Universidad Zaragoza

1542

Trabajo Fin de Grado

Operador de *tone mapping* en tiempo real

Autor

Miguel Marín Crespo

Director

Adolfo Muñoz Orbañanos

Escuela de Ingeniería y Arquitectura - Universidad de Zaragoza

2015

A mi familia,

por todo su apoyo día a día.

A mis compañeros,

por todo este camino y buenas experiencias.

A Adolfo Muñoz, por su colaboración.

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

D./Dª. Miguel Marín Crespo,

con nº de DNI 77133317B en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado, (Título del Trabajo) Operador de tone mapping en tiempo real,

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 19 de noviembre del 2015

Fdo: Miguel Marín Crespo

Resumen ejecutivo

El rango dinámico de una imagen define la diferencia entre los puntos de mayor y menor luminancia representados. Las imágenes *HDR* o de alto rango dinámico permiten mostrar una riqueza de información mucho mayor que las imágenes *LDR* o de bajo rango dinámico. Esta riqueza de información es superior en muchas órdenes de magnitud a lo que es capaz de reproducir un dispositivo de visualización estándar actualmente. Para poder visualizar este tipo de información se aplican operadores de *tone mapping* o mapeo de tonos, que transforman imágenes *HDR* en formato *LDR* para que puedan ser visualizadas.

Este trabajo fin de grado comprende la investigación y el desarrollo de un operador de *tone mapping* más avanzado que los operadores habitualmente utilizados, capaz de funcionar a tiempo real sobre una escena *HDR*. La implementación del operador utiliza comunicación con la *GPU*, ya que las tarjetas gráficas permiten realizar cálculos de iluminación y visualización en *HDR*.

Este hecho permitiría que se pudieran realizar videojuegos utilizando tecnología *HDR*, pero no podrían visualizarse actualmente sin transformarse previamente su rango dinámico. El estado del arte en videojuegos hoy en día se sitúa en compresiones del rango dinámico utilizando curvas *gamma*.

Tras la realización de este trabajo se ha logrado implementar un operador de *tone mapping* más avanzado que el estado del arte actual, capaz de aplicarse sobre una escena *HDR* variable a tiempo real, pudiendo además modificar diversos parámetros subjetivos que consiguen diferentes resultados visualizables dependiendo de la elección del usuario.

Índice

Índice de figuras

Índice de cuadros

1. Introducción

En este documento se expone toda la información recopilada y analizada en la realización del Trabajo Fin de Grado *'Implementación de un algoritmo de tone mapping en tiempo real'*.

A continuación se introducen brevemente conceptos relacionados con el trabajo desarrollado, incluyendo el contexto tecnológico del proyecto (Apartado 1.1), objetivos y alcance (Apartado 1.2), y la estructura del presente documento (Apartado 1.3).

1.1. Contexto tecnológico

En este apartado se presenta el contexto relacionado con el ámbito de este trabajo, el cual abarca conceptos sobre el alto rango dinámico (Apartado 1.1.1), los operadores de *tone mapping* (Apartado 1.1.2) y la programación en *GPU* (Apartado 1.1.3).

1.1.1. Alto rango dinámico

El rango dinámico de una escena o imagen mide la relación entre la luminancia máxima y mínima de la escena representada, siendo la luminancia la medida para cuantificar la cantidad de luz que se ve reflejada en un punto de la escena. Cuando una escena posee una diferencia muy grande entre sus valores de luminancia, se habla de una escena de alto rango dinámico (en inglés *High Dynamic Range*, de siglas *HDR*). Cuando la diferencia entre los valores de luminancia no es amplia se habla de escenas de bajo rango dinámico (en inglés *Low Dynamic Range*, de siglas *LDR*).

En el presente documento se hablará a partir de este punto de escenas HDR y LDR para referirse a escenas de alto y bajo rango dinámico, respectivamente.

El ojo humano es capaz de percibir un rango dinámico mucho mayor al de los dispositivos sobre los que se representan imágenes o vídeos actualmente: la pantalla de un móvil, una tablet, un monitor o una televisión. La idea principal de una escena *HDR* se basa en la capacidad de

Figura 1.1: *Máximos valores de luminancia para varias escenas, en candelas por metro cuadrado.*

representar un mayor rango de niveles lumínicos en las escenas.

El rango dinámico de una escena depende de su resolución o cantidad de espacio destinado a almacenar información para cada píxel. Esta resolución es lo que diferencia a una escena *HDR* de una *LDR*, ya que una escena *HDR* utiliza 32 bits por canal en coma flotante para almacenar información por cada píxel, pudiendo así almacenar un rango de valores varias órdenes de magnitud más amplio que en escenas *LDR*, las cuales utilizan 8 bits por canal (un rango de 256 posibles valores) y no guardan valores flotantes. Este rango más amplio permite almacenar mayor cantidad de información en una escena *HDR*. Esta resolución se puede cuantificar en el número de pasos que la escena (o el dispositivo de visualización) es capaz de diferenciar entre el punto más luminoso y el más oscuro (ver Figura 1.2).

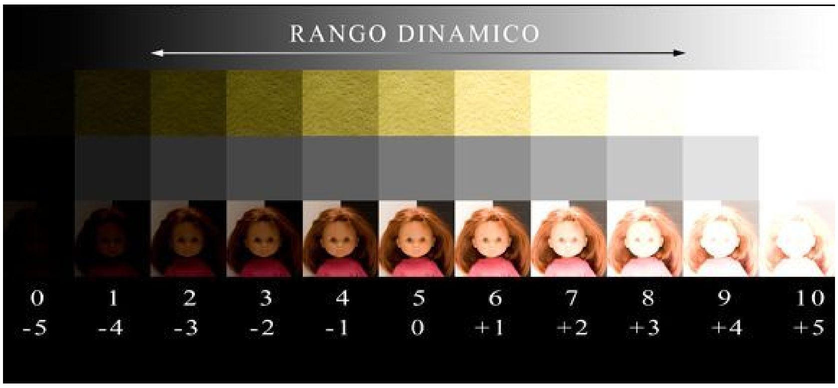


Figura 1.2: *Diferentes pasos de rango dinámico en una escena. En la escena representada se pueden apreciar diez niveles diferentes de luminancia.*

Actualmente no es posible representar imágenes *HDR* en un dispositivo convencional, por lo que es necesaria una conversión del rango dinámico para poder visualizarla. Para realizar esta conversión se utilizan operadores de *tone mapping* o mapeo de tonos (estos operadores se explican más en detalle en los Apartados 1.1.2 y 2.3).
Cuando se habla de datos *HDR* se puede hablar de fotografías editadas, de imágenes generadas sintéticamente, o de escenas creadas mediante motores de render para videojuegos; este último caso es el que interesa para la realización de este trabajo.

1.1.2. *Tone mapping*

Debido a las limitaciones de los dispositivos actuales (la pantalla de un móvil, de un ordenador, una televisión…), no es posible hoy en día visualizar contenido *HDR* mostrando todo su rango de luminancias. Por tanto, este rango dinámico necesita ser reducido, convirtiendo la escena a una resolución de 8-bits por canal (ver Apartado 1.1.1) antes de poder ser representada en un dispositivo.

Los operadores de *tone mapping* (mapeo de tonos en español) son los encargados de realizar esta reducción para intentar emular la visión humana. Este tipo de técnicas también son comúnmente utilizadas para producir imágenes conservando o exagerando el contraste localmente con el fin de conseguir diversos efectos artísticos. En la Figura 1.3 se muestran algunos ejemplos de utilización de operadores.

Este proyecto se basa en implementar una de estas técnicas que permita aplicarse sobre la escena **en tiempo real** pasando de una escena *HDR* a otra *LDR*. Este operador se explica en detalle en el Apartado 2.4.

En el presente documento se utilizarán a partir de ahora las siglas TMO para referirse a un operador de tone mapping.

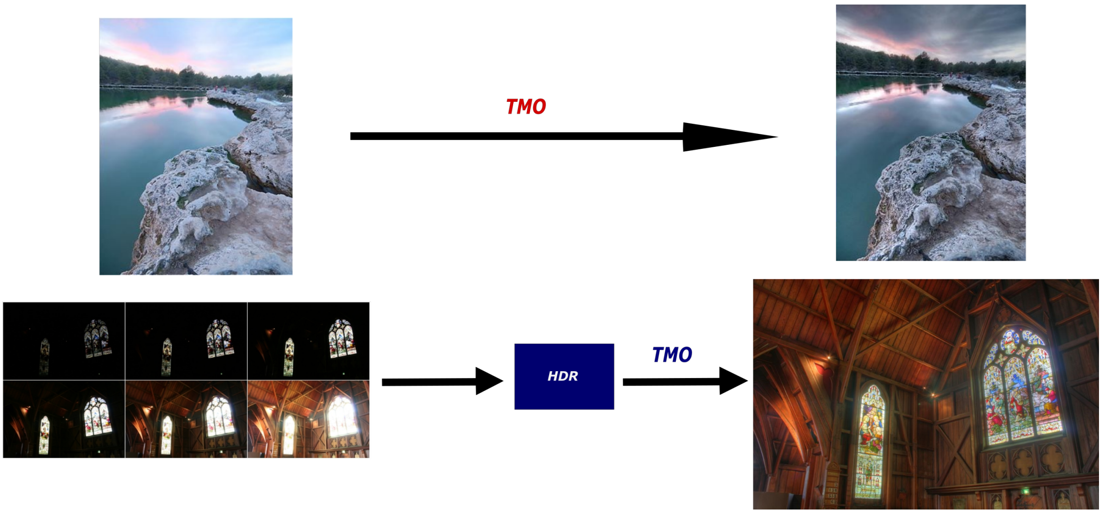


Figura 1.3: *Efectos derivados de la utilización de TMOs. A la imagen LDR superior se le aplica un TMO para aumentar su contraste. El TMO de la parte inferior permite visualizar en formato LDR la imagen HDR creada a partir de varias imágenes similares con distintos valores lumínicos.*

1.1.3. Programación en *GPU*

Desde hace varios años se ha utilizado la Unidad de Procesamiento Gráfico (*Graphics Processing Unit* o *GPU* en inglés) como *hardware* en la programación de herramientas relacionadas con los gráficos, debido a sus ventajas respecto a la programación íntegramente en la Unidad Central de Procesamiento (*Central Processing Unit* o *CPU* en ingles).

En el presente documento se hablará a partir de este punto de CPU y GPU para referirse a la Unidad Central de Procesamiento y a la Unidad de Procesamiento Gráfico de un ordenador, respectivamente.

La mayor diferencia entre una *CPU* y una *GPU* se entiende por la forma en que procesan las tareas: mientras que una *CPU* está formada por varios núcleos optimizados para el procesamiento en serie, una *GPU* consta de millares de núcleos más pequeños y eficientes diseñados para ejecutar múltiples y pequeñas tareas simultáneamente en paralelo. La programación con *GPUs* tiene sus ventajas respecto a las *CPUs* en cálculos que puedan realizarse mediante paralelismo, lo cual puede ser interesante en operaciones con gráficos, como se realiza en este trabajo. Utilizar la *GPU* también tiene sus inconvenientes respecto a la *CPU*, como la complejidad de su utilización, o la falta de continuidad en sus arquitecturas. Debido a la rápida y constante evolución del *hardware* gráfico, implementaciones de algoritmos que funcionaban óptimamente en un modelo de *GPU* funcionan subóptimamente, o incluso dejan de hacerlo en modelos superiores.

En los últimos años las *GPUs* han pasado de ejecutar una parte de las etapas de la denominada *pipeline* gráfica (explicada en los Apartados 3.2.1 y B.1 del presente documento) a poder incluso programar esta secuencia de etapas, lo cual facilita mucho las cosas a la hora de comunicarse con la *GPU*. En la programación en *GPU* toman un importante papel los *shaders*, pequeños programas de apenas unas líneas de código que se pueden compilar independientemente, y cuyos lenguajes son de alto nivel, lo que los hace independientes del *hardware* (ver Apartado 3.2.3 del presente documento para obtener más información sobre los *shaders* y su uso en este proyecto). Estos lenguajes dependen de la *API* utilizada como enlace (generalmente DirectX [1] u OpenGL [2]).

En este proyecto se ha utilizado la *API* OpenGL (ver Apartado A.1.2) junto con el lenguaje de *shading* GLSL (ver Apartado A.1.3).

1.2. Objetivos y alcance del proyecto

El objetivo final de este trabajo es implementar un *TMO* capaz de funcionar **a tiempo real**. Esta implementación engloba tanto el desarrollo del algoritmo como la obtención de los parámetros necesarios para realizarlo, y la representación de la escena final.

Los objetivos específicos de este proyecto son:

- Estudiar el concepto de *tone mapping*, las técnicas y los operadores más utilizados.
- Desarrollar un algoritmo que implemente un *TMO* sobre una escena *HDR* que se aplique a

tiempo real.

- Integrar el algoritmo dentro de un sistema que genere *HDR* en tiempo real, represente el efecto del *TMO* sobre la escena *HDR* generada y visualice la escena *LDR* final.

1.3. Estructura del documento

Esta memoria se ha dividido en 5 secciones o capítulos que recogen la información relacionada con este proyecto.

La sección actual (Sección 1) sirve como introducción a los temas relacionados con el proyecto, mientras que las siguientes explican en detalle el trabajo realizado. En la Sección 2 se presenta un análisis de los conceptos relacionados con el proyecto, siguiendo con una especificación del diseño de la aplicación y la implementación del operador en la Sección 3. Por último, se muestran los resultados obtenidos tras la realización del presente trabajo (ver Sección 4), así como una última sección (Sección 5) exponiendo las conclusiones tras la realización del trabajo, así como una valoración personal.

Adicionalmente se ha completado la información de este documento con una serie de anexos que relacionan temas complementarios a la realización de este proyecto: aspectos relacionados con el diseño de la aplicación (Anexo A), aspectos técnicos de OpenGL (Anexo B), conceptos sobre espacios de color (Anexo C), así como la gestión del proyecto (Anexo D).

2. Análisis

En este capítulo se realiza un análisis del trabajo realizado. En primer lugar se enumeran los requisitos del proyecto, tanto funcionales (Apartado 2.1) como no funcionales (Apartado 2.2). Seguidamente se presentan algunos operadores de *tone mapping* (Apartado 2.3), para terminar analizando el operador implementado en este trabajo (Apartado 2.4).

