Escuela Politécnica Superior de Jaén
Grado en Ingeniería Informática

Universidad de Jaén
Escuela Politécnica Superior
de Jaén

TRABAJO FIN DE GRADO

SISTEMA DE RENDERING 3D DIFERIDO

Alumno

Alejandro Gemas Toribio

Tutor

Carlos Javier Ogayar Anguita
(Departamento de Informática)

Septiembre, 2020

(Página intencionalmente en blanco)

Universidad de Jaén

Departamento de Informática

Don Carlos Javier Ogayar Anguita, tutor del Trabajo Fin de Grado titulado: '**Sistema de rendering 3D diferido**', que presenta Don Alejandro Gemas Toribio, otorga el visto bueno para su entrega y defensa en la Escuela Politécnica Superior de Jaén.

Jaén, a 7 de septiembre de 2020

El alumno:

Alejandro Gemas Toribio

El tutor:

OGAYAR ANGUITA CARLOS JAVIER - 26039170L

Firmado digitalmente por OGAYAR ANGUITA CARLOS JAVIER - 26039170L
Nombre de reconocimiento (DN): c=ES, serialNumber=IDCES-26039170L, givenName=CARLOS JAVIER, sn=OGAYAR ANGUITA, cn=OGAYAR ANGUITA CARLOS JAVIER - 26039170L
Fecha: 2020.09.07 19:51:36 +02'00'

Carlos Javier Ogayar Anguita

(Página intencionalmente en blanco)

Agradecimientos

A mi madre, porque nunca dejó de creer en mí y me apoyó hasta cuando nadie lo hacía. A mi hermano, por dame esa alegría por la casa que animaba a cualquiera. A mi tía, a la que estaré eternamente agradecido por ese ordenador con Windows 95 que me regalaste; sin eso yo no hubiese acabado aquí. A Marta, porque sin ti no hubiese tenido las fuerzas suficientes para conseguir todo lo que me he propuesto; gracias por todo. A Raúl, que eres un grandísimo amigo y aún más como compañero; cada día aprendrí mucho de ti. A todos y cada uno de mis compañeros de la carrera que me hicisteis la vida un poco más sencilla, sobretodo todo tú Rubén.

Y por último, a mi tutor Carlos Ogayar por ayudarme en todo lo posible y por esa increíble paciencia durante el proceso.

Gracias por todo.

FICHA DEL TRABAJO FIN DE TÍTULO	
Titulación	Grado en Ingeniería Informática
Modalidad	Trabajo Teórico/Experimental
Especialidad (solo TFG)	Tecnologías de la Información
Mención (solo TFG)	Sistemas Gráficos
Idioma	Español
Tipo	Específico
TFT en equipo	No
Autor/a	Alejandro Gemas Toribio
Fecha de asignación	03/04/2020
Descripción corta	Este trabajo consiste en la realización de un estudio teórico-práctico acerca de la técnica de rendering 3D diferido (deferred rendering), que por lo general permite mayor rendimiento que el enfoque tradicional (forward rendering) con escenas que presentan una iluminación muy compleja. Debe hacerse un estudio teórico sobre la citada técnica. Posteriormente debe realizarse un prototipo que utilice este método de rendering, basado en OpenGL, Vulkan, Metal u otra tecnología similar. El software debe permitir representar escenas 3D geométricamente complejas, con una gran cantidad de luces y en tiempo real.

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

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

Contenido

Índice de ilustraciones

Índice de tablas

1 INTRODUCCIÓN

Escoger qué algoritmo de iluminación usar es crucial para el desarrollo del mismo; puesto que afectarán a la duración del proyecto y a la complejidad a la hora de añadir nuevas características. Deferred rendering es una técnica que nos permite ahorrar ciclos de GPU de una manera eficiente, pero no es la única.

A lo largo de este trabajo, hablaremos sobre los distintos tipos de algoritmos que existen, enfocándonos en Deferred Rendering. Los analizaremos, optimizaremos y compararemos con diferentes técnicas en diversos escenarios donde pondremos a prueba sus características y deficiencias. Además, los acompañaremos de técnicas muy usadas como el “Bump Mapping” o efectos postprocesado como la corrección del gamma o High Dynamic Range (HDR).

1.1 Objetivos del trabajo

Los objetivos del trabajo son:

- Analizar la técnica “Deferred Rendering” y observar sus beneficios.
- Analizar de la ténica “Forward Rendering” y “Forward Plus Rendering”, para poder comparar los resultados con “Deferred Rendering”.
- Estudiar en qué casos tienen dificultades para someterlos a pruebas para ver cómo afectan los rendimientos en ellos.
- Investigar cómo adaptar la técnica del “Normal Mapping” a cada uno de ellos.
- Estudiar en qué consisten los efectos postprocesado usados en este estudio.

1.2 Descripción de la situación de partida

1.2.1 Descripción del entorno actual

Cuando se desea realizar una aplicación gráfica desde cero, la mayoría de ocasiones se parte de la misma técnica: Forward Rendering. Esta técnica es la que se enseñan en las asignaturas, ya que no pueden permitirse poder enseñar otra por problemas de tiempo y horarios. Esto provoca que, si alguien quiere hacer un entorno gráfico, se encontrará con un sistema que tendrá muchas limitaciones a la hora de rendimiento y a la hora de implementar nuevas técnicas.

1.2.2 Resumen de las deficiencias y carencias identificadas

Las deficiencias que nos podemos encontrar, si recurrimos a Forward Rendering, es que no podremos usar muchas luces a la hora de iluminar nuestra escena. Esto es, porque el algoritmo debe recorrer cada objeto que haya en la escena por cada luz que se desee usar. Esto provoca una caída de rendimiento considerable en nuestro sistema 3D si, además, se le añaden cálculos de luces más complejos. Esto implica que, si se desea añadir otras técnicas que hagan más realista nuestro renderizado, nos encontremos con un grave problema a la hora de incorporarlos.

Este trabajo tiene como objetivo analizar y demostrar que existen más técnicas de renderizado, sin incrementar la dificultad del desarrollo. Se verá qué inconvenientes nos podemos encontrar con todas ellas y cuáles son las mejores para nuestras necesidades.

1.3 Estructura del documento

La memoria está estructura de la siguiente forma:

- En este primer capítulo se introduce al tema que se va a desarrollar durante este trabajo. Además, se pondrán las bases teóricas de las técnicas que se van a desarrollar en el apartado 1.4 “Situación actual de la tecnología”.

- En el capítulo dos, se enfoca en todo lo relacionado sobre el análisis del proyecto. En él, se especifican cosas como qué metodologia se ha usado, planificaciones, presupuestos y primeros requisitos encontrados.
- En el tercer capítulo, se entrará en más detalle sobre el diseño del mismo. Ahí se especifican de forma más concreta los requisitos y diagramas usados, los de la aplicación o las implementaciones de los renders que se van a desarrollar.
- En el cuarto capítulo entramos en detalles del desarrollo, exponiendo cada una de las fases que ha habido según la metodología que se ha especificado en el capítulo dos.
- Habrá un quinto capítulo donde se pone a prueba el proyecto en una serie de escenas y se sacarán conclusiones del mismo.
- Y un sexto capítulo donde se expondrán unas conclusiones finales y futuros trabajos a desarrollar.

1.4 Situación actual de la tecnología

En esta sección, se hablará de cuáles son las técnicas más comunes que se usan en un sistema gráfico y cuáles son sus inconvenientes. Además, abordaremos temas como qué alternativas hay al Deferred Rendering o sobre efectos de postprocesamiento.

1.4.1 Forward rendering

Forward rendering es un algoritmo de renderizado muy usado dentro del mercado, dado que su sencillez es un punto a favor para ahorrar tiempo y complejidad a la hora del desarrollo.

Su funcionamiento consta de una única fase: para cada objeto que haya en la escena, realizamos los cálculos de iluminación de cada una de las luces que estén en ella. Para renderizarla, se debe realizar una serie de pasos al que se le denomina como “pipeline rendering”. El pipeline rendering son las etapas por la que debe pasar cualquier geometría para ser dibujada en OpenGL. Para que la gráfica sepa cómo ha

de hacerlo, OpenGL recurre a los denominados “shaders”. Un shader, es un programa que envía instrucciones a la gráfica especificándole cómo ha de ejecutar esa etapa. Así, podremos tener diferentes shaders para diferentes tratamientos de una misma etapa.

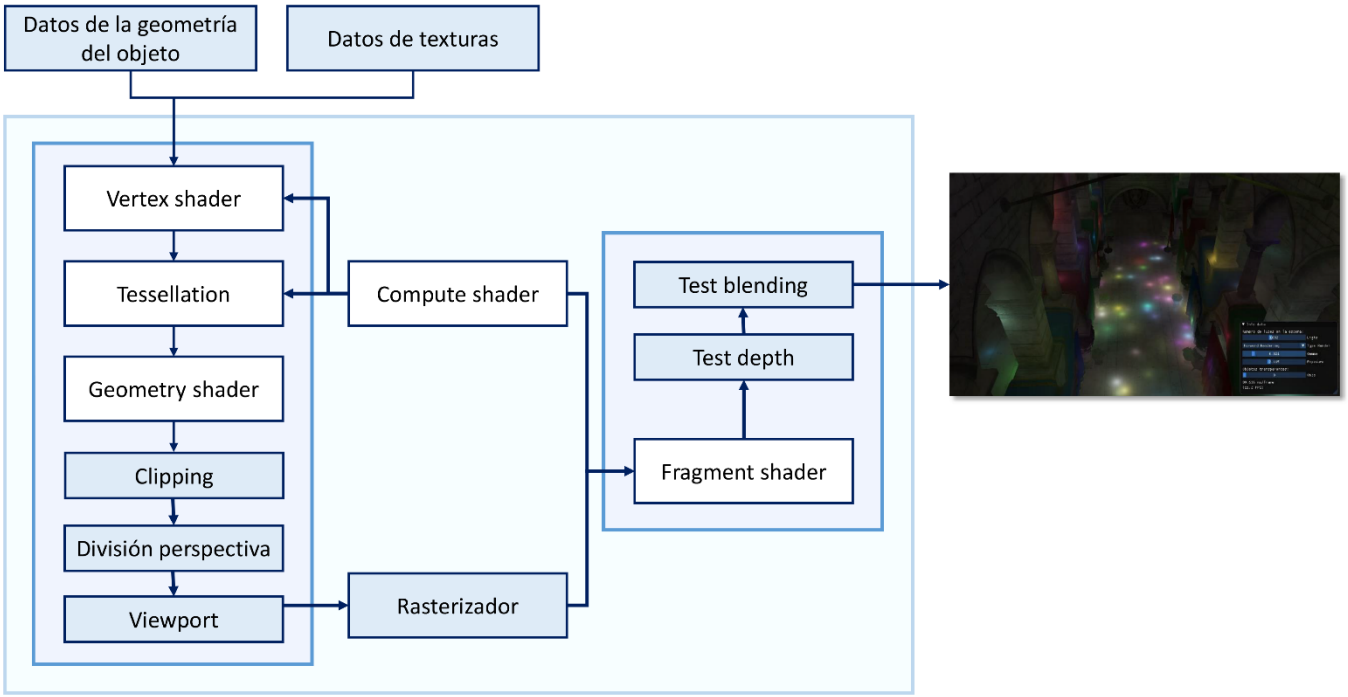


Ilustración 1.1 Pipeline rendering en OpenGL

Las etapas del pipeline son:

- **Transformación de modelado (Vertex Shader)**. En esta etapa, situa el objeto en la posición especificada en la escena.
- **Transformación de visión (Vertex Shader)**. En esta parte, se transforma todo a coordenadas de visión. Esto es, todo según el punto de vista de la cámara. Aquí, las coordenadas son relativas a las coordenadas de visión, no como en la etapa anterior que todo estaba en coordenadas de mundo.
- **Transformación de perspectiva (Vertex Shader).** Para realizar ciertos calculos como qué partes de la escena se quedan fuera de la ventana de visón de la cámara, es mejor hacerlo con un volumen pequeño (no es lo mismo hacerlo sobre un rango [0,1] a otro de [0,250] en cada componente). Por todo eso, en esta parte se transforma todo a un cubo canónico (1x1x1).

- **Tessellation.** En esta etapa, se busca subdividir los vertices en otras geometrías de losn vertices que nos llegan de la etapa anterior. Es poco común verlo.
- **Geometry Shader.** Es una etapa opcional entre los vertex shaders y los fragment shaders. Recibe el conjunto de vertices antes de ser mandados a la etapa de sombreado. Aquí, podemos realizar calculos sobre los vertices.
- **Clippin, división de perspectiva y viewport.** Un vez que sabemos qué queda fuera y qué queda dentro, se pasa a preparar la ventana final o viewport. En esta etapa, se transforma el volumen canónico de la etapa anterior, a una con las dimensiones específicas en el protecto. Por ejemplo, a 1920x1080.
- **Rasterización.** Todas las etapas anteriores, estabamos trabajando sobre los objetos en 3D. En esta etapa, se transforma todo a píxeles. Para ello, se recorrer cada triangulo de la escena y calcula qué pixeles le corresponde a la ventana final o viewport. Aquí también se realizan los calculos de, por ejemplo, colocar las texturas finales con sus coordenadas o colocar el color que le correspondiera.
- **Compute Shader.** Esta etapa se podría decir que no entra dentro de las etapas dibujado. Su objeto es realizar calculos complejos aprovenchado la pontencia que tiene la GPU.

  Sombreado (Fragment Shader). En esta etapa, que se explicará más adelante, es la encargada de calcular el color final del pixel correspondiente. Es decir, se encarga de calcular la iluminación que tiene ese pixel y lo oscurece o lo ilumina más según las luces en la escena.
- **Test de profundidad.** Con el test de profundidad, vemos cómo de lejos están los objetos de la cámara. Puede pasar que, un objeto que esté delante de otro, se procese primero y se quede detrás de éste. En esta etapa se rectifica justo eso, es decir, coloca los objetos en el orden correcto según su profundidad.

La mayoría de las etapas mencionada, las realiza automáticamente OpenGL. Solamente habrá que crear, mediante shaders, dos de ellas: los vertex shaders y los fragments shaders. Dentro de la etapa de los vertex shaders (Transformación de modelado, visión y perspectiva), especificaremos como tratar con los vértices de los objetos. Además, nos llega información útil como las tangentes y bitangentes, las coordenadas de textura o las matrices de visión y de proyección para situar el objeto en su respectiva posición. Durante la etapa de los fragments shaders (etapa de sombreado), especificaremos cómo se iluminará cada pixel, es decir, aquí estarán todos los cálculos de iluminación. En esta etapa, se recorre cada fragmento que compone la ventana final de la aplicación. Cada pixel de esa ventana representa un fragmento. En la ilustración 1.2, podemos ver como se recorre cada objeto denominado como “Geometry” y pasa por sus diferentes etapas.

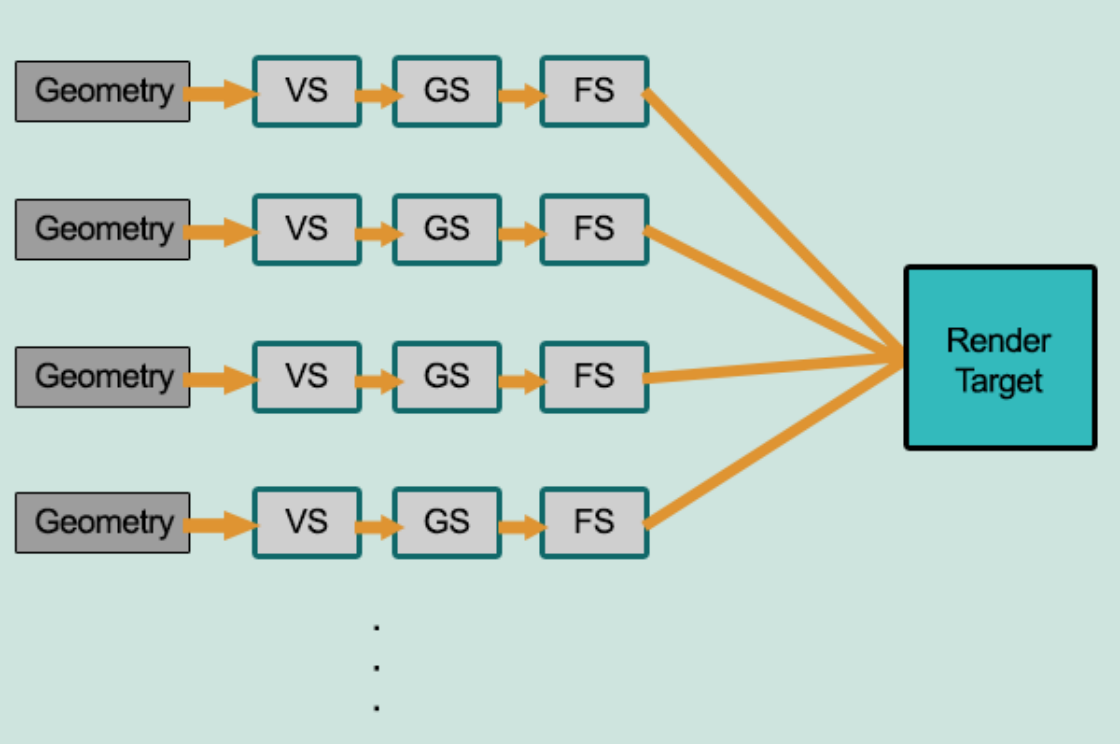


Ilustración 1.2 Ciclo de dibujado en Forward Rendering

Claramente podemos observar que trae consigo grandes limitaciones. Primero, dependemos del número de objetos que tengamos en nuestra escena. Al aumentar este número, aumentará el número de llamadas para calcular con los shaders,

haciendo que traiga consigo el segundo problema: las veces que nos recorremos los fragmentos o pixeles.

Dado que debemos llamar a cada objeto para su dibujado, éste a su vez, deberá llamar a sus respectivos shaders (con sus pasadas en los vértices y en los fragmentos). Entonces, si tenemos 'x' número de elementos, nos recorreremos 'x' veces los mismos fragmentos. Por ejemplo, supongamos que tenemos 20 objetos en una escena, con una resolución de 1080 pixeles. Por tanto, 1.080 * 1.920 = 2.073.600 pixeles/fragmentos que debemos calcular. Esos pixeles son solo para un único objeto, ahora debemos multiplicarlo por el número de objetos que haya en pantalla. El resultado es tan grande, que podemos hacernos una idea de hasta qué punto es limitante usar Forward Rendering. Si queremos hablar en términos de complejidad, estamos ante un *O(Número de objetos * Número de fragmentos * Número de luces)*, una cifra considerablemente significativa.

