UNIVERSIDAD NACIONAL DEL SUR

Tesis de Magister en Ciencias de la Computación

Pipeline Gráfico de Superficies Fotorrealista de Tiempo Real

José Ignacio Schneider

BAHÍA BLANCA ARGENTINA

2013

UNIVERSIDAD NACIONAL DEL SUR

Tesis de Magister en Ciencias de la Computación

Pipeline Gráfico de Superficies Fotorrealista de Tiempo Real

José Ignacio Schneider

BAHÍA BLANCA ARGENTINA

2013

Prefacio

Esta Tesis es presentada como parte de los requisitos para optar al grado académico de Magister en Ciencias de la Computación, de la Universidad Nacional del Sur, y no ha sido presentada previamente para la obtención de otro título en esta Universidad u otras. La misma contiene los resultados obtenidos en investigaciones llevadas a cabo en el Departamento de Ciencias e Ingeniería de la Computación, durante el período comprendido entre el 24 de Junio de 2008 y el 15 de Abril de 2013, bajo la dirección de la Dra. Silvia M. Castro, Profesora Titular del Departamento de Ciencias e Ingeniería de la Computación.

José Ignacio Schneider

jis@cs.uns.edu.ar

Departamento de Ciencias e Ingeniería de la Computación

Universidad Nacional del Sur

Bahía Blanca, 15 de Abril de 2013

UNIVERSIDAD NACIONAL DEL SUR

Secretaria General de Posgrado y Educación Continua

La presente tesis ha sido aprobada el . . . /. . . /. . . , mereciendo la calificación de . . . . . (. . . . .)

Agradecimientos

Necesito agradecerle a mucha gente, espero no olvidarme de nadie, porque todos fueron muy importantes en esta etapa, probablemente mucho más importantes de lo que se imaginan.

Quiero agradecerle especialmente a mi directora de tesis, Silvia Castro, por guiarme y ayudarme a darle forma a este trabajo. A Sergio Martig, mi director de beca, que lamentablemente no pudo ver la tesis terminada, pero que sin dudas me ayudó a crecer como investigador. Ambos me ayudaron, entre otras cosas, a expresarme mejor y a organizar mejor las ideas.

A Gustavo Schefer por haberme dado consejos y por haber compartido conmigo incontables horas de discusiones sobre la programación orientada a datos, iluminación y tantos otros temas. A Mehar Gill por haber aportado recursos e ideas que ayudaron a mejorar este trabajo en varios frentes. A mis compañeros del VyGLab que me ayudaron en diferentes etapas de mi postgrado y con los que establecí además una amistad que espero que perdure en el tiempo. A Edgar Bernal, Michael Brown, Soren Hannibal y Federico Lois por sus consejos, puntos de vista y hasta código fuente, que resultaron invaluables.

A la gente que participó en ADVA, ECImag y WAVi que en un momento clave de esta etapa me ayudaron a abrirme al mundo y de esta forma potenciar mi aprendizaje. A la comunidad de CodePlex, GameDev, a Wolfgang Engel, Matt Pettineo y tantas otras personas que con sus aportes ayudan a que todos tengamos acceso a las últimas tecnologías gráficas. A la comunidad de CgCars que me ayudó a potenciar mi lado artístico y por lo tanto a entender mejor qué resultados buscar y esperar.

También necesito agradecer a mi familia y amigos por haberme ayudado a mantener una vida personal en los momentos más duros del proceso de escritura. A los chicos del básquet, con los que todas las semanas compartíamos momentos muy divertidos, y me permitieron establecer el cable a tierra más efectivo. Y a Renguito, por haberme mirado con cara tierna todos los días desde su cucha.

Pero por sobre todo, quiero agradecerle a Deniz, el amor de mi vida, por haberme motivado y presionado cuando más lo necesitaba y por haberme acompañado durante toda la realización de esta tesis.

A todos, ¡¡¡muchas gracias!!!

Resumen

Desarrollar un motor gráfico fotorrealista de tiempo real es un gran reto que involucra relevar e implementar un gran número de tecnologías gráficas complejas. También es necesario diagramar y estructurar los diferentes elementos de la plataforma de software a desarrollar con el objetivo de lograr código eficiente, flexible y mantenible.

En esta tesis se llevó a cabo un relevamiento de tecnologías para la creación de un pipeline gráfico de superficies fotorrealista en tiempo real, enfocándose en aspectos que se consideraron vitales a la hora de establecer los cimientos del pipeline. Específicamente, este trabajo se enfoca en realizar un análisis de los distintos sistemas de color y el tratamiento que debe efectuarse sobre éstos, en introducir las tecnologías de iluminación global de tiempo real contemporáneas más importantes y en realizar un análisis de los distintos pipelines gráficos de iluminación local disponibles.

Además, se diseñó e implementó un *framework* para la rápida generación de aplicaciones gráficas fotorrealistas. Este proyecto se denominó XNA Final Engine y se encuentra desarrollado sobre una plataforma de software de rápido desarrollo, el lenguaje C# y la API gráfica XNA. En éste se implementó un pipeline de iluminación local *deferred lighting*, se incorporaron tecnologías de iluminación global en el espacio de pantalla e iluminación ambiental representada con armónicos esféricos y se representó el color en alto rango dinámico, realizando los cálculos de iluminación en el espacio lineal y aplicando mapeos tonales para su correcta reproducción en pantalla. También se realizó una evaluación de la viabilidad de incorporar los fundamentos de la programación orientada a datos y el desarrollo basado en componentes sobre una plataforma de rápido desarrollo.

Abstract

Developing a real-time photorealistic graphics engine is a challenge that involves the reviewing and implementation of a large number of complex graphics technologies. It is also necessary to outline and structure the different elements of the developed software platform with the aim of achieving efficient, flexible and maintainable source code.

The contribution of this thesis is a survey of technologies for the creation of a real-time surface photorealistic graphic pipeline, focusing on aspects which were considered vital in establishing the foundations of the pipeline. Specifically, this document focuses on the analysis of several color systems, the introduction of the most important contemporary real-time global illumination technologies and the exploration of the different local illumination graphics pipelines available.

In addition, a framework for the rapid generation of photorealistic graphics applications was designed and implemented. This project was named Final XNA Engine and is built on a platform of rapid development, the C# language and the XNA API. In this engine a local illumination deferred lighting pipeline was implemented, screen-space global illumination technologies and ambient lighting represented as spherical harmonics were incorporated; the color was represented in high dynamic range, lighting calculations were done in linear space and a tone mapping was applied to achieve a correct reproduction of the color on the screen. A feasibility study of incorporating the fundamentals of data-oriented programming and component-based development in a platform of rapid development was also performed.

Tabla de Contenidos

Capítulo 1 Introducción

El renderizado fotorrealista consiste en la generación de imágenes por computadora que aparentan ser capturas del mundo real realizadas por una cámara o por el propio ojo humano. Es uno de los desafíos más grandes de la computación gráfica, debido a que el objetivo es realizar (o aparentar) una distribución físicamente correcta de la luz sobre ambientes sintéticos y transformar esta representación de la luz en una imagen que tiene en cuenta la percepción del ojo humano. Esto significa simular la propagación de la luz y la interacción de ésta con las superficies del ambiente sintético y representar y tratar adecuadamente la información de color. Un renderizado de este tipo idealmente requiere una simulación compleja, lo que es inviable en la mayoría de las plataformas, particularmente las de tiempo real. Sin embargo, es posible simplificar el problema realizando aproximaciones o simplemente reemplazando ciertos comportamientos de la luz con algoritmos o técnicas basadas en un estudio perceptual.

En pocos años, la computación gráfica fotorrealista de tiempo real se convirtió de un deseo a una realidad. En la actualidad ya es posible renderizar escenas complejas, utilizando técnicas simplificadas de iluminación global completamente dinámicas, representando el color en alto rango dinámico, simulando interacciones complejas como reflexiones dinámicas y dispersión a nivel de subsuperficie (*subsurface scattering*) y simulando fenómenos perceptuales como la profundidad de campo (Figura 1-1). No obstante, todavía deben realizarse varias simplificaciones. No es posible alcanzar una correcta y completa distribución dinámica de la iluminación indirecta, por lo que es necesario recurrir a un pipeline de iluminación local que anexa aproximaciones de iluminación indirecta. La mayoría de las interacciones de la luz con las superficies se modela sobre superficies sin profundidad, utilizando renderizado volumétrico para sólo un pequeño conjunto de objetos como lo son, por ejemplo, las nubes. Además, la precisión y la resolución en la que se realizan los cálculos y se representa la geometría son típicamente limitadas.

1.1 Motivación

Desarrollar un motor gráfico fotorrealista de tiempo real es un gran reto que involucra relevar e implementar un gran número de tecnologías gráficas complejas. También es necesario diagramar y estructurar los diferentes elementos de la plataforma de software a desarrollar con el objetivo de lograr código eficiente, flexible y mantenible.

Se debe entonces realizar un análisis y evaluación completos de un conjunto de tecnologías gráficas que son clave a la hora de definir las bases de un motor gráfico. Los tipos de pipelines gráficos de iluminación local sobre los que se puede sostener, la representación y tratamiento del color y los métodos para aproximar una simulación de la iluminación global, son posiblemente los mejores candidatos para realizar un relevamiento inicial. Posteriormente, se incorporarán conceptos adicionales que son importantes a la

hora de generar fotorealismo; entre éstos se destacan el renderizado volumétrico y la interacción de la luz con pequeñas partículas suspendidas en el medio (*scattering*), interacciones complejas de la luz con las superficies, sistemas de animaciones, fenómenos relacionados con el enfoque, visión estereoscópica, etc.

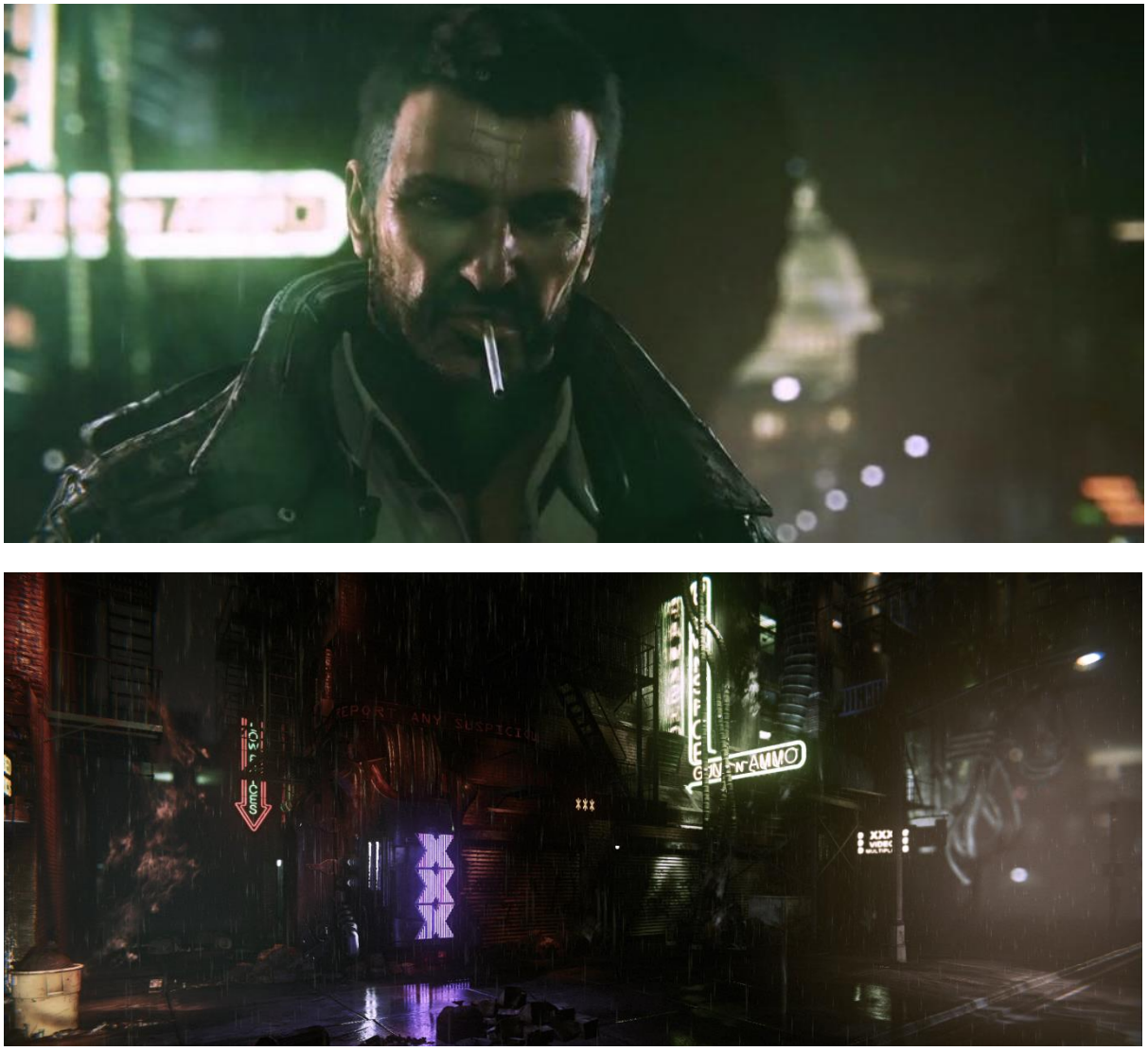


Figura 1-1 Capturas de pantalla de Samaritan, una demostración técnica desarrollada por Epic Games que incluye tecnologías como *subsurface scattering* y reflexiones dinámicas de alta calidad (Mittring & Dudash, The Technology Behind the DirectX 11 Unreal Engine 'Samaritan' Demo, 2011).

A medida que el hardware de tiempo real aumenta en prestaciones, las tecnologías gráficas aumentan en cantidad y complejidad y al mismo tiempo aumenta el tamaño y la complejidad de la arquitectura de software sobre la que se sustenta. Actualmente, las aplicaciones gráficas se desarrollan utilizando la programación orientada a datos y el desarrollo basado en componentes; un paradigma de programación que permite crear una arquitectura de software más efectiva para estas aplicaciones. Sin embargo, todavía no se utilizan lenguajes y APIs gráficas que sigan el paradigma de rápido desarrollo en combinación con este paradigma de programación.

1.2 Contexto de la Tesis

Este trabajo se llevó a cabo en el Laboratorio de Investigación y Desarrollo en Visualización y Computación Gráfica (VyGLab) del Departamento de Ciencias e Ingeniería de la Computación de la Universidad Nacional del Sur. En éste se investigan y desarrollan tecnologías de renderizado de superficies y volúmenes, interfaces no convencionales, realidad aumentada y modelos de visualización unificados.

También se estableció una estrecha relación con la materia Computación Gráfica, perteneciente al plan de Ingeniería en Sistemas de Computación del DCIC, Universidad Nacional del Sur, en la que se realizaron trabajos finales utilizando el motor gráfico diseñado e implementado a lo largo del desarrollo de esta tesis.

1.3 Contribución de la Tesis

La contribución de esta tesis la constituye un relevamiento de tecnologías para la creación de un pipeline gráfico de superficies fotorrealistas en tiempo real, enfocándose en aspectos que se consideraron vitales a la hora de establecer los cimientos del pipeline. Específicamente, este trabajo se enfoca en:

- Realizar un análisis de los distintos sistemas de color de mayor utilidad para tiempo real y el tratamiento que debe efectuarse sobre éstos para mostrar correctamente el resultado en pantalla.
- Introducir las tecnologías de iluminación global de tiempo real contemporáneas más importantes, incluyendo tecnologías estáticas, semidinámicas y completamente dinámicas
- Realizar un análisis de los distintos pipelines gráficos de iluminación local disponibles, mostrando sus ventajas y desventajas.

Simultáneamente, se diseñó e implemento un *framework* para la rápida generación de aplicaciones gráficas fotorrealistas. Este proyecto se denominó XNA Final Engine y se encuentra desarrollado sobre una plataforma de software de rápido desarrollo, el lenguaje C# y la API gráfica XNA. En éste se implementó un pipeline de iluminación local *deferred lighting*, se incorporaron tecnologías de iluminación global en el espacio de pantalla e iluminación ambiental representada con armónicos esféricos, y se representó el color en alto rango dinámico, realizando los cálculos de iluminación en el espacio lineal y aplicando mapeos tonales para su correcta reproducción en pantalla. También se realizó una evaluación de la viabilidad de incorporar los fundamentos de la programación orientada a datos y el desarrollo basado en componentes sobre una plataforma de rápido desarrollo. Esto derivó en una reestructuración del *framework* que demostró que es posible sortear limitaciones, como la administración automática de memoria del lenguaje C#, manteniendo al mismo tiempo las ventajas de estos principios.

1.4 Estructura de la Tesis

En esta tesis incluimos los conceptos relevantes, de modo tal que la lectura de la misma sea autocontenida. A continuación describimos su estructura:

Capítulo 1 Introducción. Se introducen los objetivos y el marco en el que se desarrolla esta investigación. También se enumeran cuáles son las contribuciones de la misma.

Capítulo 2 Luz. Se realiza una descripción física del comportamiento de la luz con el objetivo de comprender qué es necesario modelar y representar. Específicamente se analiza el comportamiento ondulatorio y corpuscular de la luz.

Capítulo 3 Percepción Visual. Se describe la estructura del ojo humano haciéndose énfasis en los fotorreceptores y el procesamiento de las señales que éstos producen. Se analizan además las consecuencias de que la luz sea censada en un rango acotado y dinámico, y se estudia la sensibilidad al contraste y la percepción de color con respecto al contexto.

Capítulo 4 Modelos Computacionales de la Luz. Se presentan los modelos generales que dictan reglas básicas de cómo representar la luz desde los enfoques cuántico, ondulatorio y corpuscular, se introduce la radiometría con el objetivo de disponer un sistema de medición formal que permita cuantificar consistentemente la luz en los algoritmos de iluminación y se discuten los modelos específicos que permitan modelar eficientemente la interacción entre la luz y una superficie, y la propagación de la luz en una escena. Por último, se introducen los algoritmos más importantes de iluminación global y se discute en líneas generales el por qué la iluminación local es atractiva aún si se desean generar gráficos fotorrealistas.

Capítulo 5 Color. Se describe el espacio de color de color RGB dado que es el sistema de color utilizado por los sistemas display actuales. Además, se introduce la transformación gamma, el espacio de color sRGB (derivado del RGB) y se discuten las técnicas más populares de compresión para formatos de color RGB de bajo rango dinámico.

Representar el color en alto rango dinámico es muy importante si se tiene como objetivo lograr un renderizado fotorrealista. Por esta razón, se introducirán las principales opciones a la hora de adoptar un formato de color de alto rango dinámico y se considerará la precisión, rango y costo computacional de estos formatos. Dado el bajo rango dinámico de los dispositivos de display, una representación de color de alto rango dinámico debe ser convertida a un sistema de color RGB de bajo rango dinámico. Dado a que esta conversión es compleja e influencia significativamente la calidad de los resultados, se introducirán un conjunto de técnicas para realizar dicha tarea de manera satisfactoria.

Capítulo 6 Iluminación Global. Se discute el enfoque más simple, la luz ambiental (una luz que se asume que se encuentra infinitamente distante) y los elementos más eficientes para almacenar información de bajo detalle, entre los que se destaca los armónicos esféricos. Luego se introduce la oclusión ambiental,

la oclusión direccional, y un conjunto de técnicas de iluminación global más complejas que van desde la técnica menos adaptable, mapas de luces, pasando por técnicas semidinámicas, volúmenes irradiantes y transferencia de radiancia precomputada, hasta abarcar las técnicas derivadas del algoritmo general introducido en radiosidad instantánea que consideran el dinamismo de los objetos y la iluminación.

Capítulo 7 Pipelines Gráficos. Se describe brevemente el pipeline general implementado en el hardware de las GPUs y se realiza un relevamiento de los pipelines gráficos de iluminación local contemporáneos más importantes y utilizados que se ejecutan eficientemente bajo este hardware.

Capítulo 8 Caso de Estudio: XNA Final Engine. Se introduce XNA Final Engine, un *framework* para la generación rápida de aplicaciones gráficas fotorrealistas. Se analizan las posibles plataformas de desarrollo y las razones de la elección de C# y XNA. Luego se realiza una descripción de la arquitectura general del *framework* y de las tecnologías gráficas implementadas. También, se introduce brevemente el editor y los desarrollos realizados sobre el *framework*. Por último, se realiza un análisis de desempeño del *framework*.

Apéndice A Programación Orientada a Datos y Desarrollo Basado en Componentes. Se analiza por qué la programación orientada a objetos no es un paradigma adecuado para el desarrollo de aplicaciones gráficas complejas y se describen las principales limitaciones del hardware que condicionan a estas aplicaciones. Se presenta la programación orientada a datos que se centra en los datos y su flujo y que permite sortear estas limitaciones, y se introduce el desarrollo basado en componentes que brinda una estructura clara para agrupar un conjunto de datos en entidades. Por último, se describen las tareas de alto nivel a realizar con el objetivo de producir una aplicación gráfica siguiendo estos principios.

Capítulo 2 Luz

En este capítulo se tratarán los fundamentos de la óptica, que permitirán comprender la naturaleza de los fenómenos visuales que deberán simularse si se desea lograr fotorrealismo y poder seleccionar así los modelos de propagación de la luz más adecuados.

En primera instancia se introducirá el espectro electromagnético, destacando al espectro visible, que es la porción de éste que es percibida directamente por el ojo humano. Luego se discutirá la naturaleza de la luz, formulando la propagación de la luz tanto desde el punto de vista de la teoría corpuscular como desde la teoría ondulatoria.

2.1 Radiación Electromagnética y Espectro Electromagnético

La radiación electromagnética es una forma de energía que se propaga a través del espacio y está integrada por campos eléctricos y magnéticos que oscilan en fase y son perpendiculares entre sí y perpendiculares a la dirección de propagación (Glassner, 1995).

El espectro electromagnético comprende el rango de todas las posibles frecuencias de radiación electromagnética. Se extiende desde la radiación de menor longitud de onda, como los rayos gamma y los rayos X, pasando por la luz ultravioleta, la luz visible y los rayos infrarrojos, hasta las ondas electromagnéticas de mayor longitud de onda, como son las ondas de radio (Figura 2-1).

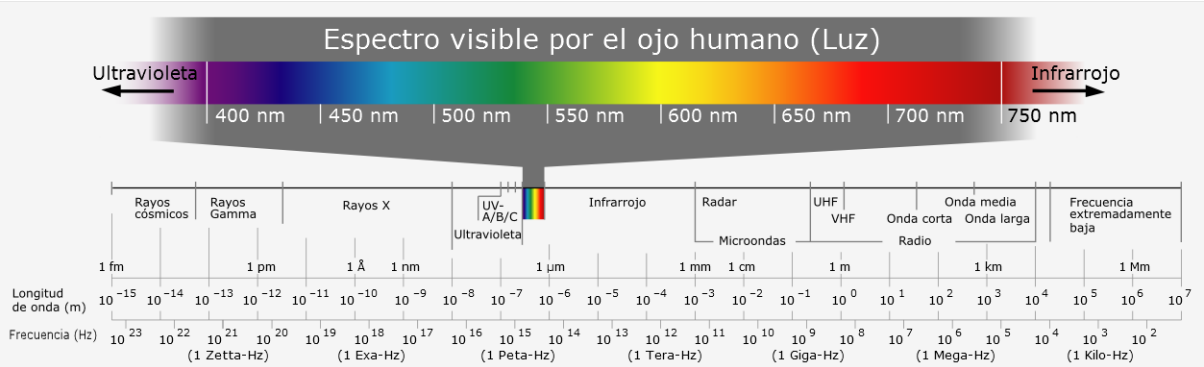


Figura 2-1 Espectro electromagnético. Se destaca el espectro visible (Horst, 2007).

2.2 Luz Visible

Se denomina luz visible (o espectro visible) a la región del espectro electromagnético que el ojo humano es capaz de percibir directamente. No hay límites exactos que marquen los extremos del espectro visible; un ojo humano típicamente responderá a longitudes de onda que abarcan desde 380 a 750 nm, aunque algunas personas pueden ser capaces de percibir longitudes de onda en un rango ligeramente mayor, y en algunos situaciones puntuales se puede llegar percibir luz infrarroja y luz ultravioleta. Debido a las características de los receptores del ojo, una persona puede identificar la radiación electromagnética de distinta longitud de onda y la intensidad de ésta.

2.3 Naturaleza Dual de la Luz

A través de la historia, y aún en la actualidad, se realizaron diferentes investigaciones y experimentos para dilucidar la naturaleza de la luz. Éstas dieron lugar a la formulación de dos teorías bien diferenciadas: la teoría ondulatoria que considera que la luz es una onda electromagnética y la teoría corpuscular que estudia la luz como si se tratase de un torrente de partículas capaces de portar radiación electromagnética.

Algunos comportamientos de la luz sólo se pueden explicar si se considera a la luz como una onda electromagnética (interferencia, difracción, polarización) y otros sólo se pueden explicar si se la considera como partícula (la radiación del cuerpo negro, efecto fotoeléctrico, efecto Compton).

A principios del siglo 20 se establece la mecánica cuántica y se realizan investigaciones analizando fenómenos submicroscópicos que permiten explicar la naturaleza dual de la luz, esto es, la luz se comporta simultáneamente como una partícula y como una onda electromagnética. De esta manera se concilian ambas teorías y se define que la luz son fotones, es decir, pequeños paquetes de ondas localizados físicamente. El fotón se propaga en el espacio de manera ondulatoria y probabilística pero interactúa con la materia como un corpúsculo completo y localizado (Glassner, 1995; Dutré, Bala, & Behaert, 2006).

Es importante remarcar que las investigaciones sobre la naturaleza de la luz todavía no son concluyentes. Además, investigaciones actuales como las realizadas por Carver Mead sostienen que la naturaleza de la luz se puede explicar sólo con la teoría ondulatoria (Mead, 2002); por el contrario, otros investigadores como David Bohm, sostienen actualmente como válida sólo la teoría corpuscular.

2.4 Teoría Corpuscular

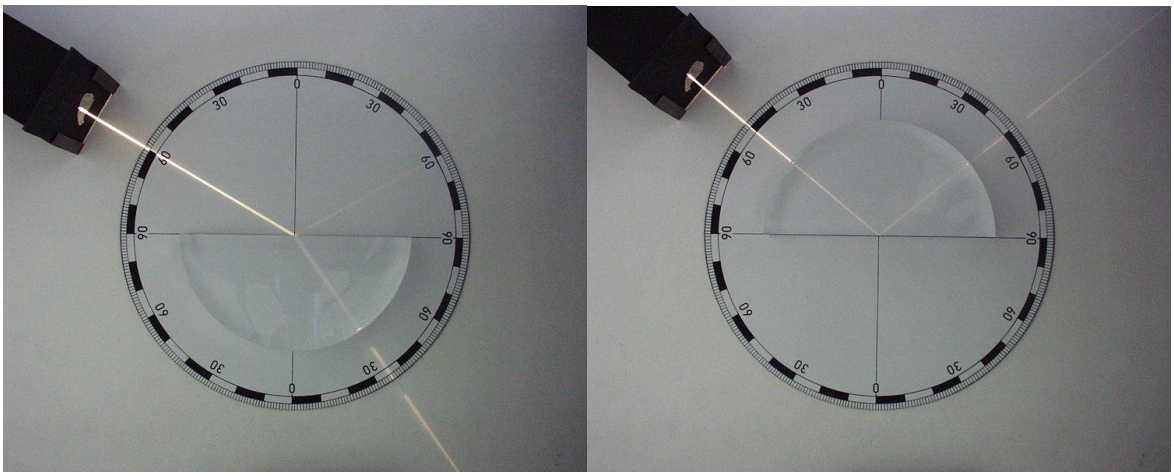
La teoría corpuscular estudia la luz como si se tratase de un torrente de partículas capaces de portar radiación electromagnética. La propagación de la luz se puede formular utilizando el Principio de Fermat que establece que la trayectoria seguida por la luz al propagarse de un punto a otro (también denominado

rayo) es tal que el tiempo empleado en recorrerla es mínimo; cuando se trata de un medio homogéneo, la distancia mínima entre 2 puntos es la recta entre estos puntos. Este enunciado no es completo y no cubre todos los casos, por lo que existe una formulación moderna del principio de Fermat que estipula que el trayecto seguido por la luz al propagarse de un punto a otro es tal que el tiempo empleado en recorrerlo es estacionario respecto a posibles variaciones de la trayectoria.

A partir del principio de Fermat pueden derivarse las leyes de la refracción y de la reflexión. Además, a partir de este principio, también se pueden describir distintos sistemas ópticos con distintos tipos de lentes.

2.4.1 Refracción

La refracción es el cambio de dirección que experimenta una onda debido al cambio de velocidad que se produce al pasar de un medio a otro (Figura 2-2a). Cuanto mayor sea el cambio de velocidad mayor será el cambio de dirección. Este fenómeno sólo se produce si la onda incide oblicuamente sobre la superficie de separación de los dos medios y si éstos tienen índices de refracción distintos. Si el ángulo es perpendicular a la normal del borde de la superficie la onda cambiará de velocidad pero no de dirección.



(a) (b)

Figura 2-2 (a) Refracción, la onda cambia de dirección al pasar de un medio a otro. (b) Reflexión interna total, se produce cuando un rayo de luz se encuentra con un medio de índice de refracción mayor que el índice de refracción en el que éste se encuentra y con un ángulo tal que no es capaz de atravesar la superficie entre ambos medios reflejándose completamente (Sándor, 2005).

La Ley de Snell, o Ley de Refracción, se deriva del principio de Fermat y describe la relación entre los ángulos de incidencia y de refracción. Concretamente afirma que el producto del índice de refracción del primer medio por el seno del ángulo de incidencia es igual al producto del índice de refracción del segundo medio por el seno del ángulo de refracción, es decir:

$$n_1 \sin\theta_1 = n_2 \sin\theta_2 \qquad \text{(Ecuación 2-1)}$$

donde, n_1 y n_2 son los índices de refracción de los medios, θ_1 es el ángulo de incidencia y θ_2 es el ángulo de refracción. Esta ecuación es simétrica, lo que implica que las trayectorias de los rayos de luz son reversibles. Es decir, si un rayo incide sobre la superficie de separación con un ángulo de incidencia θ_1 y se refracta sobre el medio con un ángulo de refracción θ_2, entonces un rayo incidente en la dirección opuesta desde el segundo medio con un ángulo de incidencia θ_2 se refracta sobre el medio 1 con un ángulo θ_1.

La refracción es un fenómeno que se puede apreciar fácilmente cuando se sumerge parcialmente un objeto en el agua (Figura 2-3). El aire tiene un índice de refracción aproximado de 1.0003 y el agua tiene un índice de refracción aproximado de 1.33; esta diferencia produce un cambio de dirección de la luz cuando ésta ingresa al agua.

Figura 2-3 La diferencia en los índices de refracción del agua y el aire produce el cambio de dirección de la luz cuando ésta se propaga de un medio a otro.

También se produce cuando la luz atraviesa capas de aire a distinta temperatura. Este fenómeno puede apreciarse comúnmente en rutas (Figura 2-4) o en la salida de escape de una turbina.

Figura 2-4 En las rutas, la diferencia de temperatura en el aire puede producir efectos de refracción y de reflexión interna total.

Un rayo de luz que se propaga en un medio con índice de refracción n_1 e incide con un ángulo θ_1 sobre una superficie que lo separa de un medio de índice n_2, con $n_1 > n_2$, puede reflejarse totalmente en el interior del medio de mayor índice de refracción (Figura 2-2b). Este fenómeno se conoce como reflexión interna total o ángulo límite y se produce para ángulos de incidencia θ_1 mayores que un valor crítico cuyo valor es:

$$\theta_c = \arcsin\frac{n_2}{n_1} \qquad \text{(Ecuación 2-2)}$$

Si $n_1 > n_2$, entonces $\theta_1 > \theta_2$. Eso significa que cuando θ_1 aumenta, θ_2 llega a 90° antes que θ_1, punto en el que el rayo refractado sale paralelo a la frontera. Si θ_1 aumenta aún más, θ_2 no puede ser mayor que 90°, por lo que no hay refracción al otro medio y la luz se refleja totalmente. Los espejismos, la ilusión óptica en la que los objetos lejanos aparecen reflejados en una superficie lisa como si se estuviera contemplando una superficie líquida que en realidad no existe, son un ejemplo de reflexión interna total (Figura 2-4).

La longitud de una onda también incide en el cambio de dirección; a menor longitud de onda mayor desviación. Este último comportamiento puede apreciarse con el fenómeno conocido como dispersión cromática, fenómeno en el cual un rayo de luz blanca o policromática (que contiene todas las longitudes de onda) atraviesa un prisma y se produce la separación de la luz en sus diferentes longitudes de onda (Figura 2-5).

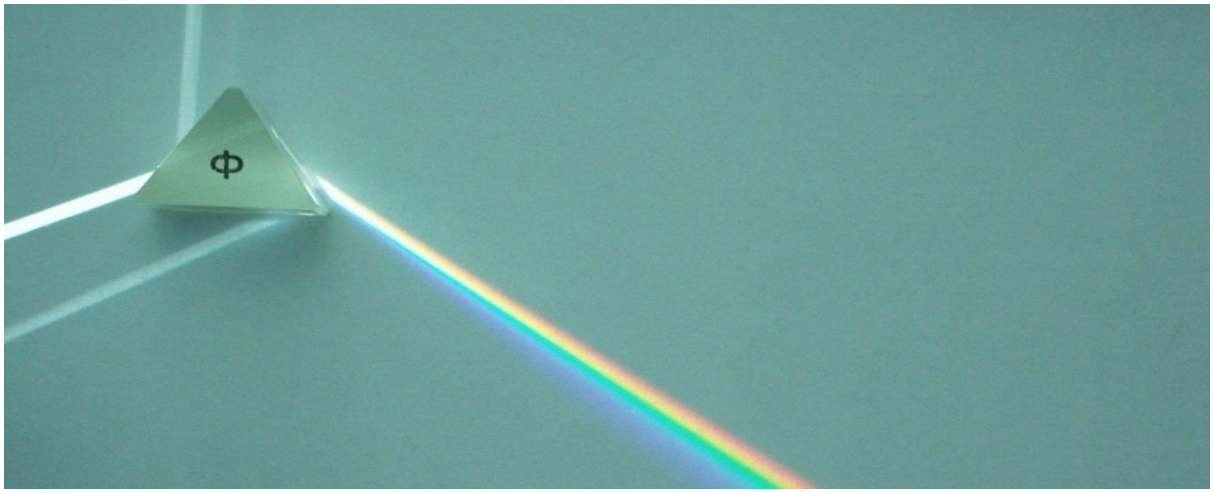


Figura 2-5 Al pasar la luz blanca por un prisma ésta se descompone en los diferentes colores del espectro visible. Las ondas rojas son las que menos se desvían y las violetas las que más lo hacen (Sándor, 2005).

Otro fenómeno provocado por la dispersión cromática es el arcoíris. Cuando los rayos del sol atraviesan las gotas de humedad contenidas en la atmósfera terrestre, éstas actúan como prismas, dispersando la luz en todas sus longitudes de onda. Similarmente, el cielo se ve de color azul debido a que cuando la luz solar intercepta la atmósfera, las partículas de polvo y gases presentes actúan como prismas. Los rayos de longitudes de onda larga (rojos y amarillos) casi no se desvían, mientras que los rayos violetas y azules son

los más desviados. Éstos varían su trayectoria, y vuelven a chocar con otras partículas, y así sucesivamente, rebotando una y otra vez hasta que alcanzan el suelo terrestre. Por esta razón, cuando la luz llega a los ojos el cielo se percibe como azulado (no se percibe violeta porque la luz del sol contiene más luz azul que violeta y porque el ojo humano es más sensible a la luz azul).

2.4.2 Reflexión

La reflexión es el cambio de dirección de una onda que ocurre en la superficie de separación entre dos medios, de tal forma que la onda regresa al medio inicial y con la misma longitud de onda (Figura 2-6). Del Principio de Fermat se puede derivar la Ley de Reflexión que sostiene que el ángulo de incidencia del rayo con respecto a la normal es igual al ángulo del rayo reflejado con respecto a la normal y además el rayo reflejado y el incidente están en lados opuestos a la normal:

$$\theta_1 = \theta_2 \qquad \text{(Ecuación 2-3)}$$

donde θ_1 es el ángulo de incidencia y θ_2 es el ángulo de reflexión.

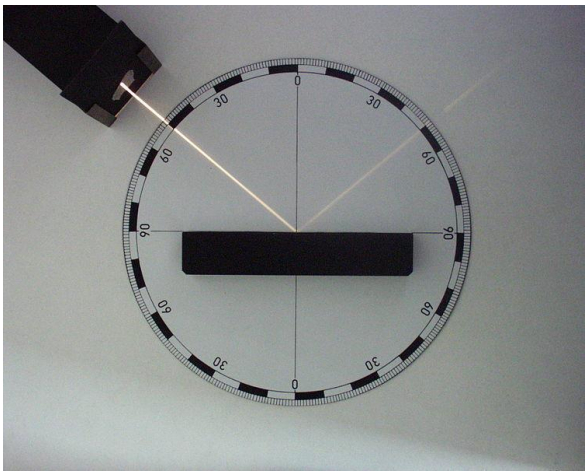


Figura 2-6 Reflexión de la luz sobre una superficie (Sándor, 2005).

2.4.3 Radiación del Cuerpo Negro y Efecto Fotoeléctrico

Existen varios fenómenos de la luz que sólo pueden ser demostrables con la teoría corpuscular. Según el orden histórico, el primer efecto que no se pudo explicar por la concepción ondulatoria de la luz fue la radiación del cuerpo negro.

Un cuerpo negro es un radiador teóricamente perfecto que absorbe toda la luz que incide en él; la radiación incidente no es reflejada ni atraviesa el cuerpo negro. A pesar de su nombre, el cuerpo negro emite luz dado que todos los cuerpos con temperatura superior al cero absoluto emiten radiación electromagnética, siendo su intensidad dependiente de la temperatura y de la longitud de onda considerada. Por lo tanto, el cuerpo negro se convierte en un emisor de radiación térmica que constituye un sistema físico idealizado para el estudio de la emisión de radiación electromagnética. Ningún objeto puede ser un cuerpo negro perfecto, todo objeto emite o absorbe mejor unas longitudes de onda de luz que otras. Estas pequeñas variaciones dificultan el estudio de la interacción de la luz, el calor y la materia. Afortunadamente, es posible construir un cuerpo negro prácticamente perfecto.

A principios del siglo XX, los científicos Lord Rayleigh, y Max Planck (entre otros) estudiaron la radiación de cuerpo negro utilizando un dispositivo similar al anteriormente mencionado. Tras largas investigaciones, Planck fue capaz de describir perfectamente la intensidad de la luz emitida por un cuerpo negro en función de la longitud de onda y la temperatura. Específicamente concluyó que a medida que se incrementa la temperatura de un cuerpo negro, la cantidad total de luz emitida también aumenta y el pico de intensidad se desplaza hacia longitudes de onda menores. Por ejemplo, una barra de hierro se vuelve naranja rojiza cuando se calienta a altas temperaturas y su color se desplaza progresivamente hacia el azul a medida que se calienta más.

En ese entonces la teoría ondulatoria era la teoría dominante. Al aplicar las teorías clásicas a la radiación de cuerpo negro se obtenía una curva teórica de la radiación emitida, pero ninguna curva teórica coincidía con la curva real. La más conocida era la propuesta por Lord Rayleigh en 1900, y perfeccionada por Sir James Jeans en 1905. Esta se deducía de manera lógica a partir de las teorías conocidas, pero la curva que se obtenía a partir de la fórmula de Rayleigh-Jeans se ajustaba muy bien para longitudes de onda largas, pero divergía hacia el infinito para longitudes de onda cortas. Rayleigh, Jeans y Einstein rápidamente concluyeron que la fórmula teórica era imposible, lo que dio lugar a que la ley propuesta por Rayleigh y Jeans se conociese “catástrofe ultravioleta”, pues la divergencia se producía para pequeñas longitudes de onda, en la región ultravioleta del espectro.

Sin embargo, cinco años antes, Max Planck había formulado una ley que hasta ese entonces fue ignorada por la comunidad científica y que permitió posteriormente construir una teoría que explica dicho fenómeno siendo actualmente uno de los pilares de la física cuántica. Ésta establecía que los minúsculos osciladores que componían la materia (en términos actuales los átomos) no podían tener una cantidad de energía arbitraria, sino sólo valores discretos entre los que no era posible la existencia de un valor intermedio. Específicamente postuló que la luz de frecuencia ν es absorbida en múltiplos enteros de un cuanto de energía igual a $h\nu$, donde h es una constante física universal denominada Constante de Planck.

En 1905, Albert Einstein utilizó la base de la teoría cuántica recién desarrollada por Planck para explicar otro fenómeno no comprendido por la física clásica, el efecto fotoeléctrico. Este efecto puede describirse del siguiente modo: cuando un rayo monocromático ilumina la superficie de un material (metales, no metales, líquidos y gases), se desprenden electrones en un fenómeno conocido como fotoemisión o efecto fotoeléctrico externo. Este fenómeno tiene algunas particularidades. Si un rayo monocromático no produce el efecto fotoeléctrico, un rayo monocromático de igual frecuencia pero mayor

intensidad tampoco lo produce, sin importar cuán intenso sea. Por el contrario, si un rayo monocromático produce el efecto fotoeléctrico, un rayo monocromático de igual frecuencia pero menor intensidad también lo produce, sin importar cuan débil sea. La cantidad de electrones emitidos es proporcional a la intensidad del rayo incidente, pero la energía emitida por éstos no depende de la intensidad sino de la frecuencia del rayo incidente, aumentando a medida que la frecuencia lo hace. El efecto fotoeléctrico se manifiesta para radiaciones electromagnéticas de frecuencia igual o mayor que un cierto umbral; este umbral varía dependiendo del tipo de superficie. Además, la emisión de los electrones es inmediata ya que la superficie no acumula energía para disparar los electrones en una etapa posterior; el efecto fotoeléctrico se produce como un efecto directo e inmediato de la interacción entre la radiación electromagnética y la superficie.

