FACULTAD DE INFORMÁTICA

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

Mención en Tecnologías de la Información

Proyecto de desarrollo en investigación

Diseño e implementación de un motor gráfico interactivo basado en voxels

Autor: Víctor Castro Cabado

Director: Diego Andrade Canosa

Director: Emilio José Padrón González

A Coruña, Febrero 2018

A mi preciosa calamidad

Resumen: El presente documento recoge la memoria de desarrollo de un motor gráfico interactivo, una aplicación capaz de sintetizar imágenes en tiempo real con una aproximación híbrida que posibilita tanto la representación de modelos tridimensionales (poligonales) como la visualización de datos volumétricos mediante voxels.

Durante la creación de dicho motor se ha pretendido aprender y sobretodo experimentar con diferentes técnicas gráficas cuyos resultados han guiado el desarrollo del sistema que finalmente se presenta.

El software, apodado "Unnamed Engine", implementa un conjunto de los subsistemas que habitualmente podemos encontrar como parte de la arquitectura de un motor de juego (game engine) moderno: gestores de recursos, renderizado diferido, culling, iluminación, sombreado, efectos de post-procesado y herramientas de *debugging*. En otro orden de cosas, la aplicación se nutre del concepto de voxel y actualmente cuenta con mecanismos para el tratamiento y visualización de los mismos, así como capacidad de voxelización de escenas, rendering volumétrico y generación procedural.

El motor ha sido desarrollado principalmente sobre el estándar C++11 siguiendo un paradigma orientado a objetos, junto con la API moderna de OpenGL (v4.5) y otras librerías de utilidad.

Si bien la herramienta ha alcanzado un tamaño considerable a lo largo del último año, cabe destacar que el software presentado no constituye un producto final, sino más bien el núcleo de un proyecto a más largo plazo. El sistema ha sido diseñado y desarrollado haciendo especial énfasis en su escalabilidad, facilitando la continuación y crecimiento del proyecto en el futuro, lo que ha sido uno de los principales objetivos a conseguir.

Lista de palabras clave:

Unnamed Engine, voxel, voxelization, rendering engine, rendering pipeline, shading, deferred shading, post-processing, interactive rendering, volume rendering, real time rendering, game architecture, rasterization, raytracing, raymarching.

Herramientas y tecnologías empleadas:

- **Hardware:**
  - **Desarrollo**: Portátil Intel Core i3 @ 2.0GHz e Intel HD Graphics 5500
  - **Pruebas**: PC Intel Core i3 @ 4.2GHz y aceleradora gráfica NVIDIA Geforce GTX1050
- **Software:**
  - S.O. desarrollo y pruebas: GNU/Linux Debian 9 (Linux Kernel 4.9.0 amd64)
  - Lenguajes: C++11, GLSL
  - Librerías: GLEW, GLFW3, GLM, SOIL, Assimp, Libnoise, ImGui
  - Compilación: GCC, Makefile
  - Otras herramientas: Git, Doxygen, Gdb, Valgrind, RenderDoc

Índice general

Índice de figuras

Capítulo 1

INTRODUCCIÓN

La disciplina que trata la generación de gráficos por ordenador puede considerarse un subcampo dentro de las ciencias de la computación que se encuentra íntimamente relacionado con otras disciplinas como la Visión Artificial, el Procesamiento de Imágenes, Aprendizaje máquina y Visualización de datos, entre otros. Los gráficos generados por ordenador son una parte muy importante de la evolución de la informática en las últimas décadas, ya que aportan a los usuarios la capacidad de interactuar con los dispositivos de una forma cada vez más intuitiva, empezando por clásicas interfaces bidimensionales hasta complejos entornos interactivos e inmersivos. Tecnologías como la Realidad Virtual o la Realidad Aumentada se encuentran actualmente en pleno crecimiento y los primeros productos comerciales ya están al alcance de muchos, abriéndose así un nuevo marco para la experimentación, reinvención y creación de nuevas herramientas.

Ya sea orientado al ocio, la ingeniería o la medicina, los sistemas gráficos han evolucionado lo suficiente como para convertirse en un área de estudio muy amplia y compleja que engloba multitud de arquitecturas, técnicas, algoritmos, trucos y *hacks*, que pretenden, para cada caso y según las necesidades, una mejora en la capacidad, el rendimiento, la velocidad o la calidad visual de las imágenes resultantes.

El software que aquí se presenta ha sido diseñado como parte de un motor de juego moderno y por tanto, y en cuanto a gráficos se refiere, tiene una orientación clara: la

búsqueda de equilibrio entre rendimiento y calidad visual que permita una interacción fluida con el sistema y pueda ser utilizado en ordenadores domésticos de gama media.

1.1. Contexto y antecedentes

En el contexto de los sistemas gráficos, el uso principal del *voxel* suele relacionarse con la visualización de imágenes biomédicas, un elemento especialmente útil para la reconstrucción de volúmenes a partir de una sucesión de imágenes transversales, como las resultantes de una resonancia magnética; pero también es un concepto de utilidad a la hora de trabajar con nubes de puntos generadas a través de escáneres LIDAR, visión artificial o efectos especiales en la industria audiovisual [1].

Los primeros ejemplos en el sector de los videojuegos que hacen uso del voxel, a principios de los años 90, son aquellos que lo utilizaron como una fórmula para la creación de terreno con un mayor nivel de detalle del que permitían las aproximaciones poligonales sobre el hardware de la época. Sin embargo, la evolución y los avances en computación gráfica desde entonces han permitido que este concepto se expanda. Actualmente una gran variedad de motores lo incluyen de una forma u otra, ya sea por su aspecto estético, como pieza angular que sustenta la jugabilidad, enfocado a la generación procedural de contenido [2], orientado a la visualización de grandes conjuntos de datos [3], o bien como potente herramienta para el cálculo de complejos efectos gráficos en tiempo real [4] [5].

1.2. Motivación y objetivos

El interés personal del autor por el funcionamiento de los sistemas gráficos a bajo nivel, la arquitectura, métodos y técnicas aplicadas en el desarrollo de motores gráficos e interactivos, constituyen la principal motivación del proyecto. En el contexto planteado, el diseño de un sistema de estas características se presenta para un alumno de último curso de grado como un reto que combina múltiples disciplinas en las que se ha profundizado, tanto

dentro como fuera de las aulas, y que culmina, de algún modo, con la implementación presentada.

A raíz del estudio de diversas alternativas que se están poniendo en práctica en relación a la aplicación del voxel en los sistemas interactivos, se han realizado múltiples iteraciones de desarrollo, con una duración aproximada de 2 meses, en las que paulatinamente se ha estudiado, experimentado, implementado y ampliado la funcionalidad del software que se presenta, su resultado visual, y el rendimiento.

Los objetivos principales que se han tratado de cumplir a lo largo de la elaboración de este Trabajo Fin de Grado son:

1. Aprendizaje práctico de los modelos de programación gráfica moderna y las diferentes técnicas de rendering, tales como *ray-tracing*, *ray-marching* y el clásico rasterizado poligonal.

2. Estudio del *pipeline* programable de la API de OpenGL moderna y el lenguaje GLSL para la creación de programas de sombreado (*shaders*) enfocados a la ejecución en GPU.

3. Aprendizaje y profundización en el uso de características avanzadas del lenguaje C++11 y otras librerías de utilidad para el proyecto.

4. Estudio y puesta en práctica de algoritmos y métodos de renderizado en tiempo real, tales como iluminación avanzada o efectos de post-procesado

5. Estudio del voxel como herramienta genérica y su utilización en todo tipo de sistemas. Aplicación de los conceptos mencionados en el motor desarrollado.

6. Desarrollo de los cimientos de un *game engine* que pueda ser ampliado en el futuro.

1.3. Estructura de la memoria

En la primera mitad de esta memoria comenzaremos con un repaso de la planificación y metodología utilizadas a lo largo del proyecto, seguido de una introducción a las bases del funcionamiento de los sistemas gráficos modernos. Explicaremos los fundamentos tecnológicos implicados, realizando hincapié en la estructura y funcionamiento de la API de OpenGL y en el voxel como fórmula para la representación regular de escenas, visualización de datos clínicos, nubes de puntos o generación procedural.

En la segunda mitad de este documento se realizará una introducción a las características de un motor de renderizado, explicando los principales recursos, algoritmos y técnicas necesarias. Continuaremos con una revisión de la arquitectura global del motor implementado y desglosaremos sus diferentes componentes, explicando capacidades, flaquezas y posibles mejoras de los mismos. Finalizaremos con una visión del *pipeline* completo y la orquestación de sus componentes para la obtención de la imagen final.

La memoria presentada la constituyen un total de 8 capítulos, siendo esta introducción el primero de ellos, y en los que se desarrollan los siguientes contenidos:

Capítulo 2. Planificación y Metodología
En este capítulo se detalla la organización, planificación y metodología empleados en el desarrollo de este proyecto. Se realiza una introducción a los métodos de desarrollo ágil junto con la justificación de su aplicación a este trabajo. Se incluye un histórico de las actividades llevadas a cabo en forma de diagrama de Gantt. Se incluye también una descripción de algunas herramientas empleadas para el seguimiento, junto con algunas estadísticas del propio desarrollo.

Capítulo 3. Fundamentos de Computación Gráfica
El tercer capítulo realiza una introducción a la generación de imágenes por ordenador y se exponen los fundamentos que sustentan la computación gráfica. Se explican las características principales de un *rendering pipeline* y se repasan brevemente algunas de las

técnicas más utilizadas. También se incluyen en este capítulo los modelos matemáticos básicos involucrados en los sistemas gráficos.

Capítulo 4. Fundamentos Tecnológicos

En el cuarto capítulo se detallan las características principales de la API de OpenGL y su funcionamiento, continuando con una introducción al renderizado volumétrico, donde se presentan brevemente las principales técnicas existentes. El capítulo finaliza hablando de la primitiva voxel y describiendo algunas de las técnicas y usos más interesantes en el ámbito de los sistemas gráficos en tiempo real.

Capítulo 5. Estructura del motor de renderizado

En este capítulo tratamos las características propias de los motores de renderizado. A continuación, realizamos una revisión completa del análisis y diseño del sistema propuesto en este trabajo, realizando un pequeño recorrido por los diferentes módulos que lo componen.

Capítulo 6. Rendering Pipeline

A lo largo de este capítulo describimos todas las diferentes etapas del *pipeline* gráfico, explicando cómo el flujo de datos entre ellas compone la imagen final en la pantalla. Detallamos, una a una las diferentes técnicas y algoritmos aplicados, de tal forma que se obtenga una visión completa de todos los procesos implicados en la síntesis de cada fotograma.

Capítulo 7. Voxelización y Voxel Rendering

El penúltimo capítulo presenta una síntesis de los aspectos más relevantes relativos a la integración de los voxels como parte del *pipeline* gráfico y los posibles usos futuros que dicha representación puede llegar a proveer.

Capítulo 8. Conclusiones

El último capítulo se dedica a resumir el trabajo realizado en este proyecto y las conclusiones a las que se ha llegado con el mismo. Se incluye también el trabajo futuro que se espera poder llevar a cabo con la base del desarrollo actual del motor.

Capítulo 2

PLANIFICACIÓN Y METODOLOGÍA

En este capítulo se explican la organización y metodología empleados para el desarrollo del software que se presenta en este Trabajo Fin de Grado. Debido a la inexperiencia previa del autor en materia de programación gráfica, unido al carácter investigador del propio proyecto, no resultaba sencilla la aplicación de metodologías de planificación clásicas ya que se desconocía gran parte de los componentes necesarios para el sistema final. Es por ello que se optó por el uso de un sistema ágil de desarrollo que permitiese, de forma incremental, el crecimiento del software a medida que se fijaban nuevos objetivos.

A lo largo de este capítulo se introducen, de forma breve, algunos de los conceptos relacionados con las metodologías de desarrollo ágil y su aplicación en la planificación del presente proyecto.

2.1. Desarrollo ágil de software

Dentro de la disciplina de la ingeniería del software, el desarrollo ágil de software abarca los métodos y metodologías que se utilizan como la base de un desarrollo incremental

a través de continuas iteraciones.

El objetivo principal de estos métodos consiste en desarrollar, en lapsos cortos de tiempo (de entre una y cuatro semanas), pequeños prototipos funcionales de la aplicación que se está construyendo, permitiendo, de esta manera, analizar y evaluar el resultado para fijar nuevos objetivos y prioridades en la siguiente iteración.

Aunque estos métodos puedan ser vistos como una nueva forma de volver a las prácticas observadas en los primeros años del desarrollo software (*code and fix*), el desarrollo ágil proporciona un contexto conveniente en desarrollos en los que los objetivos no están del todo determinados, pudiendo ser reconducidos minimizando riesgos.

2.1.1. Agile Manifesto

En el año 2001, el término '*ágil*' referido al desarrollo software fue popularizado a través del denominado *Manifesto for Agile Software Development* o simplemente *Agile Manifesto*, un documento firmado por diecisiete desarrolladores software que se reunieron en el *Snowbird Resort* en el estado de *Utah* para discutir acerca de sus métodos de trabajo, lo que sentó las bases de lo que ahora se conoce como Desarrollo ágil de Software.

El manifiesto en si se sostiene sobre cuatro pilares fundamentales:

- **Individuos e interacciones:** El desarrollo ágil considera que la auto-organización y la motivación son muy importantes, al igual que las interacciones entre compañeros de equipo.

  Los equipos de desarrollo ágil suelen localizarse en oficinas abiertas donde todos los desarrolladores se mantienen al tanto del trabajo del resto o trabajan de forma conjunta (por ejemplo en parejas en la misma estación de trabajo).

- **Software funcionando:** El desarrollo ágil prioriza la capacidad de producir *demos* funcionales frente a un desarrollo pesado plagado de documentación técnica.

- **Colaboración con el cliente:** Gracias a la capacidad de adaptación de nuevos re-

quisitos u objetivos, se valora la colaboración estrecha con el cliente de forma que el software se construye, iteración a iteración, a medida.

- **Respuesta ante el cambio:** Finalmente, el desarrollo ágil minimiza riesgos gracias a una gran capacidad de adaptación, centrándose en dar respuesta rápida a los cambios que surjan durante todo el ciclo de vida del proyecto.

2.1.2. Principios Ágiles

Estos son los 12 principios que contiene el manifiesto, extraídos directamente de su versión traducida publicada en `http://agilemanifesto.org/`:

– Nuestra mayor prioridad es satisfacer al cliente mediante la entrega temprana y continua de software con valor.

– Aceptamos que los requisitos cambien, incluso en etapas tardías del desarrollo. Los procesos ágiles aprovechan el cambio para proporcionar ventaja competitiva al cliente.

– Entregamos software funcional frecuentemente, entre dos semanas y dos meses, con preferencia al período de tiempo más corto posible.

– Los responsables de negocio y los desarrolladores trabajamos juntos de forma cotidiana durante todo el proyecto.

– Los proyectos se desarrollan en torno a individuos motivados. Hay que darles el entorno y el apoyo que necesitan, y confiarles la ejecución del trabajo.

– El método más eficiente y efectivo de comunicar información al equipo de desarrollo y entre sus miembros es la conversación cara a cara.

– El software funcionando es la medida principal de progreso.

– Los procesos ágiles promueven el desarrollo sostenible. Los promotores, desarrolladores y usuarios debemos ser capaces de mantener un ritmo constante de forma indefinida.

- La atención continua a la excelencia técnica y al buen diseño mejoran la agilidad.
- La simplicidad, o el arte de maximizar la cantidad de trabajo no realizado, es esencial.
- Las mejores arquitecturas, requisitos y diseños emergen de equipos auto-organizados.
- A intervalos regulares el equipo reflexiona sobre cómo ser más efectivo para a continuación ajustar y perfeccionar su comportamiento en consecuencia.

2.2. Aplicación al presente proyecto

Ya que gran parte de las necesidades de un sistema de este tipo eran desconocidas de antemano, un desarrollo ágil de pequeños prototipos funcionales, a medida que se profundizaba en la materia, se postulaba como la mejor metodología de desarrollo, posibilitando la mejora e incremento continuado de los todos los componentes implicados.

En el desarrollo del motor gráfico presentado, se han aplicado muchas de las ideas recogidas en el *Agile Manifesto*. En nuestro caso, se han realizado iteraciones con una duración aproximada de un mes, tras las cuales se mantuvieron reuniones donde fijar nuevos objetivos y funcionalidades a implementar. En estas reuniones se expuso a los directores, de forma presencial, todo el trabajo realizado durante cada iteración y las conclusiones alcanzadas con las mismas, posibles mejoras y cambios surgidos también del propio proceso de aprendizaje; también se mantuvo el contacto vía otros medios de comunicación como e-mail.

Debido al evidente componente investigador que tiene como objetivo este proyecto, sumado al desarrollo de las bases de un *game engine* propio desde cero y su aplicación en la visualización de voxels, la planificación realizada de las tareas y sus tiempos no pudo estimarse con precisión al comienzo del trabajo. Igualmente, se proporciona un diagrama de *Gantt* en el que se pueden observar las diferentes tareas acometidas y sus temporalidades a lo largo de las diferentes iteraciones. En el capítulo de Conclusiones (Capítulo 8) se

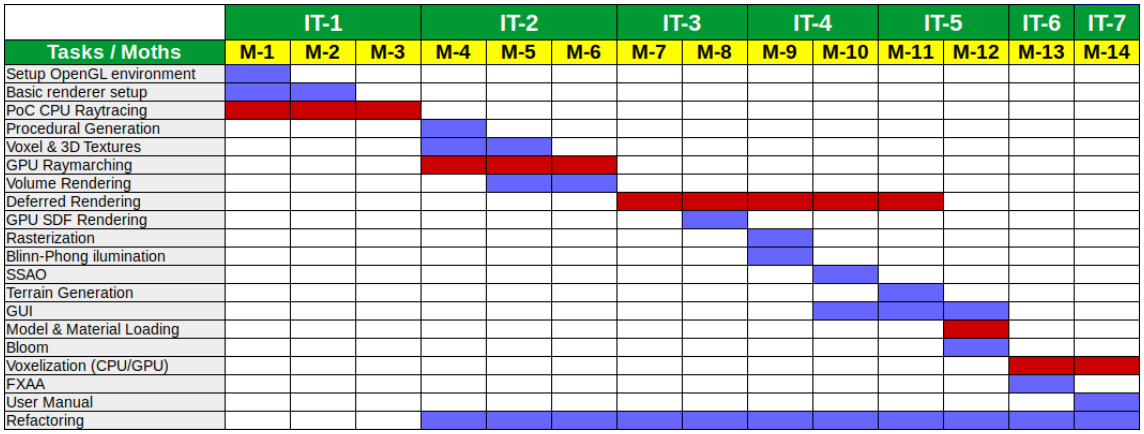


Figura 2.1: Diagrama de Gantt del proyecto

incluye una narración de las distintas iteraciones que dieron lugar al producto finalmente desarrollado.

2.3. Herramientas de apoyo

Para el control de las diferentes revisiones del software, producidas a lo largo de todas las iteraciones, se ha utilizado el sistema de control de versiones **Git**, un software libre capaz de rastrear de forma eficiente todos los cambios realizados sobre los ficheros a través del tiempo.

Se han mantenido dos copias del código fuente del proyecto en la "nube": un repositorio principal en la plataforma *GitHub*, y un *mirror* secundario en la plataforma *BitBucket* en el que se han volcado las diferentes *demos* para revisión por parte de la dirección del proyecto.

En la Figura 2.2 puede observarse un gráfico que representa el número de contribuciones a lo largo de los meses de desarrollo. Como puede apreciarse, la cantidad de *commits* es mucho más elevada en la dos primeras iteraciones, esto es debido a que el desarrollo de gran parte de las bases del motor se realizó en el primer período de dos meses, entre principios de febrero hasta finales de marzo.

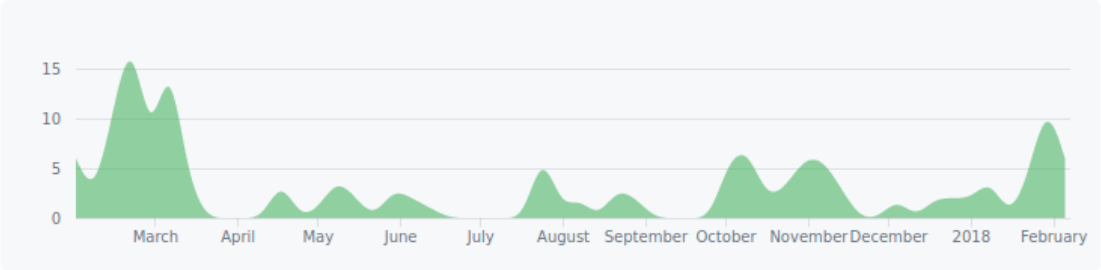


Figura 2.2: Contribuciones (*commits*) realizadas sobre el código a lo largo del desarrollo (extraídas del repositorio en GitHub)

Posteriormente se distinguen otras tres iteraciones, correspondientes a los meses de abril, mayo y junio. En ellas se añadieron las principales funcionalidades del sistema. Durante los meses de julio, agosto o septiembre se incorporaron múltiples nuevas funcionalidades y se comenzó el desarrollo de gran parte de la documentación.

Los meses de octubre, noviembre y diciembre fueron dedicados a añadir múltiples efectos gráficos que potenciasen la calidad visual, se añadieron los mecanismos de voxelizado, se amplió la documentación disponible y se desarrolló la interfaz de desarrollo, permitiendo un completo control y reconfiguración del *pipeline*. Finalmente, entrado el año 2018, se completó y corrigió esta memoria, realizando cambios surgidos a raiz de una profunda revisión de todo el trabajo hasta la fecha, de cara a su presentación.

Para generar la documentación técnica del código fuente se ha empleado la herramienta *Doxygen*. *Doxygen* es el estándar de *facto* para la generación de documentación de código C++ a través de anotaciones y comentarios en los ficheros fuente, siendo capaz de generar tanto un manual en línea (HTML) como un documento de referencia en LaTeX. Sirve como una potente herramienta capaz de extraer la estructura completa del código (clases, atributos, métodos, enumeraciones...), permitiendo visualizar tanto definición como implementación y relaciones de dependencia entre las entidades que forman el programa.

Capítulo 3

FUNDAMENTOS DE COMPUTACIÓN GRÁFICA

En el curso de este capítulo se presentan brevemente algunos de los conceptos básicos relativos a la computación gráfica tridimensional. Tras una síntesis de diferentes técnicas para la generación de imágenes digitales, se introducen algunos de los fundamentos técnicos subyacentes.

3.1. Introducción

Un gráfico es una imagen o representación visual de un objeto, de la misma forma, el término *computer graphics* (Gráficos por computador) hace referencia a la capacidad de los ordenadores para mostrar gráficos e imágenes en pantalla. El término fue acuñado en 1960 por los investigadores Verne Hudson y William Fetter de la empresa Boeing y actualmente es una disciplina amplia y variada, presente en multitud de campos.

La generación de gráficos tridimensionales, en todas sus facetas, parte de un concepto denominado *rendering*. El **rendering** es el proceso algorítmico que sintetiza las imágenes de forma computacional y requiere, inicialmente, de una gestión de objetos o modelos

que se pretenden visualizar. Habitualmente estos modelos se representan como conjuntos de vértices con coordenadas y otras propiedades asociadas, aunque también es posible encontrar para diferentes tipos de *rendering*, modelos representados mediante funciones matemáticas, densidades o distancias, entre otros.

En la segunda parte del proceso, es necesario someter la escena a una secuencia de cálculos que constituyen el *pipeline* de renderizado, principalmente: proyección, iluminación y sombreado.

3.2. Rendering Pipeline

El *rendering pipeline* es la "cadena de montaje" de la imagen final; por un extremo recibe las definiciones de los objetos (vértices y propiedades) que conforman la escena y devuelve, a su salida, la imagen resultante de observar esta escena a través de una cámara, definida en su mismo espacio.

Los ajustes de este proceso, afinados por los programadores y artistas en las diferentes etapas, junto con las características propias del *pipeline* utilizado, conforman el resultado estético de la imagen final sintetizada. Es por ello que se suele clasificar el *rendering* en dos estilos principales:

- **No Fotorrealista:** No busca imitar la iluminación del mundo real, por lo que el estilo entre unos y otros *renders* puede diferir enormemente en función de la estética que se desea obtener; por ejemplo, el *toon shading* hace referencia al *rendering* que pretende parecer un dibujo animado, pero también encontramos estilos que aparentan ser pinturas o ilustraciones, por ejemplo.
- **Fotorrealista:** Su objetivo es lograr una iluminación y aspecto final que aproximen la realidad. Es bastante habitual el uso de modelos de iluminación geométricos, con "trucos" que permiten ofrecerle a nuestros ojos una sensación de realismo a través de aproximaciones, pero sin tratar de modelar de forma rigurosa el comportamiento de la luz en su interacción con la materia.

Sin embargo, también se incluyen en esta categoría los **PBR** (*Physically Based Rendering*), modelos que simulan numéricamente una aproximación rigurosa de la iluminación a través de los principios de la física, por ejemplo la conservación de energía o el uso de BRDFs (funciones de distribución de reflectancia).

En la Figura 3.1 podemos observar un ejemplo de dos estilos de *shading* aplicados en la misma escena, uno fotorrealista en la imagen de la izquierda y otro que no lo es en la de la derecha.

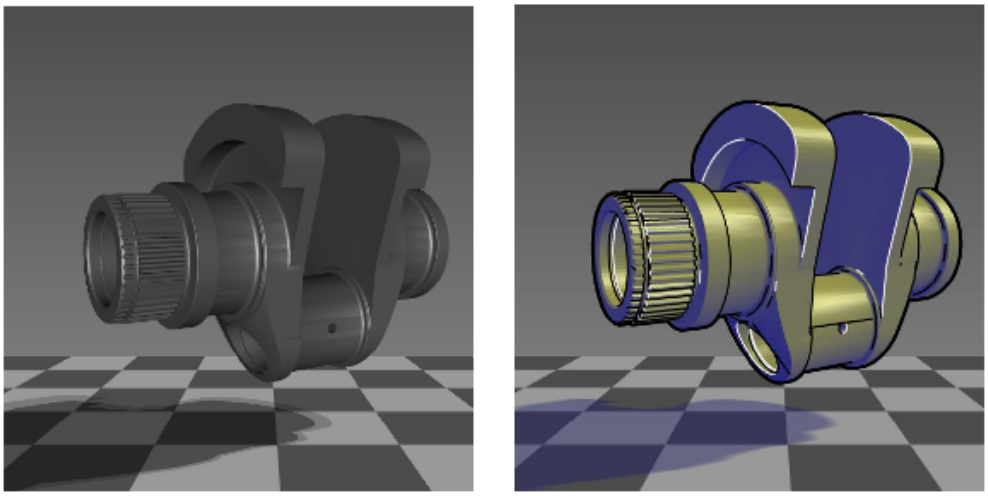


Figura 3.1: Ejemplos de shading

Las diferentes aproximaciones del proceso de *rendering* repercuten directamente sobre los tiempos de generación de las imágenes, es por tanto necesario clasificar también el proceso según el objetivo que persiga en términos temporales:

- **Offline rendering:** Lo encontramos, habitualmente, asociado al *rendering* fotorrealista, en situaciones en las que el proceso involucra una alta cantidad de geometría, iluminación físicamente realista o un post-procesado pesado y en resumen, en casos en que los cálculos para definir la imagen final son costosos.

  El proceso en estos casos se puede extender, para una sola imagen, durante horas, incluso en *rendering farms* (Granjas o clusters de renderizado).

  Este es el tipo de *rendering* principalmente utilizado en la industria cinematográfica

o en visualización arquitectónica, por poner un par de ejemplos.

- **Real-time rendering:** También denominado *Online rendering* o *Interactive Rendering*, persigue que la síntesis de las imágenes sea tan rápida como para permitir la interacción en tiempo real con el usuario: mínimo de entre 20 y 30 FPS (fotogramas por segundo). De esta forma se dota al sistema gráfico de una respuesta inmediata ante la ejecución de acciones, por ejemplo el movimiento de la cámara, generando así una sensación de continuidad y permitiendo la interactividad entre los usuarios y el propio entorno virtual.

  Para conseguir unos tiempos lo suficientemente pequeños, este tipo de *rendering* sacrifica la calidad visual en pro de la velocidad. En torno a esta idea se han desarrollado métodos y técnicas muy variadas que permitan aproximar el resultado de un buen *rendering* a través de cálculos más ligeros, trucos ingeniosos o la preparación previa de los datos, de forma que también contengan parte de la información necesaria pre-computada *offline*.

  Esta clase de renderizado es el que podemos encontrar en los videojuegos, programas de modelado y animación tridimensional, o programas CAD (*Computer-Aided Design*); requieren de una interactividad en tiempo real con el usuario, que navega, modela, escala, rota y transforma los objetos del espacio virtual.

En este trabajo estudiaremos algunas de las técnicas implementadas para la obtención de un *rendering pipeline* interactivo cuyo objetivo principal es la gestión de contenido formado por voxels, pero que también permita la inclusión de elementos intrínsecamente poligonales.

3.3. Técnicas de Rendering

Actualmente existen múltiples técnicas de *rendering*; éstas aunque variadas, se enfocan a los dos principales objetivos del proceso: determinar la visibilidad de los objetos de la escena y determinar el color definitivo de los píxeles de la imagen final.

Siguiendo este criterio, podemos clasificar las técnicas de renderizado en las siguientes familias, aunque otras clasificaciones son posibles:

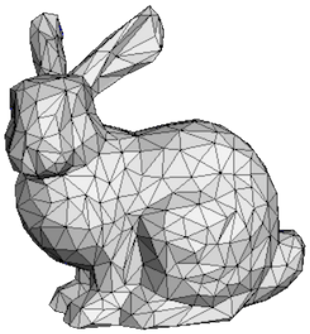


Figura 3.2: Modelo representado por una malla poligonal

- **Rasterization**: En esta familia se incluyen los sistemas de *rendering* basados en la proyección geométrica. Los vértices de los polígonos son proyectados geométricamente sobre un plano bidimensional que se sitúa en la propia escena tridimensional. Es la familia más extendida en cuanto a renderizado en tiempo real, las tarjetas gráficas (GPUs) están principalmente diseñadas y enfocadas para acelerar este proceso de una forma extremadamente eficiente.

- **Ray Casting**: Las técnicas de *ray casting* se basan en el cálculo de intersecciones de forma analítica. La idea principal es emitir (*casting*) rayos de luz que proceden de la cámara y pasan a través de cada pixel de la imagen final, adentrándose así en la escena (imitando el funcionamiento de una *pinhole camera* o cámara estenopeica).

  Mediante el cálculo de las intersecciones de estos rayos (líneas) con los objetos de la escena (triángulos, habitualmente), se determina el color del pixel correspondiente a cada rayo.

- **Ray Tracing**: Su principal diferencia con respecto a las técnicas de *ray casting* es que los rayos provenientes de la cámara (*primary rays*) no sólo determinan el objeto con el que se intersecan, sino que la propia intersección genera rayos secundarios (*secondary rays*) que imitan el flujo natural de la luz, rebotando entre los objetos de la escena un número determinado de veces.

Sus resultados pueden llegar a ser extremadamente fieles a la realidad pero su naturaleza analítica lo convierte en un proceso pesado, por ello ha sido principalmente utilizado para *offline rendering*. En la actualidad, sin embargo, es posible conseguir ratios de imagen lo suficientemente rápidos para *online rendering*, ya sea con software de cluster o acelerado a través de GPUs [11, 27].

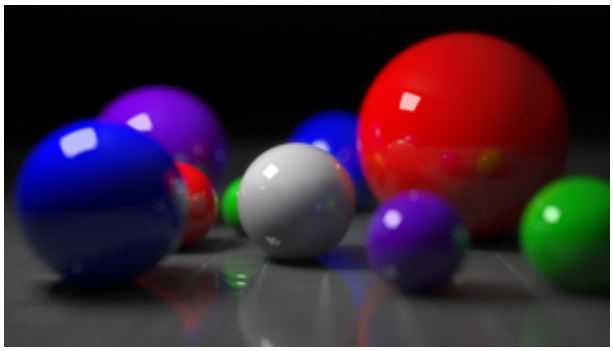


Figura 3.3: Renderizado realizado mediante *ray tracing*

La Figura 3.3 presenta un ejemplo de *render* realizado con *ray tracing*, la esfera es una figura primitiva extremadamente sencilla de dibujar con procesos de *casting* mientras que en las técnicas poligonales requiere de una gran cantidad de vértices para obtener un aspecto lo suficientemente redondeado.

- **Ray Marching**: Podría considerarse un subgrupo dentro de las técnicas de *ray casting*, en concreto, en las técnicas de *ray marching* no se computan intersecciones per se, sino que se toman muestras a lo largo de la trayectoria de los rayos emitidos por la cámara. Estas muestras pueden ser funciones que determinan distancias o una composición de múltiples funciones, cuyo resultado permite conocer qué distancia puede avanzar (*marching*) el rayo sin impactar o cruzar ningún objeto de la escena. Estas técnicas facilitan la incorporación de potentes efectos visuales, que resultan un verdadero reto en los métodos basados en rasterización, como oclusión ambiental, iluminación global o soft shadows, por poner algunos ejemplos. En *ray marching* el avance del rayo es como máximo la distancia más corta a todos los objetos de la escena. La Figura 3.4 ilustra este proceso: como podemos observar, se realizan incrementos más cortos sobre el rayo generado desde la cámara cuanto

más cerca nos encontramos de los objetos que forman la escena.

