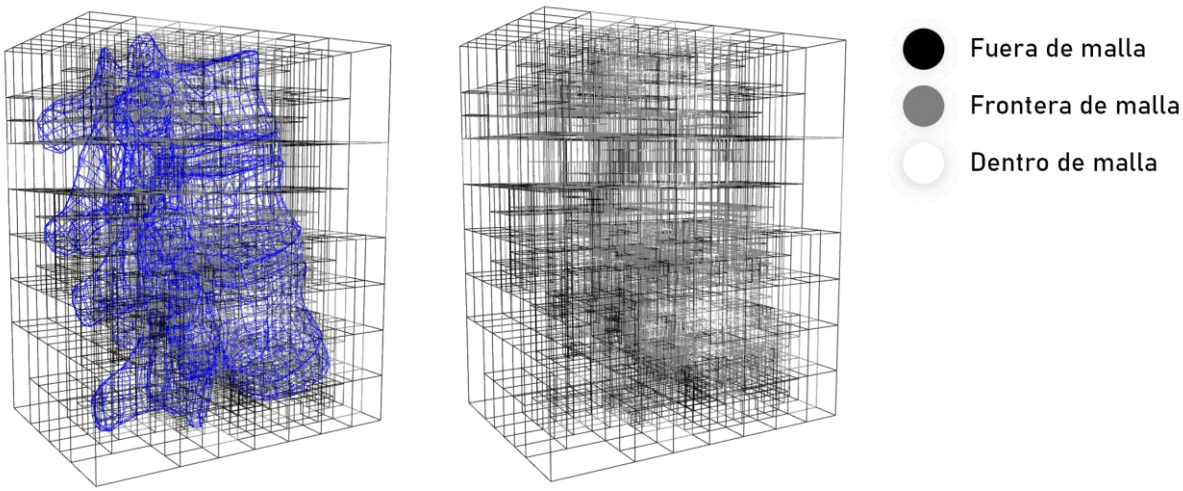


Ilustración 153. Distinción de nodos de un octree en función de su posición respecto de la malla.

También cabe destacar el importante papel de la posición inicial de las partículas en el esquema que se plantea, debido a que la estructura del sistema es la que origina los movimientos iniciales en ausencia de más información. De esta manera, un sistema denso y compacto como un cubo comienza el proceso mediante un movimiento de caída. Esto no sucede para una esfera o cualquier otro modelo,

donde el movimiento se asemeja a una desintegración, dado que el gradiente obtenido se acerca más a la normal del modelo (Ilustración 155).

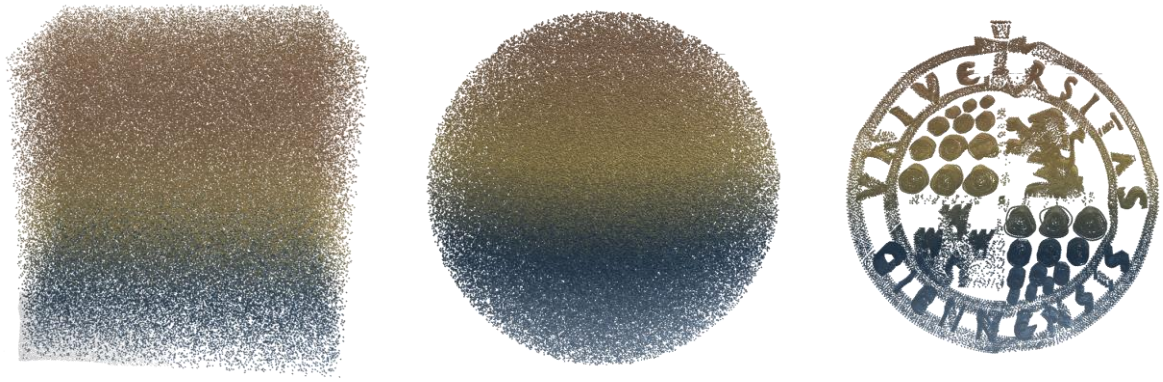


Ilustración 154. Tres posibles escenarios en la inicialización del sistema de partículas.

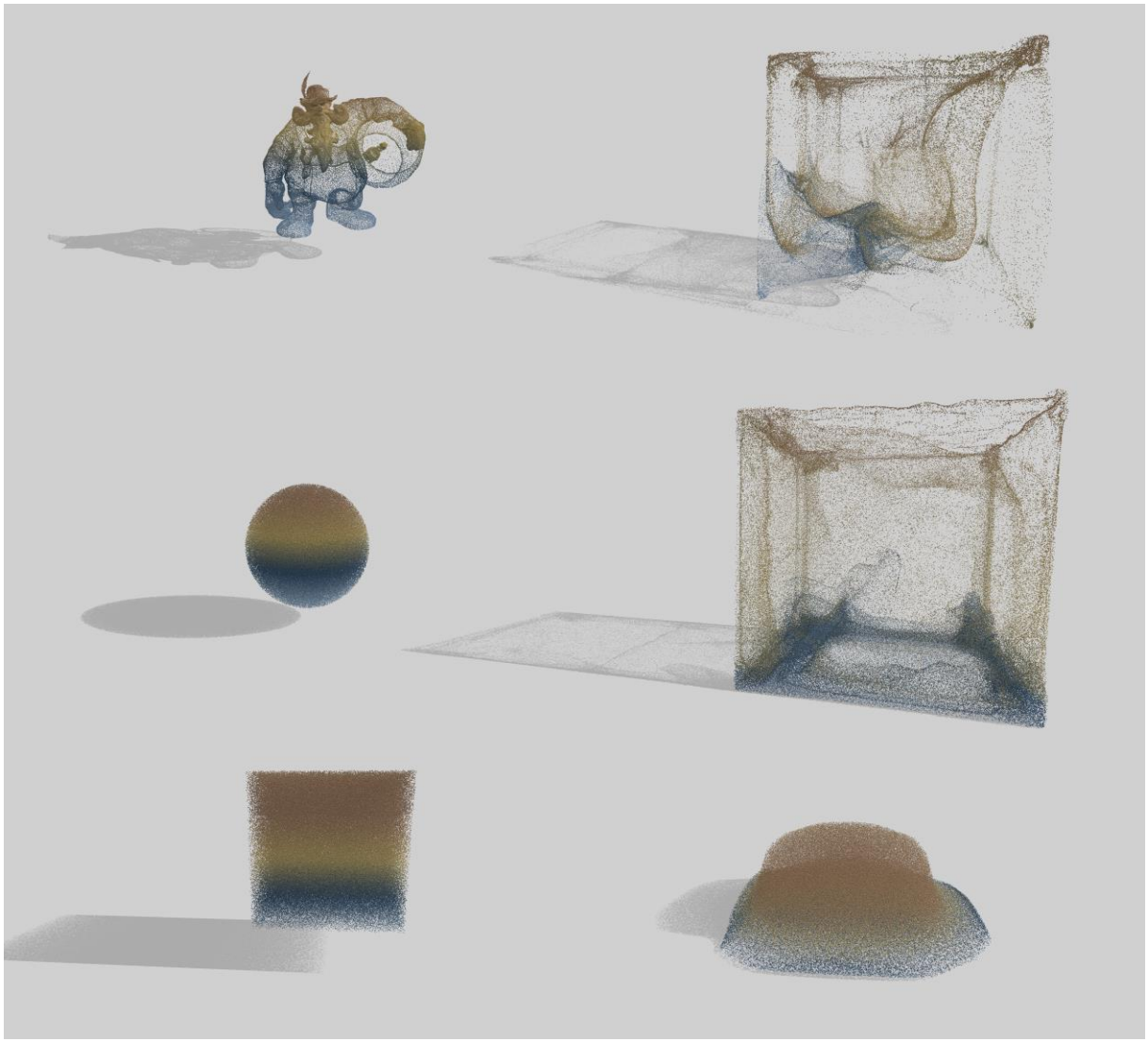


Ilustración 155. Movimiento inicial de tres modelos diferentes.

Algunos ejemplos del sistema MPM en funcionamiento se encuentra en la Ilustración 156 y la Ilustración 157:

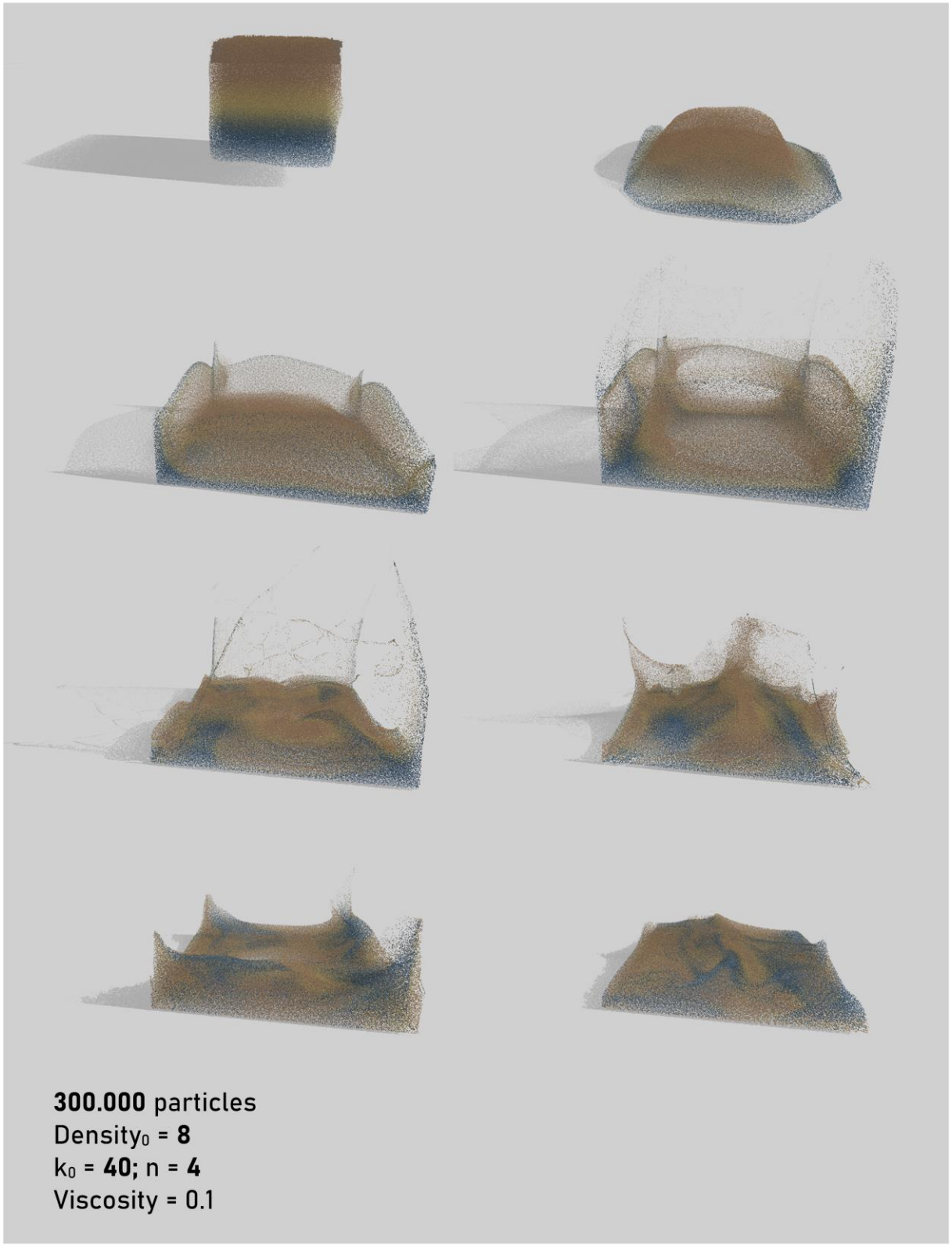


Ilustración 156. Sistema de partículas MPM en ocho instantes diferentes.

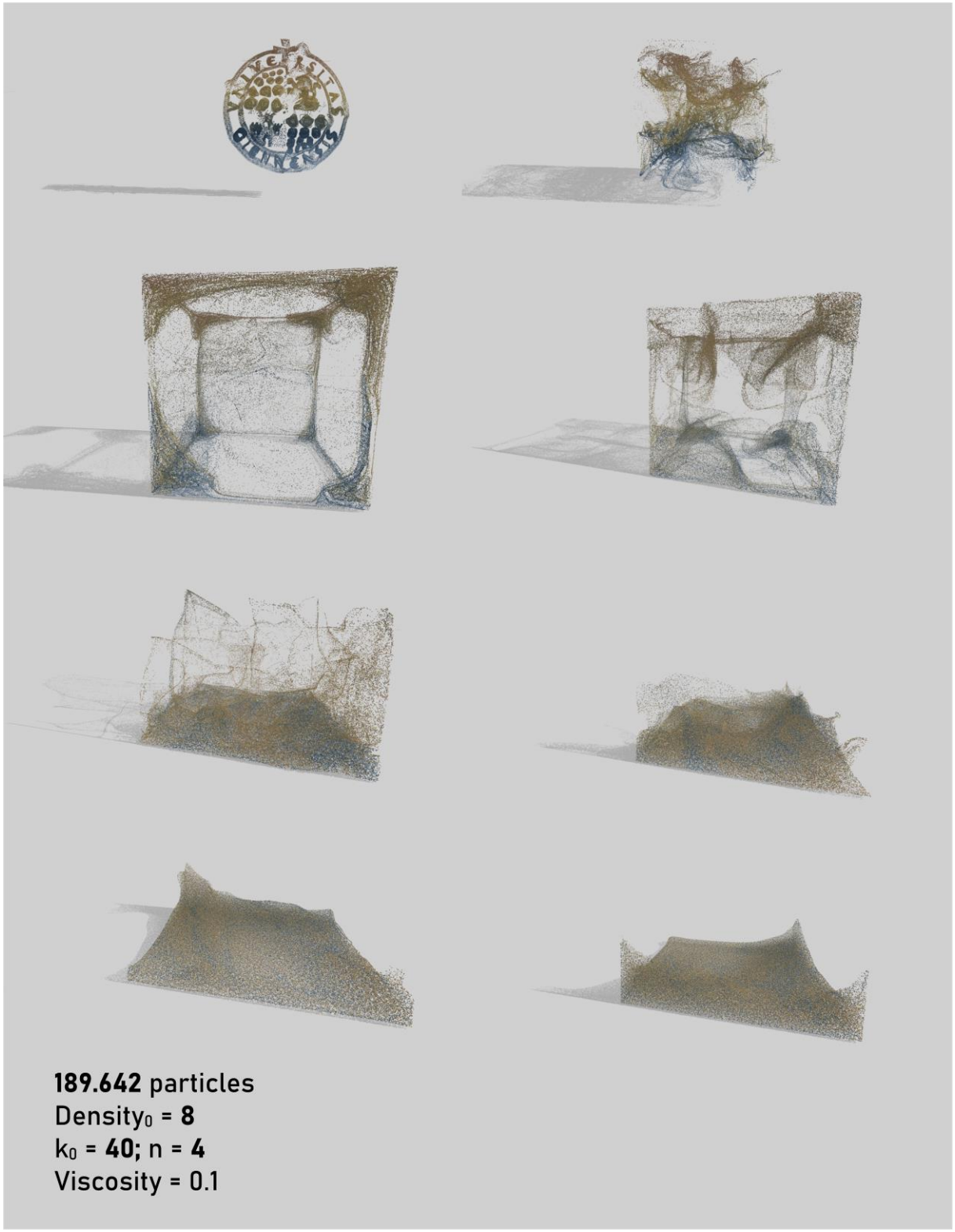


Ilustración 157. Sistema de partículas MPM en ocho instantes diferentes.

Por último, para finalizar este apartado dedicado al tercer escenario, se muestran los tiempos obtenidos en la ejecución de una animación MPM que forma sus partículas inicialmente como un cubo. El número de partículas será diferente en cada prueba. Nos interesa especialmente el tiempo medio dedicado a un *frame*, pero

también el tiempo total que es necesario para generar un determinado número de *frames*, el cual se muestra en el título de la prueba (Tabla 46).

Tiempo medio empleado en una iteración de un sistema MPM				
250.000 partículas (200 *frames*)				
Ejecución	**Tiempo de respuestal total (ms)**		**Tiempo de respuesta por frame (ms)**	
1	14.728		73,64	
2	14.470		72,35	
3	14.577		72,88	
4	14.603		73,01	
5	14.641		73,2	
	Tiempo medio (ms)	14.603,8	**Tiempo medio (ms)**	73,016
500.000 partículas (200 *frames*)				
Ejecución	**Tiempo de respuestal total (ms)**		**Tiempo de respuesta por frame (ms)**	
1	29.097		145,48	
2	29.061		145,3	
3	29.056		145,28	
4	29.075		145,37	
5	29.127		145,63	
	Tiempo medio (ms)	29.083,2	**Tiempo medio (ms)**	145,41
1.000.000 partículas (200 *frames*)				
Ejecución	**Tiempo de respuestal total (ms)**		**Tiempo de respuesta por frame (ms)**	
1	58.144		290,72	
2	58.798		293,99	
3	58.860		294,3	
4	58.646		293,23	
5	58.460		292,3	

Tiempo medio (ms)	58.581,6	**Tiempo medio (ms)**	292,9

Tabla 46. Tiempo medio de cálculo de un frame, y tiempo total necesario para la obtención de 200 frames.

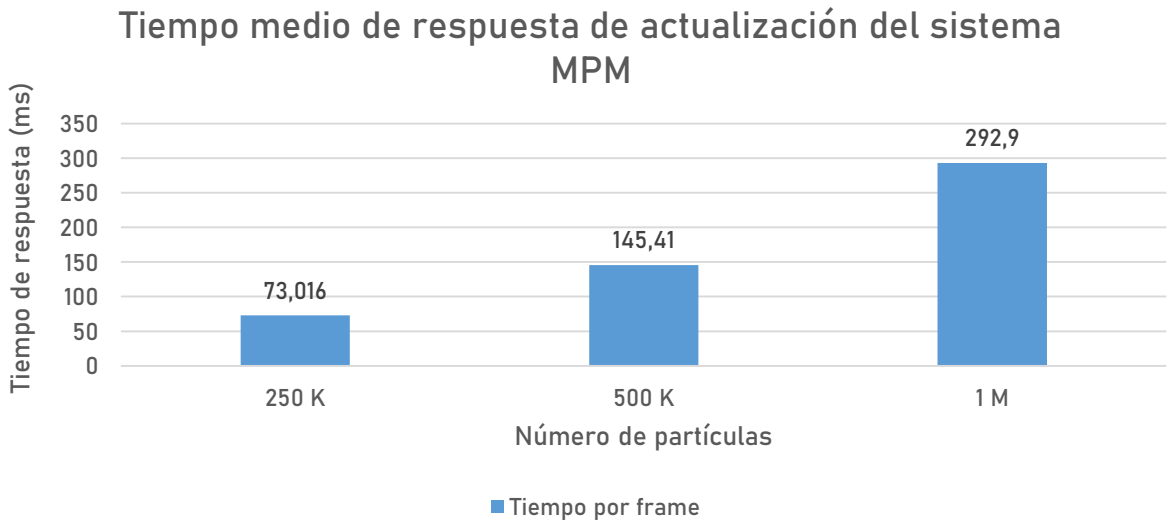


Ilustración 158. Diagrama de barras donde se representa el tiempo medio necesario para actualizar el sistema MPM en cada nuevo frame.

2.6.7 Diagrama de clases

Finaliza la descripción de esta iteración mediante un diagrama de clases que ilustra las principales entidades que han surgido en esta nueva iteración, o que hayan podido modificarse (Ilustración 159). Se incluyen en este apartado entidades independientes entre ellas, en lugar de presentarlas durante la descripción, con el objetivo de reducir la extensión de esta quinta iteración. En algunos puntos del diagrama se duplican clases vinculadas a diagramas previos, con el fin de aclarar la estructura expuesta.

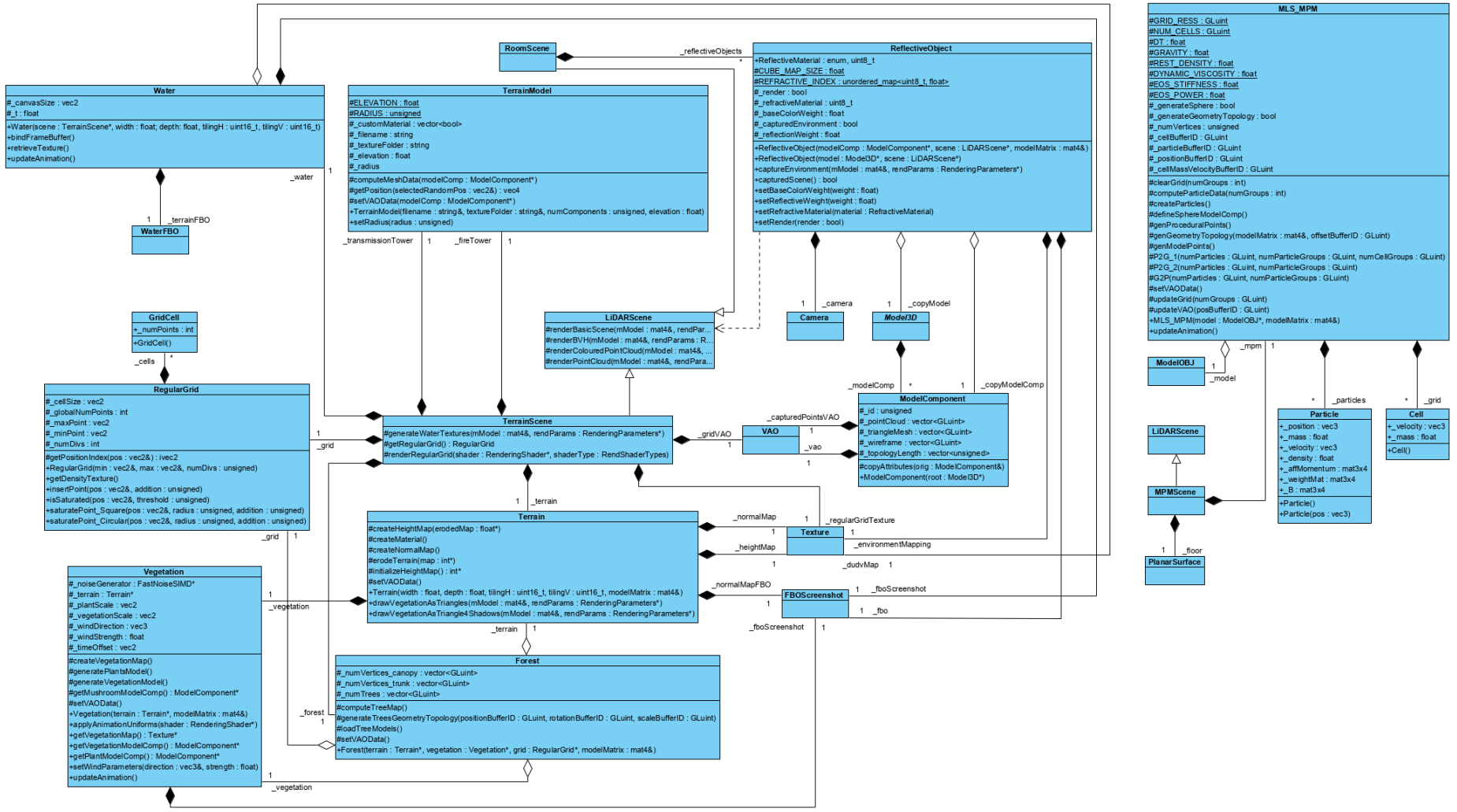


Ilustración 159. Diagrama de clases correspondiente a la iteración cinco. Únicamente se muestran aquellas clases introducidas y modificadas en esta iteración.

2.7 Sexta iteración

La penúltima iteración de este trabajo tiene como objetivo desarrollar la interfaz de la aplicación, la cual permitirá al usuario modificar el *renderizado* del escenario o configurar cualquier proceso, como podría ser el proceso de escaneo LiDAR. Igualmente, aprovechando la introducción de la interfaz, se añaden nuevos modos de *rendering* que podrán ser seleccionados a través de un menú. En el punto en el que nos encontramos aún no se ha completado el proceso de escaneo, y esto es algo que habrá que considerar en la primera sección de este apartado.

2.7.1 Desarrollo de la interfaz gráfica

En las primeras iteraciones se citaba una entidad ***GUI***, la cual únicamente inicializaba el contexto de la librería ***Dear ImGui***. Por tanto, el trabajo desarrollado en esta iteración completa dicha entidad a través de un método `render`, el cual deberá ser invocado desde *InputManager*, la entidad donde se reciben los eventos de redibujado (y cualquier otro evento de la aplicación). A diferencia de este evento, la captura de la escena no incluye la interfaz, dado que se considera que su propósito está más bien relacionado con la documentación del escenario, y no de la aplicación.

El aprendizaje de la sintaxis de la librería no se produce a través de una documentación (de la cual carece), sino de un ejemplo que incluye Dear ImGui precisamente con este propósito. En dicho ejemplo se incluyen todas las opciones posibles en la librería. Si el lector se encuentra familiarizado con OpenGL, un ejemplo representativo de la sintaxis de esta librería serían las operaciones `glBegin` y `glEnd`. De esta manera, para crear un *widget* es necesario indicar una orden similar a `ImGui::BeginWidget`, y deberá cerrarse este contexto mediante `ImGui::EndWidget`. A diferencia de `glBegin`, `ImGui::BeginWidget` no siempre permite iniciar un contexto, aunque no se debe entender como una imposibilidad de creación, sino como el resultado del estado de la aplicación. Por ejemplo, podría controlarse la creación de una ventana mediante un *checkbox*, cuyo valor booleano posibilite o no su representación, aunque no es la situación más frecuente; el escenario más normal puede impedir iniciar un *widget* cuando este se ha colapsado u ocultado en la interfaz. Nótese como esta librería dispone de un gran número de herramientas en forma de elementos gráficos que podemos representar en la aplicación, pero su respuesta ante

determinados eventos es completamente opaca para el programador. Esta situación presenta ventajas y desventajas; ciertamente, nos libera de controlar la actualización de la interfaz frente a los eventos de la aplicación, pero nos impide conocer si un determinado evento, como un *click*, ha tenido efecto en la interfaz. En última instancia, esto repercute en la capacidad de reducir *renderings*, dado que ante ciertos eventos (y el desconocimiento de sus consecuencias) habrá que *renderizar* de nuevo la escena (incluso más de una vez, para cerciorarnos de que interfaz y escena se actualizan correctamente).

En este punto de la aplicación, aún no hemos explorado el comportamiento completo del escaneo LiDAR, por lo que buena parte de la interfaz de la solución final aún no se ha desarrollado. En cualquier caso, no se pretende que este apartado sirva como referencia para conocer qué elementos conforman la interfaz o cuál es su finalidad; para ello ya disponemos de la sección Manuales de usuario. Por tanto, aquí nos limitaremos a describir aquellas decisiones tomadas sobre la interfaz.

En primer lugar, la raíz de esta interfaz será una barra de menús, que nos dará acceso al resto de ventanas de la interfaz, pero que también nos permitirá comprobar FPS o bloquear el movimiento de la cámara. Una consecuencia de no conocer si un evento de usuario ha tenido efecto sobre la interfaz es que dicho evento puede actuar simultáneamente sobre la interfaz y la escena. Por ejemplo, deslizar un *slider* podría también originar un movimiento *pan* de la cámara. Por tanto, se habilita un bloqueo de cámara.

Ilustración 160. Barra de menús y opciones disponibles.

Los submenús considerandos se enumeran a continuación:

1. ***Rendering***. Permite controlar qué se visualiza y cómo. Por ejemplo, el usuario puede seleccionar la representación de cierto tipo de nube de puntos, así como el tamaño de los propios puntos.
2. ***Screenshot***. Nos permite configurar el factor por el que debe multiplicarse el tamaño actual de la ventana, a partir del cual se obtiene el tamaño de las imágenes que se capturen. También es posible indicar el nombre del fichero donde se almacena la imagen (por defecto, *Screenshot.png*).
3. ***LiDAR***. Configura todo el proceso de escaneo, y por tanto, podemos entender que será el submenú más importante. Se divide a su vez en tres pestañas: configuración general, configuración relacionada con LiDAR aéreo y configuración de LiDAR terrestre. Se encuentra mucho más completa la pestaña general; las pestañas específicas de cada tipo de LiDAR presentan opciones muy puntuales.
4. ***Scene***. Control de las propiedades de la escena. Se crea este submenú con el único objetivo de modificar las propiedades más bien relacionadas con el material de cada elemento de la escena, por lo que las principales consecuencias se encuentran en ciertos tipos de visualizaciones, aunque también se identifican errores que podrán modelarse modificando el valor de algunos de estos parámetros.
5. ***Point Cloud***. Permite configurar el almacenamiento de la nube de puntos seleccionando el nombre del fichero donde se almacena el resultado. También es posible indicar si se desea almacenar la nube tras cada escaneo.

Además de la configuración, también es posible acceder a una ventana de ayuda (opción *Help*), donde se muestran los controles de la aplicación, así como una pequeña descripción del proyecto.

Se debe destacar el importante papel que toma una entidad citada al inicio del desarrollo, ***RenderingParameters***. No es más que una estructura accesible a través del *Renderer* de la aplicación, que como bien indica su nombre, almacena todos los parámetros relativos al *rendering* de la aplicación. Más allá de los valores numéricos, dispone de una extensa lista de atributos booleanos que servirán como nexo de *ImGui*

y nuestra aplicación. Por ejemplo, al crear un *widget* de tipo *checkbox* habrá que indicar la dirección en memoria de una variable booleana, que será la que se modifique cuando el usuario haga uso de dicho *widget*.

2.7.2 Nuevos modos de rendering de nube de puntos

Aprovechando la introducción de un menú de control de *rendering*, es posible introducir algún modo de visualización nuevo. Más concretamente, se incluye el *renderizado* de la nube mediante una textura que indicará la altura de cada punto dentro del AABB de la escena, de tal manera que en un *shader* esto se traduce en la utilización de unas coordenadas de textura (u, v) como $(height, 0.5)$ o $(0.5, height)$, en función de si la textura es horizontal o vertical, respectivamente. Este *rendering* se puede observar en la Ilustración 161 y la Ilustración 162.

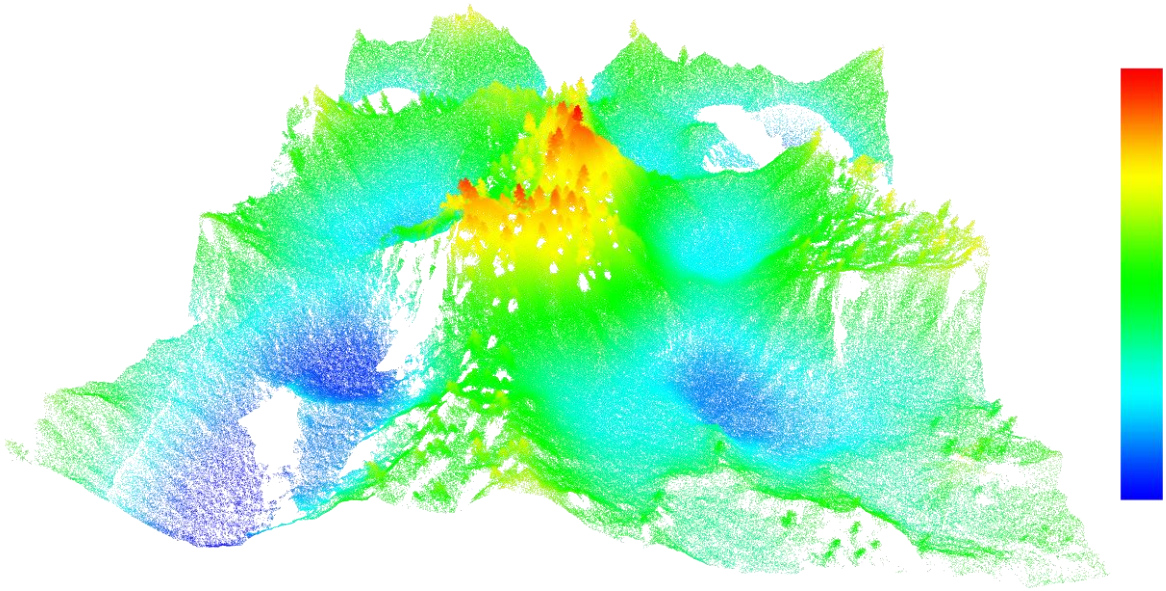


Ilustración 161. Rendering de nube de puntos en función de la altura de cada punto sobre el escenario del terreno.

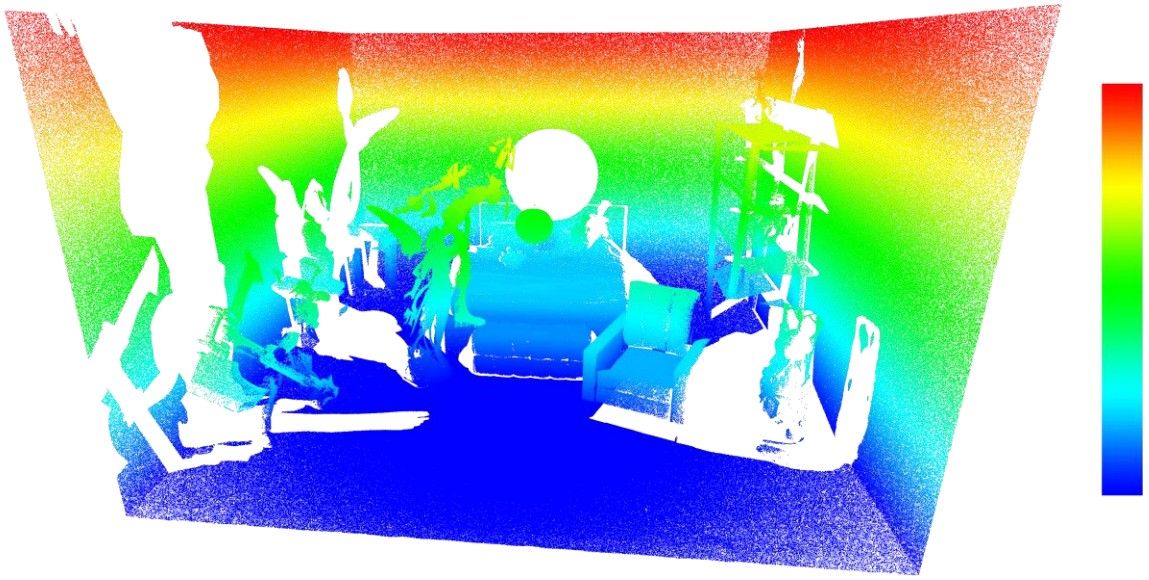


Ilustración 162. Rendering de nube de puntos en función de la altura de cada punto sobre el escenario de la habitación.

Este mismo modo de visualización se puede presentar mediante una escala de grises, donde el cero absoluto se alcanza en la altura mínima (Ilustración 163).

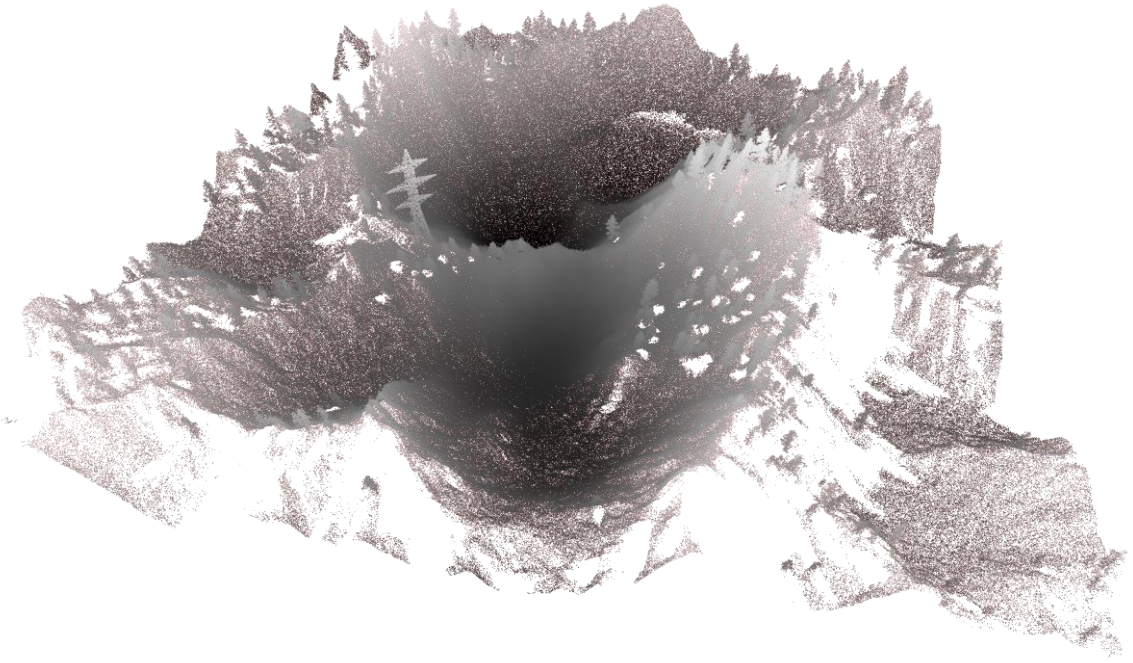


Ilustración 163. Rendering de nube de puntos en función de la altura de cada punto sobre el escenario del terreno (escala de grises).