2.1. Requisitos funcionales

Los requisitos funcionales son aquellos que determinan el comportamiento de un sistema.

Los requisitos funcionales para este trabajo son los siguientes:

- Implementar un algoritmo que aplique un *TMO* capaz de funcionar a tiempo real sobre una escena dinámica representada en *HDR*.
- Mostrar la escena *LDR* resultante de aplicar el algoritmo sobre una escena *HDR* dinámica.
- Habilitar al usuario la posibilidad de cambiar ciertos parámetros del algoritmo para observar distintos efectos del *TMO* sobre la escena en tiempo real.

2.2. Requisitos no funcionales

Los requisitos no funcionales son aquellos que determinan el diseño e implementación de un sistema.

Los requisitos no funcionales para este trabajo son los siguientes:

- Hacer uso del procesador gráfico para minimizar tiempos de cálculo en tiempo real.
- Estudiar y aprovechar las posibilidades que ofrece OpenGL en cuanto a programación de gráficos y en *GPU*.
- Lograr que el algoritmo alcance una tasa mayor que 24 *frames* por segundo, lo que se define como tiempo real para el ojo humano.

2.3. Operadores de *tone mapping*

Como se ha explicado previamente en el Apartado 1.1.2, los operadores de *tone mapping* o *TMOs* transforman información *HDR* en *LDR* para que esta información pueda ser visualizada. Estos operadores pueden ir desde un escalamiento lineal del rango de la escena, hasta sofisticadas aproximaciones multi escalares que intentan emular fielmente la visión humana. Muchos *TMOs* están compuestos de un gran número de operaciones por píxel independientes unas de otras, lo que los hace idóneos para una implementación en *GPU*.

2.3.1. Tipos de operadores

Una de las formas más sencillas de implementar un *TMO* consiste en realizar un escalado lineal de la luminancia de la escena sobre la del dispositivo de visualización. Esta implementación es muy simple y sencilla de realizar, pero no suele generar escenas perceptualmente agradables.

El **estado del arte** actual en videojuegos para aplicar *TMO* a tiempo real se basa en aplicar una **curva *gamma*** (letra griega γ) sobre el escalado de la imagen. Esta curva consiste en un escalamiento no lineal del rango dinámico de la escena, variando la luminancia a partir de la Ecuación (2.1).

$$L_{out} = A \cdot L_{in}^{\gamma} \tag{2.1}$$

donde A es una constante y las entradas y salidas son valores reales no negativos. En el caso común ($A = 1$) las entradas y salidas se encuentran entre 0 y 1. Un valor de γ menor que uno se suele denominar gamma de codificación, mientras que un valor de γ mayor que uno se suele denominar gamma de decodificación. En la Figura 2.1 se puede apreciar gráficamente el efecto de la aplicación de esta curva en una escena.

El algoritmo creado para este trabajo implementa un *TMO* algo más complejo que el actual

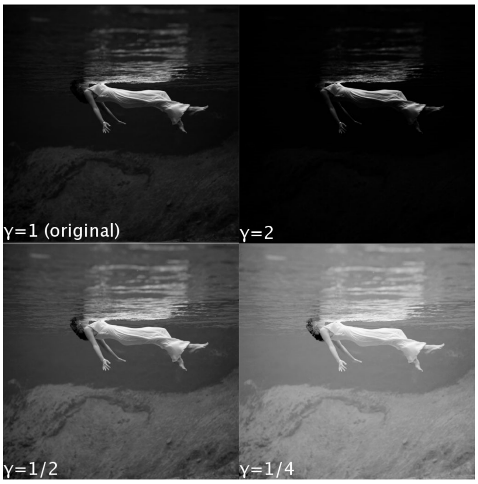


Figura 2.1: *Aplicación de diferentes valores de curvas gamma a una escena*

estado del arte, manteniendo la posibilidad de poder variar su efecto sobre la escena **a tiempo real**.

A continuación se enumeran brevemente algunos *TMOs* (ver Cuadro 2.1). Existen muchísimos en la actualidad y no es objetivo de este proyecto el análisis en profundidad de varios modelos, sino la implementación de uno en concreto: el operador global de Reinhard [7].

TMO [autor, año]	**Descripción**
Histogram adjustment [Ward, 1997] [4]	Este operador intenta preservar la visibilidad en la escena intentando ajustar el histograma.
Bilateral Filter [Durand and Dorsey, 2002] [5]	Operador local que intenta visualizar la escena por descomposición utilizando filtro de borde de preservación (*bilateral filtering*).
Photographic Reproduction [Reinhard, 2002] [7]	Simula la técnica *dodging and burning* usada en fotografía tradicional, permitiendo diferentes exposiciones. Se ha elegido su versión global para implementar en este proyecto.
Logarithmic Mapping [Drago, 2003] [6]	Este operador reduce el ratio del contraste utilizando una compresión logarítmica de los valores de luminancia, imitando la respuesta humana a la luz.

Cuadro 2.1: *Operadores de tone mapping.*

Uno de los *TMOs* más populares es el de Reinhard [9] (año 2002), cuya versión global es la implementada en este trabajo; en la siguiente sección se analizan en detalle los cálculos teóricos del operador.

2.4. Fundamentos teóricos del *TMO* implementado

Para la realización de este trabajo se ha implementado el *TMO* global de Reinhard [9]. Este operador es de los más conocidos por su simplicidad y efectividad. La idea principal consiste en escalar la luminancia de la escena, partiendo de la obtención de la luminancia media y la luminancia máxima.

A continuación se explica el algoritmo paso a paso. En la Figura 2.2 se puede apreciar un esquema de las fases del algoritmo.

Figura 2.2: *Esquema de las fases del TMO implementado*

2.4.1. Cálculo de la luminancia para cada píxel

En esta primera fase del operador se obtiene la luminancia de cada píxel de la escena. Para poder obtener valores de luminancia es necesario trabajar con un modelo de color que utilice este tipo de valores, por lo que es necesaria una conversión entre el espacio de color inicial de la escena a otro espacio que tenga en cuenta la luminancia. Por ello, antes de empezar a detallar el operador se procede a explicar brevemente el concepto de espacio y modelo de color, y los utilizados en este algoritmo.

Modelos y espacios de color

Un modelo de color es una fórmula matemática abstracta que describe cómo se representan los colores. Para ello, se basa en tuplas numéricas compuestas normalmente por tres o cuatro valores o componentes de color.

Un espacio de color es un sistema de interpretación del color, es decir, una organización específica de los colores en una imagen o vídeo. Depende del modelo de color en combinación con los dispositivos físicos que permiten las representaciones reproducibles de color.

En esta primera fase del operador, la escena inicialmente está en formato *RGB*, quizás el modelo de color más conocido y utilizado (siglas de *Red Green Blue*, en español Rojo Verde Azul). Este modelo de color está basado en la síntesis aditiva, con la que es posible representar un color mediante la mezcla por adición de los tres colores de luz primarios. El problema de este modelo de color respecto a los cálculos del algoritmo es que no tiene en cuenta la luminancia; por tanto, es necesaria una conversión a otro modelo de color que sí la tenga en cuenta.

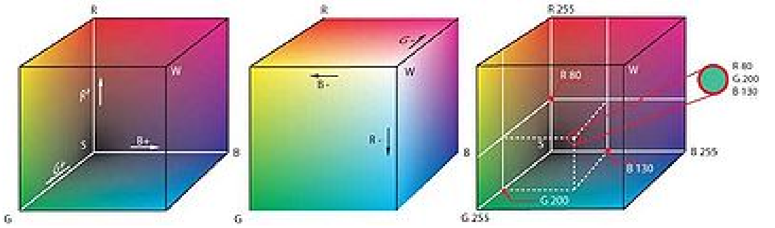
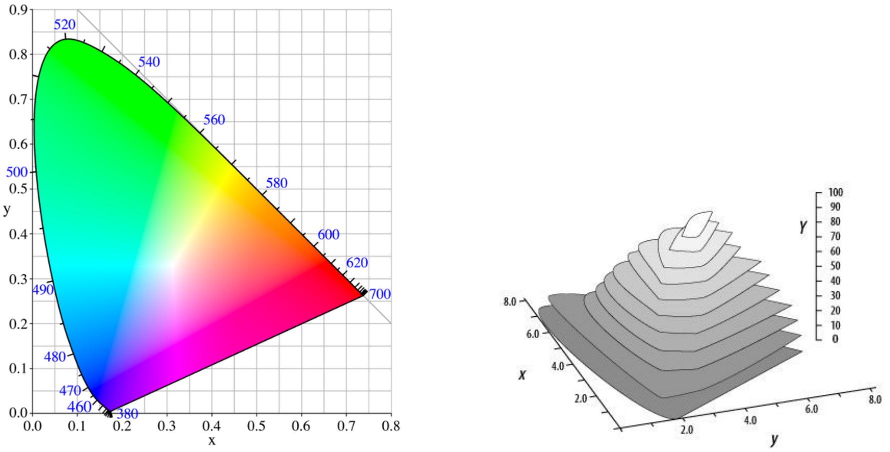


Figura 2.3: *Esquema del modelo de color RGB. En este cubo cada coordenada espacial representa la cantidad del color correspondiente.*

El modelo de color elegido es el XYZ (también relacionado con el modelo similar denominado xyY) [10], modelo de referencia para definir los colores que percibe el ojo humano; se basa en tres colores primarios que representan el color utilizando dos parámetros (X y Z), mientras que el tercer parámetro Y representa en este espacio la luminancia. En la Figura 2.4 se observa gráficamente el esquema de este modelo de color.



(a) Diagrama de cromaticidad (b) Influencia de la luminancia Y

Figura 2.4: *Esquema del modelo de color XYZ. Como se puede observar en la Figura b, la luminancia se representa sobre el diagrama de cromaticidad como una tercera dimensión z, mientras que en los espacios horizontal y vertical (ejes x e y) se define la cromaticidad, parámetros X y Z del espacio.*

Nota: Complementariamente a lo explicado con anterioridad, en el Anexo II del presente documento se explican los diferentes espacios y modelos de color utilizados para este proyecto con mayor detalle.

Una vez comentados brevemente estos conceptos, se procede a explicar la primera fase del algoritmo, en la que es necesaria la conversión de un espacio a otro.

Cálculo del algoritmo

El operador empieza con el cálculo de la *key* de la escena, que indica el brillo total subjetivo de la escena. La Ecuación (2.2) representa la conversión del espacio RGB al espacio XYZ, cuyo término Y es el que define la luminancia. Este término es el que en las sucesivas ecuaciones del algoritmo se denomina Lw:

$$XYZ = \begin{bmatrix} 0{,}4124 & 0{,}3576 & 0{,}1805 \\ 0{,}2126 & 0{,}7152 & 0{,}0722 \\ 0{,}0193 & 0{,}1192 & 0{,}9505 \end{bmatrix} \cdot RGB$$

$$Y = L_w(x,y) = 0,2126 * R + 0,7152 * G + 0{,}0722 * B \tag{2.2}$$

donde Lw representa la luminancia para el píxel de coordenadas (x, y), siendo x un valor entre 1 y el número de píxeles de la escena en el eje horizontal, mientras que y es un valor comprendido entre 1 y el número de píxeles de la escena en el eje vertical. Las variables R, G y B representan los valores de color rojo, verde y azul de la escena en formato RGB, respectivamente.

Una vez calculada la luminancia para cada píxel, se procede a calcular la *key* de la escena, igual a la luminancia media logarítmica de la escena (denominada Lm_w):

$$Lm_w = e^{\frac{1}{N}\cdot\sum_{x=1}^{x=n}\sum_{y=1}^{y=m} log(\delta+L_w(x,y))} \tag{2.3}$$

donde Lm_w es la luminancia media logarítmica de la escena, calculada a partir del valor de la luminancia media para cada píxel (denominado L_w y calculado en la Ecuación (2.2)) y un pequeño valor denominado δ utilizado a modo de *offset* para evitar singularidades que pudieran ocurrir si apareciesen píxeles totalmente oscuros o negros en la escena (la ecuación, en tal caso, estaría intentando calcular $log(0)$). Cabe destacar que se trata de un cálculo realizado para todos los píxeles de la imagen, de coordenadas (x, y), siendo x un valor entre 1 y n, que representa el número de píxeles de la escena en el eje horizontal, mientras que y es un valor comprendido entre 1 y m, que representa el número de píxeles de la escena en el eje vertical.

Para el *TMO* implementado también se calcula la luminancia máxima de la escena.

$$L_{max} = max[L_w(x, y)] \tag{2.4}$$

2.4.2. Mapeo de la luminancia para cada píxel

Una vez se tiene Lm_w, calculada en la Ecuación (2.3), se mapea la luminancia para cada píxel de la escena a partir de un parámetro de entrada del *TMO* que será definido por el usuario (valor que se denominará como a). Este valor es subjetivo y está basado en el brillo deseado para la escena. El mapeo se realiza de la siguiente manera, dando como resultado la luminancia final de cada píxel de la imagen (*L(x,y)*):

$$L(x, y) = \frac{a}{Lm_w} \cdot L_w(x, y) \tag{2.5}$$

donde Lm_w se calcula en la Ecuación (2.3) y L_w*(x,y)* se obtiene a partir de la Ecuación (2.2). El valor del parámetro a es subjetivo, ya que depende del tipo de imagen al que se quiera aplicar el *TMO*. En el Cuadro 2.2 se exponen los valores recomendados para este parámetro en función del tipo de escena.

Tipo de escena	**Valor del parámetro *a***
Escenas oscuras o con poca cantidad de luz	0.09
Escenas con una cantidad moderada de luz	0.18
Escenas luminosas o con gran cantidad de luz	0.36

Cuadro 2.2: *Valores del parámetro a idóneos según el tipo de escena.*

(a) $a = 0.18$ (b) $a = 0.36$ (c) $a = 0.72$

Figura 2.5: *Diferentes valores del parámetro subjetivo a definido por el usuario como el brillo subjetivo ideal, aplicados a la misma escena. Como se puede apreciar, un aumento del valor produce un aumento de la luminancia de la escena.*

En casos de escenas muy oscuras o muy luminosas, el parámetro se puede ver reducido a la mitad o aumentado al doble respectivamente. En la Figura 2.5 se puede observar el efecto del parámetro a sobre la luminancia de la escena.

