Estudio e implementación de técnicas de renderizado en tiempo real sobre un prototipo de motor gráfico

Alejandro Catalán Cebollada

TREBALL FI DE GRAU

GRAU EN ENGINYERIA EN INFORMÀTICA

ESCOLA SUPERIOR POLITÈCNICA UPF

ANY 2015

DIRECTOR DEL TREBALL

Javier Agenjo, Departament GTI

upf. Universitat Pompeu Fabra Barcelona

Agradecimientos

Quiero agradecer a toda la gente, familia, amigos que durante el tiempo que ha durado el desarrollo de este proyecto y sin saber de qué les estaba hablando, han escuchado mis largas explicaciones sobre el proyecto y me han ayudado sin decir una palabra.

Le agradezco a mi tutor del TFG y profesor, Javi Agenjo, no haberme puesto las cosas fáciles y enseñarme lo interesante que puede llegar a ser el mundo de la programación gráfica. Por último, le agradezco a mi compañero de batallas durante la carrera, Héctor Ruiz, el aguantarme estos 5 años.

Resumen

En el presente trabajo se realiza el análisis de diferentes técnicas de renderizado en tiempo real, y la implementación de la técnica Deferred Shading sobre un prototipo de motor gráfico implementado en WebGL. El objetivo del proyecto es comprobar la capacidad del entorno de ejecución para renderizar en tiempo real escenas con un sistema de iluminación utilizando las diferentes técnicas de renderizado.

La primera parte del proyecto se basó en el aprendizaje de la tecnología necesaria y la implementación del motor gráfico desarrollado para la web utilizando la API de WebGL, utilizando Forward Rendering como técnica de renderizado.

En la segunda parte del desarrollo, se estudian las diferentes técnicas de render y se implementa Deferred Shading, para después realizar técnicas de Benchmarking a diferentes niveles de exigencia para estudiar y comprobar las ventajas y desventajas sobre Forward Rendering, y poder sacar conclusiones.

Summary

This project does an analysis of the different real time render techniques, and implements the Deferred Shading technique on a graphic engine prototype using WebGL. The purpose of this project is to test the capability of the runtime environment to render scenes with a lighting system using different real-time rendering techniques.

The first part of the project was based on learning the required technology to implement the graphic engine using the WebGL API for the web, and using Forward Rendering as render technique.

In the second part, the different render techniques are studied and it is implemented the Deferred Shading, and then perform benchmarking techniques with different levels of demand to study and check the advantages and disadvantages over Forward Rendering, and finally get some conclusions.

Índice

1. INTRODUCCION

En la última década la potencia de los navegadores web ha ido evolucionando de forma exponencial. Estos han llegado a tal nivel que son capaces de renderizar imágenes 2D/3D sin la utilización de ningún plug-in externo, utilizando WebGL que es una API Javascript de renderizado de Computación Gráfica 3D y 2D interactiva, dando cabida al desarrollo de aplicaciones con animaciones 2D y 3D.

Además, hoy en día en la campo de la Computación Gráfica 3D, existen diferentes técnicas para realizar los cálculos necesarios para renderizar en una imagen 2D los datos geométricos que se utilizan para representar una escena 3D. Estas técnicas se separan en dos grandes subcampos, que separan la generación de una imagen para ser almacenada y vista en otro momento, y por otro lado tenemos la generación de imágenes en tiempo real.

En el primer subcampo, el de la generación off-line de imágenes, el proceso de generación de la imagen es más elaborado y detallado, ya que no está limitado en cuestión de tiempo de generación, y puede realizar más cálculos para incrementar la calidad de la imagen resultante generando un coste computacional mayor. Como ejemplos de este subcampo tenemos Ray Tracing, Radiosity, Ambient Occlusion. Algunas de estas técnicas se utilizan junto a técnicas de renderizado en tiempo real para añadir efectos especiales de iluminación a la escena.

El renderizado en tiempo real, se concentra en la generación de imágenes en tiempo real a partir de un modelo 2D o 3D, normalmente en referencia en la Computación Gráfica 3D interactiva, y utilizando la GPU para realizar la mayoría de los cálculos necesarios para el renderizado de la imagen. Los términos “tiempo real” en este tipo de renderizados vienen dados ya que la generación de las imágenes es lo suficientemente rápida como para generar la ilusión de movimiento en el cerebro del usuario. La Rasterización es la técnica más popular debido a su gran velocidad, aunque la Rasterización es solamente el proceso de computación del mapeado de la geometría de la escena a los pixeles, y no define una forma particular de calcular el color de esos pixeles. Para el cálculo del color, se sirve de técnicas de Shading que veremos y estudiaremos en este trabajo.

1.1. Motivación

La principal motivación de este trabajo es el estudio de diferentes técnicas de Deferred Rendering, además su implementación sobre un engine gráfico WebGL desarrollado expresamente para el proyecto. Estas motivaciones vienen dadas por el interés personal hacia la computación gráfica y el desarrollo de aplicaciones gráficas de entretenimiento que se basan en las técnicas estudiadas para sus sistemas de renderizado de escenas.

1.2. Background

Durante los últimos años de la carrera, he dedicado todos mis esfuerzos a aprender lo máximo posible en el área de la Computación Gráfica, Interactividad, Ingeniería de Juegos, con el objetivo de salir con una buena base al mundo laboral, que me permita acceder a empresas que trabajen en estas áreas. Realizar este proyecto supone un nuevo reto, que me aportará nuevos conocimientos que me servirán de cara al futuro.

2. ANÁLISIS

2.1. La Web

Internet está concebido como un medio para compartir información entre diferentes *peers*. Entre otros muchos servicios de Internet, se creó la World Wide Web (WWW), que es un conjunto de documentos y otros recursos web interconectados entre sí mediante la utilización de hyperlinks y URLs.

Estos documentos web, inicialmente estaban formados por páginas estáticas que contienen información en forma de texto plano. Con el tiempo, fueron apareciendo lenguajes de programación y librerías que se utilizarían para el desarrollo y mejora de las capacidades de estos documentos web:

- **HyperText Markup Language (HTML):** lenguaje de marcado, que a partir de 1991, mediante el uso de 22 etiquetas permitió dar una estructura a estas páginas, además de permitir mostrar imágenes. Y que ha evolucionado hasta la versión HTML5 con alrededor de 100 etiquetas.
- **Cascading Style Sheets (CSS):** lenguaje utilizado para definir y crear el aspecto que tendrá el documento web. La idea detrás de CSS es separar la estructura del documento, de su aspecto visual.
- **JavaScript:** lenguaje de programación interpretado, orientado a objetos, basado en prototipos. Se utiliza principalmente en el lado del cliente (client-side), permite mejoras en la interfaz de usuario y páginas web dinámicas.
- **<canvas>:** es uno de los elementos que incluye HTML5, que nos permite crear representaciones dinámicas programables de imágenes de mapa de bits y formas 2D. JavaScript puede acceder a la zona a través de un conjunto de funciones similares a las de otras APIs de dibujo 2D, permitiendo que los gráficos sean generados dinámicamente. Algunos de los usos previstos la generación de gráficos, animaciones, juegos, y la composición de imágenes.

- **CSS3:** la evolución de este lenguaje ha llegado al punto de poder crear animaciones, gradientes, transiciones, transformaciones 2D y 3D, sobre los elementos del documento web.
- **WebGL:** El siguiente paso evolutivo para las páginas web es la generación de gráficos 2D y 3D acelerados por hardware (GPU), en tiempo real, sin la necesidad de plugins. WebGL técnicamente es una API de JavaScript, que permite usar la implementación nativa de OpenGL ES 2.0.

Todos estos elementos, librerías y lenguajes de programación, nacen con el objetivo de crear páginas web cada vez más dinámicas e interactivas y transmitir la información de una forma más visual para el humano, ya que los humanos son capaces de asimilar muchísima más información en un formato visual que a través de las palabras.

2.2. Graphics Pipeline

En un sistema de generación de imágenes en tiempo real, la Graphic Pipeline es la encargada de representar una escena 3D que recibe como entrada y generar una imagen 2D como resultado. En el pasado, las GPU estaban más limitadas en las acciones que podían llegar a realizar, no podían cambiar como pintar cada pixel aparte de enviando una textura diferente, tampoco se podía modificar los vértices una vez en la tarjeta. Pero los tiempos han cambiado y hoy en día las GPU son capaces de permitir al desarrollador programar como pintar los pixeles, modificando las normales y añadiendo reflejos, dando un gran realismo a la escena.

La Graphic Pipeline es el conjunto de pasos que suceden desde que se pasan los vértices a la GPU hasta que aparece la imagen en pantalla. Estos vértices se someten a una transformación e iluminación por vértice, pasando por un programa de vértices shader, después pasa por otro programa de geometría shader que puede generar nuevas primitivas que serán rasterizadas, un tercer programa será ejecutado en cada fragmento para calcular el color de cada pixel.

Vista simplificada de una pipeline gráfica programable

2.2.1. Modelos de Iluminación y Sombreado

Se entiende por modelo de iluminación como el cálculo de la intensidad de color de cada punto de la escena. En el cálculo de la intensidad intervienen el tipo y la intensidad de las luces que afectan a la superficie del objeto, el material del que está formado el objeto y la orientación del objeto respecto a la luz.

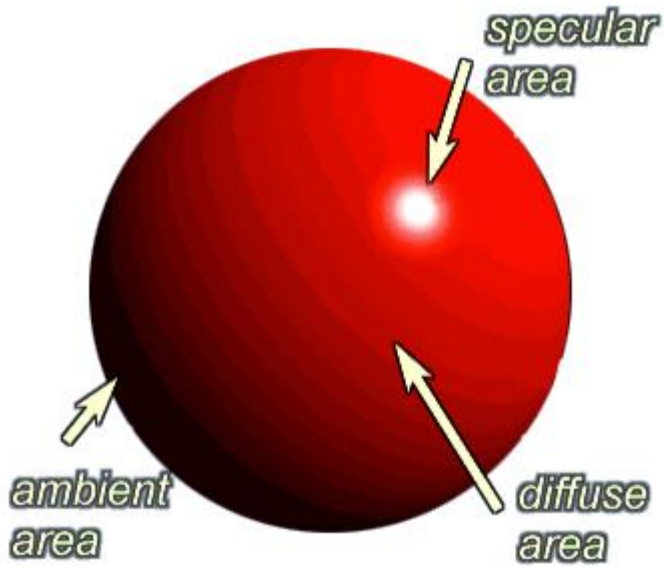
Existen dos tipos de luz, la primera, la **Luz Posicional** es una fuente de luz que emite en todas direcciones desde un punto dado, suelen representar la iluminación generada por una bombilla o una vela. El segundo tipo, es la **Luz Direccional**, es una fuente de luz ubicada en el infinito, todos los rayos de luz son paralelos y viajan en la misma dirección, se utiliza para simular la luz en escenarios exteriores como la emitida por el sol o la luna. Aunque no se puede considerar otro tipo de luz, la **Luz Focal** o Foco de luz, es un efecto que se utiliza para simular la iluminación generada por una linterna o lámpara, ya que se genera un cono limitando los efectos de una fuente de luz posicional a un área de la escena.

2.2.1.1. Modelo de Iluminación de Phong

Es un modelo empírico simplificado para iluminar puntos de una escena. En este modelo, los objetos no emiten luz, sólo reflejan la luz que les llega de las fuentes de luz o reflejada de otros objetos. Para calcular la luz reflejada por la superficie del objeto, Phong tiene en cuenta 3 aspectos:

- **Luz Ambiental**: proviene de todas las direcciones e ilumina al objeto por todos lados por igual.

- **Reflexión Difusa**: proviene de una dirección, pero se propaga en todas direcciones. Es característica de superficies rugosas, mates o sin brillo. La **Atenuación** es un aspecto a tener en cuenta para el cálculo de la reflexión difusa, ya que la luz pierde intensidad al viajar desde su origen hasta el objeto.
- **Reflexión Especular**: proviene de una dirección, y se refleja solo en una dirección o un rango de ángulos muy cercano al ángulo de reflexión perfecta. Es propia de superficies brillantes, pulidas y responsable de los brillos que suelen observarse. El color del brillo suele ser diferente del color de la superficie y muy parecido al color de la fuente de luz.



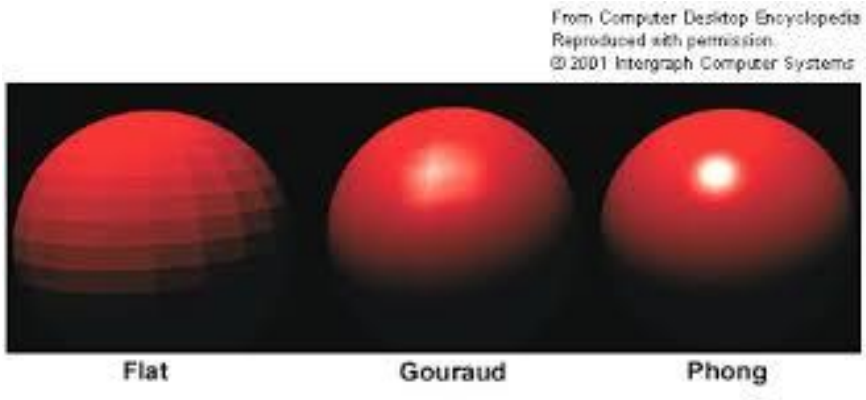
Modelo de iluminación de Phong

El modelo de Phong también tiene en cuenta el material, del que está formada la superficie del objeto, al calcular la iluminación y así proporcionar mayor realismo. Las propiedades del material vienen definidas por 4 factores, **Ambiente**, **Difusa**, **Especular** y el **Brillo**. Las tres primeras afectarán a la cantidad de Luz Ambiente, Reflexión Difusa y Especular, respectivamente, que absorberá el objeto, y el Brillo afectará a la intensidad de la Reflexión Especular.