Además de este nuevo modo de *rendering*, se introduce la selección del color de la nube de puntos uniforme, como bien se muestra en la Ilustración 164.

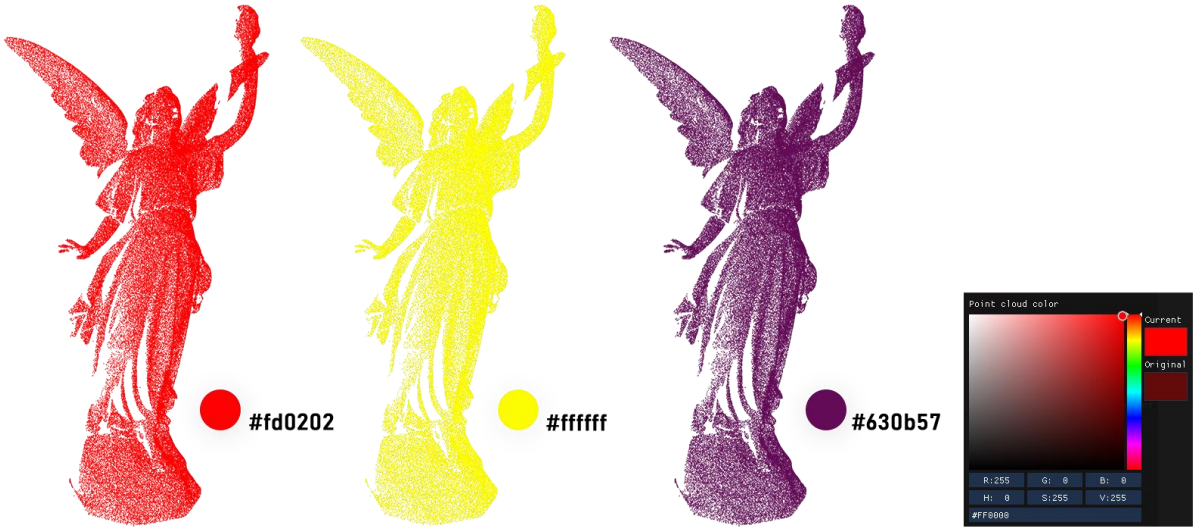


Ilustración 164. Nube de puntos renderizada con colores seleccionados de la interfaz.

2.7.3 Escaneo LiDAR aéreo

El desarrollo de la interfaz nos permite completar el segundo tipo de escaneo LiDAR contemplado en este trabajo, el aéreo, donde el dispositivo no establece una posición estática, sino que puede desplazarse a lo largo de un escenario sobre un vehículo aéreo. Se introduce en esta iteración debido a que una de las posibilidades de vuelo consiste en la especificación de la ruta de manera manual por el usuario. Sin embargo, el escaneo más completo y simple consiste en recorrer todo el escenario realizando pequeños desplazamientos. En este punto se considera que el escaneo que se realiza es completo, dado que la ejecución incremental se describe en el siguiente apartado.

Para llevar a cabo el escaneo automático de la escena desde un vehículo aéreo se define una ruta, un vector de `vec2`, en el inicio de la aplicación, una vez se conoce la caja envolvente del escenario. De esta manera, partiendo de $aabb_{min}$ es posible llevar a cabo desplazamientos en X y Z, donde el incremento en ambos ejes vendrá dado por `AERIAL_INC`, una constante que en nuestro trabajo recibe el valor 0.03. Así, se discretiza el problema de la traslación a lo largo del mapa y se definen un conjunto de puntos estáticos desde donde deberán lanzarse un conjunto de rayos. Cuando el escaneo es automático (la ruta no la define el usuario), la ruta no es importante desde el punto de vista gráfico, y por esta razón, se simplifica en gran medida la definición de estos puntos. Partimos de x_0 y z_0, y cuando se ha recorrido el eje Z completamente en x_0, es posible volver a z_0 en x_1. Dicho salto no representa un problema dado que el objetivo es escanear el escenario completo desde el aire.

El número de rayos que se define en un determinado punto se calcula como sigue:

$$n_{rays} = ceil\left(\frac{N_{rayos}}{\frac{aabb_{size_x}}{AERIAL_{INC}} * \frac{aabb_{size_z}}{AERIAL_{INC}}}\right) = \left(\frac{N_{rayos}}{aerialPath_{size}}\right) \quad (2.88)$$

El principal problema del lanzamiento de rayos en un punto es la definición del espacio donde deben instanciarse. De nuevo, dado el problema del patrón Moiré antes expuesto, los rayos deben instanciarse mediante una distribución uniforme que genera cierta irregularidad en la dirección de los mismos. En este punto, también disponemos de una función previamente empleada, `generateRandomNumber`, que generará un valor flotante mediante dicha distribución en el intervalo $[0, 1]$, aunque podrá indicarse rango y valor mínimo para transformar el intervalo en $[value_{min}, range + value_{min}]$.

Dada una posición p, una dirección de lanzamiento de rayos, dir, y un ángulo α que representa $\frac{\pi - \beta}{2}$, sea β el ángulo de apertura del dispositivo, entonces es posible calcular un único rayo mediante una distribución aleatoria uniforme tal y como se muestra a continuación:

$$range = \pi - 2 * \alpha; \;\; offset = -\frac{\pi}{2} + \alpha$$

$$sphere_{xz\_pos} = (dir_x, 0, dir_z)$$

$$axis_{rotation} = \left(-sphere_{xz\_pos_z}, 0, sphere_{xz\_pos_x}\right)$$

$$rotation_{mat} = rotate\left(-\frac{\pi}{2}, axis_{rotation}\right) \quad (2.89)$$

$$sphere_{pos} = rotation_{mat} * rotation(distUniform(range, offset), axis_{rotation}) * sphere_{xz\_pos})$$

$$ray = Ray(p, p + sphere_{pos} + jittering)$$

Mediante $axis_{rotation}$ y el valor aleatorio se pretende rotar dir dentro del intervalo $[-\frac{\pi}{2}, \frac{\pi}{2}]$ (si $\alpha = 0$) respecto de $axis_{rotation}$, mientras que $rotation_{mat}$ vuelve a rotar la dirección obtenida respecto de dir_{perp} para que el valor Y de la dirección sea siempre menor o igual que 0. La Ilustración 165 debería permitirnos comprender este

proceso. Nótese como la dirección dir indicada no es la dirección de movimiento, sino un vector perpendicular a la dirección, $(dir_z, 0, -dir_x)$.

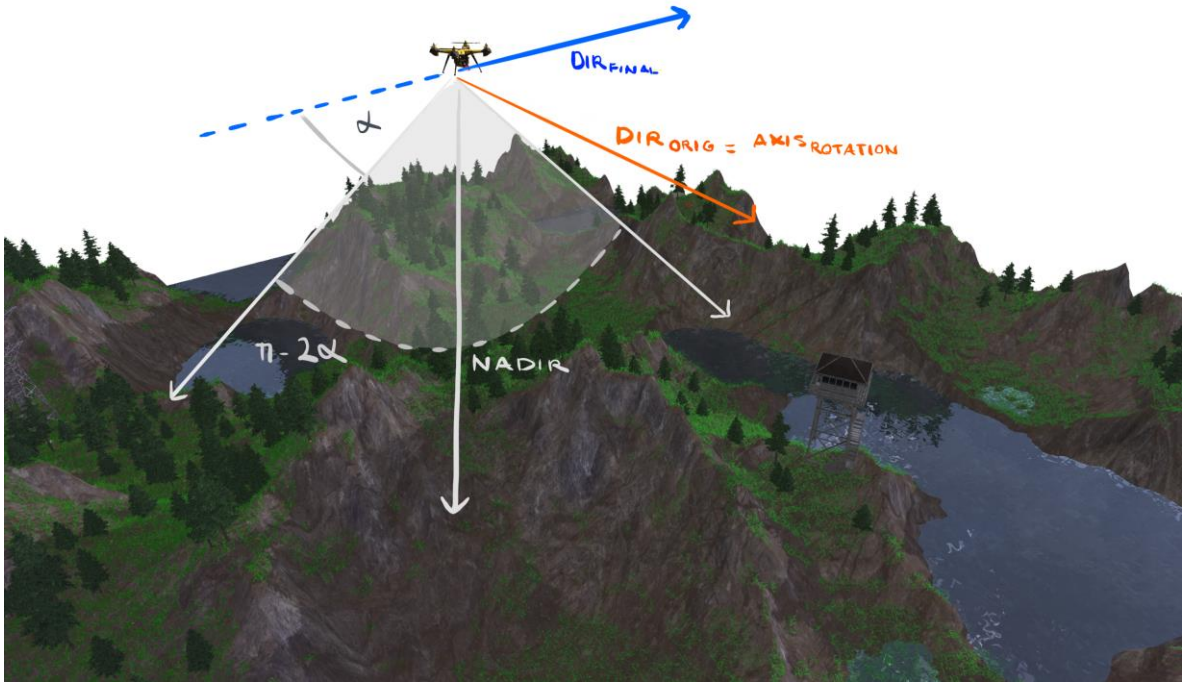


Ilustración 165. Representación de algunos vectores importantes en la generación de las emisiones aéreas.

Nótese como no sólo se genera un rayo por posición, sino que deberán generarse n_{rays}, y para todos ellos deberá calcularse $sphere_{pos}$ y ray. En la construcción del rayo se añade una variable $jittering$, un `vec3` que también se construye mediante una distribución uniforme. Dicho valor añade cierta distorsión a la dirección del rayo, lo que implica que no todos los rayos se generan en el plano formado por los vectores $(0, 1, 0)$ y $axis_{rotation}$, evitando de nuevo la visualización de posibles patrones.

El escaneo aéreo automático produce resultados tales como los que se muestran en la Ilustración 166.

Más allá de una ruta exhaustiva y paralela por el escenario, pueden plantearse otros tipos de rutas que podrían programarse en un vehículo aéreo: zigzag, elipses, etc (Dong & Chen, 2018). Sin embargo, en lugar de permitir al usuario seleccionar un tipo de movimiento de una lista, se ha habilitado un *canvas* en la interfaz que permite dibujar cualquier tipo de recorrido. No se trata de una herramienta que deba emplearse para obtener una nube de puntos exhaustiva y realista, sino que más bien se trata de una opción que permite al usuario comprobar qué se obtiene con una ruta dibujada

manualmente. La nube de puntos que se obtiene mediante esta ruta no será nunca tan completa como la que se obtendría en la versión automática.

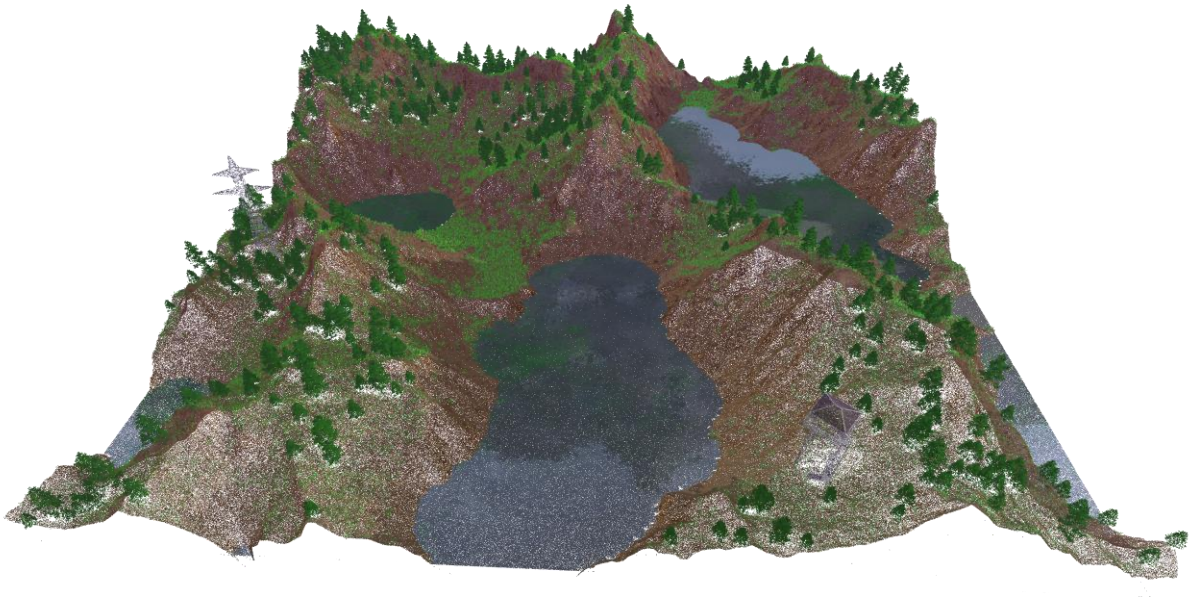


Ilustración 166. Escaneo aéreo de un terreno. Configuración: $\alpha = 45°$, número de rayos: 9 millones de rayos.

La ruta dibujada sobre un *canvas* se representa mediante una interpolación lineal, aunque la apariencia de la misma no tiene por qué representar ese comportamiento; el usuario pueda dibujar con total libertad, lo que incluye la introducción de curvas, aunque a bajo nivel estas se representen mediante pequeños segmentos. Una alternativa sería el desarrollo de una curva Spline, lo cual es completamente innecesario, debido a que genera una carga de trabajo elevada y el usuario puede igualmente modelar una curva. Por otro lado, el *canvas* definido en la interfaz representa el AABB de la escena, incluso cuando la relación de aspecto del *canvas* no es equivalente al de dicha caja envolvente (sólo XZ).

Una interpolación lineal se define como un conjunto de puntos, que podría representarse a partir de segmentos compuestos por dos puntos adyacentes. Su comportamiento se implementa bajo la entidad ***LinearInterpolation***, que a su vez toma ciertos comportamientos de una clase base, ***Interpolation***, bajo la cual también podríamos implementar una curva Bézier o una Spline, si fuera necesario.

El principal problema que nos ocupa en la utilización de esta interpolación es la obtención de puntos pertenecientes al conjunto de segmentos definidos. Una manera muy simple de resolver la obtención de puntos en la simulación es la utilización de una variable t, que representa un valor paramétrico que deberá emplear la interpolación

para obtener un punto. Por tanto, la simulación LiDAR sólo deberá consultar el punto asociado al valor paramétrico t y obtener el conjunto de rayos asociado a esta posición. La dirección de traslación del vehículo vendrá dada por t y $t + t_{increment}$, de tal manera que el punto en t_{end} no constituye un punto de escaneo (podríamos calcular la dirección considerando $t - t_{increment}$, aunque no es relevante perder un valor de t cuando se dispone de miles de valores anteriores).

En este trabajo, $t_{increment}$ se calcula de manera independiente a la longitud de los segmentos definidos en la interpolación, dado que depende de $\frac{K}{N_{rayos}}$, por lo que un valor menor de $t_{increment}$ podría obtenerse aumentando el número de rayos. De esta manera, evitamos reducir el número de rayos que se lanza en cada punto cuando la ruta definida es más amplia. K se define en la aplicación como 5.000. A partir de $t_{increment}$, se determina que $n_{rays} = K = N_{rayos}/t_{increment}$.

La consulta de un valor t dentro de una interpolación lineal se optimiza para que no sea necesario recorrer toda la estructura buscando el par de puntos entre los que se encuentra dicho valor. Antes de comenzar la simulación se construye un vector que indica el valor t que le correspondería a cada punto almacenado. Asimismo, se inicializa una variable `currentIndex` con el valor uno. Dicho índice indica dónde debe comenzar la búsqueda, de tal manera que se considera que siempre se consultan valores de t que irán aumentando, hasta llegar a $t = 1$. Su valor no se modifica mientras que $tVector[currentIndex] \geq t$, en cualquier otro caso habrá que incrementar su valor en uno, o finalizar si el índice incrementado equivale al número de puntos contenidos en la interpolación.

Dado un valor de t, la posición correspondiente se calcula como sigue:

$$weight = \frac{(t - tVector_{currentIndex-1})}{tVector_{currentIndex} - tVector_{currentIndex-1}} \tag{2.90}$$

$$p = weight * p_{currentIndex} + (1 - weight) * p_{currentIndex-1}$$

De este modo, podemos trazar sobre un *canvas* una ruta como la que se muestra en la Ilustración 167, obteniéndose el resultado que se representa en la misma imagen.

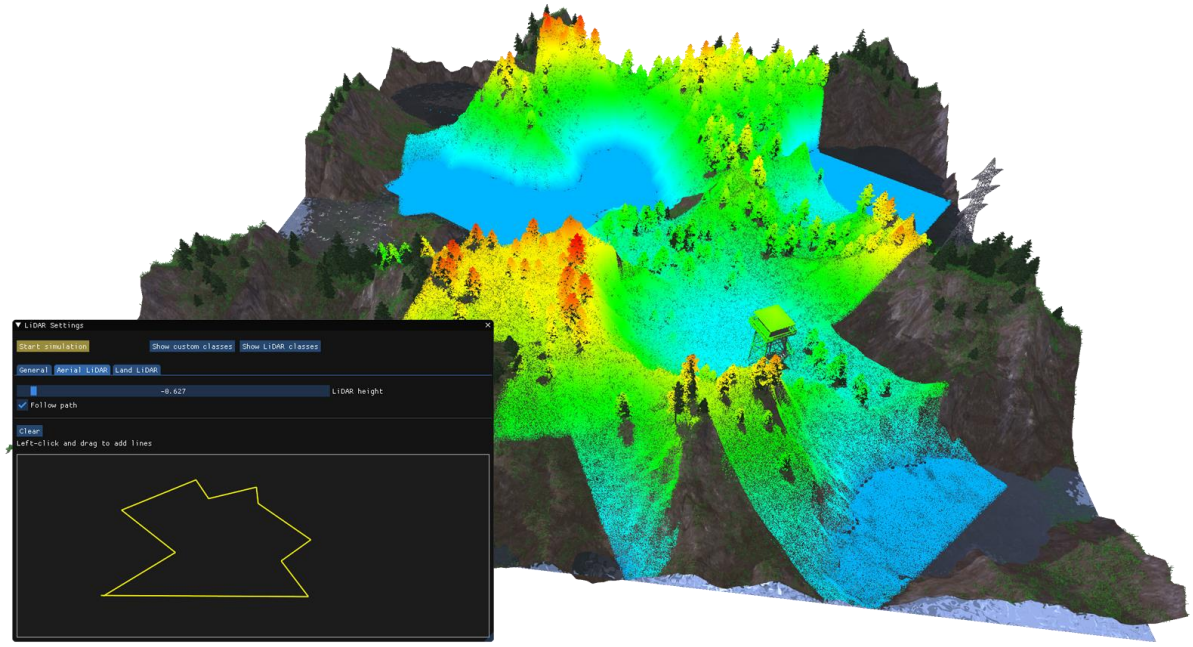


Ilustración 167. Representación de ruta trazada sobre el escenario. Resultado de escaneo siguiendo dicha ruta.

2.7.4 Escaneo incremental

Una vez desarrollados tres tipos básicos de escaneo (terrestre y aéreo, donde este último diferencia ruta manual y automática) podemos habilitar un proceso iterativo que permita al usuario descubrir el resultado mediante pequeños pasos. Dada la implementación de todos estos tipos de escaneos, no es complejo desarrollar una versión incremental sobre ellos. La base consistirá únicamente en un valor booleano, `finished`, que indica si el escaneo activo ha finalizado. Así, cuando el usuario pulse una tecla, habilitada para continuar con el proceso, se comprueba si este ha finalizado. Si no es así, deben tenerse en cuenta los siguientes aspectos:

- Escaneo terrestre. Deberá almacenarse un ángulo α que indica el siguiente escaneo vertical que debe llevarse a cabo. Por tanto, $\alpha \in [0, 2\pi]$. De esta manera, $finished = \alpha > 2\pi$. El principal problema al que nos enfrentamos en este, y en el resto de escaneos incrementales, es el progreso tan lento de la animación, dado que disponemos de millones de rayos, y por tanto, los incrementos de α son muy pequeños. La solución que se propone es resolver, en una única iteración, el escaneo derivado de varios valores de α. Se puede observar en la Ilustración 168 un proceso incompleto de escaneo de la primera escena:

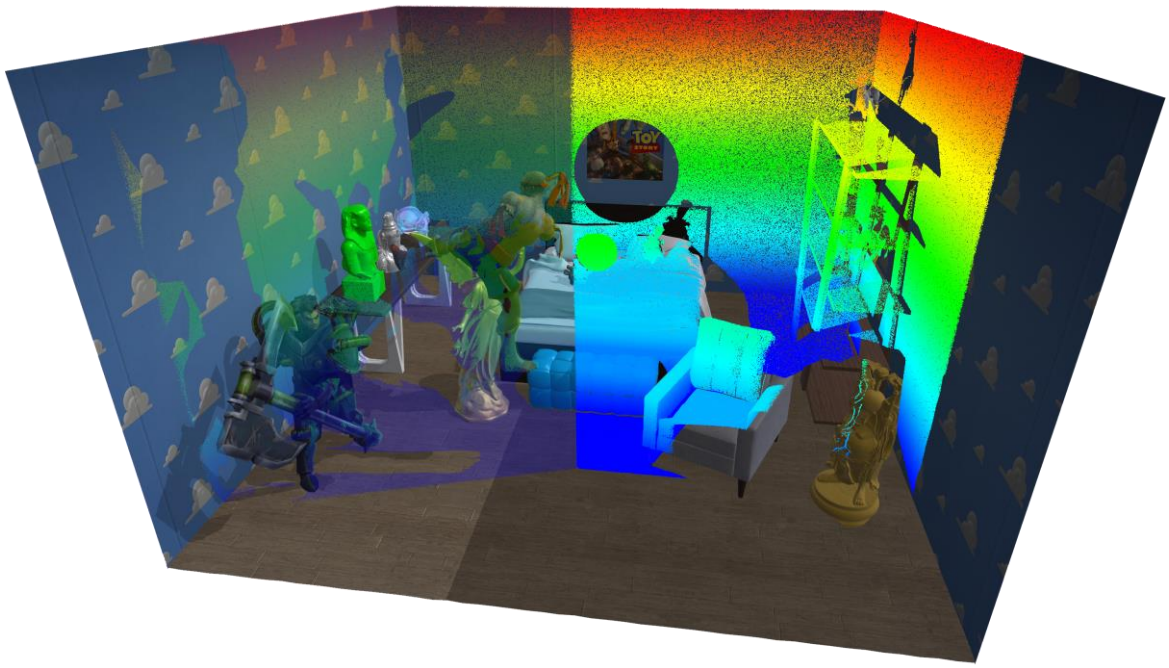


Ilustración 168. Composición de dos estados del escaneo incremental de la primera escena.

- Escaneo aéreo. Habrá que distinguir entre el escaneo automático y manual:
  - Automático. Al inicio de la aplicación se construye una ruta prefijada a partir del volumen envolvente de la escena. Por tanto, parece suficiente con almacenar el índice de la siguiente posición que debe explorarse, obteniéndose que $finished = index >= path.size()$. Del mismo modo que antes, varias iteraciones se resuelven en un único incremento para acelerar la animación. Este proceso incremental se refleja en la Ilustración 169.

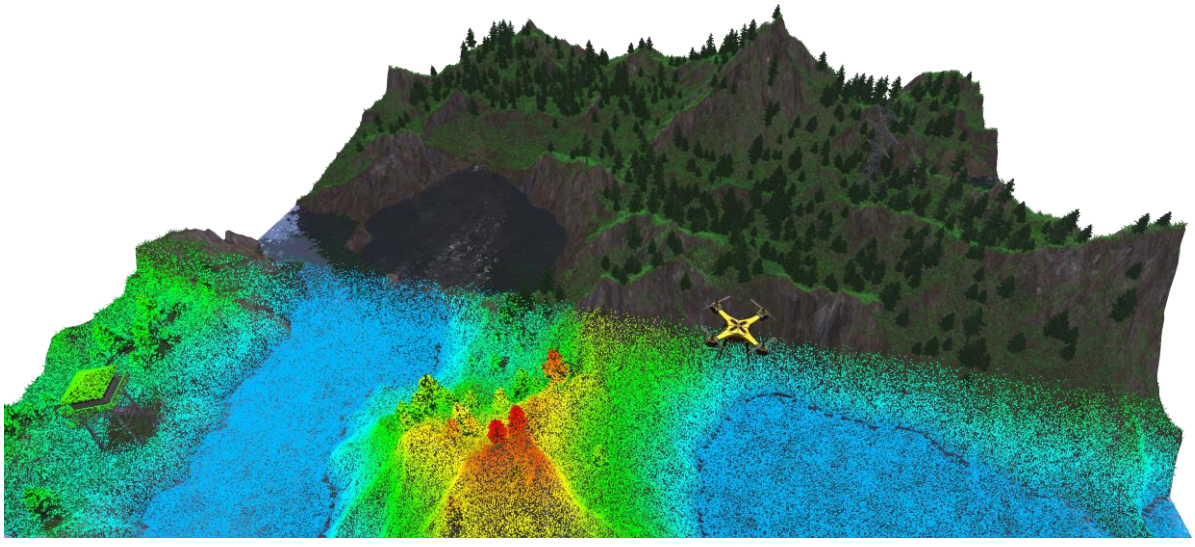


Ilustración 169. Escaneo incremental de un terreno mediante una ruta automática.

- Manual. A nivel teórico puede ser el caso más complejo, aunque en la práctica, su implementación es muy básica. Basta con almacenar el valor parámetrico t, de tal manera que $finished = t \geq 1$, aunque esto mismo nos lo indicará la propia entidad de la interpolación lineal. Se complicaría mucho más si se empleara el último valor de t, t_{end}, dado que el enfoque planteado de la interpolación es incremental, pero en la práctica se puede descartar dicho valor de t, teniendo en cuenta que $t_{end} - t_{increment}$ se encuentra muy cercano a t_{end}. En la Ilustración 170 se puede observar este proceso, junto a su ruta previamente definida en un *canvas*.

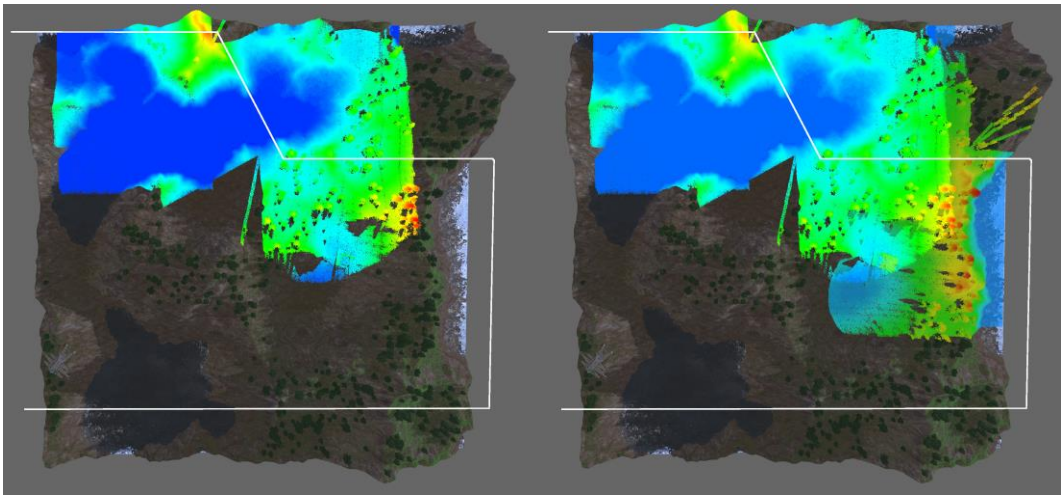


Ilustración 170. Escaneo incremental de un terreno junto a la ruta establecida.

Se debe destacar el papel de una entidad ***LiDARParameters*** en los últimos procesos introducidos, dado que actúa como una base de datos donde se almacenan todos los parámetros del escaneo (número de rayos, ángulos de *offsets*, tipo de LiDAR, seguimiento de ruta manual, etc). Esta entidad sirve igualmente como nexo entre la propia interfaz de la aplicación y la simulación.

2.7.5 Diagrama de clases

En el diagrama de clases de esta penúltima iteración se muestran aquellas entidades introducidas recientemente en la aplicación, así como aquellos cambios que hayan podido producirse sobre entidades ya existentes, tales como *LiDARSimulation* o *Model3D* (Ilustración 171). No existe necesariamente una conexión entre todas las entidades representadas en tanto que se trata de una etapa de refinamiento de comportamientos anteriores.

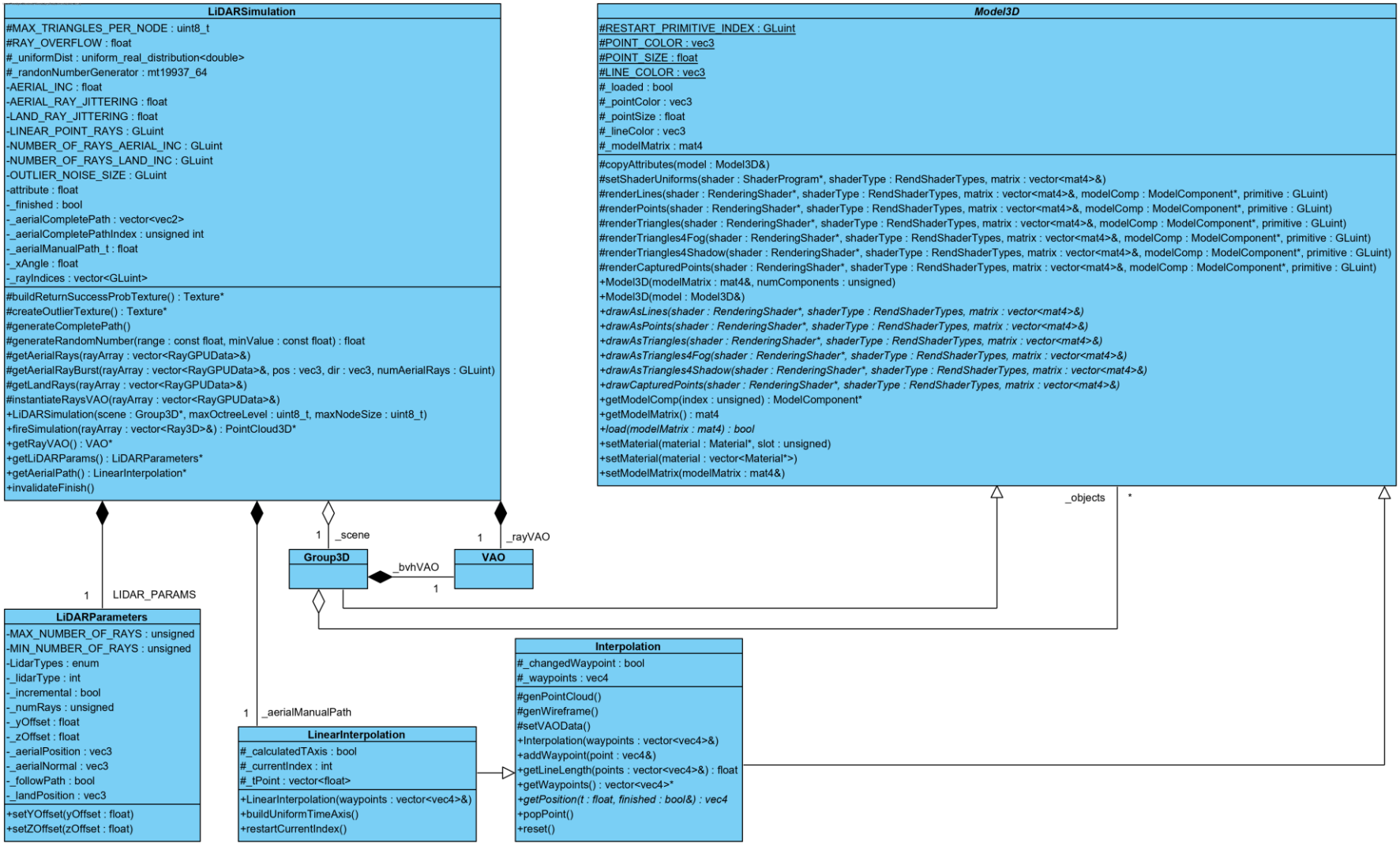


Ilustración 171. Diagrama de clases donde se muestran aquellas entidades modificadas en la iteración seis.

2.8 Séptima iteración

El objetivo de la última iteración es el perfeccionamiento del proceso de escaneo, no sólo a través de la extensión de su comportamiento, sino también a través de la introducción de errores documentados en la bibliografía de este área. Este desarrollo vendrá acompañado de otras formas de visualización de la escena, como resultado de la información obtenida en el nuevo proceso de escaneo. Por último, también se desarrolla la escritura de una nube de puntos (obtenida en el escaneo) en un fichero PLY mediante las librerías citadas en la introducción de este documento.

2.8.1 Definición de componente de un modelo

La estructura final de un componente integrado en cualquier modelo debe describirse antes de comenzar con los siguientes procesos, dado que aquí será donde se almacenen todas las nuevas propiedades. Dicha estructura se muestra en la Ilustración 172, y se describe como sigue:

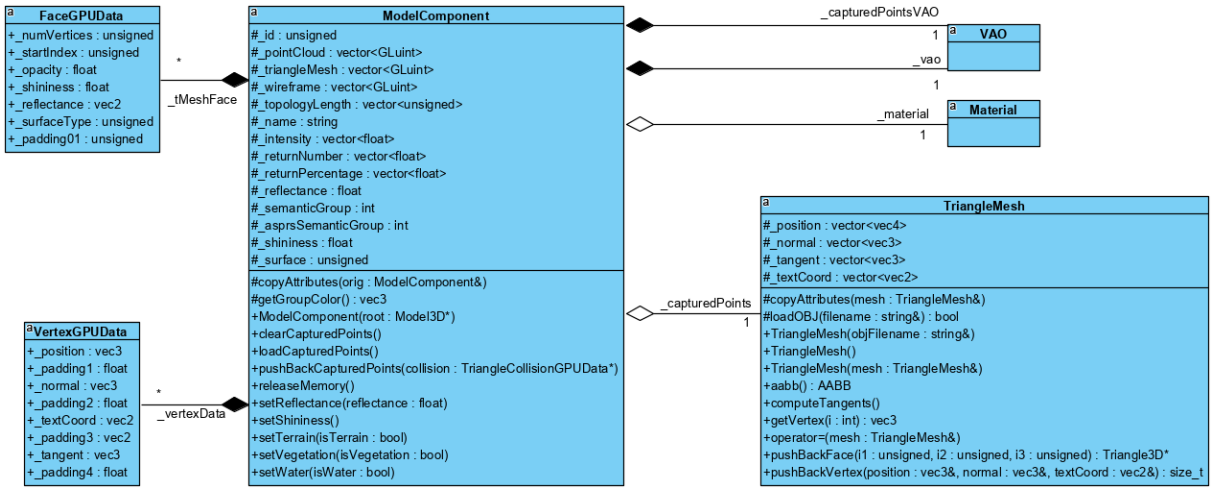


Ilustración 172. Diagrama de clases con la estructura de la entidad ModelComponent.

- Nombre. No es una propiedad ligada al comportamiento del LiDAR, sino más bien a la interfaz. Uno de los submenús disponibles permite modificar las propiedades de cada uno de estos componentes, por lo que su nombre nos ayudará a distinguirlos.
- Intensidad. Valor flotante que se encuentra ligado a la intensidad captada por el sensor LiDAR. En la práctica, este valor depende de un gran número de factores.

- Tipo de superficie. Existen superficies especialmente interesantes en el escaneo, por tanto, parece relevante hacer distinción de aquellos componentes que representan comportamientos diferentes. El objetivo en GPU es reducir la memoria empleada, y por tal razón, se comprimen todos los tipos de superficies en un único valor entero sin signo. Cada tipo de superficie (en este caso, terreno, agua y vegetación) representa un bit diferente dentro de dicho valor, por lo que el comportamiento es muy similar al de las capas en Unity3D; la versión gratuita dispone de 32 bits, y por tanto, disponemos de hasta 32 posibles superficies o grupos. Dentro del *shader*, es posible elaborar tantas máscaras como superficies se distingan (como constantes definidas mediante la sintaxis `#define`).
- Números de retorno. Para cada colisión, se almacena el número de retorno que representa, comenzando en cero.
- Número de retorno como porcentaje. A diferencia de los valores anteriores, estos se normalizan respecto del número total de retornos para una determinada colisión. Por ejemplo, si una colisión representa el tercer retorno de una emisión que ha originado seis colisiones, este valor se correspondería con $\frac{1}{2}$.
- Reflectancia. Es una propiedad que se utiliza en la simulación para afectar a la intensidad, de tal manera que se puede comprender como la capacidad de emisión de un material.
- Grupo semántico ASPRS. A partir del estándar 1.4 de LAS, se define un extenso conjunto de clases que reciben un identificador y una cadena descriptiva de dicha clase. Por tanto, cada componente podrá asociarse con una clase diferente del estándar.
- Grupo semántico. El concepto es muy similar al atributo previo, aunque la principal diferencia es que este grupo semántico no sigue ningún tipo de estándar, y por tanto, existe total libertad a la hora de definir el nivel de granularidad de nuestras clases. Mientras que el estándar ASPRS define la clase *Building*, este atributo puede distinguir paredes, mesa, silla, etc.