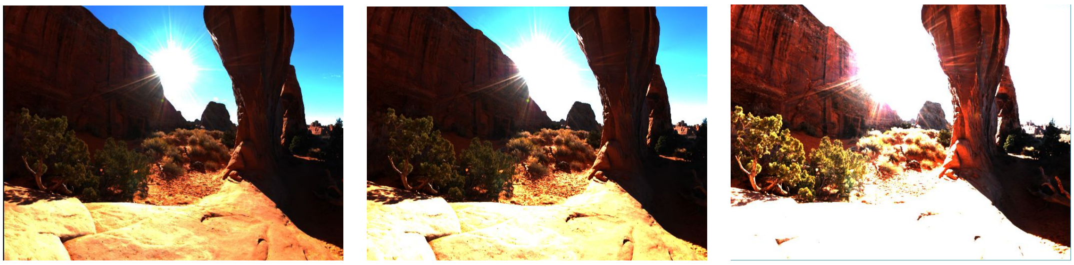
2.4.3. Compresión del rango dinámico: Efecto *burning*

Una vez se ha escalado la luminancia de la escena (ver Ecuación (2.5)), para realizar la compresión del rango dinámico de la imagen se utiliza una función sigmoidal, añadiendo el efecto de la técnica *Dodging and burning* (en español podría traducirse como 'Oscurecer y aclarar'). Esta técnica se utiliza en fotografía para controlar la exposición de la luz en ciertas zonas, pudiendo oscurecer o aclarar donde se crea conveniente; *dodging* limita la exposición para las áreas que se quieren ver más luminosas, mientras que *burning* incrementa la exposición de las zonas que deberían ser más oscuras. En el procesamiento digital de imágenes (y más concretamente, en la aplicación que implementa el *TMO* de este trabajo) también se puede utilizar esta técnica sobre la escena.

El parámetro de entrada del *TMO* denominado L_{white} se utiliza para definir la luminancia máxima deseable para la escena. Este parámetro se utiliza como un factor entre 0 y 1 que se aplica sobre el valor de la luminancia máxima de la escena, limitando este valor máximo para todos los píxeles. La Ecuación (2.6) aplica la comprensión de rango dinámico, utilizando la técnica previamente descrita.

$$L_d(x,y) = \frac{L(x,y) \cdot \left(1 + \frac{L(x,y)}{L_{white}^2 \cdot L_{max}}\right)}{1 + L(x,y)} \tag{2.6}$$

donde L_{max} representa la luminancia máxima de la escena. Como muestra la Ecuación (2.6), se limita la luminancia del píxel (*L(x,y)*) sobre el parámetro L_{white} aplicado sobre la luminancia máxima. Cabe destacar que un valor de L_{white} igual a 1 sería equivalente a no limitar la luminancia máxima. En la Figura 2.6 se aprecia gráficamente el efecto del parámetro L_{white} sobre una escena. Se puede observar que, cuanto más se limite el valor máximo de la luminancia en la escena, mayor cantidad de píxeles aparecen quemados.



(a) $L_{white} = 1$ (b) $L_{white} = 0.8$ (c) $L_{white} = 0.3$

Figura 2.6: *Diferentes valores del parámetro subjetivo* L_{white} *definido por el usuario para limitar el valor máximo de la luminancia de la escena y poder quemar determinadas zonas. Como se puede apreciar, conforme el valor disminuye una mayor cantidad de píxeles aparecen quemados.*

2.4.4. Imagen final

Una vez se ha aplicado el *TMO* a la escena utilizando las ecuaciones (2.2), (2.3), (2.5) y (2.6) se obtiene la escena final en un modelo de color que trabaja con luminancias. Por tanto, hace falta una reconversión al espacio de color *RGB* que la haga visualizable.

Para convertir la escena de formato XYZ a RGB se aplica una matriz de conversión:

$$RGB = \begin{bmatrix} 3{,}2404542 & -1{,}5371385 & -0{,}4985314 \\ -0{,}9692660 & 1{,}8760108 & 0{,}0415560 \\ 0{,}0556434 & -0{,}2040259 & 1{,}0572252 \end{bmatrix} \cdot XL_dZ$$

$$L_d = Y \tag{2.7}$$

siendo L$_d$ la nueva luminancia calculada en el operador respecto a la escena XYZ original calculada en el Apartado 2.4.1. La matriz de conversión utilizada es la inversa de la matriz utilizada en la Ecuación (2.2).

Nota: El anexo C contiene información complementaria y más detallada sobre los diferentes espacios de color utilizados en este proyecto.

3. Diseño e implementación

En la siguiente sección se explican conceptos relacionados con la aplicación desarrollada: el diseño de la aplicación (ver Apartado 3.1), conceptos técnicos que se consideran necesarios para entender la implementación llevada a cabo (ver Apartado 3.2), y finalmente la implementación del operador fase por fase (ver Apartados 3.3, 3.4 y 3.5).

3.1. Diseño de la aplicación

En el desarrollo de este trabajo se ha utilizado una metodología de diseño en cascada, utilizando un paradigma de programación orientada a objetos. La aplicación desarrollada se compone de una pequeña serie de clases: una de ellas se encarga de centralizar el proceso, mientras que las demás definen diversos elementos, como los entornos de renderizado o los *shaders*. En el apartado A.2 se explica más en detalle la metodología de diseño utilizada.

En referencia a las tecnologías elegidas para desarrollar la aplicación, se ha utilizado como lenguaje de programación C++, como *API* gráfica OpenGL y el lenguaje de *shading* GLSL. El entorno de desarrollo utilizado ha sido Microsoft Visual Studio 2013, y además se han utilizado una serie de librerías para gestionar temas como la carga de imágenes o la generación de ventanas. En el apartado A.1 se explican más detalladamente las tecnologías utilizadas.

3.2. Conceptos técnicos

En este apartado se explican varios conceptos técnicos propios tanto de OpenGL como propios de la programación con gráficos, utilizados en el desarrollo de la aplicación.

3.2.1. La *pipeline* gráfica

La *pipeline* gráfica define un conjunto de etapas que componen el proceso de renderizado de objetos en OpenGL, desde que es un objeto tridimensional inicialmente, hasta convertirse en una escena visualizable por el usuario en un dispositivo. En sus inicios, la *pipeline* gráfica tenía un comportamiento fijo, no permitía ningún tipo de parametrización por parte del usuario. Más adelante, el *hardware* evolucionó, permitiendo realizar modificaciones en la secuencia de renderizado.

A continuación se explica brevemente el proceso de renderizado en la aplicación, sintetizado gráficamente en la Figura 3.1.

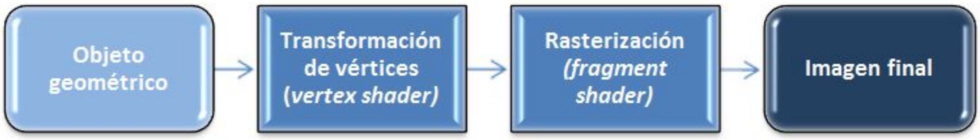


Figura 3.1: *Proceso de renderizado relacionado con la pipeline gráfica. De la geometría del objeto se inician las dos fases que se utilizan de la pipeline a través de shaders para generar una imagen final.*

La primera etapa comienza con los datos geométricos (primitivas geométricas) de la escena de renderizado, coordenadas de puntos y líneas. OpenGL trabaja con objetos denominados *Vertex Buffer Objects* que almacenan estos datos en la memoria de la tarjeta gráfica. El comportamiento de estos datos se programa mediante un *vertex shader* (en el Apartado 3.2.3 se explica el uso de estos *shaders* en este proyecto).

Tras esto, se procede a convertir las primitivas obtenidas en las etapas anteriores a un formato representable en la pantalla; es decir, pasar de las coordenadas de los vértices y triángulos a píxeles de la pantalla. Este proceso se conoce como rasterización. Finalmente, cada primitiva se asocia con un conjunto de píxeles, lo que genera un fragmento. Cada fragmento dispone de una posición asociada a la ventana. En esta última etapa se calcula el color final de cada píxel, se aplican texturas, se programan iluminaciones... Este comportamiento se programa en un *fragment shader*. Una vez realizados estos cálculos para cada píxel, el usuario ya puede visualizar la escena final.

En este apartado se explican brevemente conceptos básicos acerca de la pipeline gráfica que se consideran adecuados para entender mejor el algoritmo creado para este trabajo. En el Apartado B.1 se explica más en detalle la pipeline y sus etapas.

3.2.2. *Off-Screen Rendering: Frame Buffer Objects*

El término 'renderizar' define un proceso en el que se genera una imagen a partir de una escena en la que se define la posición de cada vértice y el valor de su píxel asociado. Normalmente se habla de renderizado en una pantalla visible cuando el resultado se visualiza directamente en la pantalla. El concepto de *Off-Screen Rendering* se basa en renderizar en un entorno que no es visible por el usuario. El proceso de renderizado es exactamente igual, diferenciándose en que **el resultado** tras cada pasada sobre la *pipeline* no es visible directamente por el usuario, en su lugar **se guarda en una textura**. La aplicación de este tipo de renderizado suele ser la generación de imágenes intermedias en texturas, con el objetivo de ser utilizadas como parte de otro renderizado posterior. OpenGL provee una extension con la que es posible generar renderizado fuera de la pantalla mediante un tipo de buffer especial: el *Frame Buffer Object* (de siglas *FBO*) [17].

En el presente documento se hablará a partir de este punto de FBO para referirse a un objeto de tipo Frame Buffer Object de OpenGL.

En este proyecto adquiere gran importancia el uso de *FBOs*, ya que todo el procesado realizado para generar la escena final se realiza *off-screen*: desde el renderizado de la imagen inicial hasta la aplicación del *TMO*, pasando por la obtención de la luminancia en la escena. En todas estas fases se utilizan texturas intermedias alojadas en un *FBO* hasta poder generar la escena *LDR* final.

3.2.3. *Shaders*

Un *shader* (o 'coloreador' en español) es un procedimiento de sombreado e iluminación utilizado para especificar el renderizado de un vértice o un píxel. Un *shader* utiliza lenguajes de alto nivel

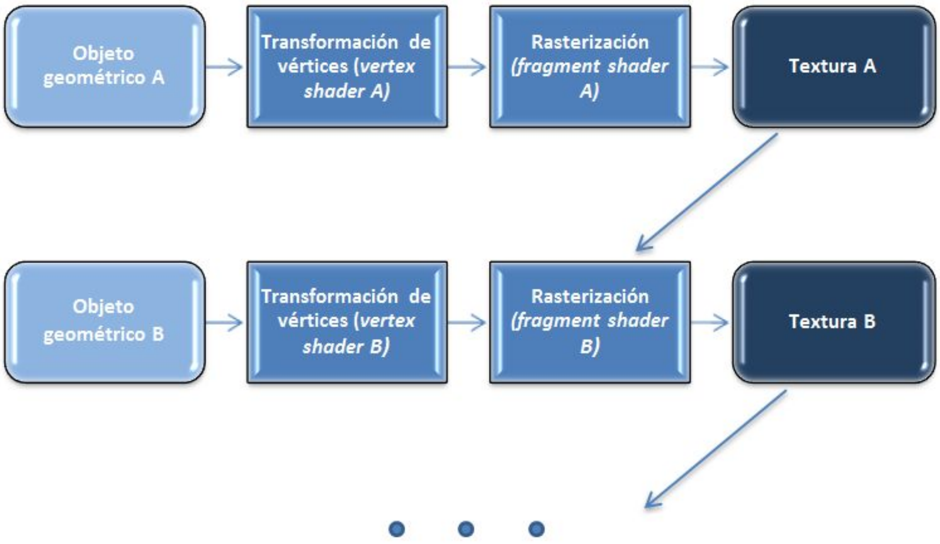


Figura 3.2: *Esquema de renderizado en un Frame Buffer Object. Como se puede observar, a diferencia del renderizado habitual el resultado se guarda en una textura que puede aplicarse a renderizados posteriores.*

que permiten ser independientes del *hardware* en el que se ejecuten. Existen varios tipos de *shaders* en función de la etapa de la *pipeline* gráfica (más información en el Apartado B.1) en la que se ejecuten, ya que realizan una función distinta. Estos programas se comunican con el código a partir de variables que pueden ser tanto de entrada como de salida. A partir de estas variables, que pueden ir desde simples números a texturas, se pueden configurar parámetros que participen en el proceso de renderizado.

Para la realización de este proyecto se han utilizado dos clases de *shaders*: *vertex* y *fragment shader*, los cuales se definen brevemente a continuación.

Un *vertex shader* define las características de cada vértice de la escena, como sus coordenadas espaciales o su orientación. Realiza operaciones matemáticas sobre las estructuras de vértices de los modelos tridimensionales; cada vértice está definido por su posición en el escenario 3D mediante sus coordenadas (x, y, z). Cada *vertex shader* trabaja con cada vértice individualmente y no puede modificar el tipo de definición, ni eliminar o crear vértices, pero sí puede modificar propiedades como la posición, repercutiendo así en la geometría del objeto.

Un *fragment shader* define el color de cada píxel de la escena. Realiza transformaciones sobre fragmentos (píxeles que poseen información añadida, como la posición o las coordenadas en la textura). Forman parte de la etapa de rasterización, y pueden devolver valores de color y también de profundidad para cada fragmento. Los *fragment shaders* trabajan con cada fragmento individualmente y sin conocimiento de la geometría de la escena, por lo que no pueden realizar efectos complejos.

Shaders en la aplicación. Los *shaders* se utilizan en fases de renderizado, por lo que adquieren mucha importancia en este proyecto, ya que son necesarias varias fases de renderizado para aplicar el *TMO* elegido (ver Figura 3.1). Concretamente, se utilizan *shaders* para el renderizado inicial de la escena elegida (ver Apartado 3.3), para conseguir la luminancia media y la luminancia máxima de la escena (ver Apartado 3.4), y finalmente para aplicar el *TMO* a la escena (ver Apartado 3.5).

Implementación del operador

En lo siguientes apartados se explica a nivel técnico la implementación de las diferentes fases del algoritmo, previamente introducido en el Apartado 1.1.2 y analizado en el Apartado 2.4 del presente documento.

El algoritmo implementa el *TMO* global de Reinhard [7]. Partiendo de una imagen *HDR* inicial, el objetivo es generar una escena *HDR* a partir de esa imagen y convertirla en una escena de bajo rango dinámico visualizable. En la Figura 3.3 se presenta un esquema a nivel global del desarrollo de la aplicación, detallado a continuación. En el Apartado 3.3 se genera la escena inicial. En el Apartado 3.4 se obtienen los valores de luminancia de la escena. Finalmente, en el Apartado 3.5 se aplica el *TMO* para generar una escena de bajo rango dinámico.