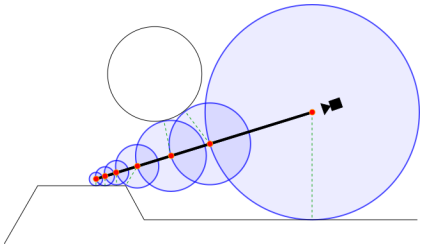


Figura 3.4: Representación del proceso de *ray marching*

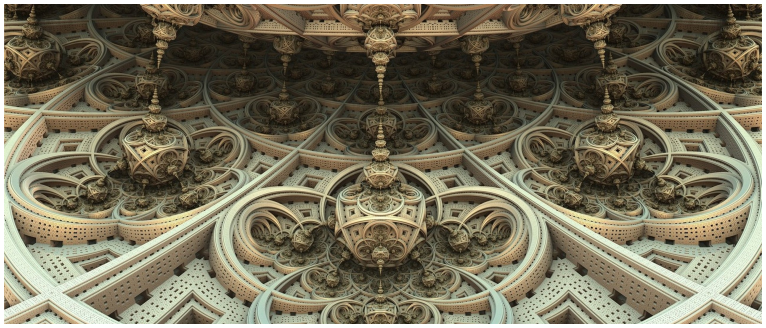


Figura 3.5: Ejemplo de *render* de una composición compleja de funciones fractales

A lo largo de las diferentes iteraciones del software presentado en este Trabajo Fin de Grado se han implementado varias de estas técnicas, enfocándolas a la representación visual de datos en forma de voxels y buscando un equilibrio que permitiese una representación en tiempo real.

3.4. Espacio Euclídeo y transformaciones

La geometría euclídea se define como el estudio de los **espacios euclídeos**, tipo de espacio en el que se cumplen los axiomas de Euclides. El término Euclídeo se utiliza para distinguir estos espacios de los "espacios curvos" de las geometrías denominadas no euclidianas.

Por lo general, los sistemas gráficos tridimensionales trabajan en un espacio euclídeo de tres dimensiones cartesianas, en el cual, mediante algunos conceptos simples de álge-

bra lineal, podemos representar todo tipo de transformaciones espaciales de puntos a puntos y vectores a vectores, en función de matrices cuadradas de dimensión 4x4 tal que:

$$p' = \mathbf{T}(p) \quad v' = \mathbf{T}(v)$$

Algunas de las transformaciones más comunes son:

- **Translación:** Permite trasladar puntos en el espacio tridimensional una cantidad determinada para cada eje.
- **Rotación:** Permite rotar tanto puntos como vectores un ángulo determinado, permite también especificar el eje de rotación: arbitrario o alguno de los ejes.
- **Escala:** Permite el escalado tanto de puntos como de vectores en un factor determinado para cada eje.

Dado que la propiedad asociativa de la multiplicación de matrices hace posible la composición de transformaciones, resulta extremadamente cómodo agrupar los parámetros de posición, rotación y escala de los objetos de la escena en forma de una única matriz por objeto. Esta matriz generalmente es denominada la matriz "modelo". Por ejemplo, una simple translación genérica se puede representar como una matriz 4x4 de la siguiente forma:

$$\mathbf{T}(\Delta x, \Delta y, \Delta z) = \begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

O, por ejemplo, un escalado:

$$\mathbf{S}(\delta x, \delta y, \delta z) = \begin{bmatrix} \delta x & 0 & 0 & 0 \\ 0 & \delta y & 0 & 0 \\ 0 & 0 & \delta z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Podemos combinar la translación y el escalado en una única matriz de transformación, tal

que:

$$\begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} \delta x & 0 & 0 & 0 \\ 0 & \delta y & 0 & 0 \\ 0 & 0 & \delta z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \delta x & 0 & 0 & \Delta x \\ 0 & \delta y & 0 & \Delta y \\ 0 & 0 & \delta z & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Y aplicarla sobre un punto o un vector:

$$\begin{bmatrix} \delta x & 0 & 0 & \Delta x \\ 0 & \delta y & 0 & \Delta y \\ 0 & 0 & \delta z & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \delta x * x + \Delta x \\ \delta y * y + \Delta y \\ \delta z * z + \Delta z \\ 1 \end{bmatrix}$$

También existen otras dos transformaciones de especial interés que son un pilar fundamental de los sistemas de proyección: serán utilizadas para la conversión entre sistemas de coordenadas, lo que se detallará en la sección siguiente.

- **Vista:** También denominada, en muchas ocasiones, como transformación *Look At* ("mirar a") define la posición y dirección de la cámara en el espacio de la escena.
- **Proyección:** Permite proyectar puntos de un espacio tridimensional a un espacio bidimensional.

A lo largo de la próxima sección se detallará la intrínseca relación de estas transformaciones con la simulación de un sistema de cámara y la forma en la que son aplicadas en las diferentes técnicas de *rendering*.

3.5. Sistema de coordenadas

Los sistemas gráficos actuales esperan que todos los vértices finalmente visibles de la imagen sean aquellos cuyas coordenadas se encuentren dentro de un rango denominado NDC (*Normalized Device Coordinates*), por ejemplo, la API de OpenGL espera que las coordenadas (*x, y, z*) se encuentren dentro del rango [-1.0, 1.0].

Sin embargo, tanto la definición de la escena como la de los propios modelos individualmente pueden trabajan en otro u otros sistemas de coordenadas. Es por ello necesario hacer uso de las transformaciones mencionadas en la sección anterior para realizar conversiones entre estos espacios.

Podemos distinguir 5 espacios de coordenadas involucrados en el proceso de proyección:

- **Local Space:** El "espacio local" es el sistema de coordenadas en el que se encuentran inicialmente los vértices que definen un modelo tridimensional. La definición de los componentes que forman el modelo se especifica alrededor de un origen de coordenadas propio.

- **World Space:** Este es el "espacio mundo" donde se representan los componentes en coordenadas globales, es decir, un mismo origen para todos los modelos de la escena.

  Las coordenadas de los vértices de un objeto se transforman de *local space* a *world space* a través del producto de las mismas con la matriz "modelo", la transformación coloca el objeto en el mundo.

- **View Space:** En el "espacio vista" (también conocido como *eye space* o *camera space*), el sistema de coordenadas es relativo a la posición de la cámara, el espacio que la cámara puede ver desde su posición, orientación y dirección.

  Las coordenadas en *world space* se convierten a *view space* a través del producto con la matriz "vista", aquella que contiene los parámetros posicionales de la cámara.

- **Clip Space:** Este espacio determina qué vértices se encuentran dentro de la imagen y cuales no. Todas las coordenadas en el espacio anterior (*view space*) se normalizan, de esta forma se obtienen las coordenadas en el anteriormente mencionado rango NDC, todos aquellos vértices con algún componente fuera del rango, son descartados.

  Para realizar esta conversión se hace uso de la matriz "proyección", aplicada sobre

las coordenadas en *view space*.

- **Screen Space:** El "espacio de pantalla" es el espacio bidimensional de la imagen final en píxeles, también denominados *texels* en términos de texturas.

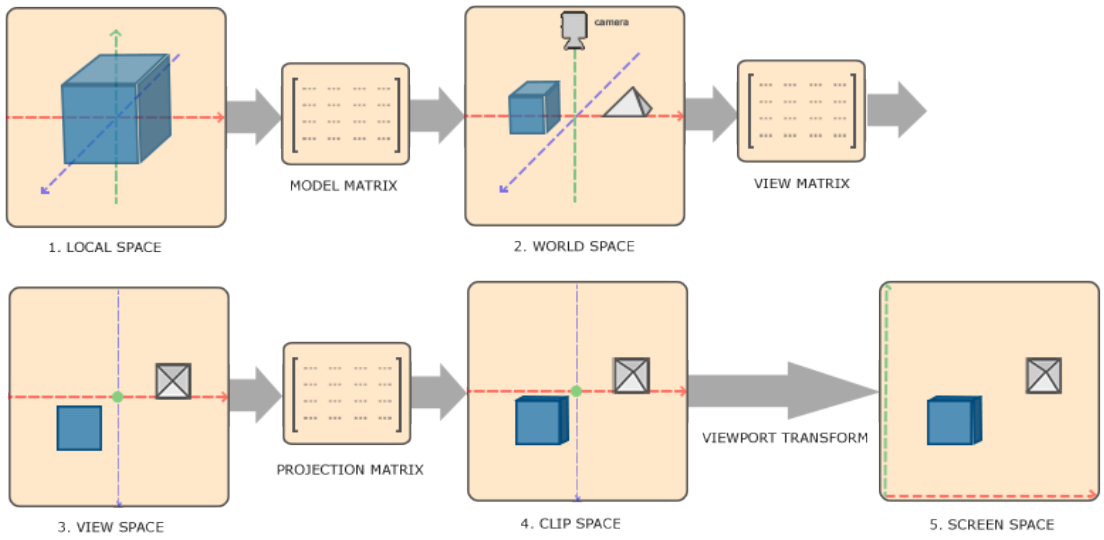


Figura 3.6: Secuencia de transformaciones entre sistemas de coordenadas espaciales

En resumen (ver Figura 3.6), el proceso de proyección geométrica consiste en aplicar sobre cada vértice de la escena, el producto de la matriz compuesta MVP (modelo $\times$ vista $\times$ proyección) de forma que se reconstruya la geometría de la escena desde el punto de vista de la cámara, según sus ajustes.

En los procesos basados en la emisión (*casting*) de rayos no siempre es necesario hacer uso de todas estas transformaciones, debido a la inherente naturaleza analítica de los mismos. Por su parte, las APIs gráficas actuales, como OpenGL, están diseñadas para trabajar en base a la proyección geométrica acelerada en la tarjeta gráfica, por lo que en muchas ocasiones los sistemas de *casting* pueden llegar a ser mucho más rápidos en procesadores convencionales: acelerados con el uso de instrucciones vectoriales y aprovechando la libertad de acceso y tamaño que suele proporciona la memoria principal del computador.

3.6. El modelo de cámara

La cámara representa el punto de vista del observador en la escena tridimensional y, al igual que cualquier otro objeto dentro del mundo modelado, posee una posición y orientación que, junto con una serie de parámetros específicos que podemos equiparar a los ajustes de óptica de una cámara en el mundo real, determinan el resultado final del proceso de proyección.

De la definición de un modelo de cámara serán extraídos los componentes de las matrices "vista" y "proyección" (la matriz "modelo" es específica de cada objeto). Además, en los sistemas de *casting*, la cámara determina el origen y dirección de los rayos emitidos.

3.6.1. Parámetros de la cámara

La definición de un modelo de cámara comienza determinando la situación de la misma con respecto a la escena. Para ello, es bastante común utilizar los siguientes cuatro parámetros, ilustrados en la Figura 3.7.

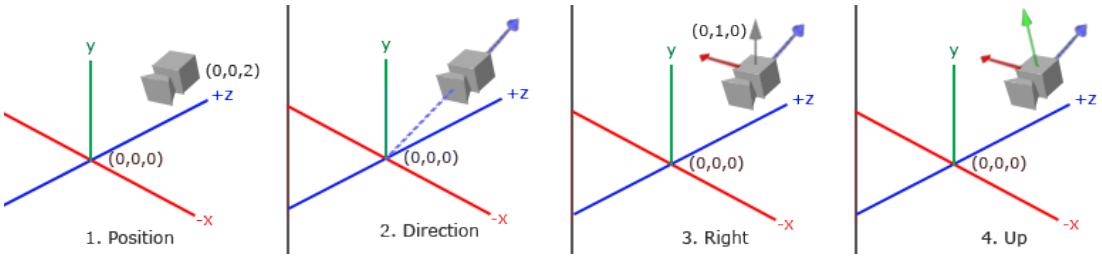


Figura 3.7: Componentes vectoriales de la cámara

- **Posición:** Representa las coordenadas en *world space*, es decir, la posición de la cámara con respecto del resto de objetos del minimundo.
- **Vector dirección:** La dirección de la cámara se representa como un vector que indica el sentido en el que la cámara está mirando, su frente.
- **Vector derecho:** Vector que indica en qué dirección está el lado derecho de la cámara.

– **Vector arriba:** Vector que indica en qué dirección está la parte superior de la cámara, con respecto al eje vertical del mundo.

A través de los parámetros anteriores podremos conocer la localización, orientación y dirección hacia la que mira la cámara, pero es necesario también especificar su profundidad y ángulo de visión, componentes que afectarán de forma notable en la proyección de la imagen final y que determinan la situación de los planos de *clipping* (aquellos planos que definen el límite entre lo que está dentro y fuera del campo de visión):

– **Near:** Distancia que determina la profundidad (*depth*) mínima del campo de visión, la geometría (o parte de ella) situada demasiado cerca de la cámara, será descartada.

– **Far:** Distancia que determina la profundidad (*depth*) máxima que alcanza la cámara, cualquier geometría (o parte de ella) situada a una profundidad mayor será descartada.

– **Field of View:** El *field of view* o *FOV* establece el ángulo de visión de la cámara, puede especificarse en dos componentes separadas: vertical y horizontal, aunque por lo general sólo se utiliza el componente horizontal junto con el *aspect ratio* (ratio de aspecto, cociente entre el ancho y alto de la imagen final).

3.6.2. Funcionamiento interno de la cámara

Teniendo en consideración todo el conjunto de parámetros descritos en el subapartado anterior, podemos definir el campo de visión de la cámara (*view frustum*) como la región de espacio en escena modelada que potencialmente podría aparecer en pantalla.

En la Figura 3.8 podemos observar una ilustración de dos tipos de proyección de la escena contenida en el *view frustum*. La proyección en perspectiva fuga las líneas al punto de la cámara, lo que provoca que los objetos más alejados parezcan más pequeños (*escorzo*), lo que resulta en una imagen más natural. Sin embargo, en la proyección ortográfica, las líneas paralelas nunca llegan a juntarse, de esta forma las dimensiones proyectadas de los objetos mantienen su tamaño y escala, lo cual es muy útil en ingeniería o arquitectura,

pues permite distinguir mucho mejor las proporciones reales.

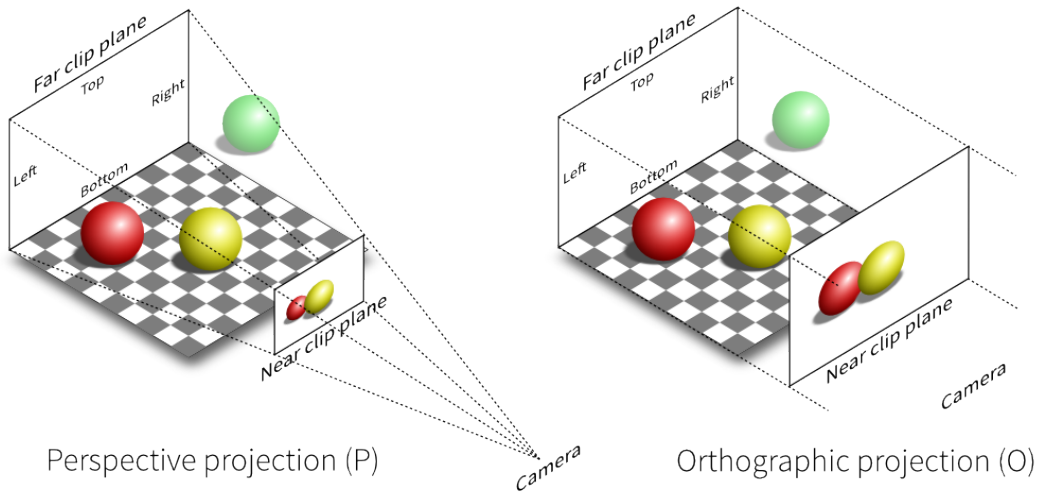


Figura 3.8: Representación de las proyecciones de cámara perspectiva (izquierda) y ortográfica (derecha)

Internamente es necesario actualizar los componentes vectoriales de la cámara con cada movimiento de la misma (posición u orientación) para poder extraer de ellos las dos matrices de interés: "vista" y "proyección".

A través de los componentes de la cámara podemos construir una matriz "vista" que definirá un espacio de coordenadas basado en su orientación (componentes dirección, derecho y arriba). A esta matriz también será necesario añadir una translación, que se corresponde a la posición (componente posición) que ocupa la cámara en *world space*, es decir, respecto al resto de objetos de la escena.

$$\textbf{LookAt} = \begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Donde R, U y D son los vectores derecho (right), arriba (up) y dirección (direction). P es la posición de la cámara en *world space*, nótese que la posición se invierte, esto es debido a que el efecto de la translación en la matriz *LookAt* es mover todo el contenido de la escena en función del lugar que ocupe la cámara. Realizar este desplazamiento en la

dirección contraría a aquella en la que nos movemos generará el deseado efecto de avance sobre la escena.

La matriz "proyección" que permite proyectar los vértices de *view space* al espacio de pantalla (*screen space*) se construye a partir de los parámetros de cámara relativos al campo de visión. La proyección de perspectiva se define de la siguiente manera, aunque otras expresiones son posibles:

$$\begin{bmatrix} \frac{1}{aspect * tan(\frac{fov}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{tan(\frac{fov}{2})} & 0 & 0 \\ 0 & 0 & -\frac{far + near}{far - near} & -\frac{2 * far * near}{far - near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Donde *fov* es el ángulo de visión vertical, *aspect* el ratio de aspecto del plano sobre el que se realiza la proyección y *near* y *far* representan la profundidad mínima y máxima del *view frustum* (piramidal, ver Figura 3.8).

La matriz de proyección ortográfica, por otra parte, se puede definir a través de los planos que forman su *view frustum* (cúbico, ver Figura 3.8). Dados los planos r, l, t, b, f y n correspondientes respectivamente a los planos: derecho (*right*), izquierdo (*left*), arriba (*top*), abajo (*bottom*), cercano (*near*) y lejano (*far*), podemos expresar la proyección ortográfica de la siguiente manera:

$$\begin{bmatrix} \frac{2}{r - l} & 0 & 0 & -\frac{r + l}{r - l} \\ 0 & \frac{2}{t - b} & 0 & -\frac{t + b}{t - b} \\ 0 & 0 & \frac{-2}{f - n} & -\frac{f + n}{f - n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Cabe destacar, por otro lado, que en el *renderizado* a través de métodos de *casting* la proyección geométrica no es estrictamente necesaria (en realidad, ya está implícita en el proceso de lanzamiento de los rayos). En estos métodos se calcula un rayo emitido (al menos) para cada pixel de la imagen final, empleando habitualmente la posición de la cámara como origen de todos los rayos, siendo la dirección de cada uno, por separado, el

vector director resultante de unir la posición de la cámara y la posición del pixel en *world space*. Podemos observar una ilustración de este concepto en la Figura 3.9.

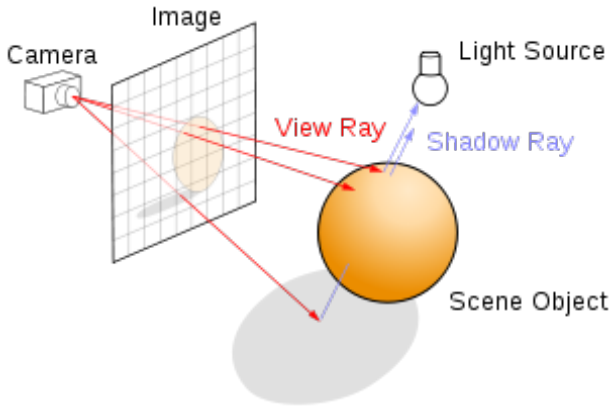


Figura 3.9: Ilustración de la emisión de rayos primarios y secundarios en técnicas de *casting*

Capítulo 4

FUNDAMENTOS TECNOLÓGICOS

En el presente capítulo se recogen de forma somera los principales fundamentos tecnológicos que forman parte de este trabajo. Así, el capítulo comienza con una breve introducción a la API gráfica multiplataforma OpenGL, describiendo brevemente su funcionamiento. A continuación se expone un pequeño resumen acerca de los sistemas gráficos volumétricos, en el que se detalla el uso y estado del arte en la visualización de nubes de puntos o voxels.

Finalmente, se estudian los requisitos de los sistemas gráficos interactivos, es decir, aquellos que producen imágenes digitales en tiempo real y permiten la interacción del usuario con el entorno gráfico. En esta parte se detallan algunos de los componentes esenciales de estos sistemas, haciendo especial énfasis en aquellos métodos y técnicas aplicadas en los motores de videojuego.

4.1. OpenGL

OpenGL (Open Graphics Library) es una especificación estándar que define una API multiplataforma que posibilita la creación de aplicaciones gráficas, facilitando la explotación del hardware gráfico, sobre el que crea una capa de abstracción.

La API de OpenGL es, por tanto, una interfaz estándar que facilita el desarrollo de aplicaciones gráficas, ocultando la complejidad de las diferentes interfaces de las GPUs (Graphic Processing Units), cuya implementación concreta dependerá enteramente de los fabricantes.

El estándar asegura que todas las implementaciones soporten el total de las funcionalidades *core* que expone la API. Así, si fuese necesario, los fabricantes deben implementar vía software aquellas capacidades no soportadas por su propio hardware.

4.1.1. OpenGL pipeline

El funcionamiento de OpenGL se basa en el uso de procedimientos a bajo nivel que afectan al funcionamiento del denominado *pipeline* gráfico o también llamado "máquina de estados de OpenGL".

El *pipeline* es la secuencia de pasos o etapas que permite obtener imágenes bidimensionales a partir de las definiciones de primitivas tales como puntos, líneas, polígonos, etc. situados en un espacio tridimensional, aplicando los fundamentos matemáticos vistos en el capítulo anterior. Al ser estas etapas independientes, la GPU puede ejecutar de forma concurrente cada una de ellas sobre distintos datos, alimentando la entrada de cada etapa con la salida de la anterior.

En las primeras versiones, el diseño de la máquina de estados de OpenGL era demasiado rígido, las etapas ejecutaban funciones pre-fijadas y no permitía la modificación de su funcionamiento, careciendo los desarrolladores de libertad a la hora de configurar la forma en la que los cálculos del *pipeline* eran realizados. Sin embargo, a lo largo de los años, esta rigidez ha ido disminuyendo y, a partir de la versión 2.0, se introduce el concepto de "etapa programable", que es la base de la programación OpenGL moderna, que se afianza a partir de la reescritura de gran parte de la API con la versión 3.0.

Las etapas programables son etapas del *pipeline* en las que el programador puede inyectar código que será ejecutado en la vasta cantidad de unidades de procesamiento que

posee la GPU, de forma paralela. Estos programas son denominados *shaders* y utilizan un lenguaje de programación propio, llamado GLSL (OpenGL Shading Language), basado en el lenguaje ANSI C y extendido para soportar operaciones sobre matrices y vectores, operaciones típicas en gráficos tridimensionales.

Actualmente, OpenGL soporta tanto su modo de funcionamiento retrocompatible (denominado *Compatibility profile*) y el modo de funcionamiento moderno, denominado *Core profile*.

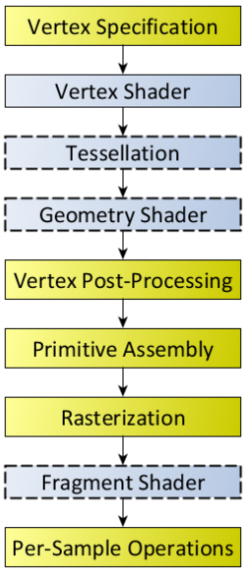


Figura 4.1: Etapas del *pipeline* moderno de OpenGL (en azul, fases programables; la línea discontinua indica que la fase es opcional)

En la Figura 4.1 podemos observar una representación de las diferentes etapas del pipeline y del flujo de datos principal entre ellas. En cada etapa se realiza una tarea concreta sobre los datos de entrada, generando una salida que será la entrada de la siguiente entrada:

1. **Vertex Specification:** En esta fase, la máquina de estados de OpenGL recibe y prepara las especificaciones de los vértices que conforman la escena.

2. **Vertex Processing:** En esta etapa, el flujo (*stream*) de vértices de la fase anterior es procesado por el *vertex shader* (primer shader programable del *pipeline*). El uso

más común del *vertex shader* es el cálculo de la proyección de las coordenadas a través de la matriz MVP (*model-view-projection*), como se explicó en el capítulo de fundamentos gráficos. Además del *vertex shader*, en esta etapa existen dos shaders adicionales, incluidos en las últimas versiones de OpenGL y que aprovechan hardware existente en las GPUs modernas: el *Tesselation Shader* y el *Geometry Shader*. Estas etapas permiten la emisión de nuevos vértices no especificados al inicio del *pipeline*, y su uso es opcional.

   - El *Tesselation Shader* posibilita la subdivisión hardware dinámica de las primitivas formadas por el flujo de vértices.

   - El *Geometry Shader* permite emitir nuevas primitivas formadas por nuevos vértices no existentes en el *stream* inicial del *pipeline*.

3. **Vertex Post-Processing:** La salida de la fase anterior, ya en espacio de coordenadas normalizado (de acuerdo con lo explicado en el capítulo anterior), es procesada aquí en varias formas. Así, por ejemplo, en esta etapa se realiza el *clipping*, que permite descartar vértices cuya proyección no coincide dentro de la imagen final.

4. **Primitive Assembly:** Fase de ensamblado de las primitivas y preparación para el rasterizado.

5. **Rasterization:** En esta fase, el proceso de rasterizado se encarga de convertir la descripción vectorial (vértices y primitivas) a un conjunto de píxeles bidimensionales o fragmentos.

6. **Fragment Processing:** Esta fase programable (y opcional) permite el procesado de los fragmentos generados por la rasterización. El uso más común del *fragment shader* es el cálculo de la iluminación de la escena, utilizando información obtenida en etapas anteriores.

7. **Per-Sample Operations:** Finalmente, la última fase del *pipeline* recoge los fragmentos de la fase anterior, realiza una serie de operaciones fijas sobre ellos y los escribe en el *buffer* de salida, como píxeles finales de la imagen, que podrá ya ser

visualizada.

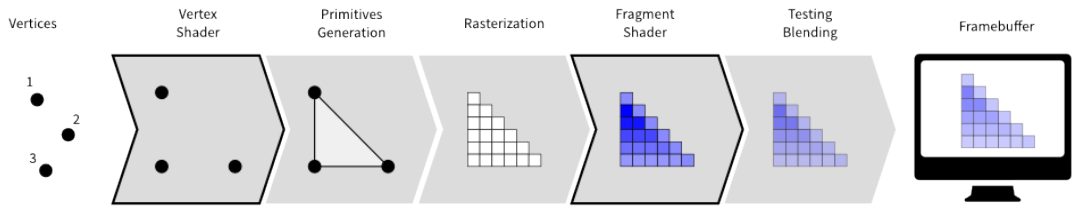


Figura 4.2: Representación gráfica del *pipeline* de OpenGL (las etapas destacadas en negrita son las etapas programables)

4.1.2. Otros usos del pipeline

Aunque el principal uso del *pipeline* de OpenGL está orientado a trabajar con descripciones geométricas (vértices, ver Figura 4.2), otros usos que aprovechen la potencia computacional de la GPU son posibles.

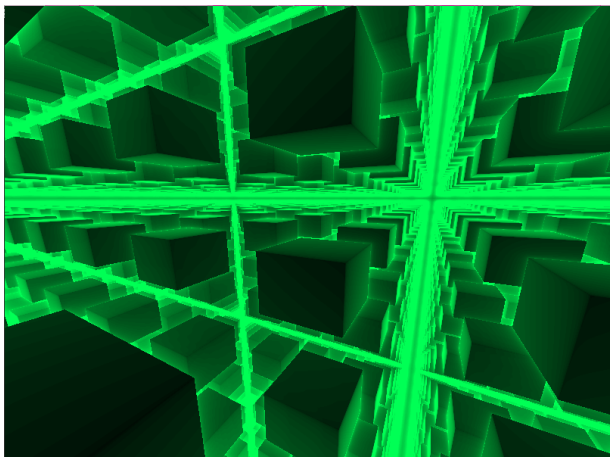


Figura 4.3: *Ray marching* en *fragment shader*, representación del coste por rayo (a mayor claridad, mayor coste en pasos)

Las capacidades que proporcionan los lenguajes de *shading* en las versiones más recientes del estándar, junto con una mayor flexibilización y programabilidad del las propias etapas del *pipeline*, ha posibilitado la utilización de la GPU como recurso computacional de uso general (GPGPU).

El aprovechamiento de la arquitectura intrínsecamente paralela de la GPU puede servir no sólo para la generación directa de imágenes (por ejemplo mediante técnicas de *casting*, Figura 4.3), sino también para realizar todo tipo de simulaciones y cálculos genéricos sobre conjuntos de datos.

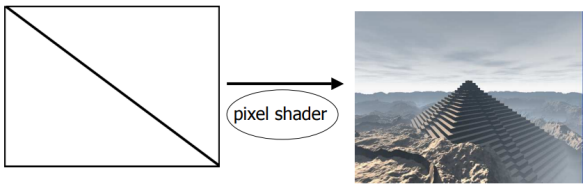


Figura 4.4: Invocación del *fragment shader* para generación procedural de terreno

Si bien existen herramientas específicas para este tipo de uso del hardware gráfico (OpenCL, por ejemplo), a través de la API de OpenGL también podemos realizar cálculo de propósito general, no directamente relacionado con el rasterizado en 2D de una escena 3D. Un ejemplo típico lo encontramos en la Figura 4.4, que muestra que podemos renderizar dos triángulos que cubran al completo el espacio de pantalla, lo que produce la invocación del *fragment shader* por cada pixel de la imagen. Esta técnica es ampliamente utilizada hoy en día, ya sea para crear todo tipo de efectos de post-procesado, como para la creación de imágenes completamente procedurales (como a menudo puede verse en el contexto de las *demoscenes* [Shadertoy.com]).

A partir de la versión 4.3 del estándar de OpenGL se introduce una nueva etapa programable en el *pipeline*, denominada *Compute Shader*, enfocada principalmente a cálculo general, por lo que la técnica anterior es incluso evitable en la actualidad.

4.2. Renderizado volumétrico

El renderizado volumétrico (*Volume Rendering*) es una disciplina que agrupa las técnicas y métodos que permiten visualizar conjuntos tridimensionales de datos o muestras. Típicamente estos conjuntos de datos son adquiridos a través de escáneres de imagen

médica, como pueden ser un TAC (Tomografía Axial Computarizada) o una RMN (Resonancia Magnética Nuclear). Normalmente las muestras son tomadas siguiendo un patrón regular, por ejemplo en base a múltiples *slices* (rebanadas) en 2D, de forma que el conjunto de todos los *slices* forma un grid o parrilla regular en 3D y puede ser tratado como un volumen de voxels. La visualización de los volúmenes descritos puede tomar varias aproximaciones distintas:

- **Basadas en *slices*:** Estas técnicas renderizan los diferentes *slices* que forman el modelo por separado, y aplican una función de transferencia de opacidad para obtener el color final de cada pixel.

  Los *slices* pueden estar alineados a los ejes de coordenadas y renderizados desde el punto de vista de la cámara o los *slices* pueden ser dinámicamente obtenidos de forma que estén orientados con el plano de visión de la cámara (el plano de proyección). Por lo general son métodos ligeros que permiten la visualización en tiempo real, en la Figura 4.5a podemos observar un ejemplo de render de este tipo.

- **Basadas en *splatting*:** Esta técnica es similar a la utilizada para visualización de nubes de puntos. Cada elemento (voxel) es dibujado como un pequeño disco o *splat*. Esta operación se realiza en orden de más lejano a más cercano a la cámara, de forma que se pueda acumular el color y transparencia del pixel final.

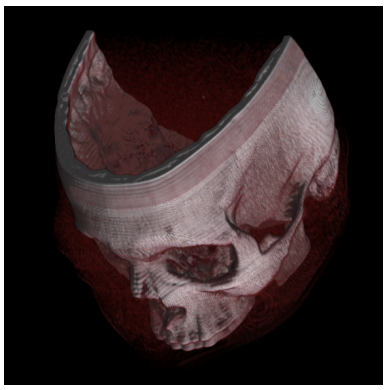
- **Basadas en *casting*:** Utilizan procesos de *ray casting* o *ray tracing* directamente sobre el grid de voxels o alguna estructura de aceleración que sirva de índice. Cada rayo emitido acumula el valor de todos los voxels que se intersecan en su trayectoria a través de la estructura, y computa el color del pixel correspondiente en función de estos valores, generando de forma efectiva un efecto de opacidad que permite distinguir el interior de los modelos (ver Figura 4.5a).

- **Basadas en *isosurfaces*:** Una *isosurface* o isosuperficie es una superficie tridimensional que representa un conjunto de puntos de valor, por ejemplo voxels. En estas técnicas se extrae una descripción de la superficie del volumen mediante un pre-procesado del mismo, por ejemplo, construyendo una malla poligonal, lo que

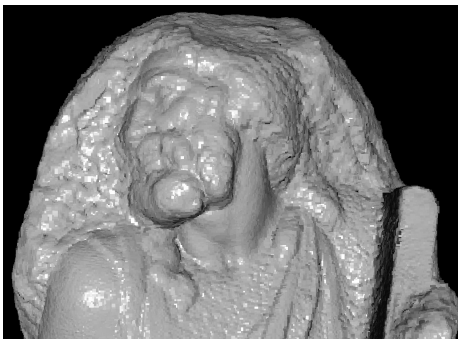
posibilita el uso del *pipeline* tradicional de rasterizado para *renderizar* la escena.

A diferencia de las técnicas anteriores, con la extracción de superficie perdemos la capacidad de representar el volumen interno del modelo a través de opacidades.

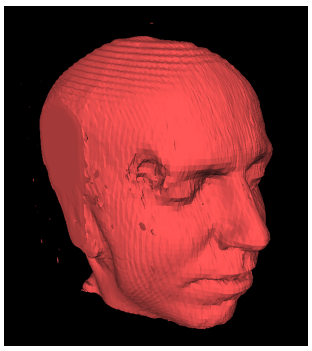
Existen múltiples algoritmos para la extracción de isosuperficies desde un conjunto de voxels, cada uno con sus ventajas, inconvenientes y variantes. Algunos ejemplos son: *Marching Cubes*, *Surface Nets* o *Dual Contourning*. En la Figura 4.5c podemos ver un ejemplo de isosuperficie extraída mediante el algoritmo *marching cubes*.