- Brillo. Una superficie donde suelen presentarse errores es en aquella que se asemeja a un espejo, y como tal, se incluye este factor para caracterizar este comportamiento.

Además, todos estos atributos y comportamientos se ven reflejadas en otras dos entidades que sirven como nexo con la GPU:

- ***MeshGPUData***. Una malla se define no sólo como un conjunto de triángulos, sino también como una superficie, la cual tendrá propiedades tales como brillo, opacidad (aún por describir), reflectancia, o el tipo de superficie. Esta última propiedad obtiene el valor almacenado en la instancia de *ModelComponent*.

- ***TriangleCollisionGPUData***. Almacena nueva información derivada de la colisión, tal como el componente con el que colisiona (nótese como esto nos dará acceso a la clase a la que pertenece la colisión), el número de retornos, el número total de retornos vinculados al rayo que provocó la colisión, el ángulo de incidencia, la intensidad evaluada, o el identificador de la colisión previa. Algunos de estos atributos se emplean durante el escaneo, pero no son necesarios en el *rendering*; por ejemplo, el ángulo de incidencia puede almacenarse en un fichero LAS, pero no tiene aplicación en este trabajo por sí sólo.

Ambas entidades, con sus modificaciones, se representan en la Ilustración 173.

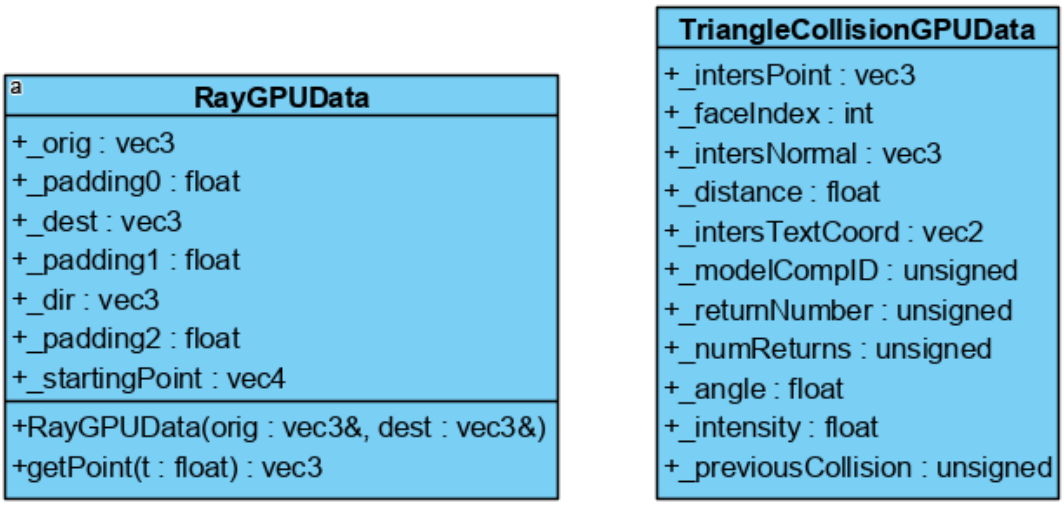


Ilustración 173. Diagrama de clases de las entidades MeshGPUData y TriangleCollisionGPUData.

2.8.2 Nuevos comportamientos de sensor LiDAR

En este apartado se describe la forma final del escaneo LiDAR, con todos sus errores y comportamiento adicionales, comenzando con una explicación de sus consecuencias en la vegetación, y finalizando con otras superficies donde también se observan problemas.

2.8.2.1 Vegetación. Múltiples retornos

La vegetación es una de las superficies más interesantes en un escenario de un terreno. Es común observar en este apartado el concepto de retorno (Dong & Chen, 2018), de tal manera que una emisión LiDAR se modela como un rayo que puede seguir su curso a través de ciertas superficies, no necesariamente afectando su dirección. De hecho, el tipo de fichero LAS, utilizado para almacenar nubes de puntos procedentes de sensores LiDAR, entre otros, define en su especificación atributos tales como `Return Number` o `Number of Returns (Given Pulse)` (The American Society for Photogrammetry & Remote Sensing, 2019). También es interesante observar el tamaño reservado para ambos atributos. En este caso, se definen 3 bits para cada uno, lo cual nos puede hacer pensar que son posibles hasta 8 retornos, sin embargo, la especificación de la cabecera también define un atributo, `Number of Points by Returns`, que no es más que un vector de 5 valores. Es decir, se considera que hasta cinco retornos son posibles con la tecnología actual (la última fecha de revisión del estándar 1.4 es el 9 de julio de 2019).

La introducción de múltiples retornos es probablemente una de las características más interesantes, dado que son muchas las aplicaciones del filtrado de puntos por el valor de retorno. Por ejemplo, descartando los primeros retornos, podemos desechar la cubierta de un bosque y recuperar únicamente el suelo. Una aplicación de este proceso es el hallazgo de restos arqueológicos en zonas inaccesibles, o de difícil visibilidad (Blakemore, 2019) (Melin, C. Shapiro, & Glover-Kapfer, 2017).

En la práctica, este enfoque supone un cambio importante del comportamiento hasta ahora descrito, dado que partimos de una versión donde sólo se considera la primera colisión, y nos desplazamos a otra versión en la que pueden existir varias colisiones (habrá que iterar doblemente; desplazamiento en un BVH, e iteración buscando diversas colisiones). De esta manera, el nuevo comportamiento se

representa en la Ilustración 174 y la Ilustración 175. Conocido el número máximo de colisiones, es posible definir un vector de tal tamaño, donde se almacenan los índices de aquellas colisiones originadas por el rayo asignado al hilo $tIndex$. Cuando no se identifican más colisiones, se debe iterar sobre todas ellas para actualizar el valor de `numReturns`. Nótese como `returnNumber` se debe *castear* a un valor entero con signo; en cualquier otro caso `returnNumber - 1` puede producir valores no deseados. Igualmente, su valor aumenta en cada iteración si existe una colisión (`continueRay`). Un valor booleano se puede *castear* a entero (con o sin signo) para producir un valor uno (*true*) o cero (*false*). Sólo continúa el proceso iterativo, aunque exista una colisión, si se dan dos condiciones: 1) el modelo intersectado representa vegetación, y 2), no se ha superado el máximo número de retornos indicados. A diferencia de la especificación LAS, se ha permitido que el número de retornos pueda ser mayor que cinco, e incluso menor de este número; es el usuario a través de la interfaz quien puede indicar este valor.

Algorithm 21 *IterateCollisions*: Resolución de múltiples colisiones de emisiones LiDAR con un BVH

```
1:  Sea tIndex el identificador del hilo
2:
3:  num_returns ← 0
4:  collisionIndices ← uint[MAX_COLLISIONS]
5:  continueRay ← true
6:
7:  index_current ← 0
8:  heap ← unsigned[100]
9:  heap[index_current] = num_clusters − 1
10:
11: while continueRay do
12:     continueRay ← checkCollision(tIndex, ray[tIndex], returnNumber, index_final)
13:
14:     collisionIndices[num_returns] ← index_final
15:     returnNumber+ = uint(continueRay)
16:     continueRay &= mesh[collision[tIndex].modelComp_ID]_surface & VEG_MASK
17:     continueRay &= num_returns < MAX_RETURNS
18: end while
19:
20: collisions_count ← num_returns − 1
21: while collisions_count >= 0 do
22:     triangleCollision[collisionIndices[collisions_count]].numReturns ← num_returns
23:     collisions_count −= 1
24: end while
```

Ilustración 174. Pseudocódigo de recuperación de múltiples colisiones para un mismo rayo.

Algorithm 22 $RayBVHCollision\_v_2$: Resolución de la colisión rayo-BVH en la séptima iteración

```
1: Sea tIndex el identificador del hilo
2:
3: collision[tIndex].face ← null
4: collision[tIndex].distance ← ∞
5:
6: index_current ← 0
7: heap ← unsigned[100]
8: heap[index_current] = num_clusters − 1
9:
10: while index_current >= 0 do
11:     cluster ← bvh[heap[index_current]]
12:
13:     if intersectAABBRay(cluster, ray) then
14:         if cluster.face! = null then
15:             if intersectTriangleRay(tIndex, ray) then
16:                 collision[tIndex].face ← cluster.face
17:                 collision[tIndex].modelComp_id ← face[cluster.face].modelComp_id
18:                 collided ← true
19:             end if
20:         else
21:             heap[index_current] ← cluster.index_1
22:             heap[index_current + 1] ← cluster.index_2
23:             index_current += 1
24:         end if
25:     end if
26:
27:     index_current −= 1
28: end while
29:
30: if collided then
31:     ray_orig ← collision[tIndex].intersPoint + ray_dir * ϵ
32:     ray_dir ← computeRayDirection(tIndex, ray_dir)
33:
34:     index_final ← atomicAdd(numCollisions, 1)
35:     triangleCollision[index_final] ← collision[index]
36: end if
37:
38: Return collided
```

Ilustración 175. Pseudocódigo de resolución de la intersección rayo-escena mediante un BVH.

Una vez identificada una colisión, es posible ajustar los parámetros del rayo para que este mismo pueda identificar otras colisiones. Bien podría utilizarse una distancia mínima para detectar colisiones más distantes, pero una alternativa mucho más elegante es modificar el punto de origen del rayo, de tal modo que pasa a ser el punto de colisión. Para evitar que este mismo rayo pueda detectar de nuevo la superficie que acaba de colisionar, se añade un pequeño *offset* pequeño dado por un factor ε. Por esta razón, el nuevo origen no es el punto de intersección, sino $p + ray_{dir} * \varepsilon$. En este problema de múltiples retornos no se modifica la dirección de un rayo, pero sí podría ocurrir ante cualquier otro evento. Por ello, se abstrae esta decisión en una función `computeRayDirection`.

Un problema que podemos encontrar con nuestros modelos de árboles es que no permiten comprobar fácilmente el filtrado de puntos para hallar el suelo del terreno. Como bien se indicaba en el capítulo de Generación procedural de bosque, la reducida complejidad de los modelos procedía de la definición de las hojas como planos, por lo que la representación de las mismas dependía de una textura *alpha*. Sin embargo, el escaneo no se resuelve de manera independiente para árboles y resto de la escena; sus vértices y triángulos conforman el BVH junto a los del resto de modelos. Por tanto, podríamos encontrar que los múltiples retornos no consiguen descubrir el suelo, sino que colisionan, una y otra vez, sobre las hojas.

Una solución es la diferenciación de esta superficie mediante la activación de un bit de `surfaceType` (*MeshGPUData*), y su correspondiente máscara, lo cual introduce una nueva rama en el flujo del programa (nunca es aconsejable en un *shader*), y también la necesidad de consultar una textura *alpha*. Otra manera de solventar este problema es el descarte aleatorio de puntos de colisión asociados a la vegetación del árbol. Desde el punto de vista de la precisión, que es el principio que ha motivado la ejecución de este trabajo tal y como se describe, introduce un error evidente. A nivel visual, el error no es fácilmente visible, y a nivel de rendimiento, no representa una carga de trabajo mucho mayor. Por esta razón, esta ha sido la solución finalmente adoptada.

Dicho descarte de colisiones sólo actúa en aquellos modelos que definen una opacidad menor que uno, por lo que dicha opacidad actúa como umbral frente a un valor aleatorio, obtenido mediante las coordenadas XY del primer vértice del triángulo, y la función `rand` utilizada anteriormente. El descarte no se produce una vez se verifica la colisión, sino en el comienzo de la función que resuelve la intersección rayo-triángulo. De esta manera, nos permite reducir la carga de trabajada derivada del algoritmo de Möller-Trumbore, aunque es cierto que abre una nueva rama mediante una estructura condicional. La opacidad de los modelos de la escena es ajustable mediante la interfaz, y por ende, también lo es el proceso de descarte de puntos (Ilustración 180). Esto implica que este comportamiento se puede anular siempre que se desee.

A nivel de *rendering*, nos interesa destacar que existe un método especializado para la visualización de los retornos, donde cada valor recibe un color diferente. Para ello se define una textura con hasta diez píxeles (y colores diferentes). Esto debería ser suficiente, teniendo en cuenta que el número máximo de retornos que define LAS

se encuentra en 5. Se habilitan dos maneras de filtrar los puntos por su valor de retorno:

- En función únicamente del valor original. Se establece así un umbral, r, para que sólo se *rendericen* aquellos puntos que verifican $r_i \geq r$, sea r_i su valor de retorno.
- A partir de un valor normalizado en $[0, 1]$. Dado r_i y n_i, donde n_i es el número de retornos de una emisión LiDAR, es posible calcular r_i/n_i. Es importante comprender por qué el filtro anterior no es suficiente para descartar la cubierta de un bosque. No todas las emisiones producen un mismo número de colisiones: escoger un umbral r aumenta las posibilidades de que sólo se *renderice* terreno, pero sólo se mostraría un subconjunto de puntos pertenecientes a dicha superficie. Sin embargo, si filtramos por $r_{norm} \in [0,1]$, y escogemos un valor alto, estaremos aproximándonos a la visualización de absolutamente todos los puntos capturados en el terreno.

A continuación, se muestran algunas imágenes que pretenden mostrar todo lo aquí descrito (Ilustración 177, Ilustración 178, Ilustración 179 y Ilustración 179).

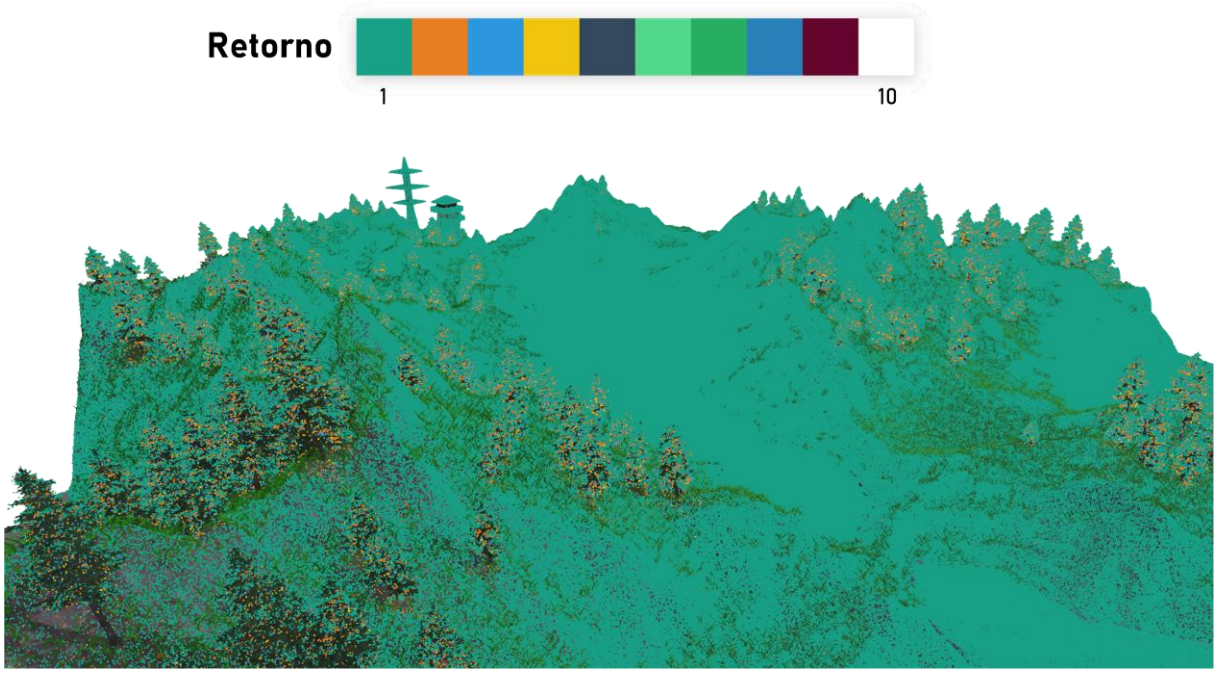


Ilustración 176. Rendering de todos los retornos capturados en un terreno.

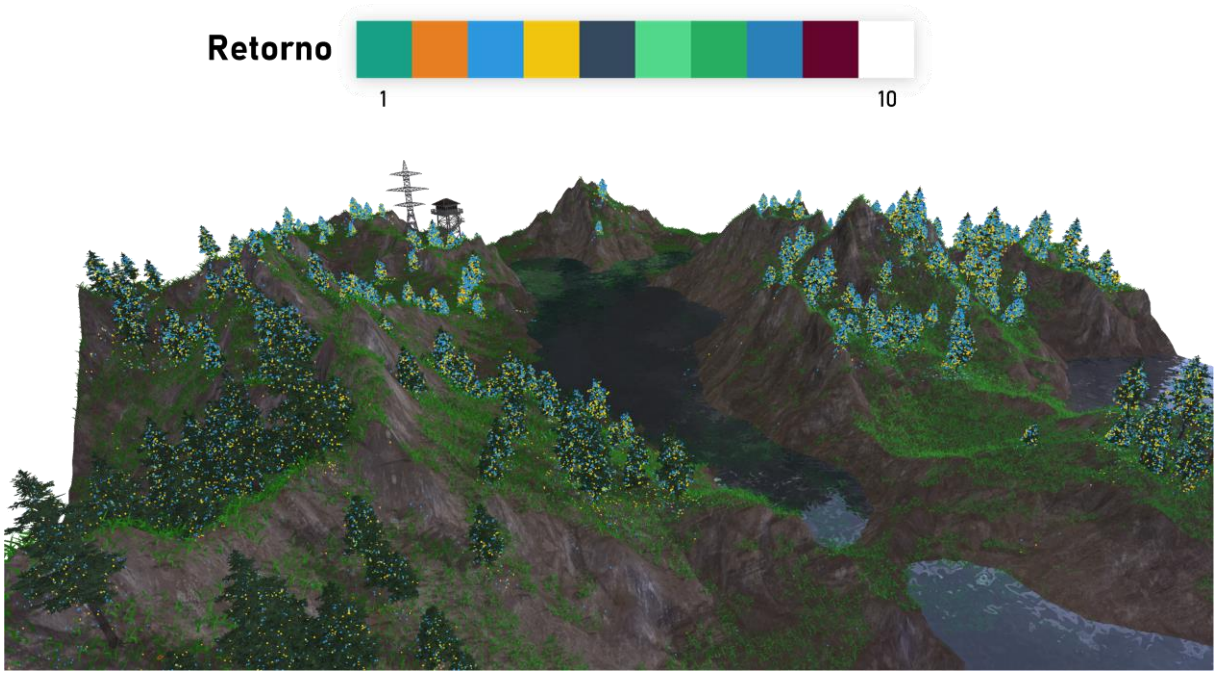


Ilustración 177. Filtrado de retornos por valor absoluto.

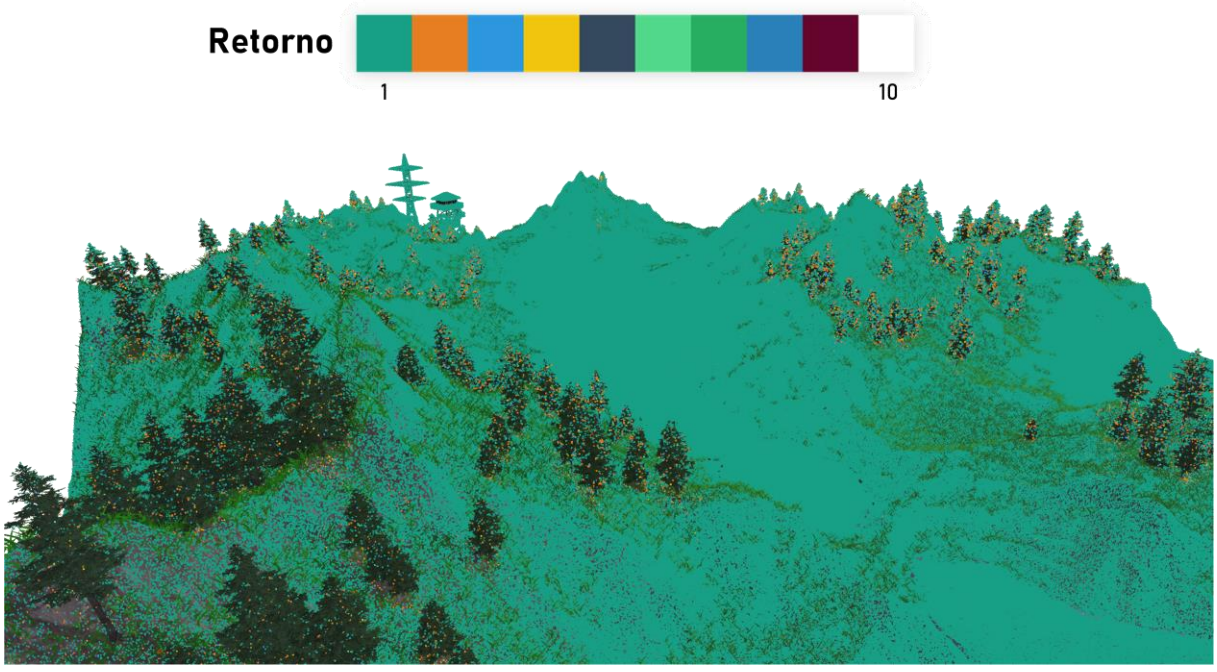


Ilustración 178. Filtrado de retornos por valor normalizado.

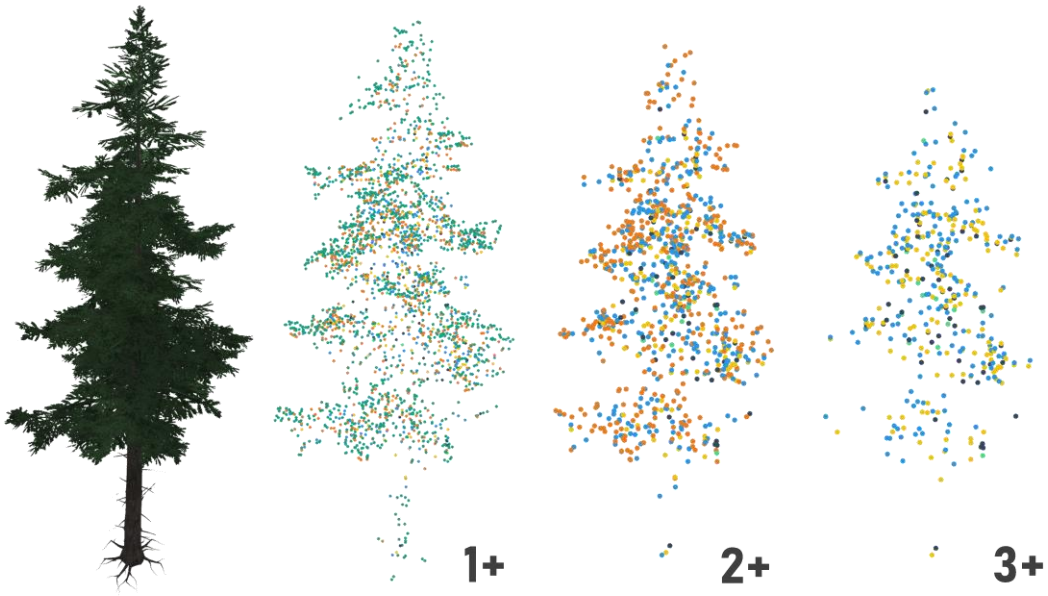


Ilustración 179. Filtrado de retornos sobre un único árbol.

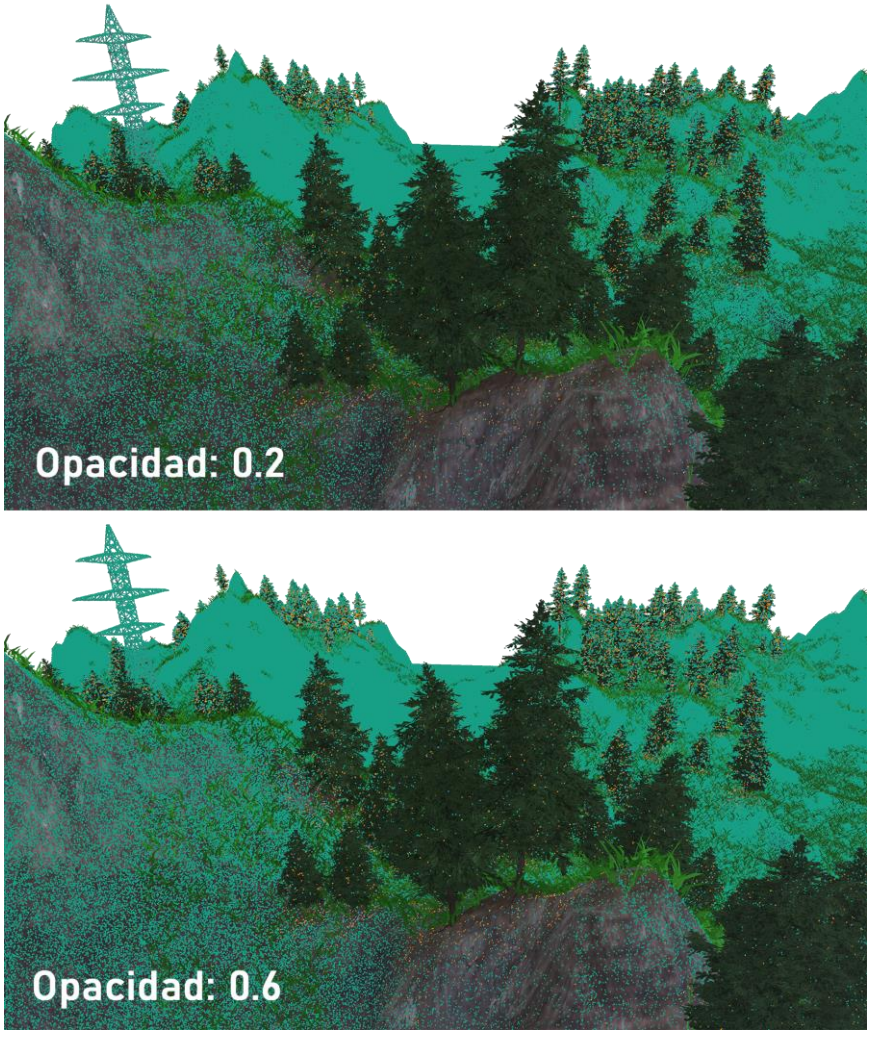


Ilustración 180. Modificación de la opacidad de las hojas de los árboles. En la segunda imagen se muestra una mayor densidad de puntos, procedente de vegetación no descartada.

Un último desarrollo que afecta a los múltiples retornos se encuentra relacionado con la probabilidad de que existan más de r retornos. Ciertamente, las superficies que permiten la continuidad de un pulso LiDAR son muy reducidas (vegetación), pero incluso en este caso, es necesario comprender que no todos los pulsos láser que colisionan con la vegetación pueden generar tantos retornos como indique el usuario (antes de colisionar con una superficie que no se preste a este comportamiento, como el terreno).

De la misma manera que es posible seleccionar el máximo número de retornos, también se habilita la selección de una probabilidad de continuidad para cada valor de retorno. Dados r retornos, se habilitan r *sliders*, que permitirán escoger dichos valores de probabilidad. La situación más normal es que este valor sea decreciente a medida que aumenta r_i, pero hay suficiente libertad para escoger el valor que se desee. El principal problema que podría plantearse es la especificación de estos valores en el *compute shader*. El método más simple consiste en la elaboración de una textura, con tantos píxeles como colisiones puedan existir, a la cual se accede mediante las coordenadas de textura $\left(\frac{r_i}{R_{max}}, 0.5\right)$ si consideramos que esta se extiende en u, y no en v (Ilustración 181). Se hace distinción de R_{max} y r_{max} debido a que ambos valores pueden, y suelen ser diferentes. La aplicación permite seleccionar hasta 10 retornos (R_{max}), pero el usuario puede escoger un valor máximo o igual o inferior a dicha constante (r_{max}).

Ilustración 181. Textura de probabilidad empleada al considerar la continuidad de los retornos.

Evitamos introducir carga de trabajo innecesaria comprobando si cierto valor aleatorio es mayor que el valor $texture\left(texReturnProb, \left(\frac{r_i}{R_{max}}, 0.5\right)\right)_r$ antes de comenzar a recorrer el BVH. Si el valor obtenido no verifica dicha condición, no existe una colisión y no será posible continuar bajo ningún concepto (`continueRay` se actualiza mediante `&=`, por lo que con un valor *false* no se continúa iterando). En la Ilustración 182 se muestran las diferencias obtenidas cuando se altera la probabilidad de que haya un segundo rayo.

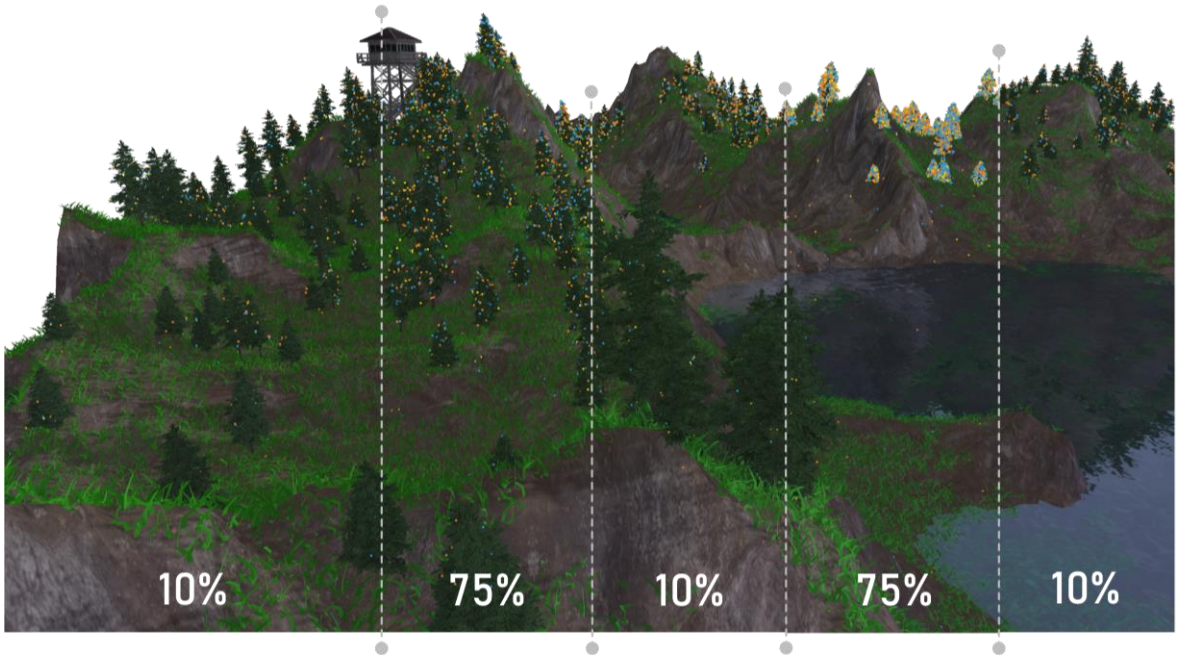


Ilustración 182. Nubes de puntos obtenidas con diferentes valores de probabilidad en la continuidad del segundo retorno.

Las consecuencias de la introducción de múltiples retornos en la simulación, a nivel de implementación, son elevadas. Cuando resolvíamos el escaneo en *Group3D*, instanciábamos *buffers* de tamaño $\min(RAYS_{iteration}, rays_{num})$, donde $RAYS_{iteration}$ era un umbral de tamaño que identificábamos en nuestro equipo de desarrollo, y que se sitúaba en torno a los 10 millones de rayos. Dicho umbral evitaba aumentar en gran medida la memoria asociada al proceso, dado que los escenarios planteados tienen una elevada complejidad, por lo que ya de por sí requieren un espacio importante. Por tanto, podíamos recuperar hasta 10 millones de colisiones. El problema de disponer de varios retornos es que este umbral se puede superar aunque el número de rayos emitidos en cada iteración no supere $RAYS_{iteration}$. Esto no implica que ahora debamos aumentar el valor de $RAYS_{iteration}$, sino que debe reducirse el número de rayos que lanzamos en cada iteración. Para un escenario donde son posibles hasta 5 colisiones, y con un umbral de 10 millones de rayos, sólo podríamos emitir 2 millones en cada iteración. Por tanto, necesitaremos un mayor número de escaneos que en anteriores ocasiones, a menos que el usuario indique que sólo desea un retorno.

El tamaño del buffer sigue siendo $RAYS_{iteration}$, mientras que el número máximo de rayos por iteración vendrá dado por $floor\left(\frac{RAYS_{iteration}}{retornos_{num}}\right)$. La repercursión de la utilización de varias ejecuciones del escaneo, en lugar de una o dos únicamente, se medirá en el apartado final de pruebas.

2.8.2.2 LiDAR batimétrico y agua

El LiDAR hasta ahora desarrollado únicamente trata de capturar la superficie del escenario, excepto en casos tales como la vegetación, donde es capaz de penetrar. Sin embargo, esto no sucede en el agua con un LiDAR básico. Al tipo de sensor LiDAR que es capaz de penetrar en el agua se le conoce como batimétrico, y para ciertas aplicaciones puede ser mucho más interesante, debido a que permite recuperar la superficie debajo del agua. Esto no implica que los puntos vinculados a agua sean totalmente irrelevantes; por ejemplo, (Korzeniowska 2012) emplea esta información para generar modelos de elevación digital (DEM) de la superficie del agua. Sin embargo, buena parte de los esfuerzos de investigación están más relacionados con la diferenciación de agua en un escenario, y frecuentemente, con su eliminación del mismo (Morsy, Shaker, & El-Rabbany, 2018)(Yates et al. 2008). Como sucede con los últimos retornos de las emisiones LiDAR en tierra, hallar una superficie no visible supone disponer de información de gran valor, dado que permite realizar hallazgos, monitorizar un entorno (Legleiter et al. 2016; Mandlburger et al. 2015), etc.

Al tratarse de una simulación, y no de un dispositivo real, podemos habilitar la introducción de ambos comportamientos en nuestra aplicación. Para ello, se incluye una opción en la configuración general de LiDAR, donde se puede seleccionar si el sensor que debe emplearse es batimétrico o no. En el caso de que así sea, se captura la superficie del terreno en lugar del agua. Si no es así, se captura la superficie de agua.

Dada una colisión, conocemos el identificador del componente al que pertenece, y por tanto, el tipo de superficie, a la cual podemos aplicar una máscara para conocer si se corresponde con agua. Se introduce así una nueva variable en el BVH *traversal*, `isCollisionValid`. Puede existir una colisión, pero esta podría no ser válida. Un LiDAR batimétrico puede impactar con el agua, pero no es esta colisión la que nos interesa, sino la que se produce con el terreno bajo el agua. Por tanto, se produciría una primera colisión, pero esta se ignora y el rayo continúa, de tal modo que es posible generar una segunda colisión. La dirección que adopta el rayo tras esta colisión se modela como la refracción de la dirección del rayo respecto de la normal del agua, aplicando el coeficiente de refracción asociada a dicho material (1.33).

La condición de iteración durante diversos retornos, para un mismo rayo, se define como sigue:

$$continueRay = continueRay \wedge \Big(\big(surface \,\&\, mask_{vegetation} \big) \vee \big(bathymetric \wedge (surface \,\&\, mask_{water}) \big) \Big) \wedge r_{current} < r_{max} \quad (2.91)$$

Del mismo modo, sólo se considera que una colisión es válida cuando se verifica $collided \wedge \big(\overline{bathymetric} \vee \overline{surface \,\&\, mask_{water}} \big)$. Por tanto, el valor $r_{current}$ sólo se incrementa cuando la colisión es válida ($r_{current} = r_{current} + uint(isCollisionValid)$. En la Ilustración 183 y Ilustración 184 se muestran algunos resultados de la implementación realizado en este apartado.