Einstein, quien había respaldado la hipótesis de Planck del cuerpo negro, utilizó el mismo planteo para dar una explicación lógica al efecto fotoeléctrico. Postuló que la energía de un rayo de luz está cuantizada en pequeños paquetes llamados fotones (Einstein originalmente los llamo *Lichtquant*, o cuantos de luz), cuya energía es equivalente a hc/λ, donde h es la constante de Planck, c es la velocidad de la luz y λ es la longitud de onda. Cada fotón interactúa con cada electrón independientemente y cuando lo hace, éste puede transferir toda su energía al electrón; dado que la transferencia no puede ser parcial, si la energía del fotón es insuficiente para liberar al electrón de la superficie, la transferencia no se realiza. Debido a que la energía del fotón está directamente relacionada con su energía y la interacción ocurre independientemente por cada fotón, si la frecuencia es baja la energía será insuficiente para liberar el electrón. Esto encaja perfectamente con las características del efecto fotoeléctrico.

El efecto fotoeléctrico por sí solo no demuestra la existencia de fotones, pero junto con evidencias que surgen de otros experimentos se pudo avalar esta interpretación. Entre estos experimentos se encuentran el efecto Compton, el efecto Zeeman, y el efecto Raman (Glassner, 1995; Gómez-Esteban, 2007).

2.5 Teoría Ondulatoria

La teoría ondulatoria estudia la luz como una onda electromagnética. La propagación de la luz se formula utilizando las teorías de propagación de ondas, en las que se destacan el Principio de Fresnel-Huygens y el Principio de Superposición, que son soluciones a las ecuaciones de Maxwell.

El principio de Fresnel-Huygens afirma que cada punto libre de obstáculos de un frente de onda[1] actúa como una fuente secundaria de ondas esféricas de frecuencia coincidente con la de la fuente. El siguiente

[1] Un frente de onda es una superficie en la que la onda tiene la misma fase en un instante dado. Los frentes de onda pueden visualizarse como superficies que se desplazan a lo largo del tiempo alejándose de la fuente sin tocarse. Un frente de onda plano es aquél en el que los rayos son paralelos entre sí. Este modelo es muy útil para representar secciones de un frente de onda esférico muy grande en relación con la sección. Por ejemplo, la luz del sol llega a la tierra con una frente de onda de radio aproximado de 93 millones de millas, por muchas razones prácticas este frente de onda se lo suele considerar plano.

frente de onda será la superficie envolvente de todas las ondas secundarias (Figura 2-7). La repetición del proceso da como resultado la propagación de la onda a través del medio (Andrews & Phillips, 2005). Además, a medida que el frente de onda avanza y su superficie aumenta, la energía se distribuye y por lo tanto hay menos energía por unidad de superficie. Específicamente la intensidad en un frente de onda esférico disminuye de manera inversamente proporcional al cuadrado de la distancia a la fuente. Este comportamiento se denomina *atenuación geométrica*.

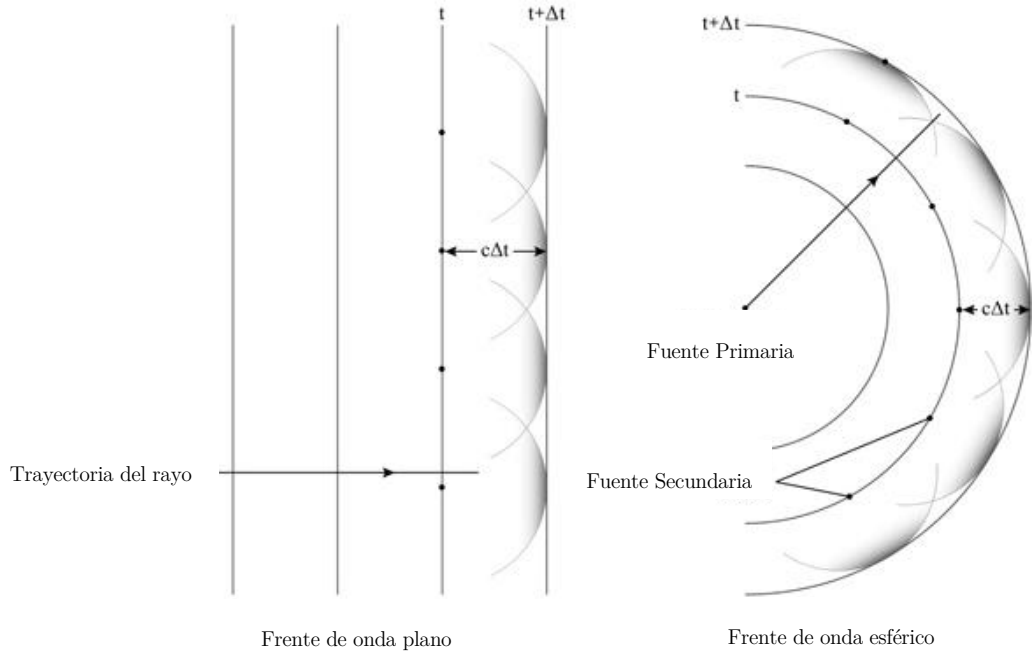


Figura 2-7 Propagación de ondas en un frente de onda plano (izquierda) y un frente de onda esférico (derecha).

Por otra parte, el principio de superposición afirma que cuando dos o más ondas se solapan en el espacio, éstas se interfieren ya sea constructivamente o destructivamente. La interferencia es aditiva, es decir que las formas de una onda se suman para producir una tercera. Si se solapan dos ondas con la misma longitud de onda y la diferencia de fase entre ellas es exactamente la mitad de la longitud de onda, las ondas se cancelarán produciéndose una interferencia destructiva máxima. En cambio, si están exactamente en fase, la onda producida tendrá doble amplitud produciendo una interferencia constructiva máxima (Figura 2-8). La interferencia sólo es temporal, las ondas no son afectadas individualmente y continuarán su propagación con su misma amplitud, frecuencia y dirección. Fresnel utilizó las propiedades del fenómeno de interferencia para completar la teoría Huygens, en la que se omitió una justificación adecuada para afirmar el por qué las ondas no viajan en sentido contrario.

Normalmente, dada la frecuencia a la que la oscila la luz (~1014 Hz), solo la variación de intensidad puede apreciarse ante un fenómeno de interferencia Sin embargo, algunos materiales sumamente finos (donde el grosor de los mismos es cercano a las longitudes de onda del espectro visible) pueden causar interferencia visible, por ejemplo las bandas de colores que se aprecian sobre las burbujas. Cuando la luz atraviesa una burbuja, una pequeña parte de ella se refleja y la mayor parte pasa por el medio agua con jabón; cuando se vuelve a encontrar con el medio aire refleja nuevamente sólo otra pequeña cantidad. Esto

sucede con casi todos los medios transparentes; muchas veces podemos ver un reflejo doble cuando una imagen se refleja en un vidrio común, pero una burbuja es millones de veces más delgada que un vidrio, y su delgadez es casi perfectamente regular y muy similar a las longitudes de onda de la luz que refleja. Por lo tanto cuando el primer reflejo se intercepta con el segundo reflejo, ambas ondas están ligeramente desplazadas en fase una de la otra y se produce un fenómeno de interferencia que es fácil de percibir (la refracción también influye en los colores que percibimos al ver una burbuja).

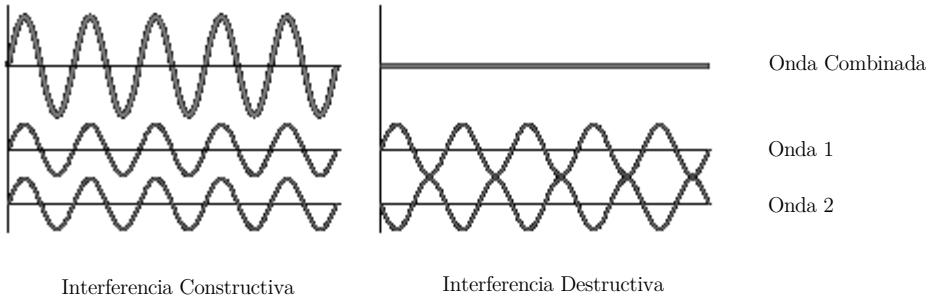


Figura 2-8 Representación gráfica de la interferencia entre dos ondas.

Fenómenos de la luz como la difracción, refracción y reflexión pueden ser explicados utilizando la teoría de propagación de ondas.

2.5.1 Difracción

El fenómeno conocido como difracción consiste en la desviación aparente de la dirección de propagación de una onda cuando encuentra algún obstáculo en su camino y se hace más notorio cuando las dimensiones del obstáculo son del orden de la longitud de onda o menor (Figura 2-9). La difracción en esencia es un fenómeno de interferencia y propagación de las ondas. De hecho, la difracción, desde el punto conceptual, como se expuso en la sección anterior, se manifiesta en todo momento aún si no hay obstáculos en su camino. Se hace más aparente cuando la luz pasa por ejemplo por una rendija o un obstáculo puntiagudo porque predominará la nueva dirección de propagación frente a la original. Este fenómeno también es responsable de algunas sombras suaves en objetos pequeños.

El fenómeno de la difracción de la luz, por lo tanto, contradice la hipótesis de la propagación rectilínea de la luz. Sin embargo, por simplicidad y eficiencia, y dado que los fenómenos de difracción podrían ser despreciables en la mayoría de los escenarios, muchos modelos consideran que los rayos de luz se propaga en línea recta. Si este fenómeno necesita modelarse, la fórmula de difracción de Kirchhoff provee un fundamento matemático riguroso.

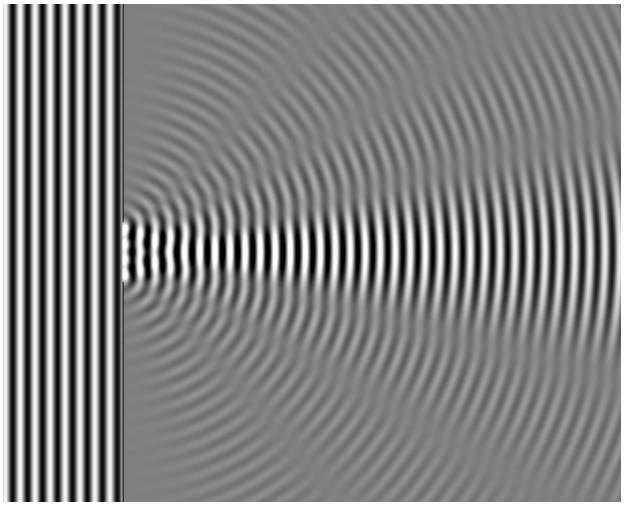


Figura 2-9 Un frente de ondas plano se encuentra con un obstáculo pequeño produciendo aparentes desviaciones de propagación. Este efecto se lo conoce como difracción (Lyon, 2009).

2.5.2 Reflexión

El comportamiento del fenómeno de reflexión también puede derivarse del Principio de Fresnel-Huygens. La Figura 2-10 muestra tres rayos de luz de un frente de onda plano interceptando una interfaz, lo que produce un nuevo conjunto de ondas esféricas en cada punto de intersección. Cada rayo intercepta la interfaz a distinto tiempo y las nuevas ondas viajan a la misma velocidad por estar en el mismo medio, por lo tanto las ondas esféricas formadas tendrán distinto tamaño. Según el Principio de Fresnel-Huygens el nuevo frente de onda es tangente al conjunto de ondas esféricas que lo forman, por lo tanto, el nuevo frente de onda se mueve en una dirección que forma el mismo ángulo que la onda de incidencia con respecto a la superficie.

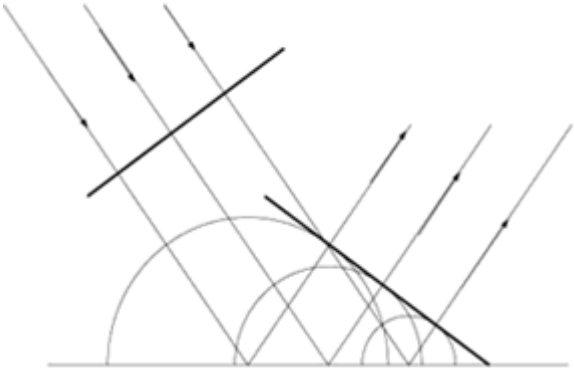


Figura 2-10 Explicación gráfica del fenómeno de reflexión utilizando el Principio de Fresnel-Huygens (Marshall, 2006).

2.5.3 Refracción

De forma análoga, el comportamiento del fenómeno de refracción también puede derivarse del Principio de Fresnel-Huygens. La diferencia clave entre los dos fenómenos se encuentra en que mientras que el primer rayo que intercepta la interfaz altera su velocidad por estar en un medio con distinto índice de refracción, los restantes aún se desplazan a la velocidad inicial por estar en el medio original, por lo tanto la dirección del nuevo frente de onda se altera de forma directamente proporcional al índice de refracción (Figura 2-11).

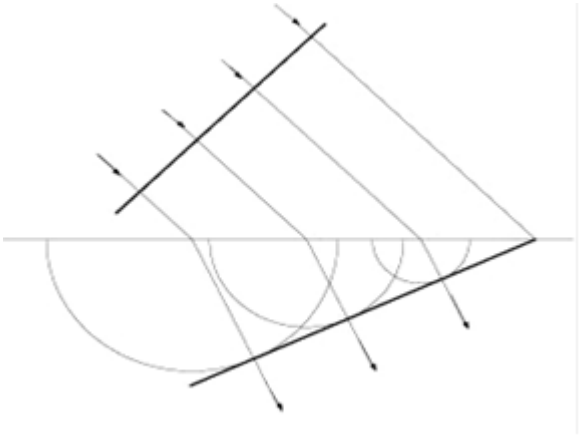


Figura 2-11 Explicación gráfica del fenómeno de refracción utilizando el Principio de Fresnel-Huygens (Marshall, 2006).

2.5.4 Polarización

La polarización electromagnética es un fenómeno que puede producirse cuando, en las ondas electromagnéticas los campos eléctricos oscilan sólo en un plano determinado. La luz natural no suele estar polarizada dado que está constituida por una serie de trenes de ondas procedentes de átomos distintos; en cada uno de estos trenes de ondas el campo eléctrico oscila en un plano determinado, siendo la orientación de éstos distinta entre sí.

El sistema visual humano no percibe la polarización de la luz; sin embargo, se pueden fabricar filtros que solo dejen pasar luz polarizada en una dirección establecida. Entre estos filtros podemos mencionar los filtros utilizados en fotografía y los lentes anti-reflejos (*anti reflex*); ambos reducen la cantidad de reflejos indeseados y filtran la luz polarizada proveniente del cielo azul (Figura 2-12). La tecnología adoptada actualmente en los cines, en la que se trata de simular la visión estereoscópica, utiliza lentes con filtros para luz polarizada. El lente derecho permite el paso de luz con cierta polarización y el lente izquierdo con otra distinta.

(a) (b)

Figura 2-12 (a) Fotografía tomada sin filtro polarizador. (b) Fotografía tomada con filtro polarizador (Namek, 2006).

2.6 Absorción

Cuando la luz llega a una superficie, esta puede refractarse, reflejarse o puede ser absorbida por la materia. Si ocurre esto último, la energía absorbida se transforma en otro tipo de energía, como calor o energía eléctrica. La luz es una forma de energía y por lo tanto sigue las leyes de física que gobiernan la transformación de la energía. Una de las propiedades más importante es la conservación de energía, esto es, la energía nunca se crea ni se destruye, solo se transforma (Feynman, 1998).

La razón por la que un material se ve de un color es porque este absorbe la luz incidente con ciertas longitudes de onda y re-emite el resto. Justamente la luz re-emitida es las que percibimos y es la que le da el color al material.

Capítulo 3 Percepción Visual

El sistema visual humano está constituido por los ojos y la porción del cerebro que procesa las señales provenientes de los mismos. Juntos, el ojo y el cerebro convierten la información óptica en la percepción visual de una escena. El objetivo de la computación gráfica es producir imágenes para ser vistas por el ser humano, por lo tanto, la información relevante debe transmitirse a éste lo mejor posible. De esta forma, se analizará la percepción visual en los usuarios, en particular la percepción de color.

En este capítulo se describirá la estructura del ojo humano haciéndose énfasis en los fotorreceptores y el procesamiento de las señales que éstos producen. Se analizarán además las consecuencias de que la luz sea censada en un rango acotado y dinámico. Por último, se estudiará la sensibilidad al contraste y la percepción de color con respecto al contexto.

3.1 Estructura del Ojo Humano

La luz que entra al ojo atraviesa varios componentes del mismo (Figura 3-1). El primero de ellos, la córnea, es la estructura hemisférica y transparente localizada al frente del ojo que permite el paso de la luz y protege al iris y al cristalino. Posee propiedades ópticas de refracción significativas que orienta u enfoca la luz incidente hacia la siguiente capa del ojo. La mayor parte de la capacidad total de enfoque del ojo se encuentra en la córnea, pero el enfoque que realiza es fijo.

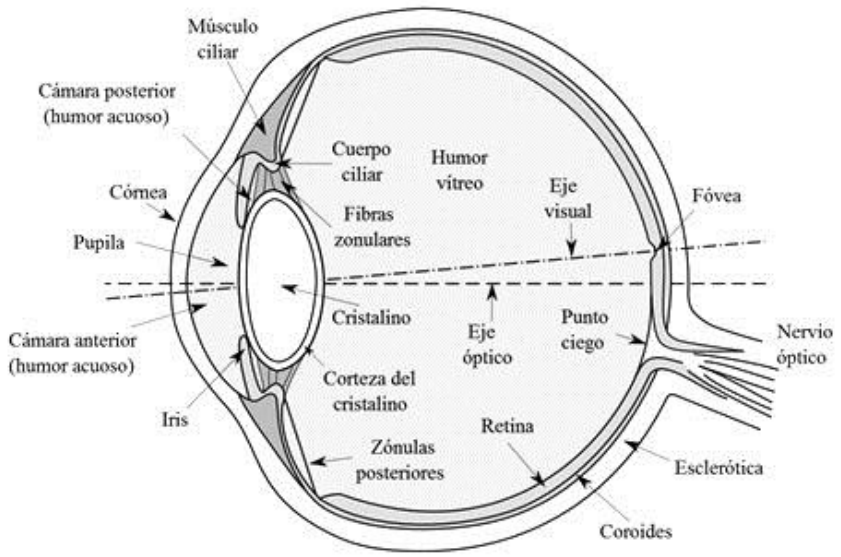


Figura 3-1 Estructura biológica de un ojo humano (Moorthy, 2006).

Antes de llegar a la pupila, la luz atraviesa el humor acuoso, el líquido transparente que se encuentra en la cámara anterior del ojo y que sirve para nutrir y oxigenar las estructuras del globo ocular que no tienen aporte sanguíneo, como la córnea y el cristalino. El humor acuoso refracta levemente la luz.

Posteriormente la luz pasa por la pupila, una abertura dilatable y contráctil que tiene la función de regular la exposición, es decir, la cantidad de luz que pasa a los componentes posteriores del ojo. En esencia funciona igual a un obturador de una cámara fotográfica, que controla la cantidad de luz entrante para impedir tanto la subexposición como la sobreexposición.

A continuación la luz atraviesa el cristalino, un lente biconvexo cuya función principal consiste en permitir enfocar objetos situados a diferentes distancias. Este objetivo se consigue mediante un aumento de su curvatura y de su espesor, proceso que se denomina acomodación. El cristalino se caracteriza por su alta concentración en proteínas, que le confieren un índice de refracción elevado y por ende una gran capacidad de enfoque. Como se describió en el capítulo anterior, cuando ocurre el fenómeno de refracción la luz de diferentes longitudes de onda se desvía levemente en distintas direcciones; por lo tanto, es posible que se produzca un efecto indeseado de dispersión cromática, denominado aberración cromática. Por ejemplo, un círculo blanco podría verse como un arcoíris circular debido a este fenómeno.

Antes de llegar a la retina, la luz atraviesa el humor vítreo, un líquido gelatinoso y transparente que rellena el espacio comprendido entre la superficie interna de la retina y la cara posterior del cristalino y que al igual que el humor acuoso tiene capacidades leves de refracción.

Por último, la luz alcanza la retina, la capa de tejido blando y sensible a la luz que recubre la parte interna posterior del globo ocular. La luz que incide en la retina desencadena una serie de fenómenos químicos y eléctricos que finalmente se traducen en impulsos nerviosos que son enviados al cerebro por el nervio óptico. Estas señales se procesan y se produce entonces la percepción visual de la escena.

3.2 Fotorreceptores del Ojo Humano

En la retina se encuentran los fotorreceptores, células especializadas en detectar la luz incidente y transformarla. Los fotorreceptores se dividen en dos categorías principales: los conos y los bastones (Guyton & Hall, 2001).

La mayoría de los conos se concentran en una región cercana al centro de la retina llamada fóvea, y la proporción desciende a medida que nos acercamos a la periferia. Es por esta razón que el sistema visual humano, como el de la mayoría de los animales depredadores, dispone de la visión foveal que permite ver con mayor nitidez el área central de lo que se está observando.

La cantidad total de conos presentes en un ojo humano es de aproximadamente 4.5 millones según mediciones probabilísticas (Oyster, 1999) y existen tres tipos de conos, cada uno más sensible de forma selectiva a ciertas longitudes de onda determinadas (Figura 3-2). Esta sensibilidad específica se debe a la presencia de unas proteínas llamadas opsinas. En los conos están presentes tres tipos de opsinas: la eritropsina, la cloropsina y la cianopsina. La eritropsina tiene mayor sensibilidad a las longitudes de onda

largas de alrededor de 564–580 nm (luz amarilla) y los conos que las contienen son denominados conos L. La cloropsina tiene mayor sensibilidad a las longitudes de onda medias de alrededor de 534–545 nm (luz verde) y los conos que las contienen son denominados M. Y por último, la cianopsina alcanzan mayor sensibilidad a las longitudes de onda pequeñas de entre 420–440 nm (luz azul) y los conos que la contienen se denominan conos S (Hunt, 2004).

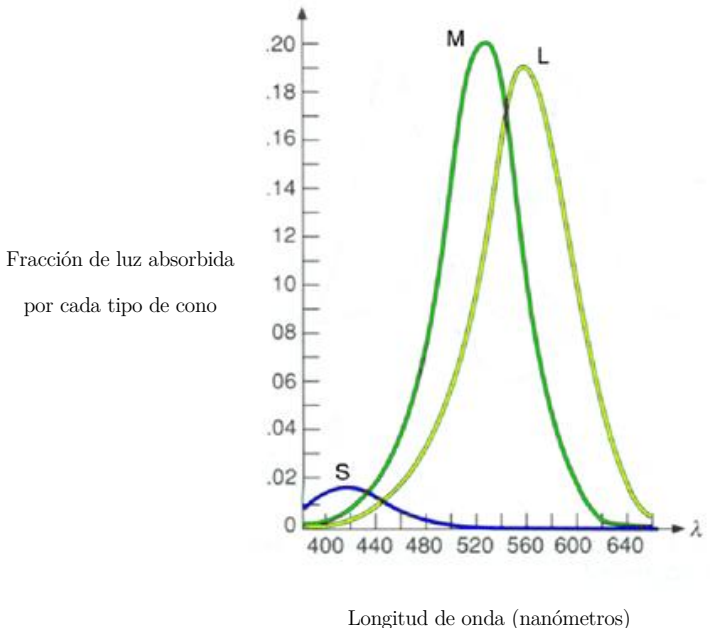


Figura 3-2 Sensibilidad aproximada de los conos del ojo humano a las frecuencias de luz visible.

Las señales producidas por los tres tipos de conos permiten que el cerebro clasifique la luz incidente en colores. Por ejemplo, el color amarillo se percibe cuando los conos L se estimulan levemente más que los conos M, y el color rojo se percibe cuando los conos L se estimulan significativamente más que los conos M. Similarmente, los colores azules y violetas se perciben cuando los conos S se estimulan más que los otros dos.

Los conos son menos sensibles a la luz que los bastones por lo que funcionan mejor cuando la intensidad de la luz es de moderada a intensa, es decir proporcionan la visión fotópica. Además, son más sensibles a la luz verde/amarilla que a otros colores debido a que la proporción de conos M y L es mayor que la de conos S. Los conos perciben mayor detalle que los bastones y se adaptan más rápidamente al cambio rápido en la imagen. Dada su forma de conexión a las terminaciones nerviosas que se dirigen al cerebro, son los responsables de la definición espacial, es decir, a los detalles de alta frecuencia de la imagen.

Los bastones, por otro lado, se concentran en zonas alejadas de la fóvea y son los responsables de la visión escotópica, es decir, la visión que se produce con niveles muy bajos de iluminación. Los bastones poseen una opsina que se denomina rodopsina y que alcanza su pico de sensibilidad a los 498 nm (color azul del espectro), aproximadamente entre la sensibilidad máxima de los conos S y M. Los bastones no permiten distinguir los colores, pero son más sensibles que los conos a la intensidad de la luz, por lo que

aportan a la visión aspectos como el brillo y el tono. Además, los bastones comparten las terminaciones nerviosas que se dirigen al cerebro y, por consiguiente, su aporte a la definición espacial resulta de poca importancia. La cantidad de bastones en un ojo humano es de aproximadamente 100 millones.

La señal producida por un bastón o un cono es simplemente una indicación de que la luz incidió y estimuló el fotopigmento; no se transmite información que describa la longitud de onda del fotón. Sin embargo, el cerebro puede recuperar/estimar parte de esta información dado que la probabilidad de que un fotorreceptor genere una señal es función de la sensibilidad espectral del fotorreceptor y la intensidad de la luz incidente. Por ejemplo, si un receptor es 30% sensitivo a una longitud de onda e iluminación ambiental particular, entonces de cada 100 fotones que incidan en el fotorreceptor, 30 generarán una señal. La combinación de resultados de las distintas señales generadas por los distintos tipos de fotorreceptores permite estimar la intensidad y la longitud de onda de la luz incidente (Glassner, 1995).

En efecto, el sistema visual humano descarta información para posteriormente estimarla, y la razón por la que esto sucede está ligada principalmente a la mejora de la resolución espacial. Si en la retina hubiera más tipos de fotorreceptores distintos (como ocurre en algunas aves que tienen de 5 a 7 tipos de fotorreceptores) entonces en una región fija el número de fotorreceptores sensibles a una banda de frecuencias específica sería necesariamente menor. El número de fotorreceptores diferentes y su densidad sólo pueden aumentar a expensas del otro. Esto también se refleja en el propio centro de la fóvea humana donde no hay conos S, solo hay conos M y L debido a que nuestro sistema visual humano evolucionó como el resto de los animales depredadores en el que la resolución espacial de nuestro punto de enfoque es más importante que la información de color.

Además, las señales producidas en los conos y bastones duran varios milisegundos dado que son reacciones químicas. Por esta razón, la señal de un fotorreceptor es en realidad un valor promediado en el tiempo, efecto que se denomina suavizado temporal (*temporal smoothing*). Este efecto influye, por ejemplo, en la percepción de las luces que parpadean. Cuando el parpadeo es lento, se pueden percibir los destellos de luz individuales. Pero a partir de una cierta frecuencia, llamado frecuencia de parpadeo crítica (*critical flicker frequency* o *CFF*), los destellos se perciben como una luz continua sin parpadeos.

Los bastones son más lentos que los conos en responder a cambios de la luz y sus señales duran más tiempo. Mientras que los conos requieren aproximadamente 60 destellos por segundo para lograr la fusión de imágenes, los bastones pueden llegar a lograrlo con tan solo 4 destellos por segundo. Por lo tanto, la frecuencia a la que el parpadeo se hace notorio depende de varios factores; si el parpadeo está ocurriendo en la visión perimetral o los niveles de intensidad de la luz ambiental son bajos, por ejemplo, los bastones serán los sensores predominantes y el CFF será menor. La frecuencia o color de la luz incidente y su intensidad también pueden afectar el CFF.

La frecuencia de actualización de las pantallas está íntimamente ligada a este fenómeno. Una pantalla de tubo de rayos catódicos (CRT) normalmente opera entre 60 y 120Hz, y la mayoría de las personas no detectan los parpadeos por sobre los 75Hz. En cambio, las pantallas LCD no presentan el fenómeno de parpadeo a pesar de que su tecnología de actualización de imágenes no suele superar los 60Hz. Esto se debe principalmente a que la retroiluminación (*backlight*) que utilizan estas pantallas opera aproximadamente a 200Hz.

La frecuencia de parpadeo crítica no debe confundirse con el fenómeno indeseable que se produce cuando un conjunto de imágenes en movimiento no alcanza la tasa de actualización necesaria para ser percibida de manera fluida (o dicha actualización es irregular), aún cuando la pantalla que las reproduce trabaja en frecuencias de actualización más altas. Este fenómeno está influenciado principalmente por el procesamiento de las señales que realiza el cerebro. Para que un conjunto de imágenes en movimiento parezca continuo la frecuencia de actualización debe superar los 16Hz, es decir, los 16 cuadros por segundo. En el cine moderno se utiliza una tasa de actualización de 24 cuadros por segundo, en la televisión se utiliza una de 25 a 30 cuadros por segundo (según la norma adoptada por cada país) y para las imágenes digitales en tiempo real normalmente se utiliza una de 30 a 60 cuadros por segundo, dependiendo de la uniformidad de distribución de los cuadros, las características de la aplicación y las preferencias de sus creadores.

El cerebro humano realiza un procesamiento complejo de las señales generadas por los fotorreceptores, corrigiendo y estimando información según distintos factores. Por ejemplo, el cerebro reduce o elimina ruido en las señales según su intensidad, y puede estimar convincentemente la información que no pudo censarse en el punto ciego de la retina, lugar donde no hay fotorreceptores por la presencia de los canales nerviosos que transmiten la información al cerebro.

3.2.1 Proceso de Adaptación Temporal

El ojo humano es sensible a aproximadamente 9 órdenes de magnitud o miles de millones de niveles de brillo distintos, y responde desde las $\sim 10^{-6}$ hasta las $\sim 10^{6}$ candelas por metro cuadrado (cd/m^2)[2] (Hood & Finkelstein, 1986). No obstante, en un determinado momento, el ojo humano sólo puede distinguir una cantidad de niveles de brillo mucho menor, en general de alrededor de 5 órdenes de magnitud (Figura 3-3). El ojo se adapta a la intensidad de la luz reconociendo el brillo en un rango acotado pero que mantiene el detalle, el contraste y la percepción de intensidad; a este proceso se lo conoce como *proceso de adaptación temporal*. Sin embargo la adaptación no es inmediata y varía desde una fracción de segundo hasta varios minutos según la diferencia de intensidades a las que se tiene que adaptar.

La adaptación se logra parcialmente debido a que la pupila ajusta la cantidad de luz que pasa hacia la retina. Sin embargo, el mecanismo más importante en el proceso de adaptación temporal se realiza en la retina, a través de los bastones y los conos. Los bastones son aproximadamente 10 veces más sensibles a la intensidad de la luz que los conos y por lo tanto tienen mayor participación en el proceso de reconocimiento de la intensidad de la luz, razón por la cual se censa mejor la intensidad de la luz en niveles

[2] Candelas por metro cuadrado es la unidad de luminancia, una medida fotométrica que mide el flujo luminoso por unidad de ángulo sólido por unidad de superficie. El flujo luminoso es la cantidad de energía luminosa emitida por una fuente de luz ajustada para reflejar la sensibilidad del ojo humano a diferentes longitudes de onda. Intuitivamente la luminancia representa una cantidad infinitesimal de flujo luminoso contenido en un rayo de luz que incide o sale de un punto en una superficie en una dirección dada.

bajos de luminancia. Los bastones son más lentos en adaptarse y podrían tardar de 20 a 30 minutos en hacerlo, mientras que los conos se pueden adaptar completamente como máximo en 10 minutos (Glassner, 1995); ésta es la razón principal por la que la velocidad del proceso de adaptación es asimétrica. Sin embargo, si bien el proceso de adaptación completo puede durar varios minutos, la mayor parte de la adaptación se realiza en cuestión de segundos (Fairchild & Reniff, 1995).

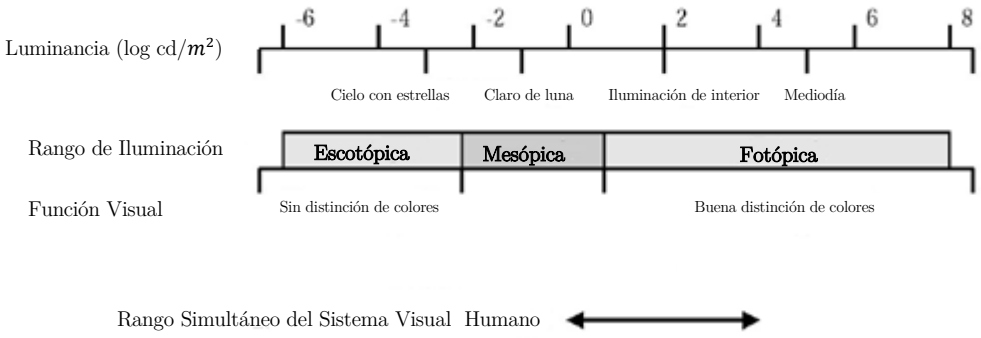


Figura 3-3 Luminancia aproximada bajo fuentes de iluminación típicas. Se muestra además el tamaño aproximado del rango simultáneo del sistema visual humano (Ledda, Chalmers, Troscianko, & Seetzen, 2005).

A bajos niveles de luminancia, los bastones son altamente sensibles a la amplitud y frecuencia de la luz incidente; es probable que una cantidad reducida de fotones produzca una señal, y un cambio en la longitud de onda promedio seguramente producirá un cambio en la respuesta. Los bastones funcionan específicamente entre los $\sim 10^{-6}$ y los $\sim 10^{1}\ cd/m^2$, por sobre ese umbral se saturan y dejan de emitir señales. En cambio, los conos funcionan entre los $\sim 10^{-1}$ y los $\sim 10^{6}\ cd/m^2$; pasado ese umbral también se saturan y dejan de emitir una señal útil (Hood & Finkelstein, 1986).

La velocidad de adaptación, la capacidad de sensibilidad de los fotorreceptores y el suavizado temporal de las señales del sistema visual humano dan lugar al fenómeno de deslumbramiento (*glare*). Este fenómeno ocurre cuando los fotorreceptores se saturan y el proceso de adaptación temporal no tiene la rapidez suficiente o la capacidad de adaptarse a esa intensidad de luz. Por ejemplo, si se mira al sol, los conos no pueden censar la intensidad lumínica aún cuando la pupila reduce significativamente la incidencia de luz. Similarmente, en escenas nocturnas, los bastones son los sensores predominantes. Ante la presencia de un destello de luz potente los bastones se saturan y el sistema visual humano debe regenerar las opsinas de los conos, pero esta regeneración no es inmediata por lo que momentáneamente se tiene una ceguera parcial o total, dependiendo de la intensidad del destello.

El hecho de que los sensores no funcionen uniformemente puede producir un efecto denominado *efecto de Purkinje*. Supongamos que se mira al atardecer a una flor roja o amarilla con hojas verdes oscuras, cuando el sol está sobre el horizonte, los conos del ojo están activos, y la flor roja o amarilla se percibirá más brillante o iluminada que las hojas debido a que los colores amarillo y rojo están cerca del pico de sensibilidad de los conos L, los conos con mayor presencia en nuestra retina. Cuando el sol se pone la intensidad de la luz disminuye y los bastones comienzan a ser los sensores predominantes, y dado que estos

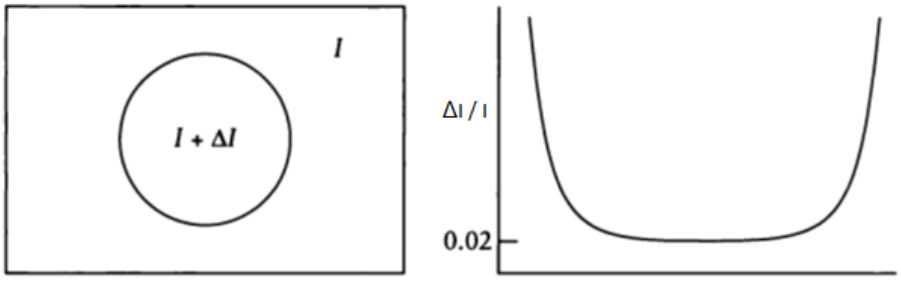
son más sensibles a longitudes de onda cortas, las hojas verdes se percibirán más iluminadas en relación con la flor (Figura 3-4).

(a) (b)

Figura 3-4 (a) Flor iluminada por el sol. (b) Simulación de la misma imagen bajo poca iluminación. Cuando los conos son los sensores predominantes (a), los pétalos se perciben más intensos/iluminados que las hojas, pero cuando los bastones son los sensores predominaste (b), la percepción es inversa (Zander, 2010).

3.2.2 Sensibilidad al Contraste

La respuesta a cambios de intensidad de la luz no es lineal. Supongamos que un observador está mirando una hoja de papel que refleja una intensidad lumínica I y sobre ésta hay dibujado un círculo con intensidad $I + \Delta I$ (Figura 3-5a), donde ΔI es la menor diferencia de intensidad que puede ser percibida para una intensidad I. Sobre un amplio rango de intensidades el cociente $\Delta I / I$ (llamado el cociente Weber) es aproximadamente constante con un valor cercano a 0,02 (Figura 3-5b). La curva que grafica I contra la función del cociente Weber se denomina *función de sensibilidad al contraste* (CSF).



(a) (b)

Figura 3-5 (a) Configuración experimental utilizada para caracterizar la discriminación de brillo (b) Función de sensibilidad al contraste (Pratt, 2001).

Esta curva sugiere que el sistema visual humano responde a relaciones de intensidades y no a valores absolutos. Además, si con estos resultados se construye una curva que representa la respuesta absoluta a la sensibilidad al contraste según la intensidad de la luz se obtendrá una función logarítmica (Figura 3-6) (Pratt, 2001); es decir, a medida que la intensidad de la luz aumenta la sensibilidad al contraste disminuye.

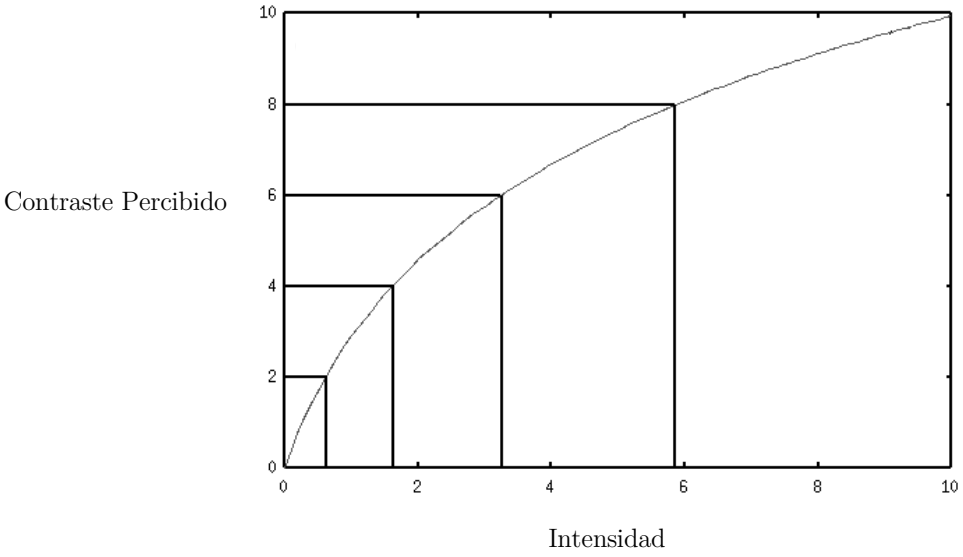
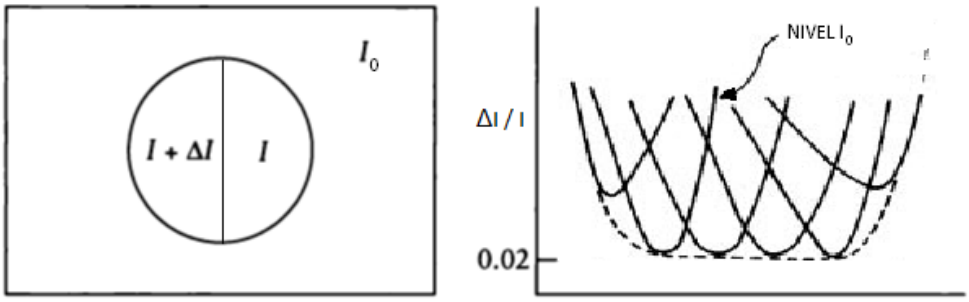


Figura 3-6 Contraste percibido aproximado según la intensidad de la luz.
A medida que la intensidad aumenta la sensibilidad al contraste disminuye.