Nota: A partir de este momento se utilizará la terminología HDR para hablar de texturas de 32 bits

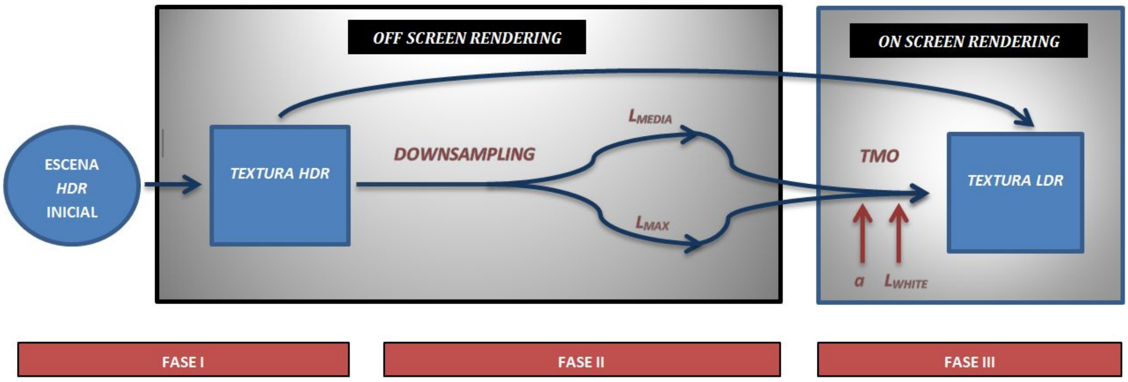


Figura 3.3: *Esquema a alto nivel de la aplicación. Como se puede observar, en la primera fase se genera la escena HDR inicial, utilizada en la segunda fase para conseguir los valores de luminancia mediante la técnica de downsampling. Con esos valores y los parámetros a y* L_{white} *elegidos por el usuario se aplica el TMO a la escena HDR generada (fase 3), dando como resultado una escena de bajo rango dinámico.*

por canal en coma flotante de resolución, mientras que se denominarán como LDR a las texturas de 8 bits por canal. En el Apartado 1.1.1 se introduce el concepto de resolución de datos HDR.

3.3. Fase I: Generación de la escena *HDR* inicial

El primer paso del algoritmo consiste en la generación de la escena *HDR* inicial a la que se desee aplicar el *TMO*. Para ello, se selecciona la imagen *HDR* inicial y se guarda en una textura *HDR*, denominada *HDRTex*. Para generar la escena inicial, se ha creado un entorno donde poder renderizar esta textura *off - screen* (ver Apartado 3.2.2) sobre otra textura *HDR* alojada en un *FBO*, denominada *ViewTex*.

Se ha elegido como escena de prueba para generar la secuencia de fotogramas en *HDR* una esfera que gira sobre su eje horizontal, siendo el punto de vista del proceso de renderizado el centro del interior de la esfera. Esto permite que *ViewTex* contenga a cada frame una porción diferente de la imagen *HDR* inicial. Cabe destacar que esta escena ha sido elegida para probar el operador, y que cualquier otra más compleja también serviría para que el *TMO* funcionase.

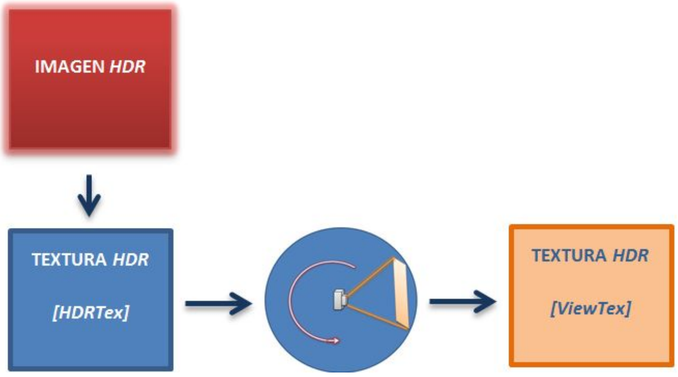


Figura 3.4: *Esquema general de la primera fase del algoritmo. La imagen HDR inicial se carga en una textura HDR que se proyecta en un entorno de renderizado. Se ha elegido un entorno esférico, en el que se renderiza una parte diferente de la imagen a cada momento. Este renderizado genera la escena inicial, guardada en otra textura HDR.*

3.4. Fase II: *Downsampling*

Una vez se ha guardado la escena inicial en una textura, el objetivo es obtener el valor de la luminancia para cada píxel de la escena. Para ello se renderiza *viewTex* sobre dos texturas *HDR* de un solo canal, utilizando un *shader* que convierte los valores *RGB* a valores de luminancia mediante la conversión definida en la Ecuación (2.2). De esta manera, se obtienen dos texturas que guardan los valores de luminancia de cada píxel de la escena inicial. Estas texturas se han denominado *DSAvgTex1* y *DSMaxTex1*, y son la fuente para el cálculo de la luminancia media y la luminancia máxima de la escena, respectivamente.

Una vez se tienen los valores de luminancia iniciales, el siguiente paso es calcular la luminancia media y la luminancia máxima de la escena. Para ello, es preciso recorrer todos los valores de luminancia de la escena (todos los píxeles de *DSAvgTex1* y *DSMaxTex1*) y aplicar los cálculos definidos en las ecuaciones (2.3) y (2.4). Estos cálculos podrían realizarse en *CPU* recorriendo secuencialmente cada píxel, pero sería un cálculo muy costoso. Es necesario conseguir realizar estos cálculos de forma eficiente aprovechando las posibilidades que ofrece OpenGL y la programación en

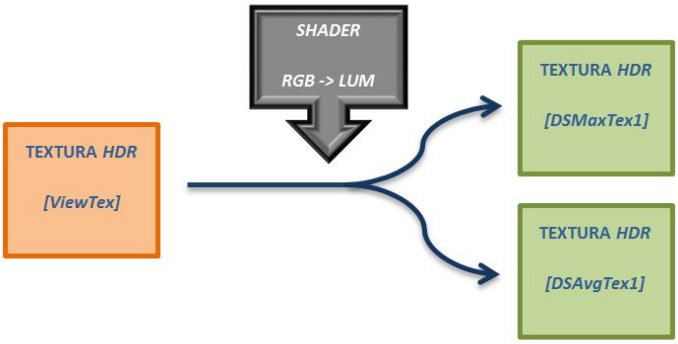


Figura 3.5: *Obtención de la luminancia para cada píxel de la escena inicial. Mediante un shader que convierte valores RGB en valores de luminancia, DSAvgTex1 y DSMaxTex1 contienen los valores de luminancia de la escena inicial.*

GPU, y utilizando la menor cantidad de recursos posible. Conseguir realizar este tipo de cálculos aprovechando estas posibilidades no es trivial, y en este proyecto se ha decidido utilizar una técnica denominada *downsampling* para ello, la cual se explica a continuación.

Definición

La técnica denominada como *downsampling* permite reducir el ratio de muestreo de una señal con el fin de conseguir reproducirla en un formato más limitado. En el caso de este proyecto, lo que se pretende reducir es el tamaño de *DSAvgTex1* y *DSMaxTex1*, con el objetivo de conseguir una textura de tamaño unidad (*1,1*) que contiene el valor deseado.

Implementación

La implementación del *downsampling* en la aplicación es la siguiente: a partir de las texturas iniciales *DSAvgTex1* y *DSMaxTex1* de tamaño (n,m) se calcula el valor medio y máximo respectivamente para cada tupla de 4 texels vecinos 2 a 2 (cabe recordar que un texel es el equivalente a un píxel en la textura), con el objetivo de que el valor obtenido represente un texel en otra textura cuyo tamaño es la mitad ($n/2$,$m/2$). Este proceso se repite hasta conseguir una textura de tamaño

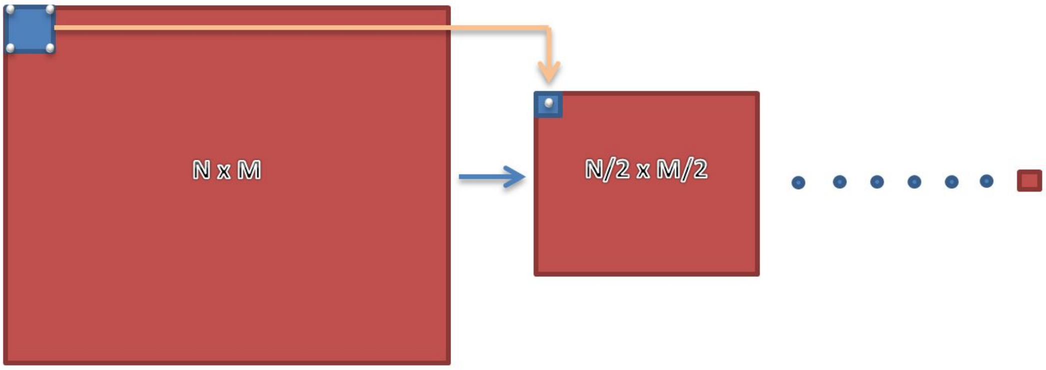


Figura 3.6: *Implementación de downsampling la aplicación*

(*1,1*), con un único texel que contiene el valor medio o máximo de la luminancia de la escena. En la Figura 3.6 se puede observar el proceso desde la textura inicial hasta la final de tamaño (*1,1*).

Mediante esta técnica se consigue un número de renderizados de orden logarítmico, siendo el coste de cada renderizado el número de píxeles de la mayor dimensión de la imagen. En la Ecuación (3.1) se muestra gráficamente el coste temporal conseguido mediante esta técnica.

$$\mathbf{O}(\log[max(n,m)] \cdot max(n,m)) \tag{3.1}$$

siendo las variables n y m las dimensiones de la imagen.

La implementación de esta técnica no es sencilla, ya que requiere de cálculos complejos y recursos de almacenamiento para guardar los valores necesarios en cada iteración. Una primera aproximación sería crear texturas de tamaño (*n,m*), (*n/2,m/2*), (*n/4,m/4*)... De esta forma se crean una gran cantidad de texturas (una por cada renderizado) hasta llegar a una textura de tamaño (*1,1*), lo que hace esta aproximación muy ineficiente, sobre todo ante texturas *HDR* de gran tamaño. En este proyecto se utiliza una aproximación mucho más eficiente en memoria para implementar el *downsampling*: se ha utilizado la denominada 'técnica del ping-pong', la cual se describe a continuación.

Técnica del ping - pong

La implementación de esta técnica se basa en utilizar dos texturas como fuente y destino del renderizado, como si de una partida de ping - pong se tratase. Una textura será la fuente del renderizado, en el que se aplican los cálculos explicados anteriormente a través de un *fragment shader* a la segunda textura. Esta segunda textura que contiene el resultado será la fuente para el siguiente renderizado. En las siguientes iteraciones, la textura destino será la fuente y viceversa, y así hasta llegar a una textura de tamaño (*1,1*). En la Figura 3.7 se puede observar el proceso de renderizado.

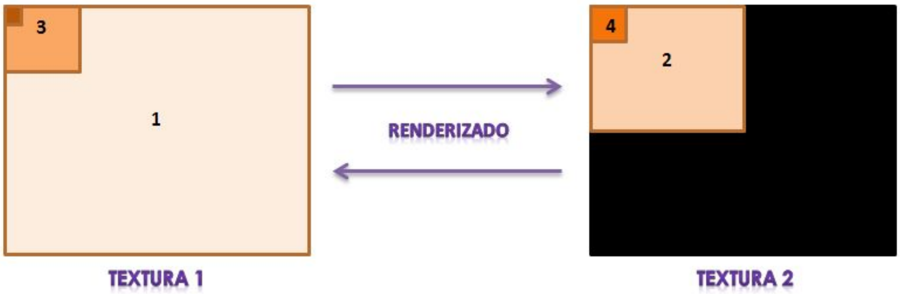


Figura 3.7: *Técnica del ping - pong: flujo total. En cada iteración se reduce el tamaño de la escena a la mitad, pivotando entre las texturas fuente y destino del renderizado.*

El objetivo de la implementación de esta técnica es conseguir el menor coste en memoria posible para obtener la luminancia media y máxima de la imagen. De esta forma, el *downsampling* tan solo necesita cuatro texturas (*DSAvgTex1* y *DSAvgTex2* para el cálculo de la luminancia media, y *DSMaxTex1* y *DSMaxTex2* para la máxima) cuyo tamaño es el de la escena *HDR* inicial. Sin la implementación del ping - pong, este coste sería de orden logarítmico respecto a la mayor dimensión de la imagen. En la Ecuación (3.2) se muestra el coste en memoria obtenido utilizando esta técnica

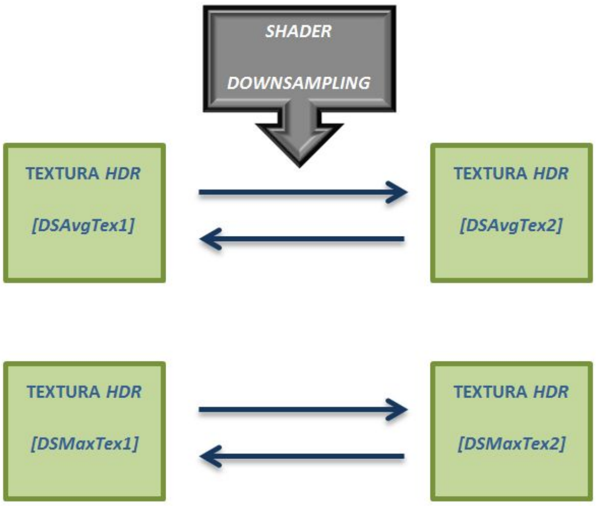


Figura 3.8: *Obtención de luminancia media y máxima a través de downsampling. El renderizado pivota entre las texturas fuente y destino hasta lograr una textura de tamaño (1,1) que contiene el valor de la luminancia media y máxima de la escena. Dependiendo del número de iteraciones, el valor final se obtendrá de una textura u otra.*

frente al obtenido sin haberla utilizado.

$$\mathbf{O}(4 \cdot T) \quad \textbf{VS} \quad \mathbf{O}(\log[max(n,m)] \cdot T) \tag{3.2}$$

donde T representa el coste en memoria de crear una textura.