2.2.1.2. Sombreado

Un modelo de sombreado se provee de un modelo de iluminación para la visualización de los polígonos que forman un objeto.

- **Sombreado Constante o Plano**: el modelo de iluminación se aplica una sola vez y sobre toda la superficie del polígono. Este método requiere la normal de cada polígono.
- **Gouraud**: el modelo de iluminación se aplica en cada vértice del polígono y los resultados se interpolan sobre su superficie. Este método requiere la normal en cada uno de los vértices del polígono.
- **Phong**: el modelo de iluminación se aplica en cada fragmento del polígono. Este método requiere la normal en el fragmento, que se puede obtener por interpolación de las normales de los vértices.



Modelo de sombreado Plano, Gouraud y Phong

2.2.1.3. Múltiples Luces

En cualquier tipo de escena, es más que probable encontrarnos con más de una luz iluminando los objetos de la escena. Por lo que cuando renderizamos el objeto, será necesario hacer el cálculo de su iluminación para cada luz. Para resolver esta situación, existen diferentes técnicas que se pueden aplicar:

- **Multipass lighting**: esta técnica se basa en calcular la iluminación, sobre el objeto, por cada luz en la escena y sumarla a la imagen final. Una primera

pasada, con la luz ambiente, y después, activando el Blending en modo aditivo, se irá sumando el color de las luces sobre la superficie del objeto en el buffer de salida.

- **Varias luces por shader**: esta solución se basa en pasar al shader un grupo de luces, para que realice el cálculo de iluminación de estas luces sobre el objeto a la vez.
- **Lightmaps**: son una estructura de datos precomputada que contienen el brillo de la superficie del objeto. Normalmente se utilizan para iluminar objetos estáticos.

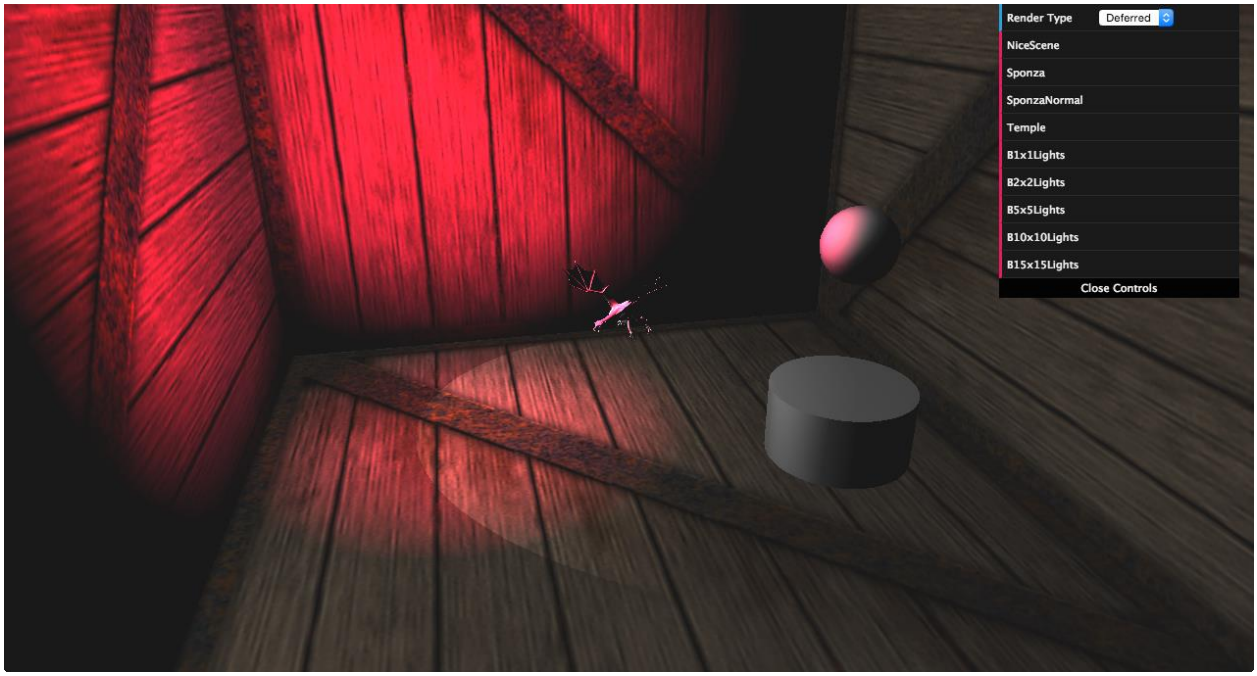


Imagen del Engine, utilizando los 3 tipos de luz.

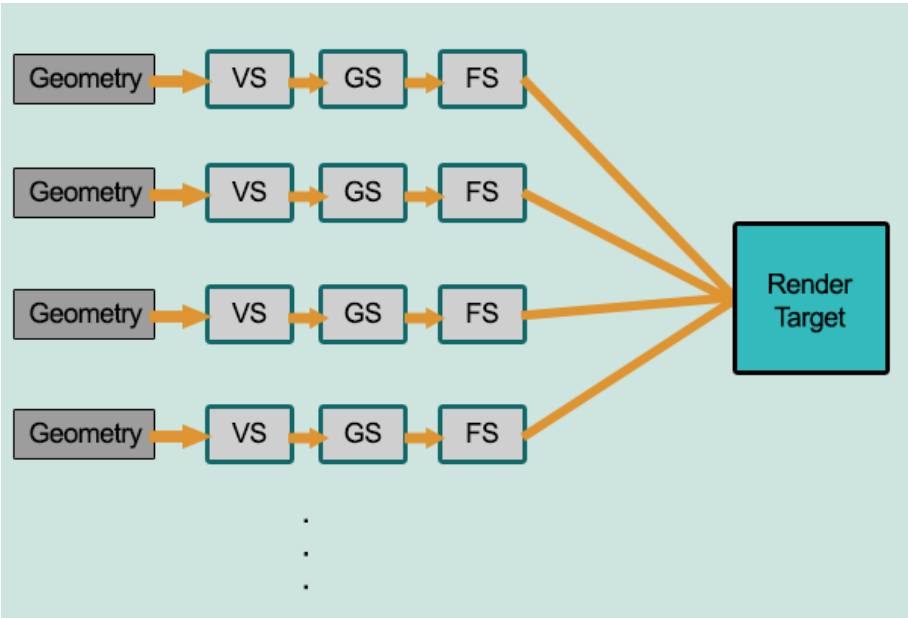
En la imagen superior obtenida del Engine, podemos observar los 3 tipos de luces en la misma escena. El detalle que identifica que existe una Luz Direccional, es en el cilindro, está iluminado por el lado izquierdo y sombreado por el derecho, además la tapa superior del cilindro tiene un color plano por lo que la luz que le afecta y el ángulo es el mismo, por lo que no es una luz puntual. También se puede observar que el plano inferior y derecho están iluminados mostrando el color de la textura, mientras que el plano izquierdo y el frontal solo están iluminados por la luz roja hasta donde alcanza, lo que significa que están "de espaldas" a la luz. Por último si observamos la esfera suspendida en el aire, vemos que su sombra empieza en un ángulo diagonal. Con estas

observaciones podemos decir que la Luz Direccional está iluminando en dirección hacia "abajo-atrás-derecha".

La Luz Puntual se observa claramente iluminando los planos frontal e izquierdo y un poco el inferior, de color rojo. Se puede observar en la esfera el brillo de la componente especular de la luz puntual. La Luz Focal, se ve en el plano inferior, creando un arco que ilumina de blanco parte de la zona donde está iluminado en rojo por la luz puntual, hacia la derecha difuminándose, ya que está en un ángulo diagonal hacia la derecha y plano queda fuera del rango antes de cerrar el círculo que se vería por el foco.

2.2.2. Forward Rendering

Forward Rendering es la técnica estándar más utilizada por los engines gráficos. Se pasa a la tarjeta gráfica la geometría de la escena, y esta se proyecta, se transforma y se parte en fragmentos o píxeles, y se realizan los cálculos necesarios para definir la iluminación según la información provista (material, luces que afectan al píxel, etc.), para generar el color del pixel que tendrá finalmente en pantalla. Es un proceso lineal por el que ha de pasar toda la geometría para renderizar la escena completa en la imagen final.

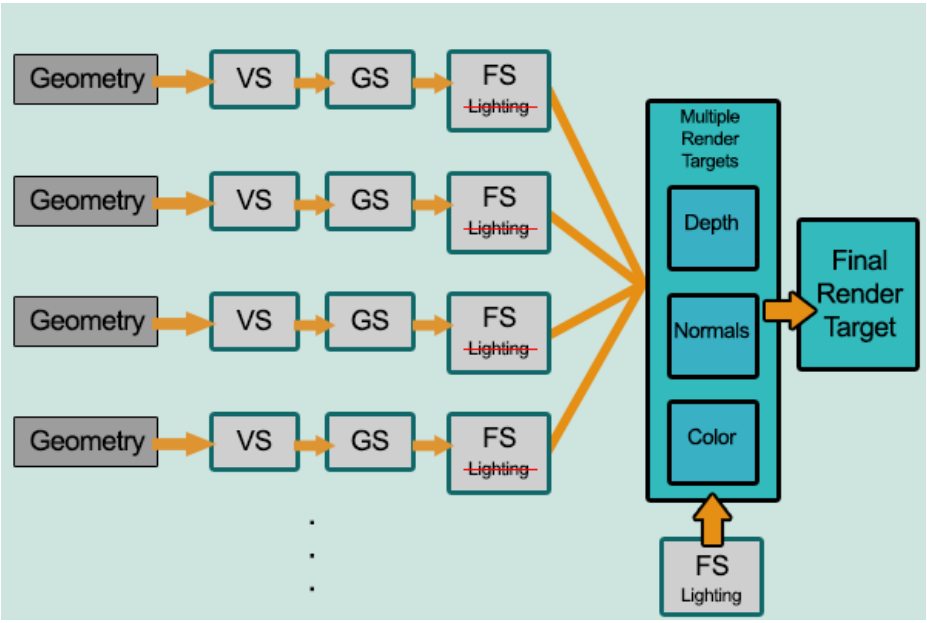


Forward rendering: Del Geometry Shader al Vertex Shader al Fragment Shader.

Una de las grandes desventajas que tiene Forward Rendering es que al incrementar el número de luces u objetos en la escena, el rendimiento desciende, debido a que Forward Rendering renderiza cada objeto una vez para cada luz. Por ejemplo, en una escena con 5 objetos y 5 luces, significa tener que renderizar cada objeto 5 veces, para que se vea reflejado en la imagen que está iluminado por 5 luces, esto significa realizar 25 operaciones por frame. Cuando incrementamos el número de luces, y tenemos 50 luces en nuestra escena, pasamos a realizar 250 operaciones por frame, esto nos da una complejidad de `O(N_fragmentos_geometria x L_luces)`. Este sistema además de que incrementa su coste a cada luz y cada objeto que se añade a la escena, también realiza trabajo repetido en cada pasada, o trabajo inútil para los polígonos ocluidos.

2.2.3. Deferred Rendering

En Deferred Rendering, se realiza una primera pasada para guardar en un buffer memoria la información necesaria que podemos obtener de la geometría de la escena y la información del material de los objetos, el Geometry Buffer (G-Buffer). La imagen final se genera al hacer el cálculo de la iluminación de la escena sobre la información guardada en el G-Buffer.



Deferred rendering: Del Geometry Shader al Vertex Shader al Fragment Shaders. Pasados a multiples Render Targets, entonces se sombrea con el paso de iluminación.

2.2.3.1. G-Buffer

Dentro del Geometry Buffer se guarda la información necesaria para reconstruir la escena en otras fases del render. Algunos de los datos que puede ser útil guardar en el G-Buffer pueden ser: la posición o la profundidad en la Screen View, Lighting Accumulation RGB (color ambiente, reflejos de escena,...), intensidad de la luz, la normal, el poder y la intensidad de la componente especular del material, la componente difusa del material (el color de la textura), o la oclusión ambiental. Cada implementación de Deferred, utilizara el G-Buffer según las necesidades del render.

G-Buffer utilizado en Killzone 2

Toda esta información del G-Buffer se guarda en diferentes buffers intermedios de memoria de la GPU, o también llamados **Render Targets Texture** (RTT), en vez de guardarlos en el Frame Buffer o el Back Buffer, que son los buffers utilizados para mostrar por pantalla o ir generando el siguiente frame que se mostrará, respectivamente, y que se van alternando en cada frame.

2.2.3.1.1. Depth Buffer

En el caso de la profundidad, se guarda en un buffer aparte, el Depth Buffer. El cálculo de la profundidad de los objetos en el espacio de vista de la cámara (frustum) es muy común. Por ejemplo, durante el render de cada objeto, se realiza un test de profundidad, que definirá si se debe guardar o no la información del objeto en ese fragmento que está renderizando en ese momento. A medida que se vayan rellenando

los fragmentos, se guardará en el Depth Buffer la profundidad correspondiente al fragmento del objeto renderizado, si se ha de renderizar un fragmento ya ocupado, realizará el Depth Test, si el fragmento tiene menor profundidad guardará su información el Screen Buffer ya que estará más próximo a la cámara y tapara el objeto que tiene detrás, sin o descarta el fragmento y se mantendrá la información del objeto que se había renderizado anteriormente en ese fragment.