La sensibilidad al contraste también depende de la intensidad lumínica del contexto. Supongamos que un observador está mirando una hoja de papel que refleja una intensidad lumínica I_0 y sobre ésta hay dos hojas de papel más pequeñas con intensidad I e I + ΔI respectivamente (Figura 3-7a). El cociente de Weber para este experimento (Figura 3-7b) se mantiene aproximadamente constante en un rango mucho menor que en el experimento anterior pero diverge de forma similar. El rango en el cual es aproximadamente constante para un contexto lumínico arbitrario es además comparable con el rango dinámico de la mayoría de los sistemas de pantalla modernos.

Asimismo, el sistema visual humano tiende a subestimar o sobreestimar la intensidad lumínica en las regiones adyacentes al límite de regiones de diferente intensidad. Supongamos que un observador está mirando un conjunto de barras verticales, cada una de las cuales refleja uniformemente la misma intensidad lumínica (o lo que es lo mismo, tiene un color constante); la variación de intensidad entre una barra y otra es equivalente a una constante arbitraria y las barras están ordenadas de la más brillante a la izquierda hacia la más oscura a la derecha (Figura 3-8). Aunque la intensidad es constante en cada barra, el lado izquierdo de las barras se verá más oscuro y el derecho más brillante, acentuándose los bordes de cada barra. Similarmente, si tenemos una transición suave entre dos barras de colores constantes, entonces se verán barras verticales inexistentes (Figura 3-9). Ambos patrones fueron creados por Ernst Mach y son conocidos como bandas de Mach.



(a) (b)

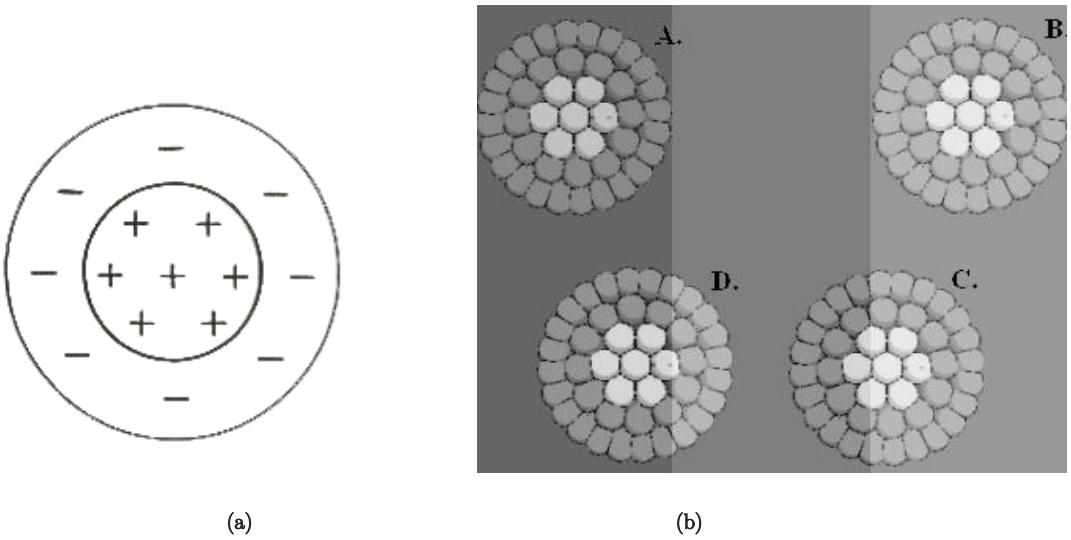
Figura 3-7 (a) Configuración experimental utilizada para caracterizar la discriminación al brillo. (b) Función de Sensibilidad al Contraste considerando el contexto lumínico. Cada función es un contexto lumínico distinto (Pratt, 2001).

Figura 3-8 Bandas de colores constantes que decrementan su intensidad en escalones constantes.

Figura 3-9 Transición suave de color entre dos bandas de colores constantes.

Esto es debido a la *inhibición lateral*; ésta tiene relación con el funcionamiento de las células ganglionares de la retina. Estas células actúan como integradores y ponderadores de las señales transmitidas por los fotorreceptores. La integración está organizada espacialmente en lo que se denominan campos receptivos, por lo que la disposición geométrica de los fotorreceptores influye en la interpretación de sus señales. En la región del centro de cada campo receptivo, las células ganglionares suman la respuesta de los fotorreceptores, mientras que restan las señales de los fotorreceptores que se encuentran en el anillo circundante, pero dentro de la región de integración (Figura 3-10a).

En la Figura 3-10b se muestra conceptualmente cuatro de estas regiones sobre tres de las barras verticales. La celda A esta completamente sobre la banda más oscura y la celda B sobre la más brillante. El centro aditivo de la celda D se encuentra en la banda más oscura pero su anillo sustractor se encuentra parcialmente sobre la banda intermedia, dado que la sustracción es menor en la celda D que en la A, la celda D se percibirá levemente más oscura. Similarmente, el sector aditivo de la celda C está sobre la banda más brillante, pero su anillo sustractor se encuentra en la banda intermedia, por lo que la sustracción será mayor en comparación con la celda B y el color percibido será levemente más brillante (Glassner, 1995; Sekuler & Blake, 1994).



(a) (b)

Figura 3-10 (a) Diagrama conceptual de la célula ganglionar (b) Análisis neuronal conceptual de las bandas de Mach. (Sekuler & Blake, 1994).

La iluminación percibida no es simplemente una función de la intensidad, como lo muestran los distintos efectos de contraste. Supongamos que un observador está mirando un conjunto de bandas de Mach en las que sobre cada una de las bandas se encuentra un cuadrado, todos con la misma intensidad. A pesar de que cada cuadrado tiene la misma intensidad, en la banda oscura éste se percibirá como más brillante y en la banda brillante se percibirá como más oscuro (Figura 3-11). Este fenómeno se lo conoce cOmo *contraste simultáneo.*

Figura 3-11 Todas las regiones interiores tienen el mismo color pero se perciben diferentes.

El *contraste consistente*, permite aceptar o reconocer una escena con diferentes niveles de iluminación. Supongamos que un observador mira un libro utilizando una determinada lámpara; las hojas reflejarán una

cantidad de luz X y la tinta reflejará una cantidad de luz Y. Si en cambio, se utiliza una lámpara con el doble de intensidad la tinta reflejará el doble de luz, pero no se percibirá doblemente brillante debido a que la intensidad de la hoja también se duplicó y la relación de intensidad se mantuvo constante. Similarmente, la *adaptación cromática* permite compensar los cambios en el color de los objetos al cambiar no solo la intensidad de la iluminación sino también el color predominante de la fuente de luz. Es decir, un objeto se percibirá del mismo color bajo la luz del sol, la luz de una fogata o la luz de una lámpara (Figura 3-12). La única excepción a este fenómeno es el efecto Purkinje, pero esta variación es menor y no afecta significativamente el proceso de reconocimiento de color que realiza el cerebro humano.

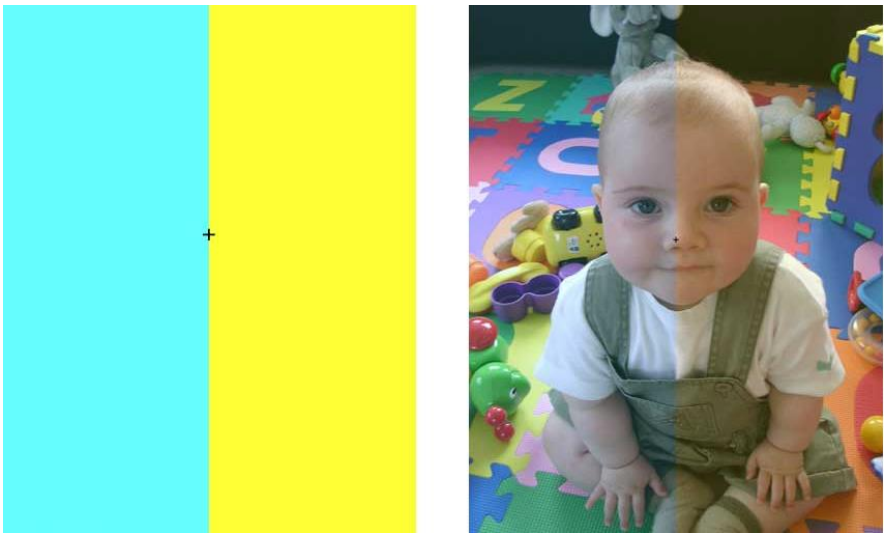


Figura 3-12 En la imagen de la derecha se pueden reconocer los colores sin importar que lado se mire. Además, si se mira fijo el signo + de la imagen de la izquierda por 30 segundos y luego inmediatamente se mira la imagen de la derecha la distorsión de color desaparecerá momentáneamente (Van den Bergh, 2004).

Las características previamente definidas son importantes para el sistema visual humano dado que permiten mantener una consistencia de la imagen mental que se tiene del mundo a pesar de cambios dramáticos de iluminación.

Por último, el *contraste de color* es el fenómeno por el cual los colores se ven con distinta intensidad en diferentes contextos o ambientaciones (Figura 3-13). Se estima que este suceso se origina por la forma en que la información de los fotorreceptores se propaga hacia el cerebro. La información de color censada por los fotorreceptores se transmite a través de tres canales. La suma de las respuestas de los conos M y L es transmitida en un canal acromático que solo transmite información de intensidad y que es denominado canal A. Un segundo canal, pero cromático, denominado rojo-verde o simplemente R/G, transmite la diferencia de respuesta entre los conos M y L. Y un último canal también cromático, denominado azul-amarillo o simplemente B/Y, transmite la diferencia de la información de los conos S y el canal A (M + L) (Figura 3-14).

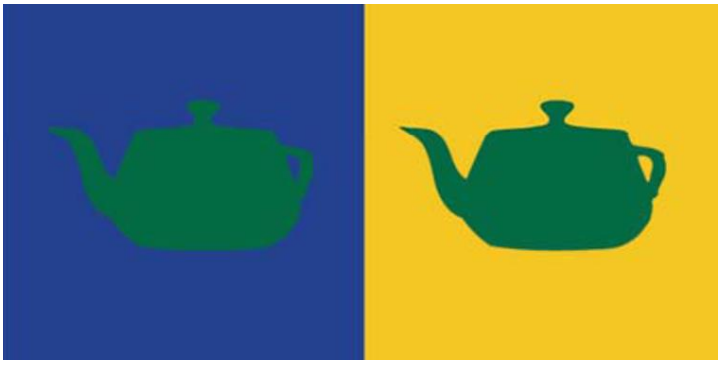


Figura 3-13 La tetera tiene el mismo color en ambos fondos. Sin embargo, los colores aparentan ser más vívidos si se encuentran circundantes a colores de menor luminosidad.

Esta es la razón por la que, por ejemplo, un color puede percibirse entre amarillo y rojo pero nunca entre rojo y verde. Esta teoría además sugiere que algunos colores aparentan ser más saturados que otros. Por ejemplo, el amarillo aparenta estar menos saturado que el rojo o el azul. Esto se debe a que se genera una fuerte señal en el canal acromático, y al menos una pequeña respuesta en algún canal cromático, siendo la proporción de las respuestas cromáticas y acromáticas la que determina si el color será visto saturado o no (Glassner, 1995).

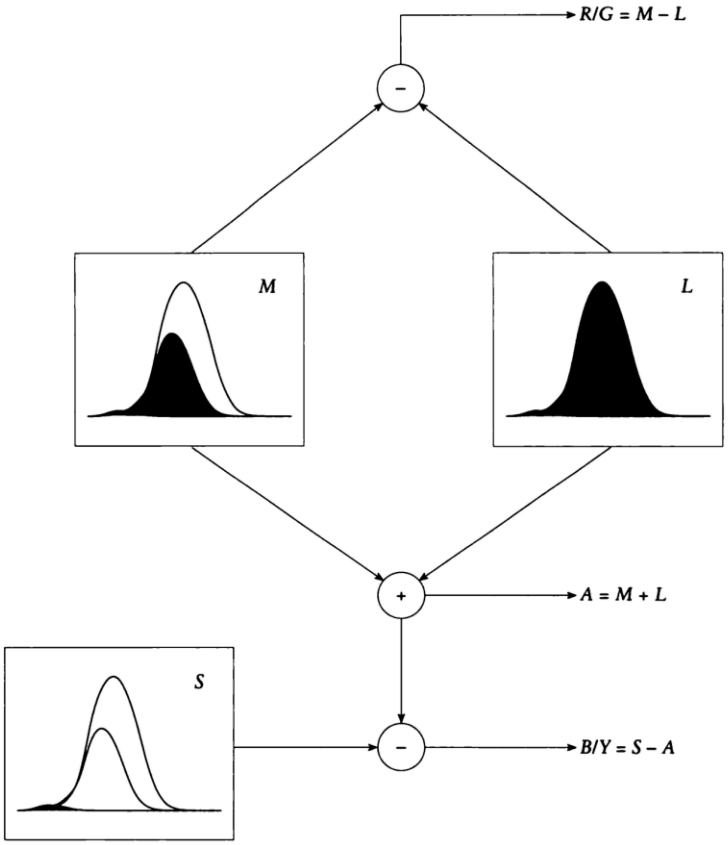


Figura 3-14 Modelo conceptual de la teoría de colores oponentes. La información de los conos es transmitida en tres canales que llevan la suma o diferencia de la información de color derivada de los fotorreceptores. (Glassner, 1995)

Capítulo 4 Modelos Computacionales de la Luz

La complejidad del mundo físico y las limitaciones de los sistemas de *display* hacen que sea imposible representar con precisión los efectos de luz que se producen cuando se observa una escena natural. No obstante, se necesita disponer de un modelo computacional eficiente que permita modelar con la mayor aproximación posible el comportamiento físico de la propagación de la luz y su interacción con las superficies.

Se presentarán los modelos generales que dictan reglas básicas de cómo representar la luz desde los enfoques cuántico, ondulatorio y corpuscular. Sin embargo, debido a la necesidad de disponer de un modelo computacionalmente eficiente, sólo se analizará en profundidad el modelo corpuscular más simple, conocido como óptica geométrica.

Además, se introducirá la radiometría con el objetivo de disponer de un sistema de medición formal que permita cuantificar consistentemente la luz en los algoritmos de iluminación.

Con estos elementos se discutirán los modelos específicos que permitan modelar por un lado la interacción entre la luz y una superficie, y por el otro la propagación de la luz en una escena. Se introducirán de esta manera los algoritmos más importantes de iluminación global y se discutirá en líneas generales por qué la iluminación local es atractiva aún si se desea generar gráficos fotorrealistas.

4.1 Modelos Generales

La física moderna todavía no ha podido comprender y/o demostrar completamente el comportamiento y las propiedades de la luz, por lo que ningún modelo puede ser catalogado como indiscutiblemente preciso, o incluso correcto.

La óptica cuántica es el modelo de luz que abarca la naturaleza dual de la luz. Es el modelo físico más completo e incluso puede explicar el comportamiento de la luz a nivel sub-microscópico. Pese a esto, es excesivamente complejo y su costo computacional demasiado alto para su utilización en computación gráfica, y menos aún en una aplicación de tiempo real.

El modelo ondulatorio es una simplificación del modelo cuántico que solo considera el comportamiento y las propiedades ondulatorias. Se apoya en las ecuaciones de Maxwell y por lo tanto simula los fenómenos de reflexión, refracción, difracción, interferencia y polarización utilizando estas ecuaciones. Sin embargo, dado la complejidad de su implementación computacional, en computación gráfica la naturaleza ondulatoria de la luz suele ser ignorada, o simulada utilizando algoritmos simples que modelan a grandes rasgos y de forma poco genérica los fenómenos enteramente ondulatorios. No obstante, si se realizan suposiciones apropiadas se pueden utilizar soluciones analíticas que permiten modelar parcialmente el comportamiento ondulatorio. Por ejemplo, se puede asumir que la superficie a evaluar es un conductor

perfecto y que las curvaturas de las facetas de la superficie son más grandes que la longitud de onda de la luz, bajo estas suposiciones se realizaron los modelos de Kajiya (Kajiya, Anisotropic Reflection Models, 1985) y posteriormente de He-Torrance (He, Torrance, Sillion, & Greenberg, 1991). Este último modelo, es útil cuando se desea predecir la apariencia de superficies casi lisas, debido a que produce variaciones en las reflexiones nítidas y difusas que no son predichas en otros modelos (Figura 4-1).

(a) (b)

Figura 4-1 Modelo He-Torrance: (a) Renderizado de una superficie lisa y (b) de una superficie casi lisa. (He, Torrance, Sillion, & Greenberg, 1991)

La óptica geométrica es el modelo más simple y el utilizado usualmente en computación gráfica. Se enfoca en comportamiento y propiedades corpusculares de la luz y se apoya en las ecuaciones de Snell para modelar la reflexión y refracción. Además, se asumen los siguientes comportamientos:

- La luz viaja en línea recta por lo cual se ignoran los fenómenos de difracción e interferencia.
- La longitud de onda de la luz es mucho más pequeña que la escala del objeto con el que interactúa, escenario en el que los fenómenos de difracción e interferencia son normalmente despreciables.
- La luz viaja instantáneamente a través del medio, lo que permite calcular el estado estacionario de la distribución de energía lumínica en la escena.
- La luz no se ve influenciada por factores externos como la gravedad o los campos magnéticos.

A partir de este momento solo se considerará la óptica geométrica debido a que es el enfoque con mayor viabilidad que puede ser utilizado en computación gráfica de tiempo real. No obstante, es posible generar de forma complementaria una representación de fenómenos ondulatorios como la difracción, pero las técnicas utilizadas a emplear no se basan en un sustento físico apropiado.

La óptica geométrica es un modelo general que indica mediante reglas cómo caracterizar las propiedades y el comportamiento de la luz. Una vez precisadas estas reglas, se deben seleccionar sub-modelos que permitan obtener una implementación computacional de la propagación de la luz y la interacción entre la luz y la materia. También es necesario disponer de un sistema estándar de medición de la luz que pueda ser utilizado en estos modelos.

4.2 Radiometría

Se necesita disponer de un sistema de medición formal que permita cuantificar consistentemente la luz en los algoritmos de iluminación. La radiometría es el área de estudio que se ocupa de estudiar cómo medir la radiación electromagnética. Por otro lado, la fotometría es el área que se ocupa de medir la radiación electromagnética en relación a la estimulación que produce en el sistema visual humano y se enfoca por lo tanto solo en el espectro visible. Típicamente, la radiometría se utiliza como sistema de medición para algoritmos de iluminación, mientras que la fotometría se utiliza para algoritmos fuertemente influenciados por el comportamiento del sistema visual humano, como por ejemplo el mapeo tonal (sección 5.5.1).

A continuación se introducirán las principales unidades radiométricas que son centrales en computación gráfica.

4.2.1 Flujo Radiante

El flujo radiante (o *flux*) expresa la cantidad total de energía que fluye a través de una región por unidad de tiempo. Se denota con el símbolo Φ y se expresa en watts (W) (joules por segundo). Por ejemplo, se puede decir que una fuente de luz emite 50 watts de flujo radiante, o que en una mesa inciden 20 watts de flujo radiante. Esta medida no especifica el tamaño de la fuente de luz o del receptor (la mesa), ni tampoco incluye una especificación de la distancia entre la fuente de luz y el receptor, ni la dirección de propagación.

4.2.2 Irradiancia y Emitancia Radiante

La irradiancia (E) es el flujo radiante incidente en una superficie por unidad de área de superficie (potencia incidente) y se expresa en watts/m^2. Por ejemplo, si 50 watts de flujo radiante inciden sobre una superficie con área de 1,25 m^2, la irradiancia en cada punto de la superficie es de 40 watts/m^2 (asumiendo que el flujo radiante se distribuye uniformemente sobre la superficie).

Normalmente el área considerada es un punto en la superficie. Sin embargo, un punto no puede ser definido físicamente en el mundo real; en cambio, se suele considerar un área infinitesimal alrededor del punto, denominada área diferencial (dA), en la que se garantiza las mismas propiedades físicas. En estos casos, la irradiancia podría ser rebautizada por ciertos autores como irradiancia diferencial (dE).

Matemáticamente la irradiancia se define como:

$$E = \frac{d\Phi}{dA} \qquad \text{(Ecuación 4-1)}$$

donde $d\Phi$ es el flujo radiante desde la fuente de luz que incide sobre el área diferencial dA.

Similarmente, la emitancia radiante (M), o radiosidad (B), es el flujo radiante emitido por unidad de superficie (potencia emitida) y también se expresa en watts/m^2:

$$M = \frac{d\Phi}{dA} \qquad \text{(Ecuación 4-2)}$$

4.2.3 Intensidad Radiante

La radiación que se origina de una determinada fuente en el espacio 3D ocupa un rango angular bidimensional; éste puede caracterizarse por un ángulo sólido medido en estereorradián (sr[3]). Así, la intensidad radiante es el flujo radiante por unidad de ángulo sólido (típicamente diferencial). Se la identifica con el símbolo I, su unidad es $watts/sr$ y su equivalente fotométrico es la intensidad lumínica:

$$I = \frac{d\Phi}{d\omega} \qquad \text{(Ecuación 4-3)}$$

donde $d\omega$ es el ángulo sólido diferencial que representa al cono infinitesimal en la dirección dada (Figura 4-2).

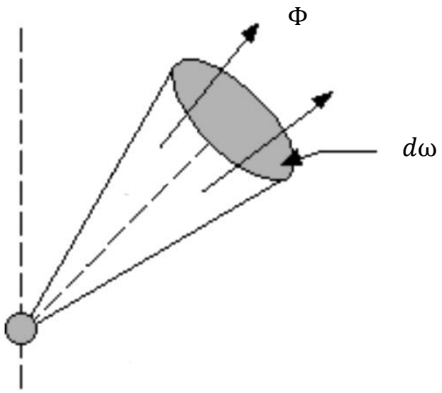


Figura 4-2 Intensidad radiante. Flujo radiante (Φ) por unidad de ángulo sólido ($d\omega$).

[3] sr es la unidad de ángulo sólido y se denominada estereorradián. Especifica el tamaño del área de una superficie proyectada sobre una esfera unitaria. El área de la superficie de un estereorradián es r^2 y la esfera completa mide 4π estereorradianes. Estereorradián, es un concepto análogo a la definición de un ángulo que es la longitud de un segmento sobre un círculo unitario.

Intuitivamente, la intensidad radiante puede ser vista como el flujo radiante que emite una superficie en una dirección d dada. El ángulo sólido es un área y puede tomar formas arbitrarias. Este ángulo sólido puede medirse calculando el área atravesada por la radiación a una distancia r de la fuente y dividiéndola por r^2. Si la radiación no se confina a un área determinada, el ángulo sólido para ésta puede determinarse integrando su densidad (en coordenadas polares).

Matemáticamente un ángulo sólido se define como:

$$d\omega = \frac{\mathrm{A}}{\mathrm{r}^2} \qquad \text{(Ecuación 4-4)}$$

donde A es la superficie correspondiente al ángulo sólido que se ubica en la superficie de la esfera de radio r centrada en el punto. Por lo tanto, la irradiancia se puede redefinir en función de la intensidad radiante (Ecuación 4-3) y la ecuación del ángulo sólido (Ecuación 4-4), con lo que se obtiene que:

$$E = \frac{d\Phi}{dA} = \int_{\Omega} \frac{d\Phi}{\mathrm{r}_{\omega}{}^2 d\omega} = \int_{\Omega} \frac{dI}{\mathrm{r}_{\omega}{}^2} \qquad \text{(Ecuación 4-5)}$$

donde Ω es el ángulo sólido total. La superficie diferencial dA está ubicada en las diferentes superficies de un conjunto de esferas centradas cada una en una fuente de luz infinitesimal que se encuentra a una distancia r de la superficie diferencial dA.

Esta ecuación muestra que la irradiancia disminuye con la distancia. Supongamos que observamos al sol desde la Tierra y desde Marte, el ángulo sólido que contiene al sol es mayor en la Tierra que en Marte, lo que intuitivamente indica que una mayor cantidad de rayos de luz inciden en un área de la superficie del planeta Tierra que en un área del mismo tamaño en la superficie del planeta Marte (Figura 4-3). Esta es la razón principal por la que el planeta Tierra es más caluroso que Marte.

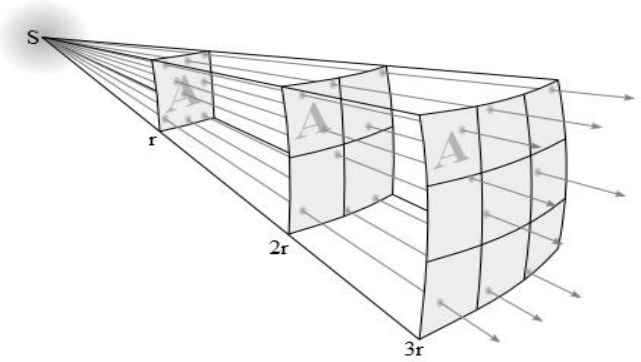


Figura 4-3 Las líneas representan el flujo radiante que emana de una fuente lumínica. La densidad de las líneas de flujo disminuye a medida que la luz se distancia de la fuente (Borb, 2008).

4.2.4 Radiancia

La radiancia es el flujo radiante por unidad de superficie proyectada por unidad de ángulo sólido. Se la identifica con el símbolo L, su unidad es $watts/(sr.m^2)$ y su equivalente fotométrico es la luminancia. Intuitivamente, la radiancia representa una cantidad infinitesimal de flujo radiante contenido en un rayo de luz que incide o sale de un punto en una superficie en una dirección dada; la radiancia es lo que miden los sensores.

La radiancia, una medida que varía con la posición x y con el vector dirección Θ, se expresa matemáticamente como $L(x, \Theta)$ (Figura 4-4):

$$L(x, \Theta) = \frac{d^2\Phi}{d\omega \, \mathrm{dA}^{\perp}} \qquad \text{(Ecuación 4-6)}$$

donde $\mathrm{dA}^{\perp}$ es el área dA proyectada sobre el plano perpendicular al rayo de luz que la intercepta en un ángulo θ definido con respecto a la normal de la superficie.

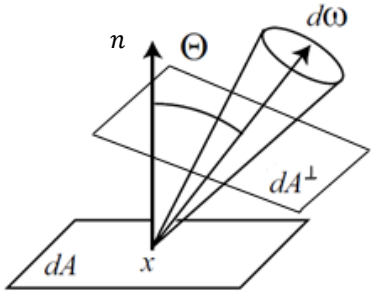


Figura 4-4 Radiancia. Flujo radiante (Φ) por unidad de área proyectada ($\mathrm{dA}^{\perp}$) por unidad de ángulo sólido ($d\omega$) (Dutré, Bala, & Behaert, 2006).

Para pequeñas áreas planas $\mathrm{dA}^{\perp}$ puede ser aproximada utilizando la fórmula (derivada de la Ley del Coseno de Lambert) (Figura 4-5):

$$\mathrm{dA}^{\perp} = \mathrm{dA} \, \overline{\cos} \, \theta \qquad \text{(Ecuación 4-7)}$$

donde $\overline{\cos} \, \theta$ es igual a 0 para direcciones de proyección que se ubican detrás de la superficie.

Por lo tanto, utilizando la (Ecuación 4-6) y la (Ecuación 4-7), la radiancia puede ser definida como:

$$L(x, \Theta) = \frac{d^2\Phi}{d\omega \, \mathrm{dA}^{\perp}} = \frac{d^2\Phi}{d\omega \, \mathrm{dA} \, \overline{\cos} \, \Theta} \qquad \text{(Ecuación 4-8)}$$

Para distinguir la radiancia incidente de la saliente, se utilizan respectivamente las notaciones $L(x \leftarrow \Theta)$ y $L(x \rightarrow \Theta)$.

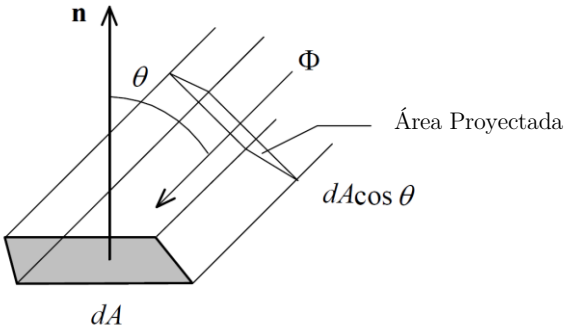


Figura 4-5 Rayo de luz interceptando una superficie diferencial dA. La superficie proyectada $dA^{\perp}$ se aproxima con la ecuación $dA \cos \theta$ (Dutré, Bala, & Behaert, 2006).

La radiancia posee dos propiedades que la convierten en una medida útil en algoritmos de iluminación. La radiancia es invariante en una línea recta, es decir, mientras se desplaza de una superficie a otra. Matemáticamente, esta propiedad se expresa como:

$$L(x \leftarrow y) = L(y \rightarrow x) \qquad \text{(Ecuación 4-9)}$$

y establece que la radiancia saliendo de un punto x hacia un punto y es la misma que la radiancia incidiendo en el punto y desde el punto x. Esta propiedad asume que la luz viaja a través de un único medio, es decir, no hay partículas que interactúen con la luz. Por lo tanto, se puede inferir que una vez conocida la radiancia incidente y saliente en todos los puntos de todas las superficies, la distribución de la radiancia de una escena también es conocida. Normalmente, los algoritmos de iluminación se limitan a calcular sólo la radiancia en los puntos de las superficies correspondientes, asumiendo la ausencia de partículas en el ambiente.

La segunda propiedad establece que la respuesta de los sensores (por ejemplo el ojo humano y las cámaras) es proporcional a la radiancia incidente en ellos. Estas dos propiedades en conjunto explican por qué el color y el brillo percibido no cambian con la distancia. Sin embargo, si bien la radiancia es invariante cuando la luz se desplaza por un medio, la irradiancia disminuye a un ritmo proporcional al cuadrado de la distancia entre las superficies (Ley de la Inversa del Cuadrado).

La irradiancia puede ser redefinida en función de la radiancia (Ecuación 4-8):

$$E(x) = \frac{d\Phi}{dA} = \int_{\Omega} L(x \leftarrow \Theta) \cos \Theta \, d\omega \qquad \text{(Ecuación 4-10)}$$

donde Ω es el ángulo sólido total. De esta forma se obtiene una fórmula de irradiancia en función de la radiancia, un valor que puede ser calculado en las fuentes de luz y que no varía en función de la distancia.

4.2.5 Radiometría espectral

Las medidas radiométricas dependen además de la longitud de onda de la luz. Cuando la longitud de onda es especificada explícitamente, las correspondientes medidas radiométricas se denominan espectrales. Por ejemplo, la radiancia sobre el espectro visible es calculada integrando la radiancia espectral sobre todo el rango de longitudes de onda del espectro visible:

$$L_\lambda(x \to \Theta) = \int_{espectro} L(x \to \Theta, \lambda)\, d\lambda \qquad \text{(Ecuación 4-11)}$$

En ecuaciones de iluminación global, los términos radiométricos normalmente no especifican explícitamente la longitud de onda. En la práctica, sólo se consideran las frecuencias que coinciden con los colores primarios aditivos del sistema de color utilizado.

4.3 Función Distribución de Reflectancia Bidireccional

Cuando se colorea una superficie, se calcula la radiancia saliente dada la magnitud y la dirección de las luces incidentes. Esta interacción entre la luz y una superficie puede ser modelada utilizando distintos modelos generales. Estos varían desde el caso más complejo en el que se modelan todos los fenómenos de interacción luz materia, hasta simplificaciones cuyo objetivo es reducir el costo computacional de su implementación a expensas de una representación física completa.

En el caso más general, se considera que la luz incide sobre una superficie en un punto p y en dirección incidente Ψ y puede abandonar la superficie en un punto q y con dirección de salida Θ. La función que define esta relación entre la radiancia incidente y la reflejada se denomina Función de Distribución de Reflectancia de Dispersión Superficial Bidireccional (BSSRDF) (Nicodemus, Richmond, Hsia, Ginsberg, & Limperis., 1977). Esta formulación, sin embargo, no considera fenómenos como la fluorescencia, en el cual la luz es absorbida a una frecuencia y es emitida a una diferente o como la fosforescencia en la cual la luz es absorbida a una frecuencia y es emitida a una diferente y a diferente tiempo.

El modelo BSSRDF es computacionalmente muy costoso, por lo que suele simplificarse asumiendo que la luz incidente sobre un punto se refleja en el mismo punto. Esta simplificación se la conoce como Función de Distribución de Reflectancia Bidireccional (BRDF) y es la más utilizada en computación gráfica (Figura 4-6). Sin embargo, ignora fenómenos como la dispersión superficial, fenómeno que puede ser simulado sobre la BRDF con modelos computacionalmente más eficientes pero físicamente menos precisos.

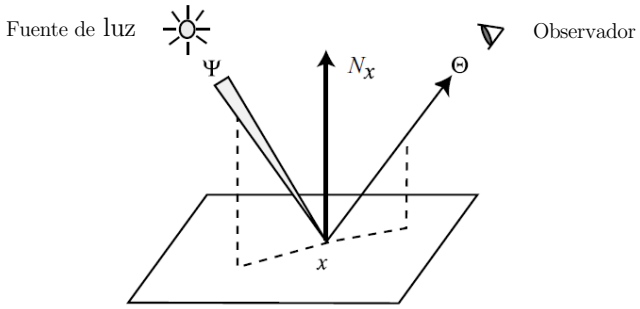


Figura 4-6 Función de Distribución de Reflectancia Bidireccional (Dutré, Bala, & Behaert, 2006).

La BRDF, denotada como $f_r(x, \Psi \rightarrow \Theta)$, se define sobre un punto x como la relación de la radiancia diferencial reflejada en una dirección de salida Θ, y la irradiancia diferencial incidente a través de un ángulo sólido diferencial $d\omega\Psi$:

$$f_r(x, \Psi \rightarrow \Theta) = \frac{dL(x \rightarrow \Theta)}{dE(x \leftarrow \Psi)} = \frac{dL(x \rightarrow \Theta)}{dL(x \leftarrow \Psi)\cos(N_x, \Psi)d\omega\Psi} \quad \text{(Ecuación 4-12)}$$

donde $\cos(N_x, \Psi)$ es el coseno del ángulo formado por el vector normal en el punto x, N_x, y la dirección incidente Ψ.

Estrictamente hablando, la BRDF se define sobre la esfera completa de direcciones (4Π estereorradianes) centrada en el punto. Esto es importante para superficies transparentes, debido que permite considerar correctamente la refracción. Sin embargo, la BRDF típicamente se limita solo al hemisferio con zenit coincidente con N_x . El término BSDF (función distribución de dispersión bidireccional) suele utilizarse para referirse a la BRDF que considera la esfera completa de direcciones. A partir de este momento sólo se considerará la BRDF limitada al hemisferio con zenit coincidente con la normal debido a que, por cuestiones de desempeño, es el modelo más utilizado en computación gráfica de tiempo real.

Una BRDF se caracteriza por el conjunto de propiedades que se detalla a continuación (Dutré, Bala, & Behaert, 2006):

(1) **Rango**: La BRDF puede tomar cualquier valor positivo y puede variar con la longitud de onda.

(2) **Dimensión**: La BRDF es una función tetra-dimensional definida en cada punto de la superficie, donde dos dimensiones corresponden a la dirección incidente y dos dimensionas a la saliente. Generalmente, la BRDF es anisotrópica, es decir que si la superficie se rota sobre la normal el valor de la función también cambiará. Sin embargo, existen muchos materiales isotrópicos para los que el resultado de la función no depende de la orientación de la superficie.

(3) **Reciprocidad**: el valor resultante de la BRDF se mantiene inalterado si las direcciones incidente y saliente se intercambian. Esta propiedad también se la conoce como Reciprocidad de Helmholtz. Intuitivamente, significa que la reversión de la dirección de la luz no cambia la cantidad de luz reflejada:

$$f_r(x, \Psi \rightarrow \Theta) = f_r(x, \Theta \rightarrow \Psi) \qquad \text{(Ecuación 4-13)}$$

(4) **Relación entre radiancia reflejada e incidente**: el valor de una BRDF para una dirección incidente específica no depende de la posible presencia de irradiancia en otros ángulos incidentes. Por lo tanto, la radiancia reflejada total producida por una distribución de irradiancia sobre el hemisferio definido alrededor de un punto de superficie opaco y no emisivo puede expresarse como:

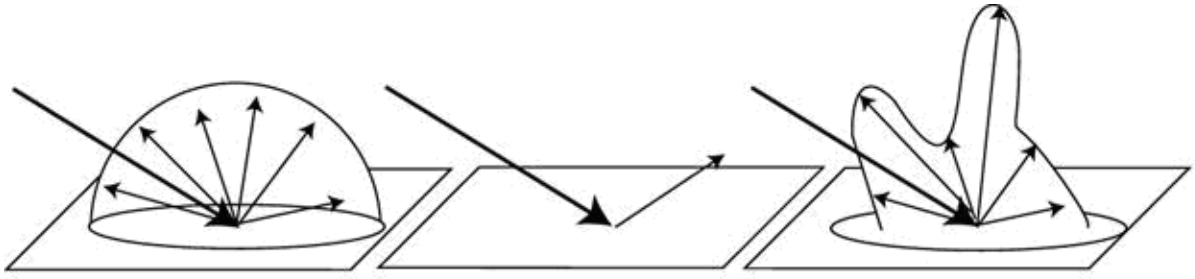
$$dL(x \rightarrow \Theta) = f_r(x, \Psi \rightarrow \Theta) dE(x \leftarrow \Psi)$$
$$L(x \rightarrow \Theta) = \int_{\Omega} f_r(x, \Psi \rightarrow \Theta) dE(x \leftarrow \Psi) \qquad \text{(Ecuación 4-14)}$$
$$L(x \rightarrow \Theta) = \int_{\Omega} f_r(x, \Psi \rightarrow \Theta)\, L(x \leftarrow \Psi) \cos(N_x, \Psi) d\omega\Psi$$

(5) **Conservación de energía**: la ley de conservación de energía sostiene que la cantidad total de energía reflejada en todas las direcciones debe ser menor o igual que la cantidad total de energía incidente sobre la superficie (excepto que la energía sea transformada en calor o en otra forma de energía).

Normalmente se utilizan modelos empíricos para caracterizar una BRDF, pero se debe asegurar que estos modelos respeten las propiedades de una BRDF. Algunos modelos no respetan la conservación de energía y la reciprocidad y por lo tanto no son modelos empíricos aceptables. Satisfacer la propiedad de Reciprocidad de Helmholtz es particularmente importante debido a que muchos algoritmos de iluminación calculan al mismo tiempo la distribución de energía considerando caminos desde las fuentes de luz y desde el punto de vista del observador.

4.3.1 Modelos generales de BRDF

Típicamente, las reflexiones que se producen sobre las superficies pueden clasificarse en 3 categorías: difusas, especulares y brillantes (*glossy*). La mayoría de las superficies reales exhiben una reflexión que es una combinación de los tres tipos (Figura 4-7).



Superficie Difusa Superficie Especular Superficie Brillante

Figura 4-7 Modelos generales de BRDF (Dutré, Bala, & Behaert, 2006).

Las superficies difusas reflejan la luz uniformemente sobre todo el hemisferio de reflexión. Esto es, dada una distribución de irradiancia, la radiancia reflejada es independiente de la dirección de salida. Tales materiales se llaman reflectores difusos y el valor de su BRDF es constante para todos los valores de Ψ y Θ:

$$f_r(x, \Psi \rightarrow \Theta) = \frac{\rho_d}{\pi} \qquad \text{(Ecuación 4-15)}$$

donde ρ_d representa la fracción de energía incidente que es reflejada por la superficie. Para modelos basados en física, ρ_d varía desde 0 hasta la cantidad de energía incidente.

Las superficies especulares sólo reflejan la luz en una dirección específica:

$$f_r(x, \Psi \rightarrow \Theta) = \begin{cases} \rho_s, & \Theta = \mathrm{R} \\ 0, & \Theta \neq \mathrm{R} \end{cases} \qquad \text{(Ecuación 4-16)}$$

donde R es el vector de reflexión y ρ_s representa la fracción de energía incidente que es reflejada por la superficie.

La ley de Reflexión especifica que las direcciones de incidencia y de salida forman ángulos iguales con respecto a la normal de la superficie y se encuentran en el mismo plano que la normal. Por lo tanto, la dirección de reflexión R se puede calcular simplemente como:

$$\mathrm{R} = 2(N.\Psi)N - \Psi \qquad \text{(Ecuación 4-17)}$$

Empíricamente los espejos pueden ser representados con este modelo; sin embargo, en la realidad, los espejos no son reflectores perfectos.

Si se desea modelar la refracción, se puede utilizar un planteamiento similar, pero calculando la dirección de transmisión T. Si la luz incidente se transmite desde un medio con índice de refracción n_1 hacia un medio con índice de refracción n_2, la ley de Snell define que:

$$n_1 \sin\theta_1 = n_2 \sin\theta_2 \qquad \text{(Ecuación 4-18)}$$

donde θ_1 y θ_2 son respectivamente los ángulos entre el vector incidente y el de transmisión y la normal de la superficie. El vector de transmisión T por lo tanto puede ser calculado como:

$$\begin{aligned} T &= -\frac{n_1}{n_2}\Psi + N\left(\frac{n_1}{n_2}\cos\theta_1 - \sqrt{1-\left(\frac{n_1}{n_2}\right)^2\left(1-\cos{\theta_1}^2\right)}\right) \\ &= -\frac{n_1}{n_2}\Psi + N\left(\frac{n_1}{n_2}(N.\Psi) - \sqrt{1-\left(\frac{n_1}{n_2}\right)^2(1-(N.\Psi)^2)}\right) \end{aligned} \qquad \text{(Ecuación 4-19)}$$

Un planto similar puede ser usado si ocurre reflexión interna total.