Una vez se ha conseguido la luminancia media y la máxima de la escena, el siguiente paso es aplicar los cálculos que conforman el *TMO* global de Reinhard [9], explicados en el siguiente apartado.

3.5. Fase III: Aplicación del operador de *tone mapping*

Una vez se han obtenido los valores de luminancia deseados, el siguiente paso es aplicar el *TMO*, tal y como se muestra en la Figura 3.9. Este operador requiere de una serie de parámetros o variables de entrada:

- **Luminancia media y máxima de la escena**: Obtenida en la anterior fase del algoritmo (ver Apartado 3.4) y alojada en una textura de un solo texel de tamaño.

- $\boldsymbol{a}$: Parámetro elegido por el usuario, pudiendo ser cambiado a tiempo real. La importancia e implicación de este parámetro en la aplicación se explica en detalle en el Apartado 2.4.2 y en la Ecuación (2.5). El valor inicial de este parámetro es 0,18.
- $\boldsymbol{L}_{white}$: Parámetro elegido por el usuario, pudiendo ser cambiado a tiempo real. La importancia e implicación de este parámetro en la aplicación se explica en detalle en el Apartado 2.4.3 y en la Ecuación (2.6). El valor inicial de este parámetro es 0,8.

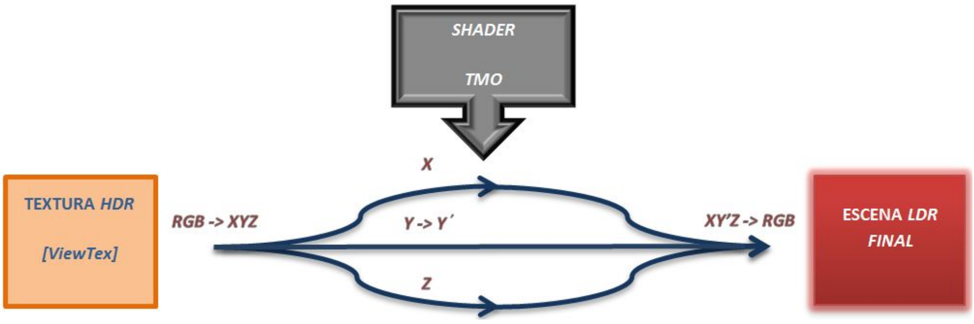


Figura 3.9: *Esquema general de la última fase de la aplicación. A partir de la escena HDR generada se aplica el TMO utilizando los parámetros obtenidos, lo que genera la escena LDR final. Para ello, el fragment shader convierte la escena al modelo XYZ y modifica el parámetro Y que indica la luminancia; los parámetros X y Z no se modifican.*

Una vez se tienen estos parámetros, mediante un *fragment shader* se aplica el *TMO* siguiendo el análisis del algoritmo detallado en los Apartados 2.4.2 y 2.4.3 y aplicando las ecuaciones (2.5) y (2.6). Este *shader* convierte la escena al espacio de color *XYZ* que trabaja con luminancias para poder aplicar los cálculos del *TMO*. Una vez comprimido el rango de la escena, se aplica la conversión al espacio de color original (ver Ecuación (2.7)), renderizando *on - screen* en una textura *LDR* la escena final visualizable. De esta forma, el usuario puede observar el efecto del *TMO* sobre la escena y modificar los parámetros a y L_{white} según crea conveniente.

El proceso finaliza cuando el usuario cierra la ventana de visualización; hasta entonces se ejecuta el proceso detallado en esta sección, cambiando **a tiempo real** la escena *HDR* y el efecto del *TMO* sobre ella.

4. Resultados

En esta sección se estudian los resultados obtenidos tras la implementación del algoritmo. En primer lugar se analiza el coste temporal medio entre cada fase del algoritmo, con el objetivo de analizar el coste de ejecución del *TMO* implementado (ver Apartado 4.1). Después se verifica el funcionamiento a tiempo real del algoritmo ante diferentes escenas (ver Apartado 4.2). Por último, se muestran gráficamente resultados de la aplicación del operador (ver Apartado 4.3).

Para la realización de las siguientes pruebas, se ha utilizado un modelo de tarjeta gráfica *Nvidia GeForce GT 630M*. Sus especificaciones aparecen en el Cuadro 4.1.

GPU	**Frecuencia**	**Núcleos**	**Memoria RAM**
GeForce GT 630M	800 MHz	96	2 GB

Cuadro 4.1: *Especificaciones de la tarjeta gráfica utilizada en las pruebas de la aplicación.*

4.1. Fases del algoritmo

En esta sección se analiza la relación temporal entre cada fase del algoritmo. El algoritmo se compone de tres fases de renderizado: en la primera se proyecta sobre un entorno esférico la imagen *HDR* con el objetivo de generar la escena inicial (ver Apartado 3.3). En la segunda fase, se obtienen la luminancia media y la luminancia máxima de la escena (ver Apartado 3.4). La tercera y última fase aplica el *TMO* a la escena inicial para generar contenido *LDR* visualizable (ver Apartado 3.5).

Nota: Los valores expuestos en este apartado corresponden a valores medios tras aplicar el algoritmo durante 1000 frames, con un tamaño de ventana de visualización de 800 x 600 píxeles.

Como se puede observar en el Cuadro 4.3, la primera fase ocupa la mayor parte del tiempo (aproximadamente el 90 %). Conforme el tamaño de la escena inicial aumenta, también lo hace el

Nº píxeles	Fase I	Fase II	Fase III	TOTAL
1024 x 512	0.015382	0.000249	0.000030	0.015661
3200 x 1600	0.015781	0.000949	0.000119	0.016849
4096 x 2048	0.016208	0.001199	0.000117	0.017524
5120 x 2560	0,024463	0,001190	0,000107	0,025760

Cuadro 4.2: *Comparativa del coste temporal de cada fase del algoritmo medido en segundos, utilizando distintos tamaños de imagen HDR.*

coste de esta fase respecto de las demás, ya que depende del tamaño de escena que se renderiza. La influencia del tamaño de la escena inicial es menor en las demás fases.

Como se ha descrito en el Apartado 3.3, la escena elegida para renderizar inicialmente ha sido una esfera que gira en torno a su eje horizontal. **Esta renderización inicial**, aún tratándose de una escena bastante simple, **ocupa un 90 % del tiempo del proceso respecto al *TMO***, el cual ocupa el 10 % restante (ver Figura 4.1). Esto implica que el coste temporal del operador implementado no afecta prácticamente al coste total del proceso, y su integración en escenas más complejas no supondría una disminución significativa de la eficiencia temporal. En resumen, el operador implementado podría aplicarse a escenas o procesos más complejos pudiendo mantener buen rendimiento a tiempo real.

Nº píxeles	Fase I	Fase II	Fase III
1024 x 512	89,58 %	9,28 %	1,13 %
3200 x 1600	92,77 %	6,45 %	0,78 %
4096 x 2048	92,83 %	6,49 %	0,67 %
5120 x 2560	94,87 %	4,69 %	0,44 %

Cuadro 4.3: *Comparativa porcentual del coste temporal de cada fase del algoritmo utilizando distintos tamaños de imagen HDR.*

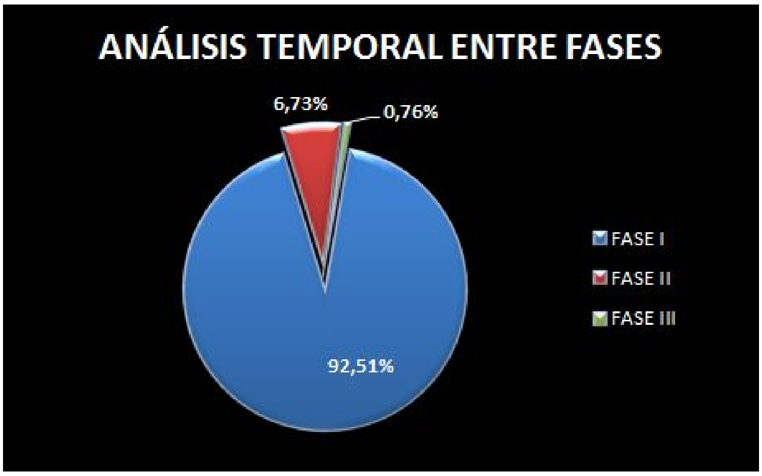


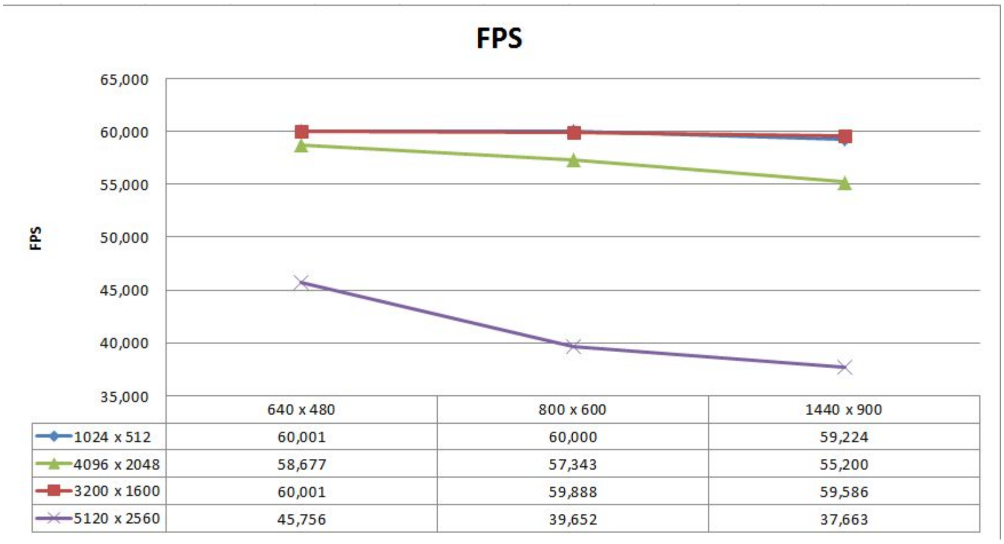
Figura 4.1: *Análisis porcentual del coste temporal entre fases del algoritmo, utilizando valores medios ante varios tamaños de imagen. Como se puede apreciar, el renderizado de la escena inicial ocupa la mayor parte del coste temporal del algoritmo.*

4.2. Análisis de rendimiento

A continuación se analiza el rendimiento del algoritmo medido en FPS (*frames* o imágenes por segundo). Esta medida representa el número de fotogramas o imágenes que se visualizan por segundo. Por lo general, una cantidad de 24 FPS consigue que el ojo humano adquiera una ilusión de movimiento. Con una cantidad mayor de 40 FPS el ojo humano no percibe un cambio significativo entre fotogramas, mientras que con una tasa de 60 FPS se tiene una sensación de realismo y fluidez total.

El algoritmo implementado en este proyecto aplica un *TMO* a una imagen *HDR* en tiempo real, por lo que la tasa conseguida debe ser superior, como mínimo, a los 24 FPS y más cercana a los 60. A continuación se exponen los resultados sobre los tamaños de imagen *HDR* utilizados en las pruebas anteriores, utilizando varios tamaños de pantalla visualizable.

Cabe destacar que el dispositivo de visualización sobre el que se realizan las pruebas tiene una frecuencia de refresco de 60Hz. Por lo general se limita la frecuencia a la del dispositivo, por lo que



	640 x 480	800 x 600	1440 x 900
1024 x 512	60,001	60,000	59,224
4096 x 2048	58,677	57,343	55,200
3200 x 1600	60,001	59,888	59,586
5120 x 2560	45,756	39,652	37,663

Figura 4.2: *Análisis del rendimiento en FPS del algoritmo ante imágenes HDR de diferente tamaño. Como se puede apreciar, la tasa disminuye ante escenas de mayor tamaño y ventanas de visualización más amplias, pero logra mantener el rendimiento a tiempo real ante resoluciones amplias. El eje horizontal de la gráfica se corresponde con la resolución de la ventana visualizable.*

las pruebas han sido realizadas con esta limitación. Realmente la aplicación es capaz de generar más FPS que los 60 que se han obtenido para las texturas de menor tamaño utilizadas (ver figura 4.2). No se ha creido conveniente desactivar esta limitación, ya que podrían producirse efectos de desincronización entre diferentes *frames* al refrescar la pantalla, y la imagen no se visualizaría correctamente. Este tipo de efecto se suele denominar *screen tearing*.

4.3. Aplicación del operador

A continuación se exponen varios ejemplos gráficos de la aplicación del *TMO* implementado, cambiando parámetros y explicando el cambio ejercido sobre varias escenas *HDR* tras convertirlas a un formato *LDR*. A continuación se muestra el efecto de la variación de la luminancia y de los parámetros del *TMO* en una escena (Figuras 4.3, 4.4, 4.5 y 4.6), así como ejemplos de diferentes orientaciones de la cámara o *frames* resultantes en diversas escenas (Figuras 4.7, 4.8, 4.9 y 4.10).

(a) Mapa de entorno aplicado a la esfera de la escena de prueba en *LDR*

Figura 4.3: *Diferentes orientaciones de la cámara en una misma escena.*

Figura 4.4: *Influencia de la luminancia media de la escena. En la segunda imagen entran píxeles oscuros en la parte derecha, por lo que disminuye. En la tercera aumenta al desaparecer los píxeles oscuros de la parte izquierda. Este efecto emula la adaptación del ojo humano.*

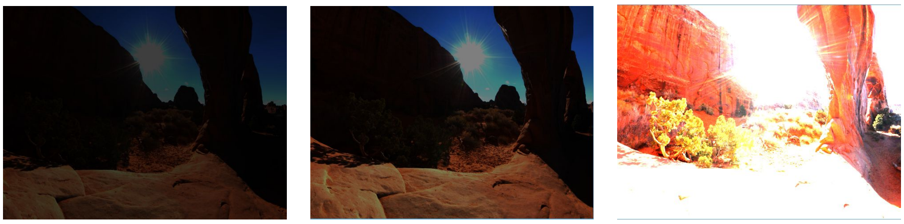
La variación de la luminancia para cada píxel en la escena final viene definida en las ecuaciones (2.5) y (2.6), en las que se muestra que la luminancia final depende de cinco parámetros:

- Luminancia original del píxel
- Luminancias media y máxima de la escena actual
- Parámetros subjetivos a y L_{white}

La Ecuación (2.5) mapea la luminancia para cada píxel, multiplicando el parámetro a por la luminancia original del píxel, y dividiendo entre la luminancia media de la escena actual. Este mapeo de la luminancia sirve en la Ecuación (2.6) para comprimir el rango dinámico utilizando. El parámetro L_{white} es un factor aplicado a la luminancia máxima de la imagen, con el objetivo de limitar el valor máximo.