2.2.3.1.2. Reconstrucción de la posición

Para recuperar la posición a partir de la profundidad, se deberá realizar el proceso inverso que realiza la Pipeline al pintar el pixel de la pantalla. Por lo que primero hay que entender como la Pipeline proyecta los objetos hacia la pantalla. Para ello, realiza 3 cambios en el sistema de coordenadas del objeto, primero transforma las coordenadas locales del objeto a coordenadas de mundo multiplicándolas por la Model Matrix, segundo transforma las coordenadas de mundo a coordenadas de cámara multiplicando por la View Matrix, y por último transforma las coordenadas de cámara a coordenadas de pantalla multiplicando por la Projection Matrix. Estos 3 pasos se pueden unir en uno solo, creando la ModelViewProjection Matrix (MVP), que es la multiplicación de las 3 matrices.

Para realizar el proceso inverso, crearemos un vector de 4 componentes, con las coordenadas de pantalla del píxel XY y la profundidad obtenida del G-Buffer que deberán estar normalizadas entre [-1,1], y por último pondremos la coordenada W a 1.0. Este vector es teóricamente el que obtenemos de multiplicar las coordenadas locales del objeto por la MVP, por lo que su proceso inverso sería multiplicar por la inversa de la MVP (MVP^{-1}), como lo que nos interesa es conseguir las coordenadas de mundo, no es necesario deshacer la transformación de la Model, por lo que finalmente multiplicamos el vector por la inversa de la ViewProjection (VP^{-1}). Por último, como estamos trabajando con una proyección en perspectiva, la proyección no es lineal como en una proyección ortogonal, por lo que se debe corregir el resultado obtenido dividiendo las coordenadas XYZ por la W del mismo. Y ahora si hemos obtenido la coordenada de mundo del pixel que se está renderizando.

2.2.3.2. Complejidad y Limitaciones

Para calcular la complejidad de Deferred Rendering, en su implementación más sencilla, Deferred Shading, generar el G-Buffer solo supone una iteración sobre toda la geometría, y una vez generado, solo es necesario realizar una iteración sobre todas las luces, pasando el G-Buffer, para calcular la iluminación de la escena, por lo que tendrá una complejidad de `O(N_fragmentos_geometria + L_luces)`.

Pero Deferred Shading tiene sus limitaciones:

- Requiere tarjetas de video con múltiples render targets o renderizar la escena para cada buffer, lo que supone una bajada de rendimiento. Las tarjetas de video antiguas no los tenían y no hay solución para estas.
- Requiere un gran ancho de banda para enviar los grandes buffers de datos de un lado para el otro. Por lo que las tarjetas de video antiguas son lentas en realizar esta tarea.
- No hay transparencias.
- El Anti-Aliasing generado por el hardware no produce los resultados correctos.
- Solo un tipo de material. Deferred Lighting permite el uso de más materiales

2.2.3.3. Tipos de Deferred Rendering

Deferred Rendering se podría decir que es el concepto general para definir la esta técnica, de la que existen diferentes implementaciones que según las necesidades de la escena se utilizara una implementación u otra.

2.2.3.3.1. Deferred Shading

Se basa en una primera pasada (por la Pipeline) para generar el G-Buffer para los objetos opacos con todas las propiedades necesarias para calcular la iluminación y sombreado de la escena. Una vez generado el G-Buffer se realiza una segunda pasada de iluminación, que utilizando el volumen geométrico o un quad para cada luz focal o

puntual, y toda la pantalla para la luz direccional, calculara la cantidad de luz que recibe la escena utilizando la información obtenida del G-Buffer con las propiedades de la superficie y la información de la iluminación generando una imagen de la escena iluminada.

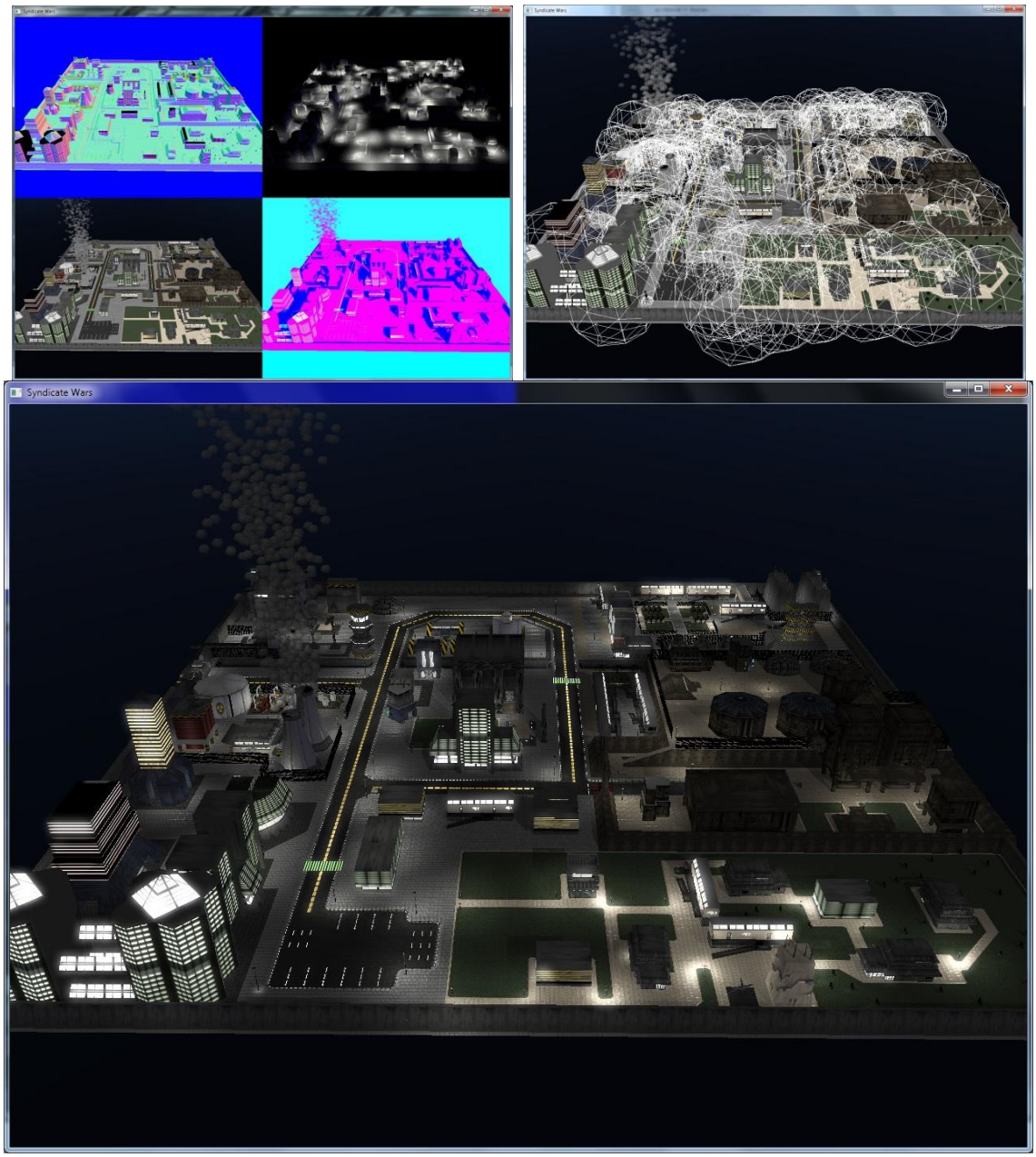
Deferred Shading G-Buffer

2.2.3.3.2. Deferred Lighting/Light pre-pass

Deferred Lighting es una modificación de Deferred Shading, esta técnica en vez de utilizar 2 pasadas, utiliza 3. La primera pasada se renderizan los objetos para guardar en el G-Buffer la información de profundidad, las normales, y el poder especular. En una segunda se computa la iluminación con la información del G-Buffer y se genera otro buffer de espacio de pantalla. En la tercera pasada, se vuelven a renderizar los objetos, que cogen la iluminación computada y la combinan con el color de las texturas y la iluminación ambiente y/o emisiva.

La aparente ventaja de esta técnica es la gran reducción del tamaño del G-Buffer. En cambio el coste para renderizar la escena es doble, ya que el separa el cálculo la irradiación (cálculo de luz) especular y difusa, mientras que Deferred Shading solo realiza un cálculo combinado de la irradiación. Dada la reducción del tamaño del G-

Buffer esta técnica supera parcialmente una desventaja de Deferred Shading, la dificultad de tener múltiples materiales.

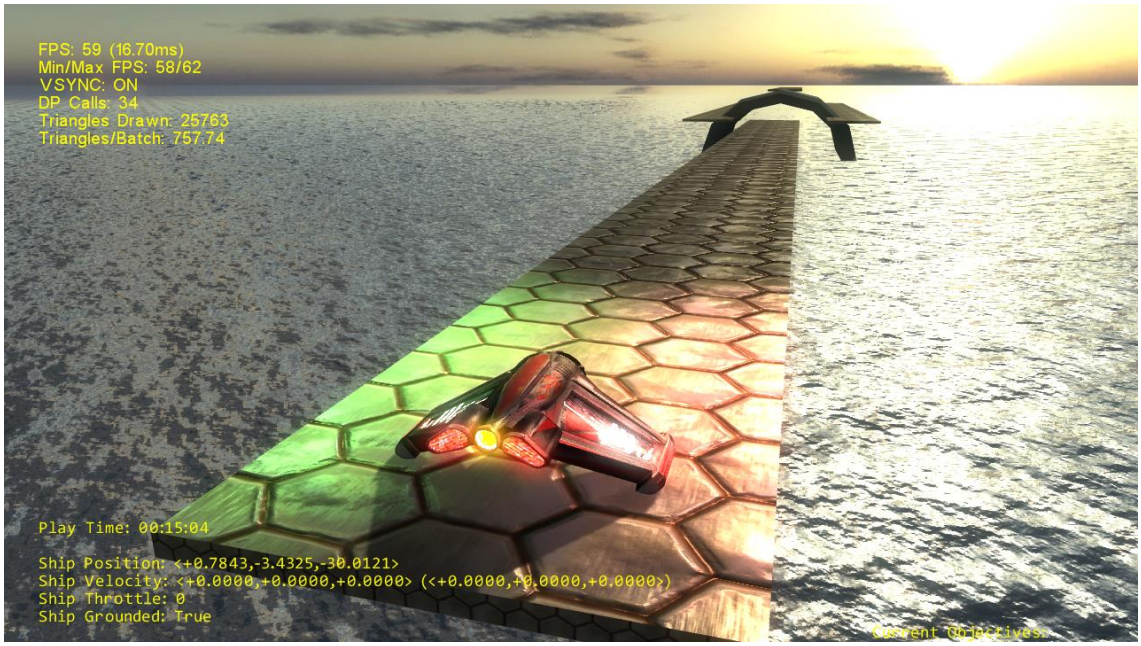


Deferred Lighting: G-Buffer, Light Volumes, Imagen Final

2.2.3.3.3. Inferred Rendering

Aunque a primera vista parece que Inferred Rendering sea una extensión de Light Pre-pass, ambas técnicas fueron desarrolladas separadas. La diferencia es que Inferred Rendering utiliza un G-Buffer con una resolución menor, resultando un uso de memoria

mucho menor. Este G-Buffer tiene los mismos elementos que con Light Pre-pass. El Lighting Buffer (L-Buffer) que se calcula con la información del G-Buffer, se renderiza con la misma resolución que el G-Buffer. Esto reduce drásticamente el número de cálculos de iluminación que son necesarios realizar por lo que incrementa el rendimiento. Pero el L-Buffer no puede ser aplicado directamente sobre la geometría en la pasada de material, ya que el material viene a resolución completa, por lo que el L-Buffer se escala a la resolución completa y se aplica utilizando un filtro bilinear.