Las ecuaciones anteriores especifican los ángulos de reflexión y refracción para la luz que arriba a una superficie perfectamente suave. Fresnel formuló un conjunto de ecuaciones llamadas ecuaciones de Fresnel que especifican la cantidad de luz que es reflejada y refractada desde una superficie perfectamente suave. La cantidad de luz que es reflejada depende de la longitud de onda y de la dirección de la luz incidente y de la geometría de la superficie. Estas ecuaciones toman en cuenta además la polarización de la luz. Las dos componentes de la luz polarizada reflejada en la superficie, la componente perpendicular (r) y la componente paralela (p), se calculan como:

$$\begin{aligned} r_p &= \frac{n_2\cos\theta_1 - n_1\cos\theta_2}{n_2\cos\theta_1 + n_1\cos\theta_2} \\ r_s &= \frac{n_1\cos\theta_1 - n_2\cos\theta_2}{n_1\cos\theta_1 + n_2\cos\theta_2} \end{aligned} \qquad \text{(Ecuación 4-20)}$$

donde n_1 y n_2 son los índices de refracción de las dos superficies de la interfaz.

Las ecuaciones de Fresnel asumen que la luz incidente es reflejada o refractada. Debido a que se asume que la superficie no absorbe luz, los coeficientes de reflexión y refracción suman 1 y por lo tanto:

$$\begin{aligned} t_p &= 1 - r_p \\ t_s &= 1 - r_s \end{aligned} \qquad \text{(Ecuación 4-21)}$$

Fresnel además definió que, para la luz incidente no polarizada, la fracción de energía reflejada es:

$$\mathrm{F} = \frac{|r_p|^2 + |r_s|^2}{2} \qquad \text{(Ecuación 4-22)}$$

Por lo tanto, la BRDF para superficies perfectamente suaves y para luz incidente no polarizada puede ser definida utilizando las ecuaciones de Fresnel de la siguiente forma:

$$f_r(x, \Psi \rightarrow \Theta) = \begin{cases} F\rho, & \Theta = \mathrm{R} \\ (1-F)\rho, & \Theta = T \\ 0, & \text{en caso contrario} \end{cases} \qquad \text{(Ecuación 4-23)}$$

donde ρ es la radiancia incidente en dirección Ψ.

Estas ecuaciones están formuladas para metales y no metales. Para metales, el índice de refracción del metal se expresa como una variable compleja $n + \mathrm{ik}$, mientras que para no metales, el índice de refracción es un número real y $k = 0$.

La mayoría de las superficies no son perfectamente especulares ni perfectamente difusas pero exhiben una combinación de ambos comportamientos; estas superficies son las llamadas superficies brillantes y sus BRDFs normalmente son difíciles de modelar con formulaciones analíticas.

4.3.2 Modelos de Iluminación

Los materiales reales tienen normalmente BRDFs complejas. En computación gráfica han sido propuestos varios modelos para capturar la complejidad de estas BRDF. A continuación se introducirán algunos de los modelos de iluminación más importantes y populares.

El modelo más simple es el modelo Lambert y es utilizado para modelar materiales difusos ideales (Figura 4-8). En este modelo, la BRDF es una constante (como se describió en la sección anterior):

$$f_r(x, \Psi \rightarrow \Theta) = k_d = \frac{\rho_d}{\pi} \qquad \text{(Ecuación 4-24)}$$

donde ρ_d es la energía total reflejada y k_d representa el color difuso.

Figura 4-8 Modelo de iluminación de Lambert.

Este modelo típicamente se combina con modelos especulares para representar superficies brillantes.

Bui Tuong Phong presentó el modelo Phong (Phong, 1975) para superficies brillosas (Figura 4-9). Aunque en la actualidad no es muy utilizado, muchos modelos, incluyendo el popular Blinn-Phong, pueden ser vistos como variaciones o mejoras de este modelo básico.

La BRDF del modelo de Phong se define como:

$$f_r(x, \Psi \rightarrow \Theta) = k_s \frac{(R.\Theta)^n}{N.\Psi} + k_d \qquad \text{(Ecuación 4-25)}$$

donde R es el vector reflexión que puede ser calculando utilizando (Ecuación 4-17).

Figura 4-9 Modelo de Sombreado Phong.

El modelo Blinn-Phong (Blinn, 1977) es una variación del modelo Phong que utiliza el vector medio (*H* o *half vector*) entre el vector Ψ y el vector Θ en remplazo del vector de reflexión (R) (Figura 4-10).

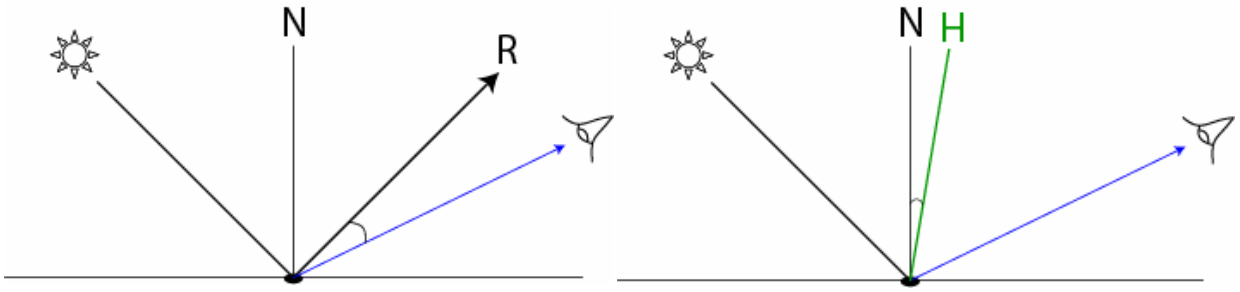


Figura 4-10 Comparación gráfica entre el vector R (izquierda) y el vector H (derecha).

La BRDF del modelo Blinn se define como:

$$f_r(x, \Psi \rightarrow \Theta) = k_s \frac{(H.N)^n}{N.\Psi} + k_d \qquad \text{(Ecuación 4-26)}$$

En (Ngan, Durand, & Matusik, 2004) analizaron la forma del lóbulo especular para evaluar la calidad de los modelos. Según estos resultados, el error logarítmico que produce el modelo Blinn-Phong es igual o menor al error producido por el modelo Phong en todos los casos de pruebas (Figura 4-11).

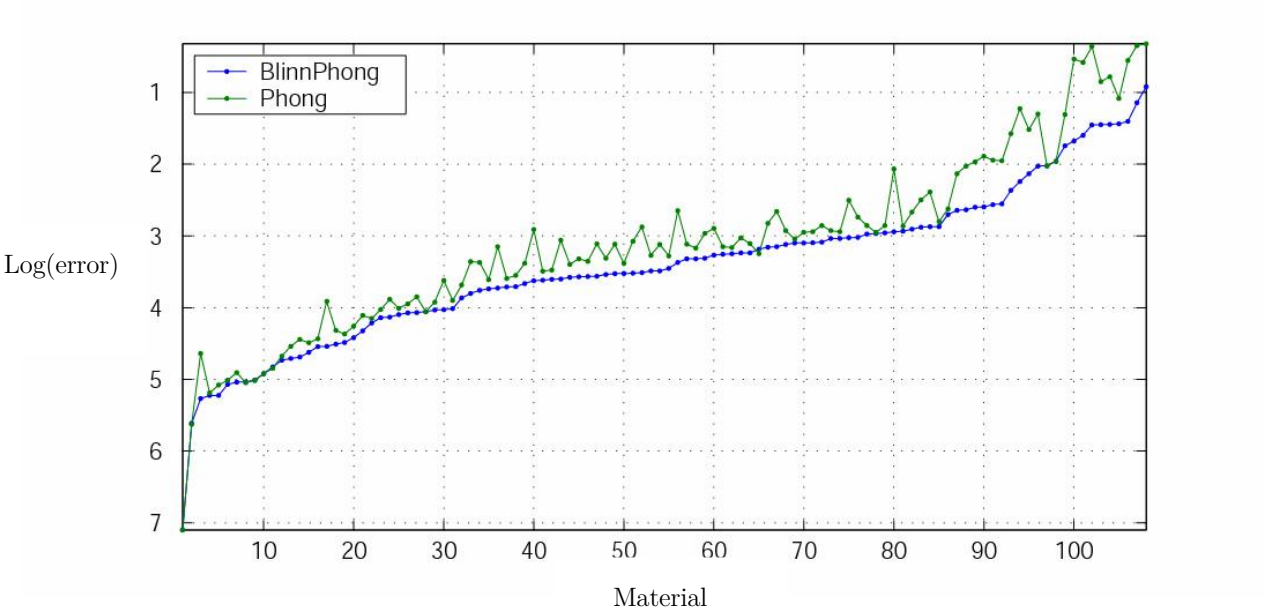


Figura 4-11 Error logarítmico de los lóbulos especulares de los modelos Blinn-Phong y Phong en un conjunto de materiales de prueba (Ngan, Durand, & Matusik, 2004).

A pesar de que los modelos Phong y Blinn-Phong son atractivos por la calidad de sus resultados, tienen un conjunto de limitaciones importantes debido a que no respetan la ley de conservación de energía, no satisfacen la ley de reciprocidad de Helmholtz y no capturan el comportamiento de la mayoría de los materiales reales. El modelo Blinn-Phong Modificado soluciona algunos de estos problemas realizando un pequeño ajuste en la ecuación:

$$f_r(x, \Psi \rightarrow \Theta) = k_s(H.N)^n + k_d \qquad \text{(Ecuación 4-27)}$$

Desafortunadamente, este modelo no es capaz de capturar BRDF realistas. Los modelos basados en la física, como los de Cook-Torrance (Cook & Torrance, 1982) y He (He, Torrance, Sillion, & Greenberg, 1991), entre otros, intentan modelar materiales reales, pero su complejidad y costo computacional son significativamente mayores.

El modelo de Cook-Torrance incluye un modelo de microfacetas que asume que la superficie del material está compuesta por una colección aleatoria de facetas planas pequeñas y lisas. Dada una distribución de microfacetas, este modelo captura las sombras entre esas microfacetas. Además, estas ecuaciones también incluyen las ecuaciones de Fresnel de reflexión y refracción (Figura 4-12).

La BRDF del modelo Cook-Torrance se define como:

$$f_r(x, \Psi \rightarrow \Theta) = \frac{F(\beta)}{\pi} \frac{D(\theta_h)G}{(N.\Psi)(N.\Theta)} + k_d \qquad \text{(Ecuación 4-28)}$$

donde F es la reflectancia de Fresnel, D es la distribución de microfacetas, y G es el término geométrico.

En Cook-Torrance se asume que la luz no es polarizada por lo tanto F se define utilizando la (Ecuación 4-22). El término de reflectancia de Fresnel se calcula con respecto al ángulo β, que es el ángulo entre la dirección incidente y el vector medio entre Θ y Ψ:

$$\cos\beta = \Psi.\mathrm{H} = \Theta.\mathrm{H} \qquad \text{(Ecuación 4-29)}$$

(a) (b) (c)

Figura 4-12 Comparación entre los modelos Blinn-Phong (a) y Cook-Torrance (b y c). Cook-Torrance permite representar una mayor variedad de materiales.

La función de distribución D especifica la distribución de microfacetas del material. Se pueden usar distintas fórmulas para especificar esta distribución; una de las más comunes es la distribución de Beckmann:

$$\mathrm{D}(\theta_h) = \frac{1}{m^2\cos^4\theta_h} e^{-(\frac{\tan\theta_h}{m})^2} \qquad \text{(Ecuación 4-30)}$$

donde θ_h es el ángulo entre la normal y el vector medio (entre Θ y Ψ) y $\cos\theta_h = N.H$. Asimismo, m es la pendiente de la media cuadrática de las microfacetas y captura la rugosidad de la superficie.

El término geométrico G captura las sombras que se generan entre las distintas microfacetas de la superficie:

$$\mathrm{G} = \min\left\{1, \frac{2(N.H)(N.\Theta)}{\Theta.H}, \frac{2(N.H)(N.\Psi)}{\Theta.H}\right\} \qquad \text{(Ecuación 4-31)}$$

Cook-Torrance permite renderizar con alto realismo una mayor variedad de materiales, en especial superficies metálicas y rugosas.

Modelos como Ward (Ward, 1992) y Lafortune (Lafortune, Foo, Torrance, & Greenberg, 1997) se basan en datos empíricos. Estos modelos tienen como objetivo permitir un control sencillo de los parámetros, y por lo tanto el aspecto, de la BRDF.

Para superficies isotrópicas, el modelo de Ward tiene la siguiente BRDF:

$$f_r(x, \Psi \rightarrow \Theta) = \frac{\rho_d}{\pi} + \rho_s \frac{e^{-\frac{\tan^2 \theta_h}{\alpha^2}}}{4\pi\alpha^2\sqrt{(N.\Psi)(N.\Theta)}} = k_d + k_s \qquad \text{(Ecuación 4-32)}$$

donde θ_h es el ángulo entre la normal y el vector medio entre Θ y Ψ.

El modelo de Ward incluye tres parámetros que describen el aspecto de la BRDF: ρ_d, la reflectancia difusa, ρ_s, la reflectancia especular y α, una medida de la rugosidad de la superficie. Este modelo respeta la propiedad de conservación de energía y es relativamente intuitivo de configurar debido a la pequeña cantidad de parámetros. Configurándolo apropiadamente se puede representar una amplia gama de materiales, comparable con las que permite, por ejemplo, el modelo Cook-Torrance.

4.4 Ecuación de Renderizado

La formulación matemática de la distribución de la luz en una escena puede caracterizarse por la ecuación de renderizado. Esta fue simultáneamente propuesta por Kajiya (Kajiya, The Rendering Equation, 1986) y por Immel (Immel, Cohen, & Greenberg, 1986) y asume la ausencia de partículas en el ambiente y la propagación instantánea de la luz.

Se asume que $L_e(x \rightarrow \Theta)$ representa la radiancia emitida por la superficie en el punto x y en la dirección saliente Θ, y $L_r(x \rightarrow \Theta)$ representa la radiancia reflejada por la superficie en el punto x y en dirección Θ. Por la ley de conservación de energía, la radiancia saliente total en un punto y en una dirección saliente particular es la suma de la radiancia emitida y la radiancia reflejada en el punto de la superficie en esa dirección. La radiancia saliente $L(x \rightarrow \Theta)$ se expresa en términos de $L_e(x \rightarrow \Theta)$ y $L_r(x \rightarrow \Theta)$ de la siguiente forma:

$$L(x \rightarrow \Theta) = L_e(x \rightarrow \Theta) + L_r(x \rightarrow \Theta) \qquad \text{(Ecuación 4-33)}$$

Por la definición de BRDF (Ecuación 4-12) y por (Ecuación 4-14) se tiene que:

$$L_r(x \rightarrow \Theta) = \int_\Omega f_r(x, \Psi \rightarrow \Theta)\, L(x \leftarrow \Psi)\cos(N_x, \Psi) d\omega\Psi \qquad \text{(Ecuación 4-34)}$$

Reemplazando la (Ecuación 4-34) en la (Ecuación 4-33) se obtiene la ecuación de renderizado:

$$L(x \rightarrow \Theta) = L_e(x \rightarrow \Theta) + \int_\Omega f_r(x, \Psi \rightarrow \Theta)\, L(x \leftarrow \Psi)\cos(N_x, \Psi) d\omega\Psi \qquad \text{(Ecuación 4-35)}$$

La ecuación de renderizado es una ecuación integral denominada ecuación de Fredholm de segunda clase debido a su forma: un valor desconocido, la radiancia, aparece en ambos lados de la ecuación y en la

derecha se encuentra bajo una integral. Resolver la ecuación de renderizado para cada punto de superficie en toda la escena no es sencillo, pero permite obtener una solución a la iluminación de la misma, específicamente una solución de iluminación global que no solo tiene en cuenta las fuentes de luz predominantes, sino la aportación de todas las superficies de la escena.

Una de las características de la ecuación de renderizado es que no especifica ninguna característica de reflectancia de las superficies involucradas. Sin embargo, como está basada en una BRDF, tiene las mismas restricciones físicas que éstas. Además, la ecuación de renderizado puede dividirse en una suma de componentes (Jensen, 1996). Por ejemplo, puede dividirse en la contribución directa desde fuentes de luz predominantes (iluminación directa) y en la contribución de la luz que se ha reflejado más de una vez (iluminación indirecta), haciendo posible resolver estos términos de forma independiente.

La ecuación de renderizado puede simplificarse como una solución de iluminación local, en la cual solo se tiene en cuenta un número finito de fuentes de luz:

$$L(x \to \Theta) = L_e(x \to \Theta) + \sum_{j=1}^{n} f_r(x, \Psi \to \Theta)\, g(x \leftarrow \Psi) I_j(x \leftarrow \Psi) \cos(N_x, \Psi) \qquad \text{(Ecuación 4-36)}$$

donde $I_j(x \leftarrow \Psi)$ es la intensidad de la fuente de luz j. El término geométrico $g(x \leftarrow \Psi)$ escala la intensidad lumínica basándose en la geometría circundante, la distancia de la fuente de luz y el tipo de la luz. Una ventaja importante de la iluminación local es que, al no considerar la interacción entre todas las superficies, solo se necesita calcular la iluminación para los puntos de superficie que son visibles desde la cámara que captura la escena, simplificando aún más los cálculos necesarios. Además, por ejemplo, las sombras se podrían incorporar directamente sobre el término geométrico requiriendo una evaluación de oclusión entre la fuente de luz y el punto de superficie x. Sin embargo, este modelo de iluminación local tiene importantes desventajas dado que, por ejemplo, no captura ningún tipo de interreflexión entre superficies.

Idealmente, para alcanzar un mayor fotorealismo se debe implementar un pipeline de renderizado de iluminación global que intente modelar la ecuación de renderizado lo más aproximadamente posible.

4.5 Algoritmos de Iluminación Global

Existe un número considerable de algoritmos que intentan resolver la ecuación de renderizado incorporando características de iluminación global. Una forma de categorizar el comportamiento de los algoritmos de iluminación global es detallando qué interacción entre superficies implementan o simulan. Específicamente, debe considerarse si la luz incidente se dispersa o se refleja de forma difusa o especular y si ésta se origina desde una fuente de luz especular o difusa.

Por lo tanto, es posible categorizar el comportamiento en cuatro clases generales (Figura 4-13) (Watt & Policarpo, 1999):

- Difuso a difuso (DD).
- Especular a difuso (SD).
- Difuso a especular (DS).
- Especular a especular (SS).

En esta sección se discutirán un conjunto de soluciones básicas que resumen la complejidad del problema y el camino general hacia las principales soluciones.

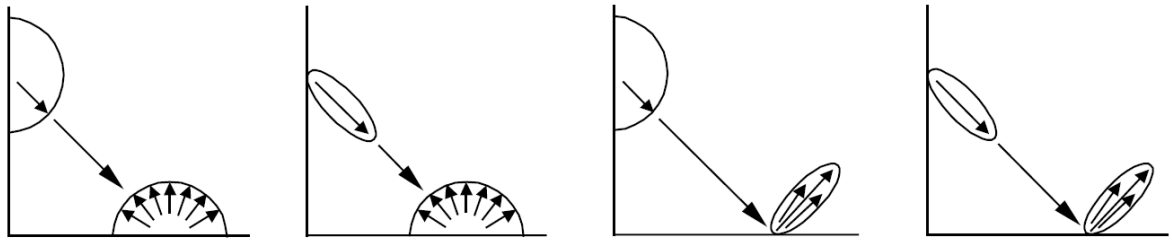


Figura 4-13 Categorización del comportamiento de las interacciones entre superficies (a) difuso a difuso (b) especular a difuso (c) difuso a especular (d) especular a especular (Wallace, Cohen, & Greenberg, 1987).

4.5.1 Trazado de Rayos de Whitted

El trazado de rayos de Whitted (Whitted, 1979) traza rayos de luz en dirección inversa de propagación, o sea, desde el punto de vista hacia la escena. El objetivo de comenzar desde el punto de vista es reducir significativamente los cálculos debido a que sólo es necesario calcular los rayos de luz que contribuyen en la captura bidimensional de la escena.

En cada interacción con la superficie se calcula un rayo reflejado, o un rayo transmitido (refracción), o ambos. El proceso termina cuando la energía del rayo cae por debajo de cierto umbral, cuando el rayo abandona los límites de la escena o cuando intercepta una superficie perfectamente difusa. De esta manera, el algoritmo sólo considera interacciones especular a especular y las sombras producidas son sombras duras (sin penumbra). Podrían considerarse interacciones difusas, pero para esto se debe calcular, por cada punto de interacción, la radiancia en todas las direcciones del hemisferio centrado en el punto de la superficie. Además, en este algoritmo se calcula un modelo local de iluminación en cada punto de interacción, lo que permite considerar reflexiones difusas (y sombras), pero esto no es lo mismo que interacciones difuso a difuso (Figura 4-14).

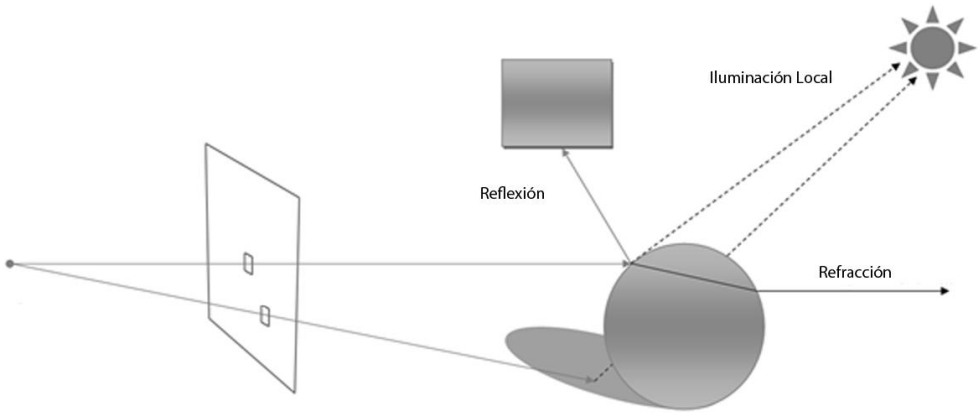


Figura 4-14 Esquema del algoritmo de trazado de rayos de Whitted (Shirley & Morley, 2008).

4.5.2 Trazado de Rayos Distribuido

El trazado de rayos distribuido (Cook, Porter, & Carpenter, 1984) está motivado (presumiblemente) por la limitación del trazado de rayos Whitted que solo considera interacciones especulares perfectas. En el mundo real, las superficies rara vez son perfectas por lo que las reflexiones suelen tener un menor o mayor grado de difuminado. Asimismo, las refracciones tampoco son perfectas debido a imperfecciones en la superficie (o en el volumen).

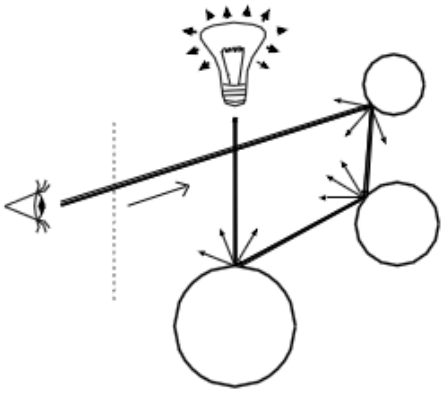


Figura 4-15 Esquema del algoritmo de trazado de rayos distribuido. Por simplicidad, en este diagrama se recorre un solo camino y se destacan los rayos que éste genera en su recorrido hasta terminar en una fuente de luz (Szirmay-Kalos, 2000).

El trazado de rayos distribuido, al igual que el trazador de rayos de Whitted, solo considera interacciones especular a especular pero simula interacciones especulares imperfectas construyendo en cada punto de interacción un lóbulo de reflexión; la forma de este lóbulo depende de las propiedades del material de la superficie en ese punto. Por lo tanto, en vez de trazar un solo rayo de reflexión y/o

transmisión, se traza un conjunto de rayos que muestrean el lóbulo de reflexión (Figura 4-15); para resolver este muestreo se utiliza la técnica Monte Carlo. De esta manera se pueden simular reflexiones difuminadas, dispersión (*scattering*) en superficies transparentes y sombras suaves (con penumbra). En este trabajo también se propone simular el funcionamiento del lente con el objetivo de modelar los fenómenos de profundidad de campo (*depth of field*) y desenfoque de movimiento (*motion blur)* (Figura 4-16).

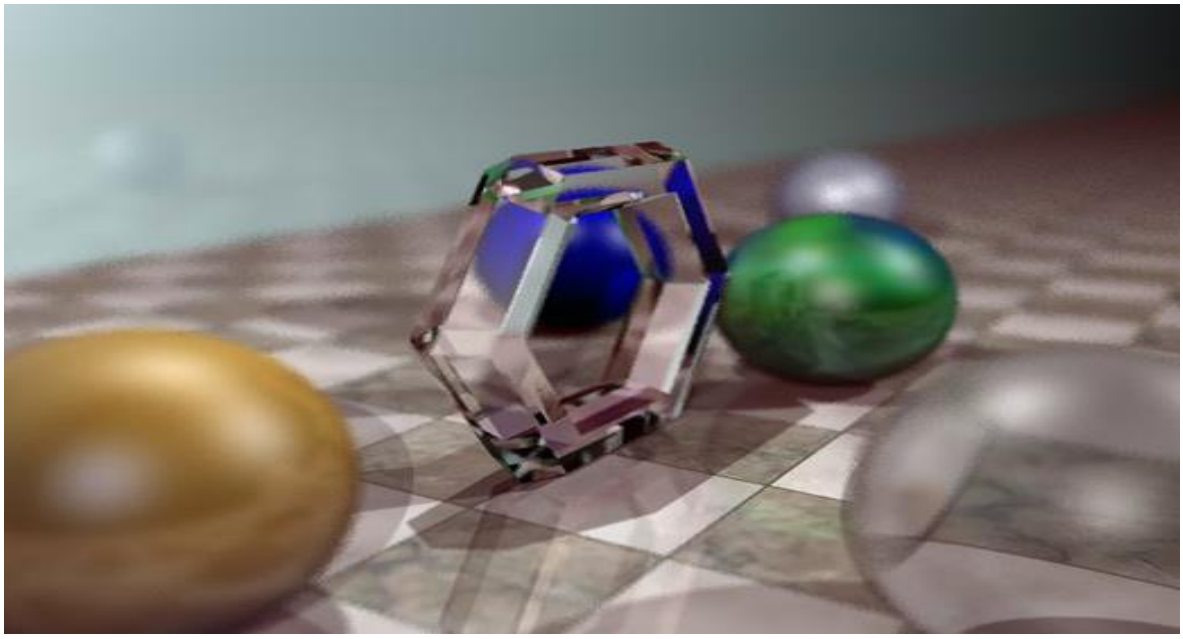


Figura 4-16 Imagen renderizada utilizando el algoritmo trazado de rayos distribuido. Se puede apreciar reflexiones y refracciones imperfectas y profundidad de campo (Flannery).

4.5.3 Trazado de Rayos de Dos Pasadas

La primera pasada del trazado de rayos de dos pasadas (o trazado de rayos bidireccional) (Lafortune & Willems, 1993) consiste en lanzar rayos desde las fuentes de luz y seguirlos con un modelo de interacciones especular hasta que interactúen con una superficie difusa. En este punto, la energía del rayo de luz se deposita o almacena en la superficie difusa, la cual debe estar subdividida de alguna manera en elementos. El tamaño del elemento es crítico, si es muy chico el elemento podría no recibir ningún rayo de luz, y si es muy grande se podría perder precisión.

La segunda pasada es un trazador de rayos del estilo de trazado de rayos de Whitted. Cuando éste alcanza una superficie difusa utiliza la energía almacenada en la primera pasada como aproximación a la iluminación difusa que recibe la superficie desde todas las direcciones posibles. Esto permite modelar las interacciones especular a difuso y por lo tanto fenómenos como cáustica (*caustic*) (Figura 4-18).

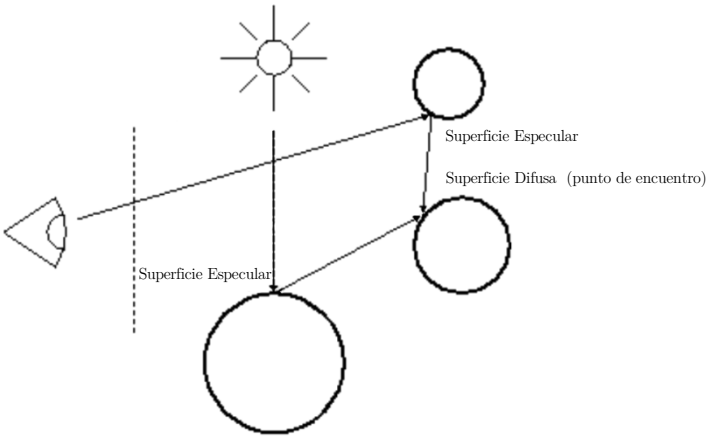


Figura 4-17 Esquema del algoritmo de trazado de rayos de dos pasadas. El punto de encuentro de las dos pasadas de este algoritmo son siempre las superficies difusas. (Szirmay-Kalos, 2000).

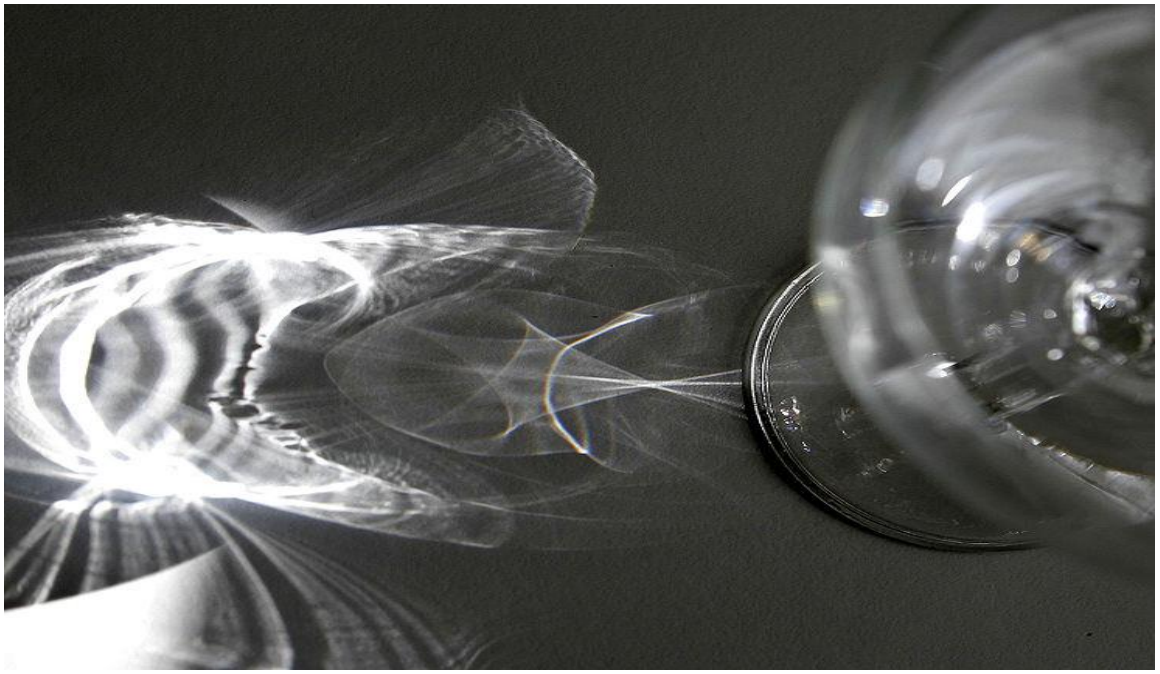


Figura 4-18 Fenómeno de cáustica. Esta fotografía capta cómo la luz atraviesa un vaso transparente y forma patrones de luz concentrados en la superficie opaca (Flagstaffotos, 2005).

4.5.4 Radiosidad

El algoritmo de radiosidad (Goral, Torrance, Greenberg, & Battaile, 1984) implementa interacciones difuso a difuso (Figura 4-19). Además, la radiosidad difiere con respecto a los algoritmos de iluminación global anteriores en que es una solución independiente de la vista.

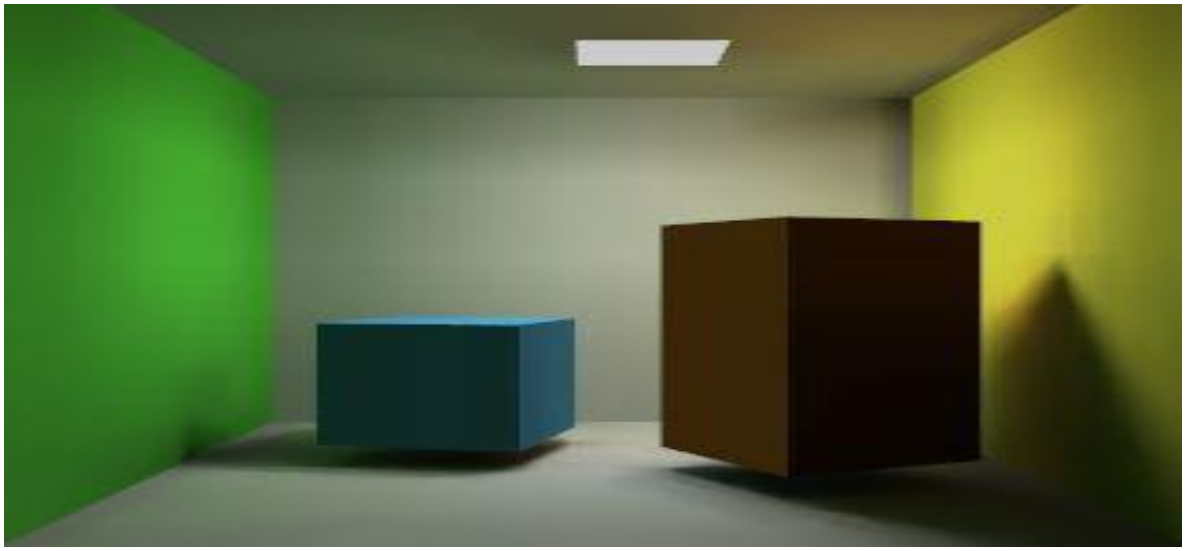


Figura 4-19 Escena renderizada utilizando el algoritmo de radiosidad.

Inicialmente a cada parche (o polígono) de la escena se le aplica un término de radiosidad constante inicial. Luego se comienza a lanzar rayos de luz desde las distintas fuentes de luz de la escena hacia todos los parches visibles desde cada uno de éstas. En cada iteracción, una cantidad de energía de luz es depositada en el parche en donde ocurrió la interacción. El parche con la mayor cantidad de energía no lanzada es seleccionado y se comienza a lanzar desde ese punto un nuevo conjunto de rayos de luz. El proceso continúa iterativamente hasta que un alto porcentaje de la energía inicial haya sido distribuida en la escena. Luego una segunda pasada calcula la iluminación para los puntos de superficie directamente visibles a la cámara utilizando la información almacenada en la primera pasada (Figura 4-20).

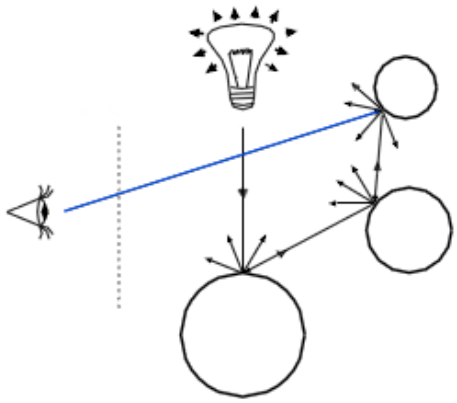


Figura 4-20 Esquema del algoritmo de radiosidad. La segunda pasada está marcada en azul (Szirmay-Kalos, 2000).

El principal problema de la radiosidad es que aunque la mayor parte de las escenas están compuestas mayoritariamente de superficies difusas, es frecuente la presencia de superficies especulares y éstas no pueden ser representadas por el algoritmo de radiosidad. Además, la escena debe dividirse en parches o polígonos, una tarea no trivial. Aún si la escena se encuentra modelada por mallas poligonales, el tamaño

de los distintos polígonos puede diferir significativamente, por lo que en distintos polígonos se podría tener excesiva o poca resolución.

4.5.5 Trazado de Rayos de Dos Pasadas y Radiosidad

La primera pasada del trazado de rayos de dos pasadas puede ser extendida para contemplar además interacciones difusas. De esta manera, en la primera pasada se lanzan rayos de luz desde las fuentes de luz hacia las distintas superficies utilizando interacciones especular a especular; cuando se intercepta una superficie difusa, el proceso continúa para este punto de superficie, pero utilizando el algoritmo de radiosidad. Finalmente, se ejecuta la pasada dependiente de la vista y se produce el resultado final.

Idealmente, se debe incluir la posibilidad de realizar interacciones especulares una vez ejecutado el algoritmo de radiosidad. Sin embargo, la extensión del algoritmo de radiosidad para contemplar interacciones difusas intermedias no es trivial y puede impactar significativamente en la performance del mismo. Esto se debe principalmente a que el algoritmo original no tiene necesidad de almacenar la dirección de incidencia de la luz (necesaria para retrasmitirlo) por cada rayo que intercepta la superficie; solo necesita almacenar el aporte de energía total de todos los rayos incidentes.

Combinar el trazado de rayos de dos pasadas y radiosidad (con interacciones especulares) es un método que aproxima muy bien la ecuación de renderizado; sin embargo, el costo computacional de la técnica es muy alto.

4.5.6 Trazado de Caminos

El trazado de caminos, o *Photon Mapping*, (Jensen, 1996) es similar en varios aspectos al trazado de rayos de Whitted dado que es dependiente de la vista y se recorre el camino inverso de propagación. Sin embargo, en el trazado de caminos se lanza un número arbitrario de rayos de luz por cada pixel de la pantalla y en vez de generar un conjunto de rayos por cada interacción con una superficie, se recorre sólo un camino que se selecciona aleatoriamente basándose en una distribución probabilística y en las propiedades del material de la superficie; además, se calcula un modelo local de iluminación por cada interacción (Figura 4-21).

Un trazado de rayos convencional típicamente realiza gran parte del trabajo en las últimas interacciones del rayo de luz original, etapa en la cual el aporte lumínico de cada interacción es cada vez menor; en cambio, un trazado de caminos equilibra el trabajo a lo largo del proceso. Un trazado de caminos además permite modelar cualquier tipo de interacción lo que lo convierte en una de las soluciones más completas y evita la necesidad de utilizar un modelo de iluminación local en cada interacción.

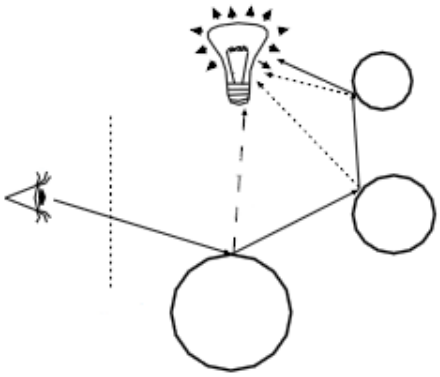


Figura 4-21 Esquema del algoritmo de trazado de caminos. Se lanzan varios rayos por cada pixel (por simplicidad sólo se muestra uno). Cada rayo sigue un único camino, seleccionando una dirección aleatoria en cada interacción (Szirmay-Kalos, 2000).

La calidad del resultado (Figura 4-22) depende fuertemente de la cantidad de rayos (también llamados fotones) lanzados por pixel. Típicamente se utilizan cientos (o miles) de rayos por pixel.

Figura 4-22 Escena renderizada utilizando el algoritmo de trazado de caminos (Chapman , 2007).

4.5.7 Conclusión

Se han realizado, y se continua haciéndolo, diferentes intentos de ejecutar eficientemente algoritmos de iluminación global en tiempo real, tanto en hardware dedicado (Schmittler, Woop, Wagner, Paul, & Slusallek, 2004) como en hardware convencional (Miller P. , 2012; García, Ávila, Murguía, & Reyes, 2012). Sin embargo, la iluminación local todavía es la opción predominante en computación gráfica fotorrealista

de tiempo real debido a que es un problema mucho más simple de resolver, que brinda resultados satisfactorios y que se ejecuta eficientemente en el hardware de las GPUs actuales. Además, en estos últimos años se han desarrollado e implementado técnicas de iluminación global simplificadas que funcionan en tiempo real eficientemente y que se complementan con el pipeline gráfico de iluminación local actual. Asimismo, otros modelos complejos pueden ser simulados de forma menos precisa pero eficientemente utilizando algoritmos simplificados que pueden ser anexados al pipeline de renderizado de iluminación local.

Por estas razones, en los próximos capítulos solo se realizará un relevamiento de técnicas fotorrealistas que se ejecuten eficientemente sobre el pipeline gráfico de las GPUs contemporáneas. Es decir, se analizarán distintas variantes de un pipeline gráfico de iluminación local complementado con algoritmos de iluminación global simplificados y otros algoritmos que permitan reproducir fenómenos físicos con bajo costo computacional.

Capítulo 5 Color

Además de modelar el comportamiento físico de la propagación de la luz, también es necesario modelar los parámetros relacionados con la percepción del sistema visual humano, entre los que se destaca el color, la respuesta perceptiva que produce el sistema visual humano al censar la luz.

La elección de los sistemas de color utilizados en computación gráfica y el tratamiento de estos colores, están ligados a la tecnología de las pantallas sobre las que se va a reproducir la imagen resultante, a las características y limitaciones del hardware sobre las que se realizan las tareas computacionales y a las particularidades propias del sistema visual humano.