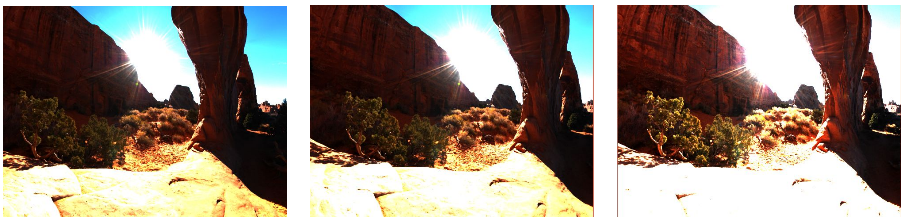
Por tanto, la luminancia de un píxel en la escena final aumenta si disminuye la luminancia media de la escena actual, si aumenta la luminancia del píxel original, y si aumenta el parámetro a. Un píxel aparecerá totalmente luminoso o quemado con mayor probabilidad si el factor L_{white} disminuye. Todas estas variaciones se explican a continuación.

(a) $a = 0.18$ (b) $a = 0.36$ (c) $a = 0.72$

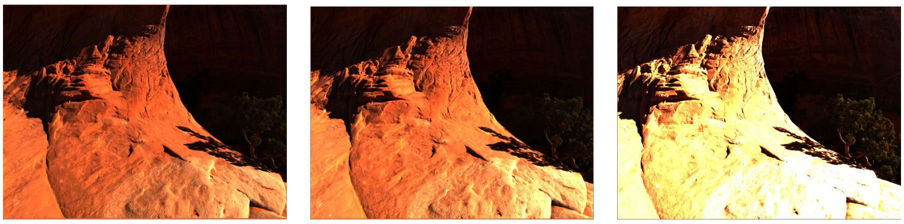


(d) $a = 0.045$ (e) $a = 0.09$ (f) $a = 2.88$

Figura 4.5: *Influencia del parámetro a en la escena. Las tres figuras superiores muestran valores lógicos del parámetro, mientras que las tres figuras inferiores muestran valores extraños que no producen efectos realistas.* $L_{white} = 1$.



(a) $L_{white} = 0.8$ (b) $L_{white} = 0.5$ (c) $L_{white} = 0.2$



(d) $L_{white} = 0.8$ (e) $L_{white} = 0.5$ (f) $L_{white} = 0.2$

Figura 4.6: *Influencia del parámetro* L_{white} *en la escena. Los píxeles cuya luminancia es igual o mayor a* $L_{white} \cdot L_{max}$ *aparecen quemados, por lo que conforme disminuye el valor, aumenta el número de píxeles que cumplen esta propiedad.* $a = 0.36$.

(a) Mapa de entorno aplicado a la esfera de la escena de prueba en *LDR*

Figura 4.7: *Diferentes frames resultantes de aplicar el TMO sobre una escena rocosa. En esta escena aparecen varias zonas rocosas de diversa luminancia, lo que permite apreciar visualmente en varios puntos la adaptación del operador.*

(a) Mapa de entorno aplicado a la esfera de la escena de prueba en *LDR*

Figura 4.8: *Diferentes frames resultantes de aplicar el TMO sobre una escena desértica. En esta escena no se aprecian zonas de grandes variaciones de la luminancia.*

(a) Mapa de entorno aplicado a la esfera de la escena de prueba en *LDR*

Figura 4.9: *Diferentes frames resultantes de aplicar el TMO sobre una escena de vegetación. En esta escena se aprecian varias zonas de contraste lumínico, ya que aparecen varias zonas soleadas entre la vegetación.*

(a) Mapa de entorno aplicado a la esfera de la escena de prueba en *LDR*

Figura 4.10: *Diferentes frames resultantes de aplicar el TMO sobre una escena de interior. En esta escena se aprecia un gran contraste lumínico en la zona exterior, lo que permite visualizar la adaptación del operador.*

5. Conclusiones

En este último capítulo se explican las conclusiones a nivel técnico y personal de la realización de este trabajo. En primer lugar se comentan posibles líneas futuras (Apartado 5.1). A continuación se exponen las conclusiones sacadas de la realización satisfactoria de este trabajo (Apartado 5.2). Finalmente, se da una valoración personal sobre el trabajo y su realización (Apartado 5.3).

5.1. Líneas futuras

Tras la realización del proyecto han ido surgiendo una serie de aspectos que se consideran interesantes de cara a mejorar la aplicación desarrollada. Dada la eficiencia temporal lograda, se podría aumentar la complejidad del operador, pudiendo añadir parámetros como el valor de la luminancia mínima, estudiando como podría afectar a un mejor resultado sobre la escena. Otra opción a tener en cuenta sería la implementación del operador local de Reinhard [8], añadiendo los cálculos que fueran necesarios.

Resultaría interesante retomar en el futuro este proyecto de cara a un posible trabajo fin de máster, por ejemplo. Se trata de un campo de investigación con muchas aplicaciones, pudiendo ampliarse sus objetivos: aumentar la complejidad del operador, añadir funcionalidades como generar la escena *HDR* inicial a partir de varias imágenes *HDR* idénticas con distinta luminosidad...

5.2. Conclusiones del trabajo realizado

Tras finalizar el proyecto, se confirma que se han conseguido los objetivos establecidos al comienzo (ver Apartado 1.2). Para alcanzar estos objetivos, ha sido necesario diseñar una aplicación que implementase un *TMO* que funcionase a tiempo real sobre una escena *HDR*, mejorando el estado del arte actual (las curvas γ en videojuegos). La premisa inicial del proyecto se ha cumplido satisfactoriamente, e incluso se ha superado con creces: el operador implementado logra una tasa de FPS bastante superior a la mínima de 24 FPS propuesta, umbral mínimo de sensación de tiempo real para el ojo humano (ver Apartado 4.2). Así mismo, se han cumplido los aspectos de naturaleza funcional y no funcional descritos en los Apartados 2.1 y 2.2, respectivamente.

5.3. Valoración personal

La realización de este trabajo fin de grado ha supuesto una experiencia muy enriquecedora y gratificante, tanto personal como profesionalmente. Antes de empezar el proyecto carecía de prácticamente cualquier tipo de experiencia o conocimiento tanto en el campo de los gráficos como en las tecnologías utilizadas: la *API* OpenGL, el lenguaje C++... Este desconocimiento, unido a la curiosidad por conocer el mundo de los gráficos y la programación en *GPU*, fueron las razones que me llevaron a embarcarme en un trabajo de este tipo.

El uso de OpenGL me ha permitido experimentar las dificultades derivadas de trabajar con una tecnología en constante desarrollo. La escasa documentación existente sobre su utilización en las versiones más modernas ha sido un escollo durante el proyecto. La utilización de OpenGL como *API* de gráficos ha condicionado el uso de C++ como lenguaje de programación, el cual era también un reto para mí debido a la escasa enseñanza recibida previamente en el graduado. Este desconocimiento ha servido para adquirir conocimientos valiosos acerca de uno de los lenguajes de programación más utilizados, lo cual es muy positivo desde el punto de vista profesional. Los conceptos relacionados con el renderizado, la programación con la *GPU* y los *shaders* también eran totalmente nuevos, por lo que he podido conocer conceptos sobre un ámbito atractivo dentro del mundo de la informática.

En definitiva, conseguir realizar este trabajo desarrollando una aplicación totalmente desde cero es bastante gratificante a nivel personal, tanto por el esfuerzo realizado como por los conocimientos adquiridos en el campo de los gráficos, lo que ha supuesto una gran experiencia.

Referencias

[1] DirectX `http://www.directx.com.es/`

[2] OpenGL `https://www.opengl.org`

[3] GLSL lenguaje de *shading* `https://www.opengl.org/documentation/glsl/`

[4] Larson, G.W. and Rushmeier, H. and Piatko, C., "IEEE Transactions on Visualization and Computer Graphics," *A visibility matching tone reproduction operator for high dynamic range scenes*, Vol.3, pp. 291-306, October 1991.

[5] Durand, F. and Dorsey, J., "ACM Trans. Graph," *Fast bilateral filtering for the display of high-dynamic-range images*, Vol.21, pp. 257-266, July 2002.

[6] Drago, F. and Myszkowski, K. and Annen, T. and Chiba, N., "Computer Graphics Forum," *Adaptive logarithmic mapping for displaying high contrast scenes*, Vol.22, pp. 419-426, 2003.

[7] Reinhard, E. and Stark, M. and Shirley, P. and Ferwerda, J., "ACM Trans. Graph.," *Photographic tone reproduction for digital images*, Vol.21, pp. 267-276, July 2002.

[8] Reinhard, E. and Devlin, K., "IEEE Trans. Vis. Comput. Graph.," *Dynamic Range Reduction Inspired by Photoreceptor Physiology*, Vol.21, pp. 13-24, December 2004.

[9] Erik Reinhard page `http://www.erikreinhard.com/hdr.html`

[10] Modelos de color *RGB* y *XYZ*: `http://www.optique-ingenieur.org/en/courses/OPI_ang_M07_C02/co/Contenu_07.html`

[11] Lenguaje de programación *C++* `http://www.cplusplus.com/`

[12] Librería *GLFW* `http://www.glfw.org/`

[13] Librería *GLEW* `http://glew.sourceforge.net/`

[14] Librería *FreeImage* `http://freeimage.sourceforge.net/features.html`

[15] Librería *glm* `http://glm.g-truc.net/0.9.7/index.html`

[16] Microsoft Visual Studio `https://www.visualstudio.com/`

[17] *Frame Buffer Objects* `https://www.opengl.org/wiki/Framebuffer_Object`

[18] Foro de preguntas y respuestas sobre programación `http://www.stackoverflow.com`

Nota: Las direcciones que aparecen en esta bibliografía están revisadas a fecha 12/11/2015.

A. Anexo I: Diseño de la aplicación

En este anexo se explican aspectos relacionado con el diseño de la aplicación desarrollada. En primer lugar se explican las tecnologías (el *software*) que se han utilizado para su creación y desarrollo (ver Apartado A.1). Seguidamente se detallan conceptos técnicos necesarios para entender el funcionamiento de la aplicación (ver Apartado 3.2). Finalmente, se explica la estructura a nivel técnico de la aplicación desarrollada (ver Apartado A.2)).

A.1. Tecnologías utilizadas

A continuación se explica el *software* utilizado para la realización de este trabajo; desde el entorno de desarrollo utilizado hasta las extensiones de librerías utilizadas.

A.1.1. Lenguaje de programación: C++

El lenguaje de programación es una de las tecnologías más importantes a la hora de desarrollar una aplicación, ya que define y limita posibles características y posibilidades a la hora de programar. Para este trabajo se ha elegido C++ [11].

C++ es un lenguaje de programación multiparadigma, ya que posee facilidades de programación genérica, estructurada y orientada a objetos. Actualmente C++ es uno de los lenguajes más utilizados en todo el mundo.

Se ha elegido este lenguaje de programación por ser uno de los más poderosos y preparados para comunicarse con la *GPU* mediante *APIs*, y por ser uno de los lenguajes más utilizados actualmente.

A.1.2. Interfaz de programación: OpenGL

OpenGL (*Open Graphics Library*) [2] es una interfaz de programación (siglas *API* en inglés) de uso libre multiplataforma creada originalmente por Silicon Graphics Inc. en 1992, diseñada para aplicaciones que produzcan gráficos en 2 y 3 dimensiones. OpenGL proporciona ventajas sobre otras opciones de programación en *GPU*.

El uso de esta *API* oculta la complejidad de la interfaz con las tarjetas gráficas, presentando una sola *API* uniforme. La interfaz consiste en funciones utilizadas para dibujar escenas tridimensionales complejas a partir de primitivas geométricas simples como puntos, líneas y triángulos.

Se ha elegido esta *API* por ser una de las más comúnmente utilizadas y de uso libre, ya que algunas de las alternativas posibles (por ejemplo, CUDA) necesitan de un tipo de *hardware* específico.

A.1.3. Lenguaje de *shading*: GLSL

El lenguaje de *shading* GLSL (*OpenGL Shading Language*) [3] es un lenguaje de alto nivel basado en el lenguaje de programación C que permite diseñar pequeños programas que serán ejecutados sobre la *GPU*, denominados *shaders*. Se utiliza para contactar con la *GPU* en etapas programables de la *pipeline* gráfica (ver Apartado 3.2.1 y Anexo B), siendo común la utilización de vectores y matrices. La principal alternativa a GLSL es HLSL, el lenguaje propiedad de Microsoft y obligatorio en el uso de Direct3D como *API*.

Se ha elegido este lenguaje de *shading* por ser el adecuado dentro de entornos que utilicen OpenGL, como es el caso de la aplicación desarrollada en este proyecto. En el Apartado 3.2.3 se explica en detalle el uso de *shaders* es el proyecto.

A.1.4. Librerías

Se han requerido una pequeña serie de librerías *opensource* que añaden funcionalidades a OpenGL. A continuación se exponen brevemente las bibliotecas utilizadas:

- *GLFW*: Biblioteca que habilita la creación y administración de ventanas y entradas de teclado y ratón, así como la recepción de eventos. [12]

- *GLEW*: OpenGL Extension Wrangler Library es una librería destinada a ayudar en la carga y consulta de extensiones de OpenGL. [13]
- *FreeImage*: Librería que permite cargar imágenes desde ficheros locales, soportando varios formatos (para el uso de esta aplicación se han utilizado imágenes *.hdr* y *.exr*). [14]
- *glm*: OpenGL Mathematics es una librería que permite utilizar diversas funciones matemáticas basadas en las especificaciones de GLSL para la comunicación *CPU-GPU.* [15]

A.1.5. Entorno de desarrollo: Microsoft Visual Studio 2013

Visual Studio [16] es un completo entorno de desarrollo (*Integrated Developing Environment* o *IDE* en inglés) de Microsoft para crear aplicaciones web, servicios web, aplicaciones móviles y aplicaciones de escritorio. Soporta múltiples lenguajes de programación y entornos de desarrollo web.

Se ha elegido este entorno de desarrollo por su claridad y facilidad para combinarse con las demás tecnologías elegidas para realizar este proyecto.