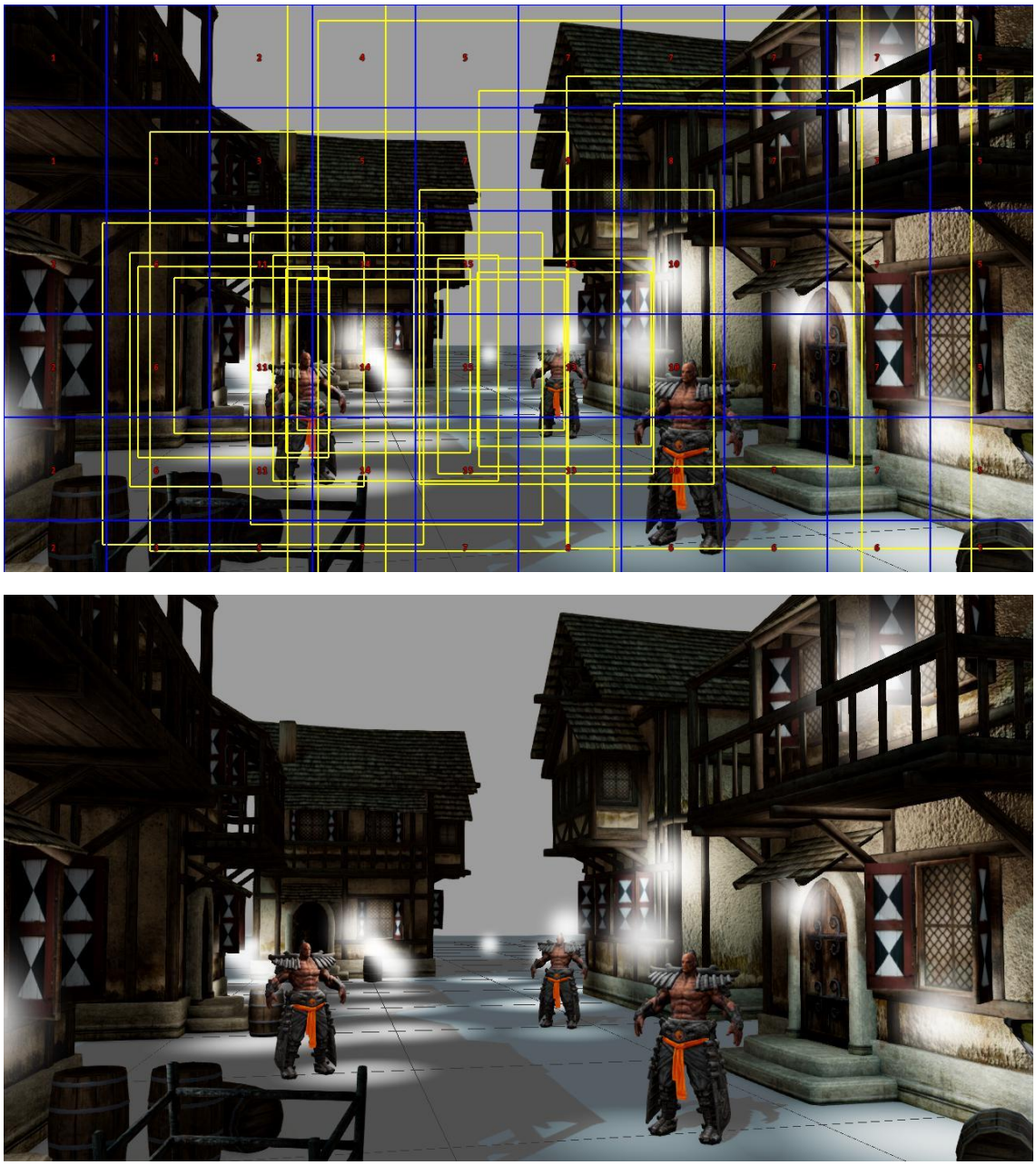
Ejemplo de imagen generada con Inferred Rendering

2.2.3.3.4. Tiled Shading

Este método trabaja almacenando luces en celdas de espacio de pantalla. Cada celda contiene una lista de las luces que le pueden potencialmente afectar, con esto consigue que cada celda sea procesada de forma independiente para computar la iluminación.

En Tiled Shading podemos encontrarnos con 2 versiones de la técnica, que son Deferred y Forward. Tile Deferred Shading es la versión que menor varianza de rendimiento tiene, y la que mejor escala con GPUs rápidas, por lo que es más adecuada con grandes cantidades de luces.

Por otro lado Tile Forward Shading, que es una generalización de Tile Deferred Shading, gestiona la iluminación y la escena de forma desacoplada, y al mismo tiempo, al contrario que el Deferred Shading tradicional y el Tiled Deferred Shading, soporta transparencias y Full Screen Anti-Aliasing (FSAA). Tile Forward Shading ha demostrado ser más competitivo con escenas con pocas luces, siendo más sencillo de implementar que otras técnicas de Forward Rendering tradicional.

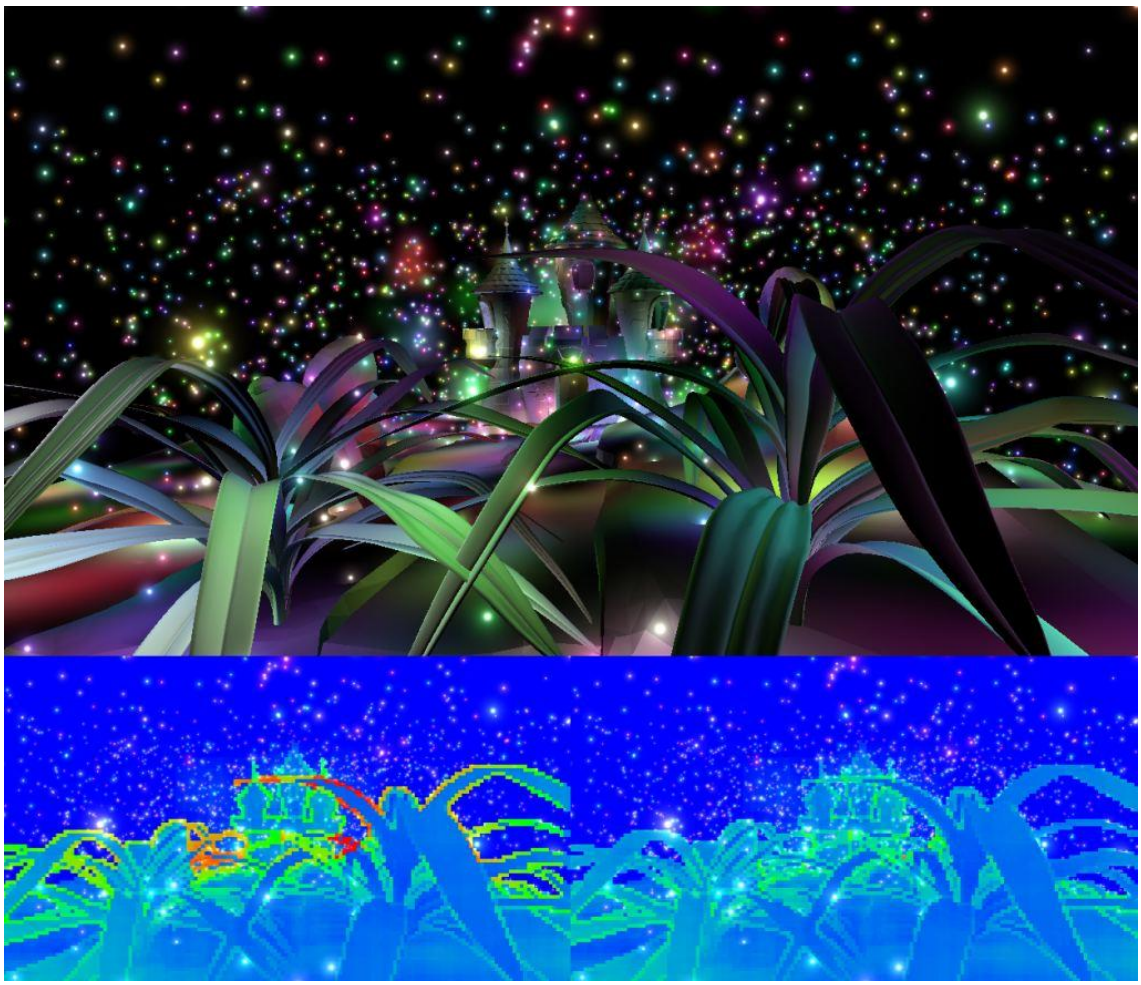


Tiled Shading Example

2.2.3.3.5. Forward+

Es un método de renderizado de muchas luces, guardando y haciendo culling solo las luces que afecten al pixel con el que se está trabajando. Forward+ es una extensión del Forward Rendering tradicional. El light culling, implementado utilizando la capacidad de computación de la GPU, es añadido a la Pipeline para crear una lista de luces, esta lista se pasa al final del shader de renderizado, que podrá acceder a toda la información de las luces.

Aunque Forward+ incrementa la carga de trabajo del shader final, teóricamente requiere un tráfico de memoria menor comparado al Deferred Lighting. Además, elimina la mayoría de las desventajas de las técnicas de Deferred, como la restricción de materiales y modelos de iluminación.



2.5D Culling for Forward+. Siggraph 2012

2.2.4. Forward Rendering vs Deferred Rendering

Una vez hemos estudiado ambas técnicas, y algunas implementaciones de Deferred Rendering, podemos compararlas entre ellas.

La complejidad de Forward es de `O(N_fragmentos_geometria x L_luces)`, mientras que la de Deferred `O(N_fragmentos_geometria + L_luces)`, lo que nos permite añadir más luces en Deferred. Por otro lado Deferred Shading nos limita el número de materiales que podemos utilizar, si queremos tener más materiales se deberá utilizar Deferred Lighting, Forward en cambio tiene más libertad con los materiales. Igual pasa con las transparencias, en Deferred hemos de utilizar Tiled Deferred Shading, mientras que Forward es capaz de renderizarlas.

En resumen, si la escena a renderizar no tiene un gran número de luces pero es más compleja artísticamente (número de materiales, transparencias,...), se puede tirar hacia delante con Forward Rendering. En cambio si el número de luces incrementa considerablemente, Deferred Rendering en cualquiera de sus implementaciones es la opción a elegir, aun teniendo cada implementación ciertas cadencias y teniendo que sacrificar ciertas características de la escena.

2.3. WebGL

Web Graphics Library (WebGL) es una API JavaScript para el renderizado interactivo de gráficos computarizados 3D y gráficos 2D en cualquier navegador web compatible con esta API, sin el uso de plugins externos. WebGL está completamente integrado en todos los estándares del navegador, permitiendo a la GPU acelerar el procesado de imágenes y efectos como parte del canvas de la página web. Los elementos de WebGL pueden estar mezclados con otros elementos HTML y componer otras partes de la página.

Los programas WebGL consisten en código escrito en JavaScript para realizar los cálculos que se realizan en el lado de la CPU, y código Shader que se ejecuta en la GPU del ordenador que está ejecutando la página web.

2.3.1. Diseño

WebGL está basado en OpenGL ES 2.0 por lo que utiliza el lenguaje de programación de shaders de OpenGL, GLSL, ofreciendo la familiaridad del entorno de desarrollo de OpenGL. Además al funcionar sobre un elemento Canvas de HTML5, tiene integración completa con la interfaz de Document Object Model (DOM). La gestión de memoria es automática gracias a que forma parte del lenguaje de programación JavaScript.

WebGL es una API de bajo nivel, por lo que su utilización no es trivial, GLSL en sí es un entorno de programación entero. Por lo que, incluso las cosas más sencillas en WebGL requieren más código de lo normal. Has de cargar, compilar y linkar los shaders, preparar las variables para pasarlas al shader y realizar las operaciones matemáticas de matrices para animar las formas. Por lo que se deberá estar familiarizado GLSL, la computación de matrices para realizar transformaciones y conocer el funcionamiento de los buffers de memoria que se utilizan para guardar vértices de posición, normales, colores, texturas.

2.3.2. Ventajas Clave

Ya que WebGL está basado en OpenGL y está integrado en la mayoría de browser más populares: Google Chrome, Opera, Mozilla Firefox, Safari, Internet Explorer. WebGL ofrece ciertas ventajas:

- Una API basada en un estándar de gráficos 3D ampliamente familiar y aceptado.
- Compatibilidad tanto entre plataformas como entre navegadores.
- Estrecha integración con HTML, incluyendo la composición por capas, interacción con otros elementos HTML, y el uso de los mecanismos de HTML para la gestión de eventos.
- Gráficos 3D acelerados por hardware en el entorno del navegador web.
- El entorno de scripting proporcionado por JavaScript permite ver y debugar el render de los gráficos sin tener que compilar y linkar el código.

2.3.3. Desventajas

Evidentemente WebGL no es perfecto, estas son algunas de las desventajas de utilizar WebGL:

- Javascript es un lenguaje no tipado, por lo que convierte cualquier cosa en cualquier cosa. El parser es lento y no suele encontrar errores, por lo que en tiempo de ejecución tampoco, y esto lleva a que se pasen por alto errores de compilación que en C++ no.
- WebGL no tiene Geometry Buffer, por lo que los Geometry Shaders no están disponibles.

WebGL tampoco es capaz de gestionar buffers de índices 32 bits, por lo que no es capaz de tener buffers de más de 65536 vértices, e implica dividir el Vertex buffer para meshes grandes.

3. DISEÑO

Gran parte del trabajo inicial realizado para este proyecto, fueron tareas de investigación necesarias para el entendimiento de los requerimientos que exige la implementación del Engine, de forma que permitiese implementar sin grandes dificultades cualquier técnica de renderizado y de shading.

3.1. Diseño del Engine

Aprovechando la potencia que aporta WebGL y siendo una tecnología con un gran futuro tanto en el campo del desarrollo web como en el desarrollo de la computación gráfica 3D, se decidió implementar el Engine en un entorno web utilizando WebGL como API de gráficos 3D. Al tratarse de un proyecto web, facilitará su ejecución, sin tener que depender de grandes IDE, sino directamente sobre un navegador web y permitiendo su ejecución en cualquier dispositivo con acceso a la web y desarrollo en cualquier bloc de notas.