En este capítulo se comenzará describiendo el espacio de color RGB dado que es el sistema de color utilizado por los sistemas *display* actuales. Debido al bajo rango dinámico de estos dispositivos y por cuestiones de desempeño, la información de color típicamente se almacena en un formato de 8 bits por canal. Además, la sensibilidad al contraste del sistema visual humano se corresponde aproximadamente con una función logarítmica, por lo que el color se transforma a un espacio logarítmico que permite obtener perceptualmente más detalle sin requerir mayor espacio de almacenamiento. La transformación a este espacio logarítmico también conocida como corrección gamma tiene una segunda función, que es la de compensar la respuesta no lineal de las pantallas. Luego se introducirá el espacio de color sRGB, derivado del RGB, que es el más popular entre los formatos del color utilizados en pantallas, y se discutirán las técnicas más populares de compresión para formatos de color RGB de bajo rango dinámico.

Representar el color en alto rango dinámico es muy importante si se tiene como objetivo lograr un renderizado fotorrealista. Por esta razón, se introducirán las principales opciones a la hora de adoptar un formato de color de alto rango dinámico y se considerará la precisión, rango y costo computacional de estos formatos. Dado el bajo rango dinámico de los dispositivos de *display*, un renderizado generado utilizando una representación de color de alto rango dinámico debe ser convertido a un sistema de color RGB de rango dinámico bajo. Dado que esta conversión es compleja e influencia significativamente la calidad de los resultados, se introducirán un conjunto de técnicas para realizar dicha tarea de manera satisfactoria.

5.1 Espacio de Color RGB

El espacio de color de mayor utilización en computación gráfica es el espacio de color RGB. En éste, cada color se representa como una combinación de tres colores primarios aditivos (rojo, verde y azul). Las razones que motivaron la adopción de este espacio de color están ligadas fuertemente a la forma en la que el sistema visual humano reconoce colores y a la tecnología utilizada en las pantallas. La especificación completa de un espacio de color RGB requiere además que se especifique el color blanco de referencia y la curva de corrección gamma, entre otros parámetros.

Con el desarrollo del estándar CIE 1931, se estableció a través de un experimento riguroso que los colores primarios aditivos ideales son los colores con longitudes de onda de 435.8 nm (violeta), 546.1 nm (verde) y 700 nm (rojo), ya que con ellos se pueden recrear todos los colores que reconoce típicamente el sistema visual humano (Figura 5-1) (Fairman, Brill, & Hemmendinger, 1997). El sistema de color CIE XY derivado de este estándar es utilizado para definir los colores primarios aditivos y el color blanco de los distintos espacios de color RGB. Las coordenadas de cromaticidad de todos los estándares de color físicamente realizables se encuentran dentro del área definida por el lugar del espectro y la línea magenta.

Dado que las pantallas utilizan emisores de luz monocromática, denominados comúnmente fósforos, la posibilidad de reproducir cada color del espectro con tan solo tres tipos de fósforos hace posible una reducción de la complejidad tecnológica de la pantalla y del costo de fabricación. Sin embargo, ninguna pantalla existente utiliza exactamente estos colores primarios aditivos, sino una aproximación de los mismos, debido en parte a la pobre eficacia lumínica de los fósforos en ciertas longitudes de onda de la luz. Asimismo, las longitudes de onda de los colores primarios aditivos utilizados suele variar en cada tipo de pantalla.

Las pantallas existentes además tienen una gran disparidad en los niveles de luminancia y en el rango dinámico que pueden alcanzar. Estas características, al igual que los colores primarios aditivos, no están completamente estandarizadas y, conjuntamente, dificultan la reproducción y la comunicación de color entre diferentes dispositivos y sistemas.

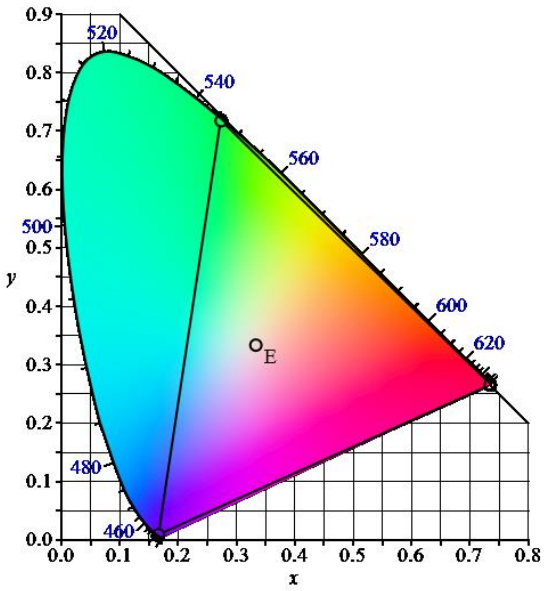


Figura 5-1 Diagrama cromático 2D del espacio de color CIE 1931. El punto E es el punto del blanco y los vértices del triángulo son los colores primarios aditivos. El área cubierta por el triángulo es el gama de color que el espacio permite reproducir (Ferrer Baquero, 2007).

5.1.1 Corrección Gamma

Típicamente, la generación de las señales que serán enviadas a las pantallas toma como entrada un conjunto de colores almacenados en un vector de tres dimensiones de 24 bits, que también corresponde a la forma en la que se almacenan computacionalmente muchas imágenes. Cada dimensión representa la intensidad de cada color primario aditivo con una precisión de 8 bits, o lo que es lo mismo, 256 valores de intensidad diferentes por cada color primario aditivo. Si bien la combinación de estos tres componentes permite reproducir 16,7 millones de colores distintos, la cantidad de niveles de brillo que es posible alcanzar es escasa en comparación con lo que el sistema visual humano puede percibir.

Asimismo, la sensibilidad al contraste del ojo humano se comporta como una función logarítmica, lo que significa que el ojo humano es más sensible al contraste cuando la intensidad de la luz es baja. Por esta razón, la información de color de cada canal suele transformarse de un espacio lineal a un espacio logarítmico, denominado espacio gamma, que permite aprovechar mejor la capacidad de almacenamiento disponible distribuyendo la precisión de forma similar a las características de sensibilidad al contraste del ojo humano. La transformación gamma tiene una segunda función, que es la de compensar la respuesta no lineal de las pantallas. La especificación completa de un espacio de color RGB requiere la selección de una curva de corrección gamma.

La transformación de un espacio lineal a uno gamma típicamente sigue una función logarítmica (o exponencial inversa):

$$L = V^{1/\gamma} \qquad \text{(Ecuación 5-1)}$$

donde γ es el valor comúnmente denominado gamma.

Similarmente, la transformación de un espacio gamma a uno lineal sigue una función exponencial:

$$L = V^{\gamma} \qquad \text{(Ecuación 5-2)}$$

El valor gamma γ y las funciones de transformación pueden variar en los distintos sistemas y pantallas. La mayoría de las pantallas son consistentes con un valor gamma igual a 2.2 sobre las funciones previamente definidas.

A pesar de ser requerido en la especificación del espacio de color RGB, no todas las texturas se benefician de la utilización del espacio gamma. En este espacio, la ganancia de precisión en los rangos bajos se logra a expensas de la precisión en los rangos altos, por lo que las texturas con predominancia de colores intensos podrían perder precisión, desde el punto de vista perceptual, si se las almacenan en espacio gamma. Si se comparan las derivadas de las curvas que definen ambos espacios (gamma y lineal) y se busca el punto donde coinciden las rectas tangenciales que definen ambas derivadas, se puede encontrar el punto de inflexión que separa los rangos en donde el espacio gamma es más preciso que el espacio lineal y

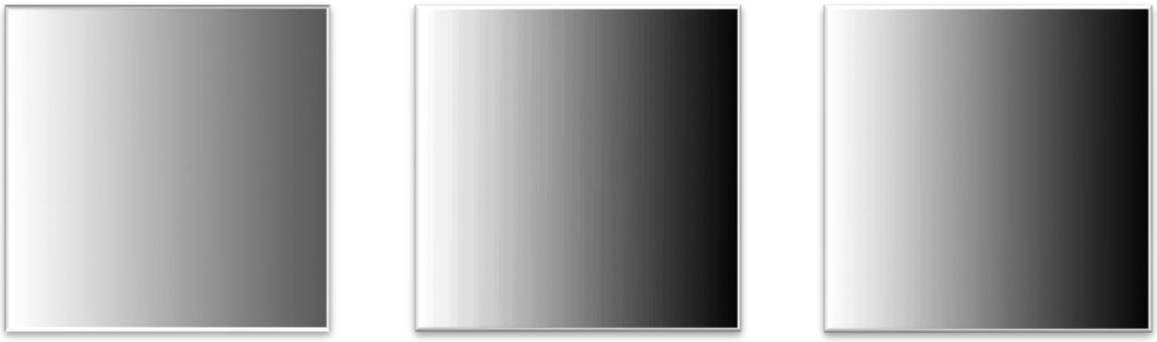
viceversa (Kaplanyan A. , CryENGINE 3: Reaching the Speed of Light, 2010). Este punto se halla resolviendo la siguiente ecuación:

$$x^{\gamma'} = x' \qquad \text{(Ecuación 5-3)}$$

Remplazando γ por 2,2 y resolviendo las derivadas se obtiene:

$$2.2\ x^{2,2} = x \qquad \text{(Ecuación 5-4)}$$

donde x es igual a $0{,}518379$ (en el espacio lineal).



(a) (b) (c)

Figura 5-2 (a) Imagen original almacenada en un formato de 16 bits por canal; todos los valores son mayores a 0,518379 (b) Imagen almacenada en el espacio gamma sobre un formato de 8 bits (c) Imagen almacenada en el espacio lineal sobre un formato de 8 bits. El contraste de las imágenes (b) y (c) se exageró de la misma manera en ambas imágenes para resaltar las bandas que se perciben en la figura (b) por falta de precisión (Hood & Finkelstein, 1986).

Esto significa que el espacio lineal tiene mejor precisión cuando el valor es mayor a 0,518379. Por lo tanto, si se analiza el histograma de la textura y se encuentra que el porcentaje de valores mayores a $0{,}518379$ es mayor que un valor arbitrario (definido de forma *ad hoc*), entonces puede resultar conveniente almacenar la textura en el espacio lineal (Figura 5-2).

Por cuestiones de desempeño o de limitaciones en el soporte de formatos que brinda el hardware, típicamente el formato de color RGB se utiliza adicionalmente para almacenar información que no es de color sino, por ejemplo, información vectorial como lo son los mapas de normales, los mapas de movimiento (*motion maps*), etc. En estos casos la utilización de una curva de corrección gamma podría ser contraproducente porque no hay típicamente un rango en el que se requiera mayor precisión que en otro. En contraste, si un mapa de profundidad es almacenado en un textura con escasa precisión (por ejemplo, un canal de 8 bits) se podría utilizar el espacio gamma para lograr mayor precisión en los zonas más sensibles, es decir, las cercanas a la cámara.

5.1.2 Importancia de calcular la Iluminación en el Espacio Lineal

Si los algoritmos de iluminación utilizan directamente los valores en el espacio gamma se obtendrán resultados que estarán considerablemente desfasados con respecto al comportamiento físico y estos errores no serán proporcionales entre sí, por lo que no pueden ser corregidos en etapas posteriores.

Descartando los fenómenos más visibles de interferencia y/o difracción, se puede asumir que si una superficie recibe una intensidad lumínica a de una fuente de luz y una intensidad lumínica b de otra fuente de luz, entonces la contribución total de luz sobre la superficie será la suma de intensidades $a + b$. Por ejemplo, si sumamos dos colores en espacio gamma 2.2 con valor (0.25, 0.25, 0.25) la suma dará como resultado el color (0.5, 0.5, 0.5). En cambio si se transforman individualmente los valores, se obtiene que un valor 0.25 en el espacio gamma equivale a un valor 0.047 en el espacio lineal; la suma de esos valores en espacio lineal dará 0.094 que convertida nuevamente en espacio gamma resulta en un valor de 0.34.

Típicamente, el error cometido consiste en no realizar la transformación del espacio gamma al espacio lineal sobre las texturas o colores de entrada y no realizar una correción gamma sobre el resultado del algoritmo de iluminación. Este doble error suele ser menos notorio que cometer solo uno de ellos, en particular si la cantidad de fuentes de luz es pequeña y la iluminación no es compleja. Sin embargo, aún en estos casos, la imagen resulta artificial debido a que los colores brillantes se saturan, los colores oscuros lucen aún más oscuros, y el detalle de baja frecuencia del color se pierde o se diluye (Figura 5-3). Si bien un artista gráfico o técnico puede compensar o disminuir la visibilidad de estos errores ajustando los parámetros y/o texturas utilizados en los *shaders*, un cambio en la iluminación de la escena hará que el error sea nuevamente percibible (Figura 5-4).

Este error también puede surgir, aunque de modo menos notorio, cuando el hardware de la GPU realiza filtrado de texturas, cuando realiza composición alfa o cuando genera los diferentes niveles de *mipmap* de una textura; esto es debido a que estos procesos involucran la interpolación de un conjunto de valores de colores. A partir del surgimiento del hardware con soporte para *shader model 4* estas operaciones se realizan considerando el espacio de la fuente (sRGB o lineal). El desarrollador debe especificar en el *sampler* de la textura o en los estados de renderizado, según corresponda, si la fuente es sRGB o lineal, tomándose por defecto el espacio lineal.

5.2 Espacio de Color sRGB

La falta de estandarización de los espacios de color RGB y la imposibilidad de utilizar de forma práctica el espacio de color CIE RGB, dio lugar a un grupo de espacios de color RGB con el objetivo de estandarizar la comunicación de colores entre sistemas computaciones y pantallas. Entre éstos se destaca el espacio de color sRGB (Stokes, Anderson, Chandrasekar, & Motta, 1996) (Figura 5-5), que define los colores primarios aditivos, el color blanco, el nivel de luminancia y una curva de corrección gamma (entre otros valores), ajustándose a las limitaciones tecnológicas de las pantallas de tubos de rayos catódicos

(CRT), las pantallas de mayor utilización en 1996, el año en que se definió este espacio de color. Este estándar ha sido ampliamente adoptado en pantallas y aplicaciones. Además, las GPUs modernas disponen de hardware específico para realizar operaciones sobre este espacio, entre las que se incluyen una rápida transformación desde el espacio de color sRGB hacia el espacio lineal (lectura de texturas) y viceversa (escritura de texturas), filtrado de texturas sRGB y composición alfa de valores sRGB.

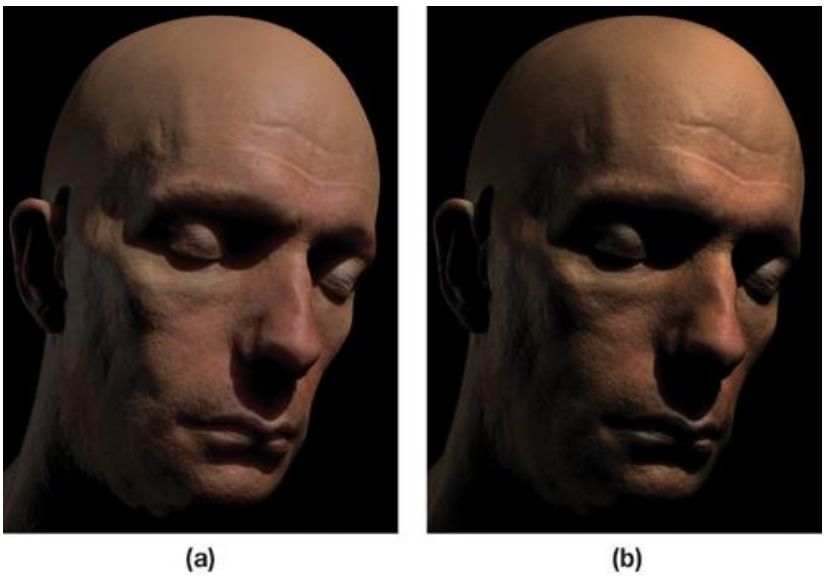


Figura 5-3 (a) Iluminación calculada en el espacio lineal realizando la corrección gamma en el resultado final (b) Iluminación calculada en el espacio gamma sin corrección gamma final (Gritz & d'Eon, 2007).

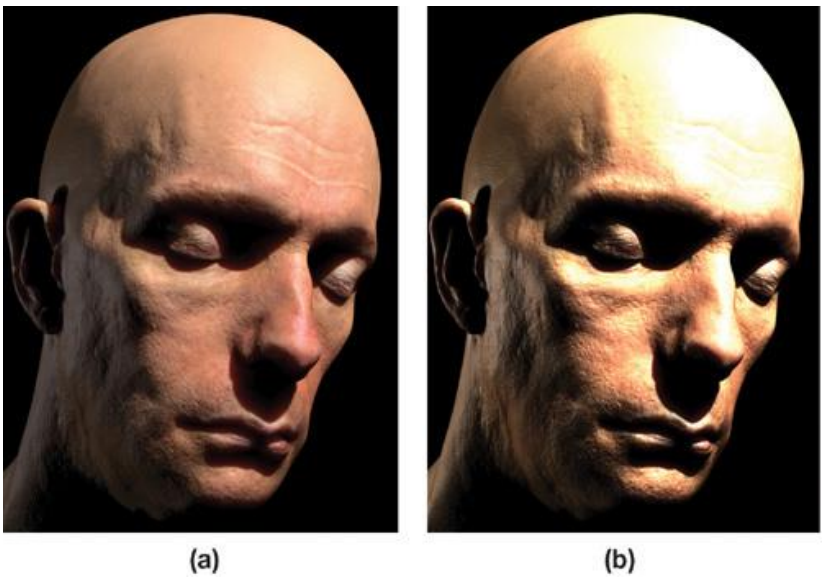


Figura 5-4 (a) Iluminación calculada en el espacio lineal y realizando corrección gamma en el resultado final. (b) Iluminación calculada en el espacio gamma y sin realizar la corrección gamma final. La intensidad lumínica se incrementó con respecto a la Figura 5-3 y los errores de iluminación de la figura (b) se acentuaron, en especial la saturación de los colores (Gritz & d'Eon, 2007).

La selección de los colores primarios aditivos del espacio sRGB ha sido criticada debido a la limitada gama de colores que se pueden representar, en especial si se los compara con espacios alternativos como el espacio Adobe RGB (Figura 5-5) o el espacio NTSC.

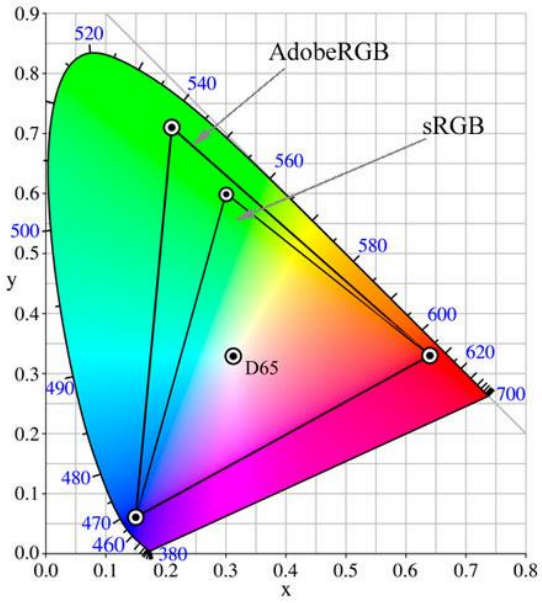


Figura 5-5 Diagrama cromático 2D del espacio sRGB sobre el espacio de color CIE 1931. Los vértices del triángulo interno son los colores primarios aditivos y el punto denominado D65 es el color blanco. El área cubierta por el triángulo es el gamut que el espacio permite reproducir. A los fines de comparación se dibuja el espacio de color Adobe RGB (Ferrer Baquero, 2007).

Tradicionalmente, la tecnología de referencia para la definición de espacios de colores fue el tubo de rayos catódicos (CRT). Actualmente, las pantallas de cristal líquido (LCD) y de plasma han extendido la gama de colores representables y, aunque el principal objetivo es el de reproducir el gamut sRGB con la mayor fidelidad posible, se han realizado esfuerzos para expandir y lograr representar correctamente espacios de colores con un gamut más amplio como el del espacio Adobe RGB (EIZO Library, 2008).

La curva gamma definida en el espacio sRGB coincide aproximadamente con un curva gamma con valor gamma 2.2 y funciones exponenciales (Figura 5-6), pero tiene un segmento en el inicio de la misma (valores cercanos al negro) en el que se comporta linealmente, con el objetivo de reducir el ruido introducido en capturas realizadas por cámaras o escáneres, fenómeno conocido como *film grain*.

La transformación del espacio sRGB al espacio lineal se puede realizar aplicando la siguiente ecuación sobre cada canal de color:

$$L_\lambda = \begin{cases} {V_\lambda}/{12{,}92}, & V_\lambda < 0{,}03928 \\ \left(0{,}055 + {V_\lambda}/{1{,}055}\right)^{2{,}4}, & V_\lambda \geq 0{,}03928 \end{cases} \qquad \text{(Ecuación 5-5)}$$

donde V_λ es el valor de color del canal que está siendo transformado.

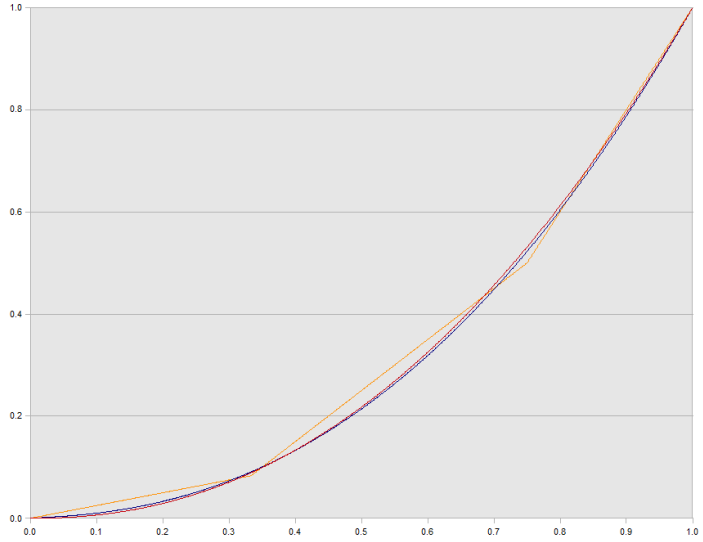
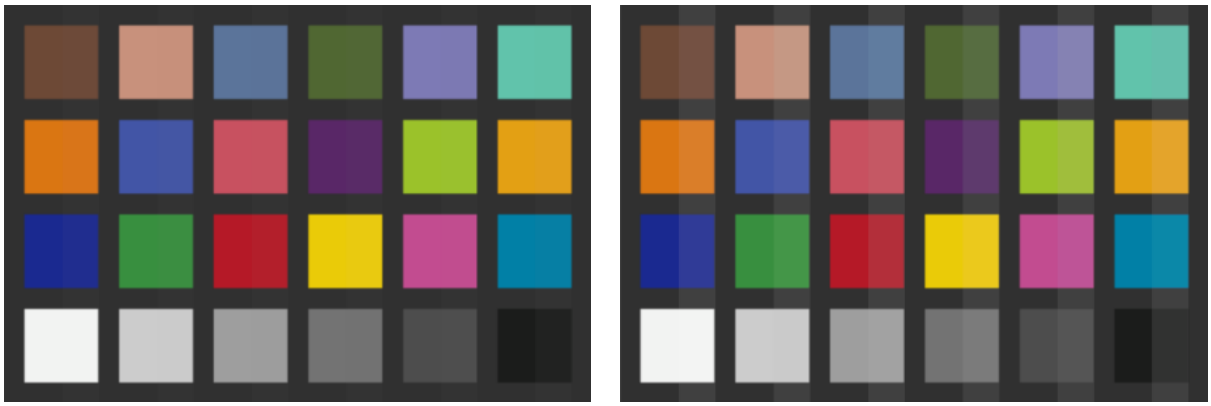


Figura 5-6 Comparación de las curvas gamma 2.2 (curva roja), sRGB (curva azul) y Xenon PWL (curva amarilla). (Hable, 2010)



(a) (b)

Figura 5-7 (a) Comparación de corrección gamma 2.2 (a la izquierda de cada cuadrado) y corrección gamma sRGB (a la derecha). Sólo el cuadrado de color negro tiene una diferencia de color significativa. (b) Comparación de corrección gamma 2.2 (a la izquierda) y corrección gamma Xenon PWL (a la derecha). La diferencia es significativa en la mayoría de los cuadros de colores (Hable, 2010).

A pesar de que el estándar sRGB normalmente se respeta, hay excepciones entre las que se destaca la corrección gamma implementada en el hardware de la consola Microsoft Xbox 360, denominada curva gamma Xenon PWL (*piecewise linear*). La curva resultante es una sucesión de 4 segmentos lineales (Figura 5-6). En tanto la diferencia entre utilizar corrección gamma 2.2 y corrección gama sRGB es pequeña (Figura 5-7a), la diferencia entre utilizar corrección gama 2.2 y corrección Xenon PWL es significativa (Figura 5-7b), en especial sobre los colores oscuros donde el sistema visual humano es más sensible. Además, el punto de inflexión que separa el rango donde la curva gamma Xenon PWL es más precisa que el espacio lineal, es menor, con un valor de 0.352941, por lo que comparativamente un mayor número de texturas se almacenan con mayor precisión en el espacio lineal. Asimismo, por defecto, el resultado de la transformación del valor de color desde el espacio gamma Xenon PWL hacia el espacio lineal se almacena

en memoria caché en un formato de 8 bits por canal, lo que produce una pérdida de precisión considerable. Alternativamente se podría utilizar una textura con formato de 16 bits por canal, pero esto provocaría almacenar innecesariamente mayor cantidad de información, degradando el desempeño (Kaplanyan A. , CryENGINE 3: Reaching the Speed of Light, 2010; Smith, 2007).

5.3 Compresión de la Información de Color

El exceso de accesos a memoria es uno de los mayores causantes de la degradación del desempeño del hardware gráfico moderno. Trabajar con formatos de color de mayor tamaño no sólo implica un aumento en el consumo de la memoria de la GPU; también conlleva un mayor consumo del ancho de banda del bus de comunicación memoria/caché de texturas, lo que provoca un aumento en el tiempo de acceso a la información de textura que no se encuentra en caché. Además, cada color consume más espacio en la caché de texturas, lo que provoca una mayor cantidad de *cache misses*, lo que reduce aún más el desempeño. Por esta razón, la utilización de formatos de color que permiten representar eficientemente el color se complementa con técnicas de compresión de texturas.

Al realizar la compresión de texturas se busca satisfacer los siguiente requerimientos (Beers, Agrawala, & Chaddha, 1996):

- *Velocidad de decodificación*. El acceso a la información debe realizarse directamente sobre los datos comprimidos y el proceso de decodificación debe ser rápido.
- *Acceso aleatorio*. Dado que no hay un orden preestablecido de acceso a los datos de la textura, cualquier algoritmo de compresión usado debe permitir el acceso aleatorio a los datos.
- *Tasa de compresión y calidad visual*. En un sistema de renderizado de tiempo real, una calidad visual mediocre podría ser tolerada si la tasa de compresión es alta.
- *Velocidad de codificación*. La codificación no suele realizarse en tiempo de ejecución, por lo que no es necesario una alta velocidad de codificación.

La indexación de color, o texturas paletizadas, es la primera técnica de compresión de textura que se propuso (Campbell, DeFanti, Frederiksen, Joyce, & Leske, 1986) y la primera en ser implementada en hardware. Esta técnica puede ser vista como un caso especial de la técnica de cuantización vectorial, en la que cada texel de la textura contiene índices a una tabla o paleta de colores (*lookup table)* limitada a 16 colores, 256 colores, 16 colores más información alfa o 256 colores más información alfa según el formato seleccionado. Si bien algunas imágenes se comprimen con buenos resultados, en computación gráfica fotorrealista la mayoría de las texturas utilizan paletas de colores de mayor tamaño, siendo los gradientes suaves de color los más perjudicados por la adopción de esta técnica de compresión. Además, cada acceso a textura necesita un acceso extra a la paleta de colores lo que podría resultar en una degradación del desempeño. Actualmente, esta técnica raramente es soportada por hardware, exceptuando dispositivos móviles y consolas de bajas prestaciones como la SONY PSP y Nintendo Wii.

Alternativamente, la información de color podría almacenarse en formatos de color de menor precisión como lo son el *formato Bgr565* que utiliza 5 bits para los canales rojo y azul y 6 bits para el canal verde, el *formato Bgra5551* que utiliza 5 bits para cada canal más un bit para el canal alfa y el *formato Bgra4444* que utiliza 4 bits para cada canal, incluyendo el canal alfa.

La compresión de texturas S3 (S3TC), normalmente denominada DXT, es una técnica de compresión con pérdida ampliamente utilizada y es soportada en todo hardware gráfico moderno. S3TC es en esencia es una extensión del concepto introducido en 1970 denominado *Block Truncation Coding* (Delp & Mitchell, 1979) y pueden presentarse de 5 formatos: DXT1, DXT2, DXT3, DXT4 y DXT5.

En S3TC, la textura se divide en segmentos de 4x4 pixeles para luego aplicar una compresión individual a cada segmento. La compresión en los distintos formatos varía solamente en la forma en la que se codifica el canal alfa. Todos los formatos crean una paleta de 4 colores por cada segmento, pero sólo se almacenan los colores de menor y de mayor luminancia; los otros dos colores se calculan como una interpolación lineal de estos colores utilizando un tercio de un color y dos tercios del otro. Además, los colores se almacenan utilizando un formato de 16 bits por color, donde los canales rojo y azul utilizan 5 bits y el verde utiliza 6 bits. Por lo tanto se necesitan 32 bits para almacenar la tabla o paleta de colores y 32 bits para almacenar los índices a la tabla (2 bits por cada texel), reduciendo los 512 bits necesarios para almacenar el segmento a tan solo 64 bits.

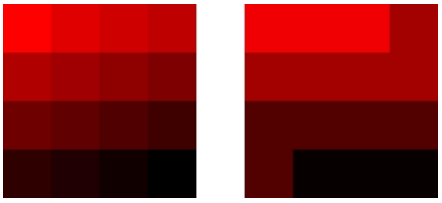
En DX1, si el primer color almacenado en la paleta tiene mayor luminancia que el segundo éste se comporta exactamente de la forma previamente expuesta. En cambio, si la luminancia del primer color es menor, el tercer color de la paleta será una interpolación lineal de los dos colores extremos utilizando la mitad de cada color, y el cuarto color de la paleta será un color completamente transparente, permitiendo de esta manera almacenar 1 bit para el canal alfa (0 o 1) a costa de uno de los colores interpolados. La tasa de compresión en este formato es 8:1 si la textura contiene información alfa y 6:1 en caso contrario.

En DX3, la información de color se almacena utilizando el esquema previamente expuesto, pero la información del canal alfa se almacena utilizando 4 bits por texel, por lo que este esquema utiliza 128 bits por segmento: 32 bits para la paleta, 32 bits para los índices a la tabla y 64 para el canal alfa, resultando en una compresión de 4:1. Dado que solo se pueden almacenar 16 valores alfa distintos de igual paso en cada texel, los gradientes suaves en el canal alfa no pueden ser almacenados satisfactoriamente. El formato DX2 es similar a DX3 pero asume que la información de color está pre multiplicada por el canal alfa y típicamente no es soportado.

En DX5, la información alfa se almacena en una paleta de forma similar a la información de color. Se almacena el valor alfa mínimo y el valor alfa máximo, excepto que éstos sean los valores 0 y 1 respectivamente. Si el primer valor almacenado es el mayor entonces se podrán indexar los valores alfa almacenados y 6 valores restantes que surgen de interpolar linealmente estos valores. En cambio, si el primer valor almacenado es menor, se podrán indexar los valores alfa almacenados, 4 valores que surgen de interpolar linealmente estos valores y dos valores extras: el valor completamente transparente y el completamente opaco. De esta forma, DX5 permite cambios graduales de mayor precisión en el canal alfa con respecto al formato DX3. El formato DX4 es similar al DX5 pero asume que la información de color está pre multiplicada por el canal alfa y típicamente no es soportado.

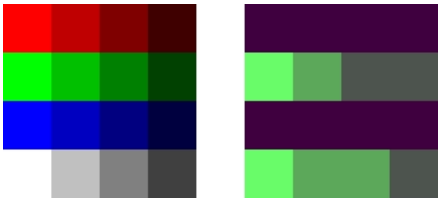
Dado que en S3TC la información de color de la paleta es almacenada utilizando un formato de color de menor precisión, los gradientes suaves pueden verse comprometidos debido a que colores similares podrían llegar a ser representados con los mismos valores en el formato de menor precisión y la compresión en un mismo segmento podría provocar la pérdida de varios colores (Figura 5-8). Conjuntamente, si la imagen contiene grandes variaciones de color en los segmentos, el segmento comprimido podría tener colores no presentes en el segmento original (Figura 5-9). Similarmente, en presencia de bordes y líneas que separan zonas de alto contraste, la compresión S3TC podría provocar la aparición de colores no presentes en el segmento original (Figura 5-10). Sin embargo, esta técnica de compresión ha demostrado ser sumamente útil en la práctica y es ampliamente utilizada. Estos problemas de compresión podrían ser reducidos realizando tratamientos a las texturas no comprimidas, como los expuestos en (Linde, 2005).

Existen otros esquemas de compresión apoyados por hardware, como 3Dc, pero su adopción no ha sido masiva.



(a) (b)

Figura 5-8 (a) Segmento sin comprimir (b) Segmento comprimido utilizando S3TC.
Utilizando esta compresión los colores similares podrían perderse.



(a) (b)

Figura 5-9. (a) Segmento sin comprimir (b) Segmento comprimido utilizando S3TC.
Utilizando esta compresión los colores sumamente diferentes resultan en colores extraños.

(a) (b)

Figura 5-10. (a) Imagen sin comprimir (b) Imagen comprimida utilizando S3TC. Ambas imágenes fueron saturadas para exponer mejor los resultados. Las zonas de alto contraste podrían provocar la aparición de colores extraños (Linde, 2005).

Los mapas MIP o Mipmaps son un método en el que a la textura principal se le anexan diferentes copias de distintos tamaños. Proveen versiones pre-filtradas de la textura principal, lo que permite reducir artefactos cuando, por ejemplo, en un fragmento debe recuperarse un valor de color de la textura que abarca una zona significativa de la misma y al mismo tiempo permiten aumentar el desempeño; una textura con Mipmaps ocupa el doble de espacio en memoria de GPU, pero si en un momento dado sólo se necesita un mapa MIP pequeño, éste ocupará mucho menos espacio en la caché de texturas (Bjorke, 2005), resultando, en esencia, en un esquema de compresión de texturas que además reduce artefactos a expensas de memoria de GPU. Si la textura de color se aplica sobre un modelo 3D o si se aplica en pantalla escalada al menos a la mitad de tamaño, entonces los mapas MIP resultan en un método de compresión de texturas efectivo (Hargreaves, Texture Filtering: Mipmaps, 2009).

5.4 Color de Alto Rango Dinámico

El sol tiene un nivel de luminancia de alrededor de 10^8 cd/m², mientras que una escena exterior iluminada sólo por las estrellas tiene una intensidad cercana a los 10^{-3} cd/m² (Figura 5-11) (Ledda, Chalmers, Troscianko, & Seetzen, 2005). Además, una misma escena puede tener grandes diferencias de intensidad lumínica entre las partes más brillantes y las partes más oscuras, es decir, tiene una alta relación de contraste o un gran rango dinámico.

El ojo humano es sensible a aproximadamente 9 órdenes de magnitud o miles de millones de niveles de brillo distintos, y responde desde las $\sim 10^{-6}$ hasta las $\sim 10^{6}$ candelas por metro cuadrado (cd/m^2) (Hood & Finkelstein, 1986). No obstante, en un momento dado, el ojo humano sólo puede distinguir una cantidad de niveles de brillo mucho menor, en general de alrededor de 5 órdenes de magnitud.

Aunque el término no está claramente definido, hablar de un alto rango dinámico (HDR) normalmente significa cubrir más de 4 órdenes de magnitud. El rango de niveles de luminancia que un formato convencional de 24 bits (como el sRGB) puede cubrir es menor (Figura 5-11) y se lo denomina *bajo rango dinámico*.

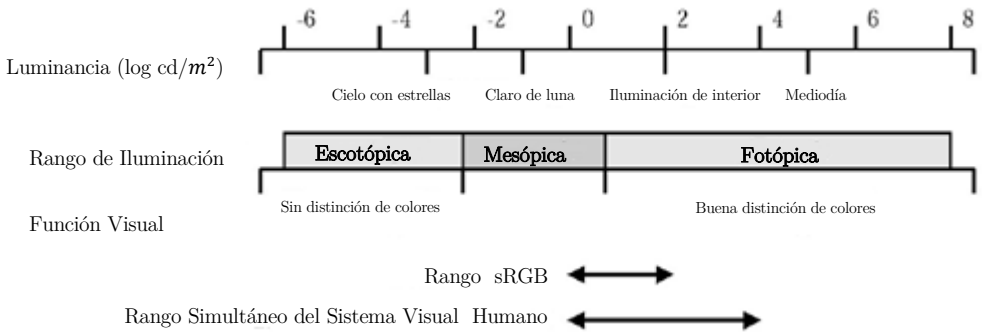


Figura 5-11 Luminancia aproximada bajo fuentes de iluminación típicas. Se realiza una comparación del tamaño del rango del sistema de color sRGB en comparación con el rango simultáneo del sistema visual humano (Ledda, Chalmers, Troscianko, & Seetzen, 2005).

El renderizado de alto rango dinámico, o iluminación de alto rango dinámico, es el renderizado que calcula la iluminación de la escena en un alto rango dinámico y mapea los colores resultantes a un rango dinámico menor que coincide con las capacidades de la pantalla. Este mapeo, denominado *mapeo tonal*, generalmente no es lineal y preserva suficiente detalle en los colores oscuros limitando gradualmente el rango dinámico de los colores brillantes, con lo que produce imágenes con buen detalle y contraste (Figura 5-12). Existen varios tipos de mapeos tonales variando en complejidad, costo computacional y realismo.

(a)

(b)

Figura 5-12 (a) Renderizado en bajo rango dinámico. (b) Renderizado en alto rango dinámico. En ésta se puede distinguir el detalle en las zonas más oscuras y los brillos especulares no lucen saturados. Además se puede simular el proceso de adaptación a la luz, fenómenos resultantes de la sobrexposición, etc. (Sousa, Crysis 2 & CryENGINE 3: Key Rendering Features, 2011)

El renderizado de alto rango dinámico además permite simular el proceso de adaptación a la intensidad de la luz que realiza el sistema visual humano. La cantidad de luz que llega a los fotorreceptores, también denominada exposición, suele ser simulada simplemente multiplicando el color resultante de la etapa de renderizado de la escena por un parámetro, que puede ser calculado automáticamente o ajustarse manualmente. Asimismo, los fenómenos de sobre exposición y la visión escotópica (distinción pobre del color) pueden simularse correctamente utilizando la información de luminancia de la escena.

5.5 Formatos de Color de Alto Rango Dinámico

Un renderizado en alto rango dinámico no necesariamente debe almacenar información en alto rango dinámico. Ciertamente es posible realizar en el *shader* de fragmentos los cálculos de iluminación, y aplicar inmediatamente sobre este resultado un mapeo tonal, produciendo un color en bajo rango dinámico que puede ser almacenado en una textura sRGB o algún otro formato de bajo rango dinámico. De esta manera se mantiene el mismo ancho de banda de acceso a memoria de la GPU y se puede realizar *anti-aliasing* del tipo *multi-sampling* y composición alfa, aún en hardware con un limitado soporte de formatos de color de *render targets* y operaciones para los mismos. Sin embargo, este método implica aplicar por cada fragmento el mapeo tonal, una operación computacionalmente costosa (aún si el mapeo es simple) y que podría ser innecesaria si el fragmento es posteriormente descartado en el pipeline de renderizado; el código fuente de cada *shader* de sombreado es levemente más complejo, interfiriendo en las tareas de optimización, análisis de rendimiento y depuración del mismo. Además, se imposibilita o dificulta la recreación de fenómenos de baja exposición y sobreexposición y el cálculo automático del valor de exposición dado que no existe una visión completa y precisa de la luminancia de la escena.

Esto motiva que en un renderizado en alto rango dinámico el resultado del proceso de iluminación se almacene en una textura que permite almacenar el color con alto rango dinámico. Esta textura es procesada posteriormente para realizar el mapeo tonal y otras operaciones de post procesamiento. Aún si se realiza en el *shader* de sombreado, parte de la información de color de entrada podría requerir un rango dinámico mayor por lo que es necesario contar con algún formato de color alternativo.

Las texturas de punto flotante de 32 bits por canal que soporta el hardware gráfico moderno pueden usarse para almacenar esta información con un rango de 79 órdenes de magnitud con una precisión de 0.000003% (el máximo error de cuantización para un valor dado) que supera de forma notoria las capacidades del sistema visual humano. Además, debido a que no está restringido a representar valores RGB positivos, puede cubrir toda el gamut percibible. A partir del hardware gráfico con soporte para *shader model 4*, sobre estas texturas se puede realizar composición alfa, filtrado y *anti-aliasing* del tipo *multi-sampling*. En cambio, en hardware anterior (incluyendo a las consolas de séptima generación) es posible que no haya soporte para estas operaciones. La principal desventaja de este formato es el requerimiento de 4 veces más espacio que las texturas sRGB no comprimidas. Además, desafortunadamente el hardware gráfico actual sólo soporta compresión de texturas de 8 bits por canal y no existen técnicas de compresión eficientes para este formato.