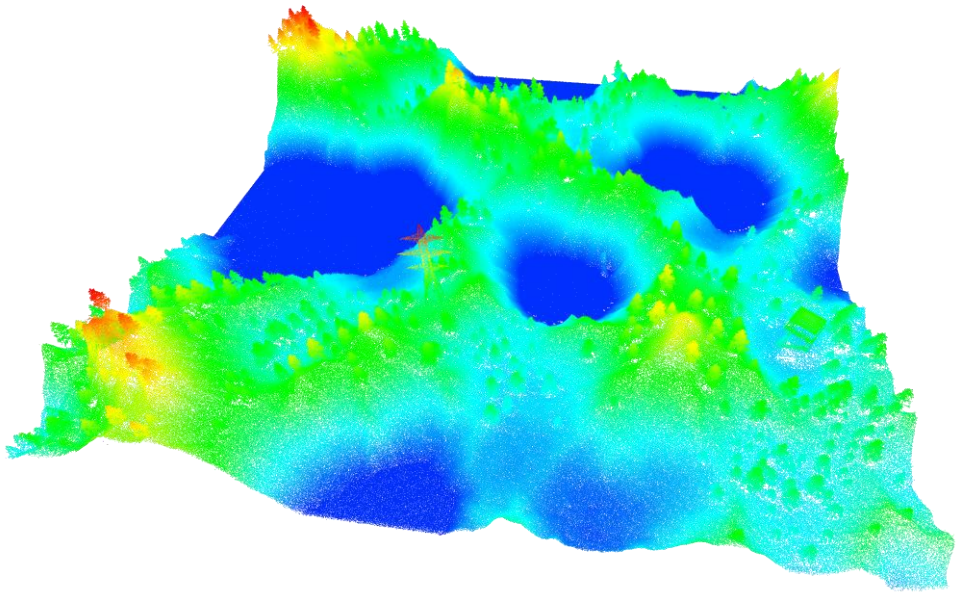


Ilustración 183. Resultado obtenido a partir de un LiDAR no batimétrico.

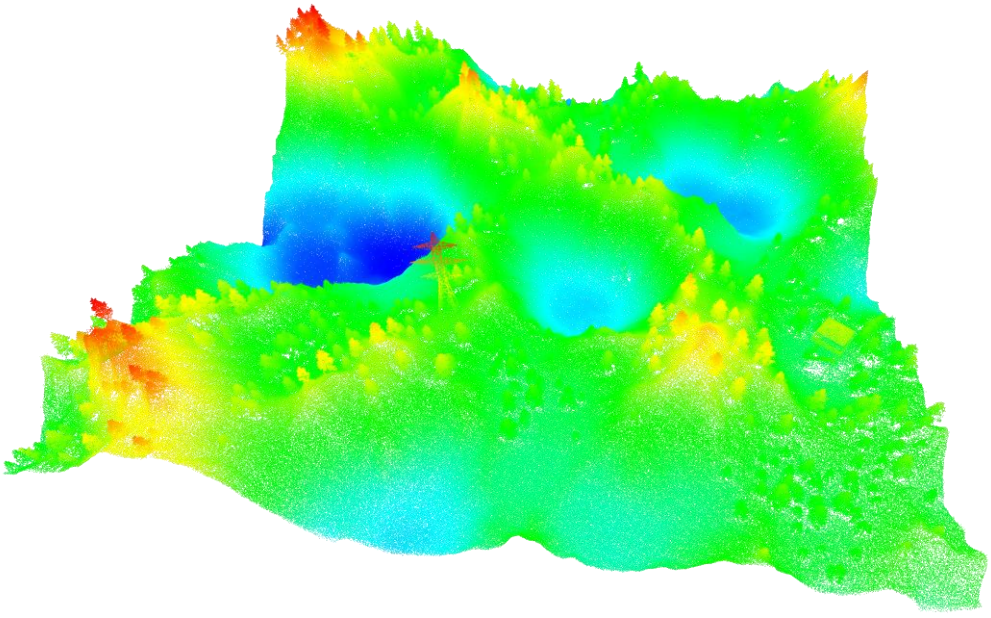


Ilustración 184. Resultado obtenido a partir de un LiDAR batimétrico.

2.8.2.3 Errores inducidos por un terreno

El error posicional inducido por un terreno se describe en (Deems, Painter, and Finnegan 2013). El primer error que se observa se encuentra en el eje Y, aunque no es una consecuencia de una incorrecta recepción de la altura, sino de un desplazamiento en XZ debido a la pendiente del terreno. Por tanto, se trata únicamente de una traslación que el usuario puede interpretar como un error en la captura de la coordenada Y de una colisión. La magnitud de este error se encuentra en el orden de $\frac{1}{1000} height_{lidar}$, sea $height_{lidar}$ la altura sobre el terreno.

Un segundo error también se produce a consecuencia de la pendiente, y lo podemos denominar '*time-walk*'. No es más que una propagación de un pulso LiDAR a lo largo de una pendiente, de manera que aumenta el tiempo de retorno y se percibe el punto más alejado de lo que realmente está. En la Ilustración 185 se puede observar como este error depende principalmente del ángulo de la pendiente, y por tanto, este será un factor importante en el cálculo del desplazamiento. En este caso, se determina que el error es aproximadamente 50 cm para $\alpha = 45°$ y $height_{lidar} = 1000\, m$. Sin embargo, se desconoce la aportación de ambos factores en el error inducido, por lo que se propone la introducción en la interfaz de un componente que permita seleccionar un factor multiplicador de error (si bien la dirección vendrá dada por el escenario planteado).

De acuerdo con la descripción del primer error, el punto trasladado puede aparecer encima o debajo de la superficie del escenario. Por tanto, la dirección de traslación no se considera dependiente de la normal del terreno, estableciéndose así que será un vector aleatorio que pueda construirse mediante la función `rand`.

Una vez se observa una colisión en el escenario, siendo esta válida, se somete al desplazamiento que se muestra a continuación:

$$axis_{horiz} = \left(rand\left(point_{intersection_{yx}}\right), 0, rand\left(point_{intersection_{xz}}\right)\right)_{norm}$$

$$strength = abs\left(ray_{startingPoint_y} - point_{intersection_y}\right) * factor_{horizontal} * rand\left(normal_{intersection_{xy}}\right) \quad (2.92)$$

$$isTerrain = surface \,\&\, mask_{terrain}$$

$$point_{intersection} += isTerrain * moveTerrain_{horizontal} * axis_{horiz} * strength$$

Nótese como $moveTerrain_{horizontal}$ será un valor flotante donde se almacena un factor multiplicador del error (por defecto, 1), y $factor_{horizontal}$ el valor dado por $\frac{1}{1000}$, que es el orden aproximado del error inducido. Este error, así como el siguiente, sólo se aplican cuando se verifica que la superficie intersectada es un terreno ($isTerrain = 1$). Por otro lado, un rayo puede producir varias colisiones, por lo que en cada una de ellas su posición origen se modifica. Por tanto, es necesario considerar un atributo $startingPoint$ que almacena la posición original del rayo, a partir de la cual se puede conocer la diferencia de altura entre el punto de colisión y el sensor LiDAR.

El segundo error descrito depende principalmente de la pendiente del terreno, el factor base aportado (aunque no se aplica a los valores que emplearemos), $factor_{vertical}$, que equivale a $\frac{0.5}{1000}$, y un factor multiplicador introducido por el usuario para ajustar la traslación vertical. Considerando estos parámetros, el segundo error se puede calcular como sigue:

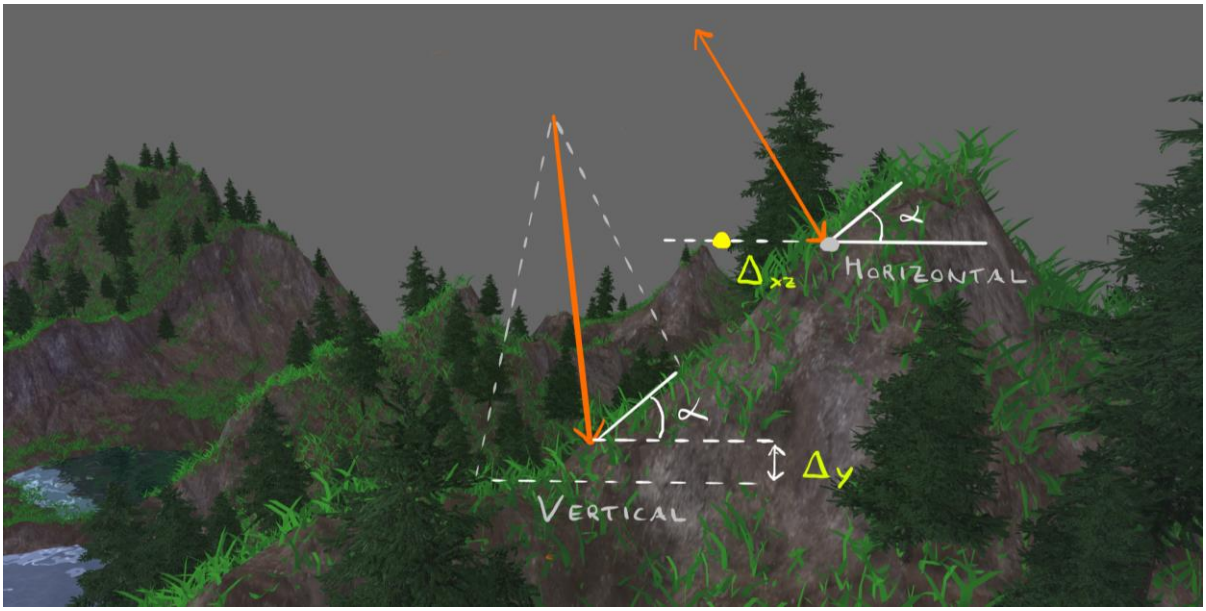


Ilustración 185. Boceto de errores inducidos por un terreno.

$$strength = \text{asin}\left(clamp\left(\frac{normal_{intersection_y}}{length(normal_{intersection})}, -1, 1\right)\right)$$

$$isTerrain = surface \,\&\, mask_{terrain} \tag{2.93}$$

$$point_{intersection} = point_{intersection} + isTerrain * moveTerrain_{vertically} * ray_{dir} * strength * factor_{vertical}$$

Un ejemplo de la traslación inducida por los dos errores descritos se muestra en la Ilustración 186:

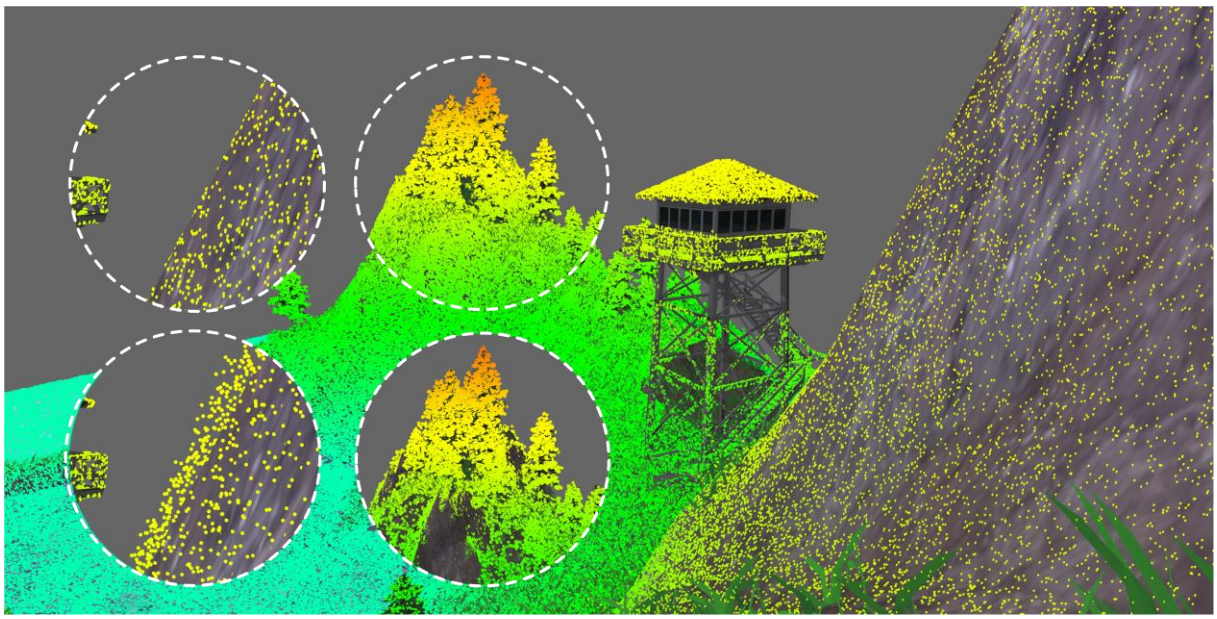


Ilustración 186. Detalle de traslación procedente de aquellos errores inducidos por el terreno. Error vertical (arriba) y ambos errores (detalles inferiores).

2.8.2.4 Superficies brillantes

Una de las superficies que más problemas genera en un escaneo LiDAR es la que se denomina como *mirror-like*; superficies brillantes tales como ventanas, parabrisas, fachadas de materiales brillantes, etc (Ullrich and Pfennigbauer 2019). El problema derivado del impacto de un rayo sobre una superficie como esta es la cadena de reflejos que se produce, lo que provoca en última instancia una dilatación del tiempo de retorno hacia el receptor del sensor LiDAR, haciendo así que el punto de colisión se perciba como más distante de lo que realmente se encuentra. En cualquier caso, el error generado se puede modelar mediante ray_{dir} y un factor de brillo de la superficie (para cada componente de la escena). A partir de estos valores se puede obtener una versión muy básica, pero válida. Sin embargo, un problema evidente es la uniformidad del error resultante, dado que todos los puntos colisionados de una misma superficie se desplazarán en la misma medida. Debido a esto que aquí se expone, se añade un factor de desplazamiento aleatorio, calculado a partir de $point_{intersection_{xy}}$, que puede reducirse mediante su división por alguna constante. El factor de desplazamiento de un componente completo podría venir dado por la distancia al sensor LiDAR, pero cuando se aplica de esta manera, es fácilmente

observable que cada punto de colisión tendrá una distancia diferente, lo que provoca, en última instancia, una distorsión de la estructura del componente, algo que no es necesariamente lo que se busca. Por tanto, se prefiere modelar la traslación de un único componente a partir de un valor aleatorio, asociado a su identificador (y por tanto, compartido por todas sus intersecciones), y un peso dado por el usuario para dicho desplazamiento aleatorio. Por último, también se permite indicar en la interfaz un desplamiento base que se aplicará a todas las colisiones de la escena.

De esta manera, lo ideal es fijar una traslación base para todos los puntos, y a partir de aquí, modelar cada componente mediante su brillo, así como un factor de traslación de modelos (se especifica para todos, pero se aplica sobre un valor aleatorio diferente para cada modelo). Aquellas superficies que desean simular este error deben almacenar un valor de brillo mayor que cero. Por ejemplo, en la primera escena disponemos de un modelo reflectivo (`Monkey`) sobre el cual simularemos este comportamiento (Ilustración 187).

Ilustración 187. Modelo reflectivo y simulación de escaneo sobre dicho objeto.

2.8.2.5 Valores de intensidad

Además de los procesos antes descritos, es posible calcular nueva información derivada del proceso de escaneo. Uno de los datos más valiosos de una nube de puntos de LiDAR es la intensidad, un valor que mide la reflectancia recibida de la superficie de un objeto (Jeong and Kim 2018). Es importante resaltar el término “recibido”, dado que el valor de intensidad no depende únicamente de las

características de la superficie, sino también de al menos otros dos factores: el ángulo de incidencia del pulso láser, y la distancia al sensor.

La distancia se puede obtener fácilmente en el escaneo. Partiendo de un rayo, se obtiene el valor paramétrico t donde se produce la colisión con el triángulo, y dicho valor es equivalente a la distancia siempre y cuando la dirección del rayo se encuentre normalizada. Sin embargo, con la introducción de múltiples retornos, donde el punto de origen se ve modificado, este enfoque no resuelve el problema planteado, dado que $orig$ no es $startingPoint$ en todos las situaciones. En cualquier caso, a partir del punto de partida, es posible calcular la distancia entre este y el punto de colisión. En el cálculo del ángulo de incidencia se debe emplear de nuevo el mismo punto de partida. Nótese como la intensidad depende del ángulo, pero no necesita explícitamente ese valor, sino alguna medida que permita darnos una idea de cómo es dicho ángulo. Por ejemplo, el producto escalar entre la normal del punto de colisión y $-ray_{dir}$ es idóneo para calcular la intensidad, dado que el producto escalar de un ángulo cero es 1, y por tanto, se reduce la intensidad lo máximo posible.

Dados estos dos valores, y la reflectancia de la superficie colisionada, se propone calcular la intensidad como se muestra a continuación:

$$intensity = clamp(reflectance - abs(dot_{product}) * angle_{weight} - distance * distance_{weight}, 0, 1) \quad (2.94)$$

El usuario podrá seleccionar dos valores, $angle_{weight}$ y $distance_{weight}$, para modelar la intensidad recuperada de la escena. La Ilustración 188 y la Ilustración 189 muestran los resultados obtenidos en nuestros dos primeros escenarios.

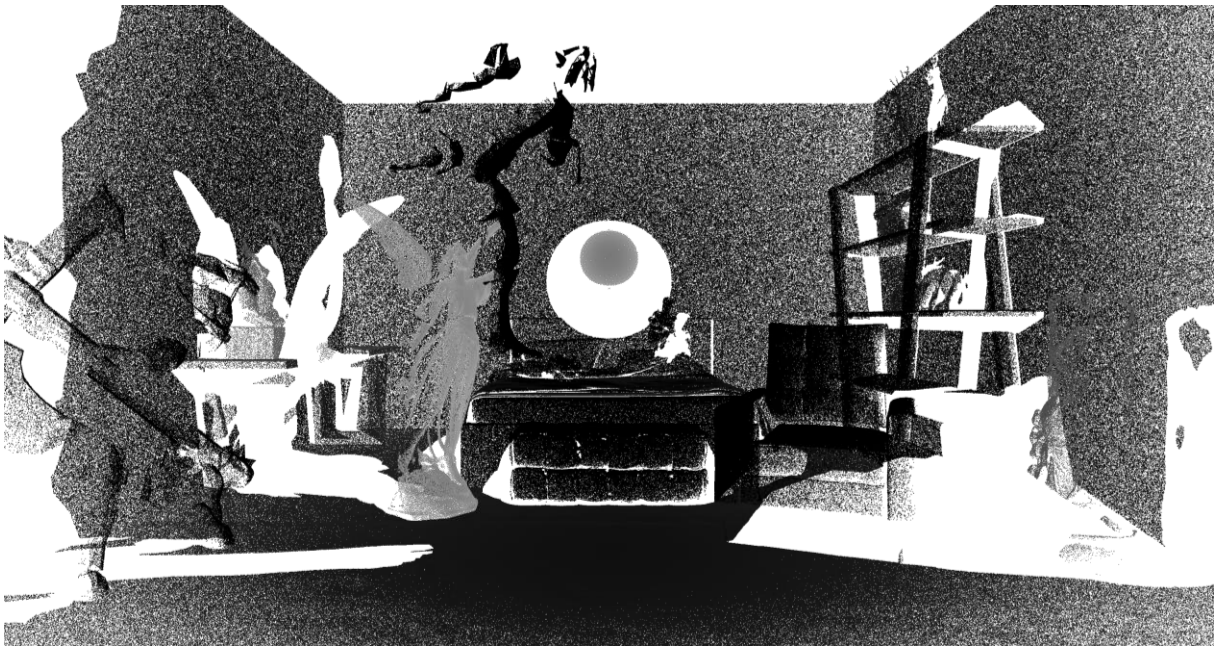


Ilustración 188. Valores de intensidad capturados en la primera escena.

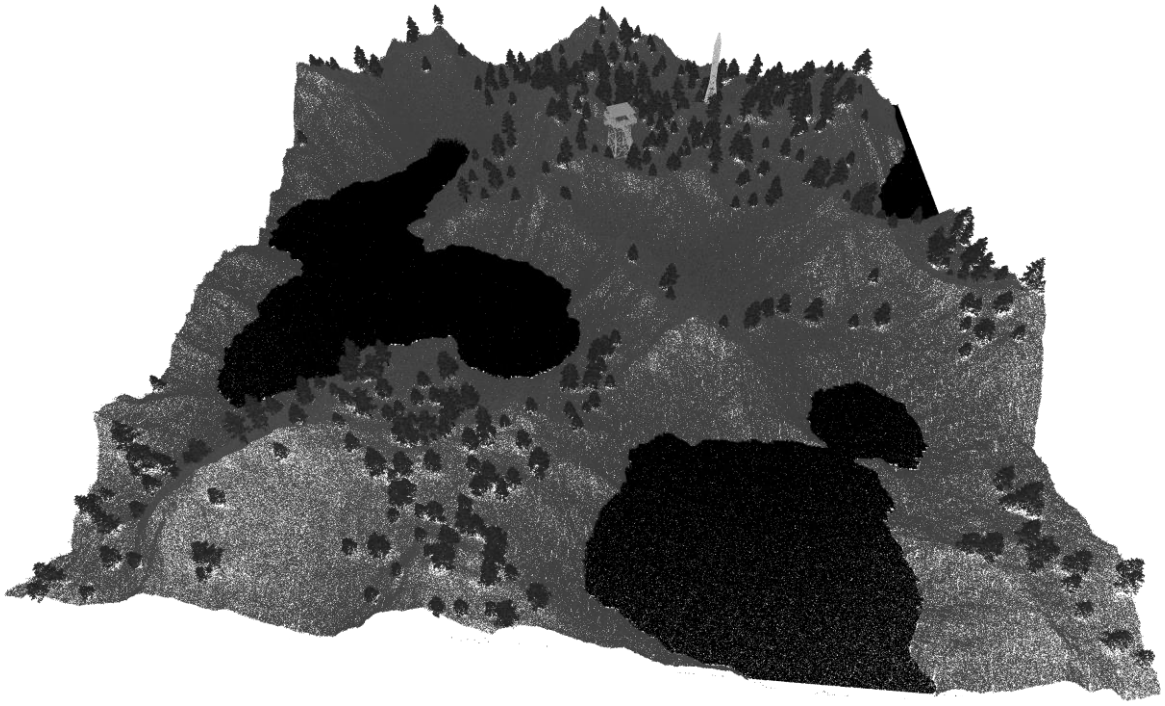


Ilustración 189. Valores de intensidad capturados sobre un terreno.

2.8.2.6 Alcance de rayos

Hasta este punto no hemos introducido ninguna restricción de alcance de los rayos, pero parece evidente que cualquier sensor debe incorporar restricciones de este tipo. El máximo rango de un pulso láser se puede calcular como sigue (Dong and Chen 2017):

$$R = \frac{1}{2} c * t_s$$
$$R_{max} = \frac{1}{2} c * t_{s_{max}} \quad (2.95)$$

Sea R la distancia a un objeto, c la velocidad de la luz (299792458 m/s), y t_s el tiempo que se mantiene activo un pulso láser (*travelling time*). Tratándose de un entorno de simulación, el cual debe ser lo suficientemente flexible para modelar cualquier sensor LiDAR, es importante introducir t_s como una variable más del sistema. $t_{s_{max}}$ será, en esta aplicación, el máximo valor seleccionable en la interfaz.

Por tanto, en la intersección rayo-triángulo se considera que existe una colisión cuando se verifique que $t > \varepsilon$ y $distance < \frac{1}{2} c * t_s$. Para evitar cierta carga de trabajo, es posible calcular directamente $range_{max} = \frac{1}{2} c * t_s$ fuera del *shader*, y especificarlo como un `uniform`. La especificación de un valor de t_s por el usuario no indica que todos los pulsos láser sean capaces de mantenerse activos durante todo este tiempo. Es decir, es posible introducir cierta función de ruido para evitar que la distancia máxima quede muy bien definida. Además, no tiene por que considerarse que $t_i < t_s$, podríamos modelar t_i en $[t_s - \rho, t_s + \rho]$, donde ρ vendrá dado por `rand`, que producirá un valor entre 0 y 1. Además, se puede reducir el peso ρ mediante alguna constante, con el fin de evitar que la discontinuidad del rango máximo sea excesiva.

Por ejemplo, la utilización de dos valores t_s diferentes produce el resultado que se muestra en la Ilustración 190.

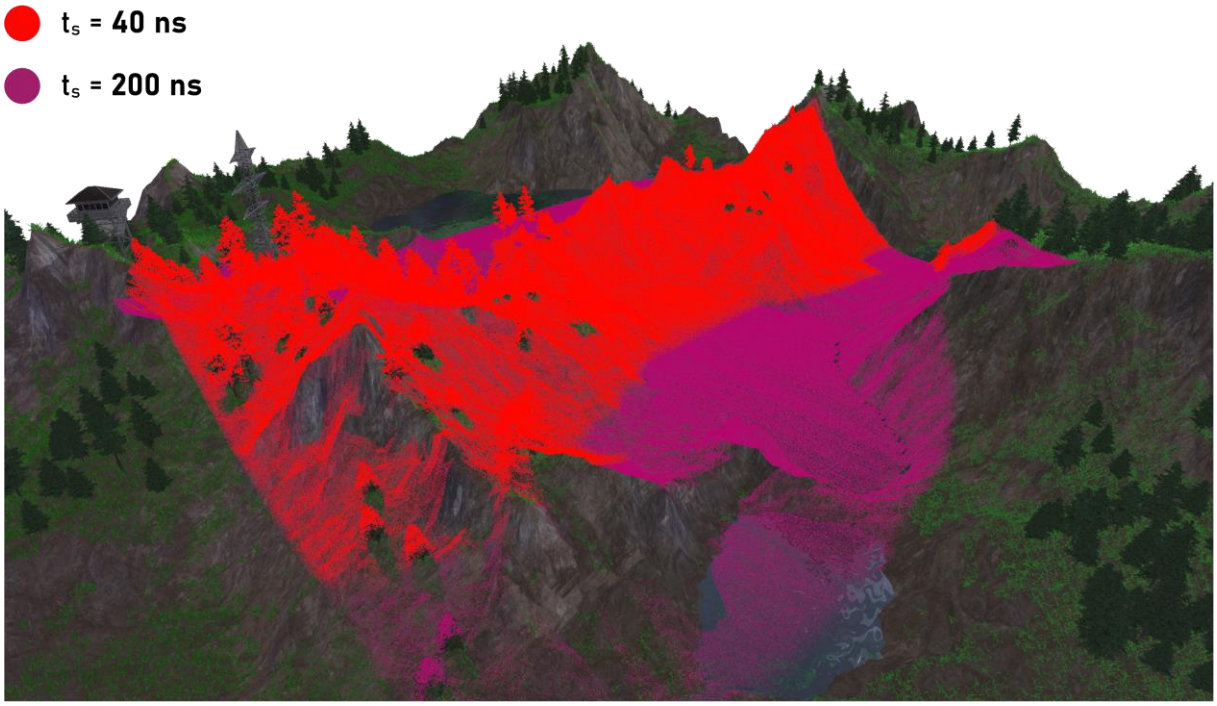


Ilustración 190. Comparación del rango alcanzado por dos valores diferentes del tiempo de actividad de un rayo.

2.8.2.7 Outliers

Un error más aleatorio es la introducción de valores anómalos en la nube de puntos. Este error no se produce como consecuencia de un determinado tipo de superficie, sino que puede tener orígenes múltiples: falsas alarmas del receptor, partículas en suspensión, fotones no suprimidos por el filtro paso de banda del receptor, etc (Ullrich and Pfennigbauer 2019). Un error como este podría incluirse tras la simulación, de tal manera que una vez capturado el escenario (de manera precisa), se pueden introducir muy pocos puntos que representen este ruido. Otra solución más eficiente es el descarte de colisiones producidas con modelos del escenario para convertirlas en puntos de ruido. Nótese como el número de puntos anómalos en una nube de puntos LiDAR no es elevado (es difícil observar un *clustering* de este tipo de puntos). Por tanto, parece razonable descartar únicamente algunas colisiones para simular este tipo de error.

Para representar un valor anómalo es necesario calcular la colisión con el escenario, lo cual nos permite situar dicho punto sobre el rayo emitido. Así, es posible distorsionar una intersección mediante la dirección $-ray_{dir}$ y un desplazamiento $displacement_{outlier}$, aunque dicha magnitud se multiplica igualmente por un valor aleatorio que evita que todos los puntos anómalos se desplacen con la misma longitud.

Una colisión pasará a convertirse en un punto anómalo cuando un valor aleatorio supere un umbral definido por el usuario. Se puede implementar esto mismo sin estructuras condicionales, tal y como se propone a continuación:

$$force = rand\left(point_{intersection_{xz}}\right) * displacement_{outlier}$$

$$isOutlier = sign(clamp(force - threshold_{outlier} * displacement_{outlier}, 0, 1)) \quad (2.96)$$

$$point_{intersection} += -ray_{dir} * magnitude * isOutlier$$

Sea $threshold_{outlier}$ un valor situado en $[0, 1]$, especificado por el usuario (por defecto, 1; no hay valores anómalos), entonces es posible comprobar si la colisión actual pasaría a ser un punto anómalo mediante el signo de un valor, aleatorio, que se traslada del intervalo $[0, displacement_{outlier}]$ a $[-threshold_{outlier} * displacement_{outlier}, displacement_{outlier}(1 - threshold_{outlier})]$. En otros términos, el segundo intervalo se puede expresar como $[-threshold_{outlier}, 1 - threshold_{outlier}]$, y aplicando una operación *clamp* sería $[0, 1 - threshold_{outlier}]$, de tal modo que sólo se obtiene el signo 1 (punto anómalo) cuando $rand\left(point_{intersection_{xz}}\right) > threshold_{outlier}$.

La Ilustración 191 y la Ilustración 192 muestran algunos resultados obtenidos a través de la introducción de valores anómalos con diferentes umbrales:

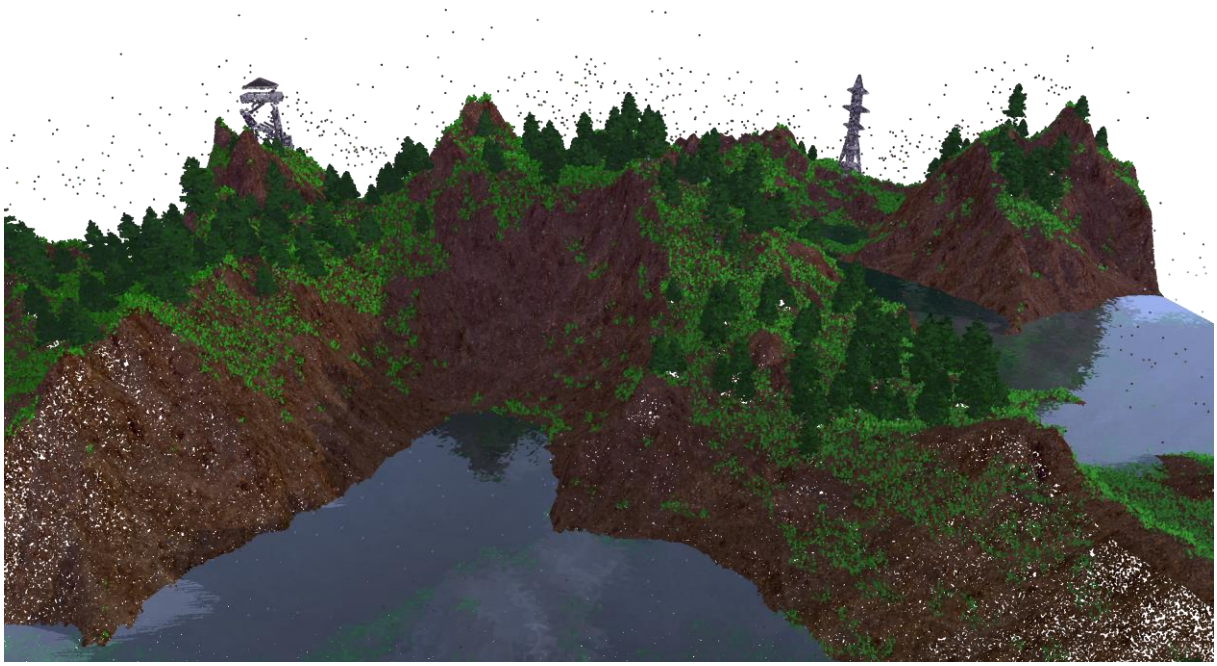


Ilustración 191. Puntos anómalos representados sobre un terreno, utilizando un umbral de 0.9987 y un desplazamiento máximo de 1.

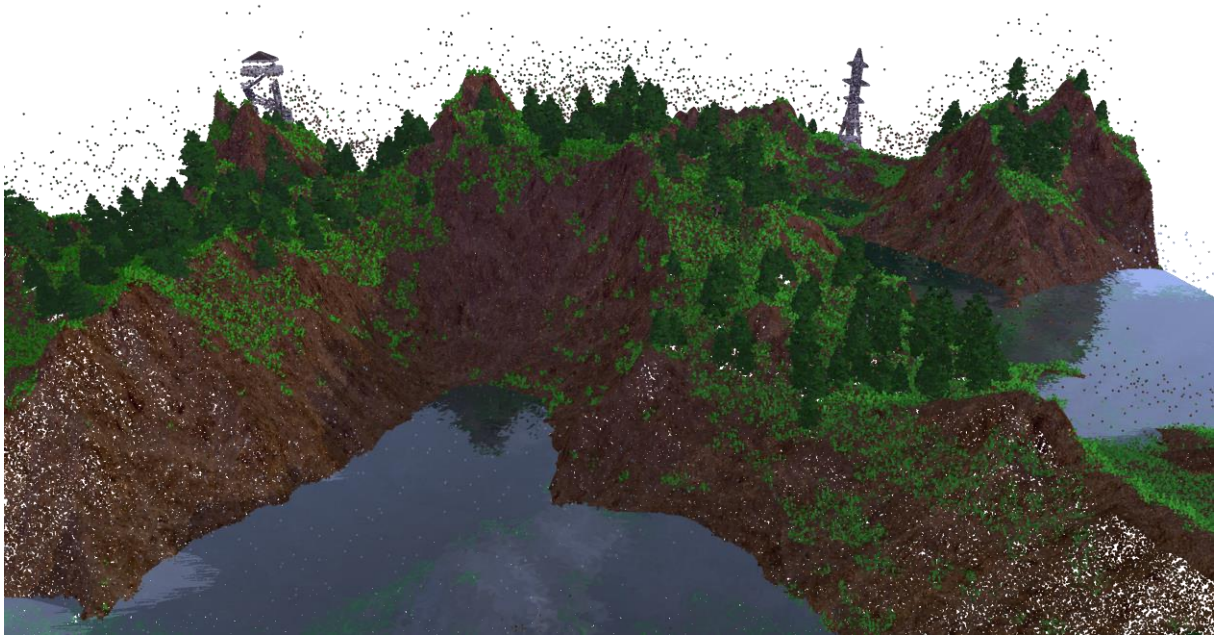


Ilustración 192. Puntos anómalos representados sobre un terreno, utilizando un umbral de 0.997 y un desplazamiento máximo de 1.55.

2.8.3 Modos de visualización adicionales

Aún en esta iteración nos quedan por describir algunos modos de visualización adicionales, todos relacionados con la asociación de modelos con algunos conceptos semánticos. En esta aplicación se distinguen al menos dos maneras de asociar un componente de un modelo con un concepto: mediante las clases que define la especificación LAS en su versión 1.4 (a lo que hemos denominado clase ASPRS), o mediante conceptos personalizados. La principal ventaja de los segundos respecto de los primeros es que no se encuentran acotados, y por tanto, es posible usar un nivel de detalle personalizado. Por ejemplo, podríamos hablar sólo de vegetación, o podríamos distinguir vegetación baja, alta, etc. Esta es una de las visualizaciones más importantes, dado que una finalidad de este trabajo es la generación de nubes de puntos etiquetadas que puedan servir para otros algoritmos relacionados con la inteligencia artificial. El principal problema que plantea la introducción de estos conceptos semánticos es la asignación de colores, de tal manera que estos sean lo suficientemente diferentes como para distinguir las clases en el *renderizado*.

Para conseguir esta distribución uniforme de colores podemos emplear lo que se conoce como *Golden ratio* (Ankerl, 2009), donde cada valor subsecuente divide un intervalo del espacio. La situación ideal sería que $i = 0$ partiera el espacio $[0, 1]$ en dos intervalos del mismo tamaño, y que los siguientes valores hicieran esto mismo

con alguno de los nuevos intervalos generados, volviendo a particionarlos equitativamente. En la práctica, no se generan particiones del mismo tamaño, pero se obtiene una solución lo suficientemente buena como para distinguir las clases en la mayoría de escenarios. Fijando los valores de saturación y valor (HSV) a 0.99, podemos considerar que el *hash* que se crea con cada nuevo valor de i determina el valor de tono (*hue*). Dado i, su *hash* se calcula como sigue:

$$hash = (i * \varphi) \,\%\, 1 \tag{2.97}$$

Donde φ es la constante *Golden ratio*, cuyo valor equivale a 0.618033988749895. A partir de aquí, sólo será necesario convertir el valor HSV en RGB, para poder emplearlo en el *shader*. Se mantiene así una estructura en Model3D, donde se asocia cada valor i con su correspondiente color RGB, aunque también es necesario mantener esta misma estructura para colores HSV, debido a que se han habilitado algunas ventanas en la interfaz donde es posible consultar a qué concepto responde cada color, y Dear ImGui trabaja sólo con valores en el espacio HSV.

Esto que aquí se describe responde al problema de la asignación de colores tanto para los conceptos personalizados, como para aquellos conceptos definidos en el estándar LAS 1.4. La principal diferencia es que los colores asociados a estos últimos conceptos se pueden calcular desde el comienzo de la aplicación, debido a que no varían y se conocen de antemano.