Para este trabajo, solo se ha enfocado en luces puntuales que se mueven a lo largo de la escena. Al solo tratarse de este tipo de luces, se puede realizar una pequeña mejora de rendimiento en los fragment shaders. Una luz, de forma usual, tiene ciertas propiedades como pueden ser su posición en el mundo, su intensidad o su color. Pero también podemos añadir un cuarto componente que nos indica el radio de actuación de la luz. Así, si un fragmento está más lejano que la distancia de actuación de dicho radio, podemos descartarlo.

1.4.2 Deferred Rendering

Viendo lo poco eficiente que es Forward Rendering, surge la técnica llamada Deferred Rendering por Michael Deering (Deering, 1988).

Deferred Rendering cambia por completo el diseño visto hasta ahora. Plantea que realmente no necesitamos realizar una pasada de luces por cada objeto que haya en pantalla, obligando a calcular todos los pixeles que tenemos en ella; sino que, con una única pasada, podemos iluminar con tantas luces como nosotros queramos. En la ilustración 1.3, podemos observar un resultado exagerado de lo que se puede obtener con él.

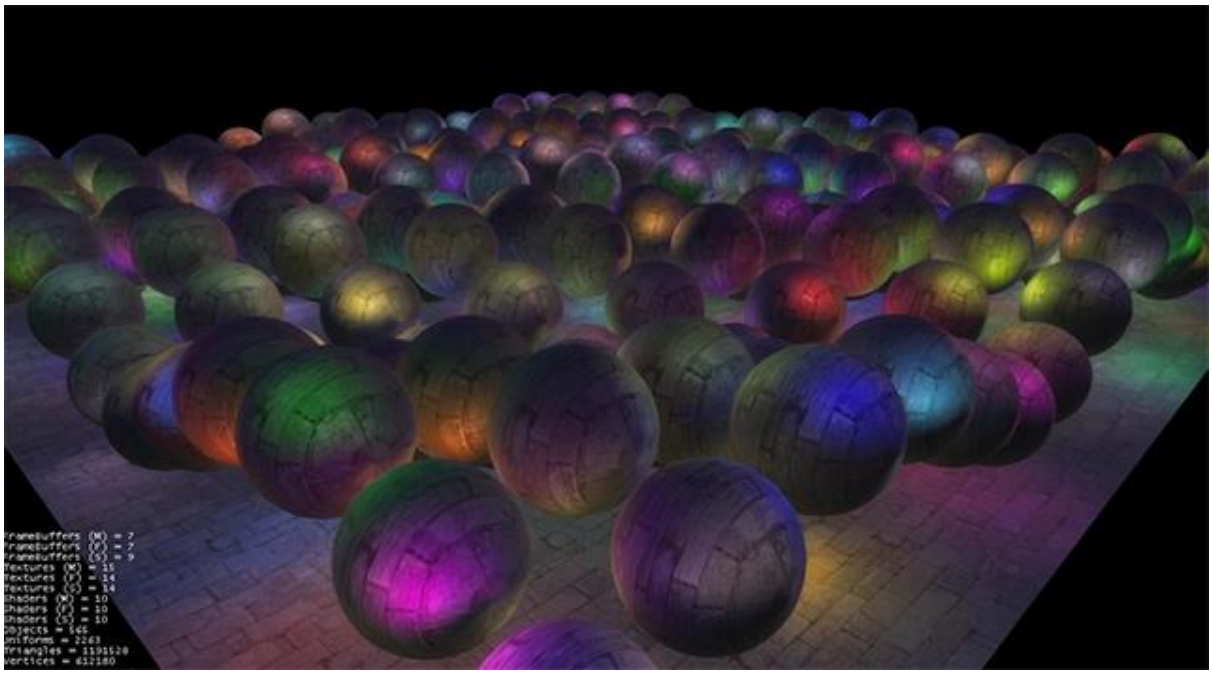


Ilustración 1.3 Ejemplo de Deferred Rendering

Esto lo consigue, dividiendo el proceso de dibujado en dos fases: la primera fase, donde trataremos la geométrica que tenemos en la escena, y una segunda fase donde trabajaremos con las luces que queremos usar.

Fase 1: Tratamiento de la geometría.

En esta fase, nuestro objetivo es juntar toda la geometría en algún almacenamiento donde podamos leer/consultar cuando nosotros queramos. La manera en la que se abordó esto, fue usar los buffers de colores. Sabemos, que un color se puede representar con el sistema clásico RGB (Rojo-Verde-Azul). Cada canal de este sistema es, en sí mismo, datos de tipo flotante. Si analizamos la geometría que usualmente hay en una escena (posiciones en el mundo, normales de los objetos, tangentes y bitangentes, etc.), tenemos que son datos de tipo flotantes: x, y, z. Un buffer de color, es un lugar de memoria donde podemos almacenar distintos valores de colores. Por tanto, si usamos los buffers como arrays, nos encontraremos con un sistema donde podemos almacenar todos los datos que queramos de una escena, siendo accesibles desde cualquier etapa de la GPU. Para recuperar la información, es como usualmente se hace usando el mapeado de texturas con las coordenadas de texturas, también conocidas como UV.

Como es lógico, aparte de los buffers correspondientes a la geometría de los objetos, también almacenaremos las distintas texturas que usen dichos objetos. Todos estos buffers que usaremos para almacenar la información de los objetos en la escena, se denominan como “Geometry Buffers”.

Una forma de ver esto sería como en la ilustración 1.4. En este ejemplo, tenemos cuatro buffers para almacenar la información de la escena y se lo aplicamos a los objetos a modo que podamos ver la información de una forma visual (las posiciones y normales se lo aplicamos como si fuesen colores). Así, el primer buffer, muestra las posiciones de los objetos; el segundo, las normales de dichos objetos; el tercer buffer para mostrar sus texturas base y un cuarto para los reflejos especulares.

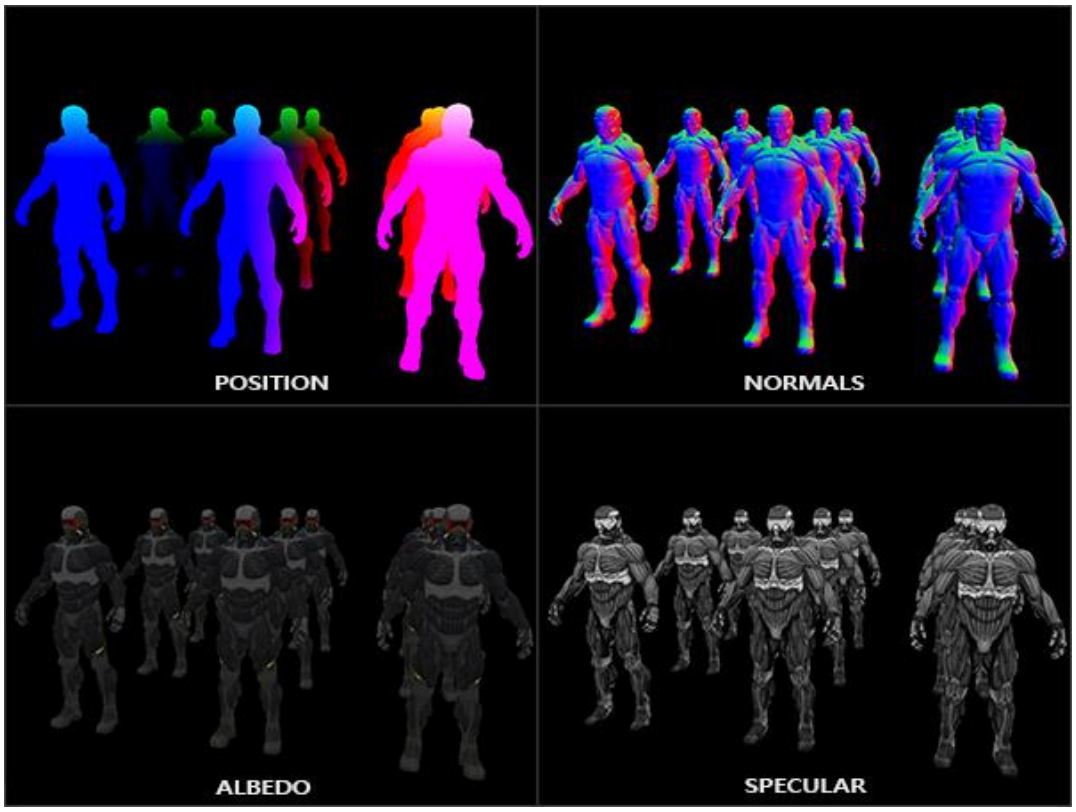


Ilustración 1.4 Ejemplo de uso del GBuffer

Como es lógico, esta etapa se ejecutará tantas veces como objetos tengamos en la escena puesto que nuestro objetivo es almacenar toda la información de nuestros objetos en los buffers. Aunque a priori pueda parecer costoso, lo único que se hace es asignar valores a los distintos buffers. Por tanto, los cálculos en esta etapa son relativamente rápidos, teniendo un *O(Número de objetos)*.

Etapa 2: Calculo de la iluminación

Dado que en etapa disponemos de la geometría que necesita ser dibujada, solo necesitaremos ejecutar este proceso una única vez. Esto nos permite que, dado que solo debemos recorrer los pixeles que forman dichos buffers, podemos realizar operaciones costosas de iluminación. Gracias a esto, conseguimos optimizar los cálculos dado que solo se recorrerán los pixeles/fragmentos una única vez, pero haciendo todos los cálculos correspondientes.

En términos de complejidad, estamos hablando de un *O(Número de fragmentos * Número de luces)*. Una cifra claramente mucho menor comparado con la que obteníamos en el Forward Rendering.

En la imagen 1.5, podemos apreciar cómo sería el ciclo competo de Deferred Rendering, pudiendo observar las dos fases ya comentadas: la fase de la geometría donde solo se guardan los valores en los buffers (eliminando la parte de iluminación), y la segunda fase en la que realizaremos el procesado de la iluminación una única vez.

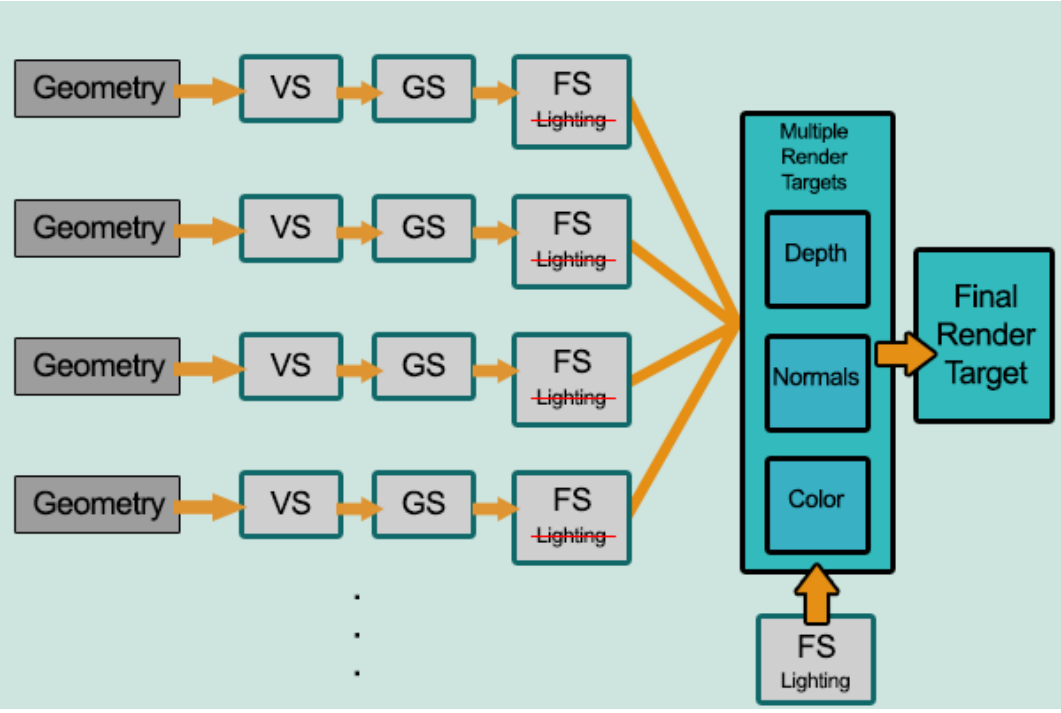


Ilustración 1.5 Ciclo de dibujado en Deferred Rendering

1.4.3 Problemas en Deferred Rendering

Deferred Rendering claramente nos brinda una mejora con respecto al tiempo de cálculo de iluminación, pero también tiene inconvenientes a tener en cuenta.

El primero es que, al usar buffers de colores, requerimos más memoria de la que usualmente necesitamos. Esto se puede incrementar gravemente si aumenta el número de información geométrica del objeto que necesitemos pasar a la etapa de iluminación. Guerrilla Games, una de las primeras compañías en implementar este sistema en el videojuego Killzone 2[1], realizó un estudio sobre este problema y observó que el mejor ajuste para poder ahorrar memoria, pero a la vez tener toda la información necesaria, es el que muestra la ilustración 1.6. En ella, podemos observar que existen varios tipos de colores RGB en los buffers. Su principal diferencia es la precisión con la que cada canal guarda sus valores. Por tanto, debemos de tener cuidado a la hora de almacenarlos, pues podríamos perder información vital. Por ejemplo, no es lo mismo perder información sobre la posición del objeto, que los valores que componen la normal de dicho objeto. Guerrilla Games hizo varias pruebas concretando que los buffers de colores con cuatro canales de tipo RGBA8 era los idóneos para sus necesidades. Además, aprovecharon partes de los buffers que no se usaban, como puede ser el canal alfa, para guardar información extra como los cálculos de profundidad o la oclusión ambiental.

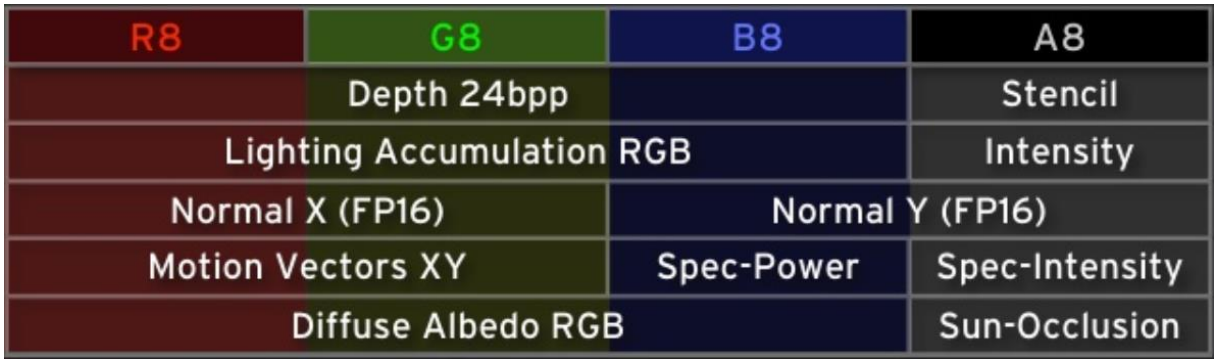


Ilustración 1.6 Estructura del GBuffer en el Killzone 2

[1] https://www.guerrilla-games.com/read/deferred-rendering-in-killzone-2

Como nota adicional al estudio que hicieron la compañía de videojuegos Guerrilla Games, es que el juego salió en 2009 para la PlayStation 3. Por tanto, las limitaciones que tenían en esa consola, son muy diferentes a las limitaciones que tenemos hoy en día; permitiéndonos poder usar canales de mayor tamaño como pueden ser RGBA32, usadas en este estudio.

Otro inconveniente importante que nos encontramos en esta técnica es el uso del blending. El blending consiste en combinar dos colores cualesquiera. Esto implica, además, poder combinar dos texturas diferentes. Por ejemplo, supongamos que tenemos una pared y queremos juntar la textura de la pared con otra textura, esto sería un ejemplo claro del uso de la técnica del blending. Esto en Deferred Rendering es imposible, ya que tenemos toda la información almacenada junta en un mismo buffer. Por tanto, no se podría saber qué parte de los pixeles que compone ese buffer es el que debe realizar el blending. Esto se puede solucionar, añadiendo una tercera fase al Deferred Rendering donde haríamos los cálculos de forma tradicional usando Forward Rendering:

Fase 1: Hacemos la pasada de la geometría.
Fase 2: Hacemos los cálculos de iluminación.
Fase 3: Hacemos Forward Rendering a los objetos restantes.

El blending también es usado para renderizar objetos transparentes, ya que aprovecha el canal alfa de las texturas. Utilizando la solución anterior, resolvería nuestro problema, aunque puede convertirse en otro de rendimiento según la cantidad de objetos transparentes que haya en ella, sin obviar cómo de compleja es su geometría.

Por último, cualquier técnica de mejora visual (como puede ser el SSAO) o efectos de postprocesamiento, hay que adaptarlos a las nuevas fases. Durante el trabajo, se investigó lo fácil o difícil que es adaptarlas, y nos encontramos con dos casos:

- El primer caso, como puede ser SSAO, es cuando se requiere de hacer llamadas de dibujado para poder realizar ciertos calculos. Esto no supone

un problema ya que, al tener toda la información almacenada en un buffer, se puede seguir la misma filosifia de trabajo como en la etapa segunda cuando realiza los calculos de iluminación. De hecho, mejora la velocidad de procesado ya que no requiere de volver a procesar todos los objetos para hacer sus respectivas llamadas.

- El segundo caso, como puede ser el HDR o la correción del gamma, es cuando se requiere hacer operaciones después de los calculos de iluminación. Estas etapas, son relativamente sencillas de adaptar, puesto que podemos separar el resultado de los calculos de iluminación en una textura. Así, lo unico que necesitamos es hacer los calculos de dicho efecto sobre la textura que le hemos pasado, consiguiendo incluso independencia de la tecnica que se use para hacer los calculos de iluminación.

En conclusión, los principales inconvenientes en Deferred rendering son:

- La cantidad de información que queramos guardar puede provocar el aumento de memoria causando un desbordamiento del mismo. Su solución, como explican los de Guerrilla Games, es realizar un ajuste de la información reduciendo la calidad de los buffers según los datos que se vayan a guardar y e intentar usar los canales libres, como por el ejemplo, el canal alfa.
- El blending supone un problema pero podemos solucionarlo añadiendo una tercera fase donde solo se dibujan los objetos que vayan a hacerlo.
- Los principales efectos postprocesado pueden implicar problemas graves en cuanto a rendimiento, pero todos tienen su adaptación al ciclo de Deferred Rendering evitando así empeorar su funcionamiento.