Existen formatos de punto flotante alternativos de menor rango que ocupan menos espacio en memoria y que son aptos para alto rango dinámico. El formato de 16 bits por canal denominado *half-format* permite representar un rango de 10.7 órdenes de magnitud, con una precisión del 0.1% (Holzer, 2007), con la posibilidad de representar valores negativos y teniendo el mismo soporte para las operaciones de interpolación de valores que el formato de 32 bits. Estas características lo hacen perfectamente apto para una adecuada representación de color en alto rango dinámico.

Alternativamente, existen formatos de punto flotante, como el RGBA1010102, que almacenan el color en 32 bits. En este formato se utilizan 7 bits para la mantisa (sin signo) y 3 para el exponente por cada canal de color, logrando un rango de 31.875 valores posibles. El rango de este formato es acotado y la falta de precisión es considerable; sin embargo, es un formato válido cuyas ventajas en desempeño y espacio de almacenamiento podrían ser determinantes. Además, podría ser el único formato de punto flotante que dispone de soporte robusto y eficiente, como ocurre en la consola Xbox 360, en la que los formatos de punto flotante de 16 bits no permiten realizar operaciones como *additive blending*.

Otra posibilidad es codificar la información de alto rango dinámico en un formato de 32 bits estándar sacrificando el canal alfa y la posibilidad de realizar operaciones como la composición aditiva. RGBE, desarrollado por Gregory Ward (Ward Larson, Real Pixels, 1991), es un formato estándar de 32 bits donde los tres primeros canales corresponden a los valores RGB y el último corresponde a un exponente compartido que se encuentra codificado en el espacio exponencial y normalizado. Con este formato se pueden representar valores RGB entre 10^{38} y 10^{-38} cd/m^2, que es significativamente mayor que lo que el sistema visual humano puede percibir. A pesar de brindar un rango de 76 órdenes de magnitud, la precisión es cercana al 1%, pero aún así suficiente para equiparar la percepción del sistema visual humano (Holzer, 2007). En este formato, la interpolación lineal de los exponentes es incorrecta, por lo que las operaciones de filtrado en el acceso a la textura o *antialiasing* deben realizarse manualmente en el código del *shader*. El hardware de la consola PlayStation 3 permite leer el contenido del *render target* en uso, por lo es posible utilizar este formato (o formatos similares) si se requiere *additive blending*.

El algoritmo de codificación RGBE es (Persson, HDR Rendering, 2006):

```
// escala = maxExp - minExp;
// bias = minExp;
// invEscalaBias = float2(1.0 / escala, -bias / escala)
float maxRGB = max(max(color.r, color.g), color.b);
color.rgb /= maxRGB;
// Se transforma el exponente a espacio exponencial y se normaliza el resultado en el rango (0, 1)
color.a = log2(maxRGB) * invEscalaBias.x + invEscalaBias.y;
```

El algoritmo de decodificación RGBE es:

```
color.rgb *= exp2(color.a * escalaBias.x + escalaBias.y);
```

Este formato puede ser combinando con las técnicas de compresión por hardware S3TC. En primera instancia, se podrían almacenar los valores en un formato DXT5 utilizando un algoritmo de codificación que considere la no linealidad del exponente, pero la pérdida de precisión en el canal del exponente podría resultar demasiado alta. Alternativamente, los valores RGB podrían comprimirse en un formato DXT1 y el exponente podría almacenarse sin ser comprimido en una textura de un cuarto de resolución con formato de 16 bits en un único canal y en el espacio lineal, aumentando la precisión del exponente a expensas de utilizar el mismo exponente en bloques de 4 pixeles. Además, de esta manera el acceso a los valores RGB podrían filtrarse por hardware. Con esta codificación se utilizan 8 bits por pixel, en vez de los 32 bits del formato RGBE original, logrando una relación de compresión igual a la compresión DXT5, a expensas de la resolución del exponente. Además, los valores RGB podrían ser transformados al espacio gamma para mejorar la precisión en los valores bajos.

Se han desarrollado formatos similares a RGBE; uno de éstos es el formato RGBS (Persson, HDR Rendering, 2006; Green & McTaggart, High Performance HDR Rendering on DX9-Class Hardware, 2006), que almacena un valor de escalado en el espacio lineal y restringe el valor máximo que los componentes de color pueden tomar.

El algoritmo de codificación RGBS es:

```
// Se restringe el valor RGB a un rango pre-establecido.
color.rgb = min(color.rgb, rangoMaximo);
float maxRGB = max(max(color.r, color.g), color.b);
color.rgb /= maxRGB;
color.a = maxRGB * invRangoMaximo;
```

El algoritmo de decodificación RGBS es:

```
color.rgb *= color.a * rangoMaximo;
```

Este formato es más rápido de codificar y decodificar que el RGBE e intenta aprovechar mejor la precisión restringiendo el rango. Sin embargo, si el valor máximo es muy alto y el valor de escalado es bajo la precisión podría resultar pobre, cosa que no sucede con RGBE por ser más preciso en los valores de menor luminancia. A pesar de esto, en la práctica no se necesitan normalmente valores máximos demasiados altos y su velocidad de codificación suele ser determinante.

RGBM, también conocido como RGBK, es una pequeña variación del formato RGBS que incluye en el proceso de codificación un redondeo hacia arriba del valor de escalado que permite principalmente rescatar y realzar los valores de baja luminancia.

El algoritmo de codificación RGBM es:

```
float maxRGB = max(color.x,max(color.g, color.b));
```

```
float M = maxRGB / rangoMaximo;
M = ceil(M * 255.0) / 255.0; // Redondeo hacia arriba
color.rgb /= (M * rangoMaximo);
color.a = M;
```

El formato RGBD sigue el mismo enfoque que el formato RGBM, pero almacena un valor divisor en vez de un valor multiplicador.

El algoritmo de codificación RGBD es:

```
float maxRGB = max(color.r, max(color.g, color.b));
float D = max(rangoMaximo / maxRGB, 1);
D = saturate(floor(D) / 255.0);
color.rgb *= (D * (255.0 / rangoMaximo));
color.a = D;
```

El algoritmo de decodificación RGBD es:

```
color.rgb *= ((rangoMaximo / 255.0) / color.a);
```

Ambos formatos, RGBM y RGBD, aunque similares, podrían suplir diferentes necesidades. Por ejemplo, el motor gráfico *MT-Framework* de *Campcom* utiliza el formato RGBM para mapas de luz y el formato RGBD para mapas ambientales, ambos comprimidos con DXT5. La razón radica en que si bien la distribución de valores del formato RGBM es mejor en la mayoría de los escenarios, la codificación de color RGBD para el rango de baja luminancia, es decir D = 1, coincide con la representación del mismo color en el formato sRGB, por lo que a un *shader* preparado para trabajar sobre texturas RGBD pueden suministrársele texturas opacas en formato sRGB. Más aún, si una textura RGBD no se beneficia de un rango mayor debido a que sus valores son cercanos al formato sRGB o por cuestiones puramente estéticas o de eficiencia, entonces puede ser transformada a sRGB simplemente cambiando la compresión DXT5 por DXT1, es decir, ignorando su valor D.

En 1998 Gregory Ward propuso otro formato denominado LogLuv (Ward Larson, The LogLuv Encoding for Full Gamut, High Dynamic Range Images, 1998). Este formato no está definido sobre el espacio de color RGB y codifica la información de color como valores de luminancia y coordenadas de crominancia CIE. Permite representar en un formato compacto y con alta precisión el rango completo de luminancia perceptible, a expensas de la velocidad de codificación y decodificación. Este formato ha demostrado ser viable en computación gráfica y ha sido incorporado en el pipeline de renderizado de proyectos como *Heavenly Sword* y *Uncharted*.

El formato propuesto presenta dos variantes, una de 24 bits y otra de 32 bits. La versión de 24 bits se divide en 10 bits para la información de luminancia (en el espacio logarítmico) y 14 bits para la información de crominancia, divididos a su vez en un sistema de coordenadas *uv*. Similarmente, la versión

de 32 bits se divide en 16 bits para la información de luminancia y 16 bits para la información de crominancia. El formato de 24 bits cubre un rango de 4.8 órdenes de magnitud y la precisión es del 1.1%; en cambio, el formato de 32 bits cubre 38 órdenes de magnitud con una precisión de 0.3% (Holzer, 2007). Dada la distribución de las partes y las limitaciones del hardware gráfico actual, en computación gráfica resulta útil principalmente la versión de 32 bits.

El algoritmo de codificación LogLuv es:

```
// Matriz para transformación de espacios.
const static float3x3 M = float3x3(
  0.2209, 0.3390, 0.4184,
  0.1138, 0.6780, 0.7319,
  0.0102, 0.1130, 0.2969);

float4 logLuvColor;
float3 Xp_Y_XYZp = mul(color, M);
Xp_Y_XYZp = max(Xp_Y_XYZp, float3(1e-6, 1e-6, 1e-6));
logLuvColor.xy = Xp_Y_XYZp.xy / Xp_Y_XYZp.z;
float Le = 2 * log2(Xp_Y_XYZp.y) + 127;
logLuvColor.w = frac(Le);
logLuvColor.z = (Le - (floor(logLuvColor.w * 255.0f)) / 255.0f) / 255.0f;
```

El algoritmo de decodificación LogLuv es:

```
// Matriz invertida para transformación de espacios en sentido inverso.
const static float3x3 InverseM = float3x3(
  6.0014, -2.7008, -1.7996,
  -1.3320,  3.1029, -5.7721,
  0.3008, -1.0882,  5.6268);

float Le = logLuvColor.z * 255 + logLuvColor.w;
float3 Xp_Y_XYZp;
Xp_Y_XYZp.y = exp2((Le - 127) / 2);
Xp_Y_XYZp.z = Xp_Y_XYZp.y / logLuvColor.y;
Xp_Y_XYZp.x = logLuvColor.x * Xp_Y_XYZp.z;
float3 color = mul(Xp_Y_XYZp, InverseM);
color = max(color, 0);
```

Existen otros formatos utilizados en renderizado de alto rango dinámico, pero no son aptos para trabajar en tiempo real por razones tales como el espacio que éstos ocupan, la distribución y tamaño de los

canales de información del formato o la baja velocidad de decodificación. El formato JPEG-HDR desarrollado por Ward y Simmons (Ward Larson & Simmons, APGV '04 Proceedings of the 1st Symposium on Applied Perception in Graphics and Visualization, 2004) consiste en realizar un mapeo tonal convencional sobre la imagen en alto rango dinámico y almacenar el resultado en un formato JPEG de 8 bits, pero anexando la información que permite reconstruir la imagen en alto rango dinámico. Esta información se codifica como una imagen de un canal de 8 bits y se anexan además fórmulas de conversión como metadatos en el archivo JPEG. Este formato permite comprimir la imagen significativamente y otorga una compatibilidad completa con el estándar JPEG debido a que un decodificador JPEG estándar ignoraría los metadatos desconocidos y mostraría en pantalla sólo la información en bajo rango dinámico producida por el mapeo tonal. El problema de este formato es el alto costo de decodificación y la dificultad de acceso aleatorio a la información.

El formato scRGB desarrollado en conjunto por Microsoft y Hewlett-Packard pretende perfeccionar el formato sRGB otorgando un mejor rango, precisión y gama de color a expensas del espacio utilizado; pero dado que las variantes ocupan 36 a 48 bits y que no existe soporte nativo por hardware, tampoco es útil en tiempo real.

Con la especificación de DirectX 11 se introdujeron dos nuevos formatos para texturas, BC6 para almacenar texturas en alto rango dinámico y BC7 para mejorar la calidad de compresión en imágenes de bajo rango dinámico.

5.5.1 Mapeo Tonal

En computación gráfica, el renderizado de alto rango dinámico permite tratar la luminancia del color de forma físicamente correcta, pero la reproducción del color resultante está limitada a las capacidades de reproducción de color de las pantallas. El mapeo tonal es una técnica que mapea un conjunto de colores en un espacio de color de alto rango dinámico a uno en un rango dinámico menor intentandoconservar la apariencia de intensidades, tonos y contraste del color. El resultado de un mapeo tonal es análogo al del proceso de adaptación lumínica del sistema visual humano, con la diferencia de que el sistema visual humano produce un resultado en un rango dinámico mayor.

La información de luminancia de una escena se puede medir con precisión y por lo tanto simular correctamente. En cambio, no está claro el funcionamiento del mapeo tonal que realiza el sistema visual humano, en donde se combinan de forma compleja aspectos psicológicos y biológicos, lo que hace difícil construir y evaluar de forma precisa y verificable los operadores de mapeos tonales.

Uno de los objetivos principales de un operador de mapeo tonal es la preservación de los detalles de contraste. La visión humana retiene el detalle aún en zonas de alto contraste, pero el rango de las pantallas es acotado por lo que es difícil lograr un balance satisfactorio que mantenga la apariencia de alto contraste y la preservación de los detalles de contraste.

Desde hace varias décadas se realizan investigaciones sobre reproducción tonal. En 1941, Jones y Condit (Jones & Condit, 1941) investigaron la relación entre el brillo y la exposición de las cámaras utilizando fotos de exteriores y propusieron métodos de predicción para ajustar la exposición. En 1967, Bartleson y Breneman (Bartleson & Breneman, 1967) realizaron una evaluación teórica de la reproducción tonal óptima en impresiones, respaldándose en datos experimentales. En 1984, Miller y Hoffman (Miller & Hoffman, 1984) propusieron el primer operador tonal, pero éste estuvo enfocado en el campo luminotécnico. En 1992, Tumblin y Rushmeier (Tumblin & Rushmeier, 1993) introdujeron el concepto de operador de mapeo tonal en el campo de la computación gráfica, y a partir de entonces se desarrollaron numerosos estudios en el tema en relación con su aplicación en computación gráfica.

Comúnmente los operadores de mapeo tonal se clasifican en operadores locales y globales. Un operador local realiza una adaptación local de intensidad utilizando el valor del pixel y un conjunto de pixeles vecinos a éste. Esta adaptación local de intensidad posteriormente gobernará la curva de compresión utilizada para mapear el color original al resultante. De esta forma, los vecinos de un pixel afectan la compresión realizada sobre el pixel, por lo que un pixel brillante rodeado de pixeles oscuros será tratado de forma diferente a un pixel brillante rodeado de pixeles brillantes. Similarmente, lo mismo ocurre con pixeles oscuros sobre un contexto brillante y oscuro respectivamente. Un operador lineal de mapeo tonal intenta emular el funcionamiento de las células ganglionares, las que provocan que el contraste se cense en pequeñas regiones llamadas campos receptivos (sección 3.2.2). En cambio, un operador global es aquel que utiliza la información derivada de la imagen completa para cada pixel evaluado, resultando en la aplicación de la misma curva de compresión para todos los pixeles de la imagen. Por esta razón, los operadores globales suelen ser computacionalmente más eficientes que un operador local.

Dado que los operadores tonales pretenden solucionar el mismo problema, hay conceptos que son comunes y aplicables a diferentes operadores tonales, como lo son el proceso de calibración y la adaptación dinámica de exposición. Por lo tanto, se describirán inicialmente estos conceptos y luego se hará un relevamiento de los operadores tonales más destacables, priorizando aquéllos cuya implementación resulta viable en computación gráfica de tiempo real.

Calibración

Idealmente los valores de entrada del operador tonal deberían estar expresados en unidades radiométricas o fotométricas, como por ejemplo la luminancia (cd/m^2). Esto le permite al operador tonal diferenciar precisamente entre una escena en un día soleado o una escena en la oscuridad de la noche. Desafortunadamente, esto no es fácil de reproducir en un renderizado de tiempo real, dado que los algoritmos de iluminación no son físicamente correctos y que las texturas, colores e intensidades lumínicas no suelen estar representadas utilizando unidades radiométricas o fotométricas y, en la mayoría de los casos, no hay siquiera una correlación entre las unidades utilizadas.

Al no conocer con precisión la escala de luminancia de la escena se podrían presentar fácilmente escenarios donde la información lumínica se opera erróneamente, por ejemplo, una escena nocturna tratada como una soleada. Para resolver este problema, los valores de color se podrían escalar y de esta manera se puede calibrar la información de entrada. El proceso de calibración también podría utilizarse para emular el ajuste de exposición que realiza la pupila del sistema visual humano o el diafragma de una cámara, con el que ajusta el rango dinámico de censado según la intensidad de la luz global de la escena.

Al no disponer de información radiométrica precisa, este escalado normalmente se realiza por prueba y error o asumiendo determinadas condiciones de iluminación. Alternativamente, se podría realizar un escalado utilizando heurísticas que infieran las condiciones de iluminación. En particular, el histograma de luminancia de la imagen resultante del renderizado podría revelar si ésta es muy brillante o es muy oscura y por consiguiente se podría obtener información para lograr un escalado automático de los valores de color.

Reinhard, inspirado en el sistema de zonas del fotógrafo Ansel Adams[4], desarrolló un algoritmo sencillo para medir la luminancia promedio de la escena:

$$Lum_{prom} = exp\left(\frac{1}{N}\sum_{x,y} log(\delta + Lum(x,y))\right) \qquad \text{(Ecuación 5-6)}$$

donde $Lum(x,y)$ representa la luminancia de un pixel en la posición (x, y) en espacio de pantalla, N es la cantidad total de pixeles y δ es un valor pequeño que evita una singularidad que podría originarse en presencia de valores muy oscuros dado que la función logarítmica tiende a menos infinito en valores cercanos a 0. El espacio logarítmico se utiliza con el objetivo de intentar compensar la respuesta no lineal del sistema visual humano.

Para obtener la información de luminancia a partir de la información de color, cada color se convierte primero al espacio de color RGB CIE XYZ:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0{,}4121 & 0{,}3576 & 0{,}1805 \\ 0{,}2126 & 0{,}7152 & 0{,}0722 \\ 0{,}0193 & 0{,}1192 & 0{,}9505 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \qquad \text{(Ecuación 5-7)}$$

Y luego a un espacio de color, como el CIE Yxy, donde se codifica o almacena la información de luminancia en un canal dedicado:

[4] El sistema de zonas es una técnica inventada por Ansel Adams a final de los años 30 para determinar la exposición óptima del film. Las zonas le proporcionan al fotógrafo un método sistemático para definir con precisión la relación entre la manera en la que ve el objeto o la escena y el resultado que alcanzará el trabajo. En cierta forma, el sistema de zonas juega el mismo papel que la gestión del color para los fotógrafos digitales. Permite una correlación directa entre el mundo visual y la copia fotográfica final.

$$\begin{aligned} Y_{\text{Yxy}} &= Y \\ x_{\text{Yxy}} &= X/(X+Y+Z) \\ y_{\text{Yxy}} &= Y/(X+Y+Z) \end{aligned} \qquad \text{(Ecuación 5-8)}$$

Esto permite poder ajustar directamente la luminancia sin afectar el tono y la saturación del color. Luego se puede hacer la conversión al sistema de color RGB utilizado las siguientes ecuaciones:

$$\begin{aligned} X &= x_{\text{Yxy}}\left(Y_{\text{Yxy}}/y_{\text{Yxy}}\right) \\ Y &= Y_{\text{Yxy}} \\ Z &= \left(1 - x_{\text{Yxy}} - y_{\text{Yxy}}\right)\left(Y_{\text{Yxy}}/y_{\text{Yxy}}\right) \end{aligned} \qquad \text{(Ecuación 5-9)}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 3{,}2405 & -1{,}5371 & -0{,}4985 \\ -0{,}9693 & 1{,}8760 & 0{,}0416 \\ 0{,}0556 & -0{,}2040 & 1{,}0572 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \qquad \text{(Ecuación 5-10)}$$

Este proceso puede ser optimizado calculando directamente la luminancia sobre el valor original:

$$Lum = (0{,}2126 \quad 0{,}7152 \quad 0{,}0722) \cdot (R \quad G \quad B) \qquad \text{(Ecuación 5-11)}$$

De esta manera sólo es necesario almacenar un valor por pixel lo que permite aumentar el desempeño reduciendo la cantidad de operaciones a realizar y el consumo de memoria.

Con el valor de luminancia y luminancia promedio calculados, la luminancia del pixel puede escalarse utilizando la siguiente fórmula:

$$Lum_{esc}(x,y) = \frac{\propto Lum(x,y)}{Lum_{prom}} \qquad \text{(Ecuación 5-12)}$$

donde $\propto$ es la denominada llave, un número relacionado con el nivel de intensidad global que no utiliza ningún sistema de unidad particular. La llave puede ser calculada utilizando diferentes fórmulas. Reinhard (Reinhard, Parameter Estimation for Photographic Tone Reproduction, 2003) la estima utilizando la siguiente fórmula:

$$\propto = 0{,}18 \times 4^{f}$$

$$f = \frac{2\log_2 Lum_{prom} - \log_2 Lum_{min} - \log_2 Lum_{max}}{\log_2 Lum_{max} - \log_2 Lum_{min}} \qquad \text{(Ecuación 5-13)}$$

donde Lum_{max} y Lum_{min} son el valores máximo y mínimo de luminancia de la imagen. El exponente f denota la distancia entre la luminancia promedio y el valor mínimo de luminancia de la imagen relativo a

la diferencia entre la luminancia mínima y máxima de la imagen. Para que la heurística sea menos dependiente de los extremos del histograma (valores que típicamente no reflejan las condiciones de luminancia global de la escena), el cálculo del valor mínimo y máximo debería excluir un porcentaje (por ejemplo 1%) de los pixeles de mayor y menor luminancia. Este método requiere analizar el histograma de la imagen, una operación que podría resultar demasiado costosa en términos de desempeño. Sin embargo, motores gráficos como la versión de 2005 del motor gráfico *Source* de *Valve* utiliza este esquema de cálculo de llave, pero el histograma se calcula cada 16 cuadros para que el costo de desempeño asociado no sea tan alto (Vlachos, 2008).

Alternativamente $\propto$ puede ser calculada utilizando la siguiente fórmula (Krawczyk, Myszkowski, & Seidel, 2005):

$$\propto = 1{,}03 - \frac{2}{2 + \log_{10}(Lum_{prom} + 1)} \qquad \text{(Ecuación 5-14)}$$

Dado que no requiere la creación y análisis de un histograma, el desempeño de esta fórmula es mejor. Además, soluciona ciertos problemas que pueden presentarse con la fórmula de Reinhard, en los que las escenas oscuras podrían verse muy brillantes y viceversa. Es posible realizar modificaciones para reducir estos artefactos indeseables (Vlachos, 2008), pero la velocidad de ejecución de la (Ecuación 5-14) la convierte en la opción más atractiva para el renderizado de tiempo real.

Adaptación Dinámica de la Exposición

Las condiciones de iluminación pueden variar drásticamente de un cuadro a otro. El sistema visual humano reacciona a esos cambios por medio del proceso de adaptación temporal. El tiempo de adaptación difiere si nos estamos adaptando de oscuro a brillante o viceversa, y si estamos percibiendo la luz con los conos o con los bastones. En computación gráfica se han desarrollado varios modelos, y se ha visto que no es necesario un modelo biológicamente preciso para obtener resultados perceptiblemente aceptables (Goodnight, Wang, Woolley, & Humphreys, 2003), por lo que generalmente se descarta modelar la adaptación completa de minutos de duración, para sólo estimar la adaptación inicial de segundos de duración, que es además significativamente más eficiente para una implementación de tiempo real.

El proceso de adaptación suele ser modelado utilizando una función del decaimiento exponencial (Pattanaik, Tumblin, Yee, & Greenberg, 2000):

$$Lum_{adapt(i)}(x,y) = Lum_{adapt(i-1)}(x,y) + \big(Lum(x,y) - Lum_{Adapt(i-1)}(x,y)\big)(1 - e^{-dt \times \tau}) \qquad \text{(Ecuación 5-15)}$$

donde dt es el tiempo de ejecución del cuadro y τ es la tasa de actualización de la función. Para contemplar la asimetría en la velocidad de adaptación, el valor τ de define en función de las características de adaptación de los conos y bastones:

$$\tau = p \times \tau_{conos} + (1 - p) \times \tau_{bastones} \qquad \text{(Ecuación 5-16)}$$

donde p indica la proporción de influencia de los conos y bastones que se calcula utilizando la luminosidad promedio del cuadro anterior y donde τ_{conos} y $\tau_{bastones}$ describen la velocidad de adaptación de los conos y bastones respectivamente. Se recomienda usar valores cercanos a 0.4 segundos para $\tau_{bastones}$ y 0.1 segundos para τ_{conos} (Krawczyk, Myszkowski, & Seidel, 2005).

El valor resultante es luego utilizado como valor de luminancia promedio del proceso de calibración; sin embargo, si se trabaja con un operador lineal de mapeo tonal en el que el proceso de calibración se realiza por zonas, esta fórmula simplificada podría generar un efecto similar al artefacto conocido como imagen fantasma, por lo que se requiere un esquema de coherencia temporal.

Si se desea prevenir que los cambios de luminancia de corta duración influencien el proceso de adaptación se podría almacenar, por ejemplo, los 16 últimos valores de luminancia promedio y actualizar el valor de luminancia si todos los valores de luminancia promedio incrementaron o decrementaron su valor. Este esquema puede ser implementado almacenando los valores históricos en una textura de 4x4. Esta textura puede ser leída con tan sólo 4 accesos a memoria utilizando filtrado por hardware, aunque acceder a la información de esta manera presupone promediar 4 valores antes de su comparación. El *shader* que realiza esta comparación a pesar de tener operaciones condicionales es muy rápido; sin embargo, almacenar los valores en la textura de valores históricos es algo más complicado y su creación debería realizarse sobre la GPU para evitar reducir el desempeño.

Otros modelos de adaptación han sido propuestos como el modelo de Ferwerda (Ferwerda, Pattanaik, Shirley, & Greenberg, 1996) y la extensión realizada por Durand y Dorsey (Durand & Dorsey, 2000).

Operadores Tonales

El operador tonal más simple es el *mapeo lineal*, que escala linealmente los valores de luminancia al rango [0, 255]. Este operador solo tiene sentido si se quieren realizar los cálculos de iluminación en bajo rango dinámico pero con mayor precisión.

Los operadores logarítmicos y exponenciales son operadores tonales levemente más complejos, y a partir de ellos se desarrollaron un número de operadores tonales con considerable éxito. Normalmente se utilizan para compararlos con otros operadores más sofisticados, tanto en costo computacional como en calidad del resultado. Sin embargo, para imágenes de rango dinámico medio, es decir imágenes con un rango dinámico levemente superior al bajo rango dinámico, estos operadores podrían ofrecer una solución competitiva con la ventaja de una mayor velocidad de ejecución.

El *operador logarítmico* es una función de compresión para valores superiores a 1, definida como:

$$Lum_D(x,y) = \frac{\log_{10}(1 + Lum(x,y))}{\log_{10}(1 + Lum_{max}))} \qquad \text{(Ecuación 5-17)}$$

donde $Lum_D(x,y)$, la luminancia comprimida, es el valor de luminancia en bajo rango dinámico ubicado en la posición (x, y) en espacio de pantalla. El valor Lum_{max} idealmente debe ser calculado utilizando la información de luminancia de la imagen;sin embargo, realizar esta operación puede resultar muy costoso para una aplicación de tiempo real, por lo que suele ser suministrado como parámetro de usuario.

El *operador exponencial* utiliza una función exponencial y se define como:

$$Lum_D(x,y) = 1 - exp\left(-\frac{Lum(x,y)}{Lum_{prom}}\right) \qquad \text{(Ecuación 5-18)}$$

Esta función está definida entre 0 para el color negro y 1 para valores infinitamente brillantes. Debido a que los valores de luminancia de la escena son siempre menores a infinito, la luminancia comprimida nunca alcanzará el valor 1. Subsecuentemente podría utilizarse una normalización para expandir y cubrir el rango completo soportado por la pantalla.

La Figura 5-13 muestra el resultado de aplicar los operadores logarítmico y exponencial sobre una imagen arbitraria. Ambos operadores logran mapear satisfactoriamente los valores de luminancia al rango soportado por la pantalla. Sin embargo, el mapeo logarítmico produce una imagen deslucida y el operador exponencial produce una imagen demasiado brillosa. Es de esperar que un buen operador tonal produzca una imagen que se encuentre en un punto intermedio.

Las diferencias entre las imágenes producidas por ambos operadores surgen por la forma de la curva de compresión y también porque uno utiliza el valor máximo de luminancia y el otro el valor de luminancia promedio. La Figura 5-14 muestra el resultado de aplicar ambos algoritmos, pero en este caso el operador logarítmico utiliza la luminancia promedio y el operador exponencial la luminancia máxima. Este pequeño cambio produce una apariencia aproximadamente inversa a la de la Figura 5-13. Las diferentes curvas de los operadores exponencial y logarítmico utilizando tanto la luminancia promedio como la luminancia máxima están graficadas en la Figura 5-15.

Drago et al. (Drago, Myszkowski, Annen, & Chiba, 2003) desarrollaron el *operador logarítmico Drago* como una extensión del operador logarítmico antes expuesto. Este operador comprime los valores de luminancia utilizando una función logarítmica, pero la base del mismo es ajustada de acuerdo al valor particular de cada pixel. Las bases varían entre 2 y 10, permitiendo la preservación del detalle y del contraste en regiones oscuras y de luminancia media y al mismo tiempo comprimiendo sustancialmente las regiones más iluminadas.

Una función logarítmica de base arbitraria b puede ser construida a partir de la función logarítmica de una base dada (por ejemplo, la base *10*) utilizando la siguiente ecuación:

$$\log_b(x) = \frac{\log_{10}(10)}{\log_{10}(b)} \qquad \text{(Ecuación 5-19)}$$

Para lograr una interpolación suave entre las distintas bases, se utiliza la función sesgo (*bias function*) de Perlin y Hoffert (Perlin & Hoffert, 1989). En esta función, la cantidad de sesgo se controla por un parámetro p de la siguiente manera:

$$sesgo_p(x) = x^{\log(p)/\log(0,5)} \qquad \text{(Ecuación 5-20)}$$

(a) (b)

Figura 5-13 (a) Resultado de utilizar el operador tonal logarítmico. (b) Resultado de utilizar el operador tonal exponencial. (Reinhard, Ward, Pattanaik, & Debevec, 2005)

(a) (b)

Figura 5-14 (a) Resultado de utilizar el operador tonal logarítmico con el valor de luminancia promedio. (b) Resultado de utilizar el operador tonal exponencial con el valor de luminancia máxima. (Reinhard, Ward, Pattanaik, & Debevec, 2005)

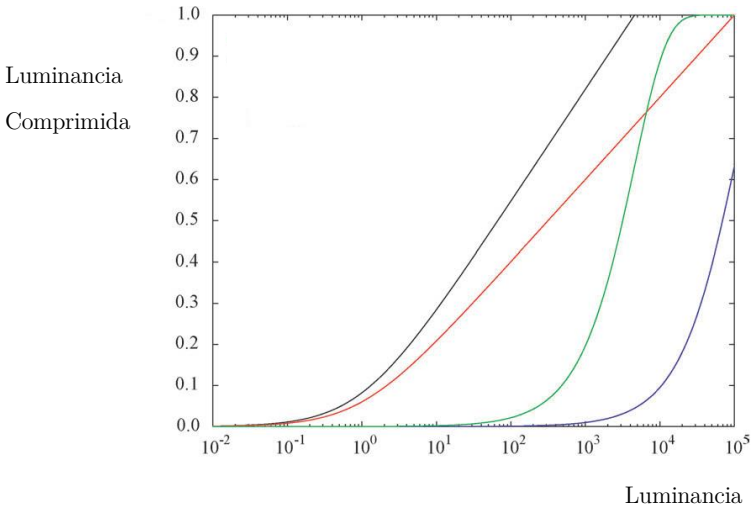


Figura 5-15 Curvas de los operadores exponencial y logarítmico. Negro: operador logarítmico utilizando luminancia promedio. Rojo: operador logarítmico utilizando luminancia máxima. Verde: operador exponencial utilizando luminancia promedio. Azul: operador exponencial utilizando luminancia máxima. (Reinhard, Ward, Pattanaik, & Debevec, 2005)

La curva del operador tonal de Drago es la misma que la del operador logarítmico, pero con una *base* b, donde b es se calcula en función al valor de luminancia del pixel con la siguiente ecuación:

$$Lum_D(x,y) = \frac{\log_b(1 + Lum(x,y))}{\log_b(1 + Lum_{max}))} \qquad \text{(Ecuación 5-21)}$$

Para interpolar suavemente entre las distintas bases las tres ecuaciones previas se fusionan y se obtiene la siguiente ecuación:

$$Lum_D(x,y) = \frac{Lum_{d,max}(x,y)/100}{\log_{10}(1 + Lum_{max})} \times \frac{\log_{10}(1 + Lum(x,y))}{\log_{10}\left(2 + 8\left(\left(\frac{Lum(x,y)}{Lum_{max}}\right)^{\log_{10}(p)/\log_{10}(0{,}5)}\right)\right)} \qquad \text{(Ecuación 5-22)}$$

Las constantes 2 y 8 limitan el rango de la base entre 2 y 10. El valor máximo de luminancia de la pantalla $Lum_{d,max}$ es dependiente de la pantalla y debería ser especificado por el usuario. Un valor de 100 cd/m^2 es apropiado en la mayoría de los casos debido a que es el valor aproximado de un monitor CRT.

El otro parámetro que debe ser especificado por el usuario es el valor p. En la práctica, un valor entre 0,7 y 0,9 suele producir buenos resultados. La Figura 5-16 muestra una imagen comprimida con diferentes valores de sesgo y se aprecia que el valor de sesgo controla el contraste de la imagen resultante. Valores altos resultan en una pérdida de contraste y una mejor compresión, mientras que valores más bajos mejoran el contraste. En la en la Figura 5-17 se grafican las diferentes curvas del operador logarítmico Drago con distintos valores p.

Figura 5-16 Resultado de utilizar el *operador logarítmico Drago* sobre una imagen arbitraria usando valores de sesgo entre 0,6 y 1 en incrementos de 0,1 respectivamente. (Reinhard, Ward, Pattanaik, & Debevec, 2005)

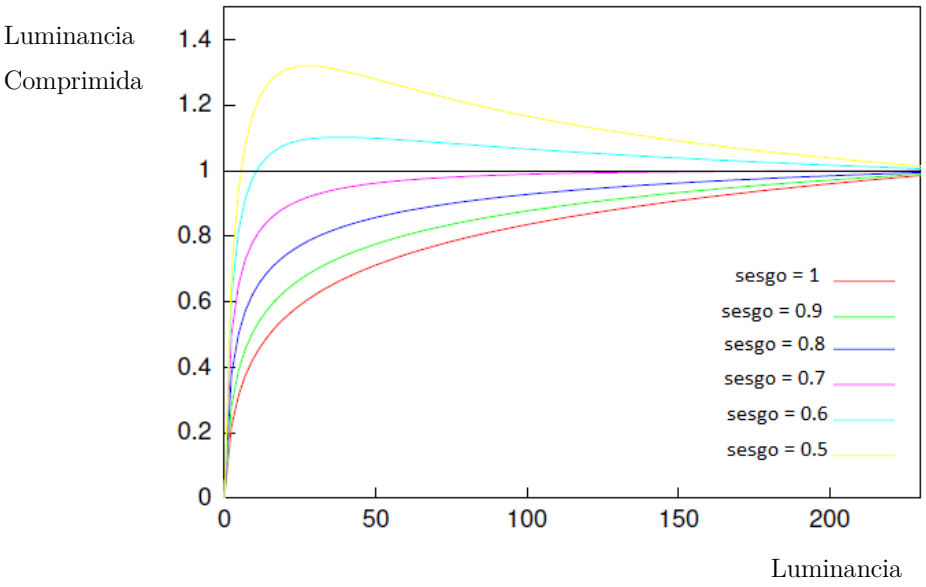


Figura 5-17 Curvas del *operador logarítmico Drago* utilizando diferentes valores sesgo. (Drago, Myszkowski, Annen, & Chiba, 2003)

Sobre un gran rango de valores el sistema visual humano produce señales que aparentan ser logarítmicas. Fuera de ese rango, las respuestas ya no son logarítmicas y comienzan a perder sensibilidad poco a poco hasta llegar a una meseta donde no hay diferencias en las respuestas. Las funciones sigmoidales, o curvas de formas S, tienen una forma (Figura 5-18) que aproxima este comportamiento razonablemente bien y son las funciones candidatas para un gran conjunto de operadores tonales.

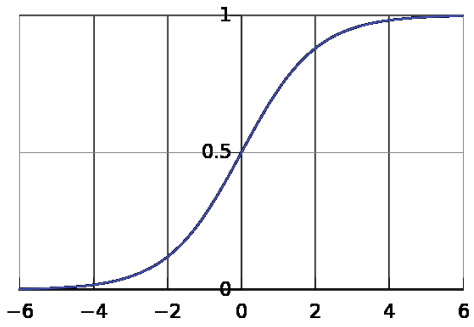


Figura 5-18 Ejemplo de una función sigmoidal.

Naka y Rushton (Naka & Rushton, 1966) fueron los primeros en medir las respuestas de los fotorreceptores y lograron representar esta información utilizando funciones sigmoidales. Reinhard y Devlin (Reinhard & Devlin, Dynamic Range Reduction inspired by Photoreceptor Physiology, 2005) profundizaron esta idea y desarrollaron un operador tonal que modela parcialmente el comportamiento de los fotorreceptores. El operador tonal incluye un simple modelo de adaptación, y parámetros para controlar el contraste, el brillo, la adaptación cromática y la adaptación a la luz:

$$
\begin{aligned}
Lum_D(x,y,i) &= Lum_s(x,y,i) + (f Lum_a)^m \\
Lum_a &= a Lum_l + (1-a) Lum_g \\
Lum_g &= c Lum_{s,prom}(i) + (1-c) Lum_{prom} \\
Lum_l &= c Lum_s(x,y,i) + (1-c) Lum(x,y) \\
m &= 0{,}3 + 0{,}7 k^{1,4} \\
k &= \frac{Lum_{max} - Lum_{prom}}{Lum_{max} - Lum_{min}}
\end{aligned}
\qquad \text{(Ecuación 5-23)}
$$

donde $Lum_D(x,y,i)$ son los valores de luminancia comprimidos para cada pixel (x,y) de la imagen para cada canal de color i, $Lum_s(x,y,i)$ son los valores de luminancia de la escena para cada canal i, $Lum_{s,prom}(i)$ son los valores de luminancia promedio para cada canal i, f es un parámetro de usuario para controlar el brillo de la imagen, m es un parámetro para controlar el contraste, a es un parámetro para controlar la adaptación lumínica, y c es un parámetro para controlar la adaptación cromática.

El mapeo tonal ha sido investigado por años en fotografía y cine, aún en la época en la que los mapeos tonales se realizaban directamente sobre la película fotográfica. Empresas como Kodak perfeccionaron el mapeo tonal en sus películas fotográficas, como la Kodak Vision Premier Film 2393 (Figura 5-19), y es por

esta razón que diferentes operadores tonales digitales intentan emular las curvas de compresión, normalmente sigmoidales, de las películas fotográficas más exitosas. Haarm-Pieter Duiker, inspirado en éstas, desarrolló un algoritmo para reproducir sus curvas características y de esta manera logró un resultado similar al conseguido en el cine y la fotografía, y es por esta razón que se lo denomina mapeo tonal fílmico (Duiker, 2003). Su algoritmo es el siguiente:

```
float3 ld = 0.002;
float referenciaLineal = 0.18;
float referenciaLogaritmica = 444;
float logGamma = 0.45;

float3 LogColor;
LogColor.rgb = (log10(0.4 * color.rgb / referenciaLineal) / ld * logGamma + referenciaLogaritmica) /
1023.f;
LogColor.rgb = saturate(LogColor.rgb);

float anchoTextura = 256;
float padding = .5 / anchoTextura;

float3 colorComprimido;
colorComprimido.r = tex2D(curvaPelicula, float2(lerp(padding, 1 - padding, LogColor.r), .5)).r;
colorComprimido.g = tex2D(curvaPelicula, float2(lerp(padding, 1 - padding, LogColor.g), .5)).r;
colorComprimido.b = tex2D(curvaPelicula, float2(lerp(padding, 1 - padding, LogColor.b), .5)).r;
```

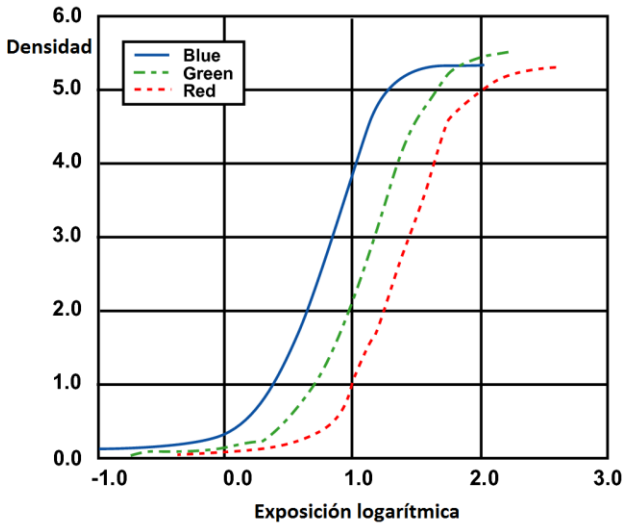


Figura 5-19 Curva característica de la película fotográfica Kodak Vision Premier Film 2393 (Kodak, 1998).