Un concepto no tiene por qué aplicarse sobre un componente, sino que también puede vincularse a un modelo cualquiera (puede contener varios componentes), o un grupo, de tal manera que el mismo concepto se aplica a todos los elementos que forman parte de él. Por ejemplo, en la Ilustración 193, la cama se registra como un grupo completo, al cual se asocia el concepto “Mueble”.

El resto de ilustraciones (Ilustración 194, Ilustración 195 e Ilustración 196) muestran la diferenciación semántica de los modelos de los dos primeros escenarios, tanto para conceptos propios como estándar.

Ilustración 193. Conceptos semánticos propios de la aplicación.

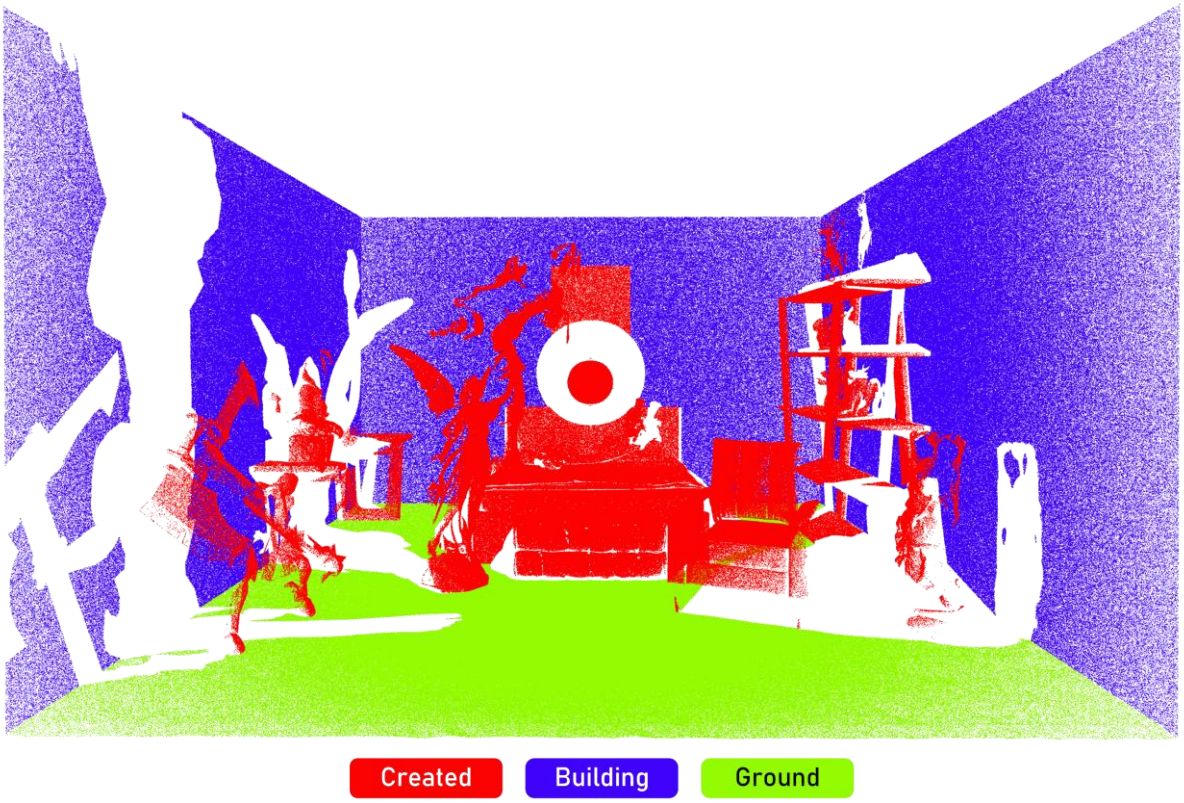


Ilustración 194. Conceptos semánticos definidos por ASPRS.

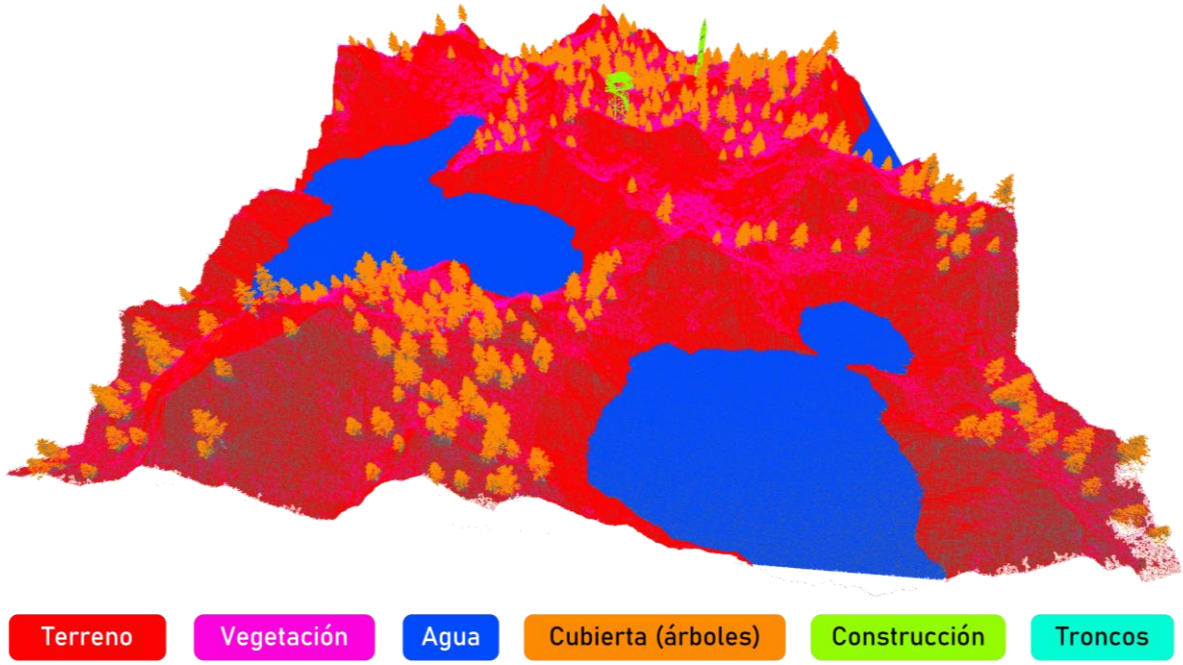


Ilustración 195. Conceptos semánticos propios de la aplicación.

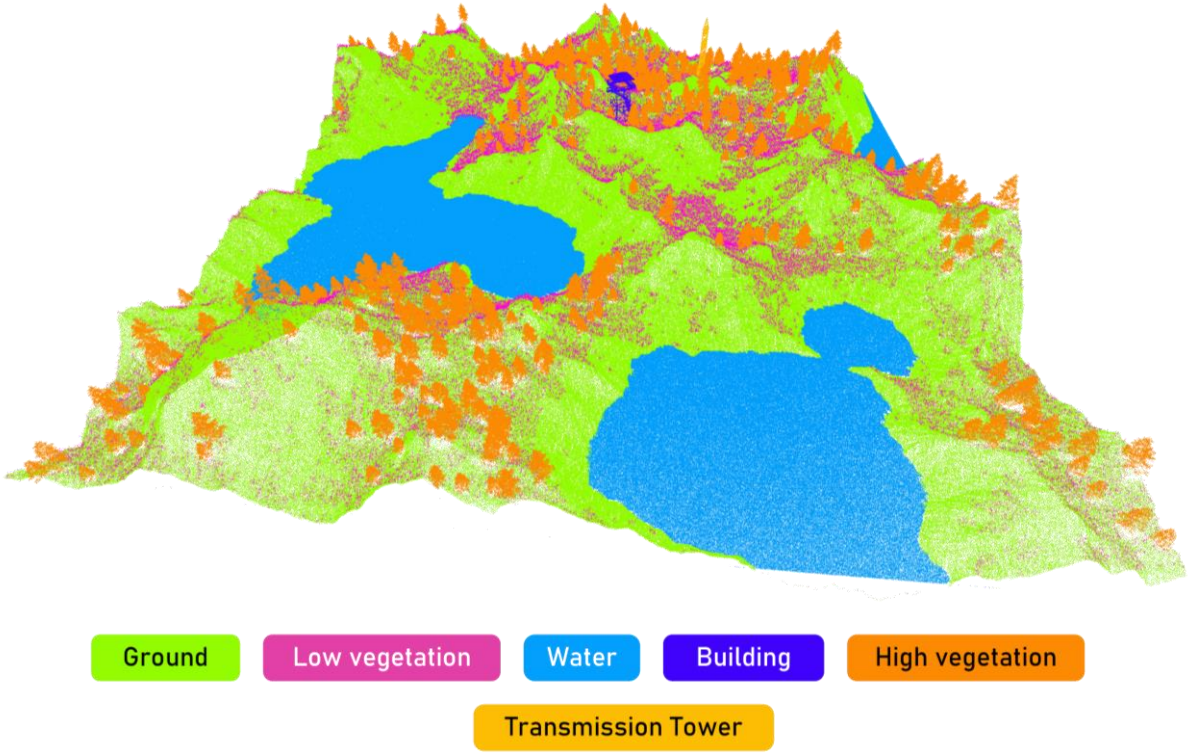


Ilustración 196. Conceptos semánticos definidos por ASPRS.

2.8.4 Almacenamiento de nubes de puntos

Dada la finalidad del trabajo, donde las nubes de puntos sintéticas deben servir como fuentes de datos en otras aplicaciones, parece necesario almacenar los resultados en un fichero, pudiendo considerarse varios formatos. La solución más

formal sería la utilización de un fichero LAS, en gran parte debido a que algunos de los conceptos introducidos siguen el estándar definido en su especificación. Para ello disponemos únicamente de una librería, denominada PDAL[25], la cual es la única que incorpora la especificación 1.4. Si bien se consigue su compilación en modo *Debug*, no sucede lo mismo en *Release*, por lo que debe descartarse su utilización.

Esto no es necesariamente una mala noticia, dado que una desventaja del uso de un fichero LAS es la necesidad de descartar los conceptos semánticos personalizados, dado que no tienen cabida en ninguno de los formatos de un registro (excepto en el caso de que ocupen un atributo dedicado a otra función). Como alternativa, se plantea el uso del formato PLY, donde podemos definir varios elementos, y cada elemento tendrá sus propios atributos (con total flexibilidad). Por tanto, un formato tan versátil parece beneficiarnos en nuestro propósito. Además, la creación de este tipo de fichero se puede llevar a cabo mediante librerías tan ligeras como *tinyply*[26]. En resumen, con el formato PLY se obtiene mayor flexibilidad, y su introducción es mucho más simple.

Para gestionar el almacenamiento de ficheros se habilita un nuevo submenú, *Point Cloud File*, donde se puede modificar el nombre del fichero destino, y se puede seleccionar si se desea, o no, almacenar la nube de puntos generada tras la simulación. No es posible guardar una nube en cualquier instante de la ejecución, sólo será posible tras la simulación (y después, se liberarán los datos). Para evitar introducir más carga en la propia simulación, el almacenamiento del fichero se realiza en *background* mediante un hilo, el cual se encarga de liberar la memoria una vez almacenadas las colisiones.

Para comprobar la corrección del fichero generado es posible emplear herramientas tales como *CloudCompare*, donde al abrirlo nos permitirá vincular algunos atributos básicos (posiciones, normales, color RGB, etc) con las propiedades personalizadas de nuestra nube. En nuestro caso, almacenaremos los siguientes campos:

- Posiciones, normales y coordenadas de textura.
- Conceptos semánticos y sus colores (estos últimos sólo para su visualización en una herramienta externa).

[25] https://pdal.io/
[26] https://github.com/ddiakopoulos/tinyply

- Valor de intensidad.
- Valor de retorno, número de retornos del pulso láser, y el resultado de la división de ambos valores, situado en el intervalo $[0, 1]$, para poder visualizarlo.

En la Ilustración 197 y la Ilustración 198 se pueden observar algunos resultados *renderizados* en la herramienta externa *CloudCompare*.

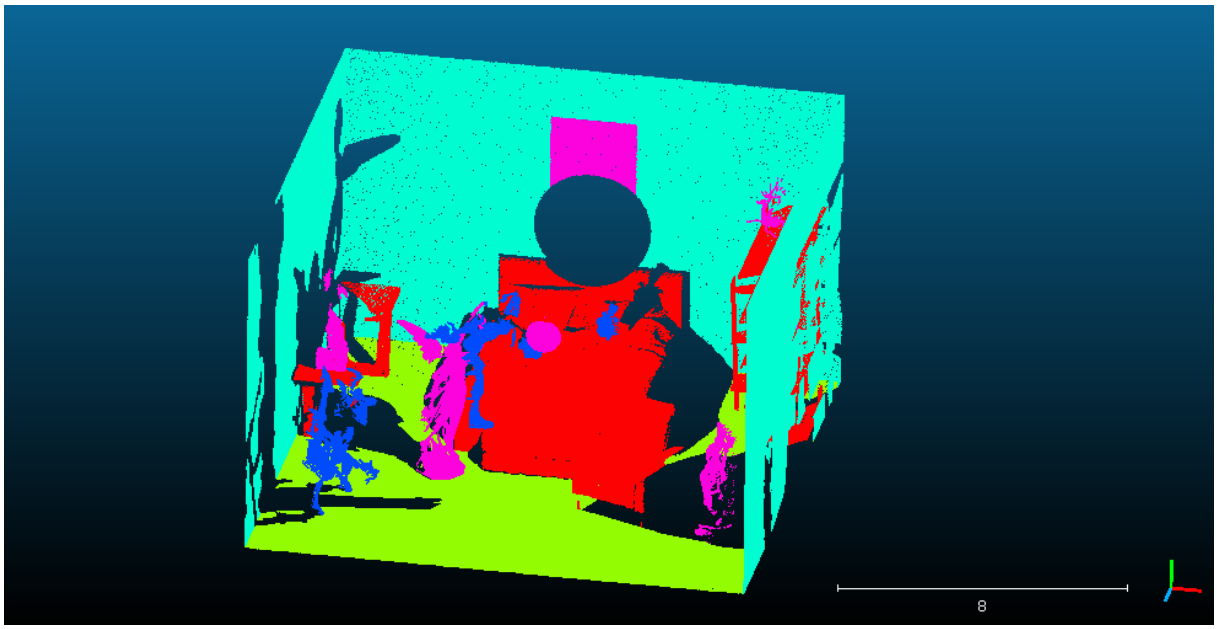


Ilustración 197. Rendering de conceptos semánticos personalizados del primer escenario en la herramienta CloudCompare.

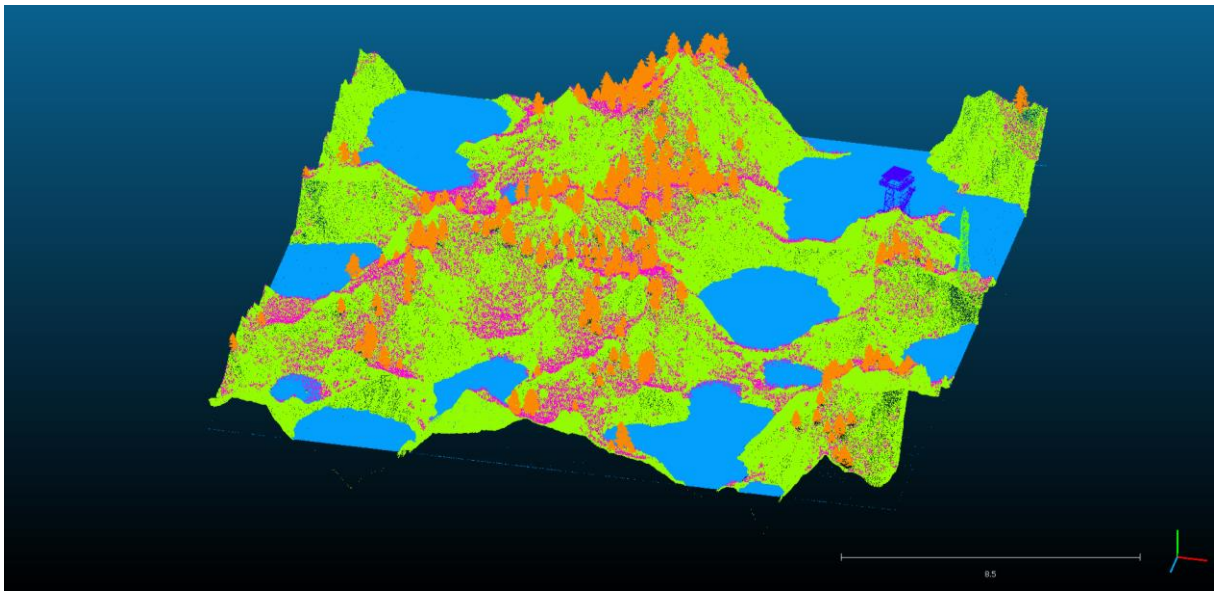


Ilustración 198. Rendering de conceptos semánticos ASPRS del segundo escenario en la herramienta CloudCompare.

2.8.5 Diagrama de clases

Para finalizar este apartado, se muestran aquellos cambios introducidos en la última iteración. En su mayoría, se encuentran vinculados con las mismas entidades que ya se modificaron en la iteración seis: *Model3D* y *LiDARSimulation*. Se añaden nuevos modos de *rendering*, y además, se incluyen y completan clases auxiliares de la simulación.

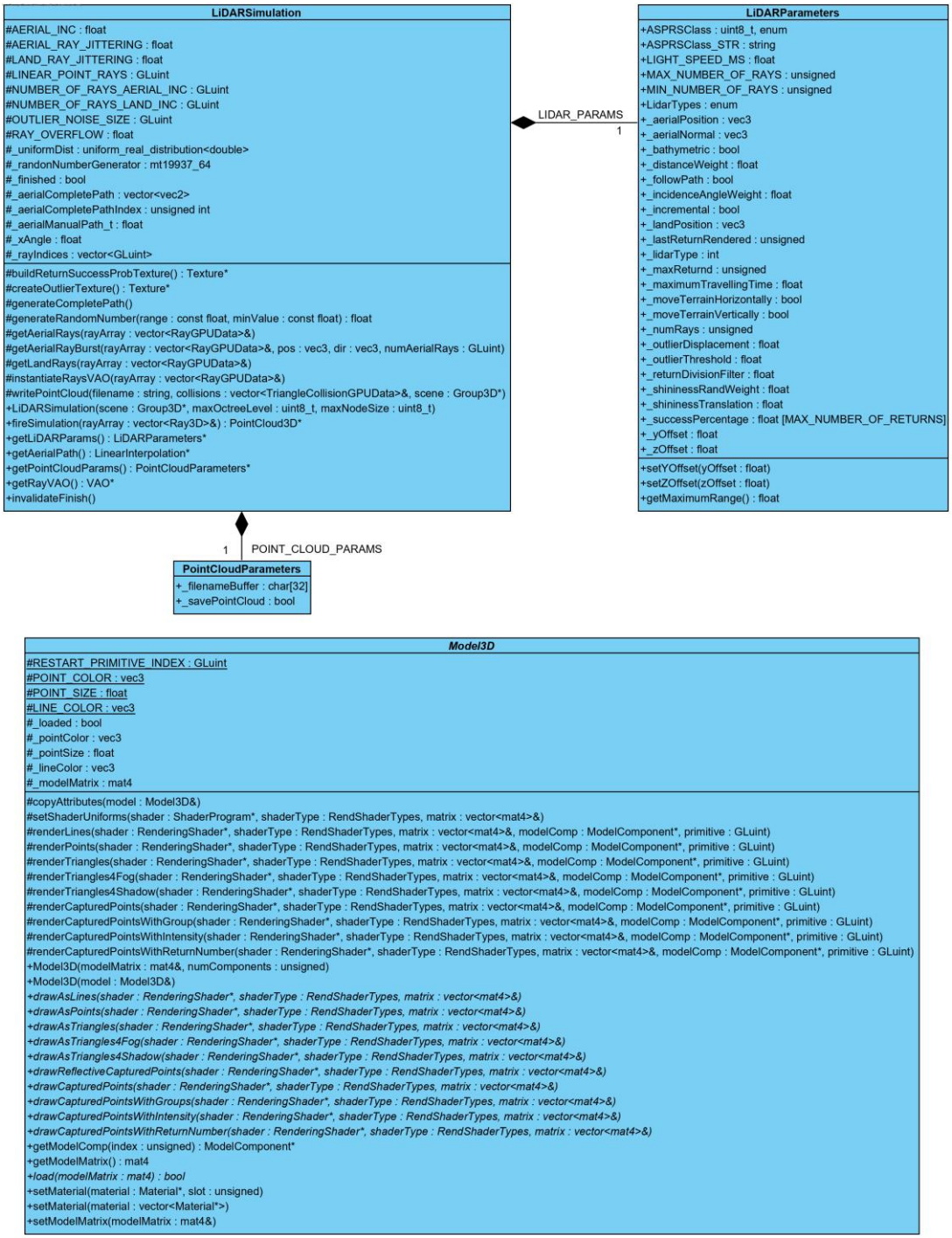


Ilustración 199. Diagrama de clases correspondiente a los cambios producidos en la iteración siete.

3 EXPERIMENTACIÓN, RESULTADOS Y DISCUSIÓN

En esta sección se documentan aquellas pruebas que se llevan a cabo sobre el sistema final. A diferencia de las que se han descrito a lo largo del desarrollo, las cuales se ejecutan sobre soluciones parciales, estas pretenden exponer el rendimiento del sistema, mientras que las anteriores nos ayudaban a justificar las decisiones tomadas a lo largo de las iteraciones. Por tanto, en este apartado nos centraremos en mostrar cómo se comporta el sistema ante diferentes escenarios y configuraciones, empleando para ello el equipo documentado en la sección Coste hardware.

3.1 Experimentaciones y pruebas

Este apartado se divide en tantas secciones como grupos de pruebas se identifican.

3.1.1 Rendimiento de escaneado

En la quinta iteración del desarrollo se mostraba el tiempo de respuesta de un proceso de escaneo incompleto, por lo que en este apartado no sólo se pretende documentar el tiempo medio del sistema final, sino también la variación de tiempo entre ambas versiones.

3.1.1.1 Rendimiento de sistema final

Las pruebas que aquí se realizan pretenden comprobar el tiempo medio de respuesta para los tres escenarios planteados, los cuales disponen necesariamente de un número diferente de vértices y triángulos. En el caso del escenario procedural (terreno), se fijará la semilla que determina la mayor o menor complejidad del mismo. En el tercer escenario, la captura de la escena podrá realizarse en cualquier *frame*, de tal manera que si hubiera alguna dependencia respecto de la estructura del sistema de partículas, esta pudiera reflejarse en el tiempo medio. El objetivo será extenuar el sistema, por lo que nos situaremos en el peor escenario posible, donde la continuidad de un rayo tras colisionar con una superficie no se ve afectada por un valor de probabilidad para cierto valor de retorno (en realidad, sí se verá afectado, pero un umbral de valor uno desactiva por completo el descarte de rayos). Así, el número de

colisiones resultantes se maximiza. De acuerdo con la tecnología actual, se establece que el número máximo de retornos será de 5.

También se proponen otros escenarios alternativos. En el segundo, decrece la probabilidad de contuinidad de un rayo a medida que aumenta el valor de retorno. Esta configuración emplea un valor de probabilidad $1 - 0.1 * i, i \in [0, 5)$, sea i el valor de retorno. Ambos escenarios se comprueban para 10 y 20 millones de rayos, obteniéndose hasta cuatro configuraciones diferentes. En la recuperación de tiempos de respuesta se mide 1) el tiempo de resolución del escaneo, incluyendo definición y lectura de *buffers*, y 2), únicamente el tiempo empleado en GPU.

No todos los escenarios necesitan aplicar las cuatro configuraciones; por ejemplo, el escenario de una habitación no tendrá superficies que permitan comprobar la continuidad de un rayo, por lo que la configuración 1 (10 millones de rayos y probabilidad de continuidad 1) es equivalente a la segunda, debido a que la variación de valores de probabilidad no tiene efectos sobre el resultado. Esto mismo ocurre para el tercer escenario (MPM). En la representación gráfica se considerará el mismo tiempo para ambas configuraciones.

Para compensar el menor número de configuraciones aplicadas en el primer y tercer escenario, se añade una quinta opción, donde se utilizan 40 millones de rayos (los valores de probabilidad no son relevantes, de nuevo). El segundo escenario se excluye de esta prueba, dado que el equipo de desarrollo no puede tolerar el elevado número de colisiones que se produce en estas condiciones. En cualquier caso, esto mismo que aquí se expone se muestra en la Tabla 47 y la Tabla 48, con el fin de aclarar qué parámetros y configuraciones se aplican a cada escenario.

Especificación de configuraciones					
	Configuración				
Parámetro	**1**	**2**	**3**	**4**	**5**
Número de rayos	10 M	10M	20M	20M	40M
Prob. de continuidad	1	$1 - \frac{i}{100}$	1	$1 - \frac{i}{100}$	1
Máx. número de retornos			5		
Batimétrico			✗		
Máx. tiempo de actividad			300ns		

$offset_{terrestre}$	0
$offset_{aéreo}$	$\pi/4$

Tabla 47. Especificación de parámetros de cada una de las pruebas que deben ejecutarse.

Aplicación de configuraciones en escenarios					
	Configuración				
Escenario	**1**	**2**	**3**	**4**	**5**
1 - Habitación	✓	✗	✓	✗	✓
2 - Terreno	✓	✓	✓	✓	✗
3 - MPM	✓	✗	✓	✗	✓

Tabla 48. Utilización de configuraciones para los tres escenarios de la aplicación.

Aplicando las pruebas en los diferentes escenarios con los parámetros documentados, se obtienen los resultados que se documentan en la Tabla 49, la Tabla 50 y la Tabla 51.

Escenario #1 **3.350.433 vértices y 6.683.902 triángulos**			
Configuración LiDAR #1			
Ejecución	**Tiempo de respuesta (ms)**	**Tiempo de respuesta de kernel (ms)**	
1	2.014	826,193	
2	2.460	789,308	
3	1.936	795,966	
4	1.962	783,629	
5	1.979	783,403	
	Tiempo medio (ms) 2.070,2	**Tiempo medio (ms)**	795,698
Configuración LiDAR #3			
Ejecución	**Tiempo de respuesta (ms)**	**Tiempo de respuesta de kernel (ms)**	
1	4.938	1.473,11	

2	4.899		1.429,85	
3	4.878		1.420,79	
4	4.850		1.431,28	
5	4.836		1.414,16	
	Tiempo medio (ms)	4.880,2	**Tiempo medio (ms)**	1.433,838

Configuración LiDAR #5		
Ejecución	**Tiempo de respuestal (ms)**	**Tiempo de respuesta de kernel (ms)**
1	9.708	5.265,55
2	9.671	5.427,52
3	9.747	5.291,67
4	9.757	5.338,7
5	9.627	5.104,45
	Tiempo medio (ms) 9.702	**Tiempo medio (ms)** 5.285,578

Tabla 49. Tiempo medio de respuesta de la simulación LiDAR frente a tres configuraciones diferentes en el primer escenario (habitación).

Escenario #2 **6.833.501 vértices y 4.624.832 triángulos**		
Configuración LiDAR #1		
Ejecución	**Tiempo de respuesta (ms)**	**Tiempo de respuesta de kernel (ms)**
1	4.690	2.811,76
2	4.600	3.013,5
3	4.580	3.016,82
4	4.635	3.308,86
5	4.786	3.911,45
	Tiempo medio (ms) 4.658,2	**Tiempo medio (ms)** 3.212,478

Configuración LiDAR #2			
Ejecución	**Tiempo de respuesta (ms)**	**Tiempo de respuesta de kernel (ms)**	
1	4.164	2.313,87	
2	4.144	2.273,3	
3	4.192	2.355,44	
4	4.181	2.375,99	
5	4.267	2.411,98	
	Tiempo medio (ms) 4.189,6	**Tiempo medio (ms)**	2.346,116

Configuración LiDAR #3			
Ejecución	**Tiempo de respuesta (ms)**	**Tiempo de respuesta de kernel (ms)**	
1	9.026	3.431,16	
2	9.104	3.425,78	
3	9.066	4.342,55	
4	9.058	3.611,15	
5	8.890	3.529,75	
	Tiempo medio (ms) 9.028,8	**Tiempo medio (ms)**	3.668,078

Configuración LiDAR #4			
Ejecución	**Tiempo de respuesta (ms)**	**Tiempo de respuesta de kernel (ms)**	
1	8.150	3.109,66	
2	8.349	3.964,11	
3	8.064	3.019,33	
4	8.173	2.973,41	
5	8.087	3.205,29	
	Tiempo medio (ms) 8.164,6	**Tiempo medio (ms)**	3.254,36

Tabla 50. Tiempo medio de respuesta de la simulación LiDAR frente a cuatro configuraciones diferentes en el segundo escenario (terreno).

Escenario #3 **18.000.000 vértices y 6.000.000 triángulos**			
Configuración LiDAR #1			
Ejecución	**Tiempo de respuesta (ms)**	**Tiempo de respuesta de kernel (ms)**	
1	1.096	923,038	
2	1.060	861,569	
3	1.050	933,88	
4	1.064	959,3	
5	1.087	910,836	
	Tiempo medio (ms) 1.071,4	**Tiempo medio (ms)**	917,724
Configuración LiDAR #3			
Ejecución	**Tiempo de respuesta (ms)**	**Tiempo de respuesta de kernel (ms)**	
1	1.841	1.243,03	
2	1.849	1.279,27	
3	1.909	1.217,58	
4	1.814	1.198,57	
5	1.867	1.202,15	
	Tiempo medio (ms) 1.856	**Tiempo medio (ms)**	1.228,12
Configuración LiDAR #5			
Ejecución	**Tiempo de respuesta (ms)**	**Tiempo de respuesta de kernel (ms)**	
1	3.358	2.311,36	
2	3.483	2.364,56	
3	3.363	2.218,61	
4	3.208	2.271,45	
5	3.200	2.086,44	
	Tiempo medio (ms) 3.322,4	**Tiempo medio (ms)**	2.250,484

Tabla 51. Tiempo medio de respuesta de la simulación LiDAR frente a tres configuraciones diferentes en el tercer escenario (MPM).

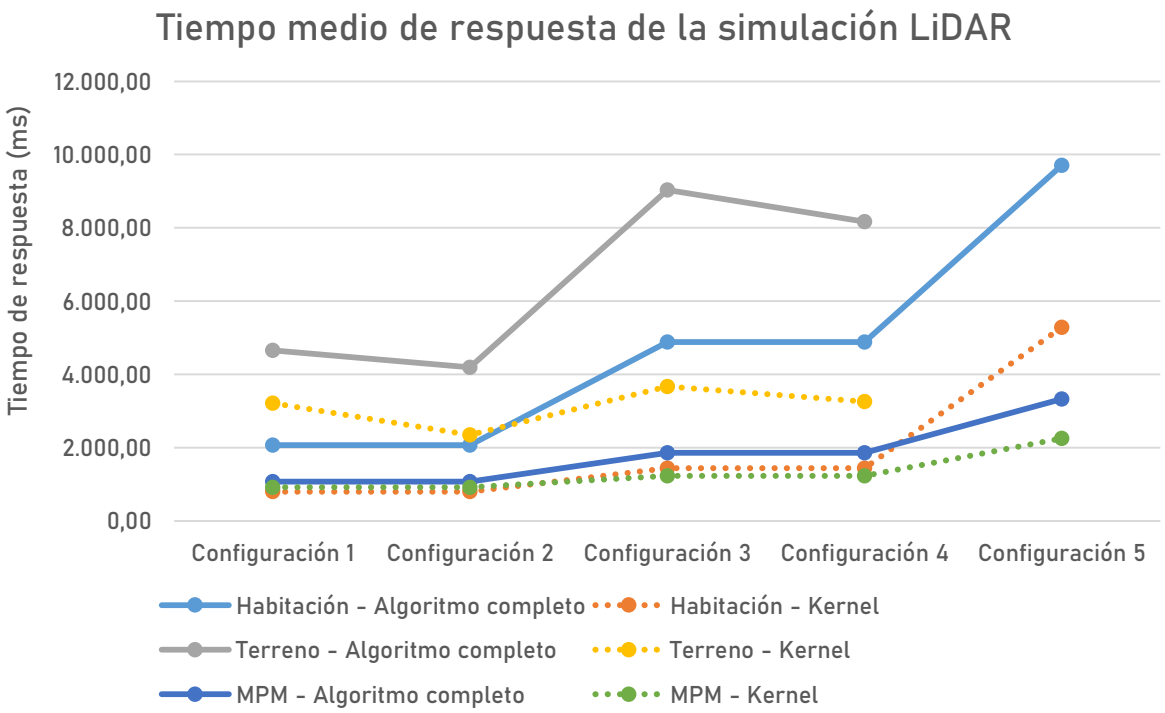


Ilustración 200. Representación gráfica de los resultados obtenidos.

Como resultado de las pruebas realizadas sobre el último escenario, también se captura la nube obtenida del escaneo LiDAR (Ilustración 201). De este modo, se evidencia la capacidad del sensor LiDAR para capturar, y distinguir, cientos de miles de partículas de un sistema MPM, donde la distancia entre ellas es muy reducida.

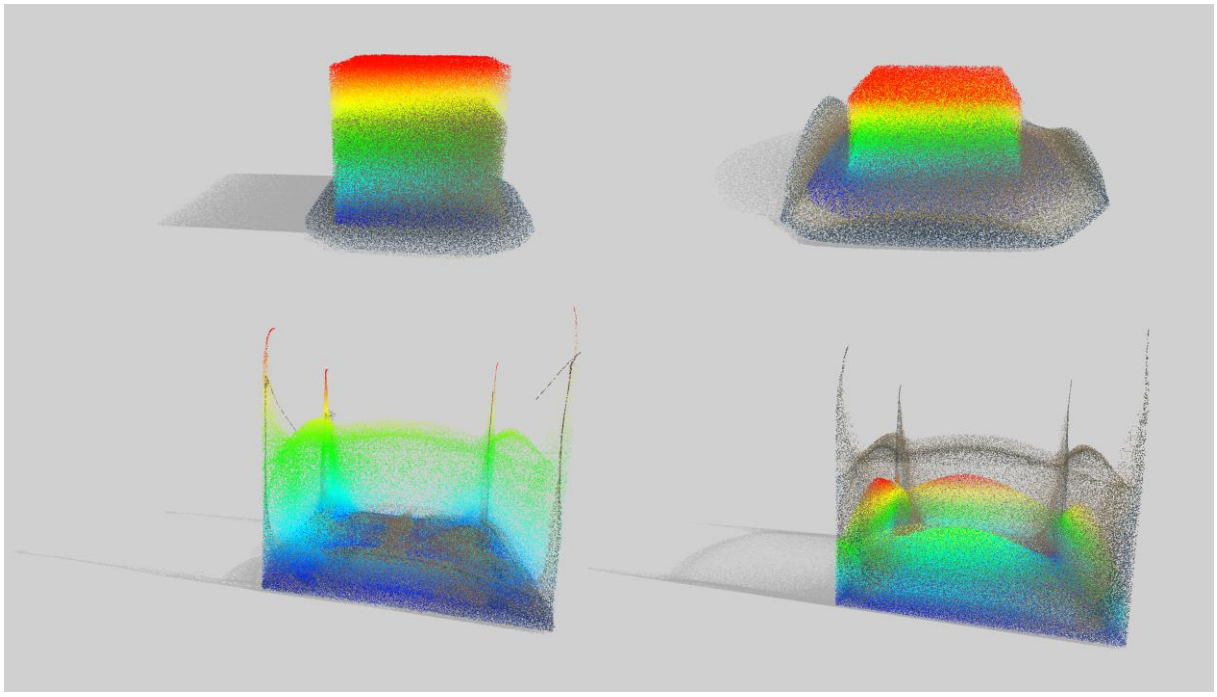


Ilustración 201. Resultados de la simulación LiDAR en el escenario del sistema de partículas MPM.

El tiempo de respuesta hasta ahora documentado representa el núcleo del escaneo LiDAR, donde reside el comportamiento de interés desde el punto de vista de la investigación. Sin embargo, antes de ejecutar esta etapa se deben inicializar los rayos que se emitirán desde el sensor, siendo esta una de las tareas que mayor carga de trabajo presenta, a pesar de su simpleza. Bien podría haberse llevado esta generación a GPU, aunque en la solución final se resuelve únicamente en CPU, por lo que nos interesa conocer la demora que supone la generación de los rayos (Tabla 52).