A.2. Estructura de la aplicación

Para la creación de la aplicación desarrollada en este proyecto se ha llevado a cabo un diseño modular, utilizando conceptos de programación orientada a objetos. Se han utilizado un pequeño número de clases para gestionar todos los elementos de la aplicación. La mayor parte del peso de la aplicación se lleva a cabo en la clase *main*, mientras que las demás clases se encargan de la inicialización y gestión de elementos necesarios en el curso de la aplicación, como elementos de OpenGL, los *shaders*, las texturas, las imágenes a cargar, o las escenas de prueba elegidas para el renderizado. Así pues, se ha conseguido un diseño sencillo para gestionar la aplicación (ver esquema en la Figura A.1). A continuación se explican las diferentes clases utilizadas en la aplicación junto con su función dentro del proceso:

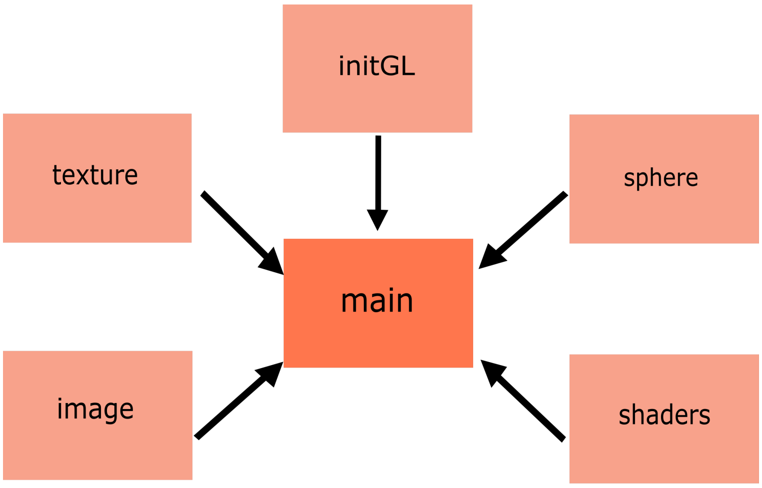


Figura A.1: *Esquema de clases de la aplicación. Como se puede observar, el diseño es bastante simple, basándose en una clase main que centraliza el desarrollo y un pequeño número de clases que gestionan diversos objetos utilizados en la aplicación.*

- ***main***: clase principal, en la que se gestionan todas las fases de la aplicación, y se lleva a cabo la aplicación del *TMO*.
- ***InitGL***: clase que encapsula inicializaciones de elementos necesarios por OpenGL, diferentes funciones o modos, como la habilitación del test de profundidad, o el color de fondo de la escena.
- ***Image***: clase encargada de gestionar la imagen fuente y sus propiedades, como si es esférica o plana, o si su formato es *HDR* o no.
- ***Texture***: clase encargada de gestionar los diferentes tipos de texturas que se utilizan en la aplicación, y sus propiedades (su tamaño, si es esférica o plana, su formato interno...).
- ***Sphere***: clase que permite generar una esfera y asociar parámetros a un *shader* para poder

renderizar una textura correctamente sobre ella. Utilizada para renderizar en un entorno esférico.

- ***Shaders***: clase que gestiona el uso de los diferentes *shaders* que se han necesitado en la aplicación, tanto su creación a partir de un fichero de texto, como su asociación de parámetros.

B. Anexo II: OpenGL

OpenGL es una API (interfaz de programación de aplicaciones, en inglés *Application programming interface*) multiplataforma, creada para el desarrollo de aplicaciones que se basen en gráficos, tanto de dos como de tres dimensiones. OpenGL posee multitud de propiedades, extensiones y características que otras *APIs* similares no tienen. Fundamentalmente OpenGL es una especificación: un documento que describe un conjunto de funciones y el comportamiento que estas han de tener. Partiendo de ella, los fabricantes de *hardware* crean implementaciones: bibliotecas de funciones que se ajustan a los requisitos de una especificación concreta, utilizando aceleración *hardware* siempre y cuando sea posible.

En este anexo se expone información complementaria acerca de características que incluye OpenGL y que se han utilizado en el desarrollo de este trabajo, pudiendo clasificarse en los siguientes temas:

- Renderizado
- Sistemas de coordenadas
- Matrices de transformación

B.1. La *pipeline* gráfica

B.1.1. Introducción

La *pipeline* gráfica define un conjunto de etapas que componen el proceso de renderizado de objetos en OpenGL, desde que es un objeto tridimensional inicialmente, hasta convertirse en una escena visualizable por el usuario en un dispositivo.

En sus inicios, la *pipeline* gráfica tenía un comportamiento fijo, no permitía ningún tipo de parametrización por parte del usuario. Más adelante, el *hardware* evolucionó, determinándose una serie de etapas, y permitiendo a los desarrolladores realizar modificaciones en la secuencia de renderizado mediante la utilización de *shaders*, pequeños programas diseñados para especificar el renderizado de un píxel. Este cambio que permitió una mayor ”programabilidad”del proceso también conlleva una falta de rendimiento, ya que la aceleración por *hardware* de la *pipeline* es más eficiente que por *software*.

B.1.2. Etapas de la *pipeline*

A continuación se describen a alto nivel las etapas que realiza OpenGL en el proceso de renderizado de escenas. Algunas de estas etapas no son imprescindibles para el proceso y no han sido utilizadas para la realización de este proyecto.

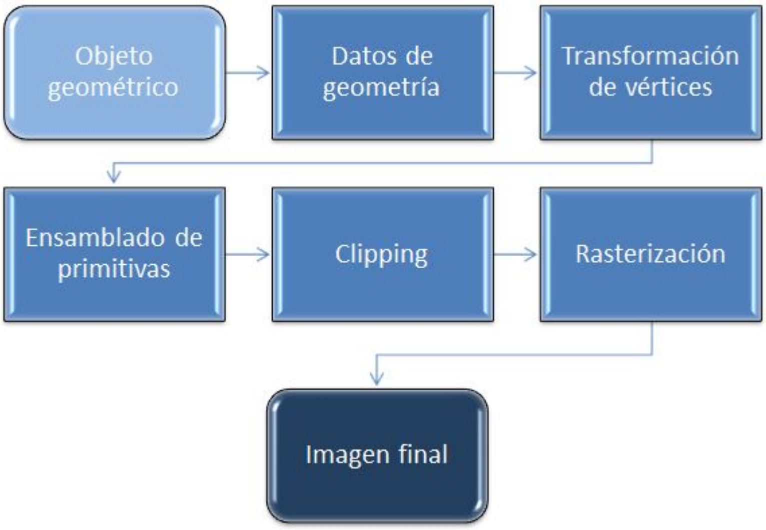


Figura B.1: *Etapas de la pipeline gráfica. Comienza con el paso de los datos geométricos situados en la CPU a la GPU, donde se realizan las demás etapas hasta conseguir la imagen final.*

La primera etapa comienza con los datos geométricos que proporciona el usuario (primitivas geométricas), coordenadas de puntos y líneas. OpenGL trabaja con objetos denominados *Vertex Buffer Objects* que almacenan estos datos en la memoria de la tarjeta gráfica.

El comportamiento de estos datos se programa mediante un *vertex shader*, que realiza cálculos para un solo píxel en paralelo con el resto. La operación más usual en un *vertex shader* es la de modificar el sistema de coordenadas del vértice.

La siguiente etapa es opcional y consiste en procesar los datos obtenidos para incrementar el número de primitivas geométricas para obtener modelos de datos más complejos. Una vez se ha transformado el modelo, es también opcional la utilización de los denominados *geometry shader* con el objetivo de modificar la geometría, actuando sobre las primitivas existentes. En el presente trabajo no ha sido necesaria la realización de estas etapas.

La siguiente etapa se denomina *clipping* y consiste en recortar u ”ocultar” las primitivas que están situadas fuera del rango espacial de visualización de la cámara. Esta etapa está generada automáticamente por OpenGL.

Tras esto, se procede a convertir las primitivas obtenidas en las etapas anteriores a un formato representable en la pantalla; es decir, pasar de las coordenadas de los vértices y triángulos a píxeles de la pantalla. Este proceso se conoce como rasterización.

Finalmente, cada primitiva se asocia con un conjunto de píxeles, lo que genera un fragmento. Cada fragmento dispone de una posición asociada a la ventana. En esta última etapa se calcula el color final de cada píxel, se aplican texturas, se programan iluminaciones... Este comportamiento se programa en los *fragment shader*. Una vez realizados estos pasos, el usuario ya puede visualizar la escena final.

B.2. Sistemas de coordenadas

Uno de los mecanismos más importantes de los que dispone cualquier motor gráfico son las cadenas de transformaciones necesarias para representar un objeto tridimensional en un dispositivo 2D. Datos geométricos como las posiciones de los vértices y las normales del objeto son procesados y

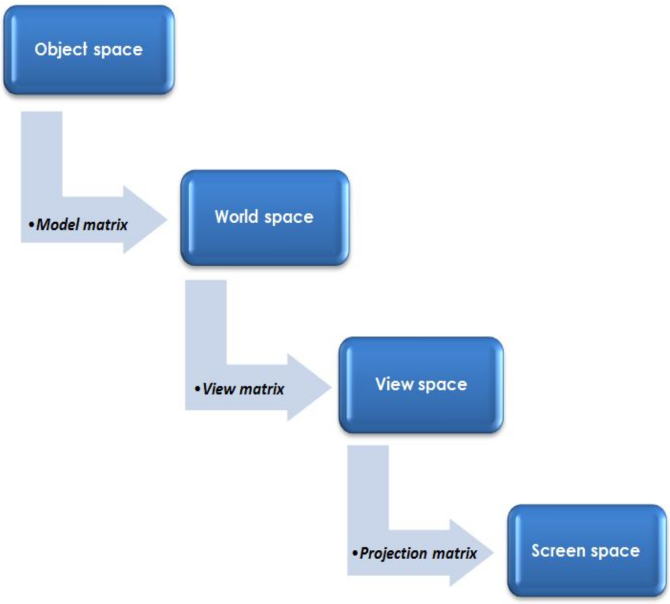


Figura B.2: *Relaciones entre los sistemas de coordenadas de OpenGL*

transformados durante las primeras etapas de la *pipeline gráfica*, hasta el proceso de rasterización. Estas transformaciones dan lugar a diferentes espacios de objeto (representados en la Figura B.2), los cuales han de ser gestionados correctamente para conseguir obtener la imagen 2D final en el dispositivo de salida.

A continuación se describen brevemente los diferentes espacios que se han utilizado.

Object space

Se trata del sistema de coordenadas locales del objeto antes de que ninguna transformación sea aplicada, es decir, su posición y orientación iniciales respecto al origen del objeto. Cuando un artista crea un nuevo modelo 3D, todos los vértices y caras son creados con un sistema de coordenadas relativo a la herramienta que utilizan (la cual suele encontrarse en object space).

Todos los modelos se encuentran en su propio espacio de objeto y si es necesario que dispongan de algún tipo de relación espacial entre ellos, es necesario transformarlos a un espacio común.

World space

Este espacio determina todos los objetos posicionados en el mundo de la escena. Debido a ello, es un sistema de coordenadas absoluto, principal y cuyo origen marca la referencia para absolutamente todo lo que se encuentre dentro de su espacio. La manera de alcanzar este espacio es multiplicando los vértices de cada objeto por una matriz (denominada model matrix), la cual representa la traslación, escala y rotación de cada objeto. Debido a esto, esta matriz es única para cada uno de los objetos.

View space

View space es un espacio auxiliar utilizado para simplificar las matemáticas y mantener todo codificado entre matrices. La idea es que es necesario renderizar hacia el observador, lo que implica proyectar todos los vértices situados en world space en la pantalla, la cual puede estar orientada arbitrariamente en el espacio.

Tomando la cámara centrada en el origen y obteniendo una vista a lo largo del eje Z es posible simplificar en gran medida los cálculos a utilizar.

Screen space

Se trata de un espacio cuboide cuyas dimensiones están comprendidas entre 1 y -1 para cada uno de los ejes. Este espacio muestra el resultado de aplicar una proyección perspectiva correcta a la escena.

En este espacio se suelen producir todas las operaciones de post-procesado de la imagen.

B.3. Multiple Render Targets

En el campo de los gráficos por computador el concepto de *Multiple Render Targets* (MRTs) define una característica que permite a la *pipeline* gráfica renderizar imágenes en múltiples texturas al mismo tiempo.

Estas texturas son utlizadas a menudo como parámetros de entrada a *shaders* o como mapas de textura aplicados a modelos tridimensionales. La aparición del renderizado MRT supuso una revolución en el mundo de los gráficos por computador en tiempo real, debido básicamente al aumento de la eficiencia de los procesos.

En lo que respecta a este proyecto, el renderizado utilizando MRT se utiliza en la etapa de *downsampling* a partir de la técnica de *ping-pong*, ya que en un mismo paso del algoritmo, el *shader* renderiza en una textura el valor medio de la luminancia, mientras que en otra renderiza el valor máximo.

C. Anexo III: Espacios y modelos de color

Para la realización de cualquier *TMO* es necesario calcular la luminancia de la imagen, tanto global como la individual de cada uno de los píxeles que la componen.

Para poder trabajar con luminancias, es necesario convertir la imagen *HDR* a un modelo de color que trabaje con luminancias. A continuación se explican los conceptos de espacio y modelo de color, y se describen los espacios de color utilizados para la realización de este trabajo.

Un espacio de color es un sistema de interpretación del color, es decir, una organización específica de los colores en una imagen o vídeo. Depende del modelo de color en combinación con los dispositivos físicos que permiten las representaciones reproducibles de color, por ejemplo las que se aplican en señales analógicas (televisión a color) o representaciones digitales. Un espacio de color puede ser arbitrario, con colores particulares asignados según el sistema y estructurados matemáticamente.

Un modelo de color es una fórmula matemática abstracta que describe cómo se representan los colores. Para ello, se basa en tuplas numéricas compuestas normalmente por tres o cuatro valores o componentes de color. Los modelos de color más conocidos son el *RGB* y el *CMYK*. Estos modelos, al ser abstractos, no sirven para describir un color concreto sin definir primero la escala o referencia. Son sistemas de color más o menos arbitrarios y sin mucha relación con las necesidades de cada aplicación. Sobre todo, si no tiene una función de asignación asociada a un espacio de color absoluto.