La textura curvaPelicula es una textura unidimensional en escala de grises que representa la curva de respuesta de los fotorreceptores. Los canales restantes de la textura podrían ser utilizados para almacenar una curva distinta por cada tipo de fotorreceptor. Además, este operador tonal transforma automáticamente el color resultante al espacio gamma especificado por el parámetro logGamma, por lo que no es necesaria una operación adicional de conversión de espacios.

En el contexto de renderizado de tiempo real, tres lecturas de texturas podrían ser prohibitivas. Probablemente la caché de texturas de una GPU moderna pueda almacenar completamente esta textura reduciendo la penalización de acceso. Sin embargo, Jim Hejl (Hable, 2010) realizó una aproximación de este operador utilizando sólo operaciones aritméticas de suma, multiplicación y división, lo que lo convierte en uno de los operadores computacionalmente más eficientes:

```
x = max(0, color - 0.004);
colorComprimido = (x * (6.2 * x + 0.5)) / (x * (6.2 * x + 1.7) + 0.06);
```

La desventaja de este operador es que no hay parámetros para configurar la curva de compresión, en particular en los extremos de la misma donde la función no es logarítmica, y por lo tanto no se adapta a todos los escenarios. John Hable (Hable, 2010) modificó el algoritmo de Hejl incorporando parámetros de usuario que permiten ajustar la forma de la curva de compresión.

Además de los presentados, existen una gran variedad de operadores tonales alternativos. Entre éstos pueden mencionarse *operadores globales simples* como el *operador de cuantización racional uniforme de Schlick* (Schlick, 1994), el *operador exponencial de Reinhard* (Reinhard, Ward, Pattanaik, & Debevec, 2005), etc. y operadores complejos, como el *operador local de variación espacial de Chiu* (Chiu, Herf, Shirley, Swamy, Wang, & Zimmerman, 1993), el *operador retinex de Rahman y Johnson* o el operador local desarrollado por Pattanaik denominado modelo del observador multiescala (Pattanaik, Ferwerda, Fairchild, & Greenberg, 1998). Este último es uno de los operadores más completos desarrollados y su propósito es modelar con exactitud las diferentes etapas de procesamiento del sistema visual humano que han sido completamente entendidas.

Composición del Resultado

Si el operador tonal se aplica sobre el valor de luminancia y no independientemente sobre los valores RGB, el valor de luminancia comprimida deberá ser recombinado con el valor de color original. Si se trabaja sobre el espacio de color CIE Yxy esta operación resulta trivial; en cambio, si solo se almacenó la información de luminancia, la composición de este valor con el color deberá mantener idealmente las características de tono y saturación del color original. Una posibilidad es transformar el color utilizando la ecuación (Ecuación 5-7) y luego aplicar la operación tonal sobre el canal de luminancia del color en el espacio CIE Yxy; esto requiere realizar un conjunto de operaciones adicionales, entre las que se incluye una

operación matricial. Alternativamente, dado que se debe mantener constante la relación de intensidad de los canales RGB entre los colores de entrada y de salida, la luminosidad puede ser recombinada de forma sencilla utilizando la siguiente ecuación:

$$\begin{bmatrix} R_D \\ G_D \\ B_D \end{bmatrix} = \begin{bmatrix} Lum_D(x,y)\frac{R}{Lum(x,y)} \\ Lum_D(x,y)\frac{G}{Lum(x,y)} \\ Lum_D(x,y)\frac{B}{Lum(x,y)} \end{bmatrix} \quad \text{(Ecuación 5-24)}$$

Esta ecuación puede extenderse añadiendo la posibilidad de controlar la saturación del color resultante:

$$\begin{bmatrix} R \\ G_D \\ B_D \end{bmatrix} = \begin{bmatrix} Lum_D(x,y)\left(\frac{R}{Lum(x,y)}\right)^s \\ Lum_D(x,y)\left(\frac{G}{Lum(x,y)}\right)^s \\ Lum_D(x,y)\left(\frac{B}{Lum(x,y)}\right)^s \end{bmatrix} \quad \text{(Ecuación 5-25)}$$

donde el exponente s es un parámetro de usuario que toma valores entre 0 y 1. Para un valor de 1, el método se comporta como la (Ecuación 5-24); en cambio, para valores menores la imagen se desatura progresivamente (Figura 5-20).

Figura 5-20. Imágenes resultantes de la operación tonal utilizando los valores 0.6, 0.8 y 1 para el valor de saturación s de la (Ecuación 5-25) (Reinhard, Ward, Pattanaik, & Debevec, 2005).

Capítulo 6 Iluminación Global en Tiempo Real

Los algoritmos de iluminación global presentados en la sección 4.5 son computacionalmente costosos y normalmente prohibitivos cuando se realiza el renderizado en tiempo real. Sin embargo, se han implementado algoritmos simplificados de iluminación global en tiempo real con significativo éxito, a pesar de que sus resultados son inferiores o menos adaptables a escenas dinámicas.

Una de las técnicas más sencillas consiste en ubicar y configurar una gran cantidad de luces puntuales en lugares estratégicos de la escena intentando simular con ellas la iluminación indirecta faltante. El costo computacional de evaluar un gran número de luces puntuales es alto, pero puede ser atenuado si se utiliza un pipeline de renderizado *deferred renderer* o *light-indexed renderer* (Capítulo 7). Sin embargo, ubicar y configurar las luces es una tarea de prueba y error que conlleva mucho tiempo y que es difícil de adaptar en escenas dinámicas, tanto si se realizan cambios en los objetos de la escena (destrucción, movimiento de objetos, etc.) como si se realizan cambios de iluminación (ciclos de día y noche, etc.). Además, entre otras limitaciones, sólo es posible representar interacciones no locales difusas y podrían surgir artefactos como *light bleeding*[5] debido a que las evaluaciones de visibilidad normalmente son muy costosas.

El resto de las técnicas suelen ser más complejas y requieren modificar o extender el pipeline de renderizado seleccionado. En este capítulo se realizará un relevamiento de un conjunto de técnicas exitosas de iluminación global en tiempo real. Típicamente, estas técnicas no son mutuamente excluyentes y se suelen combinar para producir resultados fotorrealistas al menor costo computacional posible, a expensas de realizar un cálculo físicamente más correcto y, por supuesto, más costoso.

Se comenzará discutiendo el enfoque más simple, la luz ambiental, una luz que se asume que se encuentra infinitamente distante, para introducir posteriormente los sistemas más eficientes para almacenar información de bajo detalle, entre los que se destacan los armónicos esféricos. Luego se introducirá la oclusión ambiental, una técnica de iluminación global que solo considera la visibilidad entre objetos y que, a pesar de no tener ningún sustento físico adecuado, por su relativa simplicidad y la calidad visual de los resultados obtenidos, ha logrado mucha aceptación. Luego se analizará la oclusión direccional que extiende la técnica anterior incorporando información de iluminación direccional. Por último, se discutirá un conjunto de técnicas de iluminación global más complejas que abarcan desde la técnica menos adaptable, mapas de luces, abarcando técnicas semidinámicas, volúmenes irradiantes y *precomputed radiance transfer*, hasta incluir las técnicas que consideran el dinamismo de los objetos y la iluminación derivadas del algoritmo general introducido en radiosidad instantánea y entre las que se destaca la técnica de volúmenes de propagación de la luz que calcula más de un rebote de luz e incorpora información de

[5] *Light bleeding* es un artefacto en el que puntos de superficie son iluminados por luces que no son directamente visibles debido a la presencia de un oclusor intermedio o por un error de cálculos o precisión en el algoritmo de sombreado.

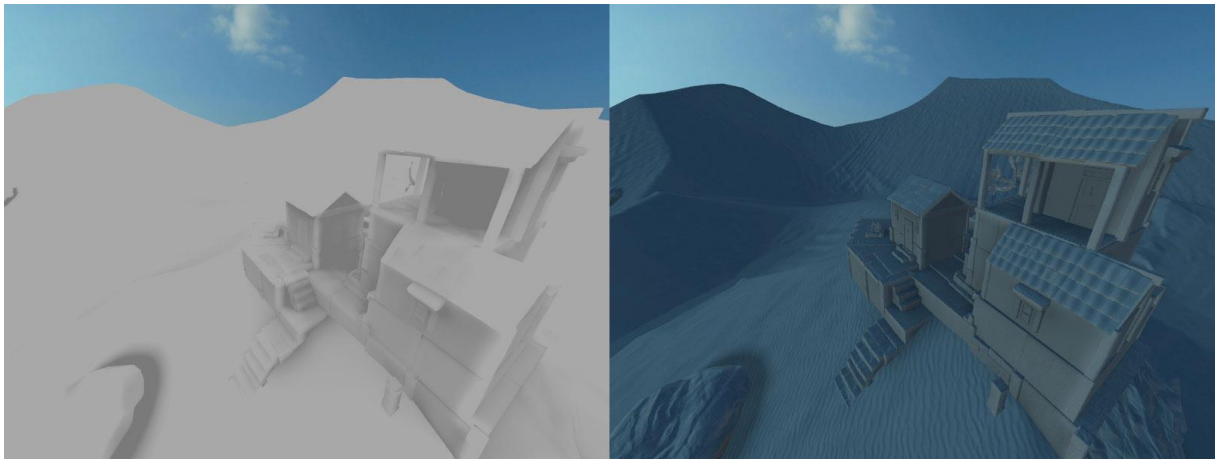
oclusión de la luz indirecta, y que ha demostrado ser suficientemente eficiente para ejecutarse en hardware contemporáneo produciendo iluminación global dinámica en escenas complejas y dinámicas.

6.1 Luz Ambiental

Una escena sin iluminación indirecta luce extremadamente irreal. Los objetos en sombra o aquéllos cuyas caras apuntan en dirección opuesta a las fuentes de luz se verán completamente negros, lo que es improbable en cualquier escena real.

La luz ambiental uniforme es el modelo más simple de iluminación indirecta, y se modela como un color constante que no depende de las fuentes de luz, la geometría de los objetos de la escena, ni el punto de vista. Este enfoque mejora sustancialmente el aspecto de la escena a un costo computacional muy bajo. La razón por la que funciona es debido a que la luz ambiental es por naturaleza iluminación difusa de bajo detalle; sin embargo, los resultados distan del fotorealismo.

Una mejor alternativa es la iluminación ambiental basada en imagen que muestrea un mapa ambiental que contiene el aporte lumínico difuso de fuentes de luz que se asume que se encuentran a una distancia infinita de la escena (Figura 6-1). Para accederlo se utilizan las normales de superficie, en vez de los vectores de reflexión, por lo que no es necesario depender de ninguna información de iluminación local.



(a) (b)

Figura 6-1 (a) Escena renderizada utilizando una luz direccional y una luz ambiental uniforme (b) Escena renderizada utilizando una luz direccional y una luz ambiental basada en imagen (Rosen, 2010).

Existe un conjunto de alternativas para almacenar estos mapas ambientales que varían en eficiencia, practicidad y fidelidad de almacenamiento. Entre los principales medios se incluyen los mapas cúbicos, los armónicos esféricos y los *ambient cube*, que serán descriptos a continuación. Esta es un área de investigación activa de importancia en computación gráfica, por lo que se siguen buscando alternativas a estos medios, como por ejemplo, la utilización de wavelets (Ng, Ramamoorthi, & Hanrahan, 2003) para

almacenar información de mayor frecuencia o la utilización de un subconjunto de armónicos esféricos denominados armónicos zonales (Sloan, Luna, & Snyder, Local, Deformable Precomputed Radiance Transfer, 2005) que permiten ser rotados de forma más eficiente, una propiedad útil en algunas técnicas.

6.1.1 Mapas Cúbicos

La alternativa más sencilla es utilizar un mapa cúbico. Al momento de muestrear este mapa se lo debe hacer en varias direcciones con el objetivo de aproximar con mayor fidelidad su contribución difusa. En (Heidrich & Seidel, Realistic, Hardware-accelerated Shading and Lighting, 1999) se propone aplicar un operador de convolución sobre el mapa cúbico, con el que se obtiene en cada texel una aproximación del aporte lumínico difuso completo para puntos de superficie con normal en dirección a ese texel. De esta manera se necesita un solo acceso a textura para obtener la contribución ambiental sobre un punto de superficie. Típicamente, el operador de convolución utilizado es un filtro de difuminado inteligente (Figura 6-2). Una ventaja de requerir información de bajo detalle es que el mapa no necesita tener mucha resolución; mapas cúbicos de 128x128x6 pixeles producen buenos resultados.

(a)

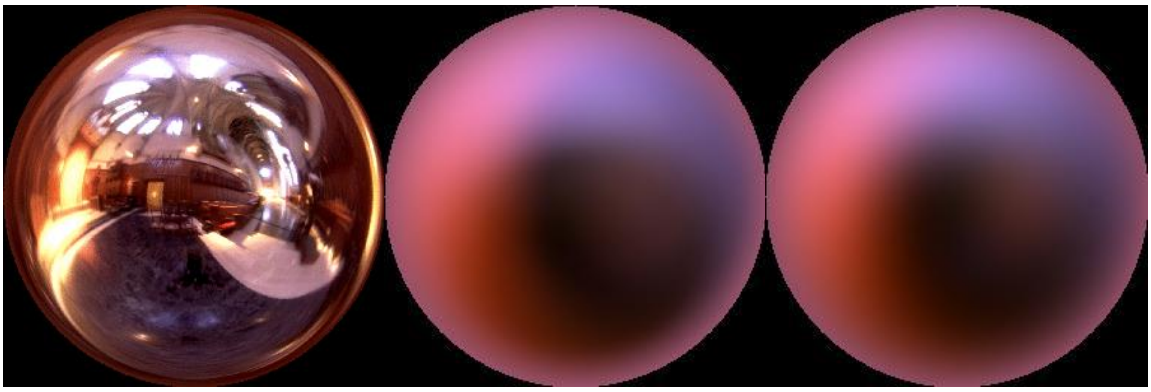
(b)

Figura 6-2 (a) Mapa de entorno o de reflexión (b) Mapa ambiental obtenido al aplicar un operador de convolución (Rosen, 2010).

6.1.2 Armónicos Esféricos

El costo computacional de mantener en memoria y acceder un mapa cúbico de tamaño reducido podría resultar alto. Los armónicos esféricos posibilitan reducir considerablemente el costo de almacenamiento y acceso a memoria manteniendo la fidelidad de la información de bajo detalle almacenada, a costa de realizar un conjunto fijo de cálculos matemáticos sencillos.

Los armónicos esféricos son un sistema matemático análogo a la transformada de Fourier, pero que se define sobre la superficie de una esfera. Intuitivamente, dada una dirección este sistema devuelve un valor que es aproximado al de la función esférica original. La aproximación es más precisa cuantas más funciones base se utilicen. Debido a que en iluminación ambiental solo se requiere información de bajo detalle y dadas las propiedades de las bases utilizadas, se puede reconstruir con gran precisión la información almacenada en un mapa ambiental con pocas funciones base (Figura 6-3).



(a) (b) (c)

Figura 6-3 (a) Mapa de entorno (b) Esfera renderizada utilizando un mapa cúbico ambiental convolucionado (c) Esfera renderizada utilizando armónicos esféricos de segundo grado (Ramamoorthi & Hanrahan, 2001).

Las funciones de la base son pequeñas piezas de señal que pueden ser escaladas y combinadas para producir una aproximación a una función. Existen muchos tipos de bases funcionales; entre éstas se pueden mencionar las bases de polinomios ortogonales, que son conjuntos de polinomios que tienen la propiedad de que si se integra el producto de dos de estos polinomios iguales se obtiene una constante y si son diferentes se obtiene cero. Si se limita al subconjunto que al integrar el producto de dos polinomios retorna como resultado 0 o 1, entonces se obtiene el conjunto de funciones conocidas como bases funcionales ortonormales.

Entre las bases funcionales ortonormales, se encuentra una familia conocida como polinomios de Legendre, que son las bases funcionales utilizadas en armónicos esféricos. Debido a que en computación gráfica es necesario almacenar colores, se utilizan solo los polinomios asociados de Legendre debido a que éstos retornan números reales (los polinomios ordinarios de Legendre retornan números complejos). Tradicionalmente los polinomios de la base de Legendre se representan con el símbolo P, tienen dos argumentos l y m, y están definidos en el rango $[-1,1]$. Los dos argumentos l (grado) y m (orden) dividen la familia de polinomios en bandas de funciones donde el argumento l es el número de banda y toma cualquier valor positivo comenzando por 0, y el argumento m toma cualquier valor entero en el rango $[0, l]$. Se pueden utilizar sistemas con una o más bandas y todos éstos constituyen las bases funcionales ortonormales. Cuantas más bandas se utilicen mayor detalle se puede reproducir. (Green R. , 2003).

Los polinomios asociados de Legendre para las tres primeras bandas son:

$$P_0^0(\cos\theta) = 1$$

$$P_1^0(\cos\theta) = \cos\theta$$

$$P_1^1(\cos\theta) = -\sin\theta$$

$$P_2^0(\cos\theta) = \frac{1}{2}(3\cos^2\theta - 1)$$

$$P_2^1(\cos\theta) = -3\sin\theta\cos\theta$$

$$P_2^2(\cos\theta) = 3\sin^2\theta$$

Utilizando la parametrización esférica estándar:

$$\vec{\omega} = (x, y, z) = (\sin\theta\cos\phi, \sin\theta\sin\phi, \cos\theta) \quad \phi \in [0,2\pi)\ y\ \theta \in [0,\pi) \qquad \text{(Ecuación 6-1)}$$

las funciones de la base de los armónicos esféricos (reales) se definen como:

$$\Upsilon_l^m(\theta,\phi) = \begin{cases} \sqrt{2}K_l^m \cos(m\phi)P_l^m(\cos\theta), m > 0 \\ \sqrt{2}K_l^m \sin(-m\phi)\, P_l^{-m}(\cos\theta), m < 0 \\ K_l^0 P_l^0(\cos\theta), m = 0 \end{cases} \qquad \text{(Ecuación 6-2)}$$

donde P_l^m es un polinomio de Legendre de grado l y orden m y K es un factor de escalado para normalizar las funciones:

$$K_l^m = \sqrt{\frac{(2l+1)(l-|m|)!}{4\pi(l+|m|)!}} \qquad \text{(Ecuación 6-3)}$$

Con el objetivo de generar todas las funciones armónicas esféricas, el parámetro m se define levemente diferente y puede tomar valores entre $[-l, l]$. Por esta razón, cada banda contiene $2l+1$ funciones de la base. Por ejemplo, un sistema de segundo grado tiene una función de la base para la primera banda, 3 para la segunda y 5 para la última, es decir utiliza 9 funciones de la base para aproximar una función esférica.

Las primeras funciones de la base se definen exactamente como lo muestra la Figura 6-4 y se pueden representar gráficamente como lo muestra la Figura 6-5.

El cálculo de los coeficientes se realiza integrando el producto de la función $f(\vec{\omega})$ y la función armónica esférica correspondiente:

$$C_l^m = \int_{\vec{\omega}} f(\vec{\omega})\, \Upsilon_l^m(\vec{\omega}) d\vec{\omega} \qquad \text{(Ecuación 6-4)}$$

Esta ecuación puede ser aproximada utilizando la integración de Monte Carlo; de esta manera, se simplifica el cálculo de los coeficientes. Particularmente, si se quieren calcular los coeficientes para un mapa cúbico se debe realizar el producto de las funciones por cada texel de la textura, luego sumar cada resultado y finalmente promediarlo teniendo en cuenta la cantidad de texels del mapa cúbico.

		Esféricas	Cartesianasas
$l=0$	$y_0^0(\theta,\phi)=$	$\sqrt{\frac{1}{4\pi}}$	$\sqrt{\frac{1}{4\pi}},$
$l=1$	$y_1^{-1}(\theta,\phi)=$	$\sqrt{\frac{3}{4\pi}}\sin\phi\sin\theta$	$\sqrt{\frac{3}{4\pi}}x,$
	$y_1^0(\theta,\phi)=$	$\sqrt{\frac{3}{4\pi}}\cos\theta$	$\sqrt{\frac{3}{4\pi}}z,$
	$y_1^1(\theta,\phi)=$	$\sqrt{\frac{3}{4\pi}}\cos\phi\sin\theta$	$\sqrt{\frac{3}{4\pi}}y,$
$l=2$	$y_2^{-2}(\theta,\phi)=$	$\sqrt{\frac{15}{4\pi}}\sin\phi\cos\phi\sin^2\theta$	$\sqrt{\frac{15}{4\pi}}xy,$
	$y_2^{-1}(\theta,\phi)=$	$\sqrt{\frac{15}{4\pi}}\sin\phi\sin\theta\cos\theta$	$\sqrt{\frac{15}{4\pi}}yz,$
	$y_2^0(\theta,\phi)=$	$\sqrt{\frac{5}{16\pi}}(3\cos^2\theta-1)$	$\sqrt{\frac{5}{16\pi}}(3z^2-1),$
	$y_2^1(\theta,\phi)=$	$\sqrt{\frac{15}{4\pi}}\cos\phi\sin\theta\cos\theta$	$\sqrt{\frac{15}{8\pi}}xz,$
	$y_2^2(\theta,\phi)=$	$\sqrt{\frac{15}{16\pi}}(\cos^2\phi-\sin^2\phi)\sin^2\theta$	$\sqrt{\frac{15}{32\pi}}(x^2-y^2).$

Figura 6-4 Las primeras 9 funciones bases de los esféricos armónicos definidas en coordenadas esféricas y cartesianas respectivamente (Jarosz, 2008).

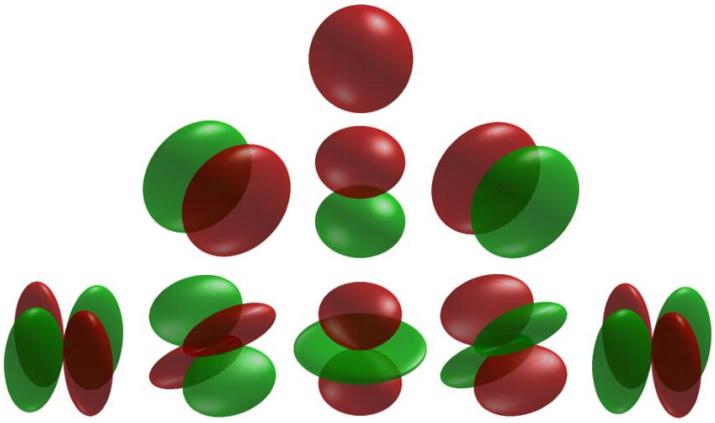


Figura 6-5 Representación gráfica de las primeras 3 bandas de los armónicos esféricos. Las porciones rojas representan regiones donde la función es positiva y las porciones verdes representan regiones donde es negativa.

Para reconstruir la aproximación de la función ($\tilde{f}$) en una dirección dada se debe resolver la siguiente ecuación:

$$\tilde{f}(\vec{\omega}) = \sum_{l=0}^{n}\sum_{m=-l}^{l} C_l^m \Upsilon_l^m(\vec{\omega}) \qquad \text{(Ecuación 6-5)}$$

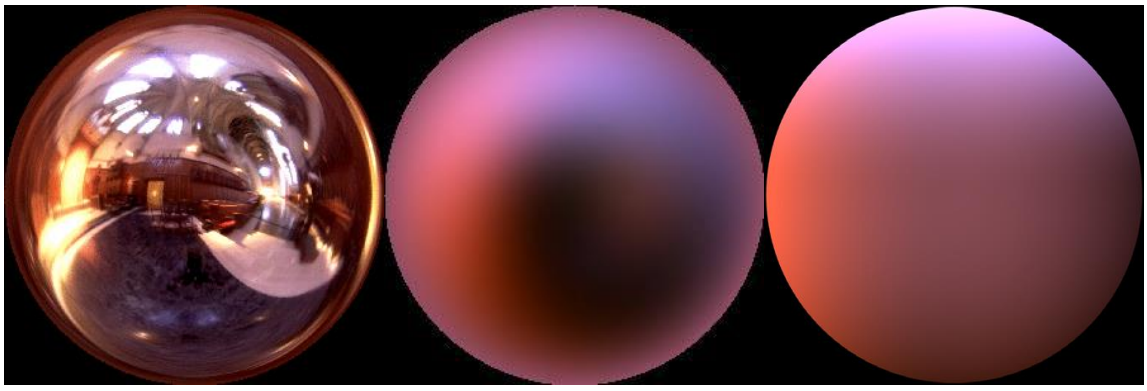
Típicamente un sistema de grado 2 es suficiente para representar con precisión un mapa ambiental, esto quiere decir que con 27 coeficientes (9 por cada canal de color RGB) se puede almacenar eficientemente un mapa ambiental. Asimismo, dados estos valores y una dirección, la reconstrucción de la función esférica, es decir el mapa ambiental, solo requiere multiplicar y sumar valores, por lo que la solución es en la mayoría de los casos considerablemente más rápida que acceder a un mapa cúbico.

6.1.3 Ambient Cube

En (Mitchell, McTaggart, & Green, 2006) se presentó un método alternativo de almacenamiento, los *ambient cube*, que en esencia son mapas cúbicos de un pixel. Al igual que los armónicos esféricos, dado su tamaño pueden ser almacenados en registros de GPU, lo que significa un rápido acceso a la información. Además, el código para muestrear estos mapas es sumamente simple y eficiente:

```
float3 normalSquared = worldNormal * worldNormal;
int3 isNegative = ( worldNormal < 0.0 );
float3 linearColor;
linearColor = normalSquared.x * cAmbientCube[isNegative.x] +
              normalSquared.y * cAmbientCube[isNegative.y+2] +
              normalSquared.z * cAmbientCube[isNegative.z+4];
```

(Mitchell, McTaggart, & Green, 2006)



(a) (b) (c)

Figura 6-6 (a) Mapa de entorno (b) Esfera renderizada utilizando armónicos esféricos de segundo orden (c) Esfera renderizada utilizando un *ambient cube* (Mitchell, McTaggart, & Green, 2006).

La fidelidad de la información almacenada es inferior a la de los armónicos esféricos de segundo grado (Figura 6-6) y carecen de algunas propiedades útiles de estos; sin embargo, proveen un mecanismo simple y

eficiente para almacenar, operar y acceder a información de bajo detalle que podría resultar útil en algunos escenarios.

6.1.4 Múltiples Mapas Ambientales

Para escenas complejas de grandes dimensiones es muy probable que se necesiten utilizar diferentes mapas ambientales en distintas ubicaciones. Las técnicas de radiosidad en tiempo real denominadas volúmenes irradiantes y *precomputed radiance transfer*, manipulan una gran cantidad de información de bajo detalle, que típicamente se almacena utilizando los métodos descriptos anteriormente. Sin embargo, su enfoque es diferente dado que buscan representar la información completa de iluminación global de la escena y con mayor granularidad, por lo que los métodos para manipular y acceder a esta información suelen ser más complejos. En esta sección solo se contemplarán los métodos básicos para manipular un conjunto reducido de mapas ambientales, dejando de lado métodos más complejos que serán discutidos posteriormente.

Como primera alternativa se podría seleccionar el mapa ambiental más cercano al objeto evaluado, permitiendo asignar manualmente mapas ambientales a objetos con el objetivo de reducir problemas de transición u oclusión, similar a técnicas utilizadas con mapas de reflexión (McTaggart, 2004). Alternativamente, se podrían interpolar dos o más mapas ambientales cercanos al objeto a renderizar; esto reduce los problemas de transición, no requiere ajustes manuales y se adapta mejor a escenas dinámicas, pero implican aumentar el acceso a memoria de texturas. Los armónicos esféricos permiten una interpolación simple y eficiente en la que sólo se interpolan los coeficientes y, además, no se producen artefactos en la transición.

En pipelines del tipo *deferred renderer* o *light-indexed renderer* la iluminación se calcula en la escena como un todo por lo que la selección de entre varios mapas ambientales se hace dificultosa y más compleja. Por esta razón, típicamente se tiene un conjunto reducido de mapas ambientales y se recurre a otra técnica (luces puntuales, mapas de luces, etc.) si se requiere mayor variedad o control. Además, debido a que la iluminación se calcula de forma diferida, la selección por distancia es el esquema más sencillo de implementar, aunque podría provocar que se perciba la influencia de fuentes de luz que están siendo bloqueadas completamente por objetos. Una solución a este problema es reservar un canal de información en el *G-Buffer* para máscaras, que indican qué mapa ambiental utilizar. Al momento de generar el *G-Buffer,* se renderiza un conjunto de volúmenes invisibles que solo afectarán el canal reservado para la máscara. Es posible hacer corresponder cada bit de la máscara con un mapa ambiental; sin embargo, se utiliza típicamente la máscara como un color en escala de grises que permite realizar una transición suave o abrupta entre dos mapas ambientales (Ownby, Hall, & Hall, 2010; Sousa, Kasyan, & Schulz, CryEngine 3: Three Years of Work in Review, 2012) (Figura 6-7).

Figura 6-7 Transición utilizando una máscara de dos mapas ambientales. En la esquina inferior derecha se puede apreciar el canal del *G-Buffer* reservado para la máscara (Ownby, Hall, & Hall, 2010).

6.2 Oclusión Ambiental

La oclusión ambiental (*ambient occlusion*) aproxima la cantidad de luz que incide en un punto sobre una superficie basándose en sus oclusores directos. Es decir, aproxima las sombras suaves de contacto que aparecen sobre los objetos y los bordes de los mismos lo que permite mejorar la percepción de profundidad, curvatura y proximidad espacial (Figura 6-8). El término oclusión ambiental fue introducido en (Landis, 2002) y fue definido con mayor precisión en (Christensen, 2003).

El algoritmo de oclusión ambiental captura un término de visibilidad general para los puntos de superficie de un objeto; este término aproxima cuánto de la luz ambiente llega a la superficie. Para esto, se trazan rayos desde un punto P de superficie sobre el hemisferio orientado alrededor de la normal del punto, contando cuántos de esos rayos interceptan a objetos vecinos dentro de un radio de influencia de valor arbitrario R y produciendo con esta información una atenuación en la irradiancia incidente en el punto.

Este simple algoritmo suele ser extendido incorporando una función de atenuación que considere la distancia entre el oclusor y el punto de superficie, lo que permite suavizar ciertos bordes oscuros que se producen por oclusores que se encuentran a una distancia R y que influencian significativamente el termino de oclusión ambiental (Figura 6-9).

Los rayos pueden ser distribuidos utilizando una distribución ponderada del coseno (*cosine-weighted distribution*), que traza más rayos alrededor de la normal lo que posibilita concentrarse en el área de mayor importancia, permitiendo reducir la cantidad de rayos emitidos. También es posible utilizar una distribución uniforme, que suelen ser más sencillas de aplicar.

Teniendo calculado el término de oclusión ambiental $AO(P)$, el término ambiental se puede calcular como:

$$K_a(\mathrm{P}) = AO(\mathrm{P}) \cdot L_a(N(P)) \qquad \text{(Ecuación 6-6)}$$

donde $N(P)$ es la normal de superficie y donde L_a es la función que retorna el valor de luz ambiental en una dirección dada.

(a)

(b)

Figura 6-8 (a) Oclusión Ambiental desactivada. (b) Oclusión Ambiental activada. Aunque las diferencias son sutiles, la segunda imagen mejora la percepción de profundidad, curvatura y proximidad espacial (Sousa, Kasyan, & Schulz, CryEngine 3: Three Years of Work in Review, 2012).

(a) (b)

Figura 6-9 Oclusión ambiental (a) sin función de atenuación y (b) con función de atenuación (Dimitrov, Bavoil, & Sainz, 2008).

6.2.1 Oclusión Ambiental en Espacio de Objeto

En su forma más sencilla, la oclusión ambiental podría precalcularse y almacenarse en los vértices de los modelos o en texturas, requiriendo en ambos casos solo un canal de datos (8 bits o menos). Los cálculos se pueden realizar con gran fidelidad dado que no son realizados en ejecución y la composición de esta información es sencilla con un bajo costo computacional. Sin embargo, este enfoque no es práctico en escenas complejas, y no permite ningún tipo de adaptación en escenas dinámicas. Es posible implementar un enfoque híbrido, es decir, precalcular un término de oclusión ambiental para los objetos estáticos y utilizar una solución de baja calidad para los objetos dinámicos, pero la inconsistencia visual entre los objetos estáticos y los dinámicos suele ser muy notoria (Kajalin, 2009).

Una de las primeras técnicas de oclusión ambiental dinámica fue la propuesta por Bunnell (Bunnell, 2005); consiste en generar dinámicamente una representación de la geometría de la escena utilizando discos orientados (posición, normal y área). En su forma más sencilla, se genera un disco por cada vértice de la escena con un área que se calcula en base a la distancia con los vecinos del vértice. El valor de oclusión para un punto de superficie o disco receptor se calcula como 1 menos un valor proporcional a la cantidad de elementos que ocluyen ese punto. Específicamente, y suponiendo la presencia de solo dos discos en la escena, se calcula:

$$1 - \frac{r \cos\theta_E max(1{,}4 \cos\theta_R)}{\sqrt{\frac{A}{\pi} + r^2}} \qquad \text{(Ecuación 6-7)}$$

donde θ_E es el ángulo entre la normal del emisor y el vector desde el emisor hacia el receptor, θ_R es el ángulo entre la normal del receptor y el vector desde el receptor hacia el emisor, A es el área del emisor, r

es la distancia entre los discos y $max(1,4\cos\theta_R)$ se utiliza para ignorar discos que no se encuentran en el hemisferio con zenit coincidente con la normal del receptor (Figura 6-10).

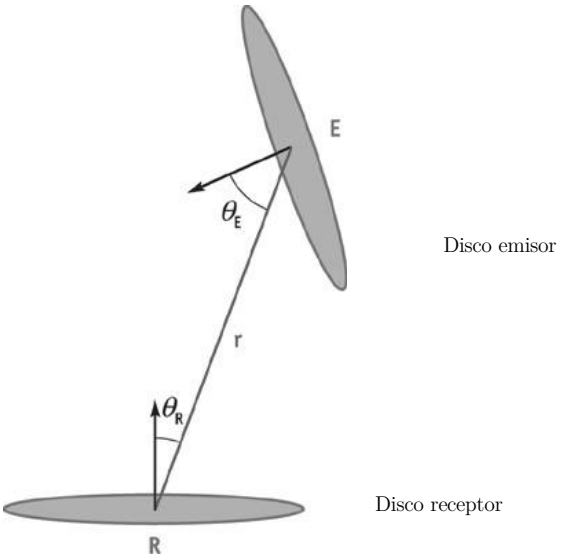


Figura 6-10 Relación entre un disco emisor y un disco receptor (Bunnell, 2005).

Luego de ejecutar este algoritmo es posible que algunos discos obtengan un valor de oclusión muy bajo (oscuro) debido a que los elementos que no son directamente visibles influyen en el resultado del algoritmo. Sin embargo, es posible realizar una segunda pasada que utiliza los cálculos de la anterior con el objetivo de atenuar el efecto de los oclusores no visibles.

A pesar de que los resultados visuales son buenos y de que el algoritmo además puede ser extendido para calcular iluminación indirecta y luces de área, su costo computacional es muy alto para escenas con alta carga poligonal. Es posible realizar un conjunto de optimizaciones, como utilizar geometría sencilla para objetos distantes, agrupar vértices cercanos o generar una jerarquía de discos para aumentar el desempeño del algoritmo de $O(n^2)$ a $O(n\log n)$. Sin embargo, estas optimizaciones hacen aún más compleja la generación de los discos orientados, las estructuras de datos crecen en complejidad y tamaño y se dificulta la creación de nuevas escenas.

También es posible la utilización de armónicos esféricos para almacenar y aproximar la oclusión ambiental de una escena, como en la técnica propuesta en (Ren, y otros, 2006). Para esto se crea una representación geométrica de la escena utilizando esferas, que es la forma geométrica que permite la mayor rapidez en cálculos de intersección de elementos, es una representación fácil de generar y animar y, gracias a su simetría, es fácil de proyectar en una representación que utiliza armónicos esféricos. Luego, por cada punto P muestreado según un criterio arbitrario, se genera una representación con armónicos esféricos de la visibilidad direccional desde ese punto de vista (Figura 6-12). Por último, se genera el término de oclusión ambiental accediendo al armónico esférico más cercano, de forma similar a un mapa ambiental.

Aunque esta técnica permite aproximar oclusión ambiental para escenas dinámicas (Figura 6-11), el algoritmo es muy dependiente de la complejidad geométrica de la escena y es típicamente prohibitivo. Una

variación de esta técnica permite incluir información de iluminación direccional (Sloan, Govindaraju, Nowrouzezahrai, & Snyder, 2007).

Figura 6-11 Oclusión ambiental generada utilizando una representación con armónicos esféricos (Ren, y otros, 2006).

(a) (b)

Figura 6-12 (a) Función de visibilidad (b) Aproximación utilizando armónicos esféricos (Ren, y otros, 2006).

6.2.2 Oclusión Ambiental en Espacio de Pantalla

Alternativamente, es posible calcular el término de oclusión ambiental en el espacio de la pantalla y a tales algoritmos se los denominan oclusión ambiental en espacio de pantalla (*Screen-Space Ambient Occlusion* o simplemente SSAO). El término fue introducido por Mittring (Mittring, Finding Next Gen - CryEngine 2, 2007).

En general, estos algoritmos utilizan la información de profundidad de los objetos renderizados (mapa de profundidad) como aproximación discreta de la geometría de la escena. Posteriormente se muestrea el mapa de profundidad y se calcula, utilizando típicamente solo esta información, el término de oclusión ambiental. En esencia, se trazan rayos desde cada punto de superficie visible por la cámara hacia otros puntos visibles (y típicamente cercanos en el espacio de la pantalla).

A pesar de que se utiliza una sola capa de profundidad y se ignora completamente la geometría fuera del frustum de la cámara, los resultados son buenos (Figura 6-8) y el costo computacional significativamente menor que el de otras técnicas. Además, este método no depende de la complejidad de la escena, no necesita ningún tipo de preparación para escenas nuevas, no se ve afectado por el dinamismo de la escena (aunque es posible que cuando la cámara se mueve y en presencia de pequeños objetos, surja ruido *jitter*), su implementación no afecta considerablemente al resto del pipeline gráfico y su costo computacional no fluctúa considerablemente entre cuadros.

Una de las mayores dificultades a resolver con este método es que no es posible trazar rayos en el mapa de profundidad utilizando geometría analítica. Una alternativa es utilizar la técnica de *ray marching* que muestrea un conjunto de valores en la dirección de avance del rayo a un paso definido y arbitrario. Intuitivamente se utiliza el mapa de profundidad como un mapa de altura y se interceptan rayos 3D con éste; para esto, se divide el rayo 3D en un conjunto de segmentos de igual paso y se verifica por cada uno de éstos (es decir para el punto extremo) si el segmento intersecta el mapa de altura. El proceso termina cuando se encuentra una intersección o cuando se alcanza una distancia determinada por la cantidad de pasos y la longitud de los mismos. La proyección se puede realizar en el espacio de la pantalla ($uv = uv + paso$) o en el espacio 3D ($P = P + paso$) proyectando cada punto muestreado en el espacio de la imagen. Si el valor muestreado se encuentra a una distancia mayor que un valor arbitrario R entonces este valor se ignora.

Si se define por cada pixel P un espacio tangencial, el hemisferio orientado sobre la normal puede ser muestreado utilizando un sistema de coordenadas esféricas con el eje zenit alineado con la normal del punto P. Por lo tanto, para calcular la oclusión ambiental se necesita un conjunto de direcciones distribuidas en el espacio de la pantalla alrededor de la normal, un conjunto de rayos por cada dirección, cada uno con un ángulo de elevación diferente en coordinadas esféricas y un número de pasos por cada rayo. Cuando un rayo intersecta el campo de altura en un punto, una contribución ambiental se suma al valor de oclusión actual sobre el punto P.

Muestrear muchos valores en el mapa de profundidad por cada punto de superficie P evaluado no es eficiente y podría resultar prohibitivo; más aún, si éstos no se encuentran cercanos en espacio de pantalla. Por esta razón, Kajalin (Kajalin, 2009) propuso que en vez de calcular la cantidad de oclusión producida por la intersección de rayos con geometría, se aproxime este término calculando la cantidad de geometría sólida alrededor del punto de interés. Específicamente, se muestrea el mapa de profundidad utilizando un conjunto de puntos de desplazamiento predefinidos por un *kernel* y distribuidos alrededor de una esfera ubicada sobre el punto de superficie P. Si el valor de profundidad del punto P es mayor que el valor de profundidad muestreado, entonces el punto P se encuentra ocluido (Figura 6-13). Si el punto P se encuentra cercano a un borde saliente el resultado será un valor bajo de oclusión; a pesar de que el

resultado no es correcto, los bordes se acentúan y visualmente se resalta la silueta de la geometría, lo cual puede ayudar a acentuar la sensación de profundidad de la escena.

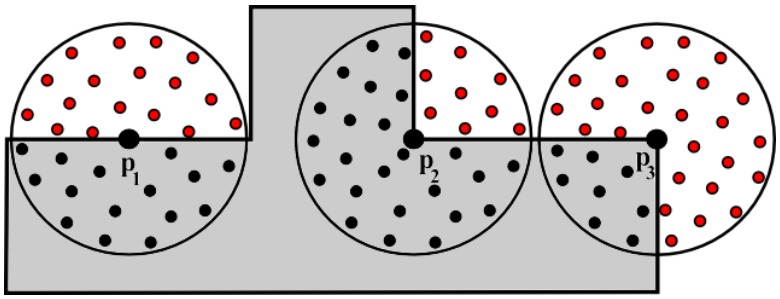


Figura 6-13 En este algoritmo se calcula el valor de oclusión calculando la cantidad de geometría sólida alrededor del punto de interés. El área gris representa la información almacenada en el mapa de profundidad; p_1, p_2 y p_3 son los puntos de superficie evaluados y los puntos negros y rojos son los puntos en los que se evalúa la presencia de geometría, siendo los rojos los puntos no ocluidos y los negros los puntos que sí lo están (Kajalin, 2009).