Generación de rayos de escaneo LiDAR				
	5 millones de rayos		**10 millones de rayos**	
Ejecución	**Tiempo de respuesta (ms)**		**Tiempo de respuesta (ms)**	
1	1.140		2.870	
2	1.143		2.656	
3	1.133		2.673	
4	1.113		2.509	
5	1.089		2.552	
	Tiempo medio (ms)	1.123,6	**Tiempo medio (ms)**	2.652
	20 millones de rayos		**40 millones de rayos**	
Ejecución	**Tiempo de respuesta (ms)**		**Tiempo de respuesta (ms)**	
1	5.143		9.407	
2	5.305		9.512	
3	5.196		9.615	
4	5.650		9.523	
5	5.346		9.274	
	Tiempo medio (ms)	5.328	**Tiempo medio (ms)**	9.466,2

Tabla 52. Tiempo medio de respuesta de la generación de múltiples cantidades de rayos.

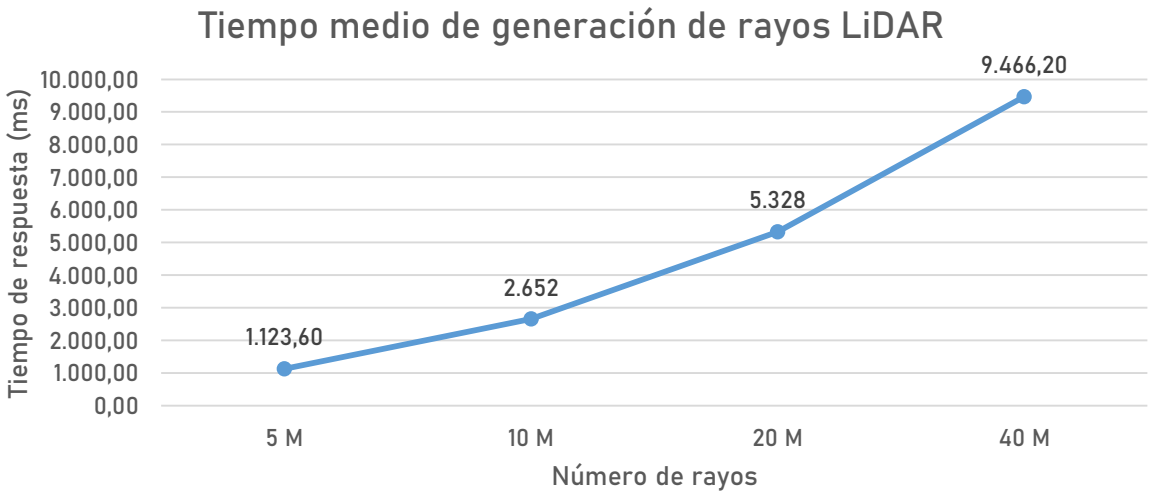


Ilustración 202. Representación gráfica del tiempo medio hallado en la generación de rayos del escaneo LiDAR.

3.1.1.2 Comparativa con la simulación básica

Esta sección nos sirve para comparar la versión final de la simulación con los resultados hallados en la quinta iteración, donde el proceso de escaneo se limita a hallar la primera colisión y no incluye errores. Dado el estado de la aplicación en ese punto, sólo es posible comparar los resultados que surgen del primer escenario (6.6 millones de triángulos), empleando 10 y 20 millones de rayos. Antes de observar los resultados, es necesario comprender que la diferencia entre ambas versiones no reside únicamente en la introducción de errores en la última, lo que supondría claramente una demora respecto de la primera versión. También se optimiza el proceso de escaneo en la especificación de *buffers* y cambia por completo el lanzamiento de todos los rayos con la consideración de múltiples retornos. Además, la estructura del *compute shader* es completamente diferente, no sólo a nivel de flujo; se evitan todas aquellas comparaciones (prescindibles) con estructuras condicionales, se sustituyen por operaciones nativas de GLSL, o se almacenan en variables aquellos valores que van a compararse.

La comparativa se documenta, en primer lugar, mediante la Tabla 53, y por último, a través de la Ilustración 203.

Escenario #1 **3.350.433 vértices y 6.683.902 triángulos**		
10 millones de rayos		
Solución	**Tiempo de respuesta (ms)**	**Tiempo de respuesta de kernel (ms)**
Inicial	6.659,2	5.246,8
Final	2.070,2	795,698
20 millones de rayos		
Ejecución	**Tiempo de respuesta (ms)**	**Tiempo de respuesta de kernel (ms)**
Inicial	13.896,6	11.986,8
Final	4.880,2	1.433,838

Tabla 53. Comparación de resultados obtenidos en las soluciones de escaneo inicial y final.

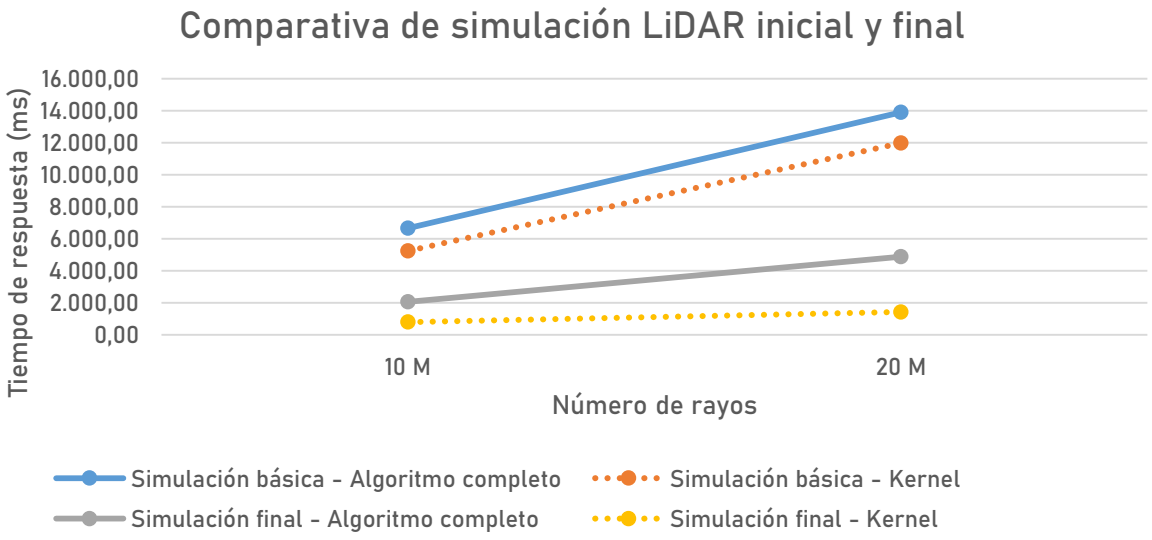


Ilustración 203. Representación gráfica, mediante un diagrama de líneas, de la comparativa entre la primera simulación y la versión final.

3.1.1.3 Escaneo en tiempo real

De los tiempos de respuesta anteriormente documentados se puede extraer que el sistema no permite la actualización de la nube de puntos en tiempo real, al menos para la cantidad de rayos propuesta en cada prueba. El tiempo medio hallado para resolver el escaneo con 10 millones de rayos no es excesivamente alto (algo más de dos segundos), al menos en la primera escena, donde no disponemos de vegetación. Por tanto, podríamos comprobar qué sucede cuando el número de rayos

es mucho menor (suponiendo una escena estática). Las pruebas que se desarrollan emplean los dos primeros escenarios, y una cantidad de rayos situada en 100.000, 500.000 y un millón. El tiempo de respuesta que se representa no sólo incluye el proceso de escaneo, sino también la generación de rayos.

Escenario #1 **3.350.433 vértices y 6.683.902 triángulos**			
	100.000 rayos	**500.000 rayos**	
Ejecución	**Tiempo de respuesta (ms)**	**Tiempo de respuesta (ms)**	
1	46	245	
2	49	222	
3	45	222	
4	44	220	
5	46	232	
	Tiempo medio (ms) 46	**Tiempo medio (ms)**	228,2

Tabla 54. Tiempo medio de respuesta de escaneos con un número más reducido de rayos, empleando para ello el primer escenario.

Escenario #2 **6.833.501 vértices y 4.624.832 triángulos**			
	100.000 rayos	**500.000 rayos**	
Ejecución	**Tiempo de respuesta (ms)**	**Tiempo de respuesta (ms)**	
1	370	613	
2	341	641	
3	321	613	
4	336	619	
5	328	626	
	Tiempo medio (ms) 339,2	**Tiempo medio (ms)**	622,4

Tabla 55. Tiempo medio de respuesta de escaneos con un número más reducido de rayos, empleando para ello el segundo escenario.

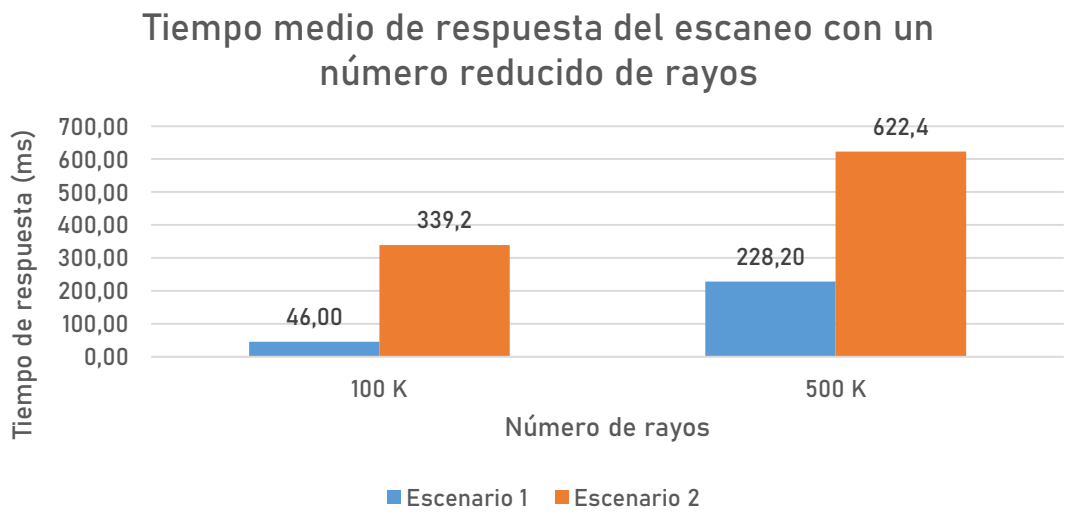


Ilustración 204. Diagrama de barras donde se representa el tiempo medio hallado en escaneos LiDAR con un número reducido de rayos.

3.1.1.4 Actualización de BVH y simulación LiDAR

Hasta este punto, siempre se ha considerado que cualquier escenario incluido en la aplicación es estático, por lo que sería suficiente con inicializar su *Boundary Volume Hierarchy* en el proceso de carga inicial de la escena. Así, se podrá ejecutar la simulación tantas veces como se desee, partiendo de dicha estructura ya construida. Por tanto, la única demora considerable después del inicio procede del escaneo. Sin embargo, no sucede lo mismo cuando la escena puede verse modificada en cada *frame*. En dicho caso, sería necesario construir la estructura de datos de nuevo (siempre y cuando existan cambios) para poder ejecutar la simulación, por lo que la demora es significativamente mayor. No sólo debe construirse la estructura, sino que es necesario agrupar toda la geometría de nuevo, calcular su AABB, obtener códigos Morton, etc.

Esta dificultad no se puede entender únicamente como un problema derivado del enfoque implementado, aunque ciertamente podrían haberse desarrollado más algoritmos, como la actualización del BVH. De esta manera, no sería necesario recurrir a la construcción completa antes de cada nuevo escaneo. Aunque se descarta el desarrollo de un algoritmo de actualización del BVH debido a la extensión temporal del proyecto, bien podría considerarse como un posible trabajo futuro.

La prueba que aquí nos ocupa pretende documentar el tiempo derivado del proceso íntegro de construcción, con todas las operaciones previas o posteriores implicadas, así como del escaneo LiDAR. Dada la complejidad del proceso que se

pretende monitorizar, se evita medir únicamente tiempos de GPU, por lo que monitorizaremos el tiempo utilizado en un método dedicado a la resolución de ambos problemas. En este caso, no nos interesa tanto el rendimiento del escaneo, sino que más bien se busca obtener una visión global de la demora obtenida. Por tanto, fijaremos el radio de búsqueda del BVH en 100, y emplearemos 5 y 10 millones de rayos en dos pruebas diferentes. También se propone variar el número de partículas representadas, por lo que plantearemos las dos configuraciones anteriores para 100.000 y 250.000 partículas.

Las pruebas se llevarán a cabo mediante una orden manual de escaneo, por lo que este se puede aplicar sobre cualquier *frame*. No se considera que la estructura de la escena pueda introducir un sesgo en los resultados de las pruebas, aunque si así ocurriera, nos permitiría documentar el tiempo medio de respuesta ante diferentes escenarios.

Escenario MPM **100.000 partículas. 7.200.000 vértices y 2.400.000 triángulos**			
	5 millones de rayos	**10 millones de rayos**	
Ejecución	**Tiempo de respuesta (ms)**	**Tiempo de respuesta (ms)**	
1	4.235	5.866	
2	4.191	5.578	
3	4.201	5.799	
4	4.264	5.843	
5	4.172	5.634	
	Tiempo medio (ms) 4.212,6	**Tiempo medio (ms)**	5.744

Tabla 56. Tiempo medio de respuesta procedente de la construcción de un BVH y del escaneo LiDAR en un frame cualquiera de una escena MPM con 100.000 partículas.

Escenario MPM **250.000 partículas. 18.000.000 vértices y 6.000.000 triángulos**		
	5 millones de rayos	**10 millones de rayos**
Ejecución	**Tiempo de respuesta (ms)**	**Tiempo de respuesta (ms)**
1	8.110	9.777

2	8.219	9.865	
3	7.948	9.872	
4	7.875	9.874	
5	7.736	10.095	
	Tiempo medio (ms) 7.977,6	**Tiempo medio (ms)**	9.896,6

Tabla 57. Tiempo medio de respuesta procedente de la construcción de un BVH y del escaneo LiDAR en un frame cualquiera de una escena MPM con 250.000 partículas.

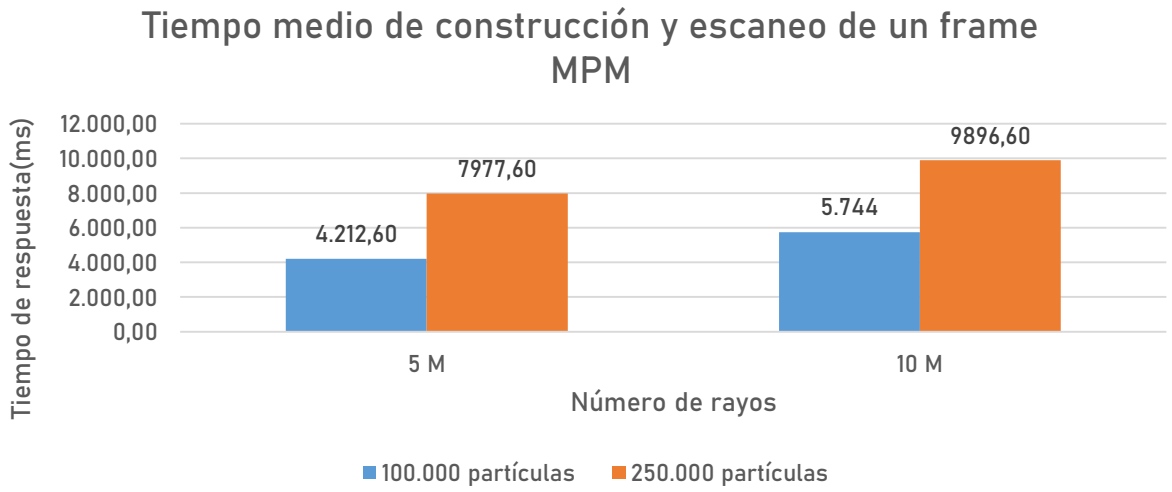


Ilustración 205. Diagrama de barras donde se representa el tiempo medio de respuesta procedente de la construcción de un BVH y del escaneo LiDAR en un frame cualquiera de una escena MPM con un número de partículas variable.

3.2 Resultados y discusión

A partir de todas las pruebas documentadas es posible extraer información muy valiosa de la solución final desarrollada en este Trabajo Fin de Máster:

1. Tras introducir errores en el comportamiento del sensor, no sólo no se observan mayores demoras que las ya conocidas, sino que **se ha reducido notablemente el tiempo de ejecución**, dado que a medida que se aplicaron dichos cambios se realizaron múltiples revisiones que permitieron optimizar el código, tanto a nivel de *shader* como a nivel de aplicación en CPU (definición y actualización de *buffers*, lectura, etc).

2. **Se identifica una dependencia alta entre el tiempo de respuesta del escaneo y la densidad de vegetación de la escena**. Parece evidente que cuando este tipo de modelo aparece en la escena, surgen más retornos, y

por tanto, el tiempo de respuesta aumenta. Los rayos generados tras atravesar una superficie no son diferentes a los rayos que se definen desde el exterior, por lo que el problema de los retornos múltiples se podría interpretar como un aumento del número de rayos emitidos en la escena (aunque con una capacidad de ver más allá de la vegetación).

3. **La generación de rayos es un punto de demora importante**. Ciertamente, podría haberse llevado este proceso a GPU. Además, el *buffer* de rayos debe generarse más tarde, por lo que su definición antes de instanciar los rayos en GPU no debería introducir una carga adicional.

4. A partir de las pruebas realizadas mediante un escaneo reducido, se extrae que **la solución final podría llegar a adaptarse un sistema de escaneo en tiempo real**, siempre que no se busque una nube de gran densidad y el escenario sea estático. En esta situación, se documenta la resolución de un escaneo con 10 millones de rayos en poco más de dos segundos, y de 100.000 rayos en 46 ms.

   Una situación opuesta se presenta en las últimas pruebas realizadas sobre el sistema de partículas MPM, donde la escena es dinámica y el número de rayos es elevado. En este caso, se obtienen tiempos de respuesta más elevados.

   Un escenario de conducción autónoma podría asimilar un menor número de rayos (por debajo del millón), pero el entorno podría ser dinámico, más allá de la movilidad del vehículo, la cual no produce una demora de la simulación.

4 CONCLUSIONES Y TRABAJOS FUTUROS

En este Trabajo Fin de Máster se ha desarrollado un sistema de escaneo 3D, basado en un sensor LiDAR, que permite capturar entornos de complejidad variable. Para ello, se emplea un enfoque basado en el comportamiento físico del sensor, alejándose así de otras soluciones halladas. Debido a este planteamiento, se observa que el problema presenta un elevado coste computacional, lo que justifica el desarrollo de la solución utilizando computación paralela. Sobre un concepto tan básico como el lanzamiento de rayos (*ray-casting*) se añaden nuevas capas de comportamiento LiDAR, basadas en las peculiaridades y errores vinculados al sensor, en las cuales se actúa sobre múltiples superficies (terreno, agua, materiales brillantes, etc). Finalmente, se introduce una interfaz gráfica que permite seleccionar los valores de aquellos parámetros que modelan la simulación del sensor LiDAR.

Dada la utilización de un *framewok* de computación paralela, se presenta esta documentación, así como la solución, como una manera de comprobar el rendimiento del escaneo, sometiéndolo a escenarios complejos y a configuraciones con un elevado número de rayos. Por tanto, la aplicación se ha orientado hacia una solución de alto rendimiento.

El trabajo se ha desarrollado considerando en todo momento las múltiples aplicaciones de este sistema; desde una aplicación de conducción autónoma, hasta la generación de nubes de puntos de gran densidad para la alimentación de algoritmos de *deep learning*. Igualmente, se presenta la posibilidad de que un usuario pueda interaccionar con el sistema para recuperar nubes de puntos útiles en cualquier otro campo. A partir de todas estas consideraciones, se desarrollan varios puntos en este trabajo.

En primer lugar, la aceleración del sistema podría permitir su integración en un sistema en tiempo real. Por otro lado, la disminución del tiempo de respuesta de la simulación también es relevante tanto para la obtención de un gran número de nubes de puntos, como para aquella situación en la que un usuario recupera una única nube a partir de la interfaz gráfica. Igualmente, se establece dicha interfaz para asegurar que un usuario pueda regular el comportamiento del sensor.

Las nubes de puntos generadas contienen múltiples tipos de información, y por tanto, también se habilitan múltiples maneras de visualizar estas nubes. Más allá de datos como la intensidad o el valor de retorno, se añade información semántica que

puede ser de gran utilidad para aquellos sistemas que emplean nubes de puntos para comprender la estructura de los objetos mediante de un entrenamiento. De esta manera, la introducción de escenarios procedurales permite que estos sistemas puedan disponer de un elevado número de nubes sobre las que aprender (todas ellas diferentes).

En definitiva, en este trabajo se propone un nuevo enfoque de escaneo que pretende simular, de la más manera más precisa posible, un sensor LiDAR. A partir de esta base aquí planteada, es posible continuar el trabajo adecuándolo a una determinada aplicación, o habilitando un *framework* que, lejos de un desarrollo experimental, permita al usuario seleccionar cómodamente todos los parámetros de la escena. No sólo nos referimos a aquellas variables vinculadas al sensor LiDAR, sino también a todos aquellos parámetros que pudieran surgir de la generación del escenario.

Además, podríamos pensar en un trabajo futuro que completara en mayor medida esta solución. La aplicación desarrollada permite ejecutar una simulación, que produce una nueva nube de puntos, pero también elimina el resultado anterior. Se podría introducir así el problema del registro de diversas nube de puntos, una tarea que puede ser tediosa en la realidad, pero que en nuestro caso no tiene por qué complicar en gran medida el sistema. Igualmente, utilizando el escaneo aéreo, se puede extender el sistema con cualquier otro tipo de ruta o movimiento que pueda adaptarse a una necesidad concreta.

Desde el punto de vista de los recursos que emplea la aplicación, parece necesario continuar este trabajo mediante la evaluación de algunas modificaciones que pudieran reducir la memoria empleada. Esto es especialmente necesario si se desea continuar con un enfoque de alto rendimiento, donde los escenarios o las nubes de puntos suponen un espacio importante de memoria. No sólo se trata de una necesidad a nivel de rendimiento, sino también de adaptación al sistema; el tamaño máximo de un SSBO viene dado por la tarjeta gráfica, y como tal, es conveniente reducir el espacio reservado. Sería interesante desarrollar un trabajo donde pueda evaluarse cuál es el efecto, en la nube resultante, de la utilización de operaciones como el empaquetado de múltiples valores en uno sólo (*pack*, *unpack*). Las consecuencias podrían evaluarse en términos de precisión, pero también de rendimiento, entendiendo que estas operaciones nativas de GLSL pueden representar cierta carga de trabajo.

Otro trabajo futuro podría ser la evaluación del sistema mediante parámetros estadísticos (error cuadrático medio, desviación, etc), los cuales permitirían medir las diferencias entre el escaneo implementado y el escaneo generado por un sensor real. No sólo se debe entender como una manera de contrastar la solución, sino también como una manera de dirigirla, con el fin de minimizar el error. En cualquier caso, es un problema que requiere de un profundo análisis, debido a que la situación ideal debería considerar un escenario modelado que se escaneara mediante ambos enfoques. Sin embargo, parece difícil obtener el resultado de un verdadero escaneo LiDAR sobre un escenario sintético, y en el caso de emplearse una malla generada a partir de un sensor LiDAR existen problemas evidentes, como la introducción de errores del sensor en la propia malla.

Por último, uno de los trabajos más interesantes que podrían desarrollarse a partir de esta simulación es la utilización de las nubes obtenidas para entrenar un sistema de inteligencia artificial. De esta manera, se podría evaluar si las nubes de puntos recuperadas de un escaneo sintético pueden ayudar a mejorar los resultados que habitualmente se obtienen en problemas tales como la segmentación semántica o la clasificación de nubes de puntos.

5 APÉNDICES

5.1 Guía original del Trabajo Fin de Máster

La guía original de este Trabajo Fin de Máster fue publicada en el primer cuatrimestre del curso académico 2020-2021[27]. Sobre dicha guía no se ha producido modificación alguna.

5.2 Instalación y configuración del sistema

En el capítulo de Alcance se indica que uno de los entregables de este trabajo es un fichero ejecutable que permite comprobar la solución desarrollada, y por tanto, esta sería la manera más sencilla de ejecutar la solución.

Sin embargo, un trabajo futuro podría partir de este proyecto, y por tanto, sería necesario configurar el entorno correctamente. Antes de comenzar esta descripción es necesario destacar que el sistema debe soportar al menos OpenGL 4.5; en cualquier otro caso es posible escoger una versión anterior modificando la primera línea de los *shaders* y el fichero *Window.cpp*, pero no se garantiza el correcto funcionamiento de la aplicación.

Los pasos necesarios para configurar el entorno son los siguientes:

1. Copiar todas las librerías ubicadas en los entregables de este Trabajo Fin de Máster en cualquier otro punto del sistema, que consideraremos de ahora en adelante que será el escritorio (*Desktop*). Una alternativa a este paso es descargar cada una de las librerías que se listan a continuación en una carpeta a la que podemos llamar *Libraries*.
    - GLEW. Versión actual: 2.1.0 (julio de 2017 (OpenGL 4.6)).
    - GLFW. Versión actual: 3.3 (abril de 2019).
    - GLM. Versión actual: 0.9.9.5 (abril de 2019).
2. Instalación de Visual Studio 2019, si no estuviera preparado este entorno de desarrollo. No se identifican dependencias respecto de extensiones o componentes muy específicos, por lo que nos basta con seleccionar las opciones Desarrollo para el escritorio con C++ y Desarrollo de la plataforma

[27] http://eps-anterior.ujaen.es/TFMtemporal/mostrarTFM.php?id=425

universal de Windows. Si no se seleccionaran en la instalación es posible hacerlo mediante la modificación de la instalación a través de *Visual Studio Installer*.

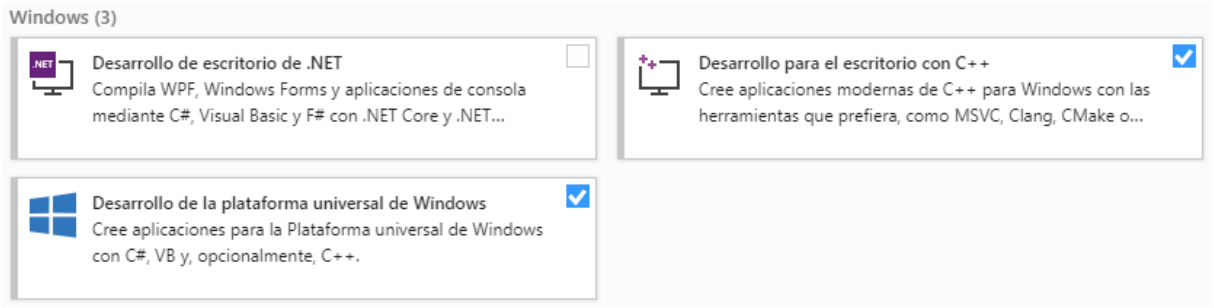


Ilustración 206. Opciones mínimas que deben seleccionarse en la instalación de Visual Studio.

3. Creación de un nuevo proyecto de Visual Studio. La plantilla de *Aplicación vacía (Windows Universal)* es suficiente.

4. El siguiente paso consiste en enlazar todas las librerías antes citadas con el proyecto. Para ello es suficiente con hacer click derecho en el nombre de nuestra solución y abrir el menú de Propiedades. El primer punto donde se vinculan las librerías se encuentra en *C/C++ > General > Directorios de inclusión adicionales*. Es decir, aquí se incluirán todos los directorios *include* de nuestras librerías, si las hubiera. A continuación se listan todas las carpetas que se deberán referenciar:

    - *%userprofile%/Desktop/Libraries/glew/include*
    - *%userprofile%/Desktop/Libraries/glfw/include*
    - *%userprofile%/Desktop/Libraries/glm* (no contiene una carpeta *include* per se).
    - *Libraries/lodepng* (se debe añadir al proyecto como código fuente).
    - *Libraries/imgui y Libraries/imgui/examples* (se debe añadir al proyecto como código fuente).

5. La función de los directorios de inclusión adicionales no es otra que poder mencionar los ficheros de programación como si se hallaran bajo el directorio *root* de recursos (en este proyecto, *ScannerSimulator/ScannerSimulator*). Por tanto, aquí se van a incluir otras

tantas líneas que no están vinculadas con librerías, sino con la propia estructura de carpetas donde se encuentra nuestra código fuente.

- *Source*.
- *Source/PrecompiledHeaders* (acceso al fichero *stdafx.h*).

6. Vinculación de directorios adicionales de las bibliotecas en la opción *Vinculador* > *General* > *Directorios de bibliotecas adicionales*, donde habrá que incluir las siguientes líneas:
   - *%userprofile%/Desktop/Libraries/glew/lib/Release/x64*
   - *%userprofile%/Desktop/Libraries/glfw/lib-vc2019*
7. También es necesario incluir dependencias adicionales bajo la opción *Vinculador* > *Entradas* > *Dependencias adicionales*, donde deben referenciarse los siguientes ficheros:
   - *opengl32.lib*
   - *glu32.lib*
   - *glew32.lib*
   - *glfw3.lib*
8. Como penúltimo paso se recomienda seleccionar el estándar utilizado, ISO C++ 17, en *C/C++* > *Idioma* > *Estándar de lenguaje C++*.
9. Por último, es necesario incluir el fichero *glew32.dll*, el cual se puede obtener de *Desktop/Libraries/glew/bin/Release/x64*, en las carpetas *x64/Debug* y *x64/Release*, situadas a su vez bajo la carpeta principal del proyecto (en nuestro caso, bajo *ScannerSimulator/*). Puede ser necesario crear ambas carpetas, dado que estas se generan cuando se compila el proyecto por primera vez con las opciones *Debug* y *Release*, algo que en este punto puede no haberse producido.

En lugar de configurar un entorno, también es posible acudir directamente al fichero ejecutable que se entrega junto a este documento. Para ello, sólo es necesario desplazarse a la carpeta *Ejecutable* y ejecutar el fichero *LiDARSimulator.exe*. Para gestionar el escenario de partida es necesario hacer uso del fichero *SceneIndex.txt*, ubicado en la carpeta *Settings*, donde debemos incluir un valor entre 0 y 2, ambos

inclusive, donde 0 cargará una habitación, 1 un terreno, y el identificador 2 un sistema de partículas MPM (250.000 partículas).

5.3 Manuales de usuario

Para completar este documento se incluye un manual de usuario que permite conocer todas las funcionalidades existentes en la aplicación, su localización, y cómo manipularlas. En primer lugar, la interacción básica entre usuario y aplicación se desarrolla mediante teclado y ratón. Por tanto, las posibilidades de interacción existentes se describen en la Tabla 58 y la Tabla 59.

Controles de teclado		
Acción	**Tecla**	**Descripción**
Orbit (XZ)	**X**	Rotación de la cámara alrededor del punto objetivo. Sólo modifica las coordenadas X y Z, razón por la cual se le da este nombre a la acción.
Deshacer *orbit* (XZ)	**Control + X**	Movimiento Orbit (XZ) en la dirección opuesta.
Orbit (Y)	**Y**	Rotación de la cámara sobre el punto objetivo
Deshacer *orbit* (Y)	**Control + Y**	Movimiento Orbit (Y) en la dirección opuesta.
Dolly	**W, S** (mientras se presiona el botón derecho de ratón)	Traslación positiva sobre el eje X de la cámara.
Truck	**D, A** (mientras se presiona el botón derecho de ratón)	Traslación positiva sobre el eje X de la cámara.
Boom	↑	Traslación positiva en el eje Y de la cámara.
Crane	↓	Traslación negativa en el eje Y de la cámara.
Restaurar cámara	**R**	Restaura los parámetros iniciales de la cámara.
Captura de pantalla	**K**	Obtiene una captura de pantalla con el tamaño indicado en el menú correspondiente.
Continuar animación	**I**	Continúa una animación en curso, incluyendo una simulación LiDAR incremental.

Tabla 58. Acciones que pueden llevarse a cabo mediante la pulsación de teclas.

Controles de ratón		
Resultado	**Acción**	**Descripción**
Zoom +/-	*Scroll wheel*	Ajuste de la distancia focal para mostrar con mayor detalle una parte de la escena (centro de todo el escenario representado).
Pan	Desplazamiento horizontal (botón derecho)	Modifica el punto objetivo en un plano paralelo a XZ. Rotación del objetivo de la cámara respecto de la posición de la misma, el cual se mantiene intacto. El objetivo rota, pero se mantiene en el plano que conforma el eje X y Z de la cámara.
Tilt	Desplazamiento vertical (click derecho)	Modifica el punto objetivo verticalmente. Rotación del objetivo de la cámara respecto de la posición de la misma, el cual se mantiene intacto. El objetivo rota, pero se mantiene en el plano que conforma el eje Z e Y de la cámara.

Tabla 59. Acciones que pueden llevarse a cabo únicamente mediante controles de ratón.

Además de la interacción mediante teclado y ratón, la aplicación también dispone de una interfaz cuya principal complejidad radica en conocer el concepto que representa cada control, especialmente cuando se trata de aquellos relacionados con el proceso de escaneo.

En primer lugar, el nodo raíz de esta interfaz es un menú que se muestra en la parte superior de la ventana en forma de barra (Ilustración 207), donde se presentan las siguientes opciones:

- *Settings*. Acceso al resto de ventanas de la interfaz, las cuales serán el siguiente objetivo de este manual.
- *Help*. Muestra información sobre el proyecto a través de la opción *About the project*.
- *Lock camera*. Bloquea movimientos de la cámara para evitar que otras acciones paralelas puedan influir en el estado de la misma. Este es el caso, por ejemplo, de la interacción con un control de tipo *slider* en la interfaz (barra de desplazamiento para seleccionar el valor de alguna variable), donde el movimiento horizontal de la misma con el ratón se podría interpretar erróneamente como un movimiento de tipo *pan* de la cámara.
- *Número de FPS (frames per second)*. No se trata de un control sobre el que interactúe el usuario, sino que indica el número de imágenes por segundo

que la aplicación es capaz de generar. El umbral para considerar una animación fluida se encuentra en 24.

Por el tipo de interfaz elegida, la cual se *renderiza* junto a la escena, se decide evitar aquellos redibujados innecesarios, razón por la cual en algunas ocasiones se muestra una menor cantidad de FPS. Por tanto, la valor real se obtiene cuando existe una interacción intensiva del usuario con la escena.

Ilustración 207. Barra principal de la interfaz, desde donde se accede al resto de funcionalidad.

La primera ventana que podemos mostrar se encuentra en la opción *Settings* > *Rendering*, donde podemos manipular qué se muestra en la escena *renderizada* y cómo (Ilustración 208). A continuación se describen todas las opciones disponibles:

- *Background color*. Color de fondo de la escena. En el desarrollo implica únicamente el color que se asigna por defecto a la imagen al limpiar los *buffers* de la escena. Se compone de valores entre 0 y 255 en los canales *Red*, *Green* y *Blue*.

- *Scattering*. Factor de iluminación global, donde 0 genera una escena donde el color de la misma depende únicamente de las luces integradas. Por esta razón, este factor se puede interpretar como una base de color que permite observar nítidamente ciertas partes de superficies en sombra. En otras aplicaciones, la iluminación global se puede implementar como una luz más de la escena.

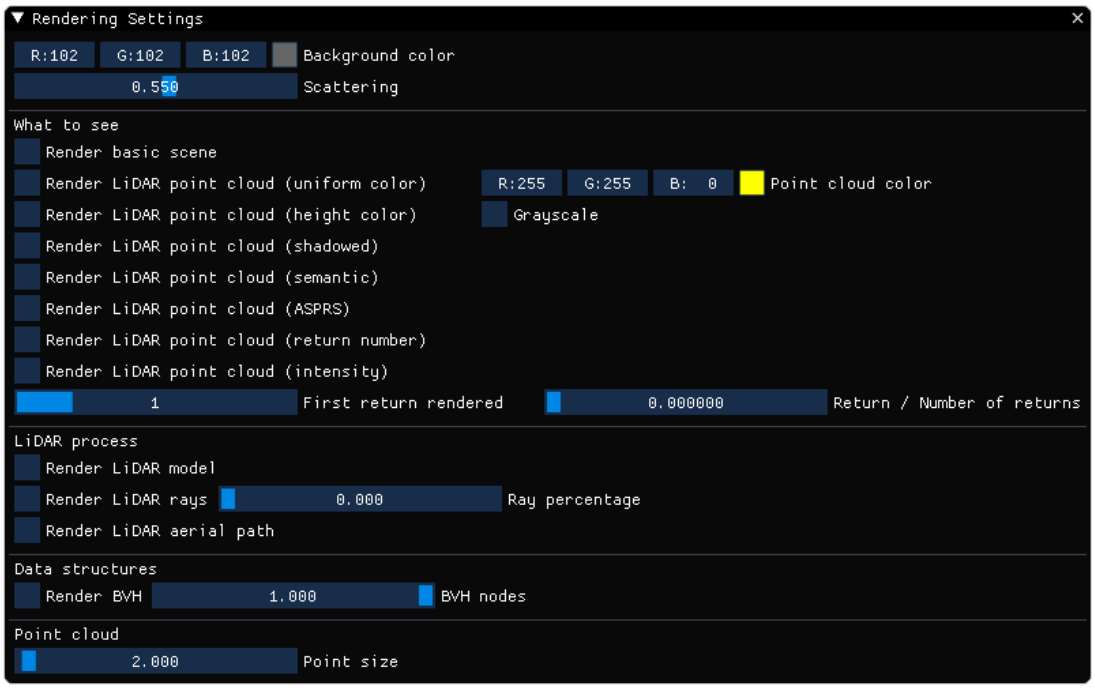


Ilustración 208. Ventana de configuración del rendering de la aplicación.