C.1. Modelos de color CIE

La CIE (Comisión Internacional de Iluminación) es la autoridad internacional en cuestiones de luz, iluminación, color y espacios de color. Gracias a esta entidad, podemos entender mejor el funcionamiento del color. La CIE estableció en los años 30 una serie de normas para los diferentes espacios de color que representan el espectro visible. Gracias a estas normas, podemos hacer comparaciones entre los diversos espacios de color de los visores y dispositivos. Para definir al espectador medio y su respuesta al color, la CIE hizo una serie de pruebas sobre una amplia muestra de personas. Definieron un espectador medio, al que denominaron “observador estándar”, con tres tipos de sensores de color que responden a diferentes gamas de longitud de onda. Así, un área de trazado completa de todos los colores visibles, la percibe como una figura tridimensional.

A continuación se explican brevemente las principales características de los modelos de color utilizados en este trabajo para implementar el *TMO*.

RGB

El modelo de color más conocido y utilizado es el *RGB* (siglas del inglés Red Green Blue”, en español Rojo Verde Azul”).

RGB es un modelo de color basado en la síntesis aditiva, con el que es posible representar un color mediante la mezcla por adición de los tres colores de luz primarios. El modelo de color *RGB* no define por sí mismo lo que significa exactamente rojo, verde o azul, por lo que los mismos valores *RGB* pueden mostrar colores notablemente diferentes en diferentes dispositivos que usen este modelo de color. Aunque utilicen un mismo modelo de color, sus espacios de color pueden variar considerablemente.

Las gamas de colores se representan mediante áreas del diagrama de cromaticidad CIE 1931 (ver Figura C.1). El extremo curvado representa los colores monocromáticos. Las áreas de la gama de colores tienen forma triangular porque, en la mayoría de los casos, la reproducción del color se basa en tres colores primarios.

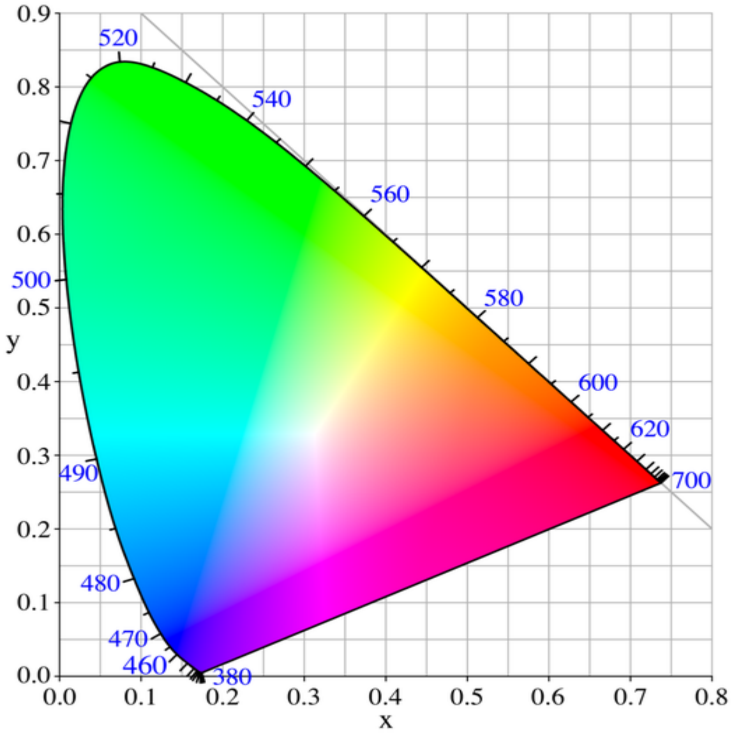


Figura C.1: *Diagrama de cromaticidad del CIE. En este diagrama la curva exterior son los colores espectrales; el resto son colores compuestos. La línea que une dos colores complementarios pasa por el blanco, que se encuentra en la posición (1/3, 1/3). Este diagrama contiene todos los colores visibles por el ojo humano.*

XYZ

La CIE desarrolló el sistema de color XYZ o estándar. En la actualidad, este sistema se sigue usando como referencia para definir los colores que percibe el ojo humano y otros espacios de color. El modelo RGB se basa en colores primarios aditivos. Por el contrario, el XYZ se basa en 3 primarios imaginarios con caracterización espectral (X, Y y Z), que son los que representan el color (ondas electromagnéticas). Éstos se combinan para formar todos los colores visibles por el "observador estándar".

XYY

La CIE quería representar de forma eficaz una figura tridimensional sobre una hoja de papel (bidimensional). Para ello, transformó el espacio tridimensional del color en dos dimensiones artificiales de color o “cromaticidad,” y una de intensidad. Seguidamente, tomaron una porción bidimensional de este espacio y le dieron el máximo nivel de intensidad. Esa porción se convirtió en el diagrama de cromaticidad o “diagrama de cromaticidad CIE xyY”, como puede observarse en la Figura C.1.

La frontera curvada externa es el nicho espectral (o monocromático), con las longitudes de onda mostradas en nanómetros. Téngase en cuenta que esta imagen muestra los colores utilizando el modelo de color *sRGB* (posible en un monitor), y los colores por fuera de dicha gama no pueden ser reproducidos de forma adecuada. Incluso, es posible que dependiendo del espacio de color y la calibración de su propio monitor (o dispositivo de representación de imágenes), los colores *sRGB* tampoco pueden ser apropiadamente representados. Este modelo representa una aproximación utilizando los colores que pueden ser vistos en un monitor o en un televisor.

D. Anexo IV: Gestión del proyecto

En este anexo se analizan las diferentes fases por las que ha pasado este proyecto desde su inicio hasta la realización de este documento final. En el Apartado D.1 se describe la metodología llevada a cabo en el proyecto. En el Apartado D.2 se detallan las diferentes fases del proyecto, junto con un análisis del coste temporal de cada una de ellas. Finalmente se describen las herramientas utilizadas en la gestión y análisis del proyecto (ver Apartado D.3).

D.1. Metodología de trabajo

En este proyecto se ha partido prácticamente de cero en cuanto a algunos aspectos importantes (programación en gráficos, algunas de las tecnologías utilizadas...) lo que ha dificultado el avance en algunas fases del proyecto. Por otro lado, esta circunstancia también ha servido para adquirir un gran conocimiento conforme iba avanzando gradualmente el proyecto y tener una mayor visión del estado durante las diferentes fases.

En la primera fase del proyecto todo el trabajo ha ido encaminado hacia el análisis y la introducción a los conocimientos relacionados con el trabajo y las tecnologías a utilizar, debido a la escasa (o prácticamente nula) experiencia en el ámbito de los gráficos y programación en *GPU* adquirida previamente. Este hándicap existente a lo largo de todas las fases del proyecto, hizo que las primeras fases durasen más tiempo del esperado previamente.
Durante las sucesivas fases (más puramente técnicas) se fueron desarrollando diversos hitos con el fin de llevar un mayor control del estado de la aplicación, esto también hizo más sencillo el proceso de cambio ante dificultades o errores encontrados en algún planteamiento inicial.

Durante todas las etapas del proyecto se ha realizado un control del coste temporal llevado a cabo, así como del grado de dificultad encontrado, así como copias de seguridad tanto internamente como en repositorios externos.

En cuanto a las dificultades encontradas, se ha seguido un proceso de comunicación constante con el tutor mediante reuniones de seguimiento, así como la utilización de diversos foros tecnológicos donde poder buscar y hallar soluciones ante los problemas teóricos y técnicos encontrados.

D.2. Planificación del trabajo

A continuación se describe globalmente la duración temporal del trabajo y de cada una de sus fases. Este proyecto comenzó en el mes de septiembre del año 2014, a principios del curso 2014 - 2015, compaginándose en principio con una asignatura. En este tiempo se llevó a cabo la fase de introducción y aprendizaje con los conceptos del proyecto, así como el inicio del proceso de análisis y desarrollo de la aplicación. A principios del mes de Diciembre este proyecto se compaginó con un trabajo que impedía dedicarle tiempo. Este hecho marca todo el desarrollo del proyecto, ya que ralentizó notablemente el avance.

A continuación se describen brevemente los objetivos y resultados de cada una de las fases del proyecto.

D.2.1. Fase de introducción

Esta fase tenía como objetivo la adquisición de conocimientos teóricos y técnicos necesarios para desarrollar la aplicación y entender los procesos que se debían llevar a cabo. En esta fase se aprendieron conceptos básicos de programación con gráficos, de la *pipeline* gráfica, así como de programación utilizando OpenGL, los *shaders* y demás tecnologías utilizadas.

D.2.2. Fase de análisis

El objetivo en esta fase era analizar en detalle el concepto de *tone mapping*, los diferentes operadores existentes, y su posible implementación utilizando las tecnologías elegidas previamente. Cabe destacar que esta fase, al igual que la anterior, supuso un gran esfuerzo al tratarse de conceptos que no habían sido vistos con anterioridad.

D.2.3. Fase de diseño

En esta fase se buscaba organizar lo que sería la futura aplicación a desarrollar, a partir de los conocimientos previamente adquiridos. Para ello, se marcaron una serie de hitos o logros que irían definiendo el siguiente paso a desarrollar, con el objetivo de tener un mayor control sobre la aplicación en todo momento.

D.2.4. Fase de implementación

Esta fase es puramente técnica, en la que se desarrolla la aplicación. Cabe destacar que esta fase y la anterior están muy relacionadas, ya que se realizaron varias modificaciones en muchos momentos de la implementación. Esta fase incluye también las pruebas realizadas para verificar que la aplicación funcionaba correctamente en todo momento y sin errores.

D.2.5. Fase de documentación

Esta última fase conlleva la creación de esta memoria detallando la información relativa al proyecto, partiendo de la experiencia y el trabajo realizados previamente. Cabe destacar que en todo momento se ha ido obteniendo información de cara a plasmarla en este documento, como enlaces de bibliografía, imágenes de interés y demás conocimientos adquiridos.

D.2.6. Análisis temporal entre fases

El coste temporal aproximado para la realización de este trabajo ha sido de 440 horas. En la Figura D.1 se muestra porcentualmente la importancia, en carga de trabajo temporal (horas), que cada fase ha adquirido en la realización del trabajo. La implementación del algoritmo ha ocupado la mayor parte del tiempo.

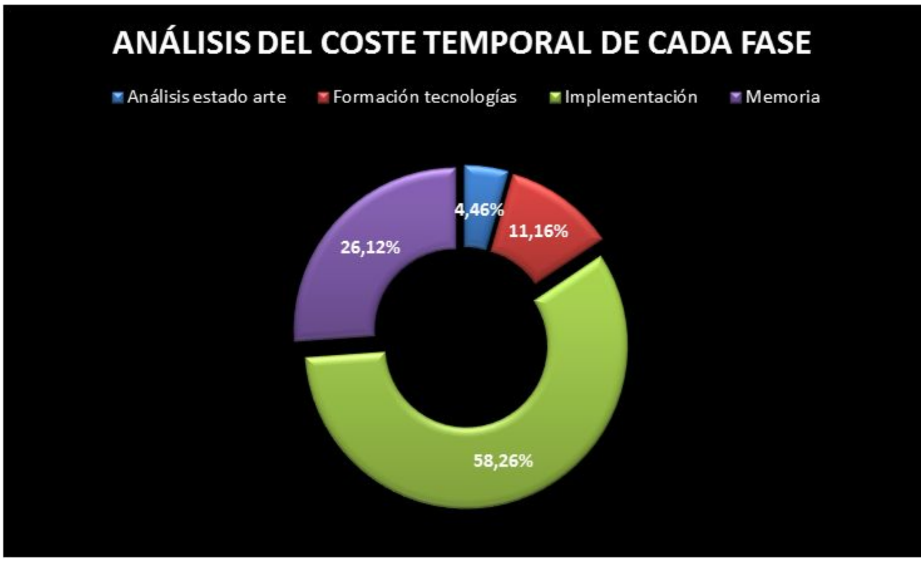


Figura D.1: *Relación de coste temporal de cada fase del proyecto. Como se puede observar, la implementación ha sido la fase que más tiempo ha llevado, seguido de la realización del presente documento.*

D.3. Herramientas utilizadas

En esta sección se presentan las diferentes herramientas y tencologías utilizadas durante el transcurso el proyecto, clasificadas por categorías en función de su ámbito.

D.3.1. Herramientas de desarrollo

Las herramientras utilizadas en el proceso de desarrollo del proyecto aparecen descritas en el Cuadro D.1, junto con su utilización.

D.3.2. Herramientas de documentación

Para el proceso de creación de la presente documentación acerca del proyecto, así como la gestión del proyecto en sus diversas fases, se han hecho uso de diversas tecnologías recogidas en el Cuadro D.2.

Herramienta	**Utilización**
Microsoft Visual Studio	Entorno de desarrollo para desarrollar la aplicación principal en un entorno Windows.
C++	Lenguaje de programación utilizado para el desarrollo de la aplicación.
GLSL	Lenguaje de programación de *shading* utilizado para interactuar (mediante OpenGL) con la *GPU*.

Cuadro D.1: *Herramientas de desarrollo utilizadas en este proyecto.*

Herramienta	**Utilización**
LaTeX	Sistema de composición y edición de textos, orientado a la creación de documentos escritos. Utilizado para crear el presente documento.
Dropbox	Plataforma de alojamiento de archivos multiplataforma en la nube. Utilizado para alojar en todo momento tanto el código desarrollado hasta el momento, como enlaces o imágenes de interés, o el presente documento.
Microsoft Excel	Aplicación diseñada para crear hojas de cálculo, distribuida por Microsoft Office. Utilizada para gestionar el coste temporal del proyecto en cada momento.

Cuadro D.2: *Herramientas de documentación utilizadas en este proyecto.*

D.3.3. Herramientas de diseño

Las herramientas enumeradas en el Cuadro D.3 se han utilizado en el proceso de diseño a alto nivel de la aplicación, de cara a enfocar desde un primer momento cada paso de la aplicación a desarrollar.

Herramienta	**Utilización**
InkScape	Editor profesional de gráficos vectoriales. Se ha utilizado para la realización de gráficos y esquemas contenidos en el presente documento.
Microsoft Word	Archiconocido editor de textos de Microsoft. Se ha utilizado la herramienta *wordArt* que permite crear gráficos y esquemas de distinto tipo.

Cuadro D.3: *Herramientas de diseño utilizadas en este proyecto.*