1.4.4 Alternativas: Forward Plus Rendering

En las anteriores técnicas, todas inciden su trabajo sobre el mayor o menor número de llamadas para realizar el cálculo de luces. Esto quiere decir que, independientemente de los cálculos de luces que se hagan en los shaders, su objetivo

es el realizar el menor número de llamadas. Forward Plus (Harada, 2012) se plantea con una filosofía diferente.

Ilustración 1.7 Ejemplo visual de Forward Plus

Partiendo de lo visto en Forward Rendering, Forward Plus trata de mejorar dicha técnica calculando qué objetos son iluminados por el área de las luces que hay en la escena. Si una luz se superpone con otra, esa parte del área tendrá mayor incidencia que una luz que esté sola. Cuando termina de calcular esto, solo ilumina dichas partes que han sido iluminadas.

Esta técnica consta de tres etapas: cálculo de la profundidad, culling de las luces y cálculo de la iluminación. En la primera etapa, se hace un cálculo de la profundidad de la escena, obteniendo así los datos de cómo de alejados están los objetos de la cámara.

En la segunda etapa, utiliza un tipo de shader llamado “compute shader”. Estos solo se encargan de realizar cálculos, no entran dentro de la etapa de iluminación. Aprovechan el hardware que tiene la GPU para hacer cálculos en paralelo. Aquí, se agrupan las luces en pequeños conjuntos de 8x8 o de 16x16 y se almacenarán sus índices en una lista donde se indicarán si son visibles o no. El objetivo de juntarlos en grupos, es el de poder procesarlos en paralelo. De hecho, usando tiles de 16x16,

podrán procesarse hasta 256 luces al mismo tiempo. Para agilizar esta etapa, se usa la técnica denominada “Frustum Culling”, que consiste en obtener una pirámide que supone el campo de visión de la cámara. Esta pirámide truncada se obtiene con los valores máximo y mínimos obtenidos del cálculo de la profundidad de la etapa anterior. En la ilustración 1.8, podemos ver visualmente cómo es dicha pirámide. Todo lo que entre dentro de ésta, será lo que se verá en la escena; todo lo que no, se puede descartar ya que no se verán en ella.

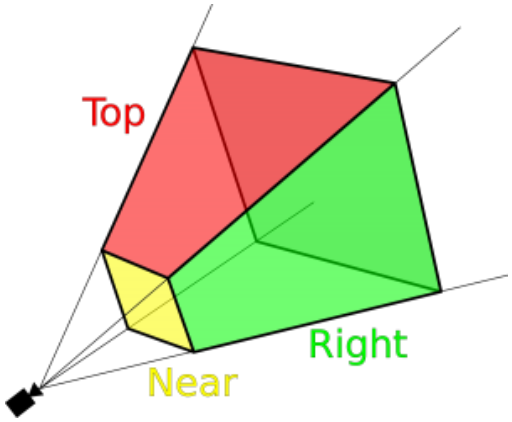


Ilustración 1.8 Ejemplo de un frustum

Una vez que tenemos esa lista de luces, pasamos a realizar la última etapa: la iluminación. En esta etapa, se realiza los cálculos de la iluminación como lo visto hasta ahora en el Forward Rendering, pero solo de las luces que hayan sido marcadas como visibles. Para mayor optimización, lo recomendable es usar frustum culling sobre los objetos para solo realizar cálculos sobre los objetos que entren dentro del frustum.

En Forward Plus también nos encontramos que necesitamos hacer una tercera pasada para el cálculo de los objetos transparentes. Si bien es cierto que no tenemos toda la información junta como en Deferred Rendering, los objetos transparentes recurren a la técnica del blending para ser dibujador. Entonces, es necesario que primero se dibujen los objetos sin transparencias y, por encima de ellos, se sumen los objetos que sí las tienen. Aunque a priori pueda parecer un problema de rendimiento, a tener ya la lista de luces visibles, solo nos recorreremos esas y no todas como se hacía en Forward Rendering.

1.4.5 Bump Mapping

Dado que nuestra idea era implementar las tres técnicas de rendering y ponerlas a prueba, se pensó si era conveniente añadir ciertos efectos visuales al render o no. Dado que Bump Mapping necesita unos cálculos muy concretos en ciertas etapas del renderizado, se estudiará como de sencillo es incorporar esta técnica a cada uno de los renders. Así, se podrá sacar conclusiones al final basándose tanto en el rendimiento como en la facilidad de incorporar o no ciertos efectos.

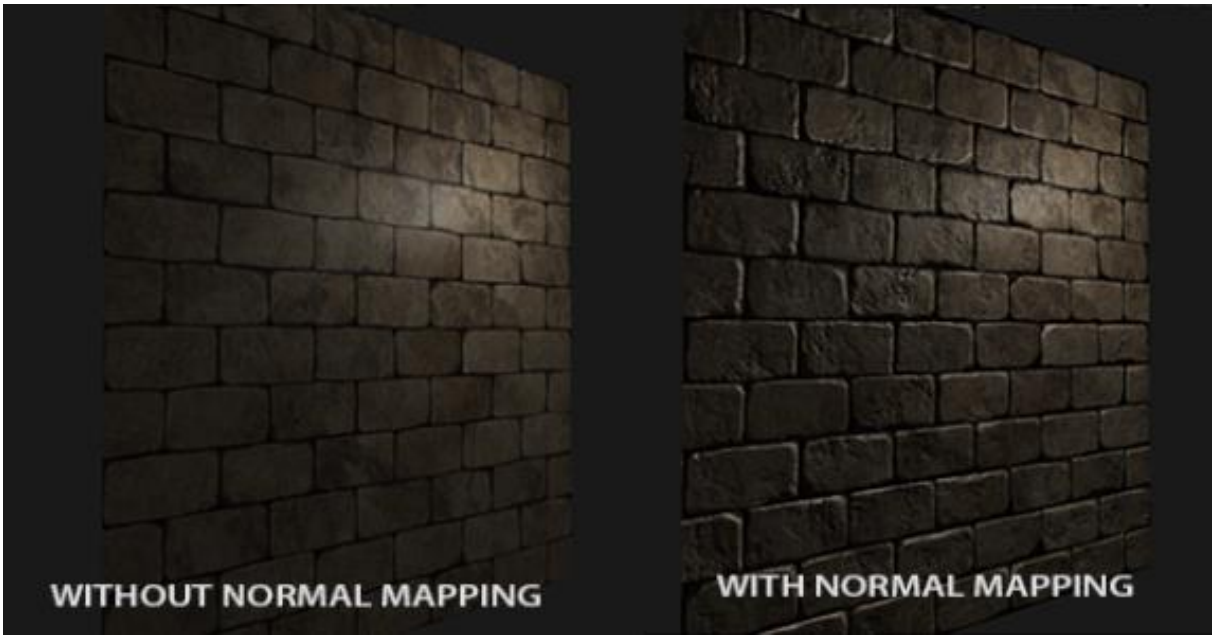


Ilustración 1.9 Ejemplo de Bump Mapping

Bump mapping (Blinn, 1978) es una técnica visual que nos permite dar mayor realismo a los objetos sin tener que crear una geometría con mucho detalle o incluso cambiarla. Partimos, de que cada objeto tiene una componente normal. Con la normal, nosotros realizamos los cálculos de iluminación. Entonces, si modificamos sus normales, modificaremos la forma con la que se iluminará dicha parte. En la ilustración 1.10, podemos ver cómo cambia una normal de un objeto sin alterarlo, a otro alterado.

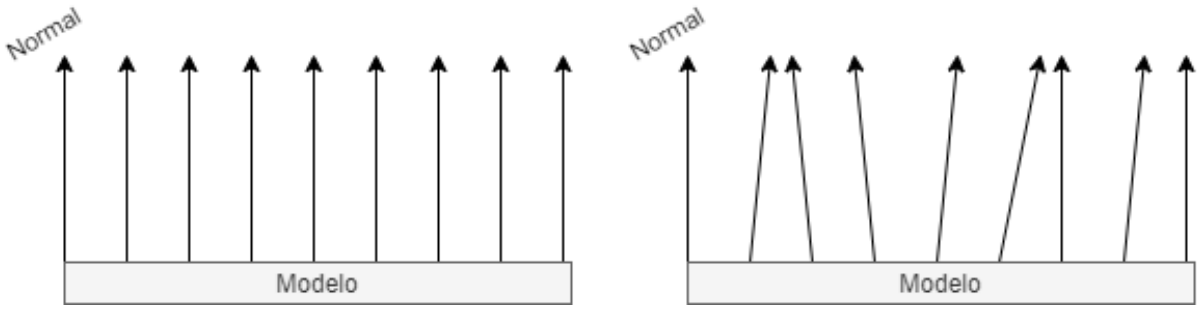


Ilustración 1.10 Ejemplo de modificación de las normales

Para ello, debemos calcular cómo han de desplazarse las normales para que el resultado quede como el deseado. Para hacer esto, necesitaremos una segunda textura que nos indique si la normal en ese punto ha de modificarse o no. Esto se consigue, usando una copia de la textura, pero con un único color. Cuanto más oscuro sea dicho color, más sensación de profundidad tendrá esa parte y viceversa.

Todo esto se calcula en la etapa de iluminación, que es cuando realizamos operaciones con las normales. Pero se nos presenta un problema: las normales han sido calculadas en el espacio del mundo. Sin embargo, los cálculos de luces los realizamos sobre el espacio de visión de la cámara. Es decir, no podemos operar correctamente con las normales ya que no están en el mismo espacio. Para solucionar esto, recurrimos a la matriz TBN. Esta matriz, está compuesta con las normales, tangentes y bitangentes del objeto. Así, con ellas podremos desplazar la normal correctamente ya que, con ellas, estará alienada como queremos a la textura.

1.4.6 Los efectos de postprocesamiento

No todo reside en el cálculo de la iluminación. Los efectos postprocesado son aquellos que se realizan una vez que tenemos todos los cálculos de iluminación hechos. Su objetivo es de realizar ciertas correcciones a la imagen resultante del proceso de la iluminación. Hay una gran variedad de efectos, pero en este trabajo solo se enfocarán en dos de ellos: la corrección gamma y en el High Dynamic Range (HDR).

Corrección gamma

Las pantallas CRT no realizan una conversión linean de la intensidad de voltaje a intensidad de luz. Cuando idearon las LCDs, las idearon con el objetivo de imitar las pantallas CRT. La corrección gama consiste en compensar el comportamiento no lineal de la pantalla.

Hay varias formas de corregir esto, pero la seleccionada para este trabajo ha sido mediante la corrección de fragments. Una vez que se ha calculado toda la iluminación, se calculará para cada fragment el valor real que debería tener según la siguiente formula:

```
Color final = pow(Color actual, vec3( 1.0 / Valor del gamma ))
```

Donde el valor gamma será elegido por parte del usuario.

High Dynamic Range (HDR)

High Dynamic Range surge como solución al problema que ocasiona el uso de los framebuffers. Esto es porque los valores quedan almacenados entre [0, 1], no pudiendo sobrepasar este último. Si por ejemplo nos encontramos con una escena cuyos objetos son muy brillantes, poner dichos objetos en 1 no producirá el efecto deseado.

HDR soluciona esto haciendo primero, que se pueda sobrepasar el valor de 1, pero al final, se vuelve todo a la escala de [0, 1]. Así, si una escena es muy brillante, la escena será brillante y viceversa. Para hacer esto, deberemos hacer esto en los fragment shader:

```
// Realizamos el HDR
vec3 NuevoColor = vec3(1.0) - exp(-ColorDelFragment * Exposicion);
// Corregimos la escena
NuevoColor = pow(NuevoColor, vec3(1.0 / ValorDelGamma));
// Asignamos el color final
ColorFinal = vec4(NuevoColor, 1.0);
```

2 ANÁLISIS DEL PROYECTO

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

A lo largo del documento se utilizarán términos y acrónimos cuya descripción aparecen en el apartado 0 ().

2.1 Requisitos iniciales del proyecto

Los requisitos iniciales del proyecto son:

- Realizar un estudio teórico de la técnica de Deferred Rendering.
- Tratar a nivel teórico las ventajas de esta técnica frente al Forward Rendering y las desventajes que presenta Deferred Rendering.
- Diseñar un prototipo donde muestre una escena 3D con carga de objetos desde un almacenamiento secundario.
- Realizar un benchmarking que permita comprobar la calidad de los resultados.
- El usuario debe moverse con libertad por la escena.
- El usuario debe poder ver información util de la escena y del rendimiento del mismo.
- El prototipo debe ser facil de usar.
- El usuario debe poder cargar cualquier objeto 3D que desee.
- El usuario debe poder cambiar entre las tres técnicas desarrolladas y cambiar el número de luces.

2.2 Alcance

Al finalizar el proyecto, se tendrá los siguientes entregables:

- Un **prototipo 3D** se tendrá un ejecutable donde se podrá mover el usuario a lo largo de una escena elegida. Para ello, se debe diseñar, de una forma sencilla, el poder cargar objetos al gusto del usuario.
- **Manual de usuario** donde se recoge la información que pueda necesitar el usuario para su uso.
- La **memoria del proyecto** donde viene todo el analisis y desarrollo realizado.

2.3 Hipótesis y restricciones

El TFT se define como una asignatura de 12 créditos, lo que supone que la duración total del proyecto será de 300 horas, incluyendo todas las etapas del ciclo de vida, con la excepción del mantenimiento. Por consiguiente, la principal restricción aplicable es la limitación de la duración del trabajo.

También se aplica una limitación de presupuesto: se debe usar en el desarrollo software libre para que los gastos sean mínimos.

2.4 Dataset de las pruebas

Para la realización de las pruebas, se ha usado un conjunto de modelos de libre acceso para que los costes del mismo no se incrementaran. El principal modelo usado para las pruebas ha sido la escena creada por Crytek llamada “Sponza”[2]. Sponza, ha sido usado por la mayoría de los estudios teóricos sobre iluminación debido a que su geometría y complejidad, es perfecta para poner a prueba los algoritmos de iluminación.

[2] https://www.cryengine.com/marketplace/product/CEMP-1102

2.5 Tecnologías utilizadas

Había dos objetivos claro cuando se planteó qué tecnologías usar. El primero, es que se buscaba que fuera software libre. Esto es muy importante dado que no queríamos que los costes se elevaran. El segundo objetivo era poder obtener todo el potencial para así optimizar lo mejor posible todas las técnicas y a su vez, no fuera un limitante para esto mismo.

Por tanto, el lenguaje de programación escogido para este proyecto ha sido C++ por su gran potencial dentro de proyectos complejos sobre gráficos. Apoyado con C++, hemos usado la librería grafica OpenGL en su última versión (4.6).

Junto a OpenGL, se han usado una serie de librerías de código abierto, proporcionándonos más control y potencial en el proyecto:

- **GLFW**: Es una librería ligera[3] que proporciona herramientas para poder gestionar y crear ventanas en windows para OpenGL. Es perfecta para poder configuar todos los parametros tipicos en la aplicación grafica tales como cambiar la resolución, pantalla completa, etc., de una forma sencilla.
- **GLEW**: Es una libreria multiplataforma[4] pensada para porpocionar facilidades al usar OpenGL en tiempo de ejecución.
- **GLM**: Es una librería matematica[5] en C++ pensada para usarse de apoyo a entornos graficos que usan shaders. Ofrece funciones y clases para facilitar la terea de calculos para luego ser pasado a los respectivos shaders sin tener que readaptar nada.
- **DEVIL**: Es una libreria de imagen[6] para OpenGL que nos permite cargar cualquier tipo de extesión de una manera sencilla e intuitiva. Actualmente suporta 43 formatos y tiene un conjunto de de funciones que nos permite cargar directamente en modo textura.

[3] https://www.glfw.org/
[4] http://glew.sourceforge.net/
[5] https://glm.g-truc.net/0.9.9/index.html
[6] http://openil.sourceforge.net/

- **ASSIMP**: Es una librería multiplaforma[7] que nos permite cargar cualquier formato 3D en nuestra aplicación. Está pensada, además, para cargar escenas con jerarquia de objetos. Junto a DevIL, es una combinación ideal a la hora de construir una aplicación grafica.
- **IMGUI**: Es una librería que nos permite construir intefaces graficas[8] dentro de nuestra aplicación 3D de una manera muy sencilla. No interfiere con el rendimiento de la aplicación y es muy flexible y personalizable.

2.6 Metodología de desarrollo de software

En la actualidad, un proyecto de software no se puede concebir sin usar una metodología de software. Esta, permite tener una visión clara de las etapas de desarrollo, tener comentarios por parte del usurario para poder modificar cosas del proyecto o poder tener un control de calidad y costes. Por todo eso, existen diferentes tipos de metodologías según el tipo de proyecto que se vaya a realizar. A continuación, se explicarán las más conocidas y cuál ha sido escogida para el desarrollo de esta aplicación 3D.

2.6.1 Desarrollo en cascada

El desarrollo en cascada consiste en separar el proyecto en diferentes etapas: diseño, implementación, correcciones, etc. Está pensado que hasta que no termine una etapa, no podrá empezar la siguiente. En la ilustración 2.1, podemos observar gráficamente como sería un desarrollo de este tipo.

Las ventajas de usar este desarrollo son varias:

- Es un modelo altamente conocido, por tanto es facil de implementar en cualquier equipo.
- Al estar todo previamente definido y diseñado, apenas habrá retrasos por modificaciones de ultima hora.

[7] https://www.assimp.org/
[8] https://github.com/ocornut/imgui

Aunque claramente, presenta una serie de desventajas:

- Cualquier error de diseño que se detecte en la etapa de desarrollo, llevará a reinicar todo el proceso haciendo que los costes y demora aumente considerablemente.
- El cliente debe esperar a que el producto esté acabado para poder probarlo, cosa que puede dilatarse mucho tiempo.
- Hasta que una etapa no esté terminada, no puede empezar otra haciendo que no se puedan programar etapas paralelamente.