Los requerimientos de diseño del Engine están basados para intentar emular las características de engines comerciales, desarrollados y utilizados por grandes empresas como Unity3D, Unreal Engine, Cry Engine, etc. El Engine creará una escena que contendrá una lista de objetos, y cada objeto contendrá una lista de componentes que le otorgan al objeto las características para cumplir la función para la que se ha creado (una cámara, una luz, un objeto con representación 3D, emisor de partículas, un disparador de eventos, etc.). Más adelante se explicará la utilidad e implementación de cada componente implementado para este engine. También el Engine contendrá un sistema de renderizado que permitirá elegir entre las diferentes técnicas de renderizado, Forward Render y Deferred Shading. Para el Deferred Shading, se creará un G-Buffer para almacenar los datos de geometría. Una Interfaz de Usuario Gráfica que permitirá al usuario interactuar con la escena representada en pantalla. Y un sistema de generación de shaders mediante el uso de “MicroShaders”, que son pequeños fragmentos de código que enlazados generan un shader completo.

3.1.1. G-Buffer

Para nuestro G-Buffer, en una primera iteración lo más sencilla posible, se guardarán la posición XYZ, la normal XYZ y el color de la textura RGB, estos 3 datos son suficientes para calcular la iluminación de la escena.

R	G	B	A
posición.x	posición.y	posición.z	
normal.x	normal.y	normal.z	
color_textura.r	color_textura.g	color_textura.b	

Esta implementación es muy poco óptima, pero para una primera iteración, en la que se preparará el Shader de iluminación, será más que suficiente.

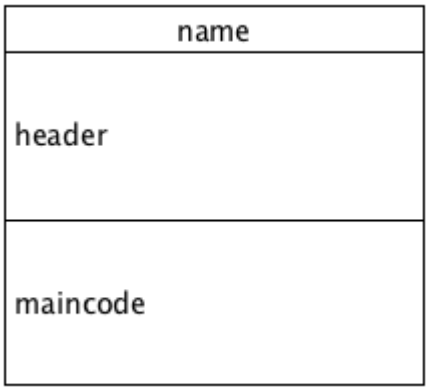
En una segunda iteración la posición se sustituirá por la distancia del fragment a la cámara o profundidad del Screen View, ya que conociendo la posición de la cámara y la profundidad a la que se encuentra el fragment, se puede recuperar la posición en el mundo del fragment.

R	G	B	A
profundidad			
normal.x	normal.y	normal.z	
color_textura.r	color_textura.g	color_textura.b	

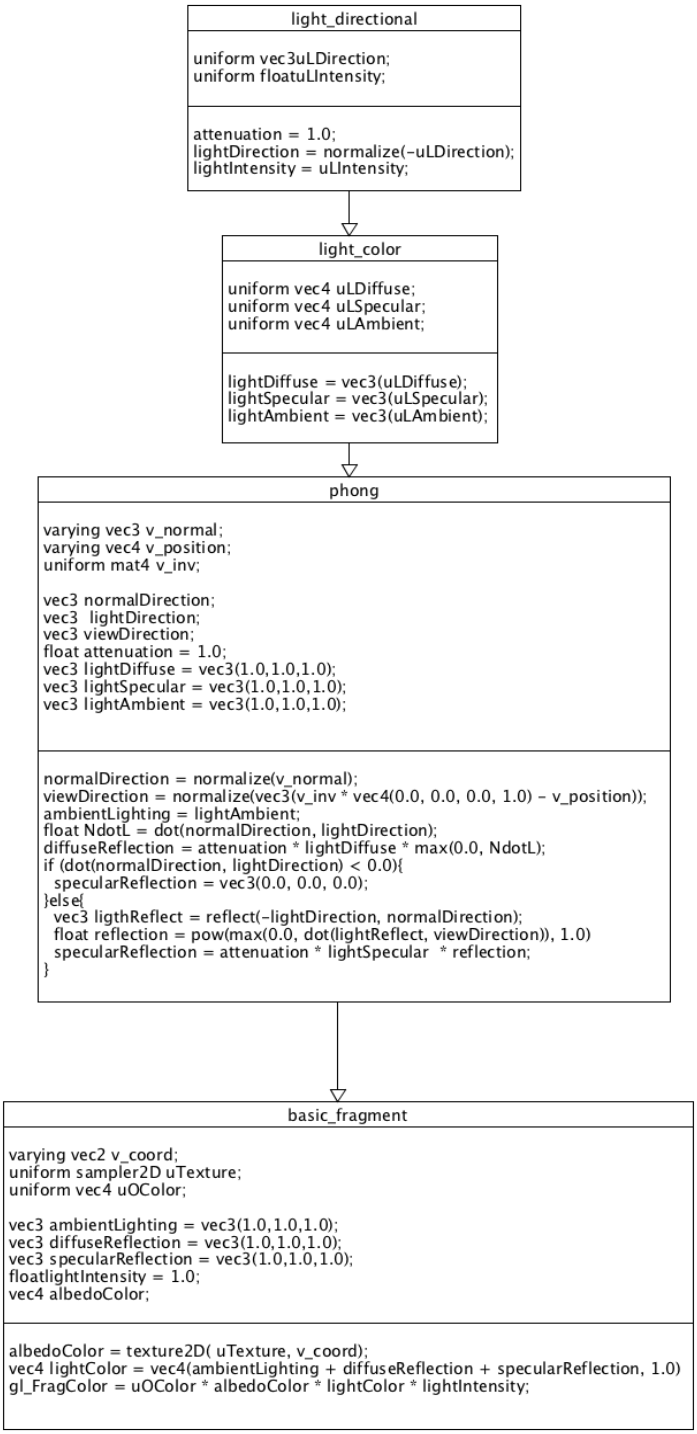
3.1.2. MicroShader Manager

Un Manager, es una clase Singletone, que sus métodos deberían ser accesibles desde cualquier otra clase de la aplicación, esté manager provee al resto del sistema, por petición, de un objeto Shader generado según los parámetros recibidos por el manager.

La peculiaridad de estos shaders es que se generan mediante la unión de pequeños trozos de código de shader (los MicroShaders) para formar el shader final. Estos MicroShaders están almacenados en un array dentro del Manager, que han sido cargados previamente de un archivo XML. Cada MicroShader se compone de 3 partes, un nombre único que lo identificará dentro del manager, un header que contiene las variables globales, uniforms, varyings y funciones que necesita definir; y un maincode que contendrá el código de la función main del shader.



Con esta estructura el Manager podrá crear un shader del tipo que sea, generando un string que contendrá el código. En la siguiente figura podemos ver cómo serían los 4 MicroShaders que forman el Fragment Shader para el cálculo de la luz direccional.



Simplemente cambiando el MicroShader “light_directional” por el correspondiente a la luz puntual o la luz de foco, estaríamos calculando la iluminación para este tipo de luces.

Para generar el objeto Shader con el Vertex y el Fragment Shader, el Manager recibe 4 parámetros: un nombre único para identificar el shader, un array de nombres

que identifiquen los MicroShaders que se quiere que formen el Vertex Shader, y otro array para el Fragment Shader, y el nombre del fichero XML en el que están almacenados para que en caso de no encontrar el MicroShader en el Manager pueda encontrarlo en el XML.

3.2. Estructura del engine

Se ha intentado que la estructura del engine sea lo más parecida posible a la de los grandes engines gráficos comerciales, intentando emular en cierta medida las características básicas que estos engines presentan, pero por otro lado intentando que sea lo más sencillo posible, para que sea mantenible, gestionable y que los objetivos de desarrollo sean asequibles para un solo desarrollador con un timing de desarrollo bastante reducido.

3.2.1. App

El documento App.js contiene la función Init, encargada de inicializar y poner en funcionamiento la aplicación, en este caso el engine, y gestionar el bucle principal definiendo el comportamiento de las llamadas del bucle, y guardar el objeto Scene.

Primero de todo al incluir el fichero se crea un objeto GL, con un ancho y alto, que definirán el tamaño del área del render. La función Init recoge el elemento <content> del HTML, del documento web sobre el que estamos ejecutando el proyecto, y crea el canvas (lienzo o rendering context) en el que se renderiza la escena y lo añade como elemento hijo al <content>.

Una vez creado el contexto GL, iniciará el bucle de animación, que consiste en llamar secuencialmente a la función Draw y Update del contexto GL. El comportamiento de estas funciones se definen a continuación, respectivamente llamaran a la función Draw de la clase Renderer pasándole el tipo de render, los objetos de la escena, las luces y la cámara activa, y Update del objeto Scene.

Además de definir el bucle principal, también se puede definir el comportamiento del mouse y del teclado, que se han implementado de forma que le pase el mensaje a la escena de forma que lo propague a todos los objetos. Se han creado dos pequeños

componentes en el mismo fichero App.js , que uno recoge el valor del teclado y el otro se suscribe a los eventos del mouse, que añadiéndolos a un GameObject podrán mover y rotar dicho objeto, más adelante se explicará que hacen estos componentes y cómo utilizarlos.

3.2.2. Scene

La escena es el contenedor en el que se guardan todos los objetos que han de existir en el mundo que vamos a crear con nuestro engine, y de gestionar el Update de todos los objetos. Esta clase, está hecha de forma que siga el patrón Singletone, ya que el engine está pensado para que solo exista una escena a la vez, y que guarda los objetos de esta.

El contenedor principal, tendrá todos los GameObjects sin importar su funcionalidad, han de estar todos reunidos en un mismo punto para que sea más fácil su gestión. Además de este contenedor, también se han creado 2 más para guardar los componentes de luz de la escena y los componentes de cámara, de forma que se pueda acceder a ellos más rápidamente.

La función Update se encarga de recorrer todos los GameObjects contenidos en la escena y llamar a la función Update de cada uno. La función que se ejecuta al pulsar una tecla del teclado, comprueba primero si ha de hacer algún cambio de estado y luego propaga a todos los objetos el evento de tecla pulsada. De la misma forma propaga a todos los objetos, el evento generado al mover el mouse.

3.2.3. Renderer

La clase Renderer se encarga de gestionar la Pipeline, recibe los objetos, las luces y la cámara activa de la escena, y según el tipo de render que este activo ejecuta la función de Forward o de Deferred. En este caso se trata de una clase estática, ya que solo tiene 3 métodos y no guarda ningún dato, por lo que no es necesario instanciarla para que haga su trabajo.

La App llamara a su método Draw, pasándole el tipo de render, y los objetos, luces y cámara, y según el tipo de render, propagara los parámetros al método ForwardRender o DeferredRender.

3.2.3.1. ForwardRender

El método ForwardRender, está implementado de la siguiente manera, primero limpia el buffer de color y el Depth Buffer, desactiva el blending en previsión de que llegue activado, y activa el Depth Test. Después de esto, calcula la matriz ViewProjection de la cámara, e inicia el bucle sobre cada objeto, en caso de que el objeto no esté activo o no tenga el componente ObjectRenderer, que veremos que hace más adelante, salta al siguiente objeto y no lo renderiza. Si pasa esta condición, empieza a recorrer el bucle de luces, al igual que los objetos, si la luz no está activa, o el objeto que tiene guardado el componente luz no está activado, saltara a la siguiente luz y no renderiza el objeto con esta luz.

Ahora que sabe que ha de pintar el objeto, con la luz actual, calcula la MVP con la ViewProjection y la Model del objeto (punto 2.2.3.1.2.), y agrupa todas las uniforms necesarias para el shader en una variable uniform, por ejemplo se le pasa la MVP, textura del objeto, atributos de la luz como la posición, color, tipo, intensidad, etc... Y le pide al componente ObjectRenderer del objeto, que renderize el objeto con las uniforms guardadas.

Finalmente se guarda en una variable que la siguiente luz no es la primera en renderizar el objeto, para en la siguiente activar el Blending aditivo y que las siguientes luces vayan sumando iluminación al objeto. Es decir está utilizando la técnica de Multipass Lighting (explicada en el punto 2.2.1.3. de este documento.)

3.2.3.2. DeferredRender

El método DeferredRender está dividido en dos partes, que son los dos pasos que se han de hacer para Deferred Shading. Pero antes de llamar a este método, cuando se incluye en el proyecto esta clase, ya se crean las 3 texturas que utilizaremos como

buffers de memoria para guardar el G-Buffer, que se crearán como variables globales fuera de la clase.

Al igual que en ForwardRender, primero calculamos la ViewPorjection de la cámara. Para generar el G-Buffer, primero enlazamos las 3 texturas como Render Target Textures (RTT), una para el albedo o color difuso del objeto (textura + color), otra para las normales, y la tercera para guardar la posición o la profundidad. Limpia los Depth y Color buffers, activa el Depth Test y desactiva el Blending ya que ahora no queremos que se mezclen los objetos al renderizarlos. Una vez hecho esto, inicia el bucle sobre cada objeto, al igual que en Forward, si el objeto está desactivado o no tiene ObjectRenderer, pasa al siguiente objeto, si no, calcula la MVP, bindea la textura del objeto para que llegue al shader, y pinta el objeto utilizando el shader que rellena las RTT, pasándole las uniforms necesarias para el G-Buffer, que habremos definido previamente. Hecho esto, pasa al siguiente objeto. Una vez pasado por todos los objetos, desenlaza las RTT y desactiva el Depth Test para terminar el bloque de generación del G-Buffer.

El siguiente paso es utilizar la información guardada en el G-Buffer y crear la imagen final con la escena iluminada. Para el Lighting Pass, lo primera será activar el Blending aditivo, enlazar las 3 texturas del G-Buffer, limpiar los Depth y Color Buffers y activar el Depth Test. Hecho esto, se inicia el bucle sobre cada luz y comprobaremos para cada luz si está activa o el objeto que la contiene está activo, si no lo están, pasaremos a la siguiente luz. Para cada luz calcularemos la MVP con la Model del objeto que contiene la luz, y la pasaremos a las uniforms, junto con las texturas del G-Buffer, y los atributos de la luz.