- Apartado "*What to see*", donde el usuario puede decidir qué *renderizar* en términos de escenario original y resultados de LiDAR. Una composición de todas las opciones que se citan a continuación se encuentra en la Ilustración 209.
    - **Escena básica**: mallas de triángulos que conforman la escena original. En algunas ocasiones, para visualizar correctamente las nubes de puntos podemos deshabilitar este *renderizado*.
    - **Nube de puntos cuyo color depende de la altura** (coordenada Y del punto).
    - **Nube de puntos que adopta los colores originales de la escena**, como si se tratara de una fusión de información de varias fuentes de datos.
    - **Nube de puntos semántica, donde el color vendrá dada por una clase específica** creada para el objeto (no sigue ningún estándar).
    - **Nube de puntos semántica, donde el color viene dado por alguna de las clases que contempla estándar ASPRS**.
    - **Nube de puntos de color dependiente del identificador de retorno**. Una emisión láser puede producir múltiples puntos, y todos

ellos se pueden identificar mediante un valor de retorno, sea el primer retorno el más cercano al dispositivo LiDAR.

- **Nube de puntos de color dependiente de intensidad**. La intensidad percibida por el sensor depende de múltiples factores: reflectancia del objeto, distancia, ángulo de incidencia, etc.

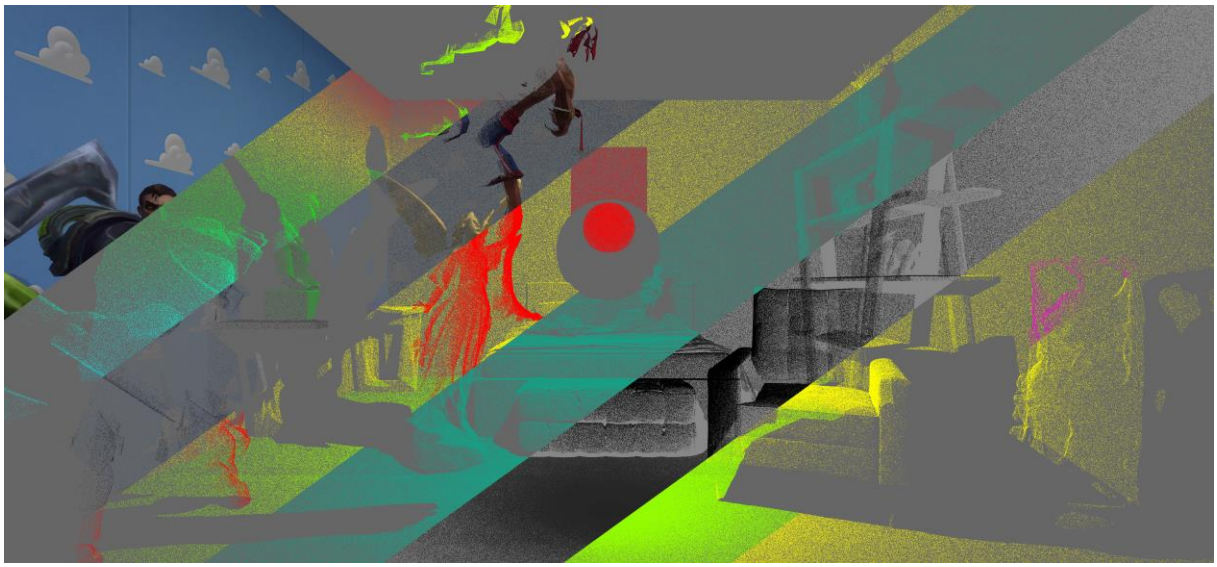


Ilustración 209. Composición de todas las nubes de puntos disponibles y la escena original.

- Primer retorno *renderizado*. El valor oscila entre 1 y el máximo retorno habilitado. Si el valor es 1 se renderizan todos los puntos, mientras que si es 3, sólo se renderizan aquellos puntos cuyo valor de retorno es igual o mayor que 3. Esta opción podría utilizarse, por ejemplo, para descartar la visualización de aquellas colisiones relacionadas con la vegetación.
- *Return / Number of returns*. Una emisión LiDAR puede generar múltiples puntos, pero todos ellos se enlazan con este número total. Es decir, dada una colisión cuyo valor de retorno es 3, correspondiente a una emisión que ha generado 4 puntos, se puede hallar el coeficiente $\frac{3}{4}$. El único objetivo de esta opción es completar el descarte de superficies tales como la vegetación. Nótese como no todas las emisiones producen el mismo número de retornos, y por tanto, la opción anterior no suele ser completamente válida para hallar únicamente el terreno. Por ejemplo, una colisión de valor de retorno 1 puede ser la última colisión generada por esa emisión, mientras que para otra emisión láser el último retorno puede tomar como valor 4.

- *Rendering* ligado al proceso de escaneo del dispositivo LiDAR:
    - *Renderizar* el modelo que representa el sensor, el cual permite al usuario ubicar la posición del mismo. El modelo depende del tipo de LiDAR seleccionado, terrestre o aéreo.
    - *Renderizar* rayos vinculados al proceso de escaneo. Se incluye un *slider* para indicar qué porcentaje de rayos debe dibujarse. En general, no es una opción demasiado útil, dado que implica dibujar millones de rayos, lo que dificulta su correcta interpretación.
    - *Renderizar* ruta aérea, siempre y cuando el usuario haya dibujado una ruta en el *canvas* que más tarde veremos.
- **Estructuras de datos: BVH (*Bounding Volume Hierarchy*).** Podemos representar la estructura de datos que permite resolver de manera eficiente todo este proceso de escaneo. Al tratarse de un árbol donde las hojas son cubos alineados con los ejes, permite añadir un *slider* donde indicar qué porcentaje de nodos queremos representar, comenzando por las hojas, de tal manera que estaríamos observando diferentes niveles de abstracción de este árbol.
- **Tamaño de los puntos de la nube.** En ocasiones es necesario modificar este valor, especialmente pensando en posibles capturas de pantalla.

Ilustración 210. Modelo original junto al resultado del escaneo del mismo. Nubes de puntos con un tamaño de punto 8 y 20.

La siguiente ventana que se puede reconocer es la configuración de las capturas de pantalla (Ilustración 211), donde se puede modificar tanto el multiplicador

del tamaño actual de ventana, como el nombre del fichero donde se almacena la captura. Aunque se puede generar una captura a tamaño real, es común utilizar un tamaño mayor para obtener mayor resolución.

La captura de pantalla que se obtiene al pulsar el botón se almacena en la carpeta donde se encuentra el fichero ejecutable, siempre que se ejecute la aplicación desde este fichero, o en la carpeta *root* de nuestro código fuente, si se ejecuta la aplicación desde el entorno de desarrollo.

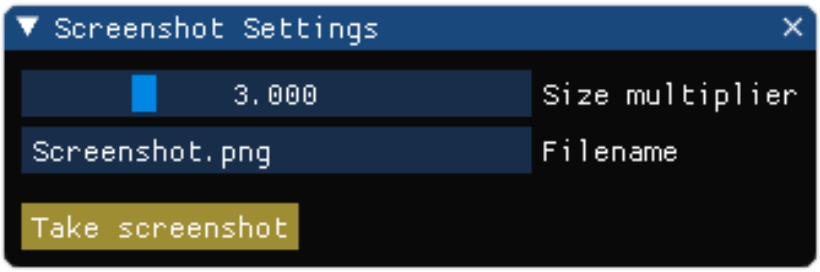


Ilustración 211. Ventana de configuración de capturas de pantalla.

Otra ventana que puede suponer el núcleo de esta aplicación es la que afecta al proceso de escaneo (*LiDAR Settings*). No sólo posibilita comenzar la simulación, sino que además incluye múltiples controles que permiten gestionar los parámetros que influyen en el resultado final (Ilustración 212, Ilustración 213 e Ilustración 214).

En primer lugar, esta ventana se divide a su vez en tres pestañas: parámetros generales y aquellos relacionados con el escaneo aéreo o terrestre. En la pestaña general se muestran todos aquellos parámetros que influencian ambos tipos de escaneos, que son la gran mayoría. A continuación se describen todos los parámetros que se presentan en esta pestaña:

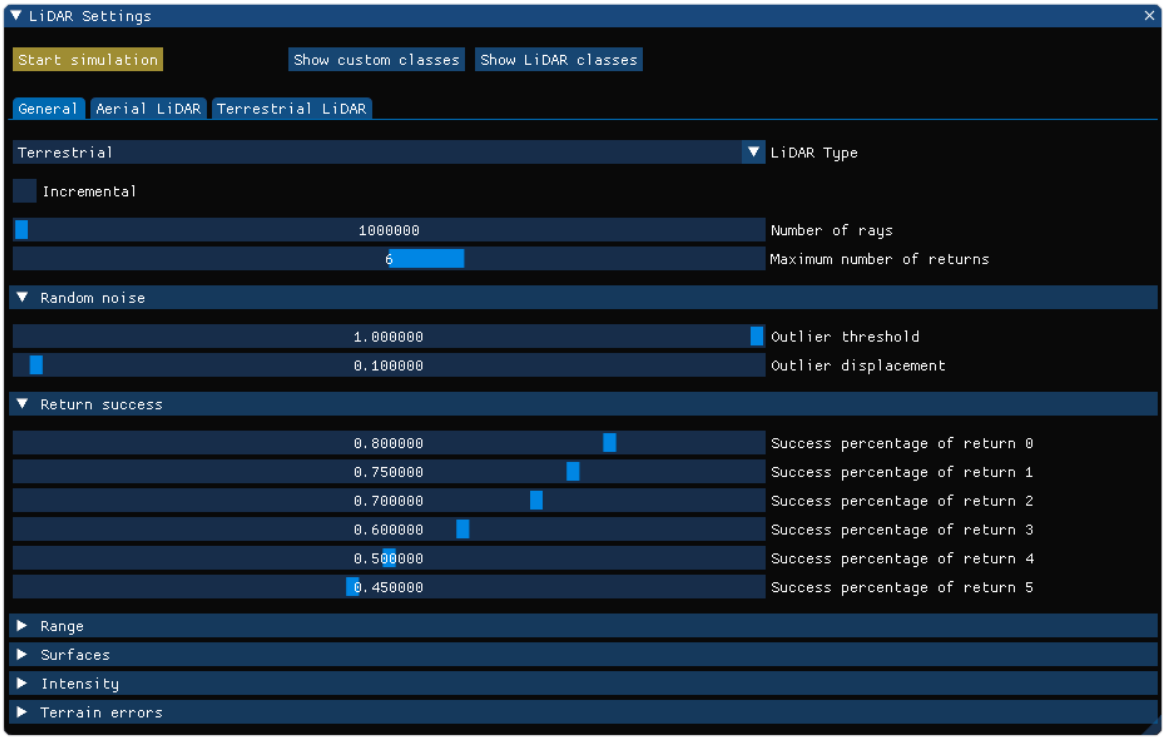


Ilustración 212. Configuración general de escaneo LiDAR. Opciones de ruido y probabilidad de éxito de retornos.

- **Tipo de dispositivo LiDAR**: aéreo o terrestre.
- **Incremental**. Se aplica a aquellos escenarios en los que se desea animar el proceso de escaneo, de tal manera que este se resuelve en pequeños incrementos.
- **Número de rayos**. Se trata de un valor que oscila entre un millón y veinte millones de rayos. El espacio que se considera es el mismo en todos los casos, pero una mayor cantidad de rayos permite capturar detalles de menor tamaño (y por tanto, aumenta el tamaño de la nube resultante).
- **Máximo número de retornos**. Este valor se sitúa en el intervalo [1, 10], sea este último un valor seleccionado por una cuestión de rendimiento, aunque los dispositivos LiDAR no son capaces de alcanzar dicho número de retornos con una única emisión láser. El valor aquí seleccionado influencia otros controles vistos (selección de máximo retorno *renderizado*, etc) y aún por ver.
- ***Random noise***. No pretende simular un error concreto y documentado en la bibliografía, sino incluir un error aleatorio que pudiera darse en la nube resultante. Para controlar este ruido aleatorio habrá que especificar a

cuántos puntos se aplica (*outlier threshold*) y en qué cantidad (*outlier displacement*). Al no utilizarse una distribución uniforme, sino valores aleatorios, el umbral no tiene necesariamente una vinculación directa con un porcentaje. Por defecto, el umbral es uno y no existe ruido en el resultado, mientras que si el valor fuera cero se aplicaría a todos. El desplazamiento de aquellos puntos erróneos viene dado por el segundo parámetro.

- ***Éxito de retornos***. Permite controlar el nivel de éxito que existe en cada nivel de retorno. Así, la probabilidad de que una colisión tenga éxito en el primer retorno debería ser mayor que en el último nivel. De nuevo, el número de controles que aquí aparecen depende del máximo retorno antes indicado. La distribución de números aleatorios no es uniforme, y por tanto, los porcentajes indicados por el usuario son meramente indicativos.

Ilustración 213. Configuración general del escaneo LiDAR. Apartados de rango y superficies.

- Rango de emisiones láser, para lo cual se indican tres parámetros:
    - Máximo tiempo de desplazamiento, en nanosegundos. A partir de este valor y la velocidad de la luz es posible calcular la distancia máxima que pueden alcanzar los rayos, de tal manera que a partir de ese valor se pueden descartar colisiones.

- Offset en la Y. Es una limitación de ángulo, especialmente en el escenario de escaneo terrestre, dado que el dispositivo no es capaz de abarcar una esfera completa, sino que existe un cierto ángulo, aplicable a los 180 grados verticales, que no podrá captarse. Más concretamente, sólo se aplica en la zona inferior (se considera que es posible emitir rayos con dirección $(0, 1, 0)$).
- Offset en XZ. De nuevo, se trata de una limitación de ángulo, pero esta vez aplicable al escaneo aéreo, de tal manera que no es posible abarcar 180 grados en el lanzamiento de rayos. Realmente se trata de una mera rotación del escenario anterior.

- **Superficies**. Este apartado nos permite manipular los resultados obtenidos en múltiples tipos de superficies, y más concretamente, superficies en las que suelen producirse errores.
  - **Superficies brillantes** (similares a espejos), donde el error se puede ilustrar como una mera traslación, de tal manera que las emisiones que captan esta superficie necesitan más tiempo para regresar debido a los rebotes. Por esta razón, los parámetros que pueden controlarse son la traslación base (se aplica de manera global a todas las superficies) y el peso de un error que se vincula a una traslación aleatoria vinculada a cada objeto diferente (además de la traslación base, se aplica un error aleatorio por malla para evitar que todas las superficies brillantes obtengan una misma traslación y resulte tan artificial).
  - **Agua**, donde se indica si el LiDAR es barimétrico o no. Si lo es podrá captar la superficie debajo del agua.

Ilustración 214. Configuración general del escaneo LiDAR. Apartados de intensidad y errores de terreno.

- **Factores de modificación de intensidad**. Algunos de los parámetros que sabemos que afectan a dicha información son la distancia del objeto, el ángulo de incidencia o la reflectancia del mismo. Este último parámetro depende más bien de la definición escena, mientras que los dos primeros parámetros se pueden modificar fácilmente en esta ventana. Se trata únicamente de dos pesos que, a mayor valor, mayor decremento producen en la intensidad captada (acercándose por tanto al cero).

- **Errores de terreno**. Dos errores vinculados a la pendiente del terreno son el error vertical y horizontal. En este caso se dispone de un intervalo claro en la bibliografía, razón por la cual aquí sólo se permite indicar si se desea incluir el error o no.

Una vez descrita la configuración general, podemos desplazarnos a las pestañas que controlan cada tipo de captura. En el caso de la captura aérea (Ilustración 215), se contempla una altura estática del vehículo no tripulado a lo largo de todo su recorrido, y por este motivo es posible seleccionar un valor de altura específico. Además, se incluye un área de dibujo donde el usuario podrá señalar una ruta de manera manual. Para aplicar esta ruta es necesario recurrir a la configuración

general. Nótese como esta ruta puede generar una nube mucho más incompleta que una ruta automática generada por la aplicación, en la cual se cubre todo el área.

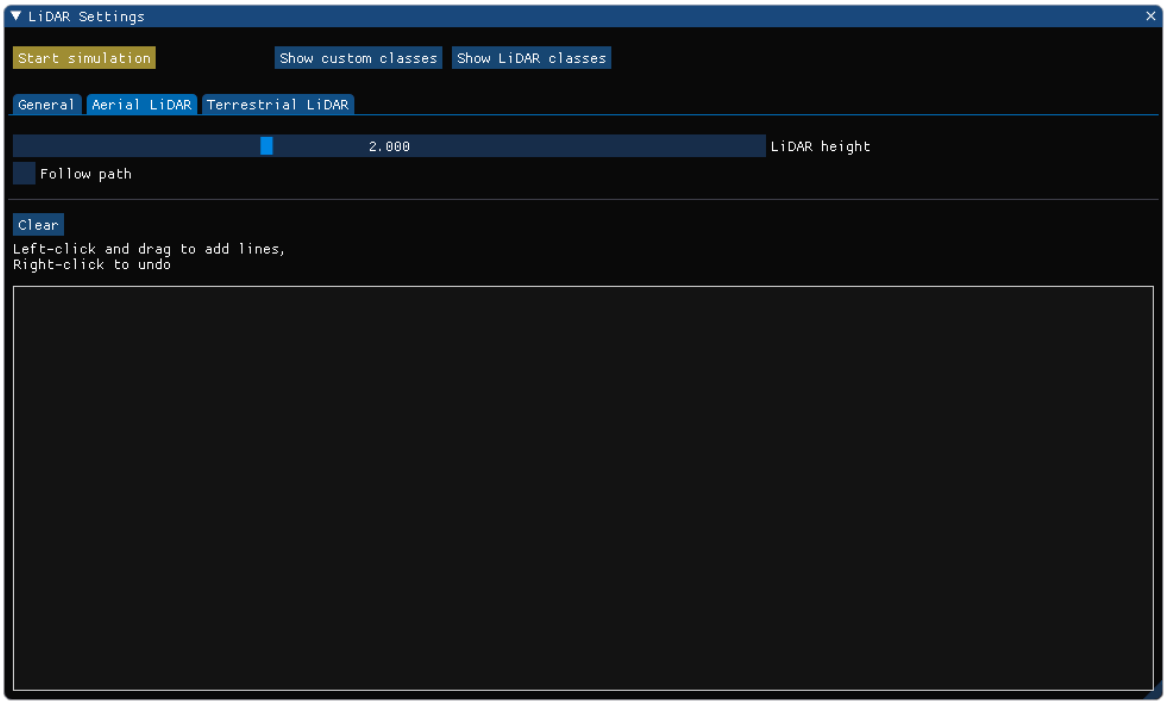


Ilustración 215. Configuración de captura aérea.

Para finalizar con la configuración de captura, se contempla un único parámetro que puede afectar a la captura terrestre, el cual es la posición estática que ocupa el dispositivo LiDAR. Los límites de estos valores dependen de la caja envolvente que define la escena.

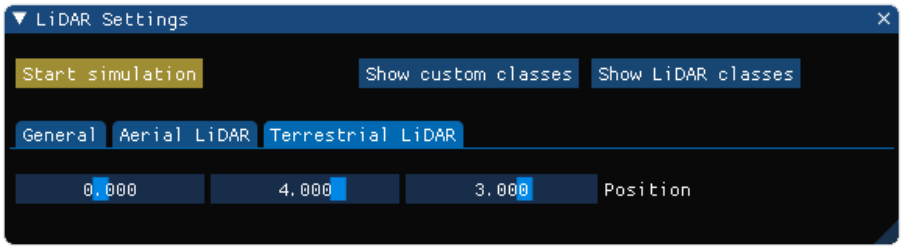


Ilustración 216. Configuración de captura terrestre.

En esta misma ventana, junto al botón que permite comenzar la simulación, podemos encontrar dos listados de clases representadas en el escenario actual. El primero de los listados (Ilustración 217) muestra aquellas clases definidas en la aplicación que no obedecen a un estándar concreto. Por tanto, presentan un nivel de detalle personalizado que podríamos incrementar o disminuir en función de las necesidades. Por ejemplo, *furniture* podría verse sustituido por un tipo de mueble u objeto más específico.

Por otro lado, la Ilustración 218 muestra todas las clases definidas en el estándar ASPRS para ficheros de nube de puntos, lo que no implica que todas ellas aparezcan en nuestra escena. Al conocer todas las clases existentes a priori, es posible asignarles un color desde el inicio de la aplicación. También cabe destacar que todos estos colores se aplican cuando se selecciona el *renderizado* de la nube de puntos en función del concepto semántico que representa cada punto.

Ilustración 217. Clases específicas de cada escenario.

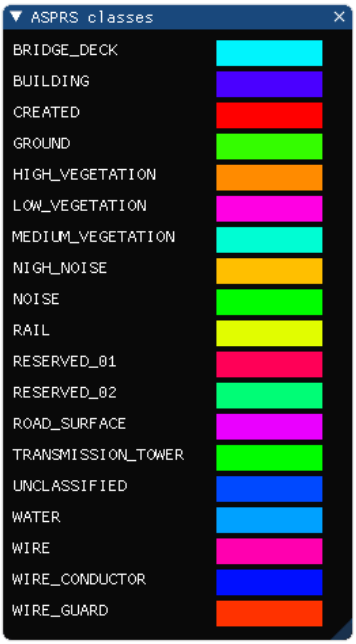


Ilustración 218. Listado de clases definidas en el estándar ASPRS, junto a su correspondiente color.

La última ventana relevante en la aplicación es el listado de objetos existentes en la escena (Ilustración 219), de tal manera que para todos ellos podemos modificar varios valores: reflectancia, brillo y opacidad. Reflectancia y brillo se aplican a la intensidad captada y al error de superficies similares a espejos, respectivamente, mientras que la opacidad no obedece a un parámetro de *rendering*, sino a la capacidad de un rayo emitido para atravesar una superficie. Esto es especialmente relevante en

la vegetación, donde los diferentes retornos generados no obedecen a refracciones, sino a la continuidad de una emisión (con la misma dirección con la que se genera).

Nótese como todos estos cambios, incluyendo los que puedan realizarse sobre la ventana que configura el proceso de captura, no se aplican de manera inmediata, sino que es necesario aplicarlos e iniciar una nueva simulación.

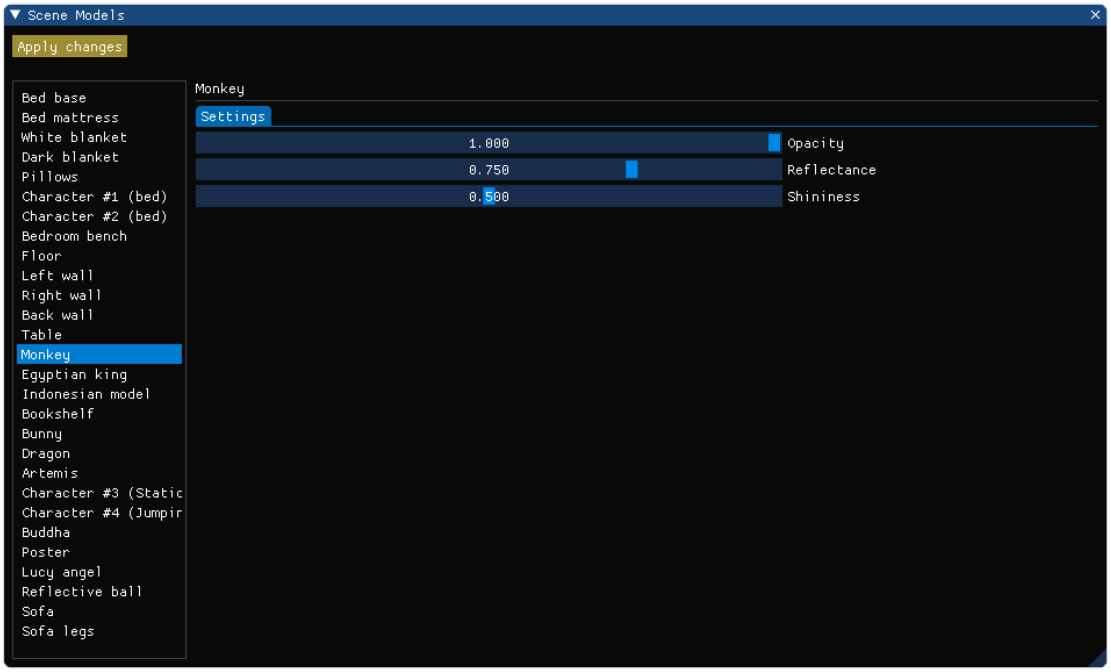


Ilustración 219. Listado de elementos situados en la escena representada.

Para finalizar este manual se muestran las ventanas correspondientes a las opciones *About the project* (Ilustración 220) y *Controls* (Ilustración 221), ambas situadas en *Help*, las cuales muestran una breve descripción del proyecto y los controles de la escena, respectivamente.

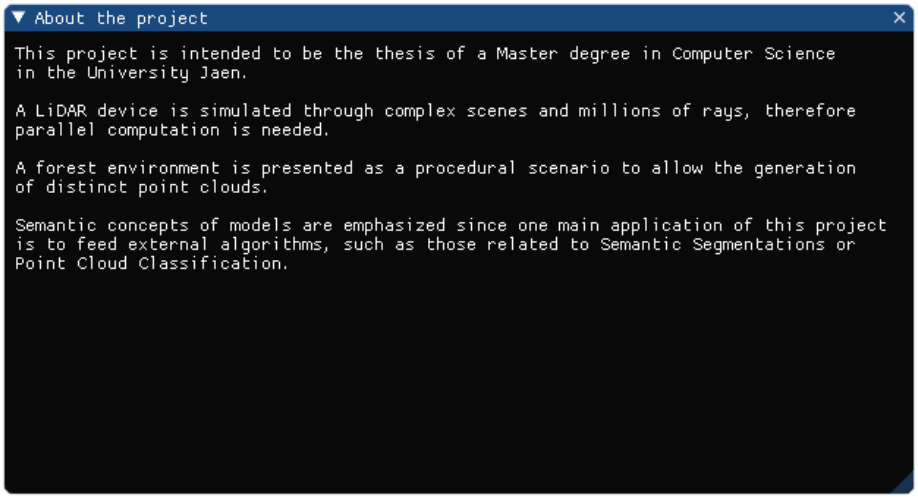


Ilustración 220. Descripción del proyecto.

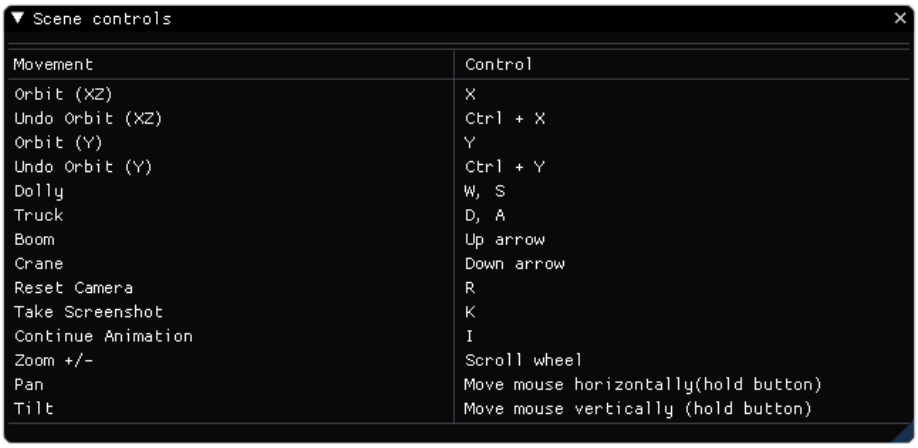


Ilustración 221. Controles de la escena de la aplicación.

6 ANEXO

En este capítulo se incluyen aquellos programas implementados y citados durante el desarrollo, de tal manera que se aíslan en este Anexo para evitar la introducción de código en el núcleo de este documento. También se describirán, brevemente, aquellas alternativas desarrolladas que finalmente se han descartado en la solución final, pero que son igualmente interesantes.

6.1 Rendering de triángulos (shader)

En este punto únicamente se pretende exponer el código contenido en el *vertex* y *fragment shader* encargado de *renderizar* un conjunto de triángulos en el escenario más básico. Es decir, a partir del programa aquí expuesto se podrán llevar a cabo las modificaciones que se crean convenientes para adaptarlo a cualquier otro modelo específico (agua, terreno, etc).

- ***Vertex shader***:

```
#version 450

// ********** PARAMETERS & VARIABLES ***********

layout(location = 0) in vec3 vPosition;
layout(location = 1) in vec3 vNormal;
layout(location = 2) in vec2 vTextCoord;
layout(location = 3) in vec3 vTangent;


// ------------- Light types ----------------
subroutine void lightType(const mat3 TBN);
subroutine uniform lightType lightUniform;

// ------------- Displacement type --------------
subroutine mat3 displacementType();
subroutine uniform displacementType displacementUniform;

// Lighting
uniform vec3 lightPosition;
uniform vec3 lightDirection;

// Matrices
uniform mat4 mModelView;
uniform mat4 mModelViewProj;
uniform mat4 mShadow;

// Vertex related
out vec3 position;
out vec3 normal;
out vec2 textCoord;
out vec4 shadowCoord;
```

```
// Lighting related
out vec3 lightPos;
out vec3 lightDir;
out vec3 vertexToLightDir;
out vec3 viewDir;


// ********* FUNCTIONS ************

//(Tangent, Binormal, Normal) matrix
mat3 getTBN()
{
	const vec3 tangent = vec3(mModelView * vec4(vTangent, 0.0f));
	const vec3 binormal = normalize(cross(normal, tangent));

	return transpose(mat3(tangent, binormal, normal));
}

// ------------- Lighting --------------

// Computes out parameters taking into account the type of light we're treating
subroutine(lightType)
void ambientLight(const mat3 TBN)
{
}

subroutine(lightType)
void pointLight(const mat3 TBN)
{
	lightPos = TBN * lightPosition;
	vertexToLightDir = normalize(TBN * (lightPosition - position));
	viewDir = normalize(TBN * -position);
}

subroutine(lightType)
void directionalLight(const mat3 TBN)
{
	viewDir = normalize(TBN * -position);
	vertexToLightDir = normalize(TBN * -lightDirection);
}

subroutine(lightType)
void spotLight(const mat3 TBN)
{
	pointLight(TBN);
	lightDir = normalize(TBN * lightDirection);
}

subroutine(lightType)
void rimLight(const mat3 TBN)
{
}

// ------------- Textures --------------

// Displacement textures; doesn’t mind wether it is real, such as displacement
mapping, or just fake, as bump mapping

subroutine(displacementType)
mat3 bumpMappingDisplacement()
{
	return getTBN();
```

```
}

subroutine(displacementType)
mat3 displacementMapping()
{
	return getTBN();
}

subroutine(displacementType)
mat3 noDisplacement()
{
	return mat3(1.0f);
}


void main()
{
	position = vec3(mModelView *  vec4(vPosition, 1.0f));
	gl_Position = mModelViewProj * vec4(vPosition, 1.0f);
	normal = vec3(mModelView * vec4(vNormal, 0.0f));
	shadowCoord = mShadow * vec4(vPosition, 1.0f);
	textCoord = vTextCoord;

	const mat3 TBN = displacementUniform();
	lightUniform(TBN);
}
```

- ***Fragment shader***:

```
#version 450

// ------------ Constraints ------------
const float EPSILON = 0.0000001f;

// ------------ Geometry ------------
in vec3 position;
in vec3 normal;
in vec2 textCoord;
in vec4 shadowCoord;

// ------------ Lighting ------------
in vec3 lightPos;
in vec3 lightDir;
in vec3 vertexToLightDir;
in vec3 viewDir;

// ----------- Materials ------------
uniform sampler2D texKadSampler;
uniform sampler2D texKsSampler;
uniform float materialScattering;	// Substitutes ambient lighting
uniform float shininess;

subroutine vec4 semiTransparentType(const vec4 color);
subroutine uniform semiTransparentType semiTransparentUniform;
uniform sampler2D texSemiTransparentSampler;

subroutine vec3 displacementType();
subroutine uniform displacementType displacementUniform;
uniform sampler2D texBumpSampler;

// ----------- Lighting -------------
```

```
subroutine vec3 lightType(const vec3 fragKad, const vec3 fragKs, const vec3
fragNormal, const float shadowDiffuseFactor, const float shadowSpecFactor);
subroutine uniform lightType lightUniform;

// Colors
uniform vec3 Ia;
uniform vec3 Id;
uniform vec3 Is;

// Spot light
uniform float cosUmbra, cosPenumbra;      // Angle interval where light fades out
uniform float exponentS;

// ----- Lighting attenuation -----
subroutine float attenuationType(const float distance);
subroutine uniform attenuationType attenuationUniform;

// Basic model
uniform float c1, c2, c3;

// Ranged distance model
uniform float minDistance, maxDistance;

// Pixar model
uniform float fMax;
uniform float distC, fC;
uniform float exponentSE;
uniform float k0, k1;

// ---------------- Shadows ---------------
subroutine void depthTextureType(out float shadowDiffuseFactor, out float
shadowSpecFactor);
subroutine uniform depthTextureType depthTextureUniform;

uniform float shadowMaxIntensity, shadowMinIntensity;        // Color range
uniform int minNeighborhood, maxNeighborhood;   // Index where matrix starts / ends
uniform int neighborhoodSize;
uniform sampler3D texOffset;
uniform sampler2DShadow texShadowMapSampler;

// ---------------- Bloom ---------------
subroutine vec3 bloomType();
subroutine uniform bloomType bloomUniform;
uniform sampler2D texBloomSampler;

layout (location = 0) out vec4 fColor;
layout (location = 1) out vec4 brightColor;

// ----------- Attenuation ------------
subroutine(attenuationType)
float basicAttenuation(const float distance)
{
	return min(1.0f / (c1 + c2 * distance + c3 * pow(distance, 2)), 1.0f);
}

subroutine(attenuationType)
float rangedAttenuation(const float distance)
{
	return clamp((maxDistance-distance)/(maxDistance-minDistance), 0.0f, 1.0f);
}

subroutine(attenuationType)
```

```
float pixarAttenuation(const float distance)
{
	float attenuation = mix(fMax * exp(k0 * pow(distance / distC, -k1)), fC *
			pow(distC / distance, exponentSE), step(distC, distance));

	return attenuation;
}

// ----------- Lighting ----------
vec3 getDiffuse(const vec3 fragKad, const float dotLN)
{
	return Id * fragKad * max((dotLN + materialScattering) / (1 +
				materialScattering), 0.0f);
}

vec3 getSpecular(const vec3 fragKs, const float dotHN)
{
	return Is * fragKs * pow(max(dotHN, 0.0f), shininess);
}

vec3 getDiffuseAndSpecular(const vec3 fragKad, const vec3 fragKs, const vec3
	fragNormal, const float shadowDiffuseFactor, const float shadowSpecFactor)
{
	const vec3 n = normalize(fragNormal);
	const vec3 l = normalize(vertexToLightDir);
	const vec3 v = normalize(viewDir);
	const vec3 h = normalize(v + l);			// Halfway vector
	const float dotLN = clamp(dot(l, n), -1.0f, 1.0f);	// Avoids Nan
	const float dotHN = dot(h, n);
	const vec3 diffuse = getDiffuse(fragKad, dotLN);
	const vec3 specular = getSpecular(fragKs, dotHN);

	return shadowDiffuseFactor * (diffuse + shadowSpecFactor * specular);
}

subroutine(lightType)
vec3 ambientLight(const vec3 fragKad, const vec3 fragKs, const vec3 fragNormal,
		const float shadowDiffuseFactor, const float shadowSpecFactor)
{
	return Ia * fragKad;
}

subroutine(lightType)
vec3 pointLight(const vec3 fragKad, const vec3 fragKs, const vec3 fragNormal, const
		float shadowDiffuseFactor, const float shadowSpecFactor)
{
	const vec3 diffuseSpecular = getDiffuseAndSpecular(fragKad, fragKs,
				fragNormal, shadowDiffuseFactor, shadowSpecFactor);
	const float distance = distance(lightPos, position);
	const float attenuation = attenuationUniform(distance);

	return attenuation * diffuseSpecular;
}

subroutine(lightType)
vec3 directionalLight(const vec3 fragKad, const vec3 fragKs, const vec3 fragNormal,
		const float shadowDiffuseFactor, const float shadowSpecFactor)
{
	return getDiffuseAndSpecular(fragKad, fragKs, fragNormal,
				shadowDiffuseFactor, shadowSpecFactor);
}
```

```
subroutine(lightType)
vec3 spotLight(const vec3 fragKad, const vec3 fragKs, const vec3 fragNormal, const
                    float shadowDiffuseFactor, const float shadowSpecFactor)
{
	const vec3 n = normalize(fragNormal);
	const vec3 l = normalize(vertexToLightDir);
	const vec3 v = normalize(viewDir);
	const vec3 d = normalize(lightDir);
	const vec3 h = normalize(v + l);                          // Halfway vector
	const float dotLN = clamp(dot(l, n), -1.0f, 1.0f);        // Avoids Nan
	const float dotHN = dot(h, n);
	const vec3 diffuse = getDiffuse(fragKad, dotLN);
	const vec3 specular = getSpecular(fragKs, dotHN);
	const float distance = distance(lightPos, position);
	const float attenuation = attenuationUniform(distance);
	float sf = 0.0f;
	const float dotLD = dot(-l, d);                           // Radial attenuation

	if (dotLD >= cosPenumbra)
	{
		sf = 1.0f;
	}
	else if (dotLD > cosUmbra)
	{
		sf = pow((dotLD - cosUmbra) / (cosPenumbra - cosUmbra), exponentS);

	}

	return sf * attenuation * shadowDiffuseFactor * (diffuse + shadowSpecFactor *
                                                      specular);
}

subroutine(lightType)
vec3 rimLight(const vec3 fragKad, const vec3 fragKs, const vec3 fragNormal, const
                    float shadowDiffuseFactor, const float shadowSpecFactor)
{
	const vec3 n = normalize(fragNormal);
	const vec3 v = normalize(-position);
	const float vdn = 1.0f - max(dot(v, n), 0.0f);

	return vdn * Ia;
}

// ----- Diffuse & specular -----
vec4 getKad()
{
	return texture(texKadSampler, textCoord);
}

vec4 getKs()
{
	return texture(texKsSampler, textCoord);
}

// ----- Semitransparent -----
subroutine(semiTransparentType)
vec4 semiTransparentTexture(const vec4 color)
{
	const vec4 semiTransparent = texture(texSemiTransparentSampler, textCoord);
	return vec4(mix(color.xyz, semiTransparent.xyz, semiTransparent.w), color.w);
}
```

```
subroutine(semiTransparentType)
vec4 noSemiTransparentTexture(const vec4 color)
{
	return color;
}

// ----- Displacement -----
subroutine(displacementType)
vec3 bumpMappingDisplacement()
{
	return vec3((2.0f * texture(texBumpSampler, textCoord)) - 1.0f);
}

subroutine(displacementType)
vec3 displacementMapping()
{
	return bumpMappingDisplacement();
}

subroutine(displacementType)
vec3 noDisplacement()
{
	return normal;
}

// ----- Shadows -------
subroutine(depthTextureType)
void shadow(out float shadowDiffuseFactor, out float shadowSpecFactor)
{
	shadowDiffuseFactor = 0.0f;

	for (int i = minNeighborhood; i <= maxNeighborhood; ++i)
	{
		for (int j = minNeighborhood; j <= maxNeighborhood; ++j)
		{
			shadowDiffuseFactor += textureProjOffset(texShadowMapSampler,
					shadowCoord, ivec2(i, j));
		}
	}

	shadowDiffuseFactor /= neighborhoodSize;		// Average
	shadowDiffuseFactor = shadowDiffuseFactor * (shadowMaxIntensity –
			shadowMinIntensity) + shadowMinIntensity;
	shadowSpecFactor = step(1.0f – EPSILON, shadowDiffuseFactor);
}

subroutine(depthTextureType)
void noShadow(out float shadowDiffuseFactor, out float shadowSpecFactor)
{
	shadowDiffuseFactor = 1.0f;		// No attenuation
	shadowSpecFactor = 1.0f;
}

// ----- Bloom ------
subroutine(bloomType)
vec3 bloomTexture()
{
	return texture(texBloomSampler, textCoord).rgb;
}

subroutine(bloomType)
vec3 noBloomTexture()
```

```
{
	return vec3(0.0f);
}

void main()
{
	const vec4 fragKad = semiTransparentUniform(getKad());
	const vec4 fragKs = getKs();
	const vec3 fragNormal = displacementUniform();

	float shadowDiffuseFactor, shadowSpecFactor;
	depthTextureUniform(shadowDiffuseFactor, shadowSpecFactor);

	const vec3 reflectionColor = lightUniform(fragKad.rgb, fragKs.rgb,
				fragNormal, shadowDiffuseFactor, shadowSpecFactor);

	fColor = vec4(reflectionColor, fragKad.w);
	brightColor = vec4(bloomUniform(), 1.0f);
}
```

6.2 Solución alternativa: agua

Una de las posibles alternativas a la representación de agua mediante un plano (más allá del procedimiento de *rendering* desarrollado) es la introducción de un océano, siempre y cuando garanticemos una morfología diferente del terreno (no tendría sentido con la estructura actual, donde se crean pequeños lagos). No se pretende describir en profundidad esta otra solución, sino sólo mostrar el procedimiento a muy alto nivel.