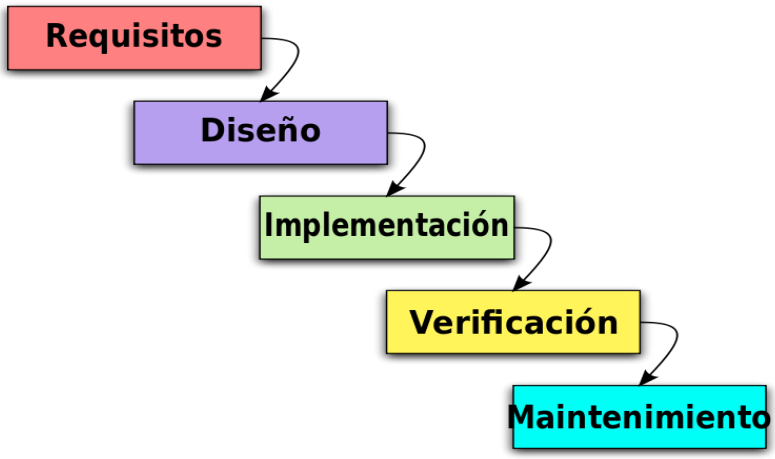


Ilustración 2.1 Ejemplo de desarrollo en cascada

2.6.2 Desarrollo en espiral

El desarrollo espiral surge cambiando las ideas del desarrollo en cascada e incremental. Las actividades conforman una espiral, siendo cada bucle del mismo un conjunto de tareas programadas. Tiene varias fases: la primera es la encargada de fijar objetivos y de realizar las investigaciones oportunas. A continuación, se realiza un análisis de riesgos de dichas tareas para ver el alcance del mismo. Le sigue la implementación y testeo, y finalmente una etapa de prueba final. En cuanto se termina ese conjunto de tareas, vuelve a empezar otra vez todo. En la ilustración 2.2, se puede apreciar de forma gráfica lo comentado.

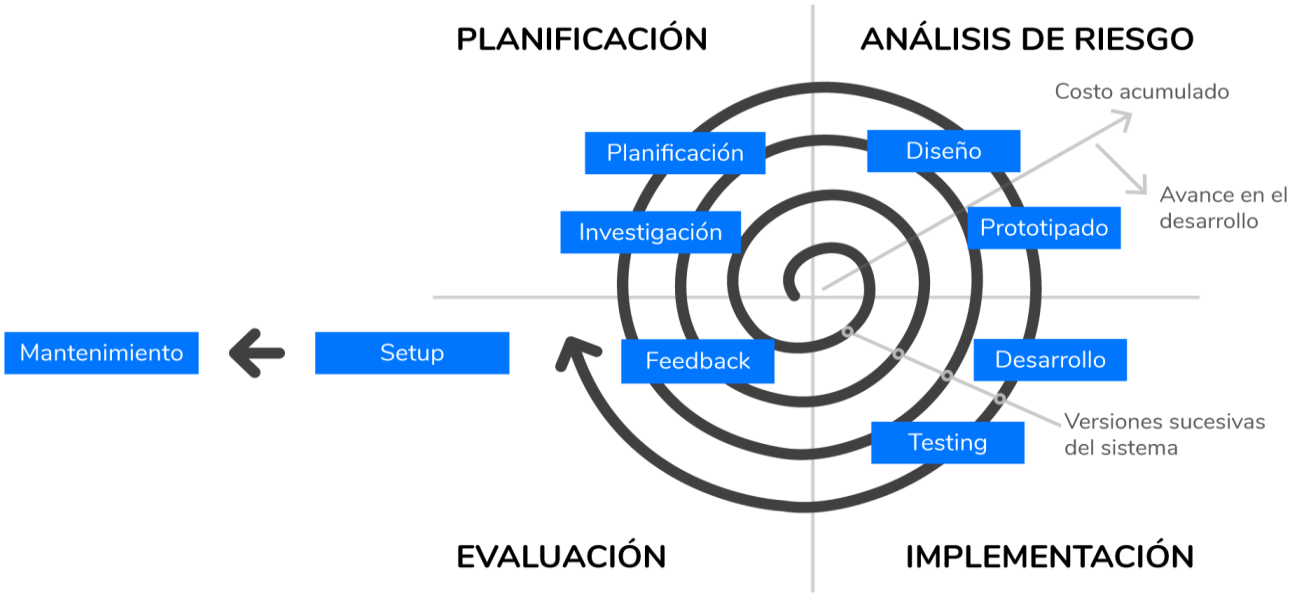


Ilustración 2.2 Ejemplo de desarrollo en espiral

También, hay que percatarse que un nuevo ciclo de la espiral crece de la siguiente forma:

- **Angular**: Realizar un nuevo avance en el proyecto.
- **Radial**: Aumenta el coste del mismo.

Por último, comentaremos sobre la metodología usada en este proyecto: Scrum.

2.6.3 Scrum

Scrum surge como una herramienta para poder aplicar un conjunto de buenas prácticas dentro de un desarrollo de software. Está pensada para proyectos muy cambiantes y complejos, donde se quiere ir obteniendo resultados tempranos; buscando que los equipos sean flexibles antes cualquier imprevisto. Su funcionamiento consiste en pequeños ciclos de tiempo definidos llamados “sprint” (entre 1 y 4 semana como máximo), donde el equipo debe ir incrementando el software final poco a poco. Para ello, se realizarán una serie de reuniones para ir monitorizando cómo va el proyecto dentro del sprint, y otras al final y al principio para revisar lo realizado en el sprint y para poder definir qué se realizará y como.

Esta metodología se basa en:

- Equipos de tamaños pequeños.
- Pequeñas tareas en poco tiempo, pudiendo hacer varias en paralelo.
- Los equipos están sincronizados y se van adaptando.
- Se fija tiempos máximos para realizar tareas.
- Los resultado se pueden mostrar cada cierto tiempo.

¿Por qué Scrum para este proyecto?

Porque era ideal para poder compaginar con las demás asignaturas que había durante el curso. Así, según cómo se fuera avanzando, se podía maniobrar de diferentes formas. De hecho, este trabajo se ha podido adaptar a una pandemia gracias a la flexibilidad que tiene.

2.7 Estimación del tamaño y esfuerzo

Ya que el presente proyecto es un TFG, no existen restricciones de tipo económico, sino de tipo temporal (un número aproximado de horas). Por consiguiente, los cálculos de tamaño del proyecto están supeditados el tiempo disponible. En cuanto al esfuerzo, se dispone de tan un solo efectivo (la persona autora del trabajo).

Para la estimación, se ha recurrido al uso de **COCOMO** (Idri, 2000) publicado por Barry Boehm en 1981 en su libro “Software Engineering Economics” como herramienta para la obtención, de manera aproximada, del esfuerzo y del tamaño del proyecto. Más concretamente, se ha usado el modelo intermedio, ya que nos ofrece un nivel de detalle y precisión mucho mejor.

Antes de ponernos a calcular los datos, necesitamos específica de qué tipo de proyecto nos entramos. Hay tres modos:

- **Modo orgánico**: son proyectos desarrollados en un ambiente conocido. El proyecto es relativamente pequeño, la mayoría de los programadores

conocen la tecnología con la que están desarrollando y hay poca innovación.

- **Modo Semiacoplado**: son proyectos desarrollados con una cierta complejidad. El proyecto es relativamente medio, la mayoría de los programadores no conocen del todo bien la tecnología con la que están desarrollando y requiere cierta investigación.
- **Modo empotrado**: son proyectos complejos, grande y requieren un nivel de investigación alto.

Este proyecto entra dentro del modo semiacoplado, así que se usarán las formulas del mismo. Para calcular el esfuerzo, es necesario definir las fórmulas del modelo intermedio:

Modo de Desarrollo	Esfuerzo Nominal	Esfuerzo Ajustado	Cronograma
Orgánico	$PM_{nominal} = 3.2 \times (KSLOC)^{1.05}$	$PM = 3.2 \times EAF \times (KSLOC)^{1.05}$	$TDEV = 2.5 \times (PM)^{0.38}$
Semiacoplado	$PM_{nominal} = 3.0 \times (KSLOC)^{1.12}$	$PM = 3.0 \times EAF \times (KSLOC)^{1.12}$	$TDEV = 2.5 \times (PM)^{0.35}$
Empotrado	$PM_{nominal} = 2.8 \times (KSLOC)^{1.20}$	$PM = 2.8 \times EAF \times (KSLOC)^{1.20}$	$TDEV = 2.5 \times (PM)^{0.32}$

Ilustración 2.3 Ecuaciones del Modelo Intermedio de COCOMO 81. [Boehm 1981]

Donde **KSLOC** está medido en miles de líneas de código y **EAF** es el "multiplicador de esfuerzo", que nos ayuda a obtener un esfuerzo ajustado según las características críticas del proyecto (nivel de conocimiento de los programadores, complejidad del producto, etc.).

Para este proyecto, las **EAF** que se usarán son:

- Complejidad del producto: nominal (1.0)
- Capacidad del programador: alto (0.86)
- Experiencia en el lenguaje: nominal (1.0)
- Uso de prácticas modernas: alto (0.91)

Antes de pasar a los calculos, hay que estimar cuantas lineas de código habrá en el proyecto. Se ha supuesto, que de media, una clase debe ocupar unas 155 líneas

(obviando las cabeceras). Si son unas 18 clases las que deben ser implementadas, obtenemos unas 2.790 líneas de código. Partiendo de eso, el esfuerzo del proyecto será:

$$\text{PMnominal} = 3.0 \text{ x } 2^{1.12} = 6.5204$$

Ese 6.5204 es en estimación de persona-mes. Para obtener el PM ajustado, solo hace falta multiplicar por su EAF:

$$\text{PMajustado} = \text{PMnominal} * (1.0 * 0.86 * 1.0 * 0.91) = 5.102$$

Una vez que tenemos el PM ajustado calculado, pasamos a obtener las variables importantes para el proyecto:

- Tiempo del proyecto:

  $$\text{TDEV} = 2.5 \text{ x } 5.102^{0.35} = 4.4223$$

  La duración, por tanto, será de unos 4.42 meses.

- Nº medio de personas:

  $$\text{Nº medio de personas} = \frac{5.102}{4.4223} = 1.15$$

  En número medio de personas, por tanto, será de 1.15 personas.

2.8 Planificación temporal

A continuación, la planificación temporal diseñada para este proyecto según la estimación de esfuerzo. Se ha tomado en cuenta que los días festivos y fin de semanas no se trabaja salvo semana santa.

Sprint	Inicio Sprint	Final Sprint	09-mar	10-mar	11-mar	12-mar	13-mar	14-mar	15-mar	16-mar	17-mar	18-mar	19-mar	20-mar	21-mar	22-mar	23-mar	24-mar	25-mar	26-mar	27-mar	28-mar	29-mar	30-mar	31-mar	01-abr	02-abr	03-abr	04-abr	05-abr	06-abr	07-abr	08-abr	09-abr	10-abr	11-abr
Sprint 1	09/03/2019	17/03/2019																																		
Sprint 2	17/03/2019	25/03/2019																																		
Sprint 3	25/03/2019	02/04/2019																																		
Sprint 4	02/04/2019	10/04/2019																																		
Sprint 5	10/04/2019	20/04/2019																																		
Sprint 6	20/04/2019	28/04/2019																																		
Sprint 7	28/04/2019	06/05/2019																																		
Sprint 8	06/05/2019	14/05/2019																																		
Sprint 9	14/05/2019	22/05/2019																																		
Sprint 10	22/05/2019	01/06/2019																																		

Ilustración 2.4 Cronograma del 9 de marzo al 11 de abril

Sprint	Inicio Sprint	Final Sprint	12-abr	13-abr	14-abr	15-abr	16-abr	17-abr	18-abr	19-abr	20-abr	21-abr	22-abr	23-abr	24-abr	25-abr	26-abr	27-abr	28-abr	29-abr	30-abr	01-may	02-may	03-may	04-may	05-may	06-may	07-may	08-may	09-may	10-may	11-may	12-may	13-may	14-may	15-may
Sprint 1	09/03/2019	17/03/2019																																		
Sprint 2	17/03/2019	25/03/2019																																		
Sprint 3	25/03/2019	02/04/2019																																		
Sprint 4	02/04/2019	10/04/2019																																		
Sprint 5	10/04/2019	20/04/2019																																		
Sprint 6	20/04/2019	28/04/2019																																		
Sprint 7	28/04/2019	06/05/2019																																		
Sprint 8	06/05/2019	14/05/2019																																		
Sprint 9	14/05/2019	22/05/2019																																		
Sprint 10	22/05/2019	01/06/2019																																		

Ilustración 2.5 Cronograma del 12 de abril al 15 de mayo

Sprint	Inicio Sprint	Final Sprint	16-may	17-may	18-may	19-may	20-may	21-may	22-may	23-may	24-may	25-may	26-may	27-may	28-may	29-may	30-may	31-may	01-jun	02-jun	03-jun	04-jun	05-jun	06-jun	07-jun	08-jun	09-jun	10-jun	11-jun	12-jun	13-jun	14-jun	15-jun	16-jun	17-jun	18-jun
Sprint 1	09/03/2019	17/03/2019																																		
Sprint 2	17/03/2019	25/03/2019																																		
Sprint 3	25/03/2019	02/04/2019																																		
Sprint 4	02/04/2019	10/04/2019																																		
Sprint 5	10/04/2019	20/04/2019																																		
Sprint 6	20/04/2019	28/04/2019																																		
Sprint 7	28/04/2019	06/05/2019																																		
Sprint 8	06/05/2019	14/05/2019																																		
Sprint 9	14/05/2019	22/05/2019																																		
Sprint 10	22/05/2019	01/06/2019																																		

Ilustración 2.6 Cronograma del 16 de mayo al 18 de junio

2.9 Presupuesto

A continuación, se detallarán los costes que ha habido durante el proyecto.

Presupuesto en hardware y software

En la tabla 2.9 se detalla el coste de hardware y software que se tendrá en la realización de este proyecto. En coste global, se define el valor del producto en el mercado. En coste real, se define el valor amortizado dentro del proyecto. Dado que los ordenadores suenen tener un periodo de vida de 5 años, se calculará el valor real dentro de su vida útil.

Concepto	Coste global	Coste real
PC: • Ryzen 7 1700 • 16GB Ram • SSD 512GB • GTX 1060	830€	110,66€
Sistema Operativo Windows 10 Professional	0,0€	0,0€
	Total	110,66€

Tabla 2.1 Costes en hardware y software

Presupuesto en personal

Aquí se recoge el presupuesto destinado a las personas que participen en el proyecto. El proyecto ha sido realizado únicamente por una única persona realizando una labor de analista de software y programador. Por consiguiente, el sueldo base que se usará, será el indicado por el último BOE disponible en la realización de esta memora; a saber, el del 18/10/2019 con una cantidad de 25.727,97€[10] brutos al año. Si le aplicamos el IRPF, obtenemos que de esa cantidad 1.633,70€ es para la seguridad social y 3.773,80€ en IRPF; quedándonos una cantidad de 20.320,50€ neto

[10] https://www.boe.es/boe/dias/2019/10/18/pdfs/BOE-A-2019-14977.pdf

al año. O lo que es lo mismo, unos 1.693,38€ al mes en 12 pagas. En la tabla 2.2, se recoge el sueldo base de dicho trabajador, un seguro laboral y los impuestos.

Concepto	Coste	Duración (meses)	Total
Sueldo base	1.693,38€	4	6.773,50€
		Total	6.773,50€

Tabla 2.2 Costes en personal

Otros

A parte del equipo y del sueldo de trabajador, este proyecto también tiene una serie de gastos relacionado con el lugar donde se va realizar, incluyendo luz e internet; o simplemente material inmobiliario como mesa, folios, etc... En la tabla 2.3 se detalla todo ese coste con IVA incluido.

Concepto	Coste	Duración (meses)	Total
Luz	58€	4	232€
Internet	30€	4	120€
Material ofimática	40€	-	40€
		Total	392€

Tabla 2.3 Otros costes

Valoración económica global

Una que se ha detallado todo el presupuesto detallando en qué va, en la tabla 2.4, se recoge todo el gasto global y el presupuesto final del mismo.

Concepto	Importe
Hardware y Software	110,66€
Personal	6.773,50€
Otros	392,00€
Total	7.276,16€

Tabla 2.4 Costes globales

Por tanto, el presupuesto necesario para llevar a cabo el proyecto es de unos **7.276,16€**.

3 DISEÑO DE LA APLICACIÓN

En este capítulo, entraremos en detalles sobre el diseño del sistema, viendo cada uno de los diagramas que han sido creados para el proyecto. Además, hablaremos sobre los aspectos más importante de la implementación del mismo, recalcando los puntos que han sido destacados cuando se han hablado a nivel teórico.

3.1 Diseño del sistema

Aquí, se presentarán los requisitos y diagramas del proyecto.

3.1.1 Análisis de requisitos

A continuación, se detallarán los requisitos funcionales y no funcionales del proyecto:

Requisitos Funcionales

Los requisitos funcionales son aquellos aspectos que puede hacer el sistema. Estos son:

- El programa debe cargar un objeto junto con sus texturas y visualizarlo en una escena.
- El programa debe permitir, mediante teclado, poder moverte libremente por la escena.
- El programa debe permitir cambiar el número de luces que hay en la escena.
- El programa debe permitir cambiar el tipo de renderizado de la escena.

Requisitos Funcionales

Los requisitos no funcionales son las capacidades que tiene el sistema internamente. Estas son:

- La aplicación debe ser robusta y poder cargar cualquier tipo de modelo.
- La aplicación debe poder cambiar entre tipo de visualización (los renders) sin problemas.

- La aplaicación debe permitir añadir nuevos requisitos en cualquier momento.