(a) *Render* volumétrico basado en *slices*



(b) *Render* basado en *splatting*



(c) Resultado del algoritmo *marching cubes* para la extracción de isosuperficie

Figura 4.5: Ejemplos de técnicas de *rendering* volumétrico

4.3. El voxel

El voxel (*Volumetric Pixel*) representa la muestra, medición o dato, contenida en una celda de un grid regular de tres dimensiones, y es por tanto, el equivalente volumétrico del pixel (ver Figura 4.6). Este sencillo concepto es utilizado en gran cantidad de disciplinas y para múltiples propósitos, principalmente en el ámbito científico. No obstante, a lo largo de los últimos años, su uso en industrias como el cine o los videojuegos lo han popularizado enormemente al rededor del ancho y largo del globo.

Un conjunto de voxels describe un volumen de datos que puede ser visualizado a través de variadas técnicas de computación gráfica. Los valores que estos voxels pueden contener dependerán exclusivamente de lo que se pretenda representar: densidad, opacidad, color, velocidad, flujo, etc. y, de forma inherente, posición tridimensional. Por norma general, la precisión o resolución de estas medidas repercute de forma negativa, generado *sets* de datos excesivamente grandes, característica que dificulta en gran medida su visualización en plataformas convencionales.

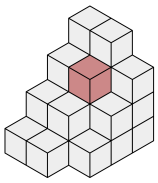


Figura 4.6: Representación de voxel (Volumetric Pixel)

Cuando se trata de visualización de voxels, el estado del arte ha derivado en dos corrientes bien diferenciadas, aunque complementarias:

- Los métodos directos utilizan procesos de *raycasting* o *ray tracing* para determinar la visibilidad de los objetos y que posibilitan, entre otros, la capacidad de representar también el interior del volumen (algo bastante útil en campos como la medicina o la ingeniería) [3, 10].

- Los métodos indirectos extraen una representación ligera y manejable de la superficie del volumen, por ejemplo mediante polígonos, que pueda ser utilizada en el

pipeline gráfico tradicional para el que las tarjetas gráficas actuales están tremendamente optimizadas [6, 7, 8].

Capítulo 5

ESTRUCTURA DEL MOTOR DE RENDERIZADO

En este capítulo se tratan las características propias de los motores de renderizado, realizando una introducción a este tipo de software, con sus usos y peculiaridades. Acto seguido, se hace una breve descripción de algunos de los conceptos y recursos requeridos por este tipo de programa, lo que sirve de base para el subsiguiente análisis de requisitos y casos de uso de nuestra aplicación. El capítulo finaliza con el diseño del sistema propuesto, *Unnamed Engine*, con el desglose de los diferentes componentes que lo conforman.

Téngase en cuenta que muchas de las figuras y diagramas a lo largo de este capítulo no contemplan la totalidad de las relaciones y dependencias entre los diferentes componentes del sistema. Para un mayor detalle se recomienda acceder a la diferente documentación técnica adjunta con este trabajo (diagrama UML completo y documentación autogenerada con Doxygen).

5.1. Introducción

Desde el punto de vista de un programador, un motor de renderizado (*rendering engine*) o motor gráfico es un *set* complejo de software cuyo principal trabajo es coordinar la ejecución y tratamiento de un serie de recursos para obtener como resultado una imagen. El motor debe decidir qué es lo que está siendo visualizado, qué debe ser dibujado y cómo debe ser dibujado, y si se trata de *online rendering* estas operaciones deberán repetirse una gran cantidad de veces por segundo, proporcionando un flujo de imágenes (fotogramas) lo suficientemente rápido para aportar sensación de continuidad a la escena, ante el movimiento de la cámara o los objetos dentro de la misma.

A nivel arquitectónico, el motor gráfico debe ocultar los detalles de las APIs gráficas (como puede ser OpenGL) sacrificando flexibilidad, pero proporcionando suficiente abstracción para el tratamiento de conceptos de un nivel más alto: cámaras, modelos, materiales o animaciones, por nombrar algunos, en contraposición con valores, vértices o triángulos a nivel bajo. Estos motores también son responsables de orquestar la composición de la imagen final, como resultado de un conjunto de operaciones e intercambio de datos entre diferentes fases, algunas de ellas ejecutadas para cada uno de los fotogramas, bajo demanda o pre-computadas.

5.2. Conceptos básicos

A continuación describiremos algunas de los principales términos relacionados con los sistemas de rendering y algunas de las entidades participantes en el proceso:

- **Cámara:** La cámara o cámaras son las entidades principales cuyo estado (posición, orientación, apertura...) determina la visibilidad de la escena representada.
- **Textura:** Las texturas pueden ser descritas como imágenes, normalmente bidimensionales, de un tamaño determinado y en las que cada pixel posee un valor concreto. Si bien uno de los principales usos de las texturas se destina al tratamiento de imáge-

nes (colores), éstas también son utilizadas para almacenar, transferir y/o procesar cualquier otro tipo de valores.

- **Malla:** Las mallas son en esencia conjuntos de geometría, normalmente triángulos, listas de vértices que contienen, a parte de su posición espacial, otras propiedades de interés.

- **Material:** Los materiales agrupan aquellas propiedades que determinan la apariencia visual de un conjunto de partes geométricas de la escena. Normalmente se asocian a las Mallas, conjunto que definiremos como Modelo.

- **Modelo:** Los modelos agrupan una o múltiples Mallas con uno o múltiples Materiales. Los modelos son normalmente cargados desde ficheros creados por modeladores o diseñadores.

- **Escena:** Una escena es un contenedor de alto nivel que contiene al completo el *set* visual de datos, normalmente representado como un grafo de Modelos y otras entidades, junto con transformaciones (la posición y orientación de los Modelos en el minimundo).

- **Viewport:** El *viewport* es un término que representa la parte de la pantalla de la aplicación en la que se está dibujando, es decir, es un área bidimensional de tamaño arbitrario.

- **Render Target:** Este término hace referencia a cuál es el objetivo (*target*) de la ejecución de operaciones de dibujado. El objetivo de las operaciones de dibujado puede ser variado: la propia pantalla, una textura, un *buffer*...

- **Draw List:** El término *draw list* hace referencia a la lista o conjunto de operaciones necesarias para dibujar la escena actual, normalmente esta lista es actualizada cada fotograma o cada pocos fotogramas con el contenido actualmente visible por la Cámara.

- **Framebuffer:** Un *framebuffer* es una porción de memoria (*buffer*) en la que se almacena un mapa de *bits* (píxeles de una imagen). En concreto, el denominado

screenbuffer es aquel en el que escribimos en última instancia cuando deseamos mostrar la imagen en pantalla.

5.3. Análisis

El diagrama de la Figura 5.1 representa, a grandes rasgos, el flujo de trabajo de *Unnamed Engine*, que puede dividirse en 5 fases principales:

Inicialización (Pipeline Setup)

La primera fase se corresponde con la secuencia de arranque del motor: creación de la ventana, inicialización de los recursos necesarios para el funcionamiento del *pipeline* (render targets, buffers...) y finalmente, configuración de las entradas y salidas en las diferentes fases de renderizado, definiendo las interconexiones entre ellas.

Administración de eventos (Event Pulling)

Inicializados los recursos principales, puede comenzar la ejecución del bucle principal, es decir, aquel que se repetirá para cada nuevo fotograma. La administración de eventos es la primera fase del bucle, en ella se actualiza el estado de los diferentes componentes del motor según los eventos que hayan sido "disparados" en el ciclo anterior (entradas de ratón/teclado, finalización de la carga asíncrona de modelos o imágenes, entre otros).

Determinación de visibilidad (Visibility Determination)

Una vez administrados los eventos y ejecutadas las correspondientes operaciones, se procede a la determinación de la visibilidad, es decir, se construye una lista de dibujo (*draw list*) con aquellos modelos que se encuentran dentro del área de visión de la cámara y por tanto necesitan ser dibujados en el fotograma actual.

Voxelizado (Voxelization)

La fase de voxelizado es la encargada de realizar un mapeo de la escena visible, construyendo una versión regularizada de la misma en forma de *grid* de voxels. El

grid construido podrá ser utilizado posteriormente para la realización de cálculos complejos de iluminación o renderizado volumétrico, entre otros [1, 2].

Renderizado (Rendering)

La fase de renderizado ejecuta, en un orden determinado, todas las operaciones de dibujado y post-procesado de la imagen determinadas en fases anteriores, generando así una imagen final que será volcada sobre la ventana del usuario. También, y de forma independiente y opcional, puede realizarse una penúltima fase justo antes de volcar la imagen a pantalla: el dibujado de la interfaz de desarrollo, por encima de la imagen final (*Gui Overdraw*), en la que se encuentran las diferentes herramientas utilizadas para reajustar las diferentes propiedades del *pipeline* programable.

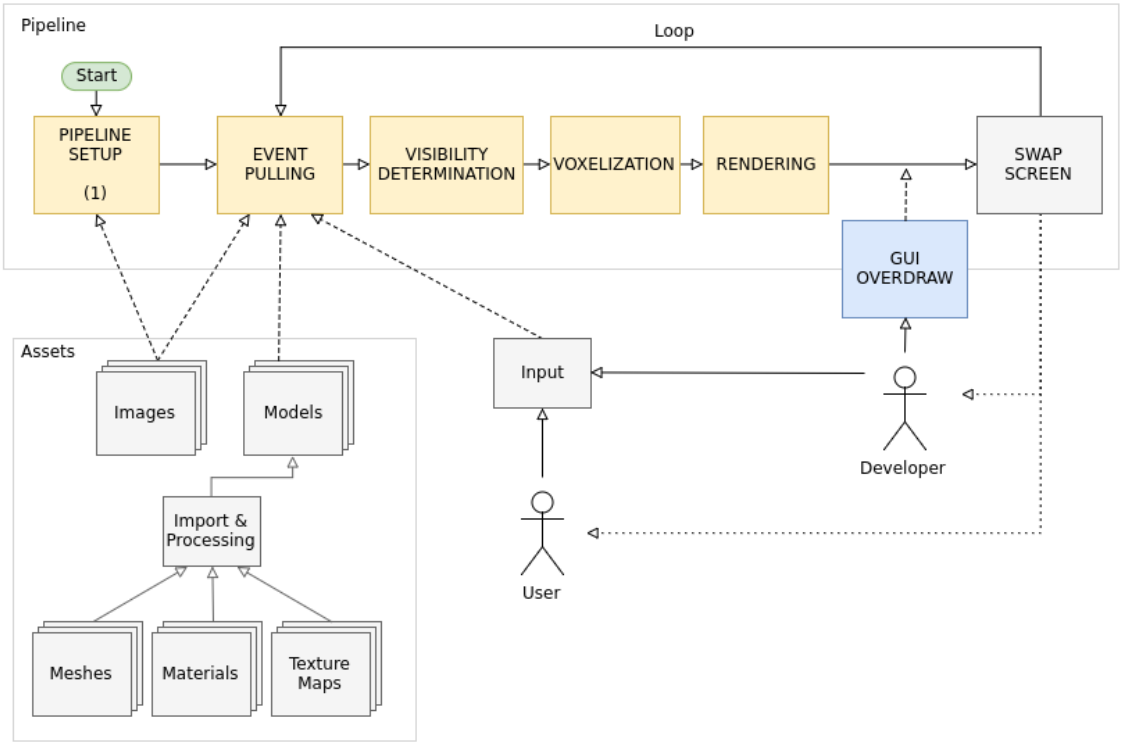


Figura 5.1: Diagrama del flujo de trabajo de Unnamed Engine

5.3.1. Casos de uso

En esta sección se presentan los principales casos de uso analizados, téngase en cuenta que, tratándose de un sub-sistema de renderizado, la gran mayoría de casos representados en la Figura 5.2 se corresponden con las operaciones disponibles para los desarrolladores, siendo las disponibles para los usuarios finales, aquellas derivadas de la aplicación del *pipeline* a un software concreto.

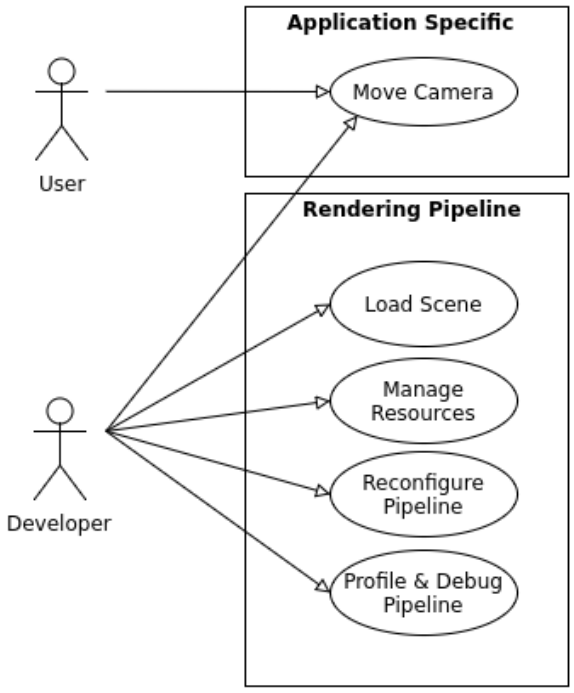


Figura 5.2: Casos de uso del software Unnamed Engine

CU01	**Mover Cámara**
Actor	Usuario/Desarrollador
Precondición	Pipeline inicializado
Postcondición	Estado de la cámara actualizado
Secuencia	1- El usuario mueve el puntero o utiliza las teclas de movimiento 2- Se actualizan los parámetros de cámara 3- Se determina la visibilidad con los nuevos parámetros cámara 4- El sistema renderiza un nuevo fotograma
Excepciones	1.a- El puntero puede estar deshabilitado por el uso de la interfaz de desarrollo, (volver a paso 1).

CU02	**Cargar Escena**
Actor	Desarrollador
Precondición	Pipeline inicializado
Postcondición	-
Secuencia	1- El desarrollador inicia la herramienta de escena 2- El usuario introduce la ruta del modelo a cargar 3- Presiona el botón LOAD 3- El sistema inicia la carga del modelo 4- El modelo aparece en pantalla
Excepciones	3.a- Si ya existía una escena cargada, el sistema liberará todos los recursos relacionados. 3.b- Si la ruta no es válida o no se encuentra el modelo, el sistema muestra un mensaje de error. Volver a paso 1.

CU03	**Gestión de Recursos**
Actor	Desarrollador
Precondición	Pipeline inicializado
Postcondición	Recurso creado/actualizado/eliminado
Secuencia	1- El desarrollador inicia la herramienta de gestión del recurso concreto 2- El desarrollador modifica los recursos 3- El sistema crea/actualiza/elimina los recursos 4- El recurso refleja los cambios en el ciclo de renderizado siguiente.
Excepciones	-

CU04	**Reconfiguración del Pipeline**
Actor	Desarrollador
Precondición	Pipeline inicializado
Postcondición	Parámetros actualizados
Secuencia	1- El desarrollador inicia la herramienta concreta 2- El desarrollador modifica los parámetros a gusto 3- El sistema actualiza los parámetros y reconfigura el *pipeline* 4- Los cambios se reflejan en el ciclo de renderizado siguiente.
Excepciones	-

CU05	**Profiling y Debugging**
Actor	Desarrollador
Precondición	Pipeline inicializado
Postcondición	-
Secuencia	1- El desarrollador inicia la herramienta concreta 2- El desarrollador reconfigura el *pipeline* haciendo uso de los casos de uso 02 a 04 3- El sistema actualiza los parámetros y reconfigura el *pipeline* 4- Los cambios se reflejan en el ciclo de renderizado siguiente y las herramientas muestran el impacto y resultado de los cambios.
Excepciones	-

5.4. Estructura y diseño

La estructura del motor se encuentra particionada en múltiples subsistemas con los que se pretende, no sólo crear una clara separación entre los diferentes tipos de funcionalidades, sino también conseguir independencia en la gestión de los diferentes recursos que participan en el proceso de renderizado.

Esta estructuración se beneficia de un rápido crecimiento horizontal, ya que permite expandir individualmente las capacidades de los diferentes recursos, con un bajo impacto en el resto del sistema. Teniendo en consideración la cantidad de subsistemas que un motor de estas características puede llegar a tener [9] (*networking*, físicas, audio, lógica, inteligencia artificial...) resulta una característica idónea para el crecimiento del proyecto en el futuro.

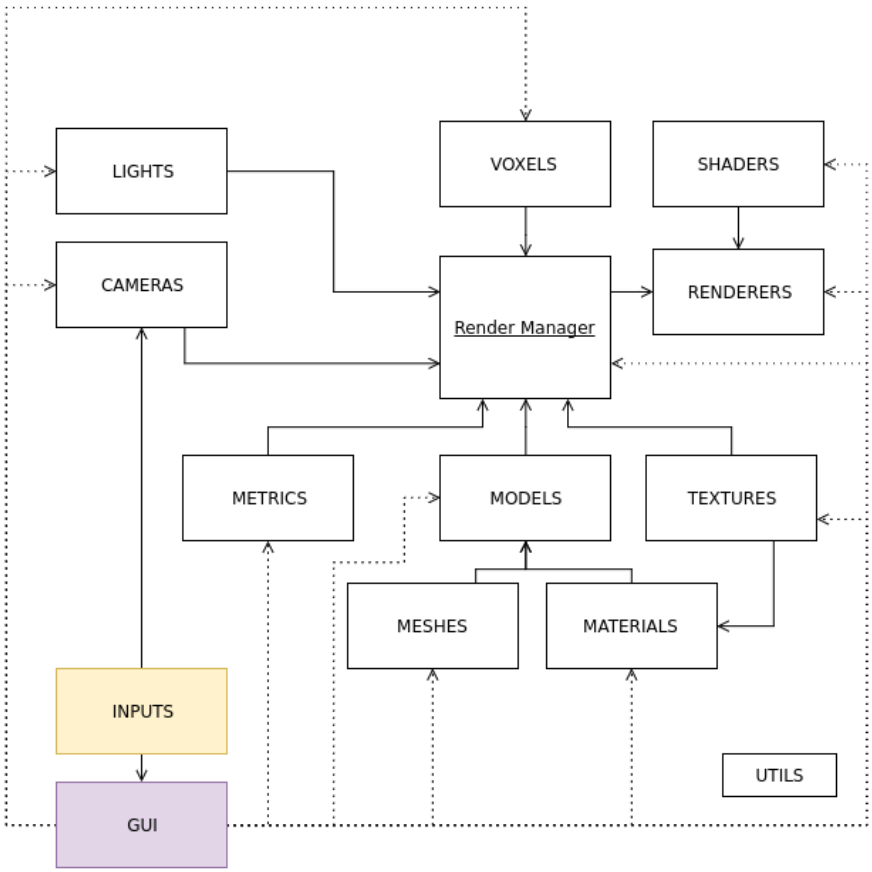


Figura 5.3: Principales subsistemas en Unnamed Engine

5.4.1. Subsistemas

En la Figura 5.3 se encuentran representados los diferentes subsistemas o módulos que se agrupan en torno al **RenderManager**. La responsabilidad principal del mismo es la orquestación de las diferentes fases del proceso de renderizado. El **RenderManager** ejecuta un variado conjunto de **Renderers**, en los que se implementan las diferentes pasadas requeridas para la composición de la imagen final (Figura 5.4).

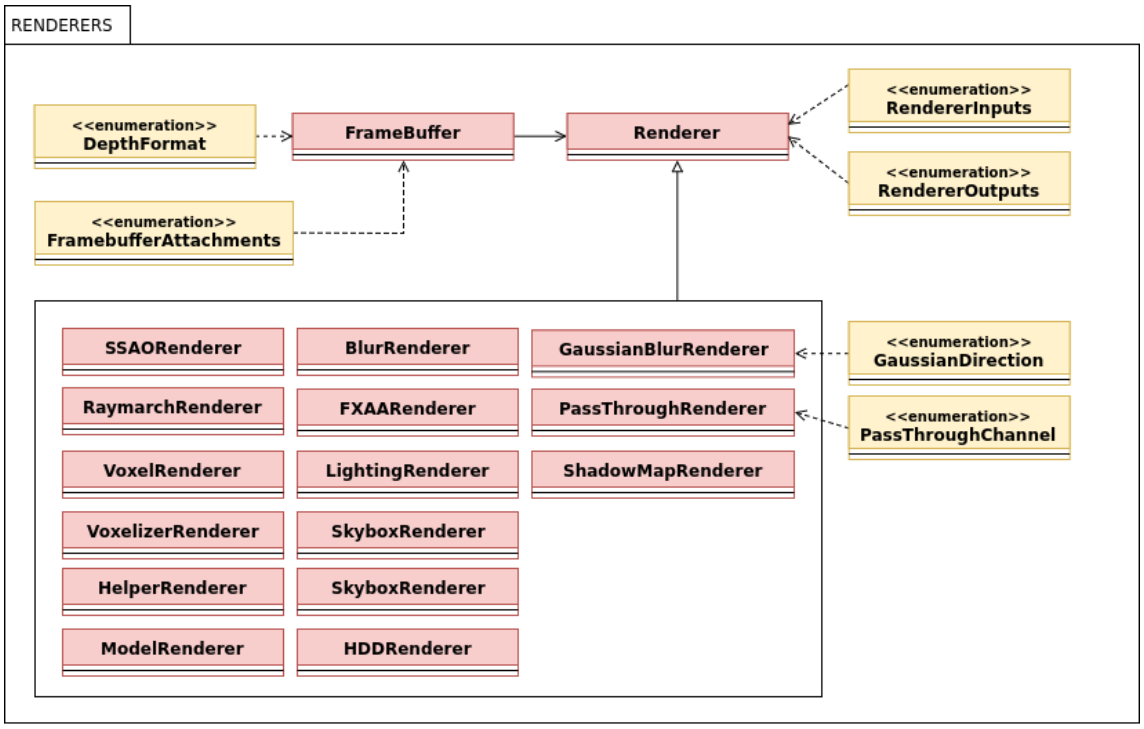


Figura 5.4: Módulo Renderers

Renderers

Los **Renderers** actualizan los parámetros de dibujado y ejecutan el correspondiente código GPU: los denominados **Shaders** (Figura 5.5). Sin embargo, para llevar a cabo la ejecución de los **Shaders** es necesario proporcionar cierto contexto de ejecución, es aquí donde entran en juego los **FrameBuffers**.

Los **FrameBuffers** sirven como objetivo de la ejecución **Shaders**. Permiten definir

dónde serán almacenados los resultados tras su cálculo en GPU. Cada **Framebuffer** posee un número determinado de enlaces (**FramebufferAttachments**) donde podrán ser asociadas texturas (**Texture2D**, Figura 5.6) alojadas en la GPU. De esta manera, los resultados escritos en las texturas pueden ser reutilizados en fases posteriores del *pipeline*. Este comportamiento será de gran utilidad, como se verá más adelante, para la orquestación de un *pipeline* de sombreado diferido (*deferred shading*), en el que se separe la fase de renderizado de la geometría de la iluminación de la escena, realizando esta última sólo sobre la porción finalmente visible por la **Camara**.

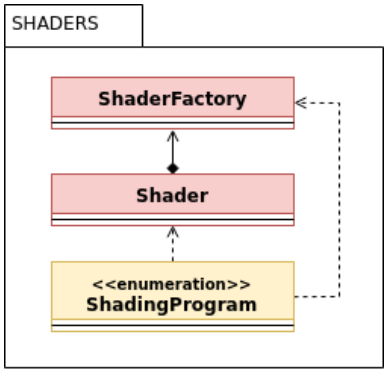


Figura 5.5: Módulo Shaders

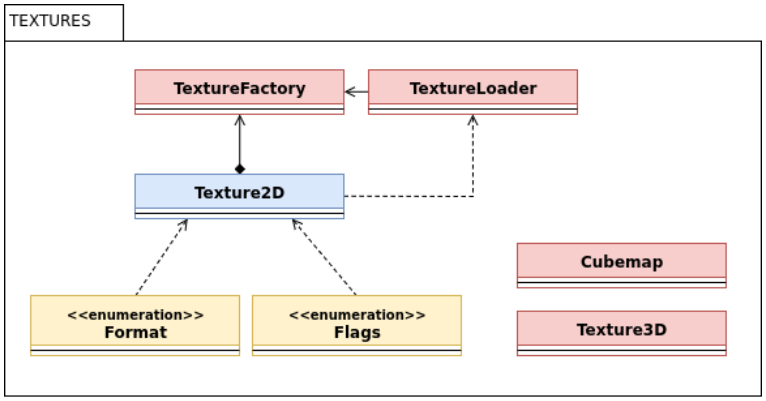


Figura 5.6: Módulo Textures

Textures

El módulo **Textures** gestiona la memoria GPU relacionada con el conjunto de imágenes que maneja el motor, sin realizar distinción entre: aquella utilizada como *render target*

(asociada en los diferentes **FrameBuffers**) y los mapas de *bits*, cargados desde fichero a través del **TextureLoader** (como aquellos utilizados en los materiales (**Materials**; Figura 5.11).

Se contemplan, sin embargo, dos tipos concretos de texturas que serán introducidos más adelante y cuya gestión no recae sobre la factoría correspondiente (**TextureFactory**), estos son: los **Cubemaps** y las texturas tridimensionales (**Texture3D**).

Cameras

En el módulo **Cameras** (Figura 5.7) se agrupan los recursos relacionados con el modelo de cámara implementado (definido en la Sección 3.6 de este documento). Adicionalmente hemos incluido también la ventana sobre la que se muestra la aplicación y la definición del **Frustum**, es decir, el área geométrica que encierra la porción de escena visible.

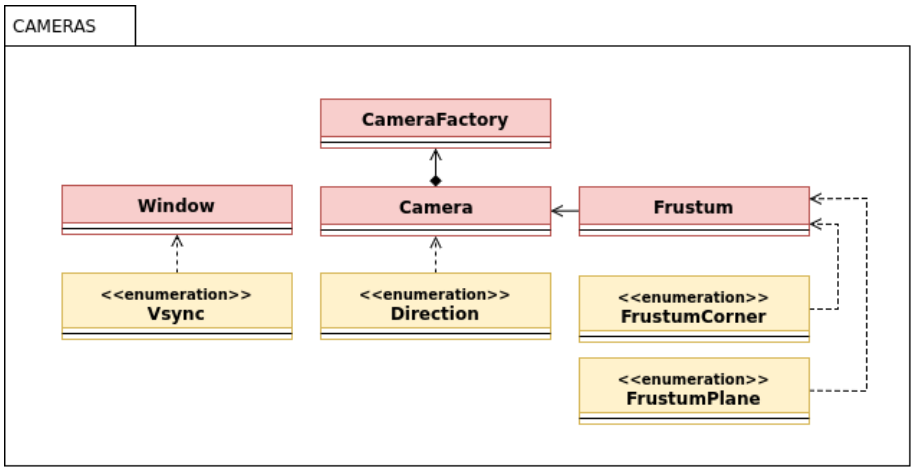


Figura 5.7: Módulo Cameras

Inputs

Como explicamos a principios de este mismo capítulo, actualmente el control del usuario se limita a modificar el estado de la **Camera** a través de la ejecución de diferentes comandos (**Command**). El módulo **Inputs** (Figura 5.8) contiene los aspectos relacionados

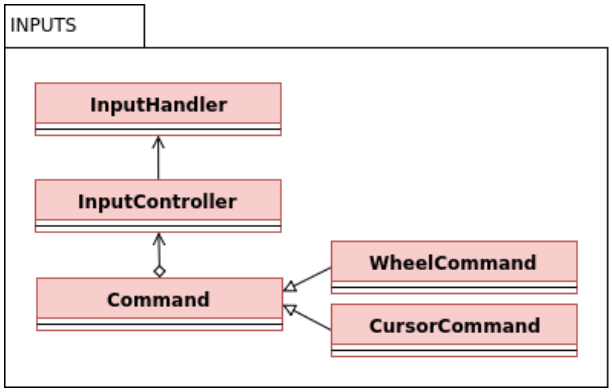


Figura 5.8: Módulo Inputs

con la gestión y ejecución de estos comandos: el **InputController** asocia las acciones del usuario, disponibles a través de los periféricos (teclado/ratón), con la ejecución de uno o varios comandos. La clase **InputHandler** mantiene el estado de las posibles entradas y según éste, determina, en cada fotograma, la ejecución de los comandos correspondientes.

De forma separada y debido a sus peculiaridades, el estado de las entradas mantenido por el **InputHandler** se deriva al módulo responsable de la interfaz de desarrollo: **GUI**, operando éste de forma aislada al sistema de comandos.

GUI

El módulo **GUI**, representado en la Figura 5.9, agrupa la jerarquía de clases que implementan las funcionalidades de la interfaz de desarrollo. Este subsistema encapsula todas las operaciones requeridas para el dibujado y manejo de la interfaz. La clase **GuiTools** agrupa las diferentes herramientas (**Tool**) de desarrollo, éstas acceden de forma independiente al resto de módulos del sistema, a través de las diferentes factorías y managers disponibles. La **Gui** de desarrollo se encuentra, por tanto, autocontenida y puede ser separada del resto del *pipeline* en las versiones de producción.

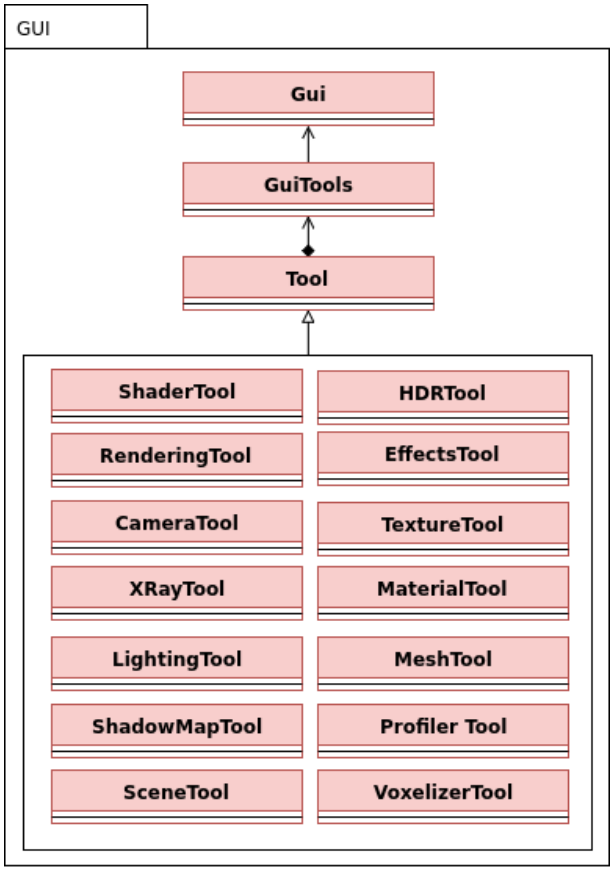


Figura 5.9: Módulo GUI

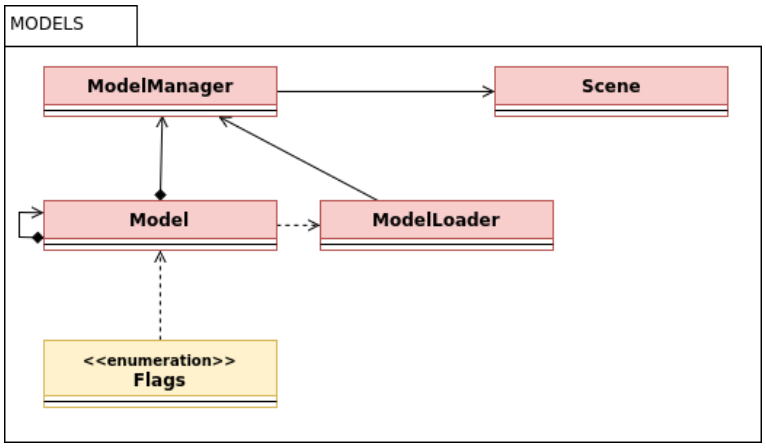


Figura 5.10: Módulo Models

Models

Cada modelo (**Model**) es representado como un árbol cuyos nodos son a su vez otros modelos. Cada nodo contiene una o múltiples mallas geométricas (**Mesh**, Figura 5.12) y

uno o múltiples materiales (**Materials**, Figura 5.11).

La clase **ModelLoader** se encarga de la lectura y carga de los modelos directamente desde fichero, tarea que realiza de forma asíncrona. En este contexto, el **ModelManager** es el último responsable de recoger los resultados y emitir las actualizaciones correspondientes en los diferentes subsistemas implicados: dar de alta las nuevas mallas y materiales cargados.

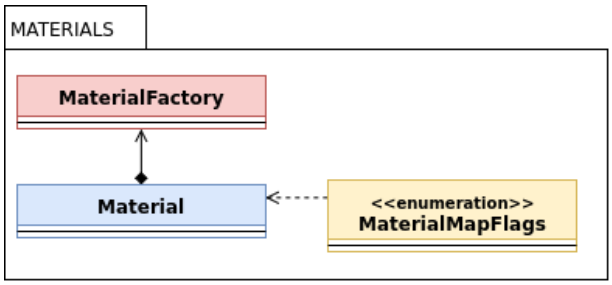


Figura 5.11: Módulo Materials

Materials

Los materiales (**Material**) son pequeñas estructuras que contienen la información necesaria para determinar el aspecto final de las mallas (**Mesh**) dibujadas en la pantalla. Estas estructuras contienen, junto con una serie de valores, múltiples mapas (**Texture2D**) a través de los cuales se pueden extraer propiedades específicas de cada polígono (triángulo), modificando el comportamiento en los algoritmos de sombreado/iluminación.

Cuando nuevos materiales son creados a través la factoría correspondiente (**MaterialFactory**), es necesario realizar también la carga en GPU de aquellas imágenes asociadas como mapas (si no habían sido cargadas anteriormente), operación que debe ser delegada al subsistema de texturas y su **TextureLoader** asíncrono. Es por ello que existe un pequeño lapso de tiempo entre la aparición de la geometría en pantalla y la correcta texturización de la misma.