Un artículo de necesaria lectura en esta área es (Tessendorf, 2001), aunque es posible encontrar publicaciones más recientes acerca de enfoques orientados a GPGPU, sea uno de ellos (Flügge, 2017). El núcleo de este trabajo es la utilización de la función inversa de la transformada rápida de Fourier (IFFT, Inverse Fast Fourier Transform) y su cálculo en GPU.

El punto de partida serán cuatro texturas de *white noise*, las cuales se suponen independientes entre ellas. A partir de las dos primeras es posible recuperar dos números gaussianos aleatorios, cuya parte real e imaginaria se almacena en los componentes *red* y *green*, respectivamente, de la textura resultante $h_0(k)$ (Ilustración 222), sea k un vector dirección obtenido a partir de una función dependiente de una posición situada en $[-\frac{size}{2}, \frac{size}{2}]$. Mediante las otras dos texturas se obtiene esto mismo, pero empleando $-k$, en lugar de k.

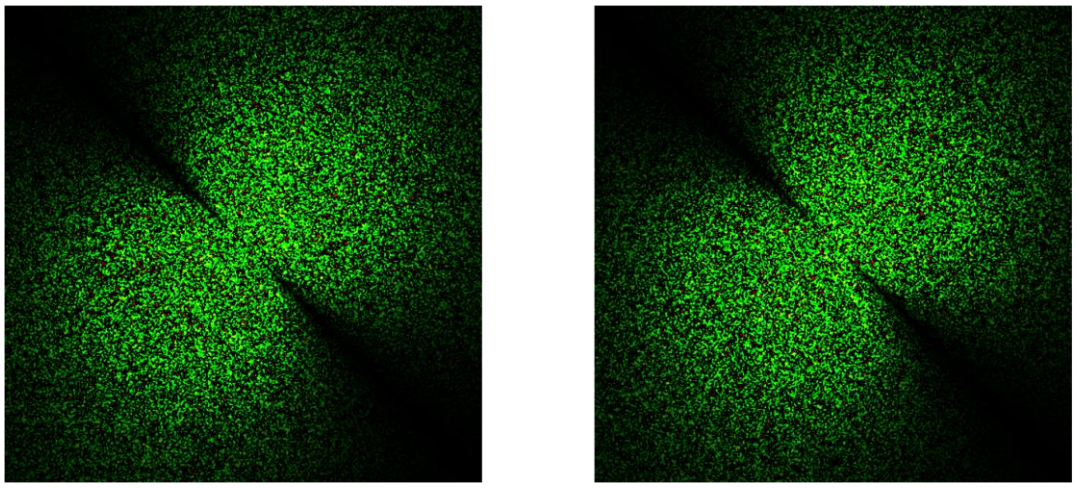


Ilustración 222. Texturas con el valor de $h_0(k)$ y $h_0(-k)$.

A partir de ambas texturas se obtienen las componentes dependientes de la variable tiempo de Fourier, generando así una nueva textura, $h(k,t)$, que nuevamente almacena la parte real e imaginaria en sus componentes *red* y *green* (Ilustración 223). La apariencia de la nueva textura es similar a la de las dos anteriores.

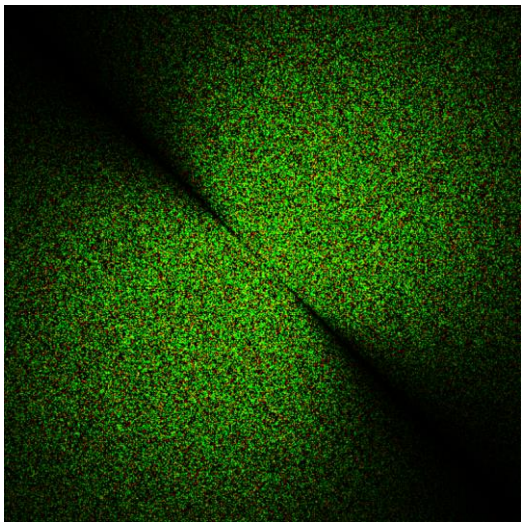


Ilustración 223. Textura con el valor de $h_0(k,t)$.

La manera en que se combinan los números complejos vendrá dada por una textura, *butterfly texture*, que puede precalcularse al comienzo de la aplicación. Se puede entender como un árbol, por lo que su anchura será $\log_2(N)$, sea N el tamaño de las texturas previas (aunque la Ilustración 224 expande la imagen a $N \times N$). Los píxeles que podemos encontrar en cada una de las $\log_2(N)$ etapas serán índices que nos permitirán conocer que dos números deberán combinarse. El proceso que aquí

se sigue se denomina como *ping-pong*; disponemos de dos texturas en las que iremos almacenando información alternativamente. El resultado se puede observar en la Ilustración 225, y a partir de este mismo, es posible hallar el resultado final de la IFFT, que será el mapa de altura de nuestro océano.

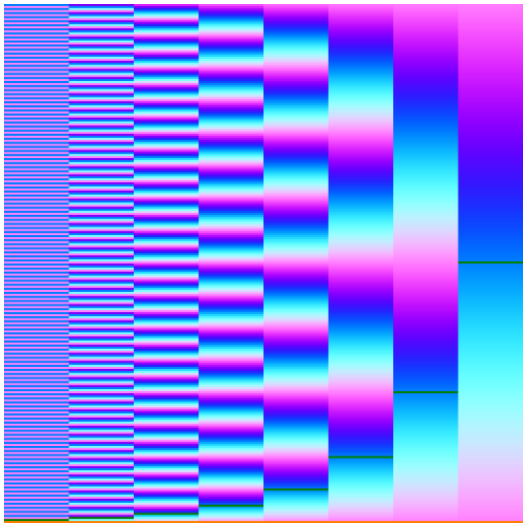


Ilustración 224. Butterfly texture.

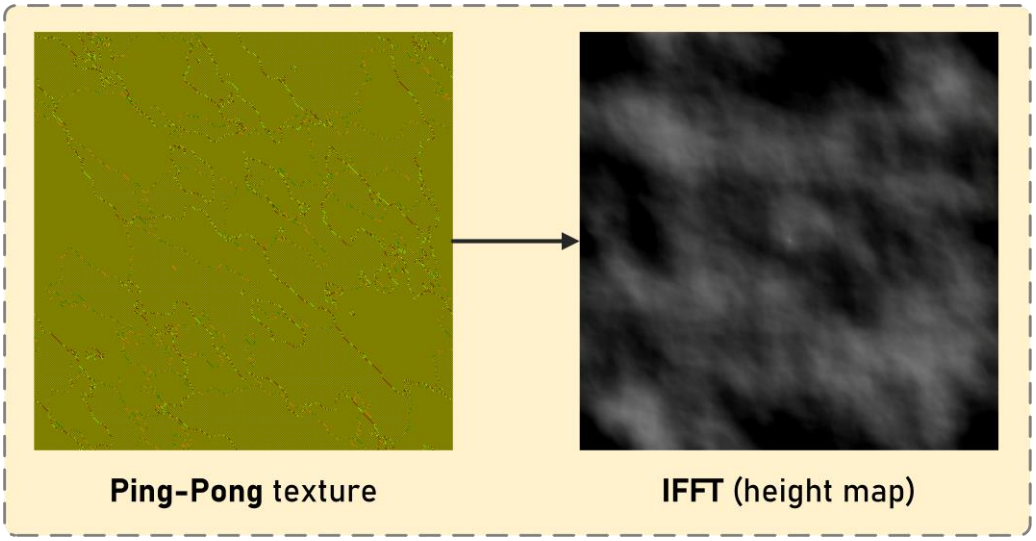


Ilustración 225. Textura obtenida del proceso de Ping-Pong (izquierda) y mapa de altura obtenido a partir de dicha textura (derecha).

Esta misma solución podría aplicarse a cualquier entorno procedural, como un terreno, y no únicamente a un océano, dado que la transformación final puede ajustarse a través de diversos parámetros. Por tanto, podría incluso introducirse en esta aplicación a través del terreno de la segunda escena.

7 DEFINICIONES Y ABREVIATURAS

7.1.1 Definiciones

Bloom	Efecto de post-procesamiento en el cual se simula la emisión de luz de una superficie mediante la introducción de brillo, el cual se expande en el área cercana a dicha superficie.
Cubemap	Mapa de un entorno, compuesto por seis caras (*up*, *down*, *left*, *right*, *front*, *back*). Se utiliza para resolver problemas tales como la inclusión de un *skybox* (fondo de la escena) o el *renderizado* de materiales con reflejo.
Deep learning	Algoritmos de *Machine Learning* que utilizan redes neuronales artificiales, esto es, estructuras con múltiples capas que intentan extraer características de más alto nivel que la información de entrada (*raw data*). Podemos encontrar métodos supervisados, semi-supervisados o no supervisados.
Displacement mapping	Técnica de modificación de la geometría de un objeto a partir de un mapa de altura. Por ejemplo, a partir de un plano se podría crear un terreno utilizando exclusivamente un mapa de altura en escala de grises (valores normalizados en el intervalo [0, 1]).
Framework	Es una abstracción en la cual se permite añadir código propio para generar una solución específica, todo ello a partir de un software con una funcionalidad genérica. Por tanto, es un entorno de software reusable que facilita el desarrollo de productos y soluciones software.
Geometría	Se utiliza el término geometría para hacer referencia a la información de los vértices de un modelo (posición, normal, etc).
Instancing	Técnica se dibujan varias instancias de un mismo objeto mediante una única llamada, eliminando así la necesidad de comunicar CPU con GPU para cada uno de esas instancias.
Octree	Estructura de datos 3D en forma de árbol donde cada nodo puede albergar hasta ocho descendientes. Cada uno de estos nodos se puede definir como un AABB, sea el nodo raíz el AABB de la escena completa.

Offscreen rendering	*Rendering* que suele utilizarse para recuperar texturas que podrán ser empleadas sobre el *rendering* que actúa directamente sobre la ventana. Este tipo de *rendering* no es directamente visible, sino que se suele utilizar para preparar recursos.
OpenGL	*Open source Graphics Library*. Interfaz de programación para la creación de gráficos 2D y 3D.
Path tracing	Técnica de *rendering* realista basada en el método de Monte Carlo. Al contrario que en *ray-tracing*, se parte de un punto de la superficie de un objeto y se calcula cuál es la iluminación que llega a la cámara.
Ray casting	Trazado de rayos, y por lo tanto, resolución de intersección rayo-superficie (se podría optimizar o adaptar dependiendo de cuál sea esta).
Ray tracing	Técnica de *rendering* realista que traza lo que se conocen como rayos a través de los píxeles del plano de la imagen, de tal forma que podrán intersectar con la geometría de la escena, aproximando con gran realismo la iluminación de ese punto.
Rendering	Generación a través de ordenador de una imagen a partir de modelos 2D o 3D. Dicha imagen podrá ser fotorrealista o no.
Shader	Programa informático que generalmente se ejecuta en GPU. También generalmente se utiliza para sombrear una escena, pero se extiende más allá de esta función: post-procesamiento, efectos especiales, cómputos paralelos, etc.
Topología	Descripción de la asociación de vértices de un modelo para posibilitar su representación. En este ámbito, es habitual que la topología se muestre como un conjunto de índices, que a su vez hacen referencia a una posición de las estructuras de datos que almacenan la geometría.

7.1.2 Abreviaturas

AABB	*Axis aligned bounding box*. Mínima caja capaz de envolver una primitiva o un conjunto de éstas (podrían formar cualquier objeto). Se alinean con los ejes X, Y y Z, de tal forma que no es necesariamente la caja de menor

	volumen, dado que no se alinea con la posible orientación de la primitiva.
API	Interfaz de programación de aplicaciones. Supone una capa de abstracción para que un software pueda acceder a ciertas funciones, procedimientos, etc.
CPU	*Central Processing Unit.* En este documento se utiliza este término para referirse a los programas que se ejecutan en esta unidad, en contraposición con el desarrollo de programas acelerados en GPU.
IDE	*Integrated Development Environment*. Aplicación que facilita al desarrollador la creación de software.
FBO	*Framebuffer Object.* Contexto de dibujo definido por el usuario, que permite realizar un *rendering* sin afectar la ventana principal de la aplicación. Por tanto, es común su utilización para generar texturas que puedan utilizarse en el *rendering* de la escena en el *framebuffer* por defecto.
GPU	*Graphic processing unit*. Coprocesador dedicado al procesamiento de gráficos u operaciones de coma flotante, para aligerar la carga de trabajo del procesador central en aplicaciones 3D.
LiDAR	*Light detection and ranging*. Dispositivo que permite medir la distancia a una superficie iluminándola con láser y midiendo, a través de un sensor, la luz reflejada.
SSBO	*Shader Storage Buffer Object. Buffer* para almacenar y recuperar datos, similar a un *Uniform Buffer Object*, pero puede ser mucho más grande y se puede escribir en él.
VAO	*Vertex Array Object.* Objeto OpenGL que contiene uno o más VBOs y uno o más IBOs.
VBO	*Vertex Buffer Object.* Objeto OpenGL que contiene la geometría de una malla de triángulos.

8 BIBLIOGRAFÍA

Ankerl, M. (9 de diciembre de 2009). *How to Generate Random Colors Programmatically*. Obtenido de Martin Ankerl: https://martin.ankerl.com/2009/12/09/how-to-create-random-colors-programmatically/

Bayer, H. T. (2015). *Implementation of a method for hydraulic erosion.* Munich: Technische Universität München.

Blakemore, E. (29 de julio de 2019). *Lasers are driving a revolution in archaeology*. Obtenido de National Geographic: https://www.nationalgeographic.com/culture/archaeology/lasers-lidar-driving-revolution-archaeology/

Dong, P., & Chen, Q. (2018). *LiDAR Remote Sensing and Applications.* CRC Press.

EMS Press. (2020). *Orthogonalization*. Obtenido de Encyclopedia of Mathematics: https://encyclopediaofmath.org/index.php?title=Orthogonalization

Flügge, F.-J. (16 de octubre de 2017). *Realtime GPGPU FFT Ocean Water Simulation.* Obtenido de Hamburg University of Technology: https://tore.tuhh.de/bitstream/11420/1439/1/GPGPU_FFT_Ocean_Simulation.pdf

Goddard Space Flight Center. (8 de mayo de 2009). *Risk Management Reporting.* Obtenido de http://everyspec.com/NASA/NASA-GSFC/GSFC-STD/download.php?spec=GSFC-STD-0002.020175.PDF#page=9

Karras, T. (26 de noviembre de 2012). *Thinking Parallel, Part II: Tree Traversal on the GPU*. Obtenido de NVIDIA Developer: https://developer.nvidia.com/blog/thinking-parallel-part-ii-tree-traversal-gpu/

Karras, T. (2012). *Thinking Parallel, Part III: Tree Construction on the GPU*. Obtenido de Nvidia Developer Blog: https://developer.nvidia.com/blog/thinking-parallel-part-iii-tree-construction-gpu/

Ken Beck et al. (2001). *Manifesto for Agile Software Development.* Obtenido de agilemanifesto: https://agilemanifesto.org/

Launch School. (2020). *Agile Planning: From Ideas to Story Cards.* Launch School.

LearnOpenGL. (2014). *Cubemaps*. Obtenido de LearnOpenGL: https://learnopengl.com/Advanced-OpenGL/Cubemaps

LearnOpenGL. (2017). *Bloom*. Obtenido de LearnOpenGL: https://learnopengl.com/Advanced-Lighting/Bloom

Lengyel, E. (2001). *Computing Tangent Space Basis Vectors for an Arbitrary Mesh.* Obtenido de Terathon Software 3D Graphics Library: http://www.terathon.com/code/tangent.html

Lighthouse3d. (2011). *OpenGL Timer Query*. Obtenido de Lighthouse3d: http://www.lighthouse3d.com/tutorials/opengl-timer-query/

Melin, M., C. Shapiro, A., & Glover-Kapfer, P. (2017). *Remote Sensing: LIDAR.* Obtenido de WWF: https://www.wwf.org.uk/sites/default/files/2019-04/Lidar-WWF-guidelines.pdf

Microsoft. (31 de mayo de 2018). *Retained Mode Versus Immediate Mode.* Obtenido de Windows Developer: https://docs.microsoft.com/en-us/windows/win32/learnwin32/retained-mode-versus-immediate-mode

Ministerio de Empleo y Seguridad Social. (6 de marzo de 2018). *XVII Convenio colectivo estatal de empresas de consultoría y estudios.* Obtenido de Boletín Oficial del Estado: https://www.boe.es/boe/dias/2018/03/06/pdfs/BOE-A-2018-3156.pdf

Morsy, S., Shaker, A., & El-Rabbany, A. (2018). Using Multispectral Airborne LiDAR Data for Land/Water Discrimination: A Case Study at Lake Ontario, Canada. *Applied Sciences*, 0-21.

Niall. (2019). *MPM Guide*. Obtenido de NiallTL: https://nialltl.neocities.org/articles/mpm_guide.html

Paris, A. (21 de junio de 2019). *Terrain Erosion on the GPU*. Obtenido de Axel Paris: https://perso.liris.cnrs.fr/aparis/public_html/posts/terrain_erosion.html

Pressman, R. (2010). *Ingeniería del Software.* McGraw-Hill.

Quilez, I. (s.f.). *Intersectors*. Recuperado el 2020, de Inigo Quilez: https://www.iquilezles.org/www/articles/intersectors/intersectors.htm

Revelles, J., Ureña, C., & Lastra, M. (2000). An Efficient Parametric Algorithm for Octree Traversal. *Journal of WSCG*.

Rodríguez, J. (2013). *GLSL Essentials.* Packt.

Schwaber, K., & Sutherland, J. (2017). *The Scrum Guide.*

Shore, J., & Chromatic. (2007). *The Art of Agile Development.* O'Reilly Media, Inc.

Talbot, J. (31 de julio de 2018). *What's right with risk matrices?* Recuperado el 30 de September de 2020, de Julian Talbot: https://en.wikipedia.org/wiki/Risk_matrix

Talwar, D. (s.f.). *Creating an urban simulation environment for teaching self-driving cars ways to interact with micro-mobility vehicles*. Obtenido de Deepak Talwar: https://www.deepaktalwar.me/lgsvl-micro-mobility

Tessendorf, J. (2001). *Simulating Ocean Water.* Obtenido de http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.161.9102&rep=rep1&type=pdf

The American Society for Photogrammetry & Remote Sensing. (9 de julio de 2019). *LAS Specification 1.4 - R15.* Obtenido de ASPRS: http://www.asprs.org/wp-content/uploads/2019/07/LAS_1_4_r15.pdf

Universidad del País Vasco. (s.f.). *El Modelo COCOMO.* Obtenido de http://www.sc.ehu.es/jiwdocoj/mmis/cocomo.htm

Visual Paradigm. (2020). *What is User Story?* Obtenido de Visual Paradigm: https://www.visual-paradigm.com/guide/agile-software-development/what-is-user-story/

Akenine-Möller, Tomas. 2018. *Real-Time Rendering*.

Akenine-Möllser, Tomas. 2001. “Fast 3D Triangle-Box Overlap Testing.” *Journal of Graphics Tools*. doi: 10.1080/10867651.2001.10487535.

Anon. 2019. *Ray Tracing Gems*.

Apetrei, Ciprian. 2014. “Fast and Simple Agglomerative LBVH Construction.” in *TPCG 2014 - Theory and Practice of Computer Graphics, Proceedings*.

Armeni, Iro, Ozan Sener, Amir R. Zamir, Helen Jiang, Ioannis Brilakis, Martin Fischer, and Silvio Savarese. 2016. “3D Semantic Parsing of Large-Scale Indoor Spaces.” in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*.

Badouel, Didier. 1990. “An Efficient Ray - Polygon Intersection.” in *Graphics Gems*.

Baldwin, Doug, and Michael Weber. 2016. “Fast Ray-Triangle Intersections by Coordinate Transformation.” *Journal of Computer Graphics Techniques (JCGT)*.

Barzel, Ronen. 1997. “Lighting Controls for Computer Cinematography.” *Journal of Graphics Tools*. doi: 10.1080/10867651.1997.10487467.

Deems, Jeffrey S., Thomas H. Painter, and David C. Finnegan. 2013. “Lidar Measurement of Snow Depth: A Review.” *Journal of Glaciology*.

Dong, Pinliang, and Qi Chen. 2017. *LiDAR Remote Sensing and Applications*.

Eisemann, Martin, Marcus Magnor, Thorsten Grosch, and Stefan Müller. 2007. “Fast Ray/Axis-Aligned Bounding Box Overlap Tests Using Ray Slopes.” *Journal of Graphics Tools*. doi: 10.1080/2151237x.2007.10129248.

Engelmann, Francis, Martin Bokeloh, Alireza Fathi, Bastian Leibe, and Matthias NieBner. 2020. “3D-MPA: Multi-Proposal Aggregation for 3D Semantic Instance Segmentation.”

Eric Freeman, Elisabeth Freeman, Bert Bates, Kathy Sierra. 2013. *Head First Design Patterns*.

Ericson, Christer. 2004. *Real-Time Collision Detection*.

Fang, Jin, Dingfu Zhou, Feilong Yan, Tongtong Zhao, Feihu Zhang, Yu Ma, Liang Wang, and Ruigang Yang. 2020. “Augmented LiDAR Simulator for Autonomous Driving.” *IEEE Robotics and Automation Letters*. doi: 10.1109/LRA.2020.2969927.

Goodwin, N. R., N. C. Coops, and D. S. Culvenor. 2007. “Development of a Simulation Model to Predict LiDAR Interception in Forested Environments.” *Remote Sensing of Environment*. doi: 10.1016/j.rse.2007.04.001.

Harris, Mark, Shubhabrata Sengupta, and John D. Owens. 2007. “Parallel Prefix Sum (Scan) with CUDA Mark.” *Gpu Gems 3*.

Hendrich, J., A. Pospíšil, D. Meister, and J. Bittner. 2019. “Ray Classification for Accelerated BVH Traversal.” *Computer Graphics Forum*. doi: 10.1111/cgf.13769.

Homepage, Developer Site, Developer News Homepage, Developer Login, Registered Developer, Developer Tools, Events Calendar, Newsletter Sign-up, Legal Information, Site Feedback, Recent Documents, Speedtree Rendering, and Variance Shadow Maps. 2009. “GPU Gems 3.” *Acm Transactions On Graphics*.

Hu, Yingsong, Weijian Wang, Dan Li, Qingzhi Zeng, and Yunfei Hu. 2019. “Parallel BVH Construction Using Locally Density Clustering.” *IEEE Access*. doi: 10.1109/ACCESS.2019.2932151.

Hu, Yuanming, Yu Fang, Ziheng Ge, Ziyin Qu, Yixin Zhu, Andre Pradhana, and Chenfanfu Jiang. 2018. “A Moving Least Squares Material Point Method with Displacement Discontinuity and Two-Way Rigid Body Coupling.” *ACM Transactions on Graphics*. doi: 10.1145/3197517.3201293.

Hwu, Wen Mei W. 2012. *GPU Computing Gems Jade Edition*.

Jeong, Jinyong, and Ayoung Kim. 2018. “LiDAR Intensity Calibration for Road Marking Extraction.” in *2018 15th International Conference on Ubiquitous Robots, UR 2018*.

Jiang, Chenfanfu, Craig Schroeder, Andrew Selle, Joseph Teran, and Alexey Stomakhin. 2015. “The Affine Particle-In-Cell Method.” in *ACM Transactions on Graphics*.

Jiang, Chenfanfu, Craig Schroeder, Joseph Teran, Alexey Stomakhin, and Andrew Selle. 2016. “The Material Point Method for Simulating Continuum Materials.” in *ACM SIGGRAPH 2016 Courses, SIGGRAPH 2016*.

Jiménez, Juan J., Carlos J. Ogáyar, José M. Noguera, and Félix Paulano. 2014. “Performance Analysis for GPU-Based Ray-Triangle Algorithms.” in *GRAPP 2014 - Proceedings of the 9th International Conference on Computer Graphics Theory and Applications*.

Karambakhsh, Ahmad, Bin Sheng, Ping Li, Po Yang, Younhyun Jung, and David Dagan Feng. 2020. “VoxRec: Hybrid Convolutional Neural Network for Active 3D Object Recognition.” *IEEE Access*. doi: 10.1109/ACCESS.2020.2987177.

Kemerer, Chris F. 1987. “An Empirical Validation of Software Cost Estimation Models.” *Communications of the ACM*. doi: 10.1145/22899.22906.

Koch, Sebastian, Albert Matveev, Zhongshi Jiang, Francis Williams, Alexey Artemov, Evgeny Burnaev, Marc Alexa, Denis Zorin, and Daniele Panozzo. 2019. “ABC: A Big Cad Model Dataset for Geometric Deep Learning.” in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*.

Korzeniowska, Karolina. 2012. “Modelling of Water Surface Topography on the Digital Elevation Models Using LiDAR Data.” *Proceedings of the AGILE’2012 International Conference on Geographic Information Science*.

Lauterbach, C., M. Garland, S. Sengupta, D. Luebke, and D. Manocha. 2009. “Fast BVH Construction on GPUs.” *Computer Graphics Forum*. doi: 10.1111/j.1467-8659.2009.01377.x.

Lee, Seongjo, Dahyeon Kang, Seoungjae Cho, Sungdae Sim, Yong Woon Park, Kyhyun Um, and Kyungeun Cho. 2015. “LIDAR Simulation Method for Low-Cost Repetitive Validation.” *Lecture Notes in Electrical Engineering*. doi: 10.1007/978-981-10-0281-6_34.

Legleiter, Carl J., Brandon T. Overstreet, Craig L. Glennie, Zhigang Pan, Juan Carlos Fernandez-Diaz, and Abhinav Singhania. 2016. “Evaluating the Capabilities of the CASI Hyperspectral Imaging System and Aquarius Bathymetric LiDAR for Measuring Channel Morphology in Two Distinct River Environments.” *Earth Surface Processes and Landforms*. doi: 10.1002/esp.3794.

Li, W., C. W. Pan, R. Zhang, J. P. Ren, Y. X. Ma, J. Fang, F. L. Yan, Q. C. Geng, X. Y. Huang, H. J. Gong, W. W. Xu, G. P. Wang, D. Manocha, and R. G. Yang. 2019. “AADS: Augmented Autonomous Driving Simulation Using Data-Driven Algorithms.” *Science Robotics*. doi: 10.1126/scirobotics.aaw0863.

Lovell, J. L., D. L. B. Jupp, G. J. Newnham, N. C. Coops, and D. S. Culvenor. 2005. “Simulation Study for Finding Optimal Lidar Acquisition Parameters for Forest Height Retrieval.” *Forest Ecology and Management*. doi: 10.1016/j.foreco.2004.07.077.

MacDonald, J. David, and Kellogg S. Booth. 1990. “Heuristics for Ray Tracing Using Space Subdivision.” *The Visual Computer*. doi: 10.1007/BF01911006.

Majercik, Alexander, Cyril Crassin, Peter Shirley, and Morgan Mcguire. 2018. “A Ray-Box Intersection Algorithm and Efficient Dynamic Voxel Rendering.” *Journal of Computer Graphics Techniques A Ray-Box Intersection Algorithm*.

Mandelbrot, Benoit B., and John A. Wheeler. 1983. “The Fractal Geometry of Nature.” *American Journal of Physics*. doi: 10.1119/1.13295.

Mandlburger, Gottfried, Christoph Hauer, Martin Wieser, and Norbert Pfeifer. 2015. “Topo-Bathymetric LiDAR for Monitoring River Morphodynamics and Instream Habitats-A Case Study at the Pielach River.” *Remote Sensing*. doi: 10.3390/rs70506160.

Meister, Daniel, and Jiri Bittner. 2018. “Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction.” *IEEE Transactions on Visualization and Computer Graphics*. doi: 10.1109/TVCG.2017.2669983.

Meister, Daniel, and Jiří Bittner. 2016. “Parallel BVH Construction Using K-Means Clustering.” *Visual Computer*. doi: 10.1007/s00371-016-1241-0.

Milioto, Andres, Philipp Lottes, and Cyrill Stachniss. 2018. “Real-Time Semantic Segmentation of Crop and Weed for Precision Agriculture Robots Leveraging Background Knowledge in CNNs.” in *Proceedings - IEEE International Conference on Robotics and Automation*.

Moller, Tomas, and Ben Trumbore. 1998. *Fast, Minimum Storage Ray-Triangle Intersection*.

Ogayar-Anguita, Carlos J., Antonio J. Rueda-Ruiz, Rafael J. Segura-Sanchez, Miguel Diaz-Medina, and Angel L. Garcia-Fernandez. 2020. “A GPU-Based Framework for Generating Implicit Datasets of Voxelized Polygonal Models for the Training of 3D Convolutional Neural Networks.” *IEEE Access*. doi: 10.1109/ACCESS.2020.2965624.

Perlin, Ken. 1985. “An Image Synthesizer.”

Perlin, Ken. 2002. “Improving Noise.” *ACM Transactions on Graphics*. doi: 10.1145/566570.566636.

Ram, Daniel, Theodore Gast, Chenfanfu Jiang, Craig Schroeder, Alexey Stomakhin, Joseph Teran, and Pirouz Kavehpour. 2015. “A Material Point Method for Viscoelastic Fluids, Foams and Sponges.” in *Proceedings - SCA 2015: 14th ACM SIGGRAPH / Eurographics Symposium on Computer Animation*.

Reeves, William T., David H. Salesinf, and Robert L. Cook. 1987. “Rendering Antialiased Shadows with Depth Maps.” in *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1987*.

Rosique, Francisca, Pedro J. Navarro, Carlos Fernández, and Antonio Padilla. 2019. “A Systematic Review of Perception System and Simulators for Autonomous Vehicles Research.” *Sensors (Switzerland)*.

Royo, Santiago, and Maria Ballesta-Garcia. 2019. “An Overview of Lidar Imaging Systems for Autonomous Vehicles.” *Applied Sciences (Switzerland)*. doi: 10.3390/app9194093.

Sans, Francisco, and Rhadamés Carmona. 2017. “A Comparison between GPU-Based Volume Ray Casting Implementations: Fragment Shader, Compute Shader, OpenCL, and CUDA.” *CLEI Electronic Journal*. doi: 10.19153/cleiej.20.2.7.

Singh, Rahul Dev, Ajay Mittal, and Rajesh K. Bhatia. 2019. “3D Convolutional Neural Network for Object Recognition: A Review.” *Multimedia Tools and Applications*. doi: 10.1007/s11042-018-6912-6.

Sobel, Irwin, and Gary Feldman. 2015. “An Isotropic 3x3 Image Gradient Operator.” *Stanford Artificial Intelligence Project (SAIL)*.

Stomakhin, Alexey, Craig Schroeder, Lawrence Chai, Joseph Teran, and Andrew Selle. 2013. “A Material Point Method for Snow Simulation.” *ACM Transactions on Graphics*. doi: 10.1145/2461912.2461948.

Su, Hu, Rui Wang, Kaixin Chen, and Yizhan Chen. 2019. “A Simulation Method for LIDAR of Autonomous Cars.” in *IOP Conference Series: Earth and Environmental Science*.

Ullrich, Andreas, and Martin Pfennigbauer. 2019. “Advances in Lidar Point Cloud Processing.”

Wang, Cheng, Ming Cheng, Ferdous Sohel, Mohammed Bennamoun, and Jonathan Li. 2019. “NormalNet: A Voxel-Based CNN for 3D Object Classification and Retrieval.” *Neurocomputing*. doi: 10.1016/j.neucom.2018.09.075.

Wolff, David. 2013. *OpenGL 4.0 Shading Language Cookbook*.

Xie, Yuxing, Jiaojiao Tian, and Xiao Xiang Zhu. 2019. “A Review of Point Cloud Semantic Segmentation.” *ArXiv.Org*.

Yates, Marissa L., R. T. Guza, Roberto Gutierrez, and Richard Seymour. 2008. “A Technique for Eliminating Water Returns from Lidar Beach Elevation Surveys.” *Journal of Atmospheric and Oceanic Technology*. doi: 10.1175/2008JTECHO561.1.

Yue, Xiangyu, Bichen Wu, Sanjit A. Seshia, Kurt Keutzer, and Alberto L. Sangiovanni-Vincentelli. 2018. “A LiDAR Point Cloud Generator: From a Virtual World to Autonomous Driving.” in *ICMR 2018 - Proceedings of the 2018 ACM International Conference on Multimedia Retrieval*.