3.1.2 Diagrama de clases

Los diagramas de clases son representaciones de las clases que compone el proyecto y su relación entre ellas. El diagrama usado para este proyecto ha sido el siguiente:

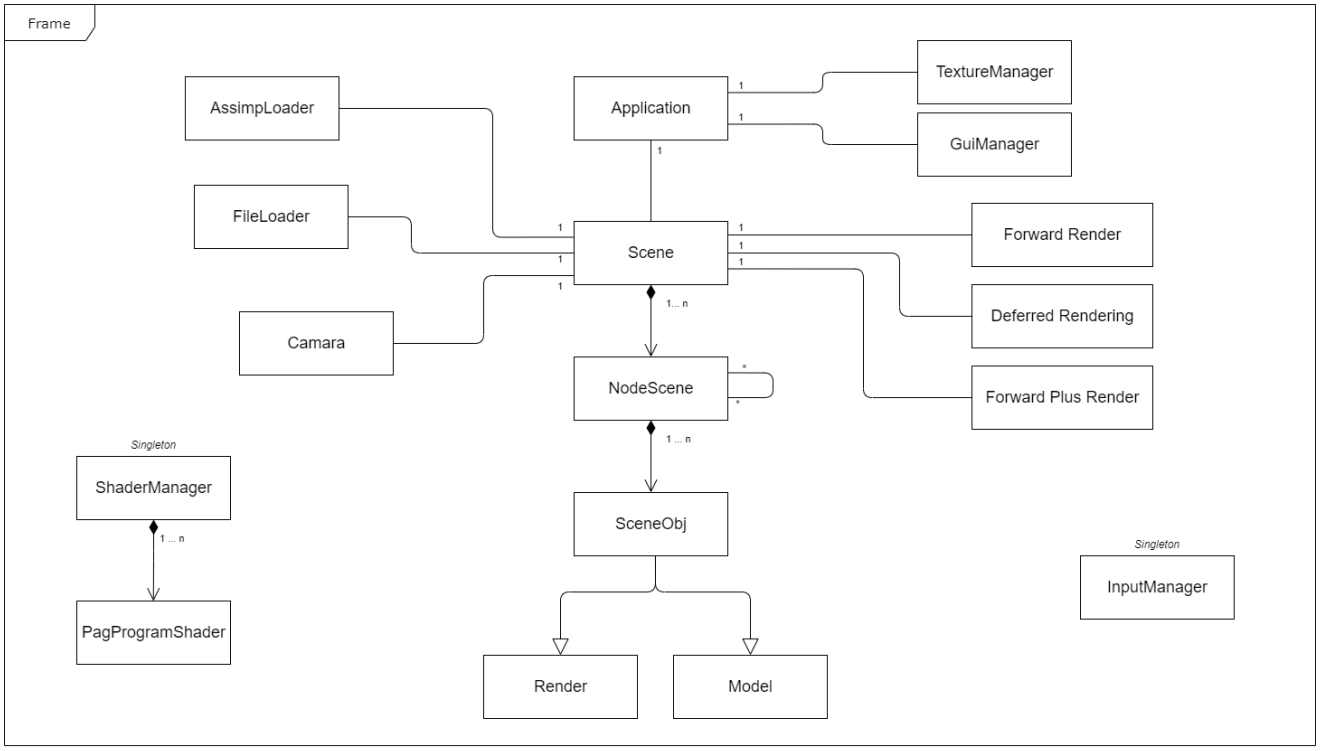


Ilustración 3.1 Diagrama de clases

3.1.3 Diagramas de clase de uso

Los diagramas de casos de uso muestran las interacciones que hay entre el usuario y la aplicación, viendo claramente los distintos tipos de interacciones que puede hacer. El diagrama usado para este proyecto ha sido el siguiente:

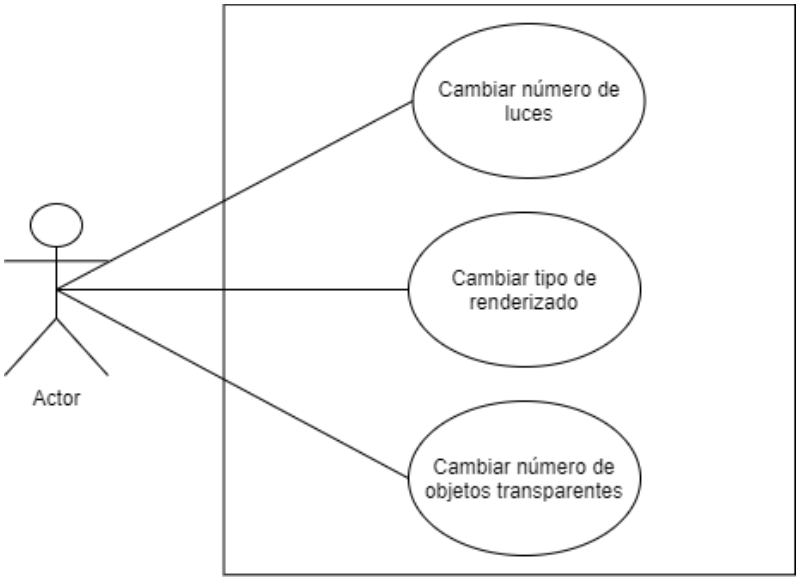


Ilustración 3.2 Diagrama de casos de uso

3.1.4 Diagramas de secuencia

Los diagramas de secuencia nos muestran el proceso por el que pasa la aplicación desde que el usuario decide realizar una acción. El diagrama usado para este proyecto ha sido el siguiente:

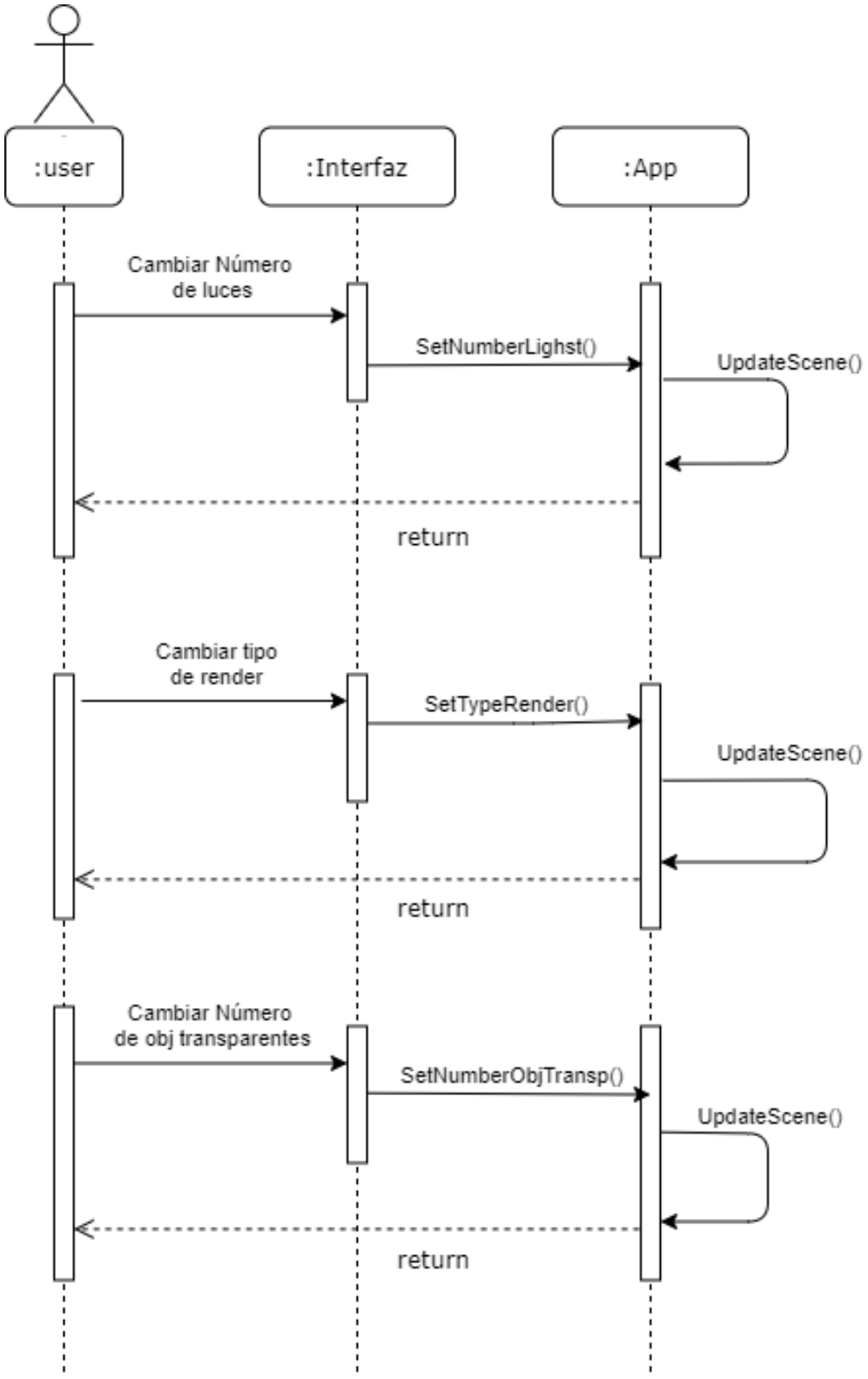


Ilustración 3.3 Diagrama de secuencia

3.2 Implementación del sistema

Una vez que se han especificado todos los diagramas necesarios para el proyecto, toca enfocarse sobre los aspectos de la implementación de las clases. Al ser un proyecto compuesto de muchas partes, solo se entrarán en detalle de las más importante para el funcionamiento del mismo.

3.2.1 La escena

Sabemos que una escena está compuesta por objetos, ya sean transparentes o no; y por una serie de luces. Pero no solo es necesario eso, se requiere de una serie de clases necesarias para que todo funcione. A continuación, las clases que componen una escena:

Cámara

La cámara es la encargada de crear las matrices de visión y de proyección. Con ellas, podremos situar los objetos en la escena y representarlos correctamente. Ha sido diseñada como una cámara libre (puede moverse en cualquier dirección), inspirada en las cámaras “First Person Shooter”.

Luces

Al tratarse de un trabajo sobre el rendimiento a la hora de elegir qué renders usar, solo se ha tenido en cuenta un tipo de luz: la luz puntual. Por tanto, ésta será almacenada en un array dentro de la escena. Está compuesta por:

- Posición: La posición dentro de la escena.
- Color: El color que produce dicha luz.
- Intensidad: intensidad con la que produce dicha luz.
- Radio: Radio de actuacion de la luz.

Renders

Los renders son las diferentes técnicas que han sido implementadas dentro del proyecto: Forward Rendering, Forward Plus Rendering y Deferred Rendering. Se hablará de esto en más profundidad en la sección 3.3.

Además de esas clases, la escena tiene los siguientes métodos:

- **InitScene**: Esta función nos permite inicializar todo lo que necesite la escena, como cargar los objetos del txt, inicializar las luces y los renders.
- **UpdateObjs**: Es llamado justo antes de la etapa de dibujado. Este método se encarga de llamar a cada uno de los objetos y actualizar su estado, ya sea posición o rotación. También actualiza la información de las luces que están en constante movimiento. Junto a eso, además se llamarán a los updates de cada render, aunque eso se entrará en más detalle en la sección 3.3.
- **SetUniforms**: Este método es llamado antes del ciclo del dibujado y justo después de "InitScene". Se encarga de pasarle a los shaders la información necesaria que no necesita ser actualizada durante el ciclo de renderizado.
- **DrawObjs**: Este método se encarga de, según el tipo de renderizado, llamar al render correspondiente.

3.2.2 Nodos y objetos escena

A la hora de estructurar la escena, se planteó que puede que haya más de un objeto a la vez en ella. Y no solo es eso, también puede pasar que esos objetos dependan de otros (padres e hijos). Por tanto, se ideó que los objetos siguieran una estructura jerarquizada de tipo nodos (clase "NodoScene") y objetos (clase "ObjScene").

La forma de estructurarse es muy parecida a usadas en motores gráficos como por ejemplo Unity. Partimos de un único nodo raíz que será el encargado de almacenar todos los demás nodos. Cada nodo, a su vez, puede contener más nodos. Al mismo tiempo, cualquier nodo puede contener cualquier cantidad de hijos. Así, si se quiere mover una parte en concreto del objeto, solo hay que ir al nodo padre y rotarlo, moverlo, escalarlo, etc. En la ilustración 3.4, se puede apreciar mejor esta estructura:

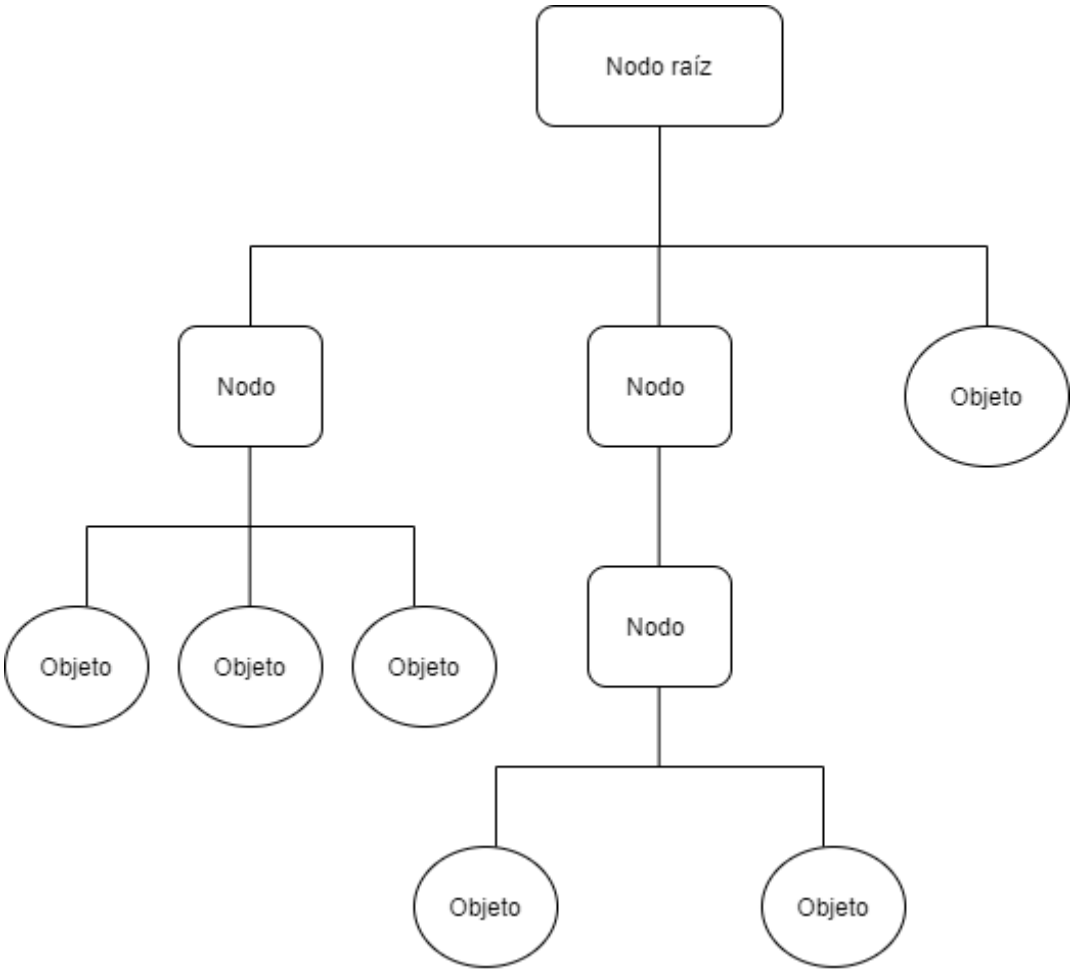


Ilustración 3.4 Ejemplo de estructura de nodos de la escena

Cuando la escena llama a los métodos de inicializar, actualizar o dibujar; el nodo padre se encarga de recorrerse, recursivamente, todos los nodos e hijos y llamar a sus respectivas funciones.

Los objetos usan la clase denominada “SceneObj”. Ésta, tiene los métodos para inicializar las partes necesarias de los shaders que han de ser actualizadas constantemente en el ciclo de renderizado. Esto puede ser las matrices de visión, la matriz de proyección, la de modelo si es que ha cambiado, o la posición de la cámara.

A su vez, esta clase hereda de la clase “Render” y de la clase “Model” encargadas de completar la funcionalidad de los “SceneObj”, explicadas en el apartado 3.2.3 “Clase modelo y clase render”.

3.2.3 Clase modelo y clase render

Todo objeto necesita posicionarse en el mundo y tener la información almacenada para poder ser dibujada. Por eso, se creó y se separó dicha funcionalidad en dos partes: la clase Model y la clase Render.

La clase Model, representa el objeto en el mundo. Es la encargada de almacenar y actualizar, la matriz de modelado que será usada en la parte de los shaders. Tiene métodos como escalar, trasladar o rotar, para poder manipularla a nuestro antojo.

La clase Render tiene toda la información necesaria para su dibujado, como puede ser sus puntos, sus normales, sus tangentes, etc. Toda esta información, es almacenada en framebuffers. Los framebuffers son espacios de memoria que tiene la GPU donde podemos alojar la información que queramos y acceder a ella en las etapas de dibujado. Dado que todos los objetos tienen información importante a la hora de su dibujado, se pensó que lo mejor sería usar una estructura enlazada. Esto es, que toda la información se almacena junta y no en framebuffers diferentes. Así, cuando se quiera acceder, no necesitas saltar a un punto de memoria lejano, sino moverte una posición de memoria al lado. En la ilustración 3.2.3, se puede ver con claridad esto último:

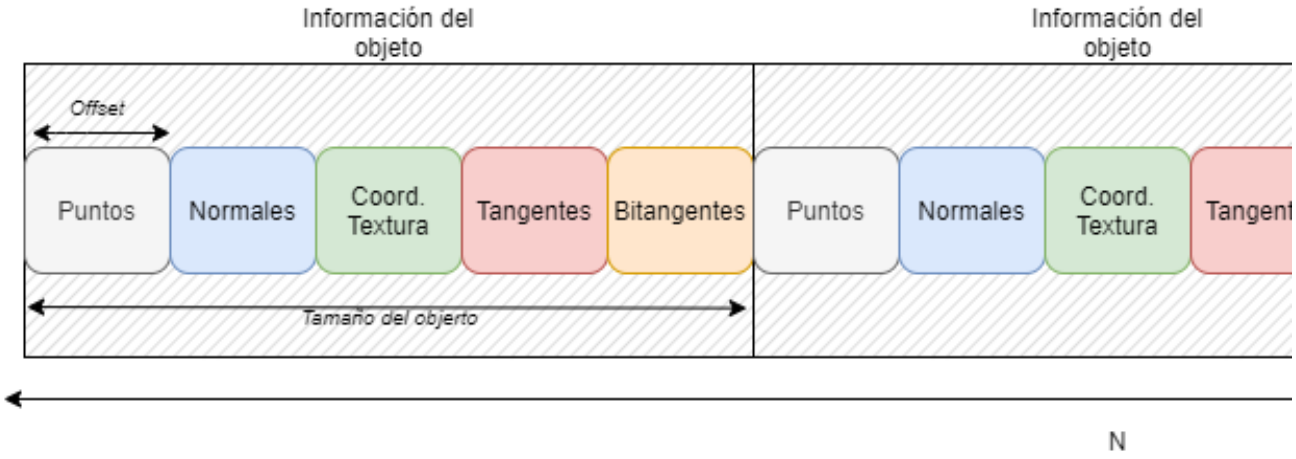


Ilustración 3.5 Ejemplo de almacenamiento en memoria de un objeto

Donde “N” representa el tamaño total del framebuffer, con “tamaño del objeto” nos movemos entre los distintos objetos almacenados en él, y con el “offset” nos movemos entre la información de cada objeto. Habrá que tener en cuenta que tendremos más de un offset según la cantidad de bytes que nos ocupe ese dato (no es lo mismo un offset de un vector de tres flotantes a un vector de dos flotantes).