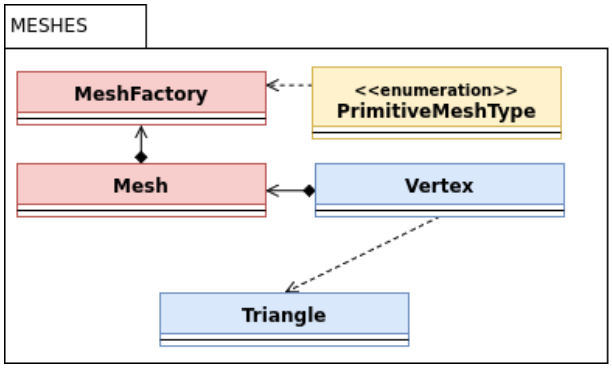


Figura 5.12: Módulo Meshes

Meshes

Las mallas (**Mesh**, Figura 5.12) son, en esencia, listas de vértices (**Vertex**) que describen la forma geométrica de los modelos **Model**. Cada vértice, a parte de sus inherentes coordenadas espaciales, contiene también información necesaria para el proceso de renderizado.

La clase **MeshFactory** se responsabiliza de la gestión de todas las mallas, realizando la carga de nuevas listas de vértices en GPU, por ejemplo resultantes del procesado de un nuevo modelo (**ModelLoader**, Figura 5.10). Adicionalmente se ha dotado a la **MeshFactory** de la capacidad de definir también mallas primitivas (cuadrado, cubo, triángulo, esfera...), que son a menudo reutilizadas por los diferentes componentes del *pipeline*.

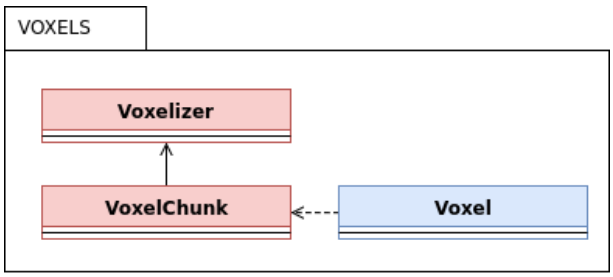


Figura 5.13: Módulo Voxels

Voxels

En la Figura 5.13 se encuentran algunas de las clases responsables de la voxelización de los modelos. Cada **VoxelChunk** almacena el resultado de la voxelización de un área cúbica de tamaño variable, pero resolución fija, entendiendo por resolución el número de **Voxel** en cada una de las dimensiones. Puede verse como un *grid* tridimensional, como el representado en la Figura 5.14.

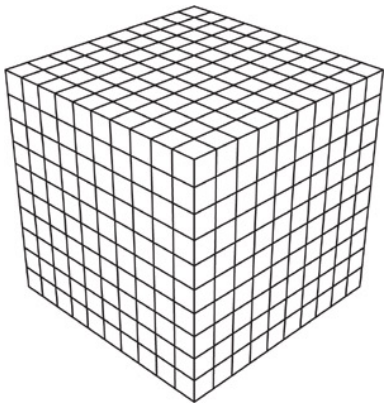


Figura 5.14: Representación de un grid tridimensional

El voxelizador (**Voxelizer**) realiza la transformación de las mallas (**Mesh**, Figura 5.12), a **Voxel**s, realizando un proceso de cálculo de intersecciones acelerado en la CPU.

Aunque funcional, la voxelización en CPU es limitada: resulta muy lenta, y no permite extraer el color de la geometría para cada **Voxel**, información únicamente accesible a través del texturizado, cuando se realiza el dibujado de la malla en GPU. Afortunadamente, el proceso anterior resulta ser extremadamente paralelizable, y teniendo en cuenta que la información de color sólo es accesible en GPU, se ha desarrollado un segundo modelo basado enteramente en **Shaders**, que aprovecha el diseño del propio *pipeline* gráfico [12, 13] y es incluso capaz de proveer de voxelización en tiempo real (en función de la resolución en **Voxel**s y el número de triángulos a voxelizar).

El voxelizador GPU no aparece reflejado en la Figura 5.13, su implementación puede encontrarse, sin embargo, como un **Renderer** más, en concreto el **VoxelizerRenderer**, (Figura 5.4) que se desarrollará con más detalle en apartados posteriores.

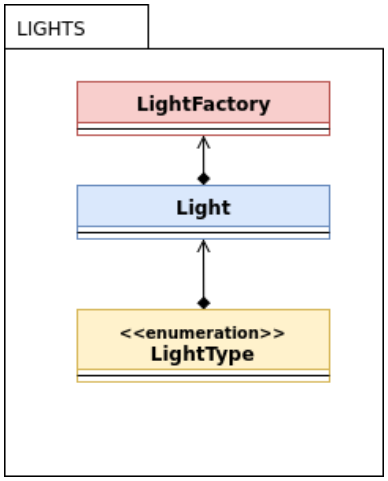


Figura 5.15: Módulo Lights

Lights

Las luces (**Light**, Figura 5.15) son pequeñas estructuras que almacenan los parámetros necesarios para simular las fuentes de iluminación en la escena. Se han implementado dos tipos de luces: luz direccional (*directional light*) y luz puntual (*point light*). La diferencia principal radica en que la luz direccional mantiene la misma dirección independientemente del punto de la escena en el que nos encontremos, es decir, simula una fuente de iluminación lo suficientemente alejada para que los rayos de luz incidan paralelamente. La luz puntual, sin embargo, incide con una dirección diferente en cada uno de los fragmentos de la escena, ya que esta dirección depende de la coordenada de origen (el punto) en el que se sitúa la fuente de iluminación.

La clase **LightFactory** gestiona el conjunto de luces que actúan en el *pipeline* de renderizado, si bien actualmente resulta una clase conveniente, se prevé su eliminación de cara a incluir el conjunto de luces como parte de la definición de la escena (**Scene**, Figura 5.10).

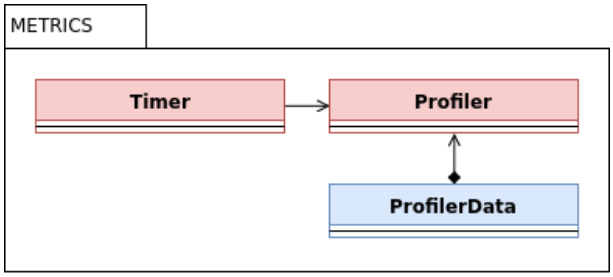


Figura 5.16: Módulo Metrics

Metrics

Las clases **Timer** y **Profiler** pertenecen al subsistema de métricas (**Metrics**, Figura 5.16).

El objeto **Timer** permite obtener datos tales como el número de fotograma actual, la duración total del último fotograma o la cantidad de fotogramas por segundo.

El objeto **Profiler** permite almacenar un número determinado de muestras etiquetadas del tipo **ProfilerData**, estas muestras son generadas cada vez que el **RenderManager** (Figura 5.4) inicia una nueva fase del proceso de renderizado y actualizadas cuando la fase finaliza. A través de estas muestras, podemos conocer y estudiar la duración individual de las diferentes fases del pipeline a lo largo de los últimos fotogramas.

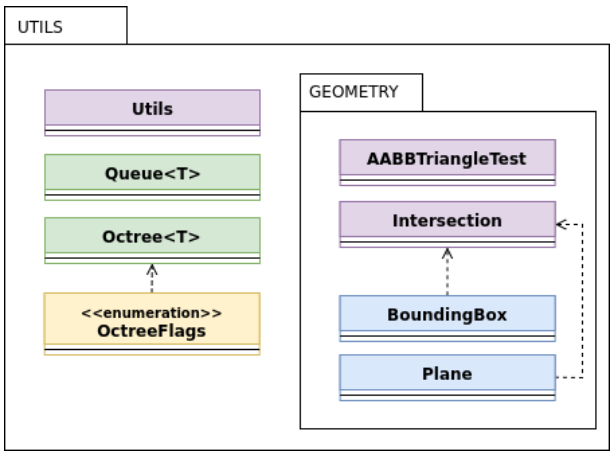


Figura 5.17: Módulo Utils

Utils

Finalmente el módulo de utilidades **Utils** aglutina funciones, estructuras y algoritmos útiles en diferentes partes del motor gráfico. Entre ellas:

- **Utils**: Contiene el conjunto de funciones de utilidad y matemáticas que no pertenecen a ningún subsistema concreto pero son utilizadas en múltiples puntos del programa.
- **Queue**<**T**>: Plantilla que implementa una cola FIFO genérica accesible de forma concurrente (*thread safe*), mayormente utilizada para el despache de tareas asíncronas como por ejemplo la carga de modelos o texturas.
- **Octree**<**T**>: Plantilla que implementa un *Octree*. El *Octree* es una estructura de aceleración en forma de árbol, en el que cada nodo tiene exactamente 8 nodos hijo. Esta estructura es utilizada como forma de particionamiento del espacio tridimensional.

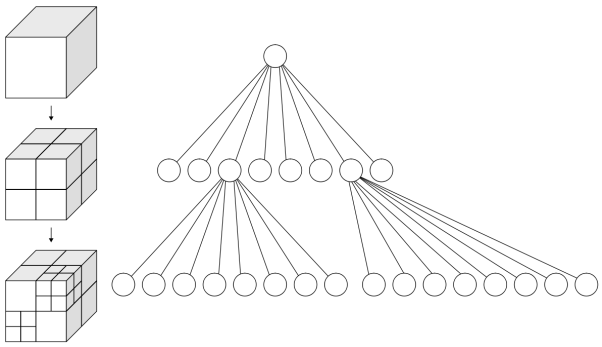


Figura 5.18: Representación de una estructura Octree

Como puede verse representado en la Figura 5.18, cada nodo es dividido a la mitad en cada uno de sus ejes, generando de esta manera 8 octantes pertenecientes al siguiente nivel del árbol. Si un nodo no contiene ningún elemento, no se subdivide ahorrando una considerable cantidad de espacio y permitiendo a los algoritmos evitar gran cantidad de cálculos, por ejemplo en procesos de *raycasting* (algo conocido como *empty space skipping*).

- **Intersection**: Agrupa varios algoritmos y test de intersección entre figuras geométricas como planos, triángulos, *bounding boxes* o el propio **Frustum** de la cámara.
- **AABBTriangleTest**: Implementación del test de intersección tridimensional entre un triángulo y una caja de tamaño arbitrario (alineada con los ejes de coordenadas) extraída de [14]. Principalmente es utilizado por el voxelizador CPU (**Voxelizer**, Figura 5.13)

Capítulo 6

RENDERING PIPELINE

Continuando el análisis de la estructura y diseño en apartados anteriores, el presente capítulo profundiza en el funcionamiento de uno de los principales subsistemas que se presenta como objeto de estudio en este documento: el subsistema de renderizado.

Comenzamos con una introducción a la técnica de *rendering* escogida para este desarrollo: el *Deferred Rendering* o *Deferred Shading*. Esta técnica, ampliamente utilizada en la actualidad, permite desacoplar en fases separadas el dibujado de la geometría respecto a la iluminación de la misma, lo que, como veremos, tiene sus ventajas e inconvenientes.

Posteriormente realizamos un desglose de las diferentes fases del *pipeline* de renderizado, que se corresponden con la gran variedad de clases **Renderer** implementadas en el motor, describiendo cómo éstas se combinan para dar como resultado el fotograma final mostrado en pantalla, tantas veces por segundo como sea posible.

6.1. Rendering Diferido

A grandes rasgos, cada vez que solicitamos el renderizado de un modelo, el deber de la GPU es iniciar la ejecución del **Shader** que se encuentre activo en ese momento. Esta ejecución determina el color de los píxeles en los que la geometría ha sido proyectada,

sin embargo también dependemos de la iluminación para determinar el color de forma definitiva.

Si incluimos la iluminación anteriormente mencionada en la ejecución del mismo **Shader** estaremos ante el *rendering* más básico, denominado *Forward Rendering* (Figura 6.1), sencillamente se provee a la gráfica de la geometría, se proyectan sus vértices transformándose en fragmentos (píxeles) para los que se calcula un color final en base a las propiedades de la geometría y las luces de la escena. Por tanto nos encontramos con una complejidad aproximada de: $O(número\_fragmentos \times número\_luces)$

El problema principal del *Forward Rendering* es que el cálculo de la iluminación se realiza para todos los fragmentos (píxeles) que ocupa cada modelo al ser proyectado, pero muchos de ellos puede que nunca lleguen a formar parte de la imagen final y acaben siendo sustituidos, por ejemplo, por los de un modelo renderizado a posteriori, situado entre la cámara y el primero de ellos. Es por tanto que, el incremento de la cantidad de luces en la escena o el incremento de la geometría renderizada degraden en exceso el rendimiento del sistema.

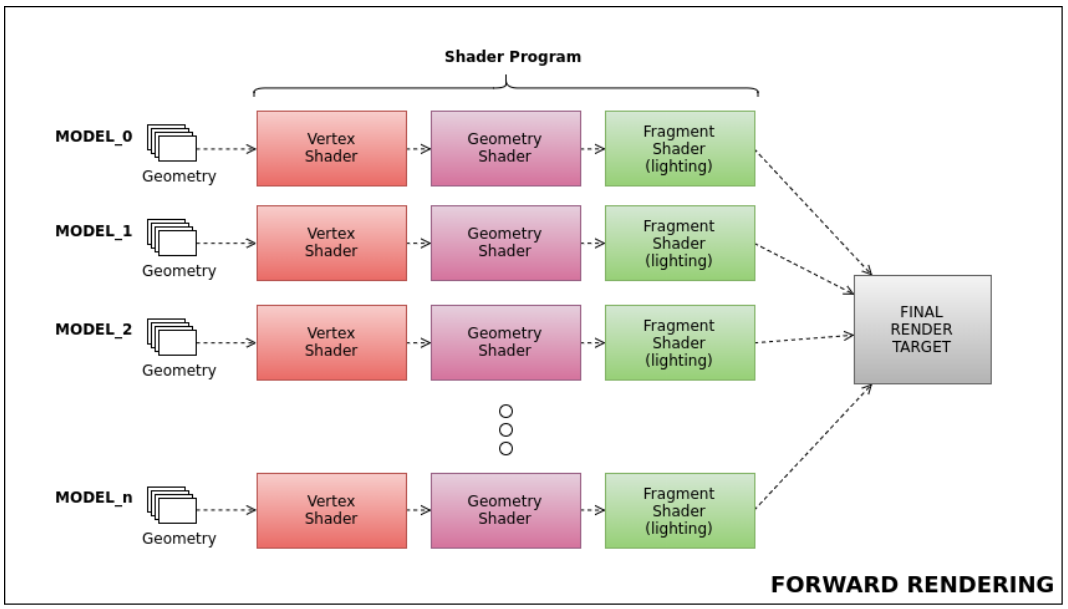


Figura 6.1: Renderizado de múltiples modelos con Forward Rendering

El *Deferred Rendering* [15, 16] consigue evitar esta situación separando el renderi-

zado de geometría del cálculo de la iluminación, de forma que podamos computar el efecto de un gran número de luces únicamente y exclusivamente sobre aquellos fragmentos finalmente visibles para la cámara (es decir, aquellos que ya sabemos que formarán parte de la imagen final). De esta manera el rendimiento de la iluminación en la escena será completamente independiente de la complejidad de la misma, pues la iluminación se realiza sobre cada pixel de la imagen final (es decir, la iluminación se realiza en espacio de pantalla o *screen space*) y por tanto su complejidad se puede representar por $O(número\_fragmentos) + O(resolución\_pantalla \times número\_luces)$

Para poder realizar la iluminación de forma diferida, es necesario contar con un conjunto de *Render Targets* intermedios (véanse como imágenes), sobre los cuales almacenaremos toda información relevante de la escena visible y necesaria para realizar una única ejecución del proceso de iluminación.

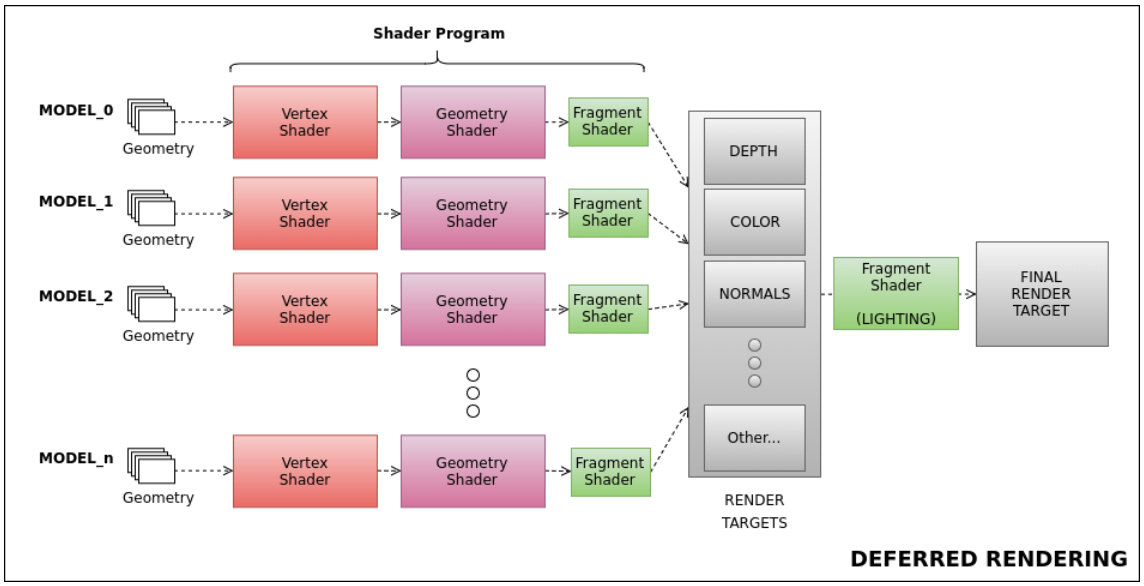


Figura 6.2: Renderizado de múltiples modelos con Deferred Rendering

En la Figura 6.2 se representa este proceso diferido. En ella se pretende reflejar, primero, la reducción del coste del *fragment shader* al eliminar todo cálculo de iluminación del mismo (que hemos dibujado con un tamaño menor) y segundo, el almacenamiento en múltiples *Render Targets* de los datos generados por el renderizado de todos los modelos y, finalmente, la única ejecución del proceso de iluminación haciendo uso de toda la

información recogida.

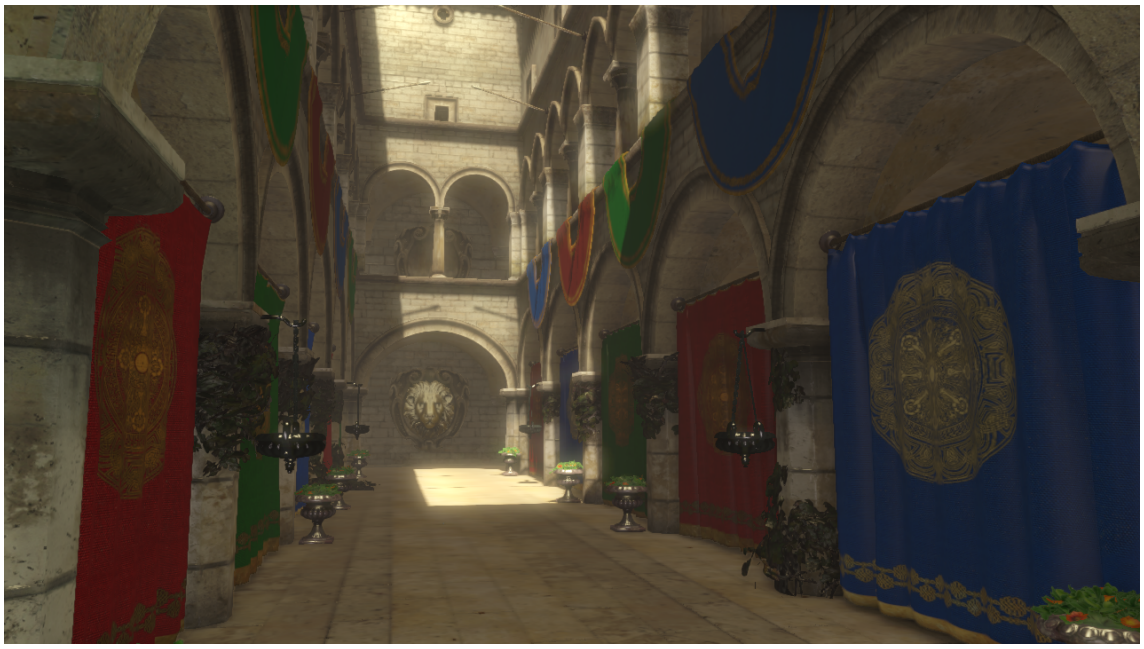


Figura 6.3: Sponza Atrium en Unnamed Engine

Sin embargo, hay que tener en cuenta que el *rendering* diferido tiene también varios inconvenientes:

- **Compatibilidad:** Este proceso requiere el uso de tarjetas gráficas medianamente modernas, los modelos de aceleradoras gráficas antiguas carecen de la capacidad de trabajar con múltiples *Render Targets* y por tanto el *Deferred Shading* es impracticable.
- **Uso de memoria:** El pipeline diferido requiere del uso de una mayor cantidad de memoria de la propia GPU, necesitamos almacenar datos intermedios, dejando así menos espacio para otros recursos también alojados en GPU como la propia geometría o texturas.
- **Ancho de banda:** A consecuencia del mayor coste en memoria, también se requerirá de un alto uso del ancho de banda disponible en la GPU al incrementarse la necesidad de transferir grandes buffers (de resultados intermedios) sobre los que realizar operaciones.

- **Transparencias:** Debido a su propia naturaleza, no resulta posible renderizar objetos transparentes. Sin embargo puede ser solucionado realizando *Forward Render* (de los objetos transparentes) sobre el resultado del *Deferred Rendering*.

6.2. Composición del Pipeline

En la Figura 6.4 se encuentran representadas las diferentes ejecuciones realizadas en GPU, necesarias para la obtención de la imagen final. Cada una de ellas se corresponden con una de las diferentes subclases **Renderer** implementadas en *Unnamed Engine* y en las que profundizaremos a lo largo de esta sección.

Cada fase del proceso ejecuta, una o varias veces, **Shader** en GPU y produce uno o varios resultados que se encuentran plasmados en la figura (Figura 6.4). Sin embargo, cabe destacar que no se corresponden uno a uno con los *Render Targets* disponibles en el sistema, ya que algunos de estos resultados intermedios pueden ser almacenados y transferidos en conjunto, como se explicará más adelante.

Los resultados generados por la GPU se escriben **buffers**, que en el contexto de los gráficos, pueden ser visualizados como imágenes bidimensionales. Estos *buffers* pueden variar de formato dependiendo el tipo de medición que almacenan, y es por ello que resulta conveniente introducir también el concepto de "canal" (*channel*).

El formato más común para estas imágenes es aquel en el que cada pixel se compone de tres canales principales denominados **R**, **G**, **B**. Las siglas provienen del inglés: *Red*, *Green*, *Blue* dado que su principal propósito es representar un color, resultado de la combinación de los tres canales. Suele añadirse un cuarto canal denominado **A** (de *Alpha*) que suele utilizarse para determinar la transparencia.

A nivel bajo, cada canal contiene un valor que está acotado por un número de *bits determinado* y, por tanto, la elección de este número determina la precisión de los valores que almacena. La combinación de un número de canales determinado junto con la cantidad de *bits* asignados para cada canal determinan el formato interno del **buffer** de datos,

que de ahora en adelante definiremos a través de la siguiente nomenclatura:

$$[R|RGB|RGBA]\ [8|16F|32F]$$

Algunos ejemplos de formatos pueden ser: R8, RGB8, RGBA8, R16F, RGB32F, donde la "F" indica que los *bits* contienen un valor en coma flotante, a excepción del formato más sencillo de 8 bit. De esta forma, es posible combinar múltiples resultados en un sólo *buffer*: por ejemplo, combinar la posición (*Position*), para la que necesitamos tres canales X, Y, Z (RGB16F) y la profundidad del fragmento (*Depth*), para la que necesitamos un único canal (R16F), utilizando el canal **A** de un buffer RGBA16F.

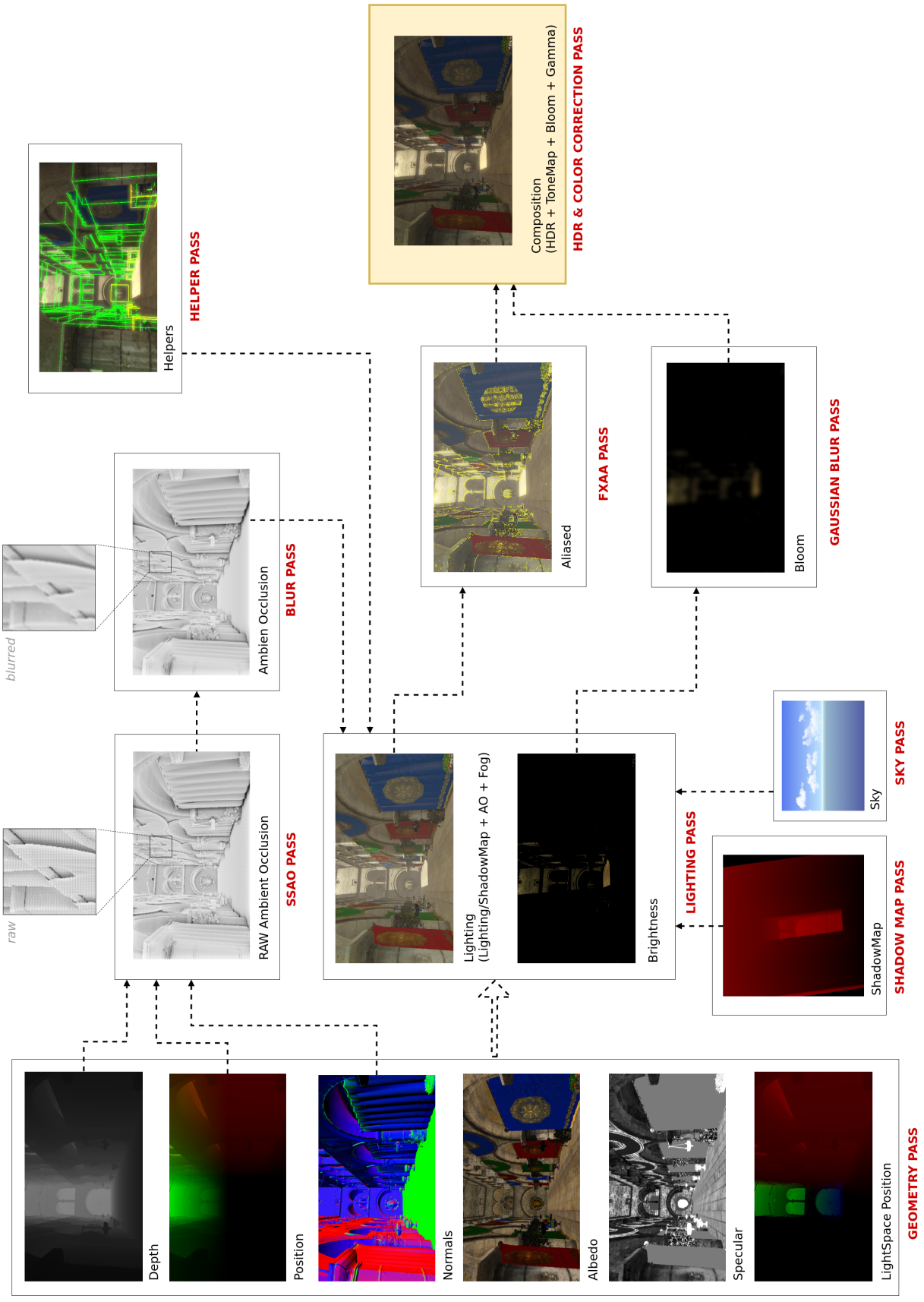


Figura 6.4: Descomposición de la imagen final en el pipeline

6.3. Geometry Pass

Denominamos "pase geométrico" (*Geometry Pass*) a la ejecución del dibujado de la geometría y sus propiedades de forma desglosada en múltiples imágenes, realizando la proyección de los polígonos y la aplicación de los materiales y texturas correspondientes. De la aplicación del *Geometry Pass* obtenemos varios buffers (imágenes) en los que se codifica, para cada pixel, parámetros necesarios para determinar el color final del mismo pixel en fases posteriores. El contenido de cada uno de estos *buffers* resultantes se detalla en los apartados siguientes.

6.3.1. Depth Buffer

El *depth buffer* (Figura 6.5), también denominado *z-buffer*, contiene la profundidad de cada fragmento (pixel) de la escena con respecto al plano de proyección de la cámara. Al realizar el renderizado de los modelos, la profundidad permite conocer la visibilidad de los fragmentos generados, determinando si cada nuevo fragmento se sitúa por detrás o por delante del generado anteriormente (para un mismo pixel) y debe o no debe ser descartado.

En Unnamed Engine, la precisión del **depth buffer** que se utiliza únicamente durante el renderizado de la geometría es de 32 bit, para el resto pases y efectos utilizamos una versión reducida a una precisión de 16 bit.

6.3.2. Position Buffer

El *buffer* de posición (Figura 6.6) contiene, para cada fragmento, su posición espacial en *view space*. El formato escogido es de 16bit para cada eje de coordenadas (x,y,z).

El *buffer* de posición y el de profundidad se almacenan juntos en un único *buffer* **RGB16F**. Es por ello que la precisión de la profundidad en pases posteriores se vea reducida, utilizándose 32 bit únicamente durante el *Geometry Pass*.

Figura 6.5: Depth Buffer o Z-Buffer, a mayor claridad, mayor profundidad

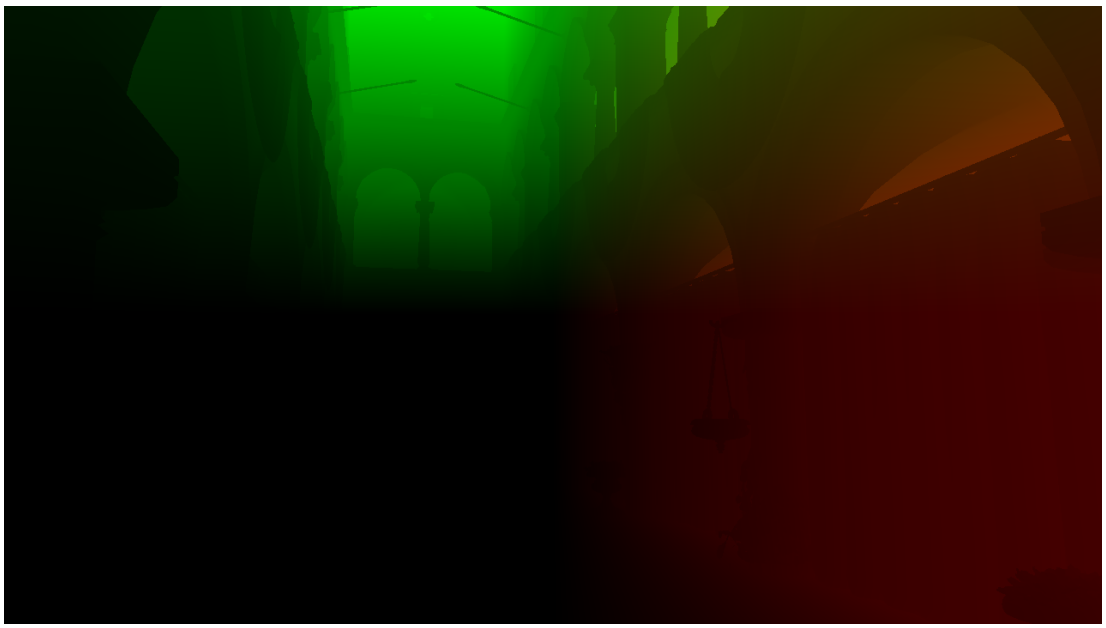


Figura 6.6: Position Buffer, las coordenadas (x,y,z) visualizadas como color RGB

6.3.3. Normal Buffer

Para cualquier superficie, un vector normal es un vector perpendicular a ella en un determinado punto, como puede observarse en la Figura 6.7. El *buffer* de normales permite

conocer la orientación de la superficie de los polígonos proyectados en cada fragmento. Esta orientación formará parte de los cálculos de iluminación, pues permite determinar, entre otras cosas, el grado en el que cada una de las luces de la escena inciden con el fragmento.

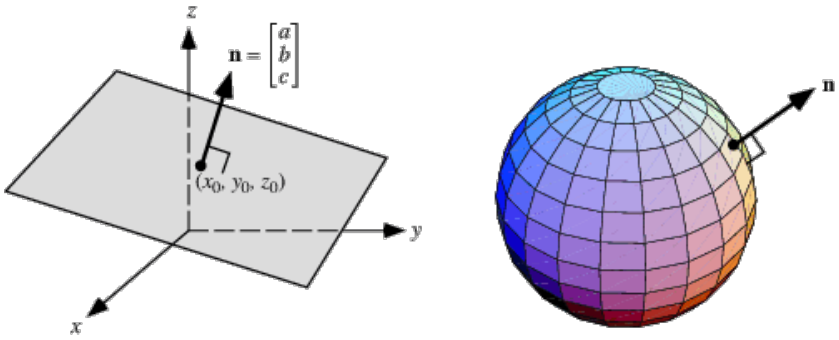


Figura 6.7: Representación del vector normal a una superficie

Como se describió en capítulos anteriores, cuando un polígono es proyectado, éste puede generar múltiples fragmentos (píxeles) adyacentes y por tanto, todos poseerían el mismo vector normal. Es por ello que, para aportar una mayor potencia visual, los vectores normales de los fragmentos son modificados con la aplicación de una textura (contenida como parte del material asignado a cada malla poligonal). Esta textura se denomina "*normal map*" y podemos ver un ejemplo en la Figura 6.8. El mapa de normales permite, en base a él, asignar vectores normales diferentes a fragmentos generados por la proyección de un mismo polígono.