Dependiendo del tipo de luz, se renderiza toda la pantalla, o una zona de la escena. Si es una luz direccional o de ambiente, que afectan a toda la escena, se le pasa al shader un Quad del tamaño de la pantalla, ya que renderizar toda la escena implica realizar el cálculo de la iluminación en toda la pantalla. En cambio, si es una Luz Puntual o de Foco, solo es necesario realizar el cálculo de la iluminación en la zona que está dentro de la esfera imaginaria que engloba la luz hasta que su color es 0, por lo que se le pasa al shader la mesh de una esfera.

3.2.4. GameObject

Un GameObject es la base para cualquier tipo de objeto que exista en la escena. Por sí solo, un GameObject no es nada, no se puede situar en el espacio, no se puede mover, no tiene una representación 3D, no ilumina la escena, no lanza ni recibe ningún tipo de evento sino que los propaga. En cambio, es capaz de contener otros GameObjects, creando una jerarquía de padres e hijos, y también es capaz de contener Componentes, que dotarán al GameObject de la capacidad de realizar ciertas tareas y aportar una utilidad a la escena.

Los métodos de esta clase son básicamente, los getters y setters del conjunto de GameObjects hijos y de los Componentes, además los métodos para añadir y eliminar un GameOjbect hijo a la colección o un Componente. También se implementa método Update, que llamara a la función Update de cada GameObject hijo, y de cada Componente. Y por último el método CallMethod que recibe 2 parámetros, el nombre del método que se quiere llamar y los parámetros que recibe el método, la llamada se propagara a todos los hijos y componentes.

3.2.5. Componentes

Al igual que los GameObjects, un Componente por sí mismo, no puede desempeñar la tarea para la que está pensado, necesita de un GameObject para existir en la escena. No se ha implementado una clase padre expresamente de la que hereden los componentes, ya que solo implementaría unos pocos atributos y se pueden incluir en cada tipo de Componente.

3.2.5.1. Transform

La Transform es el Componente que dota al GameObject, de una posición en el mundo, una rotación para darle una orientación, y una escala para redimensionar el objeto. Estos datos se guardarán en una matriz de 4x4, que representa la Model del

objeto. Este componente viene añadido al GameObject por defecto, se puede eliminar de él pero no tiene mucho sentido un GameObject sin una posición en el mundo.

Los métodos que implementa son los getter y setters de la posición, rotación y escala, que transformaran la Model, el getter y setter de la Model. Además, añade los getters de los vectores Right, Top y Front, que representan 3 vectores que apuntan hacia la derecha, arriba y delante del objeto según su rotación.

Para mover, rotar y orientar el objeto hacia un punto, implementa varios métodos. Para moverlo, se implementa el método Traslate y TraslateLocal, el primero mueve el objeto según un vector XYZ en el eje de coordenadas de mundo, y la segunda en el eje de coordenadas local, si la Model está en la matriz de identidad, ambos ejes de coordenadas son el mismo. De la misma forma, se implementa Rotate y RotateLocal para rotar el objeto.

Para orientar el objeto hacia una posición en el mundo, se ha implementado el método LookAt, este método utiliza la misma función de las matrices que utilizan las cámaras, pero invirtiendo la matriz resultante, recibe como parámetros una posición en la que se sitúa el objeto, la posición hacia dónde se orienta, y un vector de referencia para definir su vector Top. Por último, como toda clase del Engine también implementa el método Update, que actualiza la Model en caso de necesitarlo.

3.2.5.2. ObjectRenderer

El Componente ObjectRenderer es el encargado de darle una representación visual al GameObject para que sea proyectada en la pantalla. Dentro del ObjectRenderer encontramos la Mesh que le da forma al GameObject. Una Mesh es una malla compuesta de polígonos, normalmente triángulos formados por vértices, la normal de cada vértice y la coordenada de textura en caso de tener una textura relacionada. En el Componente también se guarda la Textura que da color al objeto. Y por último el shader que se utiliza para renderizar el objeto.

El único método que tiene este componente es Render que se llama en Forward Rendering para renderizar el objeto según el tipo de luz que le afecta.

3.2.5.3. Light

Como su nombre indica el Componente Light, permite que un GameObject sea capaz de iluminar la escena. Los atributos que definen una luz son, primero de todo el tipo de luz, direccional, puntual, focal o ambiente. A cada luz se le define la cantidad de color ambiente, difuso o especular con el que ilumina la escena, además de la intensidad que multiplica estos valores.

Para las luces focales, se ha definido un atributo para el ángulo de apertura del foco, y su exponente que modifica la atenuación de la luz en los bordes del cono que se genera.

Tanto para la luz puntual como la focal, primero se implementó un sistema de atenuación de la luz bastante complejo, que incluye 3 factores, la atenuación constante, lineal y cuadrática, además del rango que alcanzaba la luz. Pero se decidió simplificarlo a dos factores, la atenuación Near y Far, el Near define la distancia del origen a la que empieza a atenuarse la luz generada, y el Far define la distancia del origen en la que será completamente 0. En cualquier caso, ambos sistemas se mantienen y simplemente hay que programar el shader para que utilice un tipo de atenuación u otra.

Los métodos implementados son el getter de la posición de la luz, el getter de la dirección a la que está apuntando la luz direccional o la focal, que se definirá con la función LookAt.

3.2.5.4. Camera

La Cámara es la encargada de guardar y gestionar las matrices que se pasaran a la Pipeline para proyectar las imágenes a pantalla, estas matrices son la View y la Projection. Para definir la perspectiva, la cámara tiene los atributos Field of View (FOV) que define la apertura del campo de visión de la cámara, Aspect Ratio para definir cuantos pixeles de alto por cada pixel de ancho se pintaran, Near y Far plane para definir los límites de profundidad del frustum.

Los métodos de la Cámara son el LookAt para orientar el GameObject que contiene la Cámara hacia un punto utilizando la función LookAt de la Transform, que modificara

la Model, y acto seguido que actualiza la View Matrix. El método UpdateViewMatrix actualiza la View según los nuevos valores de la Model del GameObject que contiene la Cámara, para actualizar la View ha de hacer un LookAt a la matriz, con los vectores Eye, Center y Up, que se obtienen el Eye de la posición de la Transform, el Up del Top de la Transform, y el Center, que es el punto hacia donde está orientado, lo calcularemos restando al Center el vector Front de la Transform, que nos da un punto justo delante del Center.

El método SetPerpective recibirá el FOV, Aspect Ratio, Near y Far para guardarlos en la cámara y actualizará la Projection Matrix con el método UpdatePerspective, que llamara a la función Perspective de las matrices pasándole estos 4 parámetros y guardará el resultado en la Projection Matrix.

3.2.5.5. KeyController, MouseController & RandomMovement

Si miramos en el engine Unity3D, podemos encontrar que nos permite añadir Scripts a un GameObject, una vez añadido aparece en el Inspector como un Componente más del GameObject, y su comportamiento es el mismo que el resto de Componentes, tiene los métodos Init, Setup, Update, etc. Los Componentes KeyController, MouseController y RandomMovement que hemos creado vienen a ser como un Script de Unity3D, se han programado expresamente para la Aplicación que se ha creado para poner en funcionamiento el Engine, y no forman parte de la base de este, en otra Aplicación que utilice este Engine no tienen por qué existir, al contrario de los Componentes Cámara, Transform, ObjectRenderer o Light.

El KeyController está desarrollado para capturar los eventos de teclado, y mover el GameObject al que pertenece hacia el vector Front o el Front negado, y hacia el vector Right o el Right negado. Cambiando el atributo Front y Right del componente se puede cambiar los vectores de movimiento, y también es posible cambiar la velocidad a la que se desplaza.

El MouseController captura los eventos del Mouse, para rotar el GameObject en dos ejes. En este caso está pensado para rotar en el eje X local y en el eje Y global o del mundo, de forma que siempre gire sobre la vertical del mundo independientemente de si

está mirando arriba, abajo o al frente, y no se incline lateralmente. También es posible cambiar los ejes y la velocidad de rotación.

El RandomMovement se ha implementado para dar movimiento perpetuo a cualquier objeto. Se ha pensado para añadirlo a las luces de las diferentes composiciones que se han creado como demostración en la Aplicación.

3.2.6. MicroShaderManager

Como ya se ha explicado el MicroShader Manager es una clase Singletone que provee al Engine de los Shaders que le pide. En él se guarda la colección de MicroShaders que se han extraído del fichero XML, los Shaders que se han ido generando según se iban pidiendo para no tener que crear y compilar el Shader cada vez que se pide, y el nombre del fichero de donde se extraen los MicroShaders.

Para pedirle un Shader al Manager, se llama al método GetShader pasándole un nombre que lo defina, los nombres de los MicroShaders que formen el Vertex Shader, los nombres del Fragment Shader y el fichero del cual extraerlos. Si la colección de MicroShaders está vacía, carga el fichero XML con el método LoadXML, que parsea el fichero y guarda los MicroShaders en el Manager, una vez cargados el método se llama a sí mismo para obtener el shader.

En caso de que ya estén cargados los MicroShaders, comprueba que exista un Shader guardado con el nombre identificador que le ha llegado, si existe devuelve ese Shader sin importar que corresponda a lo que se ha pedido con los nombres del Vertex y el Fragment Shader. Si no existe el Shader, compone la string de código del Vertex y el Fragment Shader con la función ComposeShader, que recibe los nombres que se quiere que formen el código y los une en una string según el orden en el que estén los nombres.

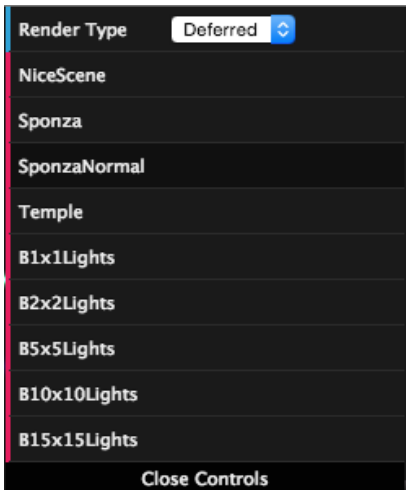
Finalmente crea un objeto Shader, con pasándole el código del Vertex y el Fragment para que lo compile, y lo guarda en la colección de Shaders y devuelve el Shader a quien lo ha pedido.

3.2.7. GUI

La clase GUI se sirve de la librería dat.GUI para crear una interfaz en la Aplicación, con la que poder añadir información y botones con los que realizar ciertas acciones que cambien la composición de la escena que se está mostrando en pantalla.

Inicialmente se mostraban todos los objetos de la escena, GameObjects con todos sus componentes, por lo que se podía controlar las luces, cámaras, etc... desde la GUI, pero finalmente se ha decidido no incluir los objetos ya que suponían una gran bajada de rendimiento en la aplicación ya que los campos que permiten modificar los parámetros de cada componente y objeto están constantemente escuchando para captar cualquier cambio en ellos, y esto era lo que generaba la gran caída de rendimiento.

Finalmente se ha decidido mostrar varios botones con los que cambiar la composición de la escena a través de las funciones de la clase Benchmark, por ejemplo mostrar la Mesh de Sponza, típica mesh utilizada en demostraciones de aplicaciones 3D, también se puede mostrar un pequeño templo de columnas, o un simple plano de NxM dimensiones con NxM luces.



GUI utilizada en el Engine

3.2.8. Benchmark

Benchmark son una serie de funciones que tienen como propósito crear una composición predeterminada de la escena con el uso de una línea de código. Esto permite tener el código de la aplicación un poco más limpio y más entendible.

El nombre viene dado por la técnica utilizada para medir el rendimiento de un sistema, lo cual no difiere de la intención de este grupo de funciones entre las que nos encontramos la función BenchmarkLights que nos permite crear un plano de tamaño NxM, con NxM luces sobre él, lo cual es un claro ejemplo de test de rendimiento sobre el número de luces que es capaz de mover el Engine. También existe la función BenchmarkLightsObjects, que crea NxM meshes del tipo que le pasemos de tamaño 1 y NxM luces, otro ejemplo que en este caso prueba el rendimiento sobre el número de luces y objetos que puede mover el Engine.

Luego tenemos las funciones Sponza y Temple que muestran la mesh del Palacio de Sponza (Croacia) y un templo de columnas respectivamente, con cierto volumen de luces. El objetivo de estas funciones es ver el comportamiento de los cálculos de iluminación sobre escenas con objetos realistas.

3.3. Funcionamiento de la App

Como demostración del uso del engine desarrollado, se ha creado una App que permite al usuario seleccionar entre diferentes escenas compuestas previamente. El primer elemento que nos encontramos en la interfaz gráfica de usuario, mostrada por la clase GUI, que se muestra en la esquina superior derecha del navegador, es un selector con el título de Render Type, aquí podremos seleccionar ver en la pantalla el render realizado con Forward Rendering, o por lo contrario verlo con Deferred Shading, mostrando o no el G-Buffer.

Para la selección de una de las escenas, simplemente pulsando, con el mouse, en uno de los botones de la interfaz gráfica de usuario. Esta acción, que se podrá repetir en cualquier momento, se encargara de llamar a la función correspondiente de la clase Benchmark.

3.3.1. Composición de una escena

La composición de una escena, está formada de un conjunto de GameObjects. Cada uno de estos GameObjects tendrá sus propios Componentes que harán que el GameObject pueda desempeñar la tarea para la que ha sido creado.