Además, también tiene un ID de cada una de las texturas que necesita para su dibujado. Cuando llega el momento de activar las texturas, llama al “TextureManager” que es el encargado de darnos el ID correspondiente.

3.2.4 Managers

De cara a facilitar el acceso a cierta información, se han ideado crear ciertas clases que gestionan varias funcionalidades importantes. De entre ellas, estás son las más destacables.

TextureManager

Las texturas pueden suponer un problema si no los controlas de forma eficiente. Al poder cargar muchos objetos iguales, esto supone cargar muchas texturas que son iguales. Por tanto, estás ocupando memoria de forma ineficaz que si reusaras las texturas ya cargadas en ella. Para ello, se ha diseño la clase “TextureManager”. Conforme se procesa un objeto, se le pasa el url de la imagen al TextureManager. Este, comprueba que no existe dicha textura en el sistema. Si es así, no hace nada. Sin embargo, si no existe lo añade al mapa de texturas. Cuando finaliza la parte de inicialización de la escena, se procede a cargar todas las texturas que sean necesarias. Dado que los objetos tienen también la url de la imagen, solo necesitan llamar al TextureManager con dicha url, y esta le dará la ID que necesitan cuando van a dibujar el objeto.

ShaderManager

El “ShaderManager”, es el encargado de contener todos los shaders que necesita la aplicación. Se encarga de crearlos, cargarlos y eliminarlos cuando la aplicación finaliza. Cada shader usa la clase “PagShaderProgram” que se ideo en la asignatura de Gráficos Avanzados de cuarto con el fin de reutilizar código ya existente de nuestro trabajo a lo largo de la carrera.

GuiManager

Para no mezclar toda la implementación de la UI en otras clases, se decidió crear una sola para ésta. Aquí, tiene los métodos necesarios para usar la librería “Dear Img”. Esta solo necesita tres métodos importantes: el “StartFrame” para especificarle que comienza un nuevo frame de dibujado, “showGUI” donde tiene toda la información de qué debe dibujar en la escena, y “DestroyGUI” para eliminar liberar memoria.

3.3 Implementación de los renders

Una vez explicado cómo están estructuradas las escenas de la aplicación, pasemos a ver cómo se han implementado las tres técnicas que se han comentado de forma teórica en el capítulo 1.4 “Situación actual de la tecnología”.

Las tres clases, siguen una misma estructura: primero, con el método “createFrameBuffer”, inicializaremos todos los buffers, si es necesario, de los renders. Después, con el método “draw”, se procederá al dibujado de esa información conforme se ha explicado teóricamente.

Con respecto a las luces, durante el desarrollo se encontró con un inconveniente. Cuando pasábamos la información, se usaba un array creado en el propio shader usando un struct como forma para dotarle de diferente información:

```
struct Light {
    vec4 Position;
    vec4 Color;
    vec4 IntensityandRadius;
};
const int MAX_LIGHTS = 100;
uniform Light lights[MAX_LIGHTS];
```

Ilustración 3.6 Ejemplo de cómo pasarle un vector de luces al shader

Pero esto nos llevaba a un problema de memoria con los shaders y es que no se puede definir un array estático de más de 300 posiciones. Esto supone un problema

ya que nuestro objetivo es poner al límite estas técnicas con un número bastante elevado de luces, llegando a las 2048. Por tanto, se optó por usar los *"*Storage Buffer*"*.

Los "Storage Buffer" son zonas de memoria que se puede especificar sin usar lenguaje GLSL. Esto es idóneo porque primero, se controla a través del código de la aplicación reduciendo así el tamaño de los shaders; y segundo, se pueden crear tan grandes como se desee. Para ello, todos los renders tienen sus respectivos métodos inicializadores y actualizadores "initBufferLights" y "UpdateLights" para crear el Storage Buffer y para acceder a ellos y actualizar la información cuando las luces cambien.

Para crearlos, solo necesitamos llamar a la función *"glGenBuffers"* como hasta ahora, crearlo con *"glBufferData"* pero especificando que es de tipo *"GL_SHADER_STORAGE_BUFFER"* junto a que será *"GL_DYNAMIC_DRAW"*. Para recuperar la información, llamamos al puntero de la zona de memoria con *"glMapBuffer"* y trataremos la información como si un array dinámico se tratase.

Una vez tratada la información común de los renders, pasemos a tratar cada uno en detalle.

3.3.1 Forward Rendering

Para implementar esta técnica, salvo Deferred Rendering y Forward Plus Rendering, no necesitamos un framebuffer donde alojar información extra pues no lo necesita. Lo único que necesitamos en obtener el nodo raíz del mundo y llamar al shader correspondiente.

Por tanto, su única fase consiste en obtener cada uno de los objetos que hay en la escena, y llamar a sus respectivos métodos de dibujado. Estos, activarán tantas texturas como les sea necesarias y llamarán al método "draw" de OpenGL, especificándole qué shader debe usar previamente. Así, la GPU, con las instrucciones de cómo debe hacerlo y con toda la información almacenada, dibujará la escena.

3.3.2 Deferred Rendering

Con respecto a Deferred Rendering, el proceso se divide en dos partes: la parte de la geometría y la parte de la iluminación.

Fase 1: Tratamiento de la geometría

La parte de la geometría se almacenará en buffers de colores. Para ello, necesitamos un ID, las dimensiones de la ventana y el tipo de color que vamos a almacenar. Todos los buffers de colores siguen una estructura del estilo:

```
glGenTextures(1, &ID_Framebuffuer);
glBindTexture(GL_TEXTURE_2D, ID_Framebuffuer);
glTexImage2D(
    GL_TEXTURE_2D, 0,
    TipoColor,
    Ancho, Alto, 0,
    TipoColor, TipoDato,
    NULL
);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

glFramebufferTexture2D(
    GL_FRAMEBUFFER,
    ColorAsignado,
    GL_TEXTURE_2D,
    ID_Framebuffuer,
    0
);
```

Ilustración 3.7 Ejemplo de creación de un buffer de color

Después, habrá que crear un último buffer de color solo que esta vez va a ir destinado al buffer de profundidad. Lo único que cambia es el tipo de color, que hay que especificarle que es de tipo *"GL_DEPTH_COMPONENT"*. Por último, asignamos los buffers de colores y el de profundidad al ID que hemos generado previamente con la función *"glDrawBuffers"*.

Una vez creados los buffers de colores, podremos usarlos en los shaders asignándoles una localización como si de una textura se tratase. En el fragment shader, podemos acceder a ellos usando:

"*layout (location = nº) out TipoDato NombreDato*"

Etapa 2: Calculo de la iluminación

Una vez que hemos guardado todos los datos en sus respectivos buffers de colores, se procede a la iluminación. En esta etapa, a nivel de implementación, nos olvidado de llamar a los "draw" de los objetos. En vez de esto, usamos geometría simple como un Quad. Creamos un Quad y llamamos a dibujarlo, pero usando el shader correspondiente de la iluminación. A este shader hay que pasarle tanto la información de las luces, como la información de los buffers de colores (de la misma forma que en el proceso anterior). Lo primero que se realiza es extraer la información como si de texturas se tratase usando la función *"texture(ID_Textura, UV)"*, y después usar dicha información de igual forma como en Forward Rendering.

La diferencia clara es que solo realizamos un único "draw" comparado con Foward Rendering que realizábamos tantos "draw" como objetos tuviéramos en la escena.

3.3.3 Forward Plus Rendering

Forward Plus coge parte de lo visto a la hora de la implementación tanto de Forward Rendering como de Deferred Rendering.

Para empezar, no requiere de buffers de colores especiales para almacenar la información. Sí necesita crear un buffer de profundidad para la primera fase, que se crea de igual forma que en Deferred Rendering. Por otro lado, aparte de los "Shared Buffer" que creamos para poder compartir información de las luces, necesitamos crear un shader buffer de índices de luces. Este buffer es donde reside todo el trabajo de Forward Plus Rendering, pues es el que nos va a indicar si esa luz ha de ser calculada o no.

Para calcular eso, pasamos a la etapa de Light Culling. Primero, necesitamos recurrir a una etapa especial de los shaders denominado “Compute Shader”. En este tipo de shaders, no se busca dibujar nada, tan solo aprovechar la potencia que tiene las tarjetas gráficas de hoy en día para calcular qué luces han de ser marcadas como visibles y cuáles no. Para ello, en GLSL, obtenemos los ID del grupo en el que vamos a trabajar. Posteriormente, calculamos el frustum de la cámara. Para ello, usamos el procesado en paralelo con la función *“barrier”* que nos ofrece OpenGL. Nuestro objetivo aquí, es buscar cuál es el máximo y mínimo valor. Como la escena puede ser considerablemente grande, usamos el procesado en paralelo para agilizar los cálculos. Todas las variables que se vayan a compartir, hay que especificarlas con la etiqueta *“shared”*:

```
shared int intCompartido;
shaded float floatCompartido;
shared bool boolCompartido;
shaded vec4 vectorCompartido[TamVertor];
```

Ilustración 3.8 Variables compartidas entre shaders

Una vez creado el frustum con los valores obtenidos, pasamos a la parte de marcado de las luces. En esta parte, buscamos qué luces entran dentro del frustum y qué luces no. Para ello calcularemos, para cada luz, la distancia que hay entre ella y cada una de las paredes del frustum. Si la distancia es menor o igual que 0, no se marca. Si sí lo es, lo añadimos al array de luces visibles. Todo eso se realiza de forma paralela de igual forma que en el cálculo del mínimo y máximo.

Cuando se termina este proceso, se pasa a la fase de iluminación. Esta fase se realiza de igual forma que en Forward Rendering salvo por un pequeño detalle: ahora no nos recorremos las luces que teníamos sino las luces que han sido marcadas como visibles. Todos los demás cálculos de luces se realizan de forma normal.

3.4 Objetos transparentes

Para dibujar los objetos transparentes, conforme se explicó en el capítulo 1.3.4 “Problemas en Deferred Rendering”, recurriremos a la técnica del blending. Esto significa, que los objetos transparentes deben ser los últimos en ser dibujado. Por

tanto, todos los objetos transparentes serán tratados en una etapa diferente al final de todos los cálculos de iluminación.