En la Figura 6.9 puede observarse el *Normal Buffer* tras la aplicación de los diferentes mapas de normales durante el renderizado de la geometría. Las normales están también en *view space*, por ejemplo, se puede observar el suelo de un color verdoso, siendo el color verde el segundo canal en formato **RGB**, lo que se corresponde con el eje de coordenadas "y" (el vector apunta hacia arriba).

Para el buffer de normales se ha establecido también una precisión de 16 bit (**RGBA16F**), que codifica los componentes (x,y,z) del vector.

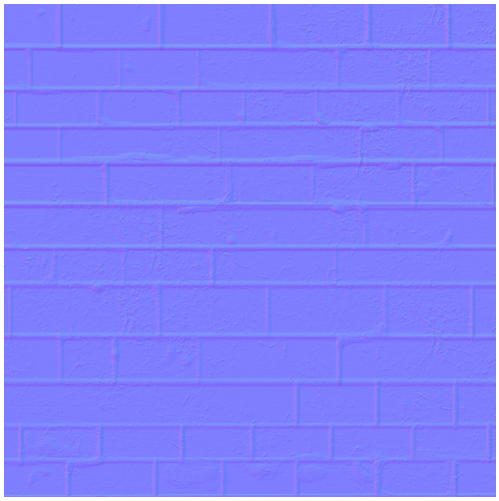


Figura 6.8: Una superficie de ladrillo, ejemplo típico de mapa de normales

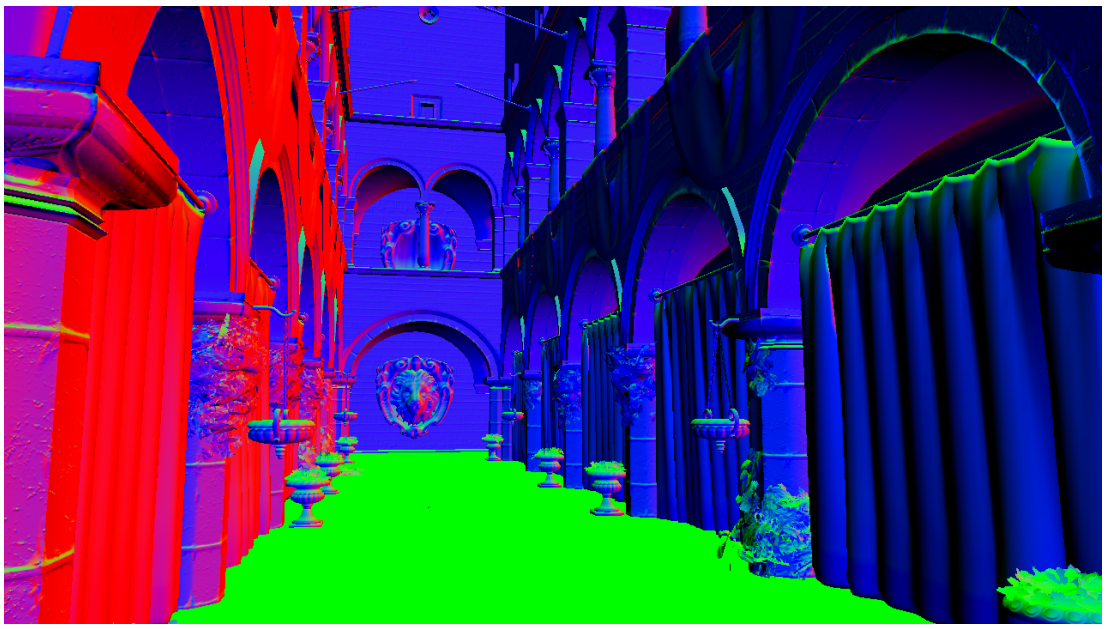


Figura 6.9: Normal Buffer, el vector normal (x,y,z) en view space, visualizado como color RGB

6.3.4. Diffuse Buffer

El *diffuse buffer* almacena el color de los fragmentos finalmente visibles de la escena. De forma similar a lo que ocurría para el *buffer* de normales, la aplicación de una textura sobre la geometría proyectada permite definir el color del fragmento en base a una

imagen, en este caso denominada "*diffuse map*", similar a la que puede observarse en la Figura 6.10.

Figura 6.10: Ejemplo de *diffuse map*

Recordemos que el texturizado es una técnica que permite mapear los vértices de un polígono sobre una imagen, como se muestra en el ejemplo de la Figura 6.11, por lo que, en el caso concreto del *diffuse buffer*, posibilita que un mismo polígono genere fragmentos de diferentes colores en base al mapa aplicado.

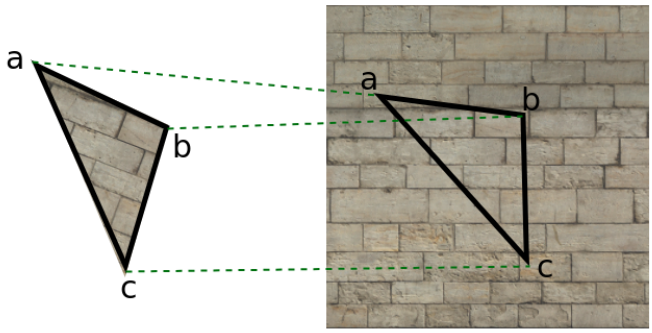


Figura 6.11: Representación del mapeado de texturas

El *diffuse buffer* utiliza el formato **RGB16F** para almacenar el color de cada fragmento (Figura 6.12), el canal **Alpha** restante se reaprovecha para introducir el componente

especular, que será detallado en la próxima subsección.

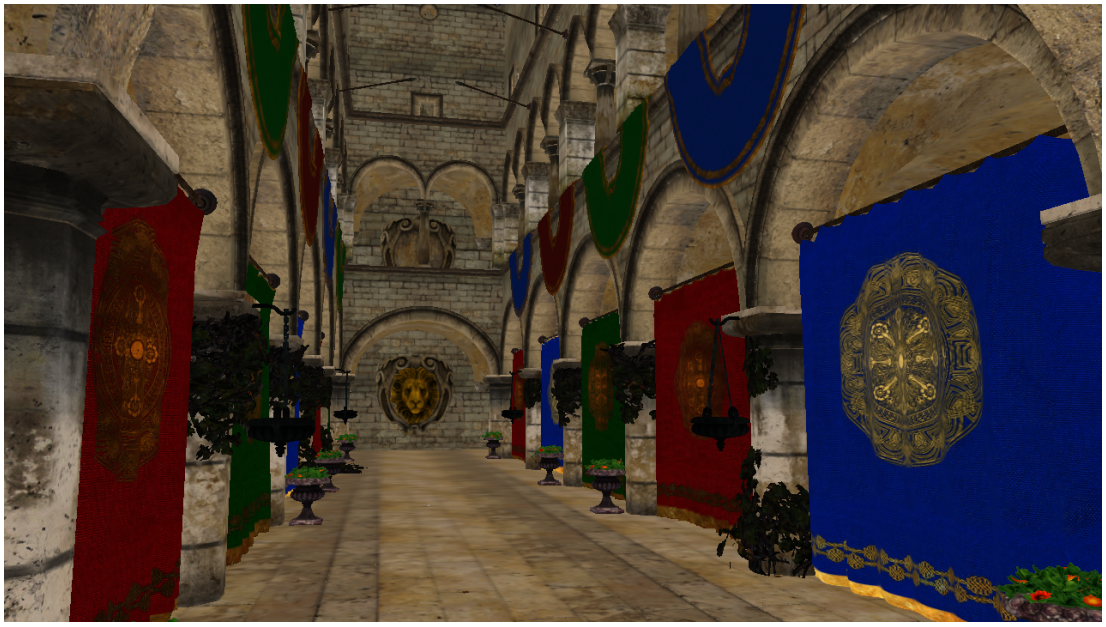


Figura 6.12: Diffuse Buffer, color extraído mediante *texture mapping*

6.3.5. Specular Buffer

La luz especular es aquella que se genera a través de la reflexión de luz en una superficie, cuanto menor sea el ángulo de incidencia con respecto al ángulo de visión, el efecto especular será mayor.

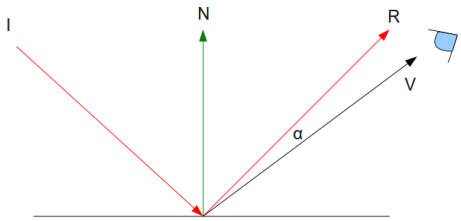


Figura 6.13: Luz especular

En la Figura 6.13: I es la fuente de iluminación, N el vector normal a la superficie, R el rayo de luz reflejado en la superficie (con ángulo de salida igual al ángulo de entrada

con respecto a la normal) y V es el vector vista, es decir, el que define el ángulo de visión sobre el punto en la superficie. Finalmente, *alpha* es el ángulo entre el vector vista y la luz reflejada, cuanto menor sea, mayor brillo producirá la superficie.

El *buffer* especular almacena, para cada pixel de la imagen, un valor de especularidad que determina "cuánto de brillante" es un objeto. Este valor lo determina el material asociado pero, al igual que en ocasiones anteriores, puede ser potenciado a través de *texture mapping*, en este caso a través de la aplicación de un *specular map* (similar al que podemos observar en la Figura 6.14).

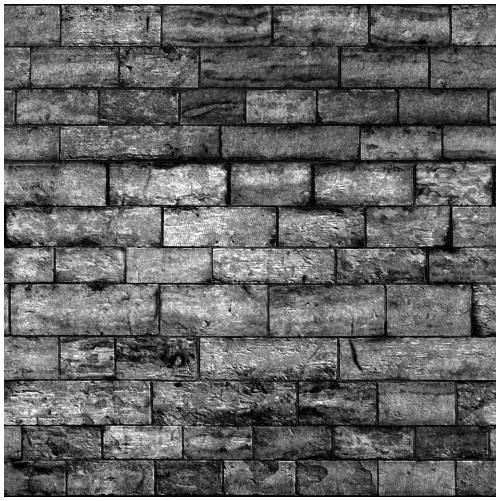


Figura 6.14: Mapa especular

La Figura 6.15 muestra el valor especular de cada fragmento de la imagen; nótese que algunos materiales carecen de mapa especular, y por tanto el valor del material es uniforme a toda la superficie de la malla (como podemos observar, por ejemplo, en las cortinas colgadas de los arcos).

6.3.6. Light Position Buffer

Finalmente se ha añadido un último *buffer* similar al *buffer* de posición, pero cuyas coordenadas se encuentran en un espacio vectorial diferente, en concreto, en el espacio vectorial de la luz direccional principal (Figura 6.16). El espacio vectorial de esta luz se

Figura 6.15: Specular Buffer

determina utilizando la posición de la misma como origen de coordenadas y la dirección en la que apunta como uno de los ejes.

La utilización de este *buffer* es provisional y responde a necesidades concretas de la implementación actual del motor gráfico. La posición de cada fragmento con respecto a la luz principal puede ser calculada directamente durante la fase de iluminación, con lo que se evitarían cálculos innecesarios (fragmentos descartados) y se ahorraría memoria. El objetivo principal de este *buffer* es brindar la capacidad de determinar si un fragmento de la escena se encuentra o no a la sombra, lo que se detallará en secciones posteriores.

6.3.7. Steep parallax mapping

Adicionalmente, y durante la ejecución del *Geometry Pass*, se lleva a cabo un pequeño proceso de *raymarching* sobre ciertas superficies concretas, aquellas cuyo material incluye una textura de profundidad (o su inversa, de altura, Figura 6.17): una imagen en escala de grises que determina la profundidad o altura de la superficie para cada *texel*. Con la aplicación de este efecto conseguimos obtener una sensación de relieve más detallada,

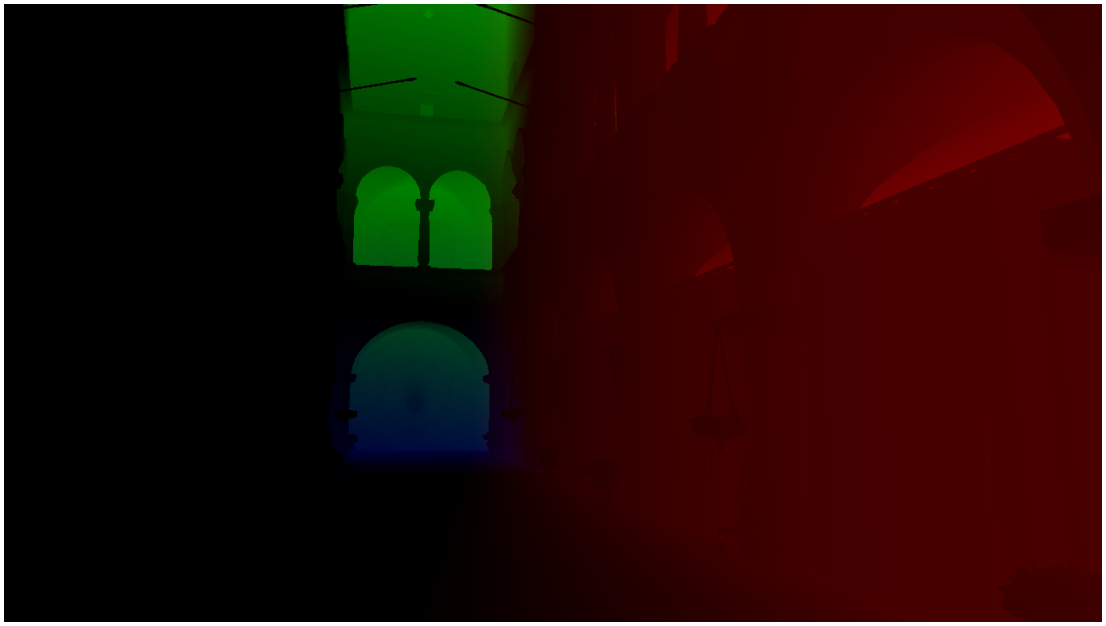


Figura 6.16: Posición en el espacio vectorial de la luz principal

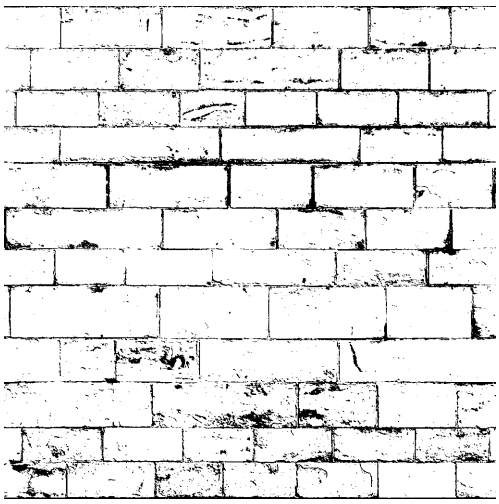


Figura 6.17: Mapa de altura, los colores más oscuros indican mayor profundidad

sobre superficies que, a efectos poligonales, son realmente planas. Esto es gracias a que el *parallax mapping* se ejecuta a nivel de fragmento (*Fragment Shader*) y de esta forma evita el uso de un mayor número de polígonos para poder describir gráficamente la superficie. El efecto permite dotar de mayor detalle a la escena de una forma similar a lo que conseguimos con otros mapas, como el de color o normales: detalle a nivel de fragmento, manteniendo el número de polígonos dibujados lo más bajo posible.

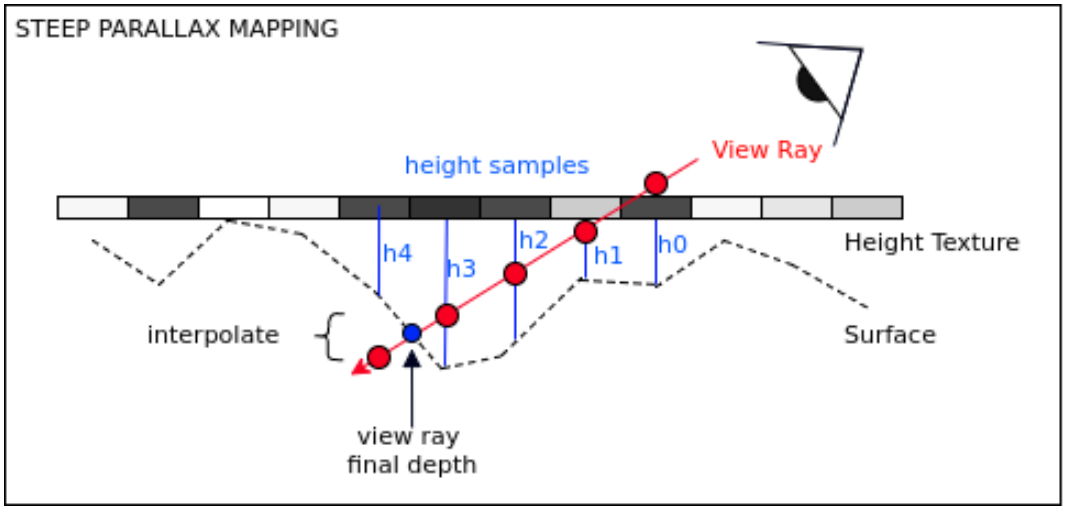


Figura 6.18: Representación del proceso de *marching* sobre una textura bidimensional

En la Figura 6.18 podemos observar una representación del proceso: *marchamos* un rayo en base al ángulo de visión sobre el fragmento, muestreando la profundidad (información proporcionada por el mapa anteriormente mencionado) y, en función de esta profundidad, determinamos si se ha realizado *hit*, es decir, si estamos dentro de la superficie.

Para *marchar* el rayo mientras no se realiza *hit*, se introduce un *offset* que define el siguiente punto de mustreo. Debido al tamaño fijo de este *offset* es posible que el avance del rayo supere la profundidad determinada por la textura. Por este motivo, tras realizar *hit* se calcula el punto real de la superficie, aproximado a partir de la interpolación del punto de *hit* y el punto de muestreo inmediatamente anterior.

EEl *parallax mapping* se realiza antes que la aplicación del resto de texturas: la coordenada real finalmente calculada en el proceso de *marching* es la que utilizaremos para realizar el mustreo de color, normales y otras propiedades del material aplicado sobre la superficie. En otras palabras, el *offset* de profundidad introducido mediante este efecto modifica también la aplicación del resto de texturas, generando de forma efectiva profundidad sobre los polígonos planos.

Si bien es verdad que es un efecto que realmente destaca más durante el movimiento de la cámara, en la Figura 6.19 podemos ver de forma comparativa el aspecto visual que

este efecto proporciona, por ejemplo, sobre el material de ladrillos con el que venimos trabajando en los últimos apartados.

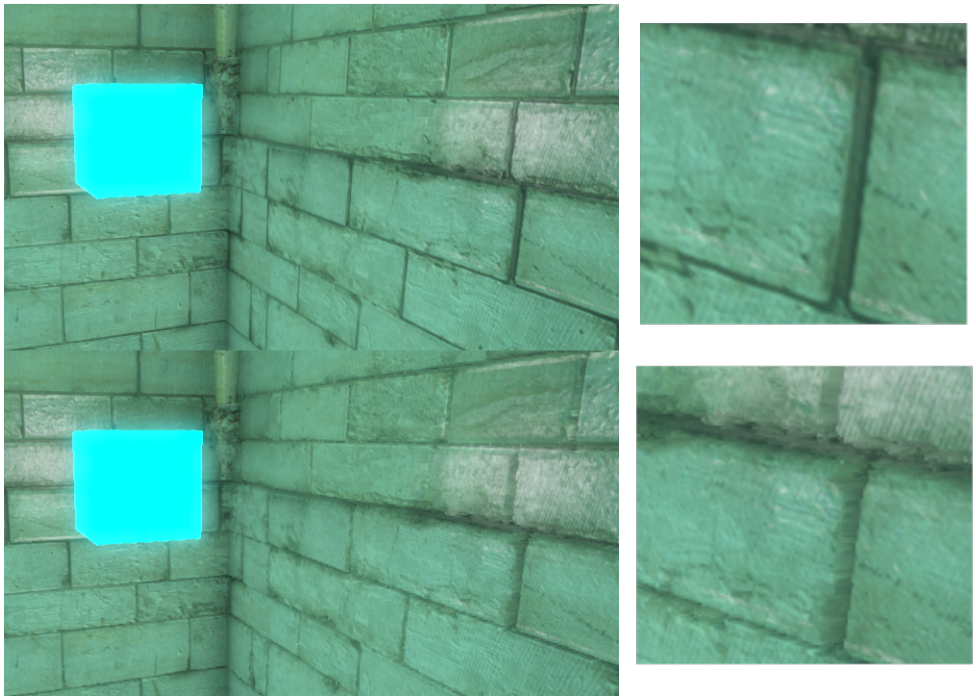


Figura 6.19: Efecto de relieve conseguido con *steep parallax mapping*

6.4. Lighting Pass

Como ya se explicó en la introducción al *Deferred Rendering*, el *Lighting Pass* es el pase que recoge toda la información generada durante las múltiples ejecuciones del *Geometry Pass* (una por cada modelo) y ejecuta el proceso de iluminación de la geometría en espacio de pantalla (*view space*), es decir, únicamente sobre los fragmentos finalmente visibles de la imagen.

En cuanto a la implementación se refiere, el **LightingRenderer** recibe, no sólo la información los pases geométricos, sino también la resultante de otras fases que se explicarán más adelante y que combina con la propia iluminación mencionada. Sin embargo, resulta conveniente detallar el proceso de iluminación primero y por separado, pues la aplicación del resto de resultados durante esta fase es meramente una decisión práctica.

6.4.1. Phong Shading

El *Phong Shading* [17] es uno de los modelos de *shading* (sombreado) que aproximan, de una forma simplificada, la interacción de la luz con las superficies del mundo real. El modelo de luz *Phong* (Figura 6.20) distingue tres componentes, que sumados, producen el efecto de iluminación deseado:

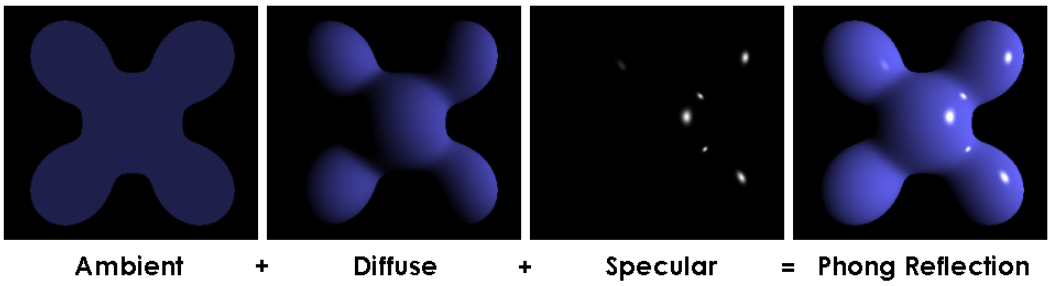


Figura 6.20: Componentes de iluminación en *Phong Shading*

- **Ambient lighting:** El componente de luz ambiental es un valor constante cuyo objetivo es eliminar la posibilidad de que en la imagen aparezcan zonas completamente oscuras, añadiendo al menos una pequeña cantidad de iluminación a todos los fragmentos de la escena dándoles algo de color.

$$CONST\ ambient\ =\ ambient\_factor * light\_color$$

- **Diffuse lighting:** El componente difuso simula el resultado del reflejo de la luz sobre un objeto, es el componente principal y por tanto determina en gran parte el color final de la superficie iluminada.

  Para obtener el componente difuso, se calcula el ángulo de incidencia de la luz sobre la superficie en cada fragmento concreto (en base su vector normal, almacenado en el *Normal Buffer*). Cuanto mayor sea la perpendicularidad del rayo con respecto a la superficie, mayor será la contribución de la luz.

$$diffuse = MAX(DOT(normal, light\_direction), 0) * light\_color$$

- **Specular lighting:** El componente especular simula el brillo que aparece en las superficies de los objetos brillantes. El efecto de este resplandor es mayor cuanto

menor sea el ángulo formado entre: la dirección de la vista, con respecto al fragmento, y la dirección del rayo de luz reflejado en el mismo.

Su valor se calcula de la forma siguiente, donde *shininess* es el valor almacenado en el *Specular Buffer*, uno de los resultados parciales producto del *Geometry Pass* que permite controlar la fuerza del brillo final.

$$specular = MAX(DOT(view\_direction, reflect\_direction), 0)^{shininess}$$

Finalmente se combinan los tres componentes descritos y este resultado se aplica sobre el color de la propia geometría, es decir, la información contenida en el *Diffuse Buffer*.

$$fragment\_color = (ambient + diffuse + specular) * diffuse\_color$$

6.4.2. Blinn-Phong Shading

El modelo *Blinn-Phong* [18] es una extensión del modelo *Phong* pero que realiza una aproximación diferente: en vez de hacer uso del vector de luz reflejado, calcula un llamado "vector de media distancia", que en efecto, es un vector unitario que se sitúa exactamente entre la dirección de la vista y la de incidencia de la luz. De esta manera, cuanto más se alinea el "vector de media distancia" con la normal del fragmento, mayor es la contribución del componente especular de la luz.

$$halfaway\_dir = NORMALIZE(light\_direction + view\_direction)$$

$$specular = MAX(DOT(view\_direction, halfaway\_dirrection), 0)^{shininess}$$

La fuerza del componente especular en *Blinn-Phong* es ligeramente superior a la de su homónimo, por lo que para obtener un mismo resultado es necesario aumentar el exponente *shininess*, como puede observarse en la Figura 6.21

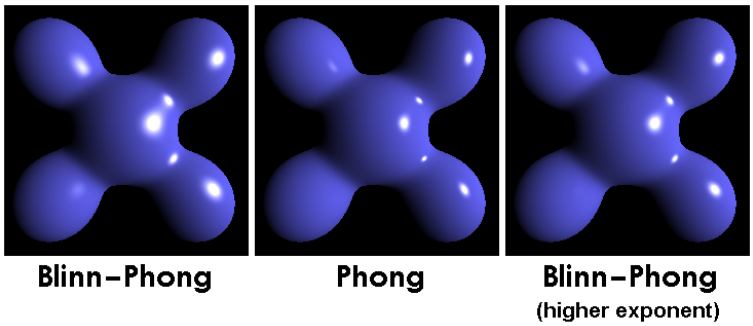


Figura 6.21: Comparativa entre la extensión *Blinn-Phong* y *Phong shading*

6.5. SSAO Pass

6.5.1. Ambient Occlusion

En el mundo real, cuando dos objetos se encuentran suficientemente próximos, la luz queda, en parte, atrapada en la zona del acercamiento, reflejándose múltiples veces entre los objetos y perdiendo energía, por lo que la zona se oscurece a la vista.

La oclusión ambiental (*Ambient Occlusion*), en el contexto de gráficos por computador, es un término que agrupa un conjunto de técnicas y algoritmos utilizados para determinar cuánto de expuesto se encuentran los puntos de una escena, ello permite simular el oscurecimiento de las zonas donde la geometría se encuentra por debajo de una determinada distancia (a menudo denominada frecuencia).

Los algoritmos de oclusión ambiental suelen ser muy costosos en términos computacionales, esto es debido principalmente a la necesidad de tomar en consideración toda la geometría al rededor de un mismo fragmento, ya no es una operación tan paralelizable que pueda realizarse en todos los fragmentos de forma independiente (como las vistas anteriormente).

En este contexto surgen múltiples aproximaciones, entre las cuales destaca el denominado SSAO (*Screen Space Ambient Occlusion* o Oclusión ambiental en espacio de pantalla).

Figura 6.22: Modelo sin oclusión ambiental y con oclusión ambiental

6.5.2. SSAO

SSAO (Screen Space Ambient Occlusion) [19] es una técnica de oclusión ambiental que puede ser ejecutada en espacio de pantalla (*screen space*) haciendo uso únicamente de la información de profundidad almacenada en el *Depth Buffer* y las normales de los fragmentos, disponibles en el*Normal Buffer*.

El fundamento de SSAO es sencillo: para cada fragmento de la imagen, se calcula un "factor de oclusión" en base al análisis de la profundidad (*Depth Buffer*) de los fragmentos que lo rodean. Este factor se utiliza posteriormente para reducir su componente de luz ambiental (que como vimos en el *Lighting Pass*, era un valor constante en toda la escena).

Para realizar este proceso de muestreo o *sampling* sobre el *Depth Buffer*, se propuso

originalmente el uso de una zona esférica al rededor del fragmento, representado en la Figura 6.23, de la que se extraen muestras en posiciones pseudoaleatorias.

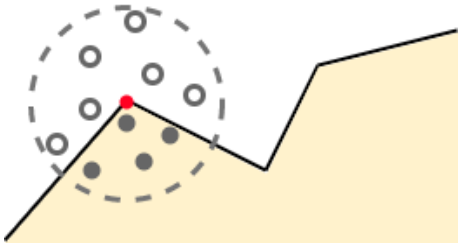


Figura 6.23: Muestreo esférico del *Depth Buffer* en SSAO

Sin embargo, el uso de una zona de muestreo esférico tiene un gran inconveniente: en ciertas situaciones, como puede ser una pared lisa, aproximadamente la mitad de las muestras se dispersarán dentro de la propia pared, haciendo que ésta se oscurezca cuando en realidad puede no existir ninguna geometría adyacente a ella que realmente lo haga.

La solución al problema descrito reside en el uso de sólo un hemisferio de la esfera, en concreto, uno que orientaremos con el vector normal del fragmento, de esta forma no será considerada la geometría que se encuentre por detrás de él. En la Figura 6.24 podemos observar dos muestreos con una zona hemisférica, nótese como en el fragmento de la izquierda el factor de oclusión es nulo, algo que no ocurriría en el caso anterior.

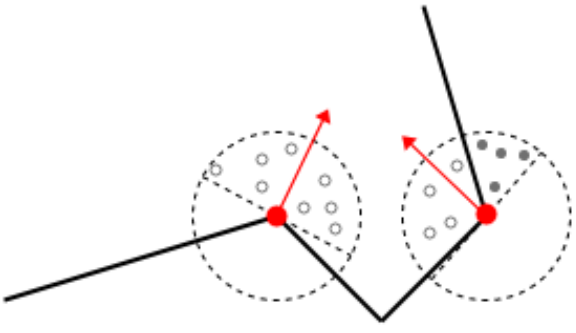


Figura 6.24: Muestreo hemisférico del *Depth Buffer* en SSAO

Una forma de mejorar notablemente el resultado del efecto y obtener resultados visualmente más realistas, es controlando en cierta medida la “pseudoaleatoriedad” de las

muestras dentro de la zona hemisférica, en concreto, agrupando las muestras de una forma más densa en torno a la posición del fragmento y dándo a estas un mayor peso en la contribución al factor de oclusión [20].

Si a esto sumamos una pequeña rotación del hemisferio alrededor de la normal del fragmento, conseguiremos incrementar la cantidad de muestras tomadas en torno al mismo. Para ello, se suele utilizar una pequeña textura (4x4, 8x8, 16x16), generada de forma aleatoria, que se repite a lo largo y ancho de toda la imagen.

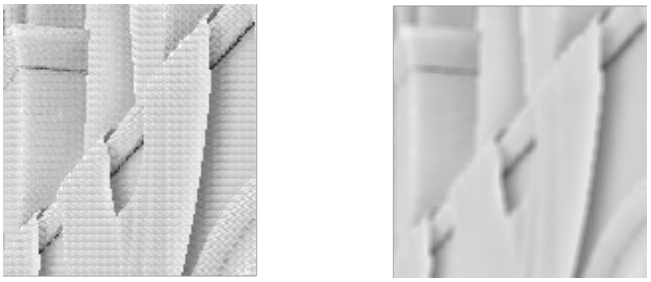


Figura 6.25: A la izquierda, resultado del algoritmo SSAO. A la derecha, tras aplicar un filtro de desenfoque (*blur*)

Los valores contenidos en esta textura determinan la rotación del hemisferio, pero dado que la misma se repite cada ciertos píxeles, se genera un efecto de cierta regularidad en el resultado final del algoritmo. La solución para este problema es mantener la textura lo suficientemente pequeña de forma que podamos aplicar un filtro de desenfoque que elimine esta alta frecuencia de la imagen. En la Figura 6.25 puede observarse el antes y después del filtro de desenfoque. En la Figura 6.26 podemos ver el contenido del *buffer* de oclusión ambiental en un momento determinado.

6.6. Shadow Map Pass

El *Shadow Map Pass* es un pase sencillo en el que se renderiza, con una proyección ortográfica, exclusivamente la profundidad (*depth*) de la geometría de la escena, pero esta vez desde el punto de vista de la luz principal (que debe ser de tipo direccional).

Figura 6.26: Ejemplo de contenido del buffer de oclusión ambiental

Este mapa nos permite comparar la profundidad de los fragmentos con respecto a la posición que ocupan en el ya mencionado *Light Position Buffer*. De esta forma podemos conocer si un determinado fragmento es visible desde la luz, y de no serlo entonces sabemos que se encuentra en sombra, proceso representado en la Figura 6.28.

Al ser necesario redibujar toda la escena (aunque de una forma simplificada) esta fase tiene un coste muy alto, y si el mapa resultante no tiene resolución suficiente, se generarán muchos artefactos para las sombras. Es por ello que el *shadow map* se renderiza una única vez en tamaño muy alto al inicio, y no se repetirá todos los fotogramas.

En la Figura 6.29 podemos ver un ejemplo de *shadow map*, la misma escena ha sido renderizada desde el punto de vista de la luz, por lo que vemos el tejado y parte de la arquería del *Sponza Atrium* (la escena de referencia).

6.6.1. Shadow Acne y Shadow Bias

Uno de los artefactos visuales más notables a la hora de implementar la técnica de *shadow mapping* es el llamado *shadow acne*, similar al que puede verse en la Figura 6.30).