Dados los Componentes que se han desarrollado en este proyecto, se pueden crear objetos que pueden desempeñar 3 tareas diferentes:

- El primero de todos, el más importante y obligatoriamente ha de existir 1 en la escena, puede haber más de una, son los objetos que harán de Cámara, estará formado por un GameObject con al menos un componente Camera, si se quiere poder mover y rotar la cámara, ha de tener los Componentes KeyController y MouseController. Al crear la Cámara se le da una posición y orientación, y los valores para definir la perspectiva.
- El segundo, son los objetos con una representación visual en la escena, si no hay nada que se pueda ver en la escena, no tiene sentido generar una escena. Estará formado por un GameObject, puede tener otros GameObjects como nodos hijo, tendrá al menos un Componente ObjectRenderer, donde le daremos una Mesh que representara su forma, sin ella seguiríamos sin ver nada en la escena, y podremos asignarle una Textura que dará color a la Mesh. Al igual que con la cámara, si queremos poder controlar el movimiento de estos objetos, que pueden representar al jugador en un juego, se deberá asignarle un KeyController y/o un MouseController. Ademas de la Mesh y la Textura, se le da una posición en el mundo, aunque por defecto aparecerá en el origen (0,0,0).
- Por último, las Luces, aunque no son estrictamente necesarias, ya que un shader puede pintar la superficie de la Mesh sin necesidad de un punto de luz. El caso más realista es, que una cámara puede grabar un objeto de cualquier forma, pero si el objeto no tiene una fuente de luz que permita a la cámara ver el color de su superficie, simplemente estará grabando una imagen en negro. Las Luces, serán un GameObject con un Componente Light. Al crear una Luz, se tendrá que definir su posición, para las luces puntuales y de foco, y dirección, para la direccional y la de foco, también se le podrá asignar las componentes de color

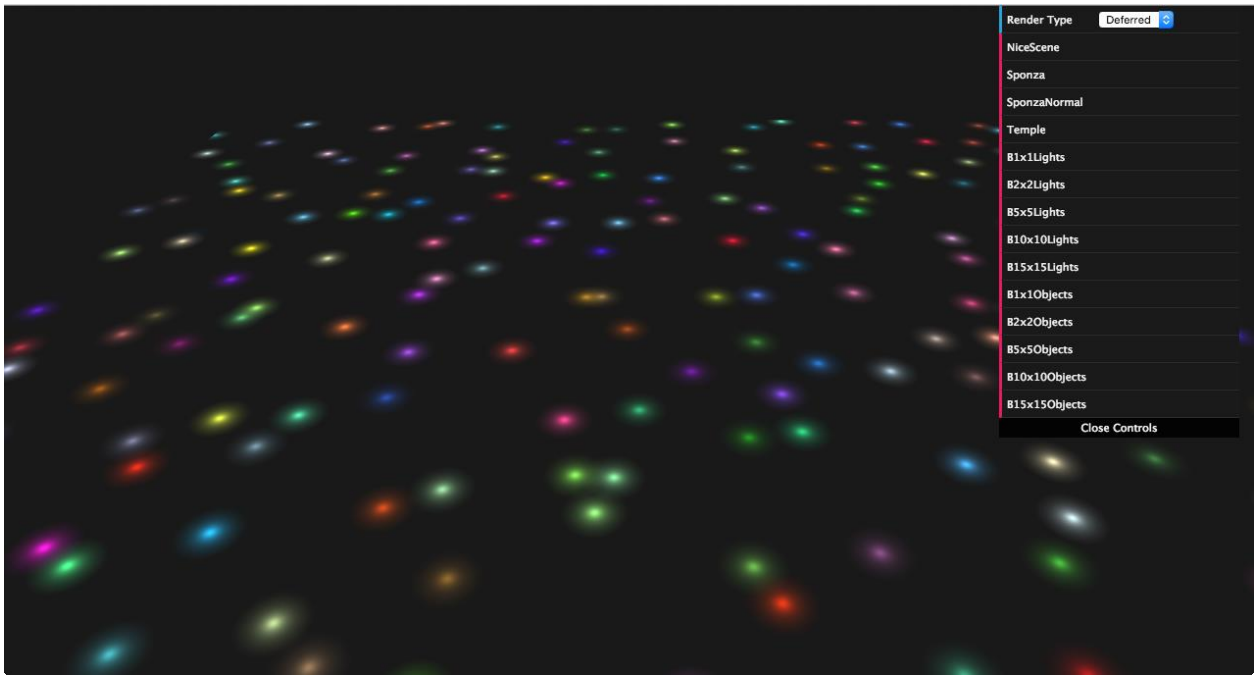
ambiente, especular y difusa, y el Near y el Far para las luces puntuales y de foco.

Con estos elementos y un poco de imaginación se pueden generar escenas todo lo complejas que el usuario desee.

4. RESULTADOS

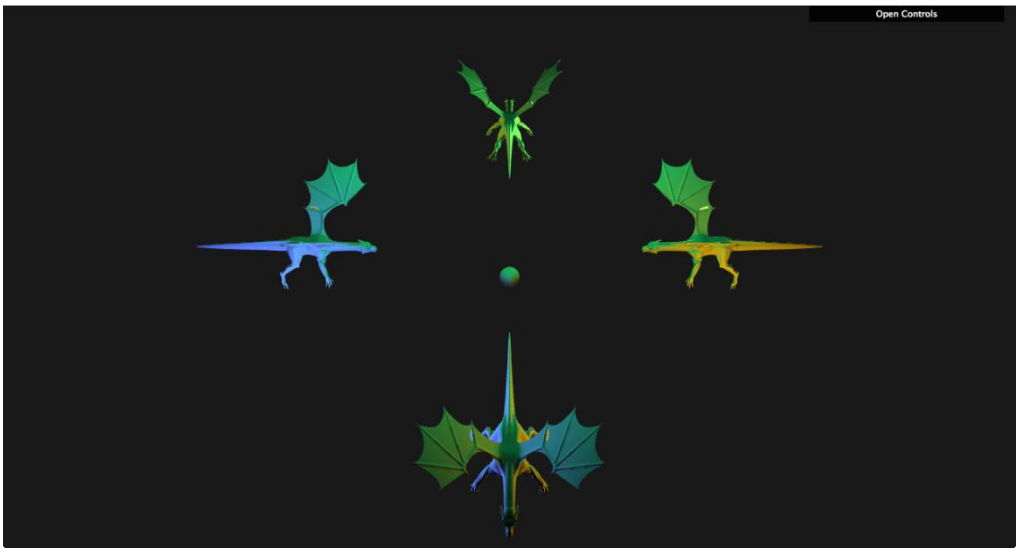
4.1. Ejemplos en funcionamiento

A continuación se pueden ver algunas capturas de imagen de los ejemplos que se pueden seleccionar en la GUI. Aunque lo ideal sería poder ver el engine en funcionamiento junto con este documento.



225 puntos de luz sobre un plano.

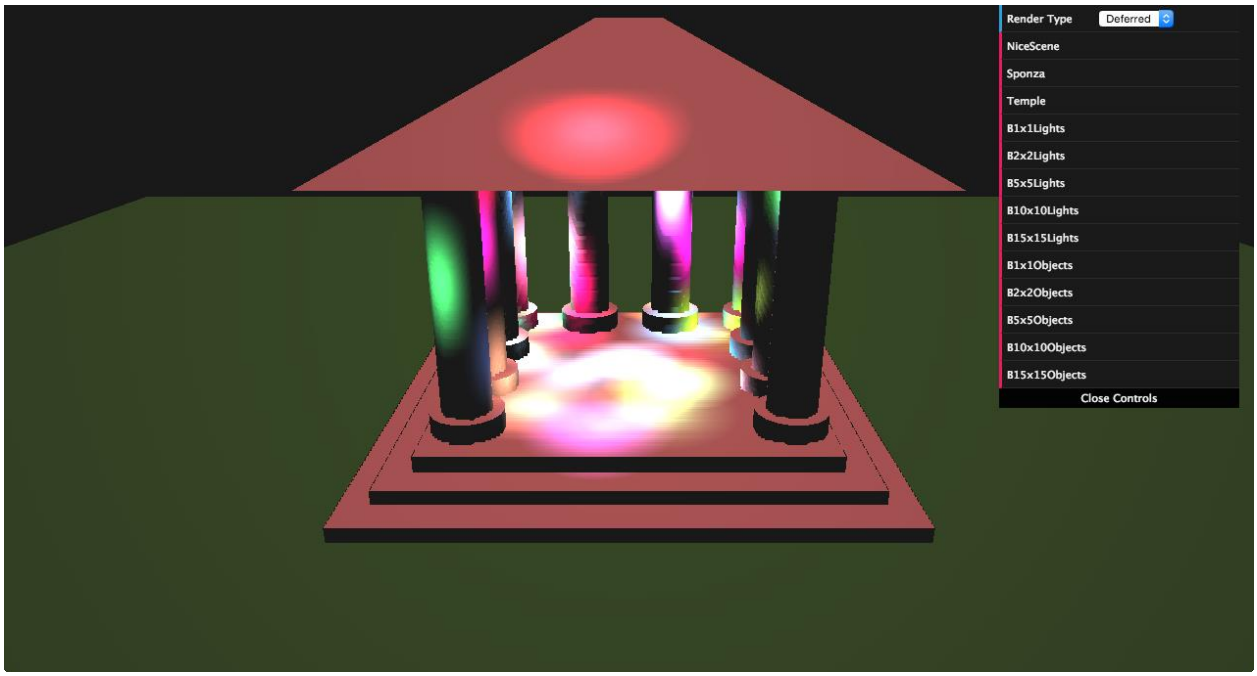
En esta primera imagen, vemos uno de los juegos de pruebas en el que se ven 225 luces moviéndose sobre un plano.



4 dragones iluminados por luces puntuales

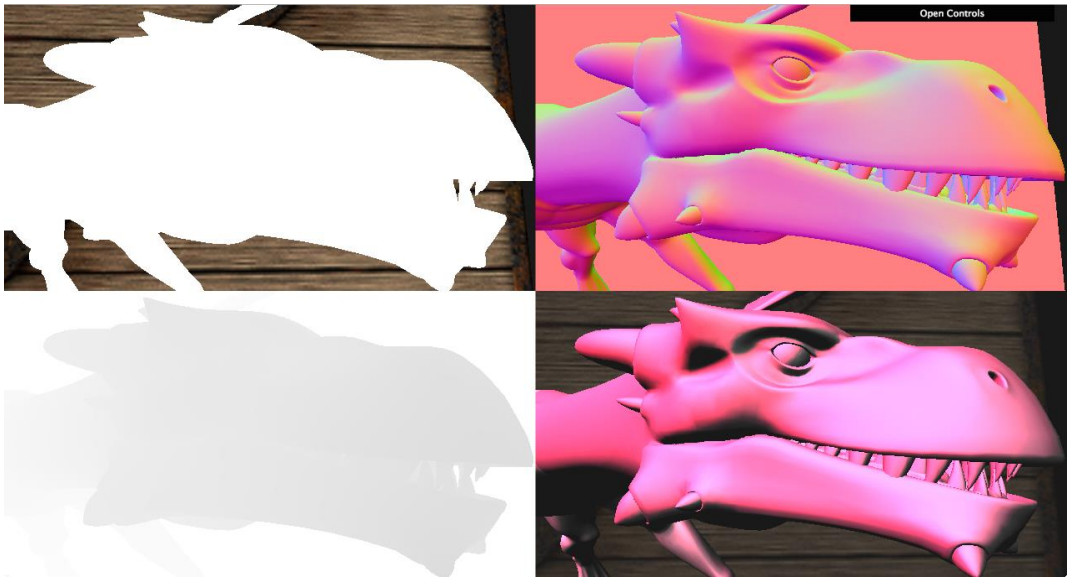
Mesh Sponza sin normales computadas, iluminada por 50 puntos de luz.

En esta imagen, vemos la mesh de Sponza, que al descargarla viene sin las normales, con 50 luces repartidas por toda la escena.



Mesh Temple, iluminada por 50 puntos de luz.

Esta es la mesh del templo de columnas, sobre un plano de color verde, para ver las luces que al moverse de forma aleatoria, se iban "escapando" del templo. Toda la escena esta iluminada con una luz direccional de color rojizo, por eso el templo se ve de ese color.



G-Buffer generado con el Engine

Color Difuso (Arriba izquierda) Normales (Arriba derecha)

Depth Buffer (Abajo izquierda) Imagen Final (Abajo derecha)

En esta imagen, podemos ver el G-Buffer y la imagen de Deferred Shading, generada por el Engine, sobre el detalle de la cabeza del Dragón en la escena NiceScene

de la GUI. Se puede ver el buffer de color difuso y el de las normales arriba, y abajo el Depth Buffer, y la imagen final con la iluminación, la cabeza del Dragón se parece a las normales, porque casualmente la luz que la ilumina es de color rojizo.

4.2. Rendimiento

Tratándose de un engine de renderizado de gráficos 3D en tiempo real, la comparación de rendimiento se puede hacer a simple vista, viendo la fluidez de movimiento de la imagen, pero también se puede cuantificar según el número de frames renderizados cada segundo, y podemos comparar los resultados de estas mediciones entre los dos tipos de render que se han implementado en el engine.