Para Forward Rendering y Forward Plus Rendering, no supone ningún problema añadir una etapa más de dibujado ya que toda la información lo tiene en un mismo framebuffer. Sin embargo, al dividir Deferred Rendering en dos etapas diferentes, no ocurre lo mismo. La clave del uso del blending, es que toda la información sobre el depth buffer (la profundidad de los objetos) está también almacenada en el "Geometry buffer". Esto implica, que si queremos realizar una etapa de dibuja tradicional usando Forward Rendering, tendremos un problema visual ya que se superpondrán los objetos realizados con Deferred Rendering con los objetos de esta etapa. Para solucionar esto, lo único que necesitamos es copiar esa información al framebuffer por defecto:

```
// Especificamos que el primero se leerá y el segundo se escribirá
glBindFramebuffer(GL_READ_FRAMEBUFFER, ID_GeometryBuffer);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);

// Pasamos la información
glBlitFramebuffer(
    0, 0, Ancho, Alto,
    0, 0, Ancho, Alto,
    GL_DEPTH_BUFFER_BIT, GL_NEAREST
);

// Especificamos que usaremos el de por defecto
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Ilustración 3.9 Ejemplo de copia de un buffer a otro

3.5 Implementación del Bump Mapping

Con respecto a la implementación de Bump Mapping, los únicos cambios que se realizan son en los shaders. Hasta ahora, en los shaders recibíamos la información de las normales y se pasaba a la etapa de los fragment shaders para realizar los cálculos de iluminación. En Forward Rendering y Forward Plus Rendering es el mismo proceso; para Deferred Rendering lo que se hacía era almacenar las normales en su respectivo framebuffer de color.

Para añadir Bump Mapping, las modificaciones que se hicieron fueron, primero, pasarle a los shaders tanto las tangentes como la bitangentes del objeto de igual forma que las posiciones o las normales; y después, calcular la matriz TBN en el vertex shader. Una vez que en el fragment shader recibe la matriz, comprueba si ese objeto tiene o no textura de normales (no todos los objetos vienen con esto). Si no lo tiene, se pasa las normales como siempre (ya sea en Forward Rendering y Forward Plus Rendering a los cálculos de luz o en Deferred Rendering al buffer de las normales). Sin embargo, si sí tiene, se calcula las nuevas normales con respecto a la matriz TBN y los desplazamientos especificados en la textura; realizando el mismo proceso como se hace normalmente, es decir, en Forward Rendering y Forward Plus Rendering lo usa para los cálculos de luz y en Deferred Rendering se lo asigna al correspondiente buffer de normales.

3.6 Implementación de los efectos postprocesado

Los efectos postprocesado van a parte del cálculo de luces. Se pueden afrontar de dos formas diferentes: o se realiza en cada shader de iluminación o se realiza de forma independiente. Como nuestro objetivo es buscar el mejor desacoplamiento, se ha optado por hacerlo aparte y así, independientemente del render que se use, los efectos serán los mismo.

Dado que HDR necesita de corrección gamma por los cambios en el brillo que produce, ambos procesos se realizan al mismo tiempo. Primero, se requiere que la escena se procese en un framebuffer propio. Así, el resultado se podrá obtener como textura. Después, lo único que se necesita es llamar al shader correspondiente (cambiando el framebuffer por el de defecto), pasarle la textura y hacer las operaciones descritas teóricamente.

4 DESARROLLO

Tal y como se ha explicado en el capítulo 2.6 “Metodología de desarrollo del software”, se ha escogido Scrum como metodología a seguir. En esta sección, se resumirá cada una de las iteraciones que ha habido en el proyecto y qué es lo que se hicieron en ellas. Serán resumidas ya que nos encontramos en un proyecto de teórico-experimental, y estas fases son secundarias. Cabe recordar que cada iteración supone a una semana de trabajo.

4.1 Primera Iteración

En la primera iteración, se sentaron las bases del proyecto. Se analizaron todos los requisitos que se pedían para el mismo, y se crearon los diagramas correspondientes. Además, se hicieron unos diseños de cómo sería la UI.

Posteriormente, se creó el proyecto en C++ y se enlazó junto con las librerías de Glew y GLFW. Se crearon las clases básicas como la de aplicación o la de escena y se buscó los modelos necesarios para poner a prueba el proyecto.

Por tanto, los sprint que se habían ideado para esta iteración fueron:

- Crear diagramas.
- Dar aprobación de los diagramas.
- Buscar modelos 3D para la investigación.
- Crear proyecto básico en C++.
- Storyboard de la UI.

4.1.1 Storyboard de la UI

La aplicación no requería de un storyboard complejo, ya que solo se quería poner una pequeña información en pantalla para no molestar con los resultados de los renders. Una cosa que sí se especificó es que dicha información se pudiera mover por toda la ventana para que el usuario lo pudiera poner conforme a su gusto. Por eso, se ideo que toda la información se pusiera en una ventana flotante.

En la ilustración 4.1, podemos apreciar el storyboard del proyecto, donde se especifica que tanto los valores gamma, como en número de luces, el número de objetos transparentes y el valor de exposición del HDR sean sliders que pueda mover el usuario a su gusto. Además, también hay un despegable donde aparecen los tres renders diferentes que tiene la aplicación. Así, el usuario podrá cambiar fácilmente entre ellos.

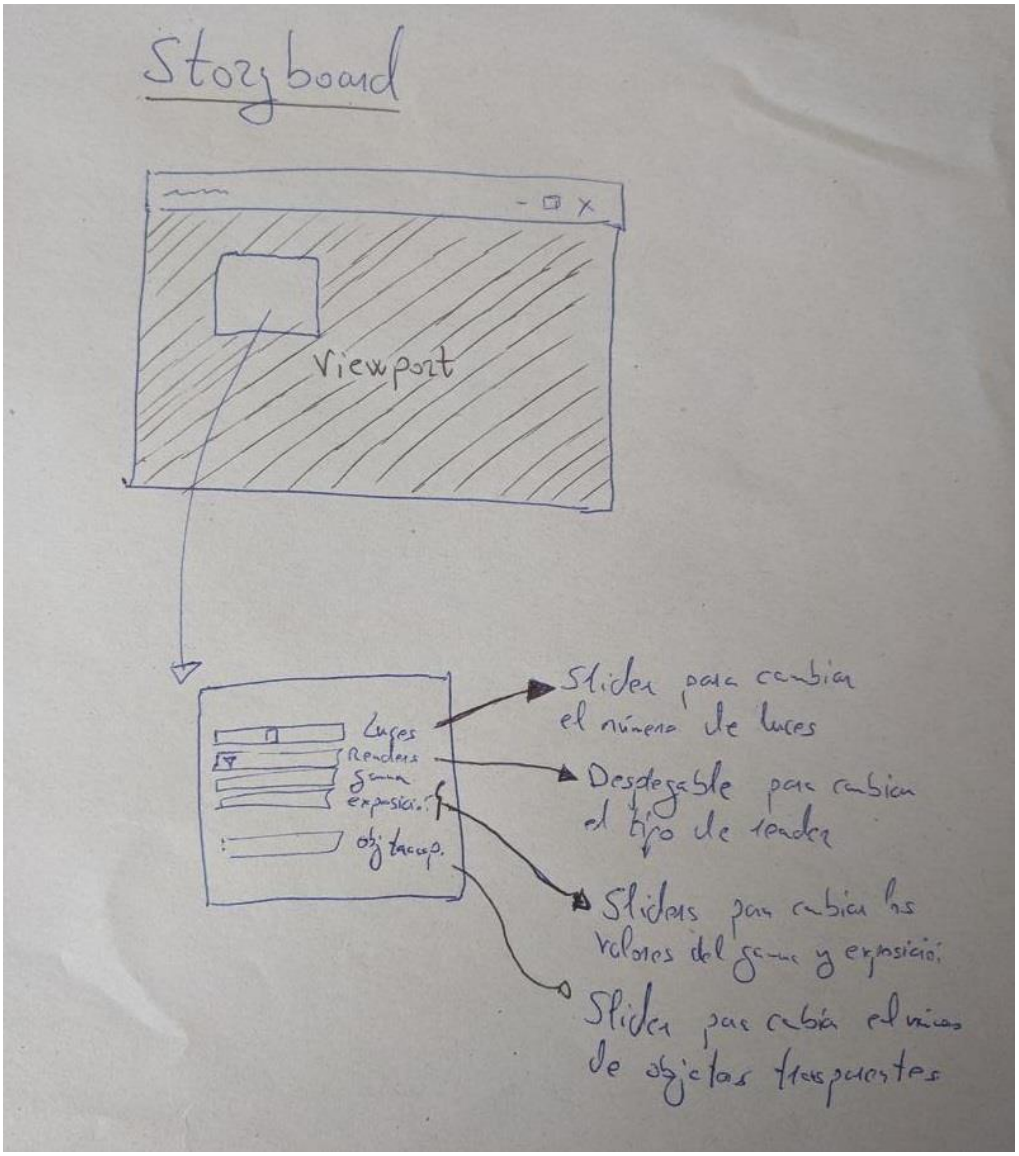


Ilustración 4.1 Storyboard de la UI

4.2 Segunda Iteración

En esta segunda iteración, entramos de lleno dentro del desarrollo. Se precisó que, para poder probar bien las escenas, era necesario crear un sistema con el que poder cargar objetos desde archivos de texto. Tras realizar una búsqueda sobre cuál era la mejor manera de cargar objetos con OpenGL, se llegó a la conclusión que Assimp era la librería idónea junto con las librerías “stb_imagen” y “DevIL” para el cargado de texturas y materiales. Además, como irían aumentando los shaders conforme fuera evolucionando el desarrollo, lo mejor era crear ya el shader manager para ir gestionando todos los shaders. Para probar todo, se creó un shader básico en el que solo pintaba los modelos con un color específico.

Entonces, los sprint que se habían ideado para esta iteración fueron:

- Crear sistema de cargado de objetos mediante Assimp y archivos de texto.
- Crear shader manager junto con la clase PagProgramShader ya probada en la asignatura “Graficos avanzados” de cuarto de carrera.
- Crear un shader básico para dibujar objetos cargados.
- Probar el cargado de modelos usando varios de ellos.

4.3 Tercera Iteración

En la iteración anterior, ya podíamos cargar los objetos que quisiéramos, pero no podíamos movernos por la escena. Por tanto, en esta iteración, se planeó el añadir tanto la cámara *“First Person Shooter”* (FPS) como las clases Render y Model. Con dichas clases, podíamos trasladar los objetos, rotarlos y escalarlos, perfecto para a prueba los experimentos cuando hubiera finalizado el proyecto; y la cámara, para poder ver con claridad los objetos desde cualquier parte de la escena.

Por tanto, los sprint que se habían ideado para esta iteración fueron:

- Crear clases Model y Render.

- Modificar el cargado de los modelos para que los objetos pudieran usar las clases que se habían creado.
- Crear cámara FPS.
- Probar cámara y manipulaciones de los objetos de la escena.

4.4 Cuarta Iteración

Una vez que podemos movernos libremente por la escena y manipular los objetos, se decidió que era el momento de implementar los nodos y los objetos escena. Para ello, se decidió que todas las etapas fueran independientes de los renders, y así no se tuviera que hacer muchas modificaciones.

Además de eso, también se creó el TextureManager dado que, si queremos cargar muchos objetos en pantalla, cargar muchas texturas también iba a suponer un problema de rendimiento. Por tanto, lo mejor era controlar las texturas y el cómo se cargaban. Para probar las texturas, se tuvo que modificar el shader básico de la segunda iteración para poder probarlas.

Por tanto, los sprint que se habían ideado para esta iteración fueron:

- Crear NodoScene y SceneObj.
- Modificar el cargado de los objetos para que sigan la estructura que se había diseñado.
- Crear TextureManager.
- Modifcar la clase Render para que usara las texturas.
- Modifcar los shaders para que usaran las texturas.
- Probar todos los cambios.

4.5 Quinta Iteración

Ya tenemos una escena que puede ser manipulable, y unas texturas que le dan algo de realismo. Antes de pasar a la parte crítica de la aplicación, se pensó que lo mejor era dejar todo el tema de las luces ideado e implementado. Por tanto, en esta quita interacción se implementó las luces y los cálculos de luces que se usarán en los tres renders que se implementarán posteriormente.

Por tanto, los sprint que se habían ideado para esta iteración fueron:

- Investigar sobre los calculos de iluminación.
- Modificación de los shaders para añadir los calculos de iluminación.
- Modificar la escena para pasarle la información de las luces a los shaders.
- Probar los objetos cargados con un par de luces diferentes.

4.6 Sexta Iteración

Una vez que el núcleo de la aplicación funciona, se pasó a la implementación de los renders. Se optó por empezar por Forward Rendering ya que estaba casi todo hecho, lo único que se tenía que pasar toda la lógica a una clase independiente. Además, se pensó que como se iban a implementar ya los renders, lo mejor era crear un desacople de los *“uniforms”* de los shaders.

Por tanto, los sprint que se habían ideado para esta iteración fueron:

- Crear clase ForwardRender y quitar la lógica que hubiera en la escena.
- Crear el método “SetUniforms” que se encargará de pasar todos los uniforms necesarios de los renders.
- Probar todos los cambios hechos.

4.7 Séptima Iteración

Una vez implementado el primer render, se pasó a implementar Deferred Rendering. Para esto, se tuvo que crear la clase DeferredShadingRender, donde contendría toda la lógica de esta técnica. Para ello, se tuvo que investigar sobre los buffers de colores y como adaptar los shaders a esta técnica.

Por tanto, los sprint que se habían ideado para esta iteración fueron:

- Investigar sobre los buffers de colores.
- Crear clase DeferredShadingRender.
- Crear los shaders “gBuffer” y “DeferredLighting”.
- Enlazar escena junto con el nuevo render.
- Probar todos los cambios hechos.

4.8 Octava Iteración

Junto con la investigación de los buffers, se descubrió los shared buffers. Estos buffers eran perfectos para poder poner a prueba los renders pasándole muchos datos. Por tanto, en esta iteración se enfocó en readaptar los renders a la nueva forma de pasarle la información.

Por tanto, los sprint que se habían ideado para esta iteración fueron:

- Investigar sobre los shared buffers.
- Readaptar el código de ForwardRender para los nuevos buffers.
- Readaptar el código de DeferredLighting para los nuevos buffers.
- Readaptar los correspondientes shaders.
- Probar los cambios hechos.

4.9 Novena Iteración

Una vez que se cambió la forma de pasar las luces a los shaders, era el momento perfecto para implementar Forward Plus. Para ello, se creó una clase que contuviera toda su lógica y se creó los respectivos shaders. Además, se empezó la investigación sobre cómo incorporar Bump Mapping a los tres renders y algunos efectos postprocesados como la corrección gamma o el HDR.

Por tanto, los sprint que se habían ideado para esta iteración fueron:

- Investigar sobre como incorporar Bump Mapping y los efectos postprocesado.
- Crear clase ForwardPlusRender y sus respectivos shaders.
- Enlazar el nuevo render a la escena.
- Probar los cambios hechos.

4.10 Décima Iteración

Recogiendo el trabajo de la etapa anterior, se procedió a incorporar los efectos postprocesado. Dado que ya se había avanzado sobre ese tema, se acordó añadir la GUI al sistema. Dado que es una ventana con un par de opciones, se esperaba que al finalizar esta iteración el proyecto estuviera listo.

Por tanto, los sprint que se habían ideado para esta iteración fueron:

- Incorporar la corrección gamma a los shaders.
- Incorporar HDR a los shaders.
- Crear clase de GUIManager.
- Reestructurar la escena para que GUIManager pudiera acceder a ciertos valores.
- Probar los cambios hechos.

4.11 Pruebas finales

Cuando finalizó el desarrollo del proyecto, se decidió realizar una última etapa donde probar toda la funcionalidad del proyecto y ver si reaccionaba acorde a lo estipulado en los diagramas y requisitos.

4.11.1 Pruebas de verificación del sistema

Se realizaron pruebas con diferentes modelos obtenidos de internet con más o menos geometría y cambiando todos los valores posibles en cada uno de ellos en búsqueda de fallos en el sistema.

Se detectaron pequeños bugs a la hora de realizar ciertas funciones como que al pulsar el botón que hace que puedas mover el ratón con libertad, no lo hacía siempre como uno se esperaba; las luces no iluminaban en todas las áreas requeridas o bugs visuales como que la UI no creaba la ventana correctamente. Todos estos bugs fueron arreglados y se volvieron a realizar las mismas pruebas. El sistema pasó por completo las pruebas.

Ilustración 4.2 Ejemplo visual de la aplicación

5 EXPERIMENTACIÓN, RESULTADOS Y DISCUSIÓN

Para los experimentos, hemos querido poner a prueba cada una de las técnicas en diferentes escenarios. Partiendo de que no se busca que la geometría de la escena pueda producir variaciones, todos los experimentos se harán sobre el mismo escenario previamente comentado en la sección 2.4 “Dataset de las pruebas”.

Ilustración 5.1 Resultado final de Deferred Rendering

5.1 Experimentaciones y pruebas

Para realizar los experimentos, hemos programado una escena que carga, aleatoriamente, tanto las luces como los objetos transparentes. Las luces, además, se van moviendo de arriba abajo. Las pruebas que se han realizado han sido las siguientes:

Prueba 1: Solo luces:

- Una escena con 20 luces.
- Una escena con 200 luces.
- Una escena con 1024 luces.

- Una escena con 2048 luces.

Prueba 2: Solo objetos transparentes (1 luces):

- Una escena con 5 objetos transparentes.
- Una escena con 30 objetos transparentes.
- Una escena con 50 objetos transparentes.