Figura 6.27: A la izquierda, modelo con iluminación y SSAO. A la derecha, se ha aplicado *shadow mapping*

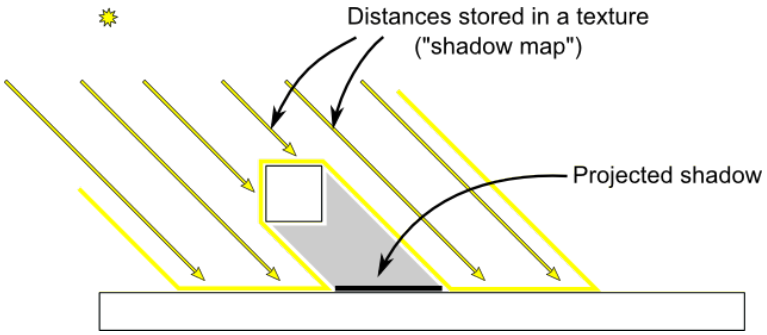


Figura 6.28: Shadowmapping

El problema deriva del carácter limitado de la resolución del *shadowmap*, y por tanto, las muestras de profundidad de algunos fragmentos pueden dar falsos positivos a la hora de determinar si están o no en sombra.

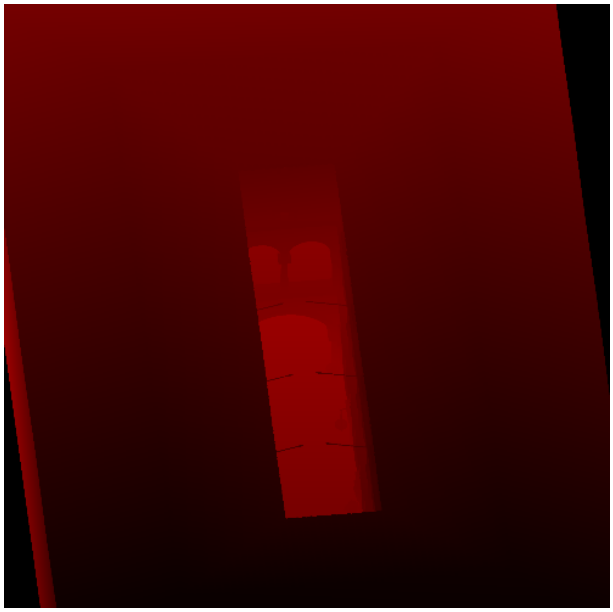


Figura 6.29: Shadowmap

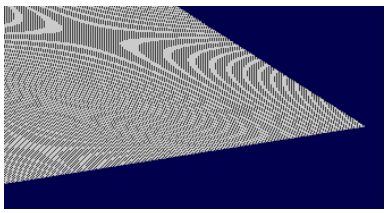


Figura 6.30: Ejemplo del artefacto visual *shadow acne*

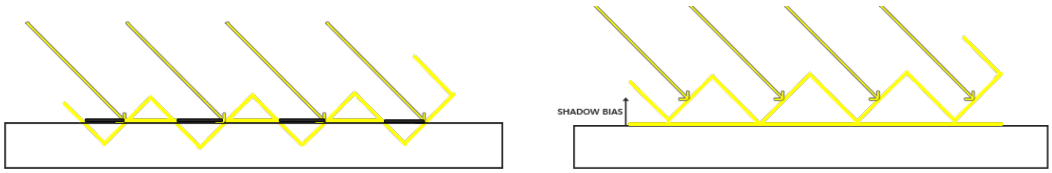


Figura 6.31: *Shadow acne*, a la izquierda, falsos positivos a causa de la resolución limitada, a la derecha, aplicación del *shadow bias*

Sin embargo, existe un pequeño “truco” denominado *shadow bias* [21] que resulta tan sencillo como incluir un pequeño *offset* a la hora de tomar mediciones de la profundidad en el *shadow map*. En la Figura 6.31 se encuentran representadas las dos situaciones.

6.6.2. Percentage-Closer Filtering (PCF)

La técnica de *shadow mapping* produce lo que se denominan como *hard shadows* o sombras "duras", sombras de bordes afilados, pues sólo determina si los fragmentos se encuentran o no a la sombra. En contraposición, las *soft shadows* o sombras "suaves", son aquellas en las que existe una penumbra, es decir, el borde de la sombra se difumina con la distancia. Podemos ver una comparación en la Figura 6.32

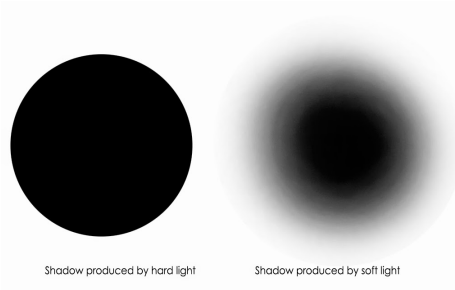


Figura 6.32: A la izquierda *hard shadow*, a la derecha *soft shadow*

El uso de una PCF (Percentage-Closer Filtering) [22] puede mitigar el efecto de las sombras "duras". PCF es un término utilizado para designar un conjunto variado de funciones de filtrado que permiten producir sombras más suaves, estas funciones, en esencia, realizan un muestreo múltiple del *shadow map* (a diferencia de la única medición realizada hasta ahora) y determinan una media en base a los *texels* que se encontraban a la sombra y los que no. Este factor permite así obtener un pequeño efecto de difuminado.

6.7. Bloom Pass (Gaussian Blur)

Debido a que el rango de intensidad o luminosidad de los monitores es limitado, para crear cierta sensación de luminosidad en la imagen se utiliza un efecto de post-procesado conocido como *Bloom* ("resplandor"), el resultado de este efecto puede distinguirse claramente en la Figura 6.33, aunque su aplicación suele ser más disimulada en que la imagen mostrada.

Figura 6.33: Efecto del *Bloom* sobre las zonas brillantes de la imagen

El efecto, en esencia, es sencillo: al finalizar el *Lighting Pass* (y aplicar el shadowmap), se extraen, a un *buffer* diferente (que denominamos *Brightness Buffer*, Figura 6.34), aquellos fragmentos que superan cierto umbral de luminosidad. Posteriormente a este *buffer* se le aplica un filtro de desenfoque de forma repetitiva, difuminando la imagen tantas veces como se desee.

Finalmente esta imagen se multiplica por el resultado del sombreado (*Lighting Pass*), de forma que se obtiene un "halo" alrededor de las zonas más brillantes de la escena, es decir, aquellas que participaron en el proceso de desenfoque.

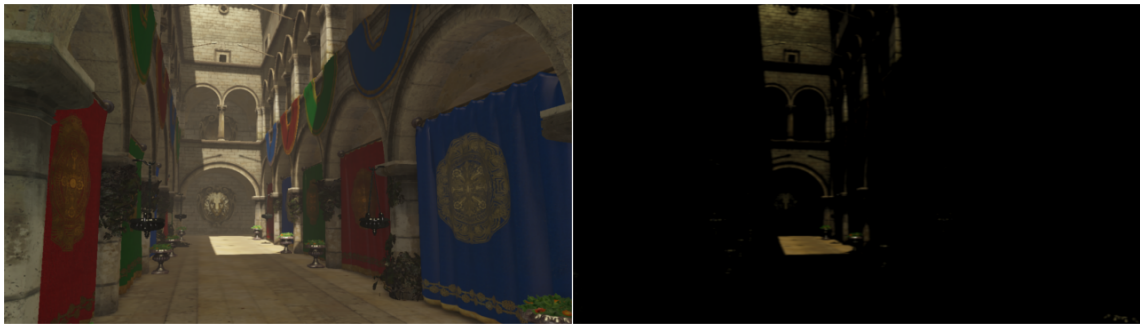


Figura 6.34: A la izquierda, resultado del *Lighting Pass*, a la derecha, contenido del *Brightness Buffer*

6.8. Antialiasing: FXAA Pass

En términos de computación gráfica, el *aliasing* es un defecto gráfico que ocurre de forma natural al tratar de representar patrones con altas frecuencias en la resolución limitada de un monitor digital. Realmente es un problema de muestreo relacionado también con disciplinas como el procesamiento de señales.

En la Figura 6.35 podemos ver un ejemplo del efecto en cuestión. En este caso, al escalar la imagen, el patrón de ladrillos de la pared aumenta su frecuencia, ya que la resolución disponible es mucho menor para representar el mismo detalle. De esta manera, en la miniatura se genera un efecto conocido como patrón de Moiré, que debe su nombre a la seda de Moiré cuyos finos filamentos producen el mismo patrón de interferencia, pero en el mundo real.

De entre las diferentes técnicas de *anti-aliasing*, FXAA (Fast Approximate Anti-Aliasing) [23] destaca por ser la única capaz de realizar *anti-aliasing* directamente sobre los píxeles de la imagen final, a diferencia de otras mucho más costosas que requieren del uso de más de un fragmento por pixel durante el renderizado, como SSAA (Super-Sampled Anti-Aliasing) o MSAA (Multi-Sampled Anti-Aliasing).

Entre sus ventajas, FXAA puede llegar a ser hasta cuatro veces más rápido en el mejor de los casos, aligera enormemente el ancho de banda en la GPU y es compatible tanto con *Forward rendering* como con *Deferred rendering*. En la Figura 6.36 puede verse la

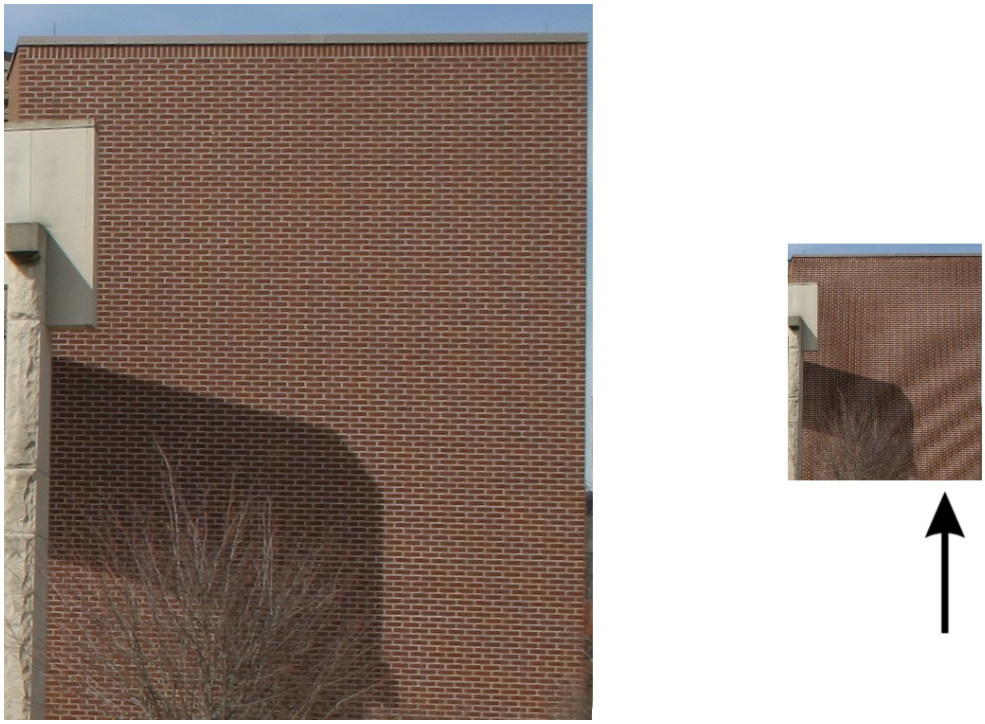


Figura 6.35: Patrón de interferencia de Moiré

misma escena renderizada en baja resolución de forma que se fuerce *aliasing* en sus ejes. Compárese el resultado con y sin el efecto de FXAA.

El algoritmo comienza con una fase de detección de ejes en la imagen (Figura 6.37); para ello compara la luminosidad de cada fragmento con respecto a la de sus adyacentes, de forma que se detecten aquellas zonas que sufren un mayor efecto de *aliasing*. Todos aquellos fragmentos que no pertenezcan a estas zonas no serán modificados.

Posteriormente, y sólo para aquellos fragmentos detectados durante el inicio, se determina la dirección del eje en el que se encuentran y se aplica un desenfoque tomando muestras de color de los fragmentos adyacentes (en base a la dirección del eje), difuminando de forma efectiva todos los bordes de la imagen, producidos tanto por la propia geometría como por altas frecuencias en las texturas.

Figura 6.36: Comparación del efecto de anti-aliasing con FXAA

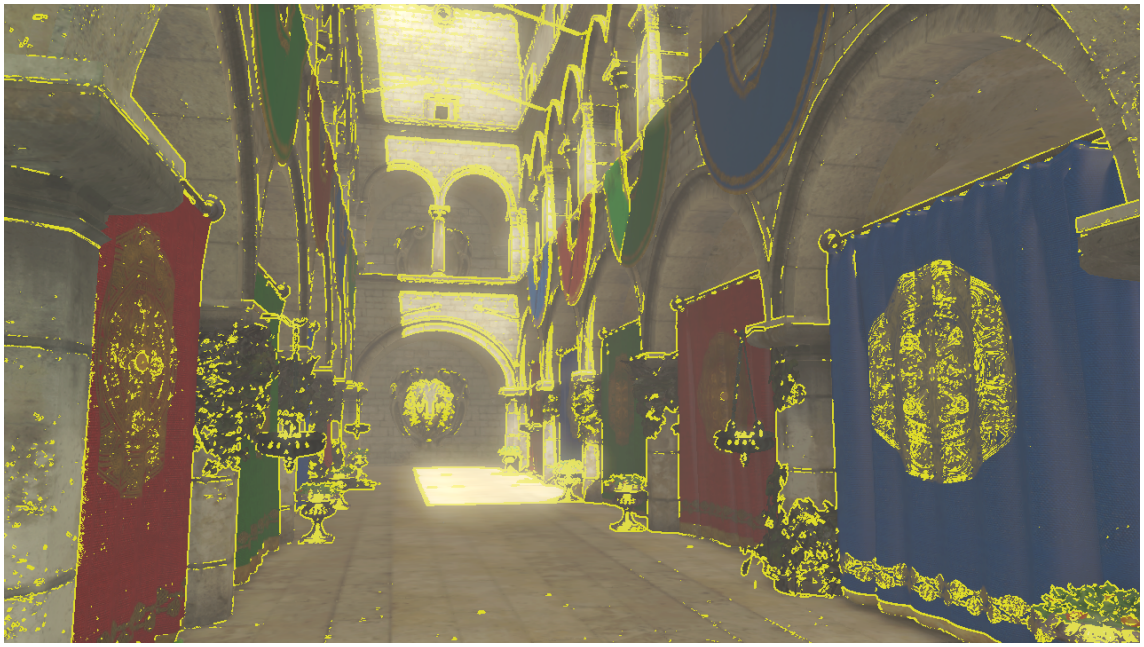


Figura 6.37: Visualización de los ejes detectados durante FXAA

6.9. HDR, Color Mapping y Corrección Gamma

Finalmente, en la última fase del *pipeline* se realizan tres operaciones destinadas únicamente a la corrección de color de los píxeles resultantes de todas las fases anteriores, y cuyo resultado conforma el fotograma final que se sirve al usuario.

- **High Dynamic Range (HDR)**: Los monitores digitales acotan el valor numérico del color de los píxeles en el rango 0.0 a 1.0, denominado LDR (Low Dynamic Range). Sin embargo, durante los cálculos de iluminación es posible que existan fragmentos cuyo color, debido a la luminosidad existente en la escena, exceda este rango.

  Ante esta situación, el valor de color de los fragmentos que exceda de 1.0 será restringido al máximo, es decir 1.0, el color blanco. Esto hace que todas las zonas luminosas de la escena se vean con el mismo tono de blanco, perdiéndose una enorme cantidad de detalle. Para añadir la capacidad de interpretar zonas de mayor luminosidad, sin perder detalle en otras más oscuras o a la inversa, es necesario, seguir una serie de pasos.

  Por un lado, trabajar directamente con un rango mayor de valores (HDR) a lo largo del resto del *pipeline*, haciendo uso de *render targets* en punto flotante. En nuestro caso, y como ya se mencionó anteriormente, trabajamos con colores de 16 bit para cada canal RGBA. Y por otro, necesitamos convertir estos valores HDR a LDR como fase final, a través de diferentes ecuaciones y/o curvas, de forma que la imagen resultante, conserve los detalles y matices tanto en las zonas más brillantes como en las oscuras.

- **Color Mapping:** El proceso de *color mapping* es simple; para cada fragmento de la imagen, se utiliza su color como índice en una *Lookup Table* (LUT) que devuelve un nuevo color correspondiente (o mapeado). La LUT que se utiliza es cargada como una textura (Figura 6.38) sobre la que se tomarán las muestras y que podrá ser modificada a través de cualquier software de edición.

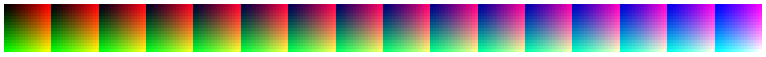


Figura 6.38: 24bit RGB LUT (*Lookup Table*)

La LUT de la Figura 6.38 contiene el conjunto de todos los colores posibles con 8 bits por canal (RGB8) a través de los cuales podremos interpolar un mayor rango de valores, por ello su tamaño es 256x16: cada *chunk* que se aprecia en la imagen contiene la combinación de los dos primeros canales (Red, Green) con el tercer canal (Blue) que se reparte linealmente a lo ancho de la imagen.

Otras alternativas disponen la LUT como una textura tridimensional, lo que aporta una serie de ventajas, sobretodo a la hora de interpolar valores en las GPUs modernas, aún así las representaciones son equivalentes (nuestra imagen desarrolla la tercera dimensión a lo ancho).

- **Corrección Gamma:** La corrección gamma es una operación no lineal, una codificación, que se realiza sobre el color final del fotograma para compensar ciertas propiedades de la visión humana. Esta corrección permite maximizar el ancho de banda en *bits* del color y luminancia de la imagen en base a cómo el ser humano percibe la iluminación en condiciones normales, que se define aproximadamente como una función potencial de la forma:

$$OUT = c * IN^{1/gamma}$$

Donde **c** es una constante, que en el caso más común equivale a 1. **IN** y **OUT** son respectivamente la entrada y salida de la codificación, cuyos valores (y como se mencionó para HDR) están acotados entre 0.0 y 1.0. En la Figura 6.39 se puede apreciar el efecto de la corrección gamma sobre la imagen final, que incluye también HDR y *Color Mapping*.

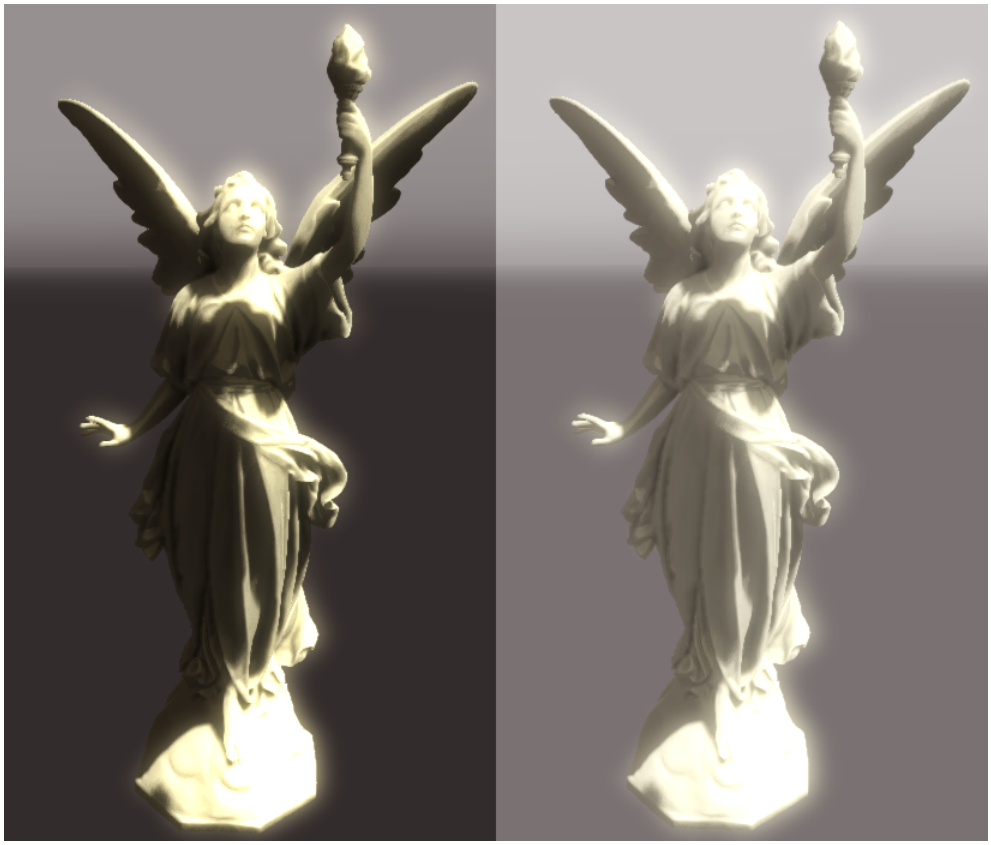


Figura 6.39: A la derecha corrección gamma (valor 2,2) y HDR

Capítulo 7

PIPELINE VOXELS

A lo largo de este capítulo se presentan las funcionalidades de voxelización y renderizado volumétrico integradas como parte del *pipeline*, profundizando en las técnicas desarrolladas y describiendo de forma detallada las distintas posibilidades que ofrece la representación volumétrica de una escena poligonal.

7.1. Voxelizado

Se denomina voxelizado al proceso de convertir una determinada representación gráfica en un conjunto de voxels. El voxelizado permite, a grandes rasgos, obtener una representación regular que pueda ser manipulada de formas en las que otras se encuentran excesivamente limitadas.

En el caso concreto de los polígonos, una representación en voxels de una escena tridimensional contiene información espacial que no se encuentra presente durante el rasterizado. Su inherente regularidad la hacen idónea para procesos de *casting*, de donde obtiene su gran potencial. El conjunto de voxels puede ser atravesado y muestreado espacialmente con facilidad, lo que permite aplicar al rasterizado técnicas como iluminación global, efectos volumétricos, incluso una mejor oclusión ambiental.

7.1.1. Voxelizado CPU

Para realizar la voxelización de polígonos en la CPU sólo es necesaria la lista de triángulos correspondiente a la malla a voxelizar y la realización de un test de intersección [24] entre estos triángulos y las diferentes AABBs (*Axis Aligned Bounding Box*) correspondientes a cada voxel.

En su versión más sencilla, para cada triángulo de la malla, se determina su intersección con la "caja" o volumen (AABB) que ocupa cada voxel. Si efectivamente se interseca triángulo y voxel, entonces podemos determinar que el voxel existe, es decir, el área espacial que representa ese voxel no se encuentra vacía. Esta aproximación cuenta con una complejidad de $O(número\_triangulos \times número\_voxels)$ por lo que el aumento del número de triángulos o el aumento de la resolución del área voxelizada (es decir, la utilización de voxels más pequeños) repercuten de forma muy negativa sobre el rendimiento.

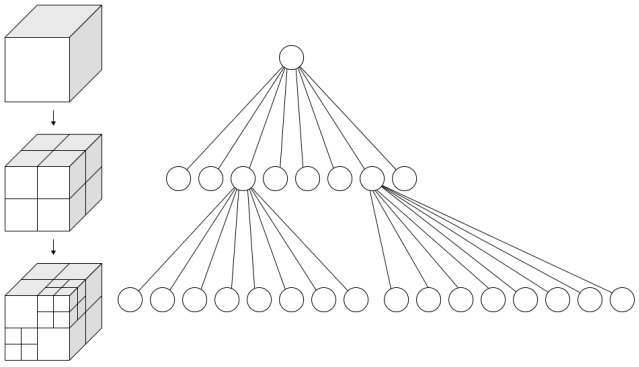


Figura 7.1: Representación de las subdivisiones de un *Octree*

En nuestro caso, y para acelerar enormemente el procedimiento, se ha utilizado una estructura de *Octree* (Figura 7.1), como la descrita en capítulos anteriores, a través de la cual guiamos los test de intersección. Con el *Octree* podemos descartar grandes conjuntos o *clusters* de voxels (octantes de las diferentes ramas del árbol) realizando directamente los test de intersección sobre aquellos voxels que potencialmente contienen cada triángulo, ahorrando una importante cantidad de cálculos. Esta solución cuenta con una complejidad aproximada de $O(número\_triangulos \times log(número\_voxels))$

La voxelización CPU nos permite obtener una fiel representación volumétrica de los

modelos poligonales, aunque para cada voxel sólo sea posible determinar su ocupación y, en todo caso, el vector normal o color obtenido directamente de la información que porta cada vértice. Por tanto, en CPU no es posible calcular información de grano fino, como la que aportan las texturas que se mapean sobre el modelo durante el rasterizado.

Las texturas residen en la memoria de la GPU, y aún manteniendo una copia en CPU, carecemos de la capacidad de interpolar estos valores sobre el área de cada triángulo, algo en lo que el rasterizado es extremadamente eficiente.

Figura 7.2: Voxelizado en CPU con una resolución 1024^3 voxels

Figura 7.3: Voxelizado en CPU con una resolución 64^3 voxels

7.1.2. Voxelizado GPU

Para solventar la incapacidad de realizar *Texture Mapping* en CPU (o la ineficiencia que supondría una implementación destinada para ello), se han estudiado técnicas de voxelizado poligonal directamente en GPU [25].

Estas técnicas permiten realizar una voxelización de grano mucho más fino, tomando en consideración, para cada voxel, los valores normales y de color interpolados sobre el área de cada triángulo, en base a la información contenida en las texturas que se aplican sobre el modelo durante el rasterizado.

La implementación GPU aprovecharía de una forma extremadamente más eficiente el carácter paralelo de todas las operaciones necesarias en el voxelizado. Incluso, y depen-

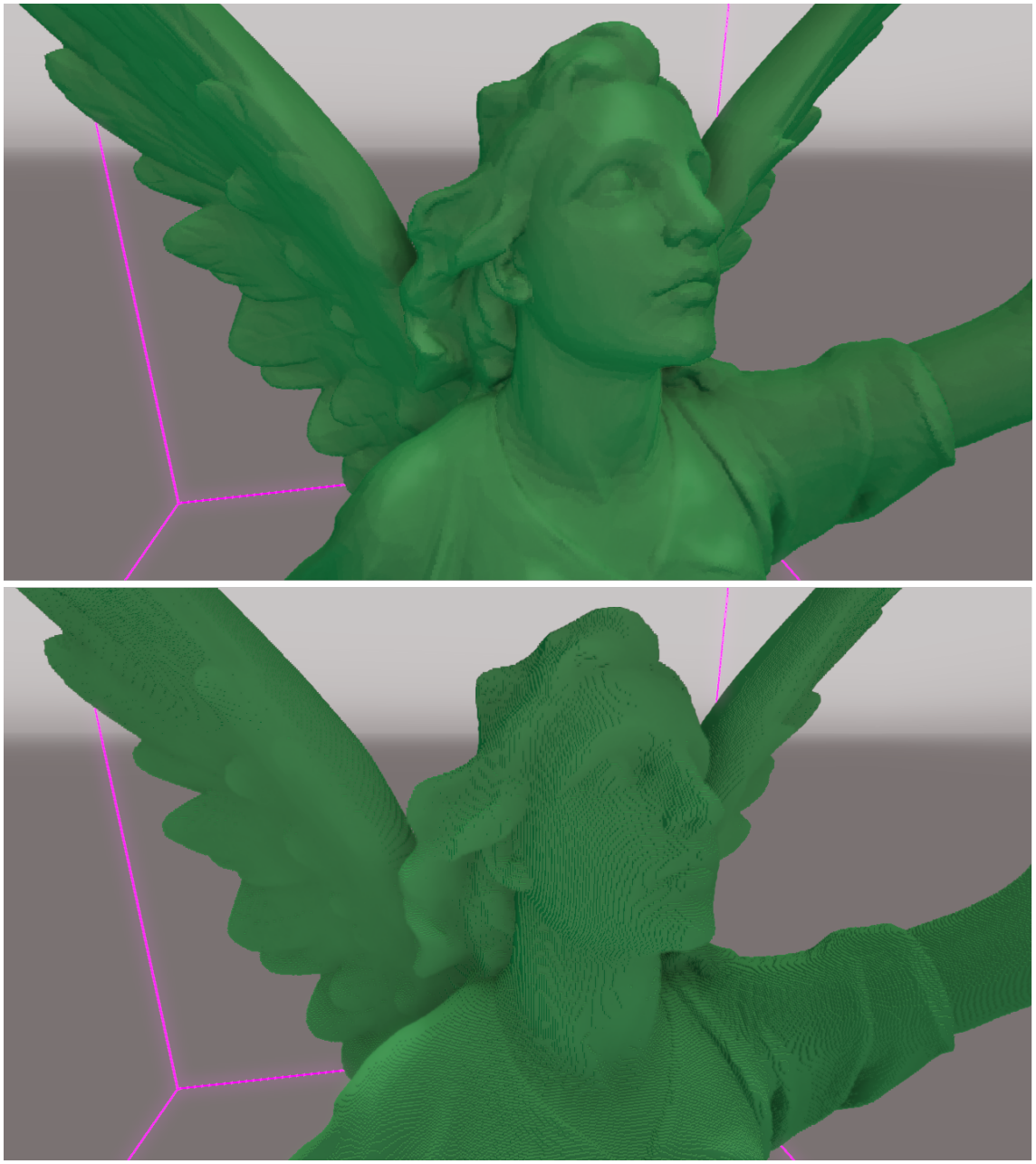


Figura 7.4: Arriba, voxels con vector normal extraído de cada triángulo. Abajo, facetas de los voxels cúbicos

diendo del *hardware* utilizado y resolución en número de voxels, posibilitaría la voxelización en tiempo real: un objetivo a perseguir si queremos dotar de dinamismo a la escena y a los efectos derivados de la representación voxelizada.

El voxelizador GPU aprovecha la estructura y funcionamiento del propio *pipeline* de

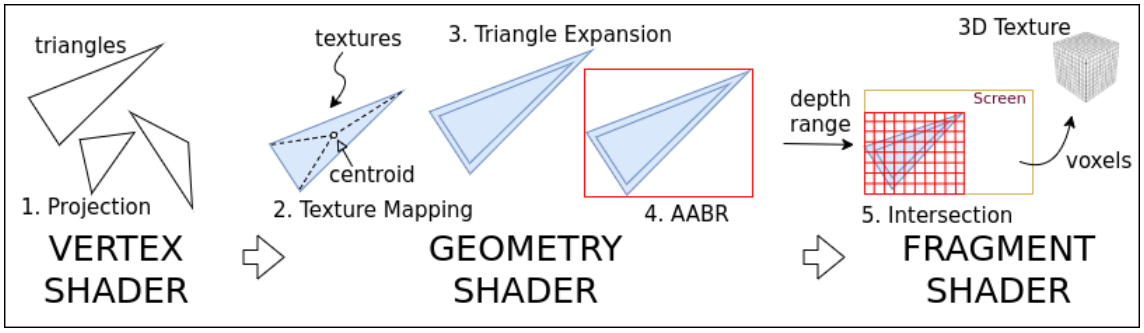


Figura 7.5: Pipeline de voxelizado en GPU

OpenGL [26]. El voxelizador comienza proyectando los polígonos como normalmente (*Vertex Shader*), pero haciendo uso de una cámara ortográfica (similar a la utilizada para *Shadow Mapping*).

Posteriormente, y a diferencia que en el rasterizado, estos polígonos son recogidos por un *Geometry Shader* que realiza tres funciones:

- *Texture Mapping*: El *Geometry Shader* se encarga de asignar a cada polígono un único color y una única normal, muestreo que se toma a partir de su centroide. Esto es así debido a que sólo existirá una única instancia de este *shader* (para cada polígono): recordemos que el *Fragment Shader* es el que posteriormente se ejecuta para cada pixel (fragmento) generado por el polígono, donde suele realizarse el mapeo de texturas normalmente.

- *Triangle Expansion*: Para realizar una voxelización conservativa y que no aparezcan "huecos" en el resultado, se expande el polígono exactamente el tamaño de un pixel en cada eje. De esta forma aseguramos la generación de fragmentos que cubran todo el área del triángulo proyectado. Asi mismo, se calcula el rango de profundidad del polígono, es decir, se determina la profundidad mínima y máxima que éste cubre con respecto al plano de proyección.

- *AABR*: Como último paso se determina el *Axis Aligned Bounding Rectangle*, que es el rectángulo de menor tamaño que encierra al polígono, y éste se emite (en sustitución), generando así suficientes fragmentos que cubran todo el área del mismo.

Finalmente, para cada uno de los fragmentos generados, se ejecuta una instancia del *Fragment Shader* cuya responsabilidad es realizar el test de intersección entre el polígono y los voxels que potencialmente lo intersecan. Para ello, se hace uso del rango de profundidad provisto por el **Geometry shader**, de forma que para cada fragmento se recorran todos los voxels que se encuentren en ese rango.

Aquellos voxels que pasen el test de intersección serán almacenados en una textura tridimensional (un grid o parrilla), a través de operaciones atómicas, ya que existe la posibilidad de que varias ejecuciones paralelas del *Fragment Shader* traten de escribir sobre las mismas posiciones, y es necesario promediar el valor de estos voxels.