4.2.1. Frames Per Second

Los Frames Per Second (FPS) es la medida de la frecuencia a la cual un reproductor de imágenes reproduce distintos frames. El sistema visual humano puede procesar de 10 a 12 imágenes separadas por segundo, recibiéndolas de forma individual. El umbral de la percepción visual humana varía dependiendo de lo que se está midiendo. El córtex visual mantiene una imagen alrededor una quinceava parte de segundo (15 FPS), por lo que si recibe otra imagen durante ese periodo de tiempo, se crea la ilusión de continuidad, permitiendo a una secuencia de imágenes fijas dar la sensación de movimiento.

4.2.2. Juegos de pruebas

Para comprobar el rendimiento se han realizado varios juegos de pruebas con diferentes escenarios.

NxN planos + MxM Luces			
N	M	Forward Rendering	Deferred Shading
1	1	60 FPS	25-30 FPS
1	2	60 FPS	25-29 FPS
1	5	60 FPS	23-27 FPS
1	10	17-20 FPS	20-24 FPS
1	15	4-8 FPS	19-20 FPS
2	2	60 FPS	29-33 FPS
5	5	26-29 FPS	25-30 FPS
10	10		16-20 FPS
15	15		11-13 FPS

Sponza + N Luces		
N	Forward Rendering	Deferred Shading
10	9-10 FPS	23-28 FPS
20	6-8 FPS	18-21 FPS
50		15-18 FPS
100		9-10 FPS
200		6-7 FPS
500		

Temple + N Luces		
N	Forward Rendering	Deferred Shading
10	23-24 FPS	30-35 FPS
20	13-16 FPS	29-33 FPS
50	7-11 FPS	27-30 FPS

100	4-8 FPS	23-27 FPS
200		19-22 FPS
500		11-14 FPS

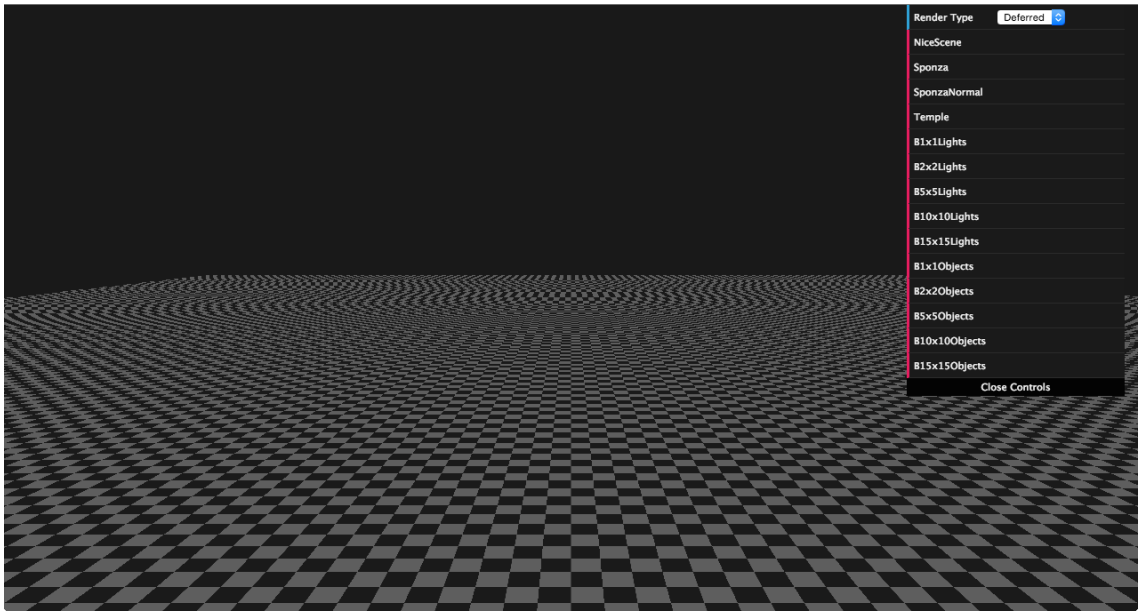
Viendo los resultados obtenidos, se puede observar varios datos. Forward Rendering tiene mejor framerate cuando se trata de pocos objetos o con una geometría sencilla y pocas luces en la escena, esto lo podemos ver cuando tenemos 1 plano con 1 luz, 2x2 luces y 5x5 luces, y 2x2 planos con 2x2 luces, en estos casos Forward va a 60 FPS, pero a partir de ahí, las mediciones de 10x10 y 15x15 luces hacen que el framerate caiga en picado en Forward. Deferred en cambio, no consigue tener un framerate tan alto, pero su curva de caída es más lenta, se puede suponer que la razón de que no tenga unos FPS tan altos es por el coste de generar el G-Buffer.

Cuando utilizamos una mesh más compleja que un plano, Forward con pocas luces ya empieza perdiendo. En el templo de columnas con 10 luces está entre los 23 y 24 FPS, mientras que Deferred entre 30 y 35. En el momento que sube el número de luces Forward cae en picado, y Deferred baja un poco pero entra dentro de los FPS aceptables para dar sensación de continuidad en la imagen, en concreto Deferred no baja de los 15 FPS hasta las 500 luces que está entre 11 y 14 FPS.

En el caso de la mesh de Sponza en ningún caso ha pasado de los 15 FPS, y Deferred a partir de las 100 está por debajo, incluso con 500 luces el contador de FPS del navegador no era capaz de actualizar su valor, como en varios casos de Forward durante todo el juego de pruebas.

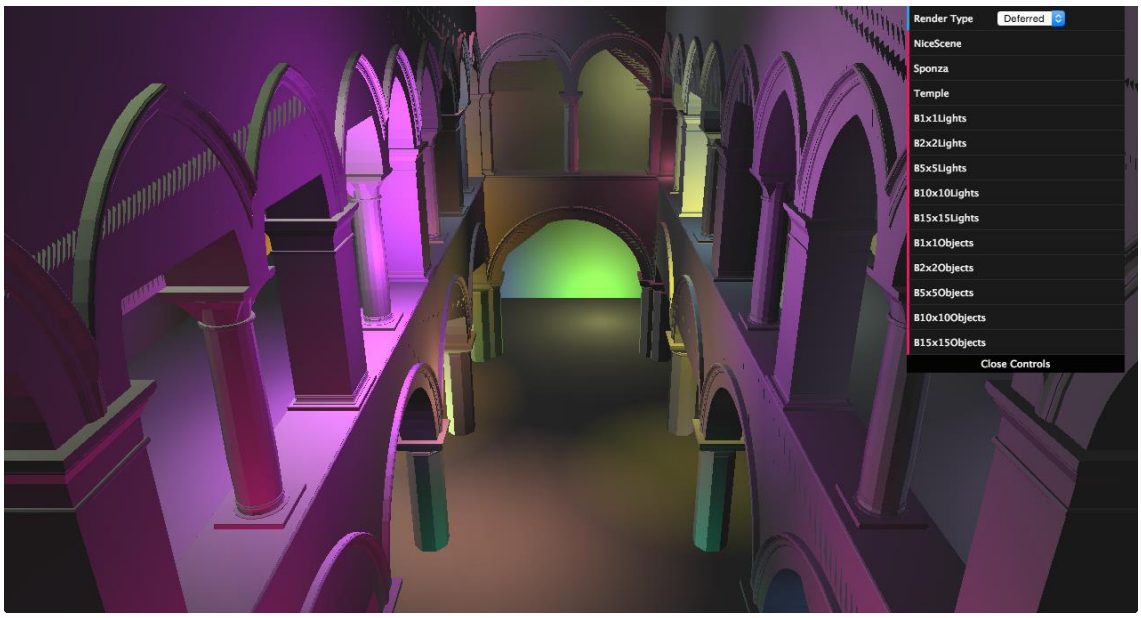
4.3. Errores

En las siguientes imágenes veremos algunos errores en el renderizado, que han quedado reflejados en pantalla.



Plano con textura de Ajedrez, Aliasing en el horizonte

En la imagen superior, podemos ver el típico caso de Aliasing, al aplicar una textura de cuadros, sobre un plano que va hacia el horizonte.



Sponza con normales computadas

En la Mesh de Sponza al venir sin normales en su fichero, han de ser computadas por el engine. Si nos fijamos, en ciertas zonas, se ven artefactos que no deberían estar, por ejemplo, a la izquierda de la imagen, en el arco más próximo de la cámara, vemos

una hilera de líneas diagonales pintadas de un color verdoso, otro ejemplo, a la derecha, tocando la GUI, sobre el segundo arco, se ven unos rombos negros. Estos dos casos son debido a que la computación de las normales no es perfecta, pero es lo suficientemente buena como para mostrar detalles de la escena, como los adornos de las columnas, que no se aprecian en la imagen de Sponza sin las normales, del apartado 3.3.2.

5. Conclusiones

Deferred es capaz de renderizar escenas con un gran número de luces al contrario que Forward, y queda comprobado con los juegos de pruebas realizados, además de mantener un framerate bastante aceptable constantemente. También se puede observar que la generación del G-Buffer supone un coste elevado, pero es un coste aceptable con tal de conseguir un buen nivel de FPS con una gran cantidad de luces en la escena.

5.1. Mejoras posibles

Con el proyecto dado por concluido, se pueden plantear ciertas mejoras de implementación, tanto por la parte que engloba al Engine, como por la tarde de implementación de las técnicas de Rendering.

Por parte del Engine en general, para acercarse al comportamiento de los engines comerciales le haría falta poder crear la composición de la escena (añadir GameObjects a la escena, añadir componentes a los GameObjects) en tiempo de ejecución, y ser capaz de guardar la composición para luego cargarla de un fichero. Esto lleva a que sea necesaria la posibilidad de gestionar la Transform de los objetos con el mouse, mediante ray picking. Evidentemente le faltan muchas características al Engine para acercarse a uno comercial que mejoren su funcionalidad, como por ejemplo componentes de sistemas de partículas, motor de físicas, animación de meshes, por nombrar algunas, pero el propósito de este Engine era que permitiese implementar las diferentes técnicas de Rendering en un entorno de trabajo controlado, por lo que no son necesarias por ahora.

Por la parte del Render de la escena, algo que se puso como objetivo personal, al inicio del desarrollo, pero que por no ser estrictamente necesario para el objetivo del proyecto, no se ha implementado finalmente es, la inclusión de sombras. La sombra proyectada en un objeto de otro que está entre él y la fuente de luz, añadiría gran realismo a la escena. Otra posible mejora es el uso de materiales en los objetos, y un mejor uso del G-Buffer en Deferred.

Como extra, la adaptación del Engine a navegadores en dispositivos móviles sería una mejora importante.

5.2. Trabajo futuro

Como objetivos futuros me propondría implementar sobre el Engine, las características mencionadas en el apartado de mejoras, empezando por el uso de sombras. Otro de los objetivos de carácter personal, es la implementación de un sistema de animaciones de meshes. A partir de ahí, una vez cubiertos los objetivos personales, se podrían implementar el resto de técnicas de Deferred comentadas en el apartado 2.2.3.3. y realizar pruebas comparándolas entre ellas, con el objetivo de ampliar conocimientos viendo resultados reales.

Bibliografía

Google Chrome team, 2011 & 2012 versions by Hyperakt and Vizzuality, 2010 version by mgmt design and GOOD. The Evolution of the Web.
http://www.evolutionoftheweb.com/

División de Arquitectura de Computadores de la Universidad Rey Juan Carlos, Informática Gráfica, CURSO 2012/13, Modelos de iluminación
http://dac.escet.urjc.es/docencia/IG/08-IluminacionSombreado4.pdf

Síntesis de Imagen y Animaci´on Institute of New Imaging Technologies, Universitat Jaume I, J. Ribelles, 2013, Modelos de iluminación y sombreado
http://ocw.uji.es/material/94021/raw

Khronos Group, WebGL Wiki, 2013, Getting Started
https://www.khronos.org/webgl/wiki/Getting_Started

Wikipedia, WebGL
http://en.wikipedia.org/wiki/WebGL

Tuts+, GameDevelopment, Brent Owens, 2013, Forward Rendering vs. Deferred Rendering
http://gamedevelopment.tutsplus.com/articles/forward-rendering-vs-deferred-rendering--gamedev-12342

GPU Gems 3, Chapter 19. Deferred Shading in Tabula Rasa, Rusty Koonce, NCsoft Corporation
http://http.developer.nvidia.com/GPUGems3/gpugems3_ch19.html

Wikipedia, Deferred Shading
http://en.wikipedia.org/wiki/Deferred_shading

Unity3D Documentation, Rendering Pipeline Details – Forward Rendering Path Details
http://docs.unity3d.com/Manual/RenderTech-ForwardRendering.html

Unity3D Documentation, Rendering Pipeline Details – Legacy Deferred Lighting Rendering Path
http://docs.unity3d.com/Manual/RenderTech-DeferredLighting.html

Utrecht University, Marries van de Hoef, Bas Zalmstra, 2010, Comparision of multiple rendering techniques
http://www.marries.nl/wp-content/uploads/2011/02/Comparison-of-multiple-rendering-techniques-by-Marries-van-de-Hoef-and-Bas-Zalmstra.pdf

Chalmers University of Technology, Ola Olsson and Ulf Assarsson, 2011, Tiled Shading - Preprint
http://www.cse.chalmers.se/~olaolss/get_file.php?filename=papers/tiled_shading_preprint.pdf

Takahiro Harada, Jay McKee, Jason C.Yang, Forward+: Bringing Deferred Lighting to the Next Level, Eurographics Short Paper, 5-8 (2012)
https://sites.google.com/site/takahiroharada/storage/Forward%2B.pdf?attredirects=0