Prueba 3: Luces y objetos transparentes:

- Una escena con 20 luces con 5 objetos transparentes.
- Una escena con 200 luces con 5 objetos transparentes.
- Una escena con 1024 luces con 5 objetos transparentes.
- Una escena con 2048 luces con 5 objetos transparentes.

Prueba 4: Luces y objetos transparentes:

- Una escena con 20 luces con 30 objetos transparentes.
- Una escena con 200 luces con 30 objetos transparentes.
- Una escena con 1024 luces con 30 objetos transparentes.
- Una escena con 2048 luces con 30 objetos transparentes.

Prueba 5: Luces y objetos transparentes:

- Una escena con 20 luces con 50 objetos transparentes.
- Una escena con 200 luces con 50 objetos transparentes.
- Una escena con 1024 luces con 50 objetos transparentes.
- Una escena con 2048 luces con 50 objetos transparentes.

5.2 Resultados y discusión

De la prueba 1, cuyos resultados están en la ilustración 5.2, podemos ver que conforme se aumenta el número de luces, Forward Plus Rendering es el que mejor resultados obtiene. Deferred Rendering obtiene buenos resultados con relativa cantidad de luces, pero conforme estas aumentan, su rendimiendo cae considerablemente.

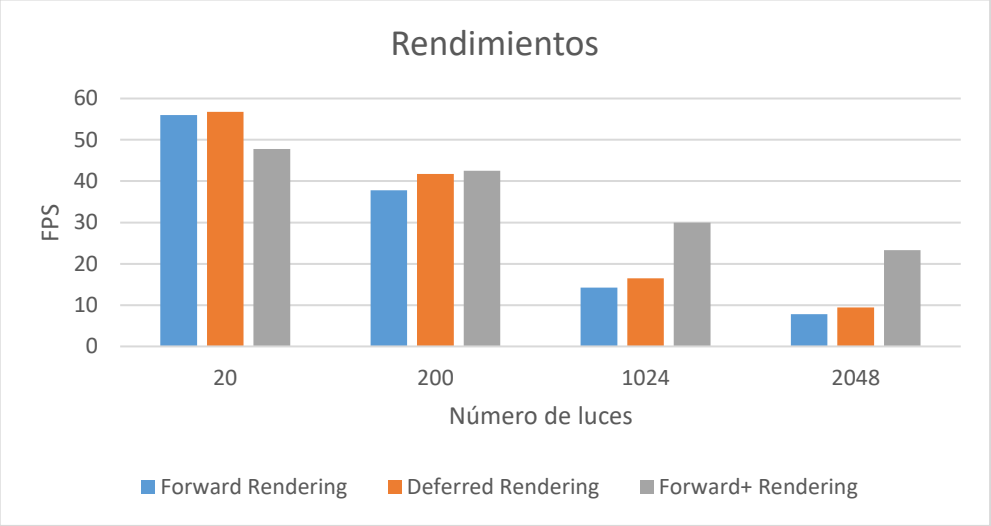


Ilustración 5.2 Rendimientos en la prueba 1

Ilustración 5.3 Muestra del Bump Mapping.

Durante la segunda prueba reflejada en la ilustración 5.4, buscábamos si Deferred Rendering sufría alguna variación en su rendimiento al añadir objetos transparentes. Dado que la prueba se realizó con una única luz, podemos ver que Forward Plus Rendering es la peor sale de la prueba. Tanto Forward Rendering como Deferred Rendering, no sufren al añadir objetos transparentes. Por tanto, la modificación que le hicimos para objetos transparentes no varía apenas el rendimiento del mismo. Y lo mismo ocurre tanto con Forward Rendering como con Forward Plus Rendering.

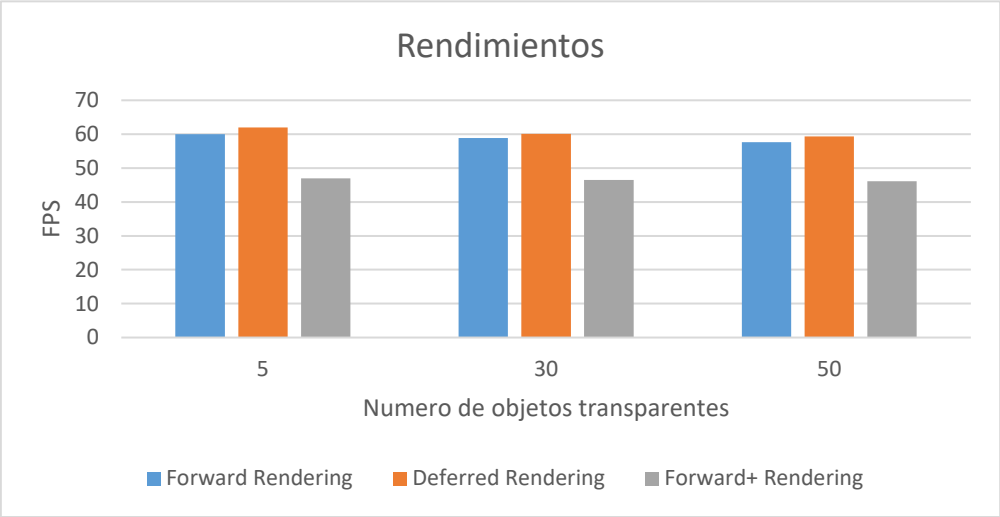


Ilustración 5.4 Rendimientos en la prueba 2

Ilustración 5.5 Ejemplo de objeto transparente

De las siguientes pruebas, representadas en las ilustraciones 5.6, 5.7 y 5.8, se analizarán de forma conjunta. De estas, podemos extraer que, aun aumentando el número de objetos transparentes, estas no sufren apenas variaciones. Como mucho, perderá en torno a unos 0.5 FPS.

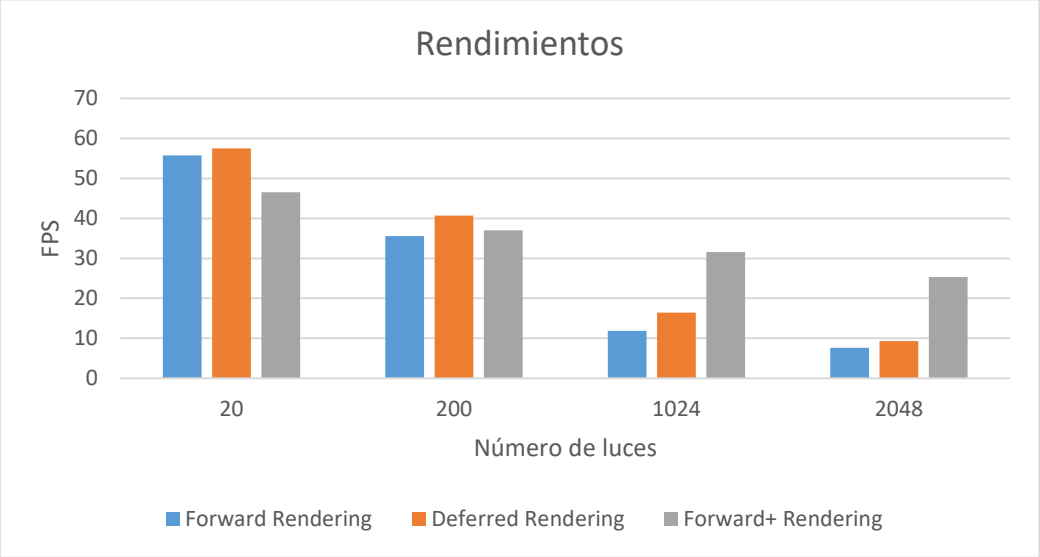


Ilustración 5.6 Rendimientos en la prueba 3

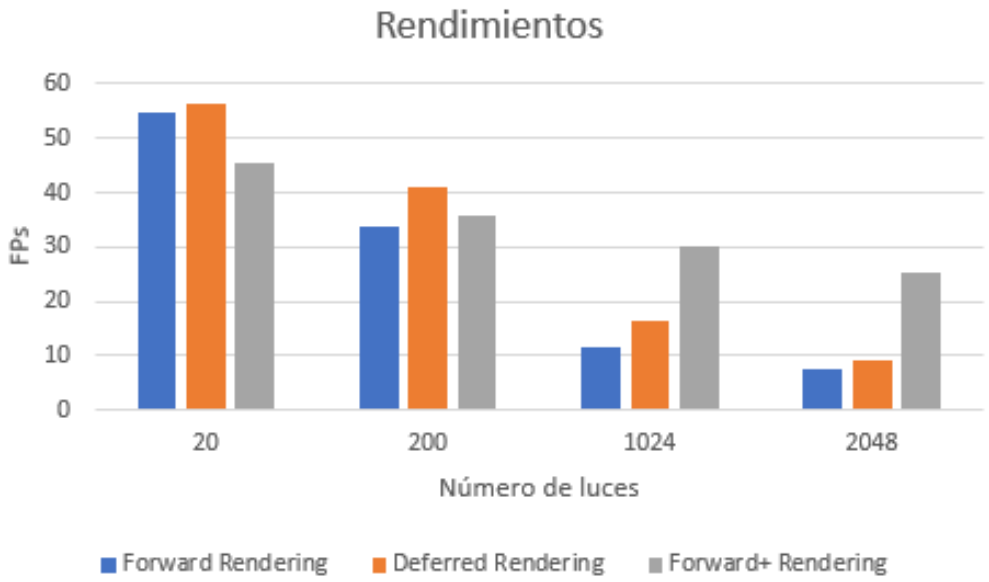


Ilustración 5.7 Rendimientos en la prueba 4

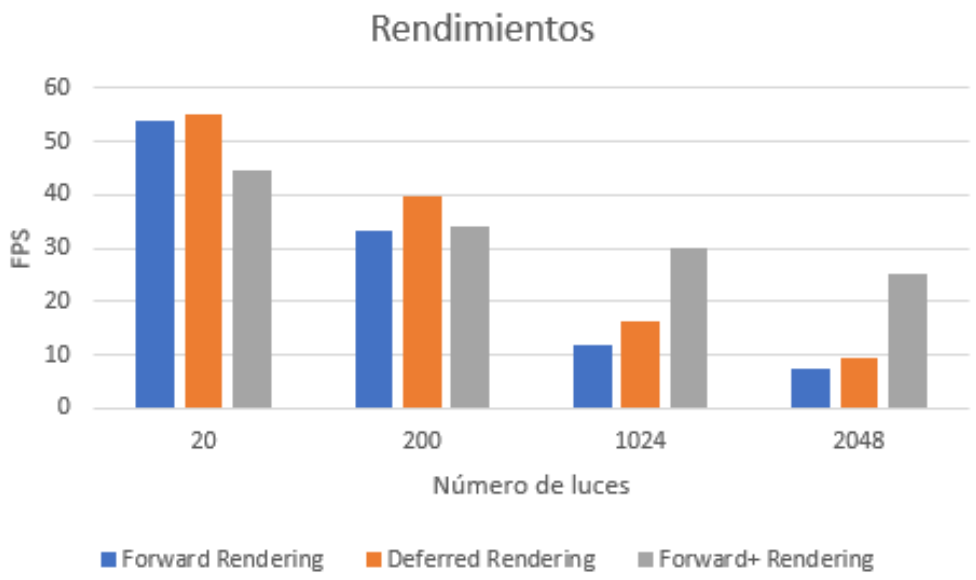


Ilustración 5.8 Rendimientos en la prueba 5

Podemos deducir, por último, es que Forward Rendering es el peor parado de todos. En cuanto a Deferred Rendering, se posiciona en el segundo lugar, obteniendo unos buenos resultados con relativamente bastantes luces. Aun así, podemos ver como conforme aumentamos de las 1024 luces, el rendimiento cae considerablemente; aunque no es del todo malo. Finalmente, Forward Plus Rendering es el que sale ganador en casi todas las pruebas, siendo una clara opción a elegir.

6 CONCLUSIONES Y TRABAJOS FUTUROS

Como conclusiones finales, podemos decir que tanto Deferred Rendering como Forward Plus Rendering son dos muy buenas opciones como alternativas al Forward Rendering. Ambos nos darán una un buen rendimiento con una gran cantidad de luces. Sin embargo, si nuestra intención es usar pocas luces, serán mejor opciones Deferred Rendering o incluso Forward Rendering. Si nuestra opción es primar la cantidad de luces que hay en la escena, la mejor opción sin duda es elegir Forward Plus Rendering.

Podemos concluir además que las tres no han presentado ningún problema a la hora de adaptar técnicas visuales, como puede ser HDR o el Bump Mapping. Pero, es importe añadir que Deferred Rendering te puede aportar un bien rendimiento, pero a cambio de un coste de memoria elevado según la geometría y la información que se quiera usar.

Como futuros trabajos, me planteo dar un paso más allá con respecto al tema de las luces. Existen técnicas de iluminado usando iluminación global en tiempo real. Esta, combinadas con cualquiera de las estudiadas en este proyecto, puede hacer que el resultado final tenga un realismo bastante considerable. Junto a eso, también me gustaría investigar sobre los trazados de rayos, que están de moda últimamente.

Por último, OpenGL ha sido la librería seleccionada para llevar a cabo el sistema 3D; pero no es la única. Vulkan ofrece un mejor rendimiento que OpenGL a cambio de un mayor nivel de desarrollo del proyecto. Ver la comparativa de un proyecto OpenGL y otro igual en Vulkan, puede ser interesante de cara a elegir la librería idónea para tu proyecto.

7 APÉNDICES

7.1 Guía original del Trabajo Fin de Título

Identificación: [19/20-2857]

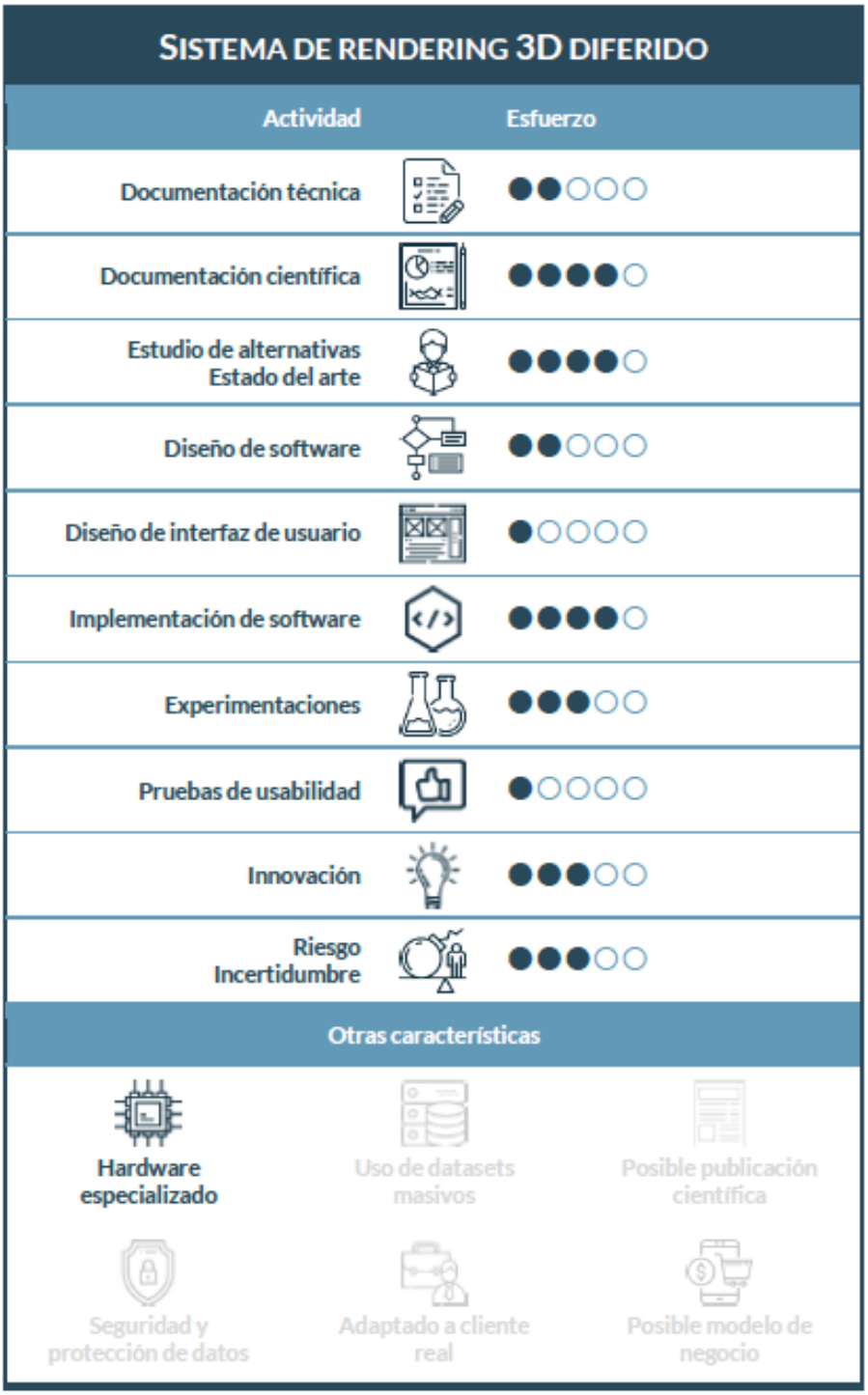
SISTEMA DE RENDERING 3D DIFERIDO

GRADO EN INGENIERÍA INFORMÁTICA

PROPUESTA DE TRABAJO FIN DE GRADO

CURSO 2018-2019

Ficha del Trabajo	
Titulación	Grado en Ingeniería Informática
Modalidad	Trabajo Teórico/Experimental
Especialidad (solo TFG)	Tecnologías de la Información
Mención (solo TFG)	Sistemas Gráficos
Idioma	Español
Tipo	Específico
TFT en equipo	No
Fecha de asignación	Haga clic aquí o pulse para escribir una fecha.
Asignado a	Alejandro Gemas Toribio
Tutores	Carlos Javier Ogáyar Anguita
Propuesta de Tribunal	
Presidente/a	Carlos Javier Ogayar Anguita
Secretario/a	Francisco de Asís Conde Rodríguez
Vocal	*Pendiente de asignación*
Suplentes	Juan José Jiménez Delgado
	Juan Roberto Jiménez Pérez
	Pendiente de asignación


SISTEMA DE RENDERING 3D DIFERIDO
Actividad
Esfuerzo
Documentación técnica
Documentación científica
Estudio de alternativas
Estado del arte
Diseño de software
Diseño de interfaz de usuario
Implementación de software
Experimentaciones
Pruebas de usabilidad
Innovación
Riesgo
Incertidumbre
Otras características
Hardware especializado
Uso de datasets masivos
Posible publicación científica
Seguridad y protección de datos
Adaptado a cliente real
Posible modelo de negocio

1 • Descripción

Este trabajo consiste en la realización de un **estudio teórico-práctico** acerca de la técnica de **rendering 3D diferido** (deferred rendering), que por lo general permite mayor rendimiento que el enfoque tradicional (forward rendering) con escenas que presentan una iluminación muy compleja. Debe hacerse un estudio teórico sobre la citada técnica, incluyendo los métodos de deferred lighting y deferred shading. Posteriormente debe realizarse un prototipo que utilice este método de rendering, basado en OpenGL, Vulkan, Metal u otra tecnología similar. El software debe permitir representar escenas 3D geométricamente complejas, con una gran cantidad de luces y en tiempo real.

2 • Conocimientos y requisitos previos

Es imprescindible tener conocimientos de OpenGL, Vulkan o Metal, así como dominar el desarrollo de aplicaciones gráficas en alguna de las plataformas actuales. También es recomendable tener conocimientos del pipeline gráfico típico, así como del uso de algún lenguaje de programación de shaders.

3 • Objetivos del trabajo

- **Realizar un estudio teórico** de la técnica de deferred rendering. Debe aplicarse a OpenGL, Vulkan, Metal u otra tecnología similar.
- Tratar a nivel teórico las **ventajas** de esta técnica frente al forward shading. Describir con detalle su funcionamiento y el procesamiento implicado en la GPU.
- Tratar a nivel teórico los **problemas** del deferred rendering, especialmente el tratamiento de múltiples materiales, la compatibilidad con métodos de antialiasing o la presencia de elementos transparentes en la escena.
- Seleccionar una plataforma para el desarrollo del prototipo, teniendo en cuenta las capacidades gráficas disponibles tanto a nivel de hardware como de software, el lenguaje de programación principal (Java, C++, Swing, etc.) y el API gráfico (ej.: OpenGL, Vulkan, Metal) que también implica un lenguaje de creación de **shaders**.
- **Diseñar e implementar un prototipo de aplicación** que muestre una escena 3D, preferiblemente cargada desde almacenamiento secundario en alguno de los formatos 3D más habituales (obj, 3ds, collada, etc.) a elección del alumno. Debe tenerse en cuenta que la geometría debe ser compleja y contener materiales y texturas que permitan la aplicación de técnicas para poner a prueba el sistema de iluminación (p.ej.: bumpmapping). Las condiciones de iluminación, así como los parámetros de los algoritmos implementados deberán ser configurables. La cámara debe ser controlable mediante una interacción básica.

- **Implementar el método de deferred rendering** concreto diseñado por el alumno, prestando especial atención a la estructura de datos y a los buffers utilizados en la GPU (G-Buffers) para conseguir el sombreado diferido.
- Realizar un **benchmarking** que permita comprobar la calidad de los resultados, así como el rendimiento obtenido.
- **Redactar las conclusiones del estudio** destacando los aspectos más interesantes sobre el método estudiado, la implementación realizada y los resultados obtenidos.

4 • Metodología de trabajo

- Elaborar una revisión bibliográfica del estado del arte sobre deferred rendering, incluyendo deferred lighting y deferred shading. Presentar las principales ventajas e inconvenientes de este método, así como una revisión de las soluciones más extendidas.
- Para el prototipo a realizar: seleccionar y utilizar una metodología basada en Ingeniería del Software para el análisis de requisitos, diseño, implementación, prueba y documentación. Se recomienda utilizar una metodología incremental.
- Detallar la estimación temporal y los costes de desarrollo.
- Completar el estudio teórico-experimental, incluyendo el desarrollo del prototipo.
- Generar una batería de pruebas (benchmarking) para el prototipo que cubra todos los aspectos implementados. Documentar convenientemente los resultados.
- Redactar la documentación final requerida, prestando especial atención a las conclusiones obtenidas con el estudio.

5 • Documentos y formato de entrega

- Para el formato de documentación de los **Trabajos teóricos o experimentales** se utilizará una documentación de formato libre. Tal y como especifican las normas de estilo de la EPSJ:

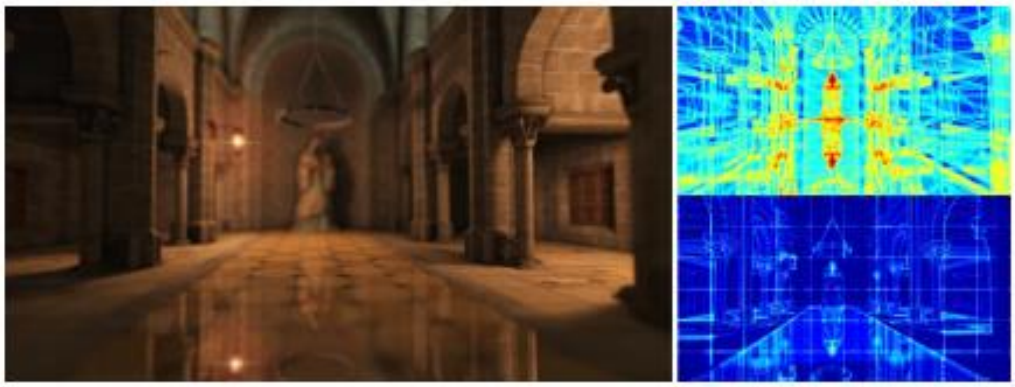
  '*Los Trabajos teóricos o experimentales se estructurarán, en la medida de lo posible, en los siguientes apartados: Introducción, Antecedentes, Objetivos, Material y Métodos, Resultados y Discusión, Conclusiones, Planos y Anexos (si proceden) y Bibliografía*'.

  No obstante, es conveniente incluir algunas secciones propias de los proyectos de Ingeniería, ya que complementan adecuadamente el trabajo. Estas secciones pueden incluir, entre otros y sin carácter limitativo: normas y referencias, definiciones y abreviaturas, alcance del trabajo, especificación del prototipo realizado, planificación temporal y presupuesto.

- Memoria del trabajo, incluyendo manuales y anexos, en formato PDF.

- Código fuente completo y ejecutables del prototipo o software desarrollado.
- Documentos y archivos adicionales a los que se haga mención expresa en la memoria.
- Fuentes de datos utilizadas. De especial importancia en trabajos teóricos o experimentales y estudios técnicos, donde es imprescindible contar con los datasets utilizados (incluidos o referenciados).
- Vídeo de demostración de la aplicación o de la experimentación realizada.
- Anexos para la presentación del trabajo:
  - Informe favorable del tutor.
  - Autorización de publicación, si procede.

6 • Algunos trabajos de ejemplo



7.2 Instalación y configuración del sistema

Durante el desarrollo de la aplicación, se ha necesitado usar librerías de terceros para el desarrollo del mismo. Esto quiere decir, que han sido compilados y añadidos al proyecto. Por tanto, para comodidad del usuario, dichas librerías se han incorporado en el proyecto final con tal de que no tenga que realizar ninguna compilación.

El único requisito es tener un sistema operativo de 64 bits bajo una distribución de Windows. Si el usuario está bajo un ordenador de sobremesa, no deberá de hacer ninguna modificación extra. Si el usuario está bajo un portátil, probablemente la aplicación sea ejecutada bajo la gráfica interna. Si se desea que la ejecute la segunda gráfica, deberá de especificarlo en el panel de administración de tarjeta gráfica.

7.3 Manuales de usuario

El manual de usuario de la aplicación es el siguiente:

- Pulsando W la cámara se moverá hacia adelante.
- Pulsando A la cámara se moverá hacia la derecha.
- Pulsando D la cámara se moverá hacia la izquierda.
- Pulsando S la cámara se moverá hacia atrás.
- Moviendo el ratón podrá mover hacia donde mira la cámara.
- Pulsando F1, el ratón será liberado para que pueda modificar la ventada de datos. Si vuelve a pulsar F1, el ratón quedará “atrapado” en el centro para que pueda mover hacia donde mira la cámara.

8 BIBLIOGRAFÍA

Adriana Gómez, M. d. (s.f.). *UN MODELO DE ESTIMACION DE PROYECTOS DE SOFTWARE.*

ASPgems. (2019). *Metodología de desarrollo de software*. Obtenido de https://aspgems.com/category/metodologia/

Blinn, J. F. (1978). *Simulation of Wrinkled Surfaces.* New York, NY, USA: Association for Computing Machinery.

Boehm, B. W. (1981). *Software Engineering Economics .* Prentice Hall.

Deering, M. e. (1988). *The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics* (Vol. 22). New York, NY, USA: Association for Computing Machinery.

Guha, S. a. (2015). *Computer graphics through openGL : from theory to experiments.* Boca Raton : CRC Press, Taylor & Francis Group.

Harada, T. a. (2012). *Forward+: Bringing Deferred Lighting to the Next Level.* The Eurographics Association.

Idri, A. A. (2000). *COCOMO cost model using fuzzy logic* (Vol. 27).

Movania, M. M. (2013). *penGL development cookbook over 40 recipes to help you learn, understand, and implement modern OpenGL in your applications.* Birmingham : Packt Pub.

Owens, B. (28 de Octubre de 2013). *Forward Rendering vs. Deferred Rendering*. Obtenido de https://gamedevelopment.tutsplus.com/articles/forward-rendering-vs-deferred-rendering--gamedev-12342

Pressman, R. (2010). *Ingeniería del Software.* McGraw-Hill.

Valient, M. (26 de Julio de 2007). *Deferred Rendering in Killzone 2*. Obtenido de Guerrilla Games: https://www.guerrilla-games.com/read/deferred-rendering-in-killzone-2

Vries, J. d. (s.f.). *LearnOpenGL*. Obtenido de https://learnopengl.com/