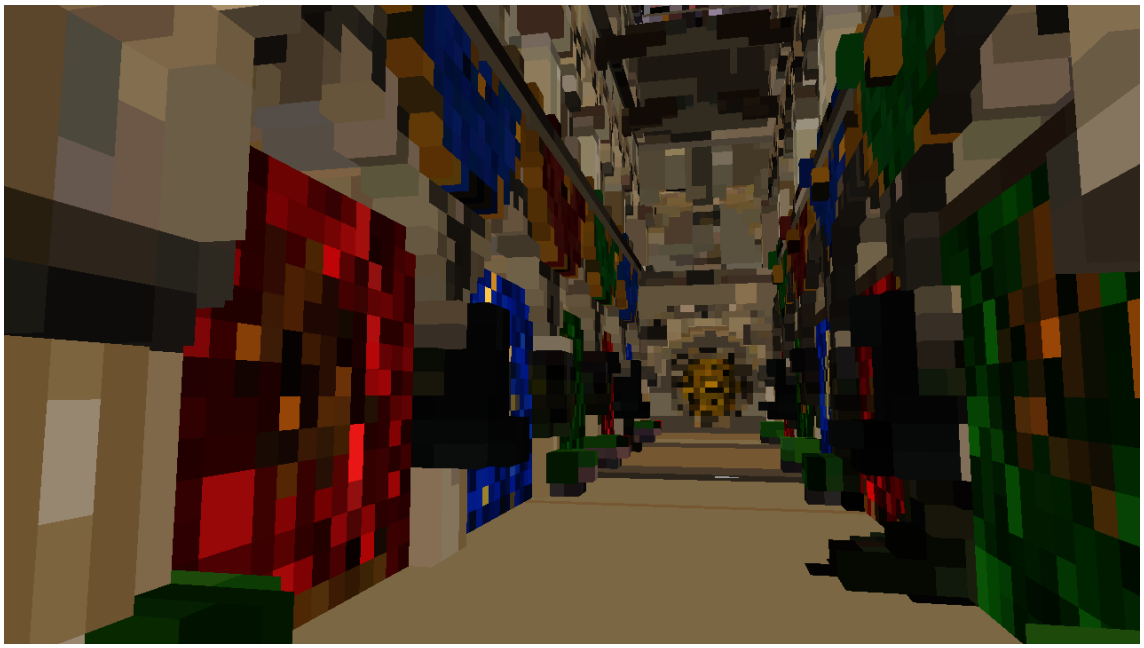


Figura 7.6: Voxelizado GPU, visualización de los colores por voxel

Esta aproximación, sin embargo, cuenta con una gran contrapartida: a diferencia que en la aproximación CPU, la resolución en voxels de una textura tridimensional está limitada por la implementación de OpenGL del fabricante, pero a cambio, obtenemos un voxelizado de resolución más baja en tiempo real, con color y normales a partir de *texture mapping*.

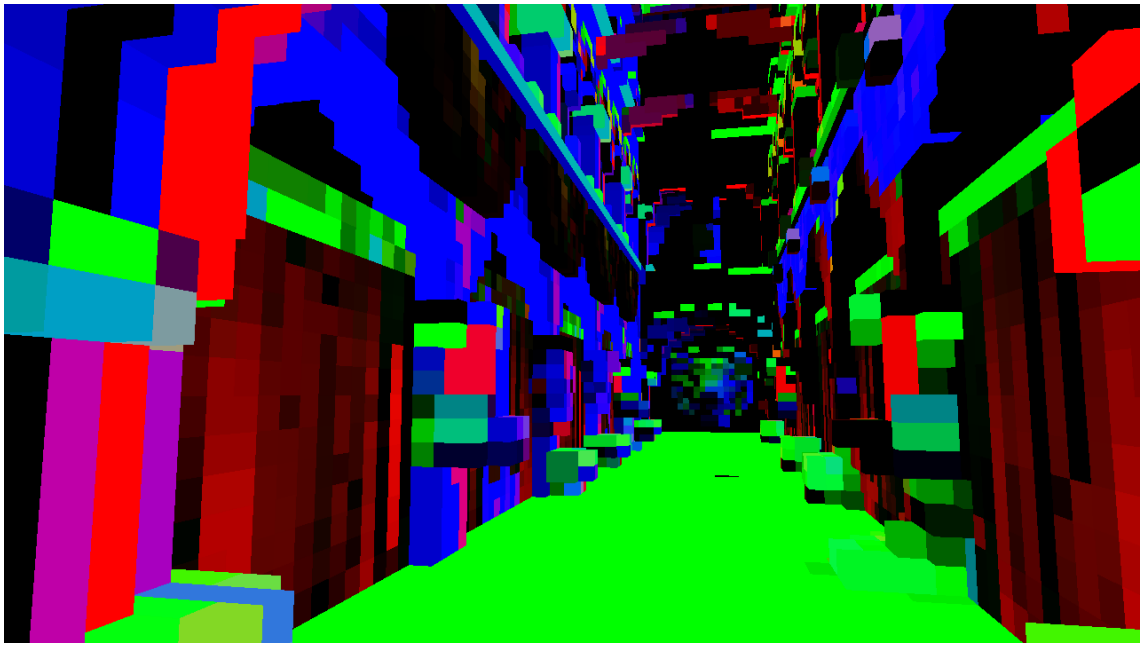


Figura 7.7: Voxelizado GPU, visualización de las normales por voxel

7.2. Rasterizado de voxels

Para visualizar voxels, una de las aproximaciones más sencillas consiste en, directamente, rasterizarlos en forma de cubos. Para llevar a cabo esta operación de forma relativamente eficiente, y ser capaces de visualizar un gran conjunto de los mismos, se ha optado por el uso de un *Geometry Shader* específico.

Este *Geometry Shader* es capaz de generar (a partir de un único vértice), un cubo completo para el que se necesita un total de 24. Este número puede reducirse, por ejemplo (y entre otras optimizaciones propias de la API gráfica), añadiendo a la información de los vértices un valor extra: 8bit (que actúen de *flags*) donde se indique qué caras del cubo es necesario emitir, determinado por un post-procesado del conjunto de los voxels.

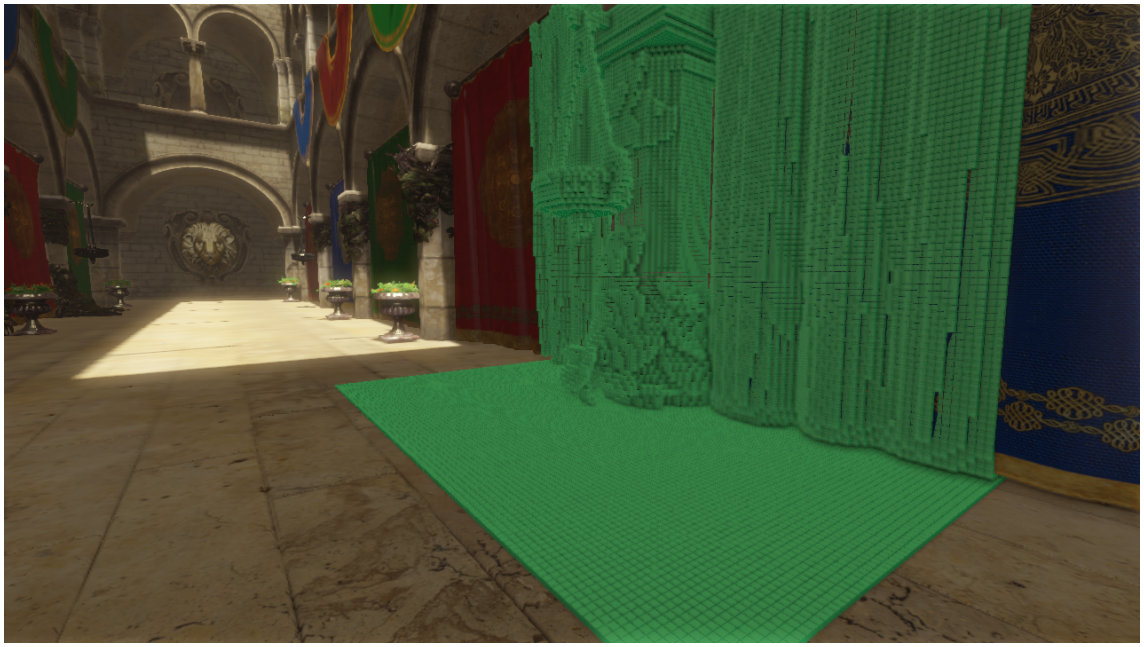


Figura 7.8: Voxelizado CPU y rasterización, se ha reducido ligeramente el tamaño de cada voxel para facilitar su visibilidad

7.3. Rendering volumétrico

Para los voxels generados por la voxelización GPU se ha implementado un pequeño *renderer* basado en *Raymarching* y *Raytracing*. De esta forma podemos visualizar directamente la textura tridimensional generada, sin necesidad de transferir su contenido a CPU, proceso que retrasaría demasiado todo el conjunto del *pipeline* y por ende la voxelización en tiempo real.

El funcionamiento de este *renderer* consta de dos pasos:

- Primero realizamos un proceso de *Raymarching* sobre una primitiva cúbica, de esta forma podemos obtener la posición de cada rayo en su "impacto" con el volumen que representa la textura tridimensional y también nos permite descartar aquellos rayos que no atraviesen este volumen.

- Una vez obtenemos el punto de entrada de los rayos en el volumen cúbico, ejecutamos un proceso de *Raytracing* en el que atravesamos, voxel a voxel, la textura,

bien hasta encontrar el primer voxel ocupado o continuando hasta la salida del rayo del mismo volumen, con la posibilidad de acumular la opacidad de todos los voxels atravesados.

Todas estas operaciones son realizadas por un único *Fragment Shader* directamente en GPU. Sin embargo, para iniciar su ejecución y generar un rayo por cada pixel, es necesario cubrir todo el espacio de pantalla con dos triángulos rectángulos (rasterizados con un *Vertex Shader*), de esta forma se iniciará una instancia del *Fragment Shader* para todos los fragmentos que conforman la ventana.

La técnica mencionada, que utilizamos también para iniciar *Fragment Shaders* como SSAO, Blur, FXAA o incluso Lighting, podría ser sustituida, en el futuro, por una petición de ejecución de los modernos *Compute Shader*, que en esencia, permiten suplir la necesidad del rasterizado de polígonos inicial, sustituyendo el uso de la ejecución de un *Vertex Shader* por una simple parametrización.

Las Figuras 7.6 y 7.7 de este mismo capítulo están renderizadas a través de este método.

Capítulo 8

CONCLUSIONES

A la luz de los capítulos previamente expuestos, cabe concluir que **Unnamed Engine**, la herramienta diseñada y desarrollada en este Trabajo Fin de Grado, constituye un buen ejemplo de aplicación de las últimas tecnologías, estándares y tendencias en renderizado 3D. El motor creado, si bien no es un producto destinado a un usuario final en su estado actual –sobre todo a nivel de GUI–, sí es una aplicación gráfica completa en un estado muy avanzado, capaz de generar, en tiempo real, fotogramas de una gran calidad visual y, gracias a la aplicación del voxel como un componente íntegro del sistema, preparado para convertirse en una potente herramienta gráfica moderna y multiplataforma.

Unnamed Engine es el resultado de un desarrollo ágil y continuado que ha permitido, a través de múltiples iteraciones, la experimentación y mejora de las diferentes capacidades individuales del proyecto planteado inicialmente, pormenorizándose en el estudio de cada una de ellas:

- **Iteración 1:** Construcción de los cimiento del sistema de *rendering* y los controles básicos. Experimentación con técnicas de *raymarching* sobre esferas y primitivas, inicialmente en CPU y pasando posteriormente a la utilización de GPU.

- **Iteración 2:** Creación de los primeros *shaders* de *raymarching* y *raytracing* sobre texturas tridimensionales generadas proceduralmente con el uso de voxels.

- **Iteración 3:** Refactorización y generalización de los componentes hasta ahora desarrollados. Modificación del *pipeline* para *rendering* diferido. Implementación de *raymarching* sobre campos distancia (SDF, Signed Distance Field), técnica que fue finalmente descartada.

- **Iteración 4:** Inclusión de rendering poligonal básico y creación de un *Geometry Shader* para el rasterizado de los voxels, aplicados a la generación procedural de terreno. Añadidos primeros algoritmos de iluminación y oclusión ambiental.

- **Iteración 5:** Particionado de la generación de terreno en bloques (*chunks*) de voxels, creación de mecanismos de generación asíncrona y paralela. Creación de la interfaz gráfica de desarrollo y configuración.

- **Iteración 6:** Desarrollo de mecanismos de carga de modelos poligonales, sistema de materiales y efectos de post-procesado para la mejora visual de la imagen. Refactorización y generalización de los componentes del sistema, mejora de los sistemas de orquestación del *pipeline*.

- **Iteración 7:** Inclusión de efectos avanzados, mecanismos de gestión de la escena y creación de herramientas para la interfaz de desarrollo. Mejora en la gestión de los recursos e implementación del voxelizado CPU y GPU, integrado a los sistemas de visualización de voxels desarrollados en iteraciones anteriores.

En atención a los objetivos expuestos al inicio de este trabajo, que en lineas generales damos por cumplidos, destacar que en el curso del diseño y desarrollo del proyecto se obtuvo también un hito inicialmente no planificado: La realización de cada una de las funcionalidades que iba requiriendo la consecución de los objetivos, en las diferentes iteraciones, proporcionó al autor una visión global de los gráficos por computador mayor de la inicialmente proyectada. Si bien la implementación presentada no incluye la totalidad de los conocimientos obtenidos en muchos aspectos durante este trabajo, se ha tratado de desarrollar un sistema equilibrado, al nivel de un alumno de último curso de grado, que comprendiese una amplia cartera de funcionalidades así como una interfaz de usuario que permita la interacción con todo su conjunto.

Nuestro motor se encuentra preparado para escalar y crecer, de la misma forma que lo ha hecho a lo largo de este último año, ya sea para aumentar su potencia visual (a través de las posibilidades que nos ofrece la representación voxelizada), o dotarlo de mayores funcionalidades, propias de los sistemas de este tipo (animación, sonido, físicas, IA, sistema de partículas...).

En concreto, alguna de las potenciales líneas de desarrollo que contribuirían a incrementar sus capacidades, rendimiento y/o calidad de imagen, serían, a nuestro juicio, las siguientes:

- ***Physically Based Rendering* (PBR):** El motor presentado utiliza el modelo de sombreado *Blinn-Phong*, sobre el que profundizamos a lo largo del trabajo, sin embargo y aunque consigue un resultado considerado bastante realista, los modelos modernos de *shading* y última generación hacen uso de una simulación más compleja de la iluminación, basadose en físicas y propiedades tales como la cantidad de partículas metálicas presentes en un material. Los resultados de estos modelos no sólo son superiores en realismo, sino que aseguran la correcta iluminación de los materiales bajo todo tipo de condiciones lumínicas, algo que no ocurre con el *Phong shading*.

- **Iluminación global y oclusión ambiental:** El estado del arte para el renderizado en tiempo real u *online* de escenas realistas, durante la realización de este proyecto, parece radicar en el uso combinado de rasterizado de polígonos y procesos de *raytracing* ligeros, gracias al uso de representaciones "gruesas" de las escenas, como las obtenidas en forma de voxels, y a las que aspira la implementación presentada.

- **Reflejos y refracciones:** Actualmente el motor carece de estas capacidades, que podrían ser incluidas a través de efectos conocidos aplicables al rasterizado, pero también como resultado de un proceso ligero de *raytracing* sobre la estructura voxelizada, similar al mencionado en el punto anterior.

- **Efectos volumétricos:** A través del uso de los voxels y las técnicas de *raymarching* podría incluirse la generación de efectos volumétricos tales como humo, nubes, la propia iluminación o incluso agua.

- **Voxel *meshing*:** Disponer de la capacidad de transformar, a la inversa, conjuntos de voxels en figuras poligonales, por ejemplo a través de la aplicación del algoritmos como *Marching Cubes* o *Dual Countouring* abriría las puertas a un gran conjunto de aplicaciones tales como la simplificación de mallas o el *sculpting* de terreno, por poner algunos ejemplos.

- **Carga y procesado de nubes de puntos y otras representaciones gráficas:** Actualmente la carga de modelos se restringe a formatos poligonales, siendo los voxels generados a partir de estos o proceduralmente. La adición de otros formatos y conversiones entre estas diferentes representaciones es una también un línea importante de trabajo que queda abierta para el futuro.

- **Mejoras generales:** En el software presentado queda aún bastante espacio para la mejora: optimización, portabilidad, trazabilidad... pero sobre todo, podría resultar de especial interés, la uniformización del acceso a los recursos y control del motor de *rendering*. Facilitando el uso de nuestro sistema como un *framework* gráfico para el desarrollo de programas y aplicaciones con un importante componente visual.

Apéndice A

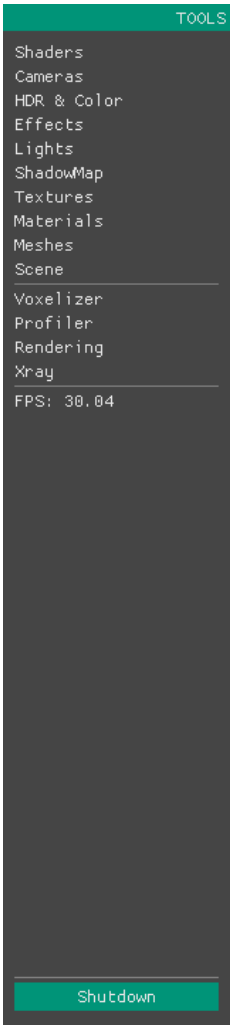
Manual de Usuario

UNNAMED ENGINE

El software **Unnamed Engine** (***Alpha 1.0***), implementa un conjunto de los subsistemas que habitualmente podemos encontrar como parte de la arquitectura de un motor de juego (game engine) moderno: gestores de recursos, renderizado diferido, culling, iluminación, sombreado, efectos de post-procesado y herramientas de debugging. En otro orden de cosas, la aplicación se nutre del concepto de voxel y actualmente cuenta con mecanismos para el tratamiento y visualización de los mismos, así como capacidad de voxelización de escenas, rendering volumétrico y generación procedural.

En este pequeño manual de usuario se enumerará y describrá cada una de las herramientas de desarrollo que han sido creadas en paralelo al núcleo del propio motor, y que permiten interactuar con las diferentes partes y funcionalidades que lo componen. Para un mayor entendimiento de las opciones y parámetros aquí descritos se recomienda utilizar como referencia la memoria propia del proyecto.

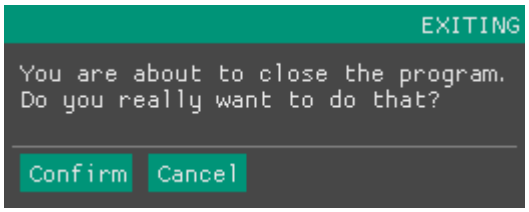
MENU



El menú lateral izquierdo del motor proporciona acceso a las diferentes herramientas desarrolladas. Para ***ocultar*** o ***mostrar*** el menú y las diferentes herramientas abiertas, se ha configurado la tecla CTRL derecho.

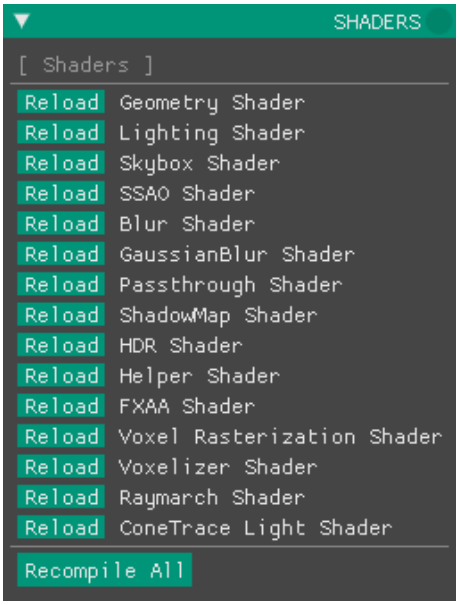
Al ***ocultar*** el menú, el cursor del ratón desaparecerá, indicando que el control del ratón se encuentra enlazado al movimiento de la cámara. Si realizamos la operación inversa, y ***mostramos*** el menú otra vez, el cursor volverá a aparecer de forma que podamos interactuar con la interfaz. Si deseamos alternar el control de la cámara y la interfaz sin ocultar el menú o herramientas abiertas, podremos utilizar la tecla de escape ESC.

En el menú lateral también podemos encontrar un botón de cierre o apagado: Shutdown, es recomendable utilizarlo para salir correctamente de la aplicación. El cierre de la propia ventana, directamente desde el Sistema Operativo en el que nos encontremos, debería ejecutar la misma secuencia de apagado, por lo que también es una opción. Lo que no se recomienda es "matar" el proceso directamente pues no se asegura la liberación de todos los recursos reservados por el sistema.



El último ítem del menú indica, en tiempo real, los fotogramas por segundo ***FPS*** del pipeline gráfico. Si se desea una información más detallada acerca de los tiempos de cada fase del sistema, puede accederse a la herramienta Profiler, que será explicada junto con el resto de herramientas disponibles del panel a lo largo de este pequeño manual de uso: Shaders, Cameras, HDR & Color, Effects, Lights, ShadowMap, Textures, Materials, Meshes, Scene, Voxelizer, Rendering, Xray.

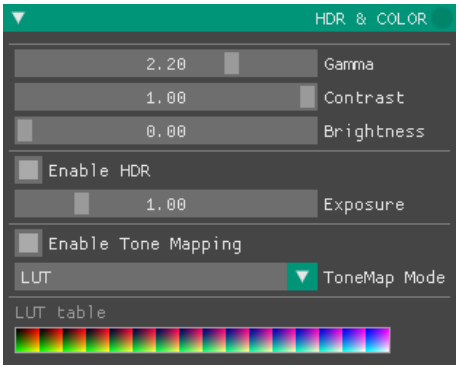
SHADERS



Esta herramienta (**Shaders**) actualmente lista los diferentes programas de shading definidos en el pipeline. A cada programa de shading, identificado por su nombre, le acompaña un botón de recarga Reload que permite volver a leer y compilar el contenido de los ficheros asociados a cada programa. De esta manera es posible aplicar las modificaciones a los programas sin necesidad de cerrar por completo el sistema.

El botón grande Recompile All sirve para realizar la carga completa de todos los shaders definidos, como si del inicio del motor se tratase. Tiene el mismo efecto que realizar Reload para cada uno de los elementos de la lista.

HDR & COLORS



La herramienta **HDR & Color** expone los controles destinados a la **correción de color**, **color mapping** y **HDR** (High Dynamic Range).

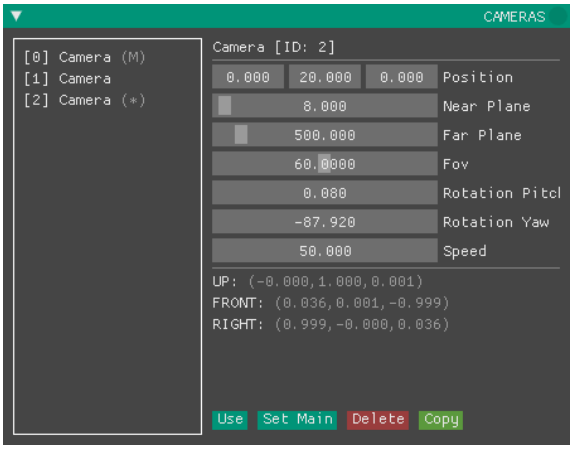
El primero de los sliders modifica el valor de la corrección **Gamma**, éste componente se encuentra establecido por defecto en 2.2, (valor estándar para la correción de la escala de luminosidad en los monitores actuales). Aumentar este parámetro resultará en imágenes más brillantes, con lo que se obtendrá una mayor definición en las zonas oscuras.

Los dos siguientes parámetros modificables son el contraste (**Contrast**) y el brillo (**Brightness**). El ajuste de estos dos parámetros influirá en el color resultante del render, previa correción gamma, de la forma: [***result = color * contras + brightness***]. Por defecto, los valores para estos parámetros son respectivamente 1.0 y 0.0, por lo que, si no son modificados y atendiendo a la forma mencionada, no tendrán efecto.

Cuando el efecto **HDR** esté activo aparecerá un nuevo slider que permite controlar la exposición (**Exposure**) de la cámara, éste parámetro deberá ser ajustado en función de la luminosidad media de la imágen.

Finalmente, el checkbox ***Enable Tone Mapping*** habilita el mapeo de colores (**Tone Mapping**). Este efecto permite teñir la totalidad de la imagen en función de una escala de colores variable. Cuando el efecto se encuentre activo aparecerá un pequeño menú desplegable que permite escoger entre tres tipos de mapeo: ***LUT*** basado en el muestreo de una LUT (Look Up Table) que podrá visualizarse al pie de la herramienta, y por otro lado ***Reinhard*** y ***Uncharted2***, funciones de mapeo ampliamente conocidas.

CAMERAS



La herramienta para cámaras (**Cameras**) permite administrar el conjunto de estas entidades disponibles en el motor. A través de esta herramienta pueden crearse y eliminarse cámaras, así como modificar los parámetros individualmente de cada una.

En la parte lateral izquierda de la herramienta se listan las cámaras disponibles en el motor. Se distinguen, sin embargo, dos cámaras especiales: la cámara principal ***(M)*** y la cámara en uso ***(*)***. Para establecer estas propiedades sobre una cámara previamente seleccionada pueden usarse los botones: Set Main y Use respectivamente.

⇨ La **cámara principal** es aquella considerada primaria y por tanto es la utilizada para determinar la visibilidad de la escena, aquellos modelos de la escena que no se encuentren en su campo de visión no serán dibujados.

⇨ La **cámara en uso** es la cámara utilizada como objetivo, aquella a través de la cual visualizamos la escena.

La creación de nuevas cámaras se realiza a partir del botón Copy, esta acción establece los parámetros iniciales de la nueva cámara copiando aquellos de la cámara ***(*)*** actualmente en uso. También es posible eliminar, con el botón Delete, la cámara seleccionada actualmente en el menú lateral izquierdo.

Parametros de cámara

⇨ **Position**: Componentes X, Y, Z de posición de la cámara en ***world space***.

⇨ **Near Plane**: Distancia de visión mínima de la cámara.

⇨ **Far Plane**: Distancia de visión máxima de la cámara.

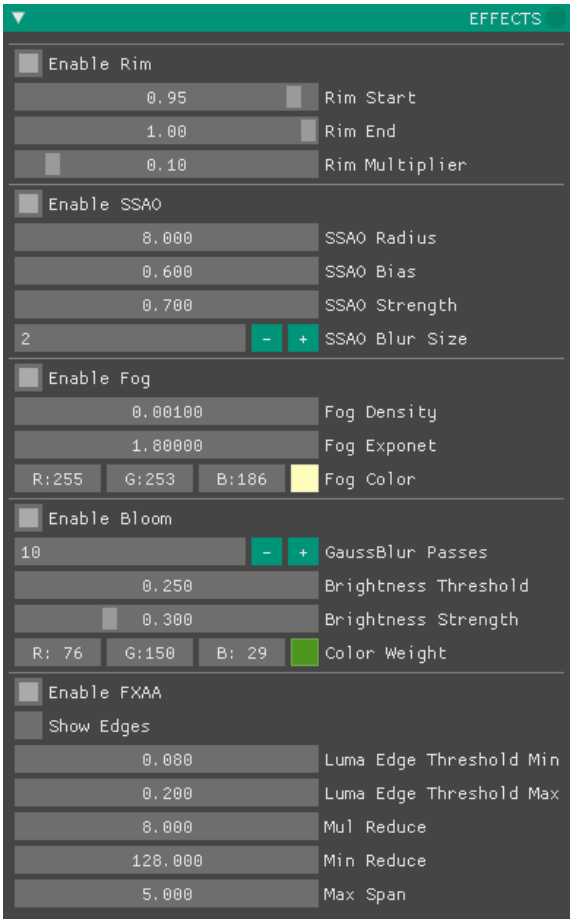
⇨ **Fov**: Ángulo de apertura del cono de visión de la cámara (Field of View).

⇨ **Rotation Pitch** y **Rotation Yaw**: Parámetros de rotación que indican la orientación del objetivo de la cámara.

⇨ **Speed**: Velocidad, incremento de posición por fotograma.

⇨ **Up**, **Front** y **Right**: Vectores generadores del espacio vectorial de la cámara.

EFFECTS



La herramienta de efectos (**Effects**) contiene los controles de los principales efectos gráficos aplicados en el pipeline.

⇨ **Rim**: El bordeado (Rim) estiliza las figuras creando un pequeño brillo en aquellas zonas que se encuentren en un cierto rango de inclinación. El multiplicador (**Multiplier**) permite modificar cúanto son efectadas estas zonas. Los parámetros **Rim Start** y **Rim end** determinan la inclinación mínima y máxima que debe cumplir una superficie para que sea aplicado el efecto.

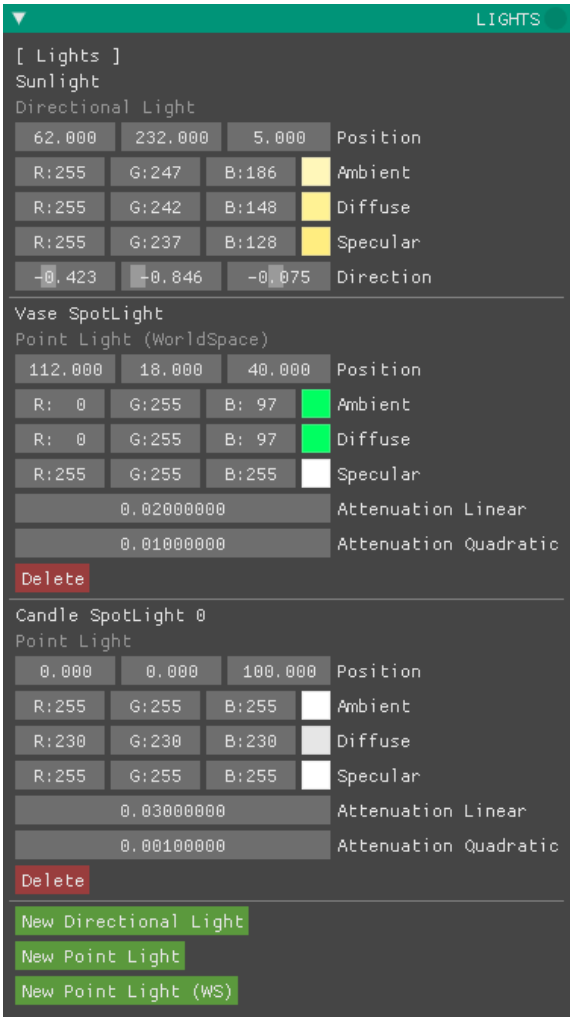
⇨ **SSAO**: SSAO (Screen Space Ambient Occlusion) es la técnica que proporciona oclusión ambiental, oscurecimiento de aquellas zonas estrechas y recovecos entre los modelos. Los parámetros modificables son: Radio (**Radius**) del hemisferio para el sampling, el **Bias** (offset) de profundidad, y el **Strength**, un exponente que permite controlar la fuerza de estas sombras. **Blur Size** determina el tamaño del núcleo de desenfoque aplicado al resultado.

⇨ **Fog**: Al activar este efecto se añadirá una neblina en base a la distancia de los modelos con respecto a la cámara. **Fog Color** permite teñir el color de la niebla mientras que la densidad (**Fog Density**) y el exponente (**Fog Exponent**) controlan el aspecto.

⇨ **Bloom**: Este efecto añade un halo de brillantez a aquellas zonas con mayor luminosidad en la imagen, los parámetros ajustables son: La cantidad de pases de desenfoque (**GaussBlur Passes**), un mayor número de pases resulta en una mayor difuminación del halo; el **Brightness Threshold**, un límite ajustable que determina cúanto de brillante debe ser una zona para generar halo; la fuerza (**Brightness Strength**) con la que el bloom es aplicado sobre el render; y finalmente el **Color Weight**, color base que se utiliza para determinar la luminancia de los fragmentos de la imagen.

⇨ **FXAA**: FXAA (Fast Approximate Anti-Aliasing) es la técnica implementada para realizar anti-aliasing en espacio de pantalla. Si se activa el checkbox **Show Edges** podremos visualizar los ejes detectados en la imagen que están siendo suavizados. La detección de ejes y el propio suavizado pueden controlarse con la lista de propiedades a continuación, aunque se recomienda mantener los ajustes por defecto.

LIGHTS



La herramienta de **Lights** permite gestionar la colección de luces aplicadas en la escena. A excepción de la luz principal, ***Sunlight***, que se encuentra asociada a un **Shadowmap**, el resto de luces pueden ser eliminadas mediante el botón Delete.

Se distinguen 4 tipos de luces: ***Directional***, ***Point***, ***Point WS*** y **[WIP]** ***Spot***. Para crear una luz de cada uno de estos tipos pueden utilizarse los botones New ....

Todas las luces comparten una serie de atributos modificables tales como la posición (**Position**) y los componentes de color (**Ambient**, **Diffuse** y **Specular**).

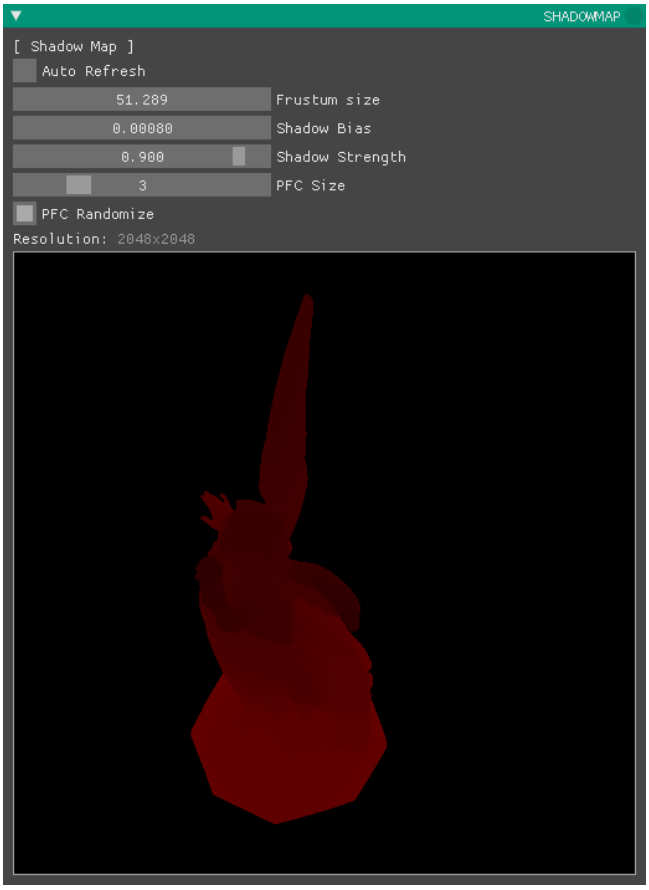
Las luces direccionales (***Directional Light***) poseen a su vez de un vector de tres componentes que indica la dirección en la que apuntan (**Direction**).

Por otra parte, se distinguen dos tipos de luces ***Point***. Si no se indica lo contrario, la posición de estas luces es dependiente de la posición de la cámara en cada momento. Si están etiquetadas como WS (***WorldSpace***), entonces la posición es relativa a la propia escena, independientemente de dónde se encuentre la cámara.

Las luces de tipo ***Point*** poseen dos atributos propios que definen la atenuación (**Attenuation**). Los dos componentes **Attenuation Linear** y **Attenuation Quadratic** definen cuánto se atenúa la luz con respecto a la distancia a su origen, atenuación lineal y cuadrática respectivamente.

Las ***Spot Lights*** no se encuentran actualmente implementadas, y por tanto no es posible aún crear luces de este tipo.

SHADOW MAP



La herramienta **ShadowMap** tiene como propósito permitir el control de la generación del mapa de sombras asociado a la luz principal (Luz direccional). El mapa de sombras determina qué partes de los modelos son visibles desde el punto de vista de esta luz.

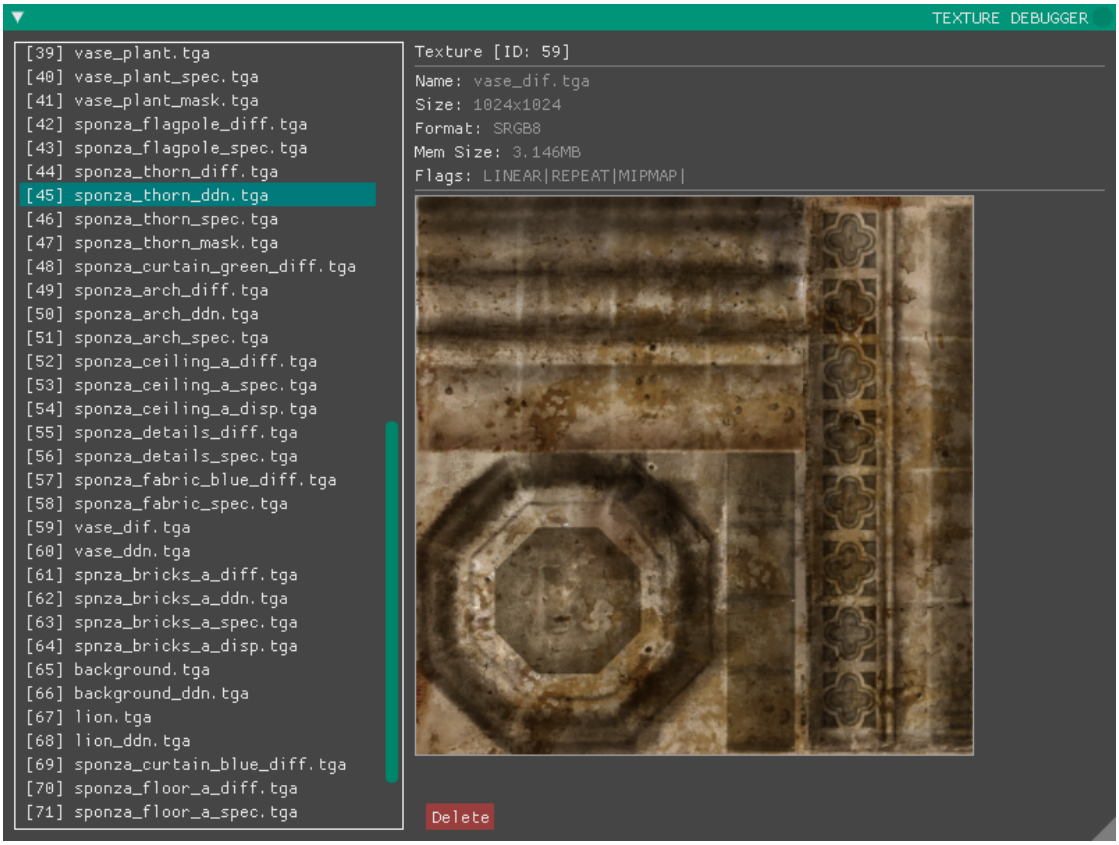
Actualmente las sombras generadas son únicamente estáticas: el mapa de sombras es una imagen de un tamaño superior a la resolución utilizada en la propia ventana de la aplicación y aunque sólo contenga información de profundiad, requiere el redibujado de todos los modelos de la escena (al menos aquellos que generen sombras). Es por tanto deseable ejecutarlo una única vez (al inicio del programa o cuando se cargue una nueva escena).

Si así se desea, puede marcarse la casilla ***Auto Refresh***, que incluirá la generación del mapa para cada fotograma, convirtiéndose por tanto en dinámico. Téngase en cuenta que, cerrada la ventana de la herramienta, el ***Auto Refresh*** perderá su efecto.

La herramienta expone 3 parámetros ajustables: el **Frustum Size** controla el tamaño del área de visión de la cámara utilizada para generar el mapa; El **Shadow Bias** es un offset utilizado a la hora de determinar si un fragmento se encuentra o no a la sombra (que permite evitar artefactos como el ***shadow acne***); El **Shadow Strength** representa el porcentaje de reducción de luminosidad para aquellas zonas que se encuentran en sombra, un valor de 0.9 significa que las zonas a la sombra reciben una reducción de un 90% de su color original.

Finalmente se encuentran dos atributos más, esta vez relativos al **PCF** (***Percentage Close Filtering***), que es una función de muestreo utilizada para suavizar los bordes de las sombras. El **PCF Size** controla el tamaño del núcleo de muestreo utilizado. La propiedad **PCF Randomize** introduce cierto grado de aleatoriedad a esta función, de forma que se difuminen aún más los bordes de las sombras.

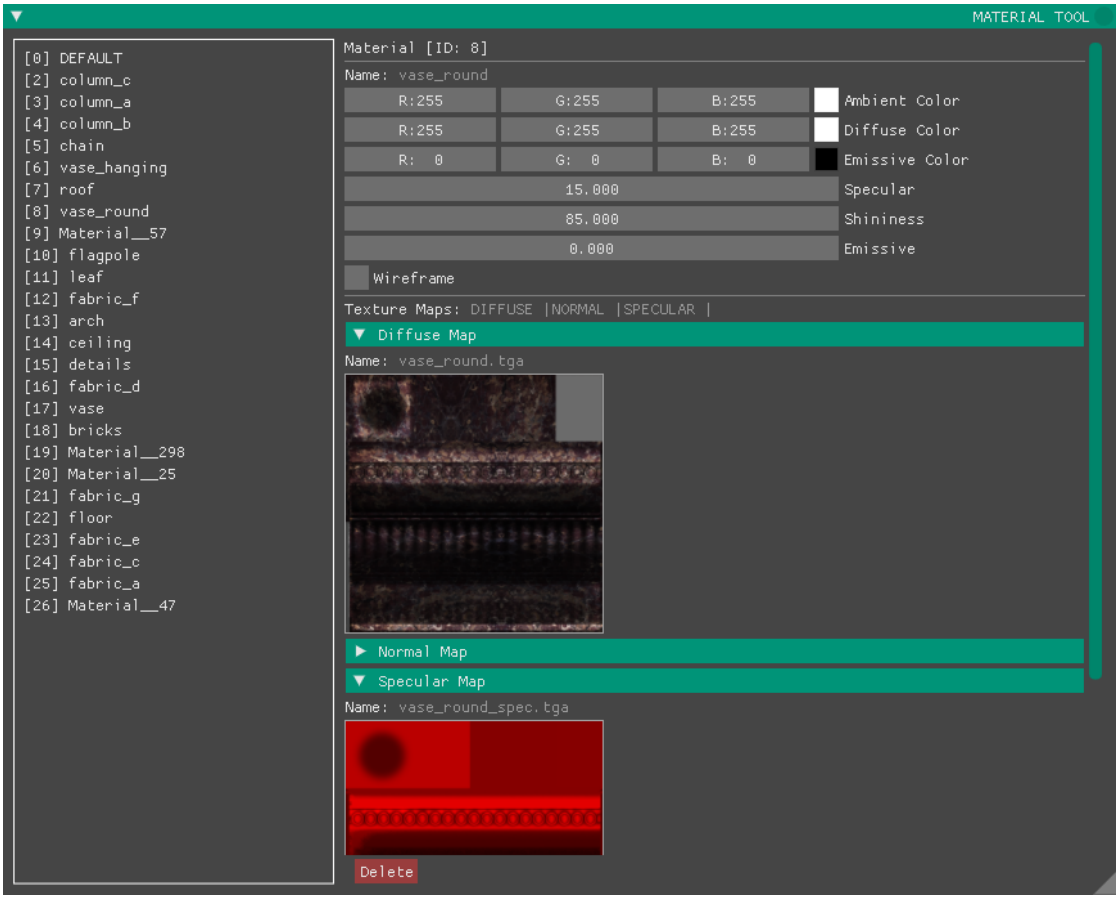
TEXTURES



La herramienta de texturas (**Textures**) lista todas las imágenes cargadas actualmente en la memoria de la GPU, esto incluye todos los mapas de materiales cargados a raíz de un modelo y los propios ***Framebuffers*** utilizados por las diferentes etapas del pipeline.

A través de esta interfaz podemos visualizar cada una de las imágenes cargadas así como la información asociada a las mismas: nombre, tamaño, formato y tamaño en memoria... La herramienta no posee actualmente ninguna otra utilidad más que el debugging del software en construcción.

MATERIALS



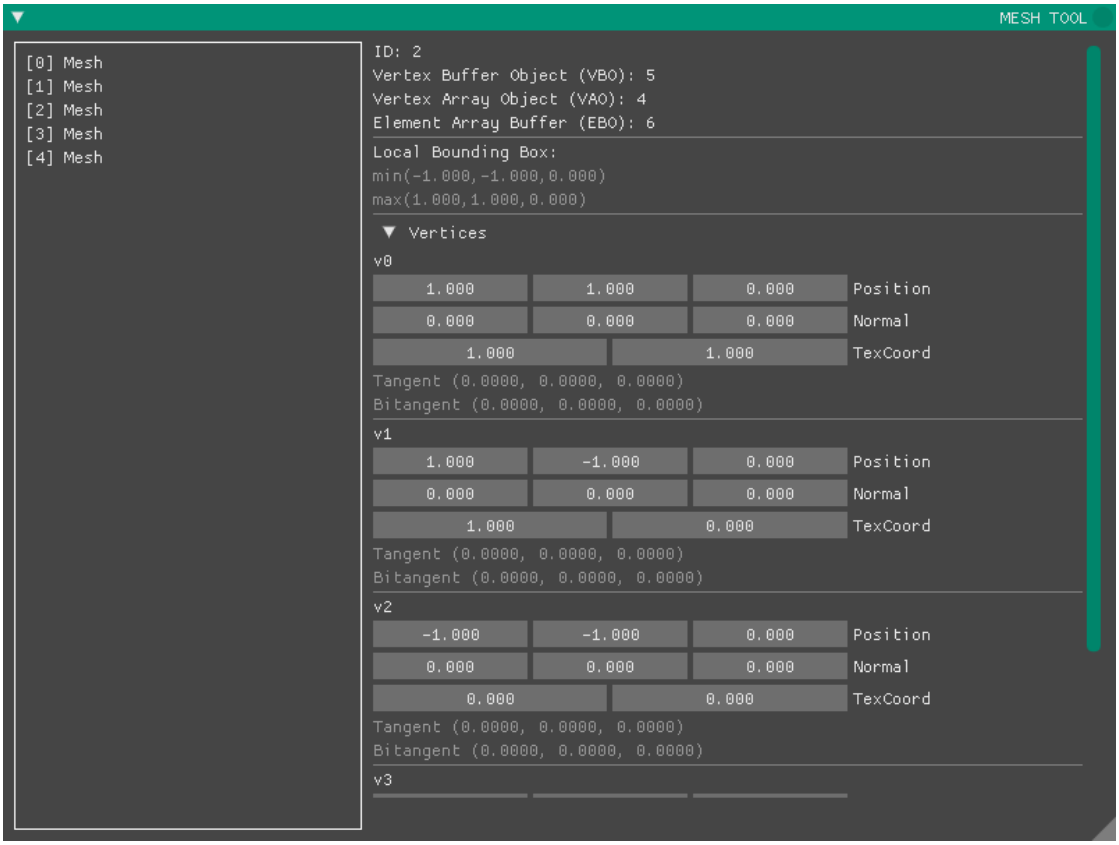
La interfaz de materiales (**Materials**) lista todos los materiales que se encuentran actualmente en el sistema. Estos materiales, y sus propiedades, son generados automáticamente a partir de la información provista por la carga de un nuevo modelo o escena.

Para cada material podemos consultar y modificar sus principales parámetros tales como los componentes de color: **Ambient**, **Diffuse** y **Emissive**; y los componentes escalares: **Specular**, **Shininess** y **Emissive**, siendo este último la fuerza con la que se emite su color de nombre similar.

En la parte inferior, se encuentran una serie de menús desplegables, uno por cada mapa (imagen) que utilice el material. En estos mapas, a fin de cuentas, podemos visualizar las mismas imágenes disponibles a través de la herramienta de texturas (**Textures**) mencionada anteriormente.

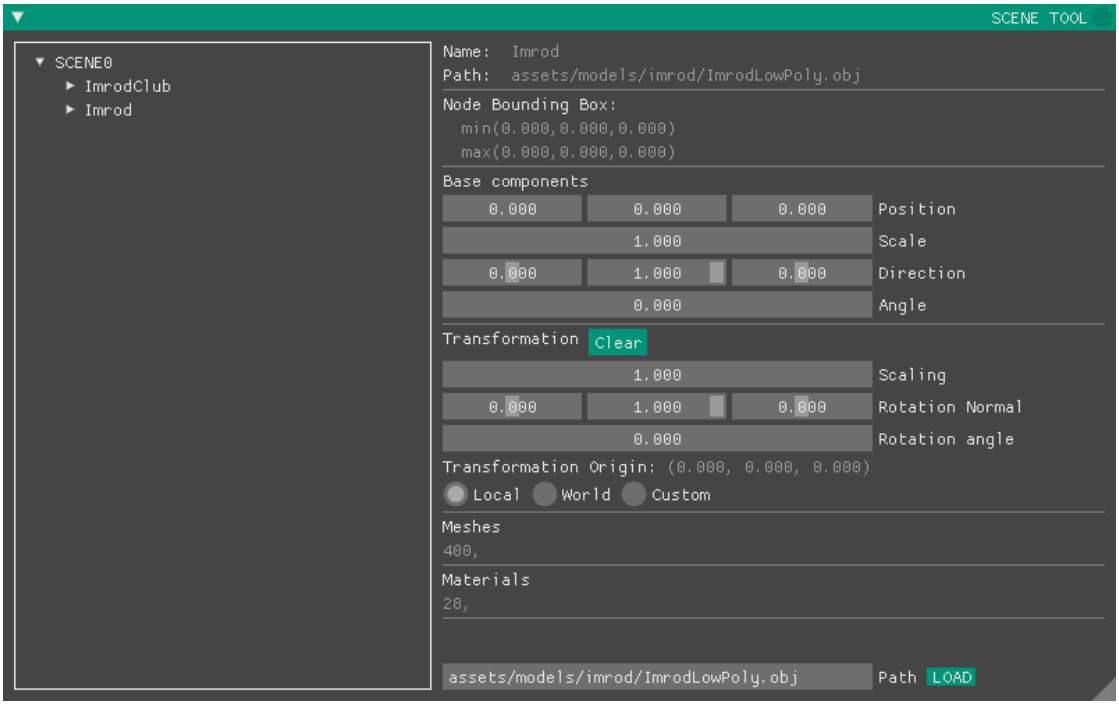
La opción **Wireframe** fuerza a que todos los modelos que utilicen el material sean dibujados en este modo.

MESHES



La herramienta de mallas (**Meshes**) permite acceder a la información geométrica de los modelos cargados en la GPU, si bien cuenta con la capacidad de editar algunas de las propiedades de los propios vértices que conforman las mallas, su utilidad no es otra más que el debugging del software en construcción.

SCENE

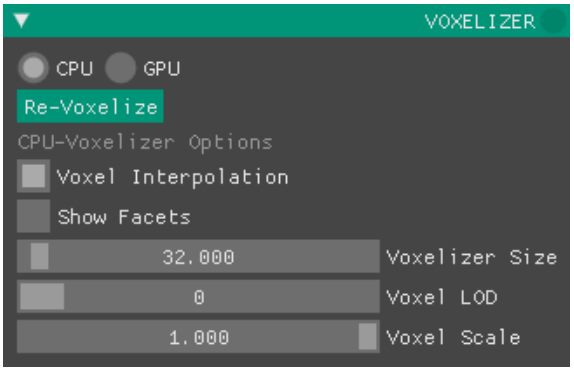


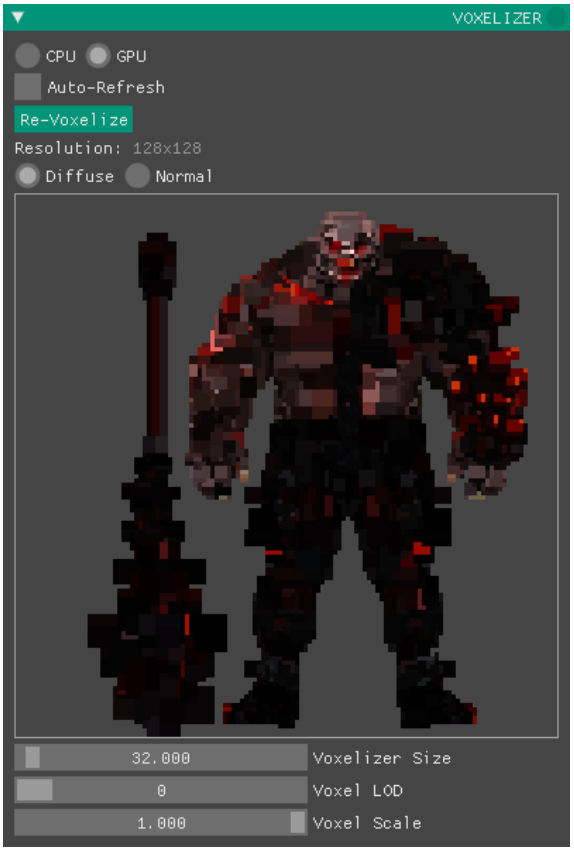
A través de la herramienta de escena (**Scene**) podemos acceder a las propiedades asociadas a los modelos que forman parte de la escena, que a grandes rasgos, son exclusivamente su posición, rotación y escala.

Sin embargo, cabe destacar que se distinguen dos bloques similares: por un lado tenemos los componentes base (**Base Componets**) y por otro las transformaciones (**Transformation**). Mientras que el primero determina la posición, escala y orientación del modelo con respecto a su propio origen (de una forma no reversible), el segundo permite realizar transformaciones arbitrarias, utilizando un origen local (**Local**), un origen absoluto (**World**) o un origen arbitrario (**Custom**), transformaciones que sí pueden ser revertidas mediante el uso del botón Clear.

Finalmente, a pie de la herramienta, se encuentra un cuadro de texto que permite introducir la ruta de una nueva escena a cargar (ruta relativa a la carpeta desde la que se ejecuta el software). Una vez introducida la ruta y tras pulsar el botón LOAD los mecanismos de carga asíncrona iniciarán el procesado de la nueva escena, dando de alta nuevas mallas y materiales.

VOXELIZER





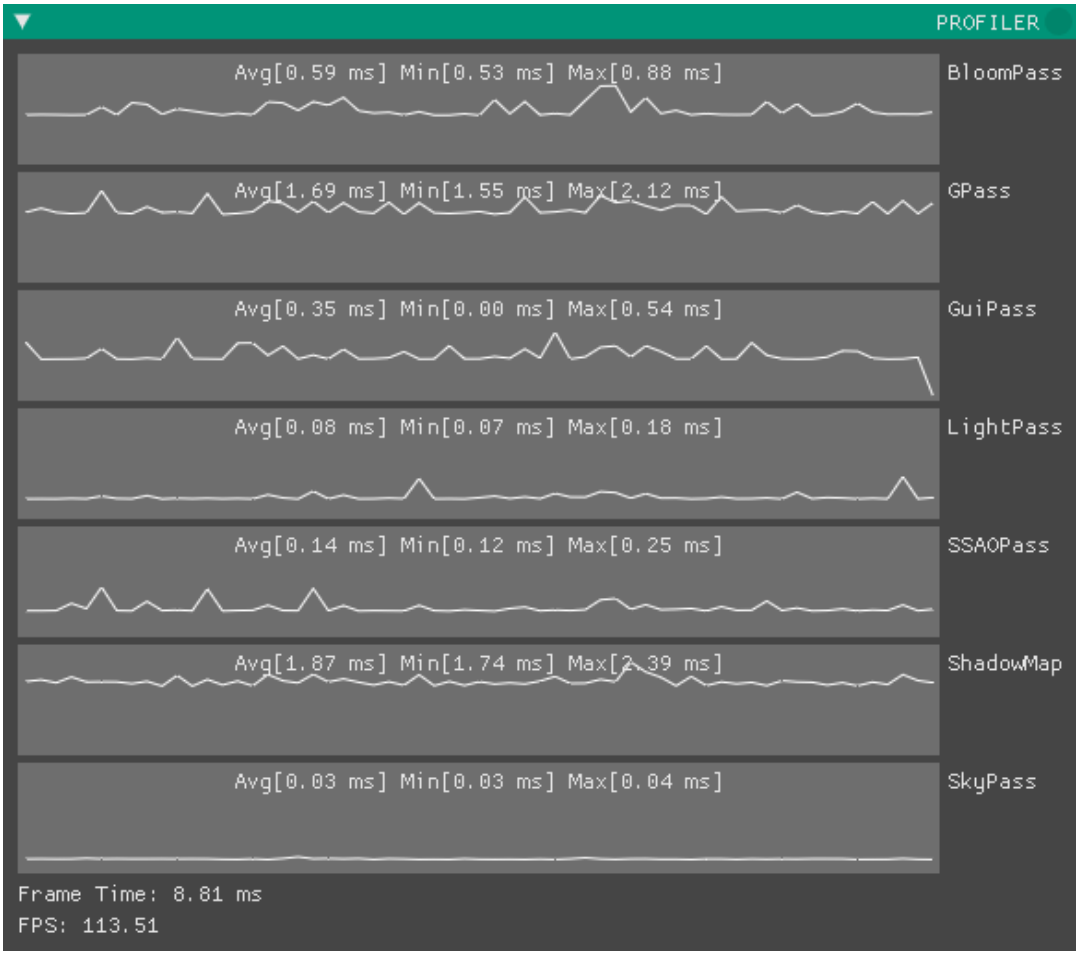
El voxelizador (**Voxelizer**) es la herramienta que permite ejemplificar el funcionamiento del mecanismo de voxelizado de la escena. Actualmente se encuentra definido un volumen cúbico que denominamos el "volumen de voxelizado". Este volumen, por defecto, se encuentra situado estratégicamente frente a la cámara principal y su visualización puede ser activada a través de la herramienta de **Rendering**, que detallaremos más adelante. La resolución en voxels utilizada para cubrir el volumen proviene de los ficheros de configuración del motor y actualmente no puede ser modificada.

El voxelizador permite el control de los dos modelos de voxelizado: **CPU** y **GPU**. Independientemente del modo de voxelización activo, podemos controlar el tamaño de este volumen de voxelizado a través del slider **Voxelizer Size**. Para voxelizar toda la geometría que se encuentre dentro del volumen definido, basta con utilizar el botón Re-Voxelize. Se recomienda utilizar una cámara secundaria (a través de la herramienta **Cameras**) y activar la opción **Show Voxelizer Volume** (herramienta **Rendering**), de forma que podamos movernos por la escena mientras que la cámara principal (y por tanto, el volumen de voxelizado) se encuentren estáticos.

Cuando el voxelizador **CPU** se encuentre activo, aparecerán dos opciones: la primera **Voxel LOD** permite seleccionar el nivel de de detalle (***Level Of Detail***), siendo 0 el máximo nivel de detalle posible. La segunda opción, **Voxel Scale** permite escalar el tamaño de los propios voxels durante su rasterizado. Para visualizar el resultado de la voxelización **CPU** debe activarse la casilla **Show Voxels** (de la herramienta **Rendering**). El checkbox **Show Facets** del voxelizador **CPU** permite alternar entre el uso de las normales extraídas para cada voxel, o el uso de los vectores normales para cada cara del voxel cúbico.

Cuando el voxelizador **GPU** se encuentre activo, podrá seleccionarse la opción **Auto-Refresh**, que ejecutará el voxelizado durante cada fotograma, aprovechando la velocidad y paralelismo que provee la propia GPU. Este modo permite también visualizar el resultado intermedio de la voxelización, pudiendo alternar entre la vista de color (**Diffuse**) y la de normales (**Normal**).

PROFILER



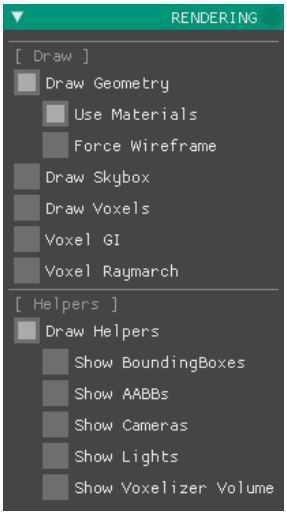
La herramienta **Profiler** está destinada a proveer información en tiempo real relativa a los tiempos de ejecución de las diferentes etapas del pipeline.

Esta herramienta mostrará una gráfica asociada a cada fase. Por defecto el **Profiler** no se incluye al construir el código fuente de la aplicación, para activarlo es necesario recompilar el software con la opción ***-DDISABLE_PROFILING***, que puede ser añadida a través del fichero ***makefile*** de construcción.

En esta herramienta también podemos visualizar, en comparación con los **FPS** (fotogramas por segundo) del motor, el tiempo medio destinado a cada frame (**Frame Time**).

Actualmente los tiempos son únicamente mediciones de las ejecuciones en GPU, es por ello que la suma de los valores mostradas en las diferentes gráficas, que no contemplan el correspondiente procesado CPU, no se corresponda con los **FPS** o el **Frame Time**.

RENDERING



La herramienta de renderizado (**Rendering**) consiste exclusivamente en un conjunto de opciones que, según se encuentren o no activas, modifican aspectos relacionados con la orquestación de las etapas de dibujado del pipeline que se enumeran a continuación:

⇨ **Draw Geometry**: Activa el dibujado de las mallas poligonales. Permite desactivar el uso de materiales (**Use Materials**), lo que fuerza un material por defecto para todos los modelos, y permite también forzar el dibujado en ***wireframe*** de las mallas (**Force Wireframe**), si así no se establece en los materiales propios de cada modelo.

⇨ **Draw Skybox**: Habilita el dibujado del fondo de las imágenes (a través del renderizado de un ***cubemap***).

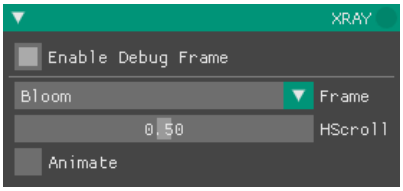
⇨ **Draw Voxels**: Cuando esta opción se encuentre activa, podrán ser visualizados los voxels (voxelizado CPU) a través de su rasterizado en forma de cubos.

⇨ **Voxel GI**: **[WIP]** Activa la iluminación global basada en voxel cone tracing.

⇨ **Voxel Raymarch**: Habilita el rendering volumétrico que permite visualizar la textura tridimensional resultado del voxelizado GPU.

⇨ **Draw Helpers**: Cuando se encuentre activo, aparecerán, de forma individualizada, las opciones para activar el dibujado de los direfentes ayudantes (**Helpers**) actualmente implementados, entre ellos, el **Voxelizer Volume** permite visualizar el volumen cúbico objetivo del voxelizado (realizado a través de la herrameinta **Voxelizer**)

XRAY



La herramienta **Xray** permite visualizar los resultados individuales de las diferentes etapas y efectos superpuesto al resultado final del pipeline. Cuando se active (a través del checkbox **Enable Debug Frame**) aparecerá un menú desplegable (**Frame**) donde podremos seleccionar la vista que nos interese, entre las disponibles: ***Position***, ***Depth***, ***Normal***, ***Albedo***, ***Specular***, ***Shininess***, ***Brightness***, ***SSAO***, ***SSAO-Blur***, ***LightSpacePosition*** y ***Bloom***.

Mediante el deslizable **HScroll** podemos indicar el porcentaje de pantalla que será utilizado para la vista del render final y la vista de debug. La función del checkbox **Animate** es modificar éste deslizable de forma que se alternen las vistas (final y la de debug seleccionada en el submenú **Frame**).

Bibliografía

[1] F. Julian, W. Magnus, K. Christopher, H. Habel *Production Volume Rendering*. SIGGRAPH Course. 2017 → págs 2 y 43

[2] B. Mark, T. Berechet, T. Mahlmann, J. Togelius. *Procedural Generation of 3D Caves for Games on the GPU*. FDG. 2015 → págs 2 y 43

[3] C. Crassin, F. Neyret, S. Lefebvre, E. Eisemann. *Gigavoxels: Ray-guided Streaming for Efficient and Detailed Voxel Rendering*. Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games. ACM. 2009 → págs 2 y 37

[4] C. Crassin, F. Neyret, M. Sainz, S. Green, E. Eisemann, *Interactive Indirect Illumination using Voxel Cone Tracing*. Computer Graphics Forum (Vol. 30, No. 7). Pages 1921–1930. 2011 → pág. 2

[5] M. Mittring. *The Technology Behind the Unreal Engine 4 Elemental Demo*. Advances in Real-Time Rendering in 3D Graphics and Games. SIGGRAPH. 2012 → pág. 2

[6] W. E. Lorensen, H. E. Cline, *Marching Cubes: A High Resolution 3D Surface Construction Algorithm*. ACM SIGGRAPH Computer Graphics (Vol. 21, No. 4). ACM. 1987 → pág. 38

[7] T. Ju, F. Losasso, S. Schaefer, J. Warren. *Dual contouring of hermite data*. ACM transactions on graphics (TOG) (Vol. 21, No. 3). ACM. 2002 → pág. 38

[8] S. Schaefer, J. Warren. *Dual Marching Cubes: Primal Contouring of Dual Grids*. Computer Graphics and Applications. Proceedings. 12th Pacific Conference. IEEE.

Pages 70–76. 2004 → pág. 38

[9] J. Gregory. *Game Engine Architecture*. CRC Press. 2009 → pág. 48

[10] M. Hadwiger, P. Ljung, C. R. Salama, T. Ropinski. *Advanced Illumination Techniques for GPU Volume Raycasting*. ACM Siggraph Asia 2008 Course. 2008 → pág. 37

[11] T. J. Purcell, I. Buck, W. R. Mark, P. Hanrahan. *Ray Tracing on Programmable Graphics Hardware*. ACM Transactions on Graphics (TOG) (Vol. 21, No. 3). ACM. Pages 703–712. 2002 → pág. 18

[12] R. Rauwendaal. *Hybrid Computational Voxelization using the Graphics Pipeline*. Doctoral dissertation. 2012 → pág. 56

[13] C. Tripiana Montes. *GPU Voxelization*. Master Thesis. Universitat Politècnica de Catalunya. 2009 → pág. 56

[14] T. Akenine-Möller. *Fast 3D Triangle-Box Overlap Testing*. In ACM SIGGRAPH 2005 courses. ACM. 2005 → pág. 60

[15] J. Klint. *Deferred Rendering in Leadwerks Engine*. Copyright Leadwerks Corporation. 2008 → pág. 62

[16] A. Lauritzen. *Deferred Rendering for Current and Future Rendering Pipelines*. ACM SIGGRAPH Course: Beyond Programmable Shading. 2010 → pág. 62

[17] B. T. Phong. *Illumination for Computer Generated Pictures*. Communications of the ACM. 1975 → pág. 79

[18] J. F. Blinn. *Models of Light Reflection for Computer Synthesized Pictures*. Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques. 1977 → pág. 80

[19] M. Mittring. *Finding Next Gen: Cryengine 2*. ACM SIGGRAPH courses. ACM. 2007 → pág. 82

[20] J. Chapman. *SSAO tutorial*. 2011 → pág. 84

[21] R. Fernando, E. Haines, T. Sweeney. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Dimensions. 2001 → pág. 87

[22] R. Fernando. *Percentage-closer Soft Shadows*. ACM SIGGRAPH Sketches. 2005 → pág. 88

[23] T. Lottes. *FXAA*. `http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf`. 2011 → pág. 90

[24] T. Akenine-Möller. *Fast 3D Triangle-Box Overlap Testing*. ACM siggraph courses. 2005 → pág. 98

[25] T. Masaya. *The Basics of GPU Voxelization*. `https://developer.nvidia.com/content/basics-gpu-voxelization`. 2015 → pág. 100

[26] C. Crassin, S. Green. *Octree-based Sparse Voxelization using the GPU Hardware Rasterizer*. OpenGL Insights, 303-318. (2012) → pág. 102

[27] O. Mattausch, J. Bittner, A. Jaspe Villanueva, E. Gobbetti, M. Wimmer, R. Pajarola. *CHC+RT: Coherent Hierarchical Culling for Ray Tracing*. Computer Graphics Forum (Vol. 34, No. 2). Pages 537–548. 2015 → pág. 18