Muestrear los valores utilizando siempre los mismos puntos de desplazamiento es visualmente notorio, en especial si este conjunto es pequeño. Una solución sencilla consiste en rotar estos puntos de forma aleatoria. Además, la densidad de puntos de muestreo cercanos al punto P idealmente debe ser mayor que en la superficie de la esfera; de esta manera, la geometría cercana (al menos en espacio de pantalla) tiene mayor peso que la geometría lejana, lo que reduce la necesidad de utilizar una función de atenuación. Asimismo, muestrear valores cercanos en el espacio de la pantalla reduce el costo de acceso a memoria, debido a que es posible que muchos de estos valores se encuentren en la caché de texturas. Si bien se muestrean valores cercanos en el espacio de la pantalla, estos valores podrían estar alejados en el espacio del mundo, por lo que es necesario atenuar o ignorar valores que se encuentran considerablemente alejados.

Este método simple produce resultados aceptables (Figura 6-14) con un costo computacional bajo con respecto a otras métodos; sin embargo ignora, por ejemplo, una evaluación que indique si los puntos comparados son visibles entre sí o existe un oclusor intermedio.

Figura 6-14 Oclusión ambiental generada utilizando el algoritmo propuesto por Kajalin (Kajalin, 2009)

Baloil y Sainz (Bavoil & Sainz, Image-Space Horizon-Based Ambient Occlusion, 2009) propusieron la técnica *Horizon-Bassed Ambient Occlusion* (HBAO) que es ampliamente utilizada en la actualidad. Similar a la anterior, esta técnica proyecta los puntos de muestreo directamente en el espacio de la pantalla, pero sigue utilizando el concepto de rayos de forma análoga a la técnica de *ray marching*. Dado que la proyección en perspectiva tiene la importante propiedad de que las líneas en el espacio de vista se proyectan en líneas en el espacio de la pantalla, los rayos se pueden trazar directamente en 2D.

Además, este algoritmo se basa en el concepto de mapeo horizonte (Max, 1986) y *Horizon-Split Ambient Occlusion* (Dimitrov, Bavoil, & Sainz, 2008). Se utilizan coordenadas esféricas con el eje zenit orientado paralelo al eje Z en el espacio de vista y, por cada dirección alrededor del punto P, se calculan ángulos horizonte incrementales mientras se avanza por los segmentos del rayo. El ángulo horizonte marca el menor ángulo en el que todos los rayos con ángulo menor o igual a éste intersectan el mapa de altura (Figura 6-15). Esto es verdad si se asume que el mapa de altura es continuo, algo que no siempre es verdadero, pero que en la práctica no produce artefactos notorios en la gran mayoría de los casos.

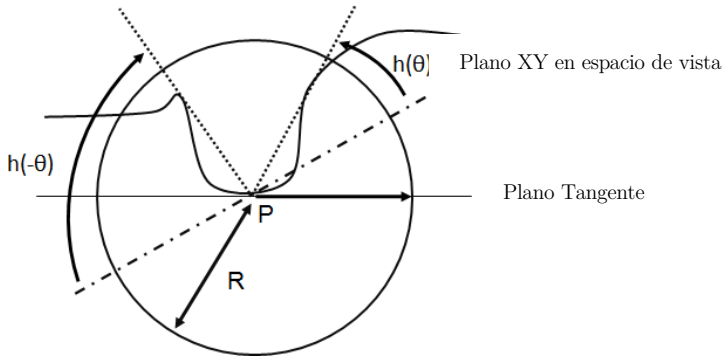
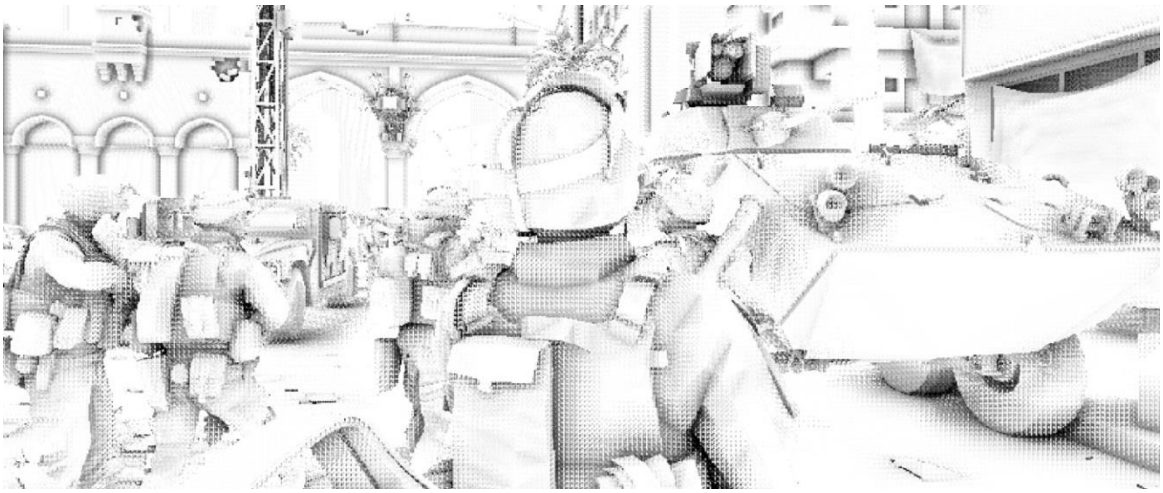


Figura 6-15 El ángulo horizonte $h(\theta)$ es el ángulo más pequeño en el que todos los rayos con ángulo menor o igual a éste, intersectarán el mapa de altura (Bavoil & Sainz, Image-Space Horizon-Based Ambient Occlusion, 2009).

A partir de estos ángulos horizonte, del plano tangente (idealmente se requiere el acceso al mapa de normales aunque es posible simplemente utilizar una aproximación) y de una función de atenuación, se calcula el término de oclusión ambiental. Este algoritmo produce resultados visualmente superiores a un costo intermedio entre las dos técnicas previamente introducidas (Figura 6-16).

En oclusión ambiental en el espacio de la pantalla, se muestrean pocos valores por punto debido a que es el principal cuello de botella del algoritmo. Aún si se utiliza una distribución con *kernel* aleatorio es posible que se genere ruido bastante notorio, por lo que típicamente se suele aplicar algún tipo de filtro. Como primera alternativa lógica, se podría aplicar un filtro de difuminado (*blur*); sin embargo, con el objetivo de preservar los bordes definidos, es recomendable usar un filtro del tipo *cross-bilateral* (Eisemann & Durand, 2004) que solo aplica el difuminado para pixeles con similar profundidad (bordes en objetos distantes) y opcionalmente similar orientación de sus normales (bordes en objetos cercanos o en el mismo objeto). A pesar de que los filtros bilaterales no son matemáticamente separables, para mejorar el desempeño se pueden dividir en dos pasadas, una vertical y una horizontal, produciendo muy pocos artefactos notorios (Figura 6-17).

Figura 6-16 Oclusión ambiental generada utilizando el algoritmo HBAO (Bavoil & Sainz, Image-Space Horizon-Based Ambient Occlusion, 2009)



(a)

(b)

Figura 6-17 (a) Resultado del algoritmo de *Horizon-Based Ambient Occlusion* (b) Aplicación de un filtro *cross-bilateral de dos pasadas sobre el resultado anterior.* (Bavoil & Andersson, Stable SSAO in Battlefield 3 with Selective Temporal Filtering, 2012)

Si la cámara está en movimiento y además se encuentran presentes en la escena objetos pequeños (relativos al radio de atenuación y al punto de vista), como lo pueden ser grillas o plantas, es posible que se genere ruido *jitter*. Aunque el filtrado de difuminado ayuda a reducir la presencia de este tipo de ruido, no es posible reducirlo completamente, por lo que podría resultar necesario aplicar un filtrado temporal (Mattausch, Scherzer, & Wimmer, 2010). Además, este filtro permite mejorar la calidad de los resultados sin aumentar significativamente el costo computacional.

El filtrado temporal utiliza la técnica de reproyección reversa (Nehab, Sander, Lawrence, Tatarchuk, & Isidoro, 2007; Scherzer, Jeschke, & Wimmer, 2007) que en esencia reúsa cálculos de los cuadros previos y los refina con el tiempo. Esto permite mantener una cantidad baja de valores muestreados por cuadro, mientras que al mismo tiempo se hace uso de los cálculos de cientos de valores que fueron muestreados en los últimos cuadros renderizados, evitando así variaciones significativas de los resultados entre los diferentes cuadros. Esta técnica es útil para un conjunto de aplicaciones, entre las que se incluyen sombras, *antialiasing* y *motion blur*. La oclusión ambiental es otro buen candidato para esta técnica debido a que no hay una dependencia direccional con respecto a las fuentes de luz o al punto de vista y debido a que la técnica considera solo la geometría cercana al punto de evaluación.

La proyección reversa asocia pixeles del cuadro actual con pixeles de cuadros previos que representan la misma posición en el espacio del mundo. Se utilizan dos *render targets* en configuración *ping-pong*, uno para la información del cuadro actual y uno para almacenar la información de los cuadros previos.

Para geometría estática es posible calcular la reproyección utilizando las matrices de vista y de proyección del cuadro anterior y del cuadro actual. Para un pipeline gráfico diferido es necesario además almacenar los valores de profundidad del cuadro actual y del cuadro previo. Con la profundidad del pixel y la ubicación de éste en el espacio de la pantalla o proyección, es posible reconstruir su posición en el mundo.

En escenas dinámicas es necesario aplicar la transformación completa del vértice para el cuadro actual, como así también para el cuadro anterior. Para un pipeline gráfico diferido, la posición previa necesita ser accedida en una pasada en la que la información de transformación ya ha sido perdida. Una solución es almacenar la profundidad anterior menos la profundidad actual en un *render target* dedicado. Debido a que son valores de desplazamiento y no valores absolutos, es posible almacenar esta información en un canal de 16 bits en vez de en uno de 32 bits, donde típicamente se guardan posiciones o profundidades.

El objetivo principal de esta técnica es distribuir los cálculos de oclusión por sobre varios cuadros usando reproyección. Siempre que sea posible, se toma la solución de los cuadros previos y se refinan con la contribución de los nuevos valores calculados en el cuadro actual. Teóricamente este enfoque permite acumular una cantidad arbitraria de valores en el tiempo. En la práctica, en cambio, esto no es aconsejable debido a que la reproyección no es exacta y requiere para su reconstrucción la utilización de un filtrado bilineal, por lo que se introducen y acumulan nuevos errores en cada paso. Por esta razón se agrega alguna función para reducir con el tiempo la influencia de cálculos antiguos. Una posible función es la secuencia Halton. La función seleccionada podría requerir almacenar información adicional, pero típicamente esta información puede combinarse en los canales no utilizados de la textura que almacena los valores de oclusión ambiental.

Cuando se reproyecta un fragmento es necesario verificar si el pixel del cuadro previo realmente se corresponde con el pixel evaluado en el cuadro actual, es decir, si el valor reproyectado es válido o no. Para detectar si un pixel es inválido se suele verificar un conjunto de condiciones (Mattausch, Scherzer, & Wimmer, 2010). El test de desoclusión es el más utilizado y uno de los más rápidos de evaluar. Éste compara la diferencia de profundidad de los pixeles:

$$\left|1 - \frac{p_c}{p_{c-1}}\right| < \varepsilon \qquad \text{(Ecuación 6-8)}$$

donde p_c es la profundidad del pixel actual, p_{c-1} es la profundidad del pixel reproyectado y ε es un umbral arbitrario. Si la diferencia es mayor al umbral ε entonces se asume una desoclusión y se descarta el valor reproyectado. Típicamente, el test de desoclusión es suficiente para descartar la mayoría de los pixeles inválidos en escenas estáticas. En cambio, en escenas dinámicas, también puede ser necesario descartar pixeles que están siendo afectados por otros que se encuentran en movimiento. Evaluar correctamente esta condición es prohibitivo, pero puede aproximarse con un conjunto de fórmulas *ad hoc* que utilizan no solo la información de profundidad de los pixeles, sino también los propios valores de oclusión y/o el mapa de normales de la escena. Por último, se debe verificar que el pixel reproyectado no se encuentre fuera del frustum de la escena.

El filtrado temporal puede ocasionar artefactos sobre la geometría estática debido a que éstos pueden recibir (si no se hace nada para evitarlo) información de oclusión de geometría dinámica; a estos artefactos se los denomina *trailing*. En (Bavoil & Andersson, Stable SSAO in Battlefield 3 with Selective Temporal Filtering, 2012) se propuso clasificar los pixeles como estables o inestables. Debido a que los objetos que necesitan el filtrado temporal son predominantemente los objetos pequeños relativos al punto de vista y al radio de atenuación, se realiza una pasada que clasifica como inestables a los pixeles que tienen al menos una discontinuidad (diferencia de profundidad por sobre un umbral) con respecto a sus vecinos. Luego se aplica un filtro de dilatación de 4x4 a la máscara resultante del paso anterior con el objetivo de abarcar su área de influencia. Los pixeles alejados pueden ser clasificados erróneamente como inestables si el umbral no tiene en cuenta la profundidad del pixel, pero típicamente estos artefactos no son notorios en objetos alejados (Figura 6-18).

La información disponible en el filtrado temporal puede ser utilizada para mejorar el filtrado de difuminado. La técnica de muestreo adaptativo (*adaptative sampling*), por ejemplo, permite no sólo mejorar la calidad de los resultados sino también el desempeño del algoritmo (Mattausch, Scherzer, & Wimmer, 2010; Bavoil & Andersson, Stable SSAO in Battlefield 3 with Selective Temporal Filtering, 2012). Esta técnica aumenta o reduce la cantidad de valores muestreados dependiendo del estado de convergencia de los valores de oclusión. Intuitivamente, si para un punto evaluado se pudo utilizar un número alto de valores históricos, entonces probablemente el resultado sea lo suficientemente bueno y menos valores de muestreo sean necesarios para el filtro de dilatación. Aún si la convergencia es alta, se debe muestrear una mínima cantidad de valores para evitar artefactos introducidos por el filtrado bilineal.

(a)

(b)

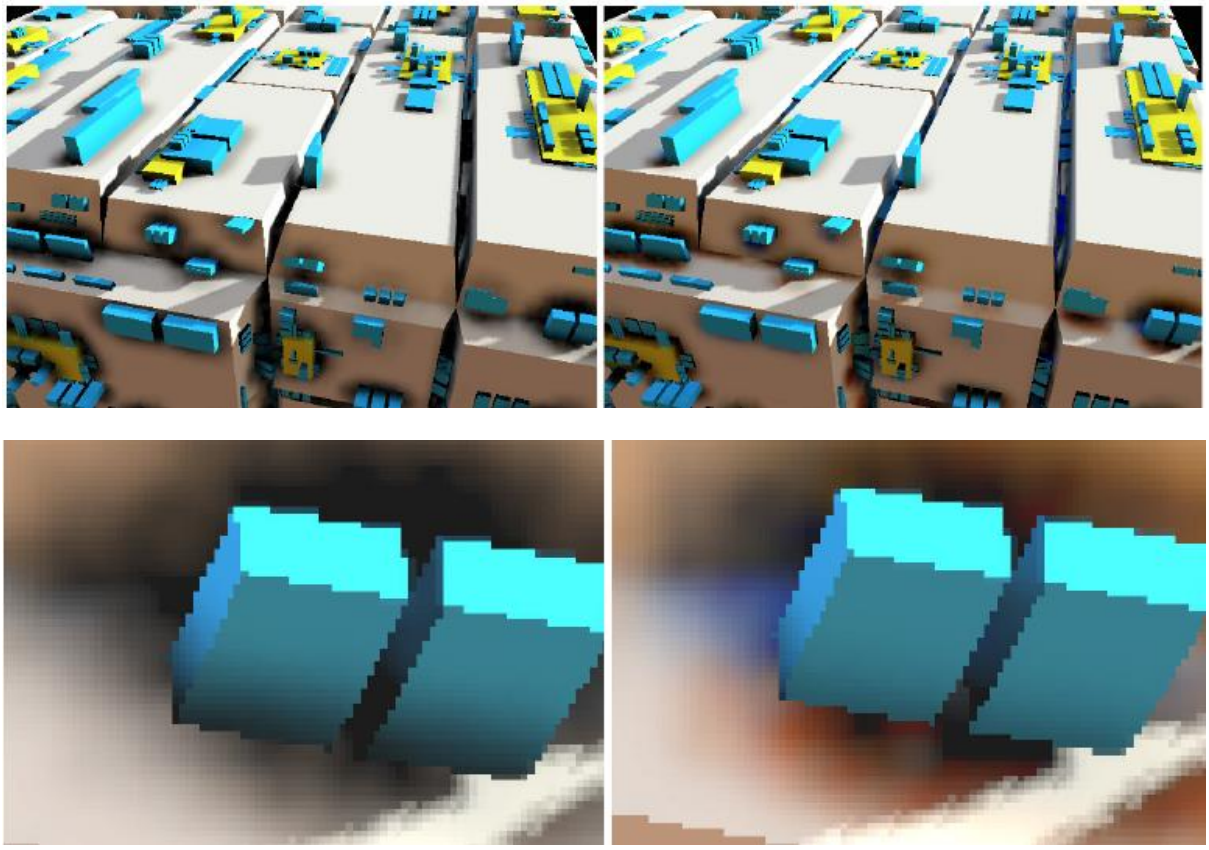
(c)

Figura 6-18 (a) Imagen Original. (b) Clasificación de objetos estables (blanco) e inestables (negro) (c) Filtro de dilatación para abarcar el área de influencia de los objetos inestables. (Bavoil & Andersson, Stable SSAO in Battlefield 3 with Selective Temporal Filtering, 2012)

6.3 Oclusión Direccional

La oclusión ambiental utiliza solo la geometría de la escena como información de entrada, ignorando toda información direccional de la luz incidente. La oclusión direccional extiende la idea original de oclusión ambiental incorporando la información direccional de la luz incidente. De forma simplificada, el

algoritmo consiste en acumular, para cada punto p, la luz ambiental proveniente de las direcciones en que ésta no se ocluye. Al igual que con la oclusión ambiental, una de las técnicas más atractivas es la que realiza sus cálculos en el espacio de la pantalla; ésta recibe el nombre de oclusión direccional en espacio de pantalla (*Screen-Space Directional Occlusion* o simplemente SSDO) (Grosch & Ritschel, 2010) (Figura 6-19).



(a) (b)

Figura 6-19 (a) Oclusión ambiental en espacio de pantalla (b) Oclusión direccional en espacio de pantalla (Ritschel, Grosch, & Seidel, Approximating Dynamic Global Illumination in Image Space, 2009).

En la oclusión direccional también se trazan rayos desde el punto P, distribuidos uniformemente. Se puede seleccionar cualquiera de los métodos expuestos en la sección anterior para distribuir y trazar los rayos, pero el cálculo de, por ejemplo, los ángulos horizonte no tiene sentido en este método. Para cada punto que no es oclusor del punto P se muestrea un mapa ambiental en la dirección del rayo, asumiendo un ángulo sólido de tamaño apropiado y teniendo en cuenta el ángulo entre la normal del punto P y la dirección de muestreo. El tamaño del ángulo sólido establece los parámetros del filtro de difuminado utilizado como operador de convolución del mapa ambiental. Si el rayo se topa con un oclusor entonces simplemente se ignora y se continúa con el próximo rayo. Los valores obtenidos se suman y se obtiene el aporte lumínico ambiental sobre el punto (Figura 6-20).

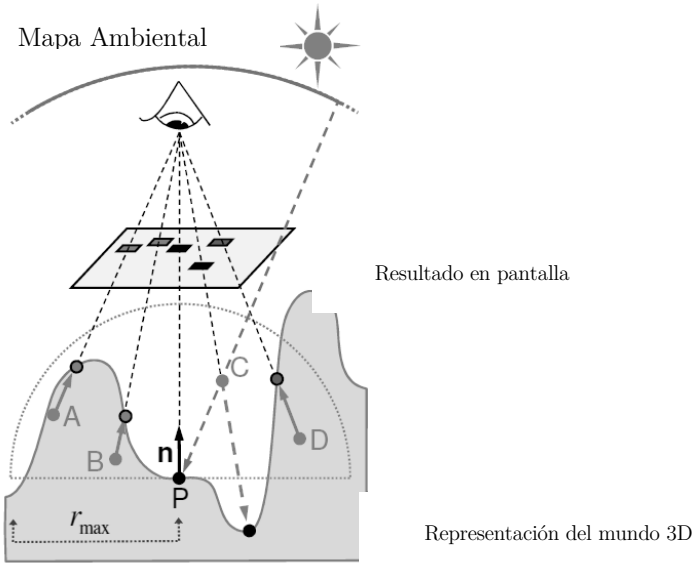


Figura 6-20 Funcionamiento del algoritmo de oclusión direccional en espacio de pantalla. Se muestrean 4 rayos, pero solo el punto C está por sobre la superficie y por lo tanto es la única dirección en la que se muestrea el mapa ambiental (Ritschel, Grosch, & Seidel, Approximating Dynamic Global Illumination in Image Space, 2009).

Este algoritmo puede ser extendido para incluir una aproximación del primer rebote indirecto de luz si, en vez de ignorar los rayos que colisionan con un oclusor, se obtiene la contribución de luz de estos oclusores. En la Figura 6-21 se puede ver una comparación entre la versión simple y la versión que incluye una aproximación al primer rebote indirecto de la luz.

Para calcular esta contribución se puede utilizar la siguiente fórmula:

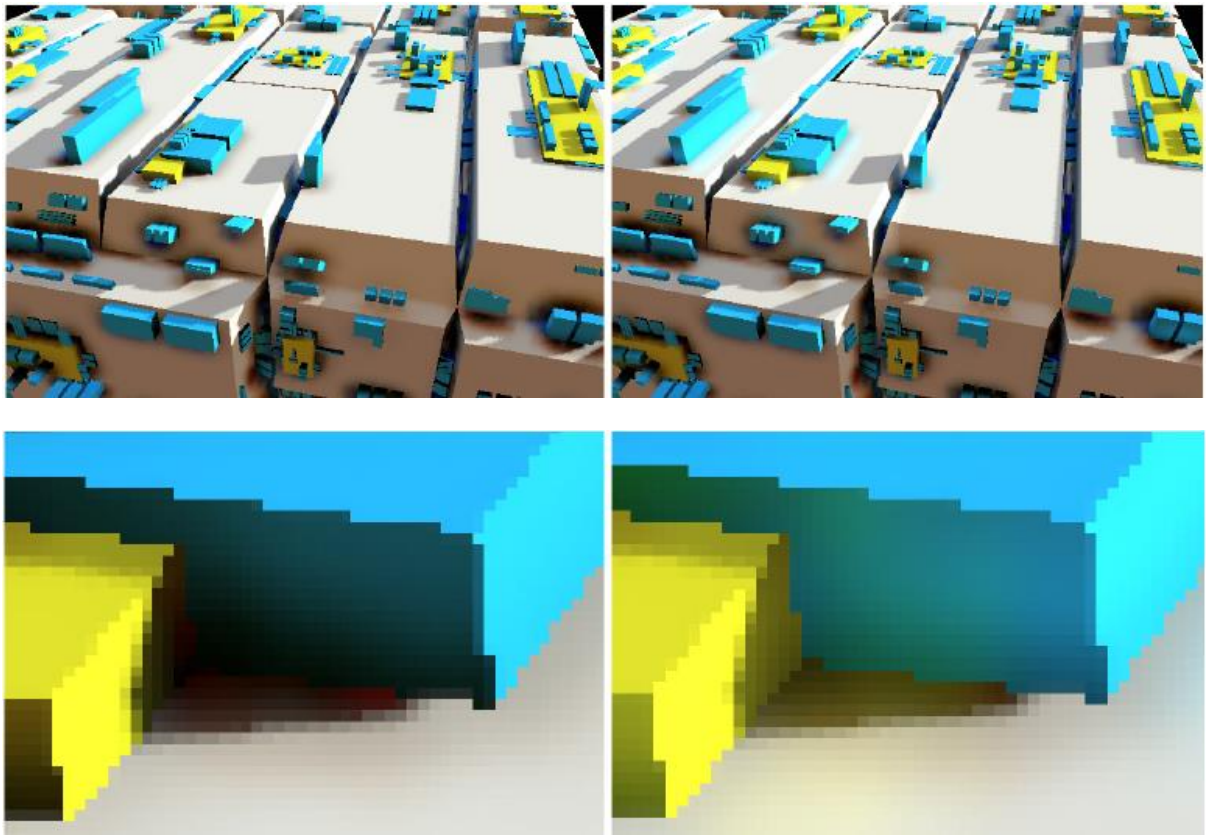
$$L_{ind}(P) = Albedo(\omega_i) \cdot \frac{A_s \cdot \cos\theta_s \cdot \cos\theta_r}{\pi \cdot d_i^2} \qquad \text{(Ecuación 6-9)}$$

donde $Albedo(\omega_i)$ es el color albedo de la superficie, ya sea proveniente de un color uniforme, de una textura o de un algoritmo procedimental; θ_s y θ_r son los ángulos entre la dirección del rayo y la normal del punto P y la normal de la superficie oclusora respectivamente; d_i es la longitud del rayo y A_s es el área de la superficie oclusora.

Debido a que obtenemos la información de un único pixel, no conocemos la forma de la superficie oclusora, por lo que la estimamos de la siguiente manera:

$$A_s = {}^{\pi \cdot R_{max}^2}/_{N} \qquad \text{(Ecuación 6-10)}$$

donde N es el número de rayos emitidos.



(a) (b)

Figura 6-21 (a) Oclusión direccional (b) Oclusión direccional + un rebote de luz (Ritschel, Grosch, & Seidel, Approximating Dynamic Global Illumination in Image Space, 2009).

Otra técnica de oclusión direccional en el espacio de la pantalla que extiende el algoritmo básico de SSAO es *Screen-Space Bent Cones* (Klehm, Ritschel, Eisemann, & Seidel, 2012). Éste se apoya en el concepto de *bent-normals,* introducido en (Landis, 2002), que son normales modificadas inclinadas en la dirección estimada que está menos ocluída, es decir, la dirección promedio de las direcciones no bloqueadas (Figura 6-22). Este cálculo se realiza simplemente sumando los vectores normalizados de dirección de los rayos que no interceptan un oclusor y dividiendo el resultado por la cantidad de estos rayos. Para realizar la distribución y trazado de rayos se puede recurrir a las mismas técnicas que las utilizadas en SSAO.

Además se introduce el concepto de *bent-cones* que son *bent-normals* aumentadas por un ángulo que abarca aproximadamente la zona no ocluida (Figura 6-22). Éstos se usan para limitar las direcciones en las que se recolecta información de la luz. Para calcularlos se utiliza un enfoque similar al cálculo de varianza de distribuciones *von Mises-Fisher* sobre esferas (Mardia & Jupp, 2009). En este caso, la varianza se calcula en el hemisferio en vez de en la esfera completa y se aproxima utilizando la longitud de las *bent-normals* no normalizadas. De esta manera el ángulo se estima utilizando:

$$C(P) \coloneqq (1 - \max(0{,}2|N_{ss}(P)| - 1))\frac{\pi}{2} \qquad \text{(Ecuación 6-11)}$$

donde $N_{ss}(P)$ es la *bent-normal* en el punto P.

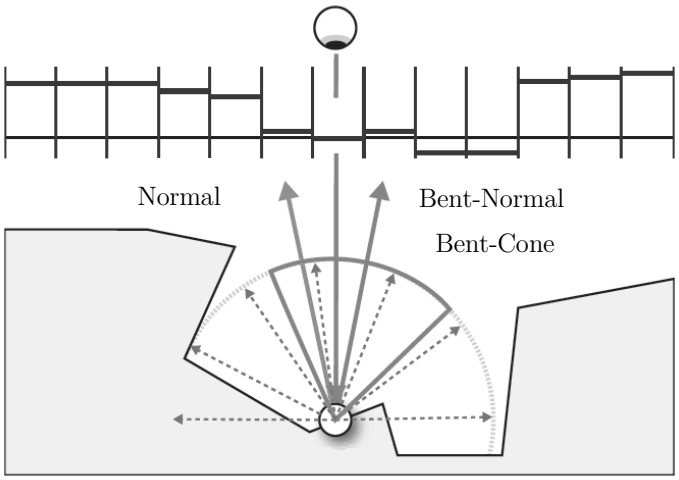


Figura 6-22 ***Bent-normals*** **y** ***bent-cones*** **(Klehm, Ritschel, Eisemann, & Seidel, 2012).**

Inicialmente, el término ambiental se puede calcular utilizando una ecuación similar a (Ecuación 6-6), pero muestreando el mapa ambiental con la *bent-normal* en vez de con la normal de superficie, por lo que:

$$K_a(\mathrm{P}) = AO(\mathrm{P}) \cdot L_a(N_{ss}(P)) \qquad \text{(Ecuación 6-12)}$$

De esta forma se obtiene información de iluminación direccional realizando un solo muestro al mapa ambiental. Si bien este algoritmo incrementa la calidad de los resultados con prácticamente el mismo costo computacional que el requerido para oclusión ambiental, no se tiene en cuenta el tamaño del área no ocluida. Sin embargo, es posible utilizar el *bent-cone* como una aproximación del tamaño de esta área. Para poder utilizarlo se debe crear un pequeño conjunto de mapas ambientales cada uno convolucionado para un ángulo sólido arbitrario (Figura 6-23). La cantidad de mapas ambientales no debe ser alta debido a los requerimientos de memoria y a la complejidad del *shader* subyacente.

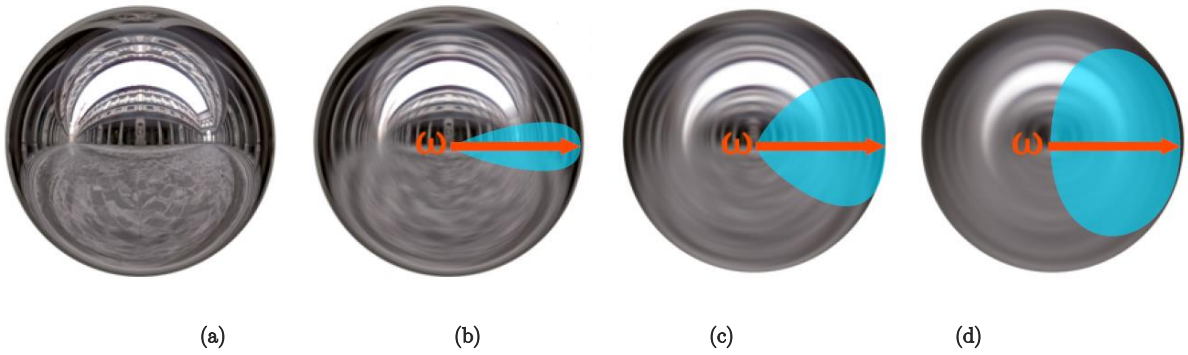


Figura 6-23 (a) Mapa de Entorno (b, c, d) Mapas ambientales convolucionados con un ángulo de 10, 45 y 90 grados respectivamente (Klehm, Ritschel, Eisemann, & Seidel, 2012).

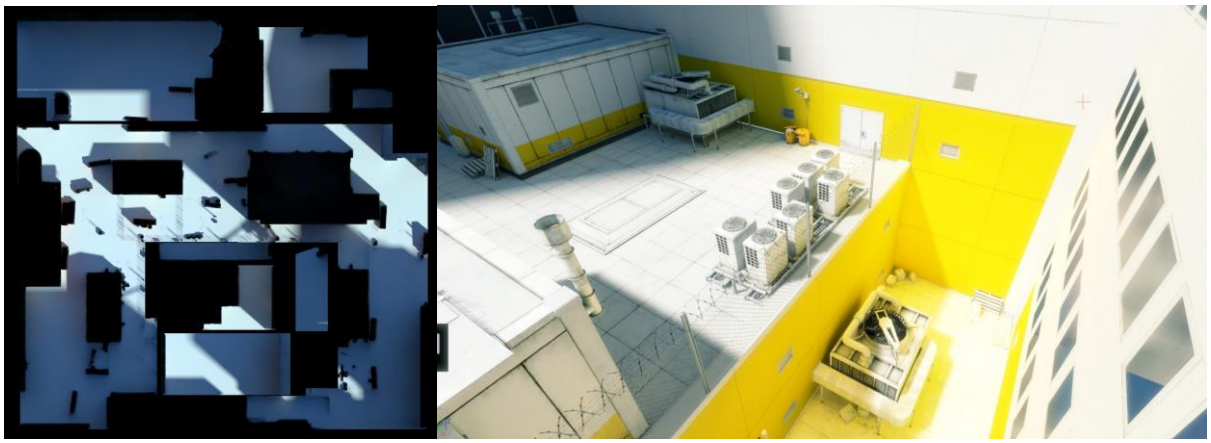
De esta manera el cálculo del término ambiental queda definido como:

$$K_a(\mathrm{P}) = AO(\mathrm{P}) \cdot L_{ac}(N_{ss}(P), \mathrm{C}(P))(1 - \cos(\mathrm{C}(\mathrm{P})))^{-1} \quad \text{(Ecuación 6-13)}$$

donde L_{ac} es la función que retorna el valor de luz ambiental en una dirección dada utilizando el mapa ambiental convolucionado con el ángulo sólido más cercano al *bent-cone* $\mathrm{C}(P)$ y donde $(1 - \cos(\mathrm{C}(\mathrm{P})))^{-1}$ es un término de normalización para compensar el hecho de que el *bent-cone* es simplemente una aproximación del área no ocluida. Este término de normalización puede ser combinado dentro de los mapas ambientales para reducir el costo computacional de la ecuación.

6.4 Mapas de Luces

Un mapa de luces (*lightmap*) es una textura que almacena información pre-calculada de la iluminación de la escena, lo que permite reducir cálculos complejos de iluminación (radiosidad, sombras suaves, etc.) a un simple acceso a textura (Figura 6-24). Aunque atractiva por sus resultados, esta técnica tiene las desventajas de estar limitada a iluminación estática sobre objetos estáticos (aunque puede ser combinada con técnicas de iluminación dinámicas), de tener altos requerimientos de memoria (en especial si se requiere almacenar información de iluminación de alto detalle), de requerir un canal adicional de coordenadas de texturas creado específicamente para este mapa y de necesitar tareas complejas de preparación y procesamiento de la escena en la etapa de producción.



(a) (b)

Figura 6-24 (a) Mapa de luces (b) Iluminación producida con un mapa de luces (Magnusson, 2011).

La resolución de un mapa de luces no es una medida adecuada para medir la precisión de los mismos debido a que no tiene en cuenta el tamaño ni la complejidad poligonal de la escena. En cambio, se utilizan como escala los lumens (*lumination elements*) por unidad de mundo, es decir, la cantidad de texels que afectan una superficie unitaria correspondiente a la escena. Cuantos más lumens por unidad de mundo se

tenga mejor será la calidad, pero la resolución del mapa podría ser prohibitiva en términos de velocidad de acceso y espacio en memoria.

Típicamente un mapa de luces solo es efectivo almacenando información de bajo detalle, sin embargo, es posible combinar los mapas de normales con mapas de luces especiales que capturan la luz incidente desde múltiples direcciones con el objetivo de reproducir iluminación de alto detalle. Esta técnica es conocida como *Radiosity Normal Mapping* (McTaggart, 2004; Mitchell, McTaggart, & Green, 2006) *o Directional Light Map.*

Por cada pixel de la superficie se transforma una base al espacio tangencial[6] del pixel, donde los vectores de la base se definen como:

$$B_0 = \left[-\frac{1}{\sqrt{6}}, -\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{3}}\right] B_1 = \left[-\frac{1}{\sqrt{6}}, \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{3}}\right] B_2 = \left[\frac{2}{\sqrt{3}}, 0, \frac{1}{\sqrt{3}}\right] \qquad \text{(Ecuación 6-14)}$$

Estos vectores son ortogonales entre sí y cubren de manera equitativa el hemisferio centrado en la normal. Con estos vectores se calculan tres mapas de luces cada uno representando la luz incidente en dirección contraria a la de los vectores de la base. La composición de los resultados se realiza modulando cada mapa de luces con la información de normales del mapa de normales (Figura 6-26).

A continuación se muestra el código fuente HLSL del algoritmo original de *Radiosity Normal Mapping* implementado por Valve para su motor Source (Green C. , Efficient Self-Shadowed Radiosity Normal Mapping, 2007):

```
float3 dp;
dp.x = saturate(dot(normal, bumpBasis[0]));
dp.y = saturate(dot(normal, bumpBasis[1]));
dp.z = saturate(dot(normal, bumpBasis[2]));
dp *= dp;
float sum = dot(dp, float3(1.0f, 1.0f, 1.0f));
float3 diffuseLighting = dp.x * lightmapColor1 + dp.y * lightmapColor2 + dp.z * lightmapColor3;
diffuseLighting /= sum;
```

donde el arreglo *bumpBasis* es el conjunto de vectores de la base (Ecuación 6-14); *normal* corresponde al valor almacenado en el mapa de normales y *lightmapColor#* es el valor almacenado en el mapa de luces correspondiente a cada vector base. No es necesaria una conversión de espacio debido a que todos los valores se encuentran en el espacio tangencial del punto de superficie.

[6] El espacio tangencial es el espacio de coordenadas definido sobre la normal de manera que el eje normal ($\vec{N}$) se corresponde con la normal, el eje tangente ($\vec{T}$) apunta en la dirección en la que la coordenada de textura u aumenta y el eje binormal ($\vec{B}$) apunta en la dirección en la que coordenada de textura v aumenta.

Es posible optimizar este algoritmo almacenando, en vez del mapa de normales, la variable *dp* del algoritmo anterior, es decir, el valor de influencia de cada mapa de luces direccional (Green C. , Efficient Self-Shadowed Radiosity Normal Mapping, 2007). Con esto se transforma el mapa de normales a una nueva base y se reducen sustancialmente los cálculos matemáticos realizados en el programa de fragmentos. Sin embargo, la información de normales no estará disponible para su uso en iluminación dinámica. Si la iluminación dinámica de la escena no depende fuertemente de los mapas de normales, se pueden utilizar los vectores de la base (Ecuación 6-14) para reconstruir la normal cuando se lo requiera.

Almacenar directamente la influencia de los mapas de luces direccionales tiene un conjunto de ventajas, entre las que se incluyen:

- No se requiere un procesamiento especial cuando se realiza filtrado o mip-mapping dado que se almacenan valores positivos.
- Se incrementa la precisión numérica dado que no se necesitan almacenar números negativos y que se almacena la información en el mismo formato que el requerido por el *shader*.
- El formato puede ser leído por herramientas de edición de imágenes sin necesidad de realizar ninguna interpretación adicional, por lo que es posible aplicar filtros directamente sobre la imagen.
- Se percibirán menos artefactos de *aliasing* cuando la textura sea minimizada.

Esta representación también permite integrar otro tipo de información en la misma textura. Por ejemplo, es posible almacenar un término de oclusión ambiental multiplicándolo directamente con los tres canales de esta textura. Alternativamente, se pueden calcular y almacenar tres términos de oclusión ambiental por cada vector dirección para simular un efecto de oclusión ambiental direccional (Figura 6-25). De esta manera se puede derivar una mayor cantidad de información de iluminación de alto detalle de un conjunto de mapas de luces de bajo detalle, pero esta información extra no puede ajustarse a la interacción entre objetos dinámicos.

(a) (b) (c)

Figura 6-25 (a) Mapa de Normales (b) Mapa de Normales almacenado en la nueva base (c) Inclusión de la información de oclusión ambiental direccional (Green C. , Efficient Self-Shadowed Radiosity Normal Mapping, 2007).

(a)

(b)

(c)

(d)

Figura 6-26 (a, b, c) Contribución ambiental de cada vector de la base (d) Composición de los resultados (McTaggart, 2004).

Uno de los problemas más importantes de los mapas de luces es su alto requerimiento de espacio de almacenamiento en la memoria de la GPU. id Software (pionera en implementar mapas de luces en tiempo real) ha desarrollado, a través de su existencia, un conjunto de técnicas exitosas para reducir el consumo de memoria de GPU de sus mapas de luces. Entre éstas se encuentra la técnica *Surface Caching* (Abrash, 1996) y más recientemente, la técnica *MegaTexture*, también conocida como Sparse Virtual Textures o Virtual Texture Mapping (van Waveren, 2009; Barret, 2008; Chajdas, Eisenacher, Stamminger, & Lefebvre, 2010). En esta última técnica se tiene una única gran textura que representa el mapa de luces completo de la escena y que podría ser tan grande como se lo requiera; algunos mapas de luces del juego Rage de id Software, por ejemplo, alcanzan los 64 GB (Obert, van Waveren, & Sellers, 2012) (Figura 6-27). En memoria de GPU sin embargo, solo se mantiene la información necesaria en el futuro inmediato y en el nivel de detalle apropiado, por lo que la información de objetos alejados o parches del terreno lejanos se cargan utilizando niveles de *Mipmap* de menor precisión. Esta información se agrupa en una textura del tipo atlas de mucho menor tamaño, que está organizada en mosaicos de un tamaño apropiado que idealmente maximiza la eficiencia del hardware utilizado. Para poder acceder a esta información se necesita una pequeña textura o tabla de indirección que mapea las coordenadas de textura originales a las coordenadas de la textura cargadas en memoria de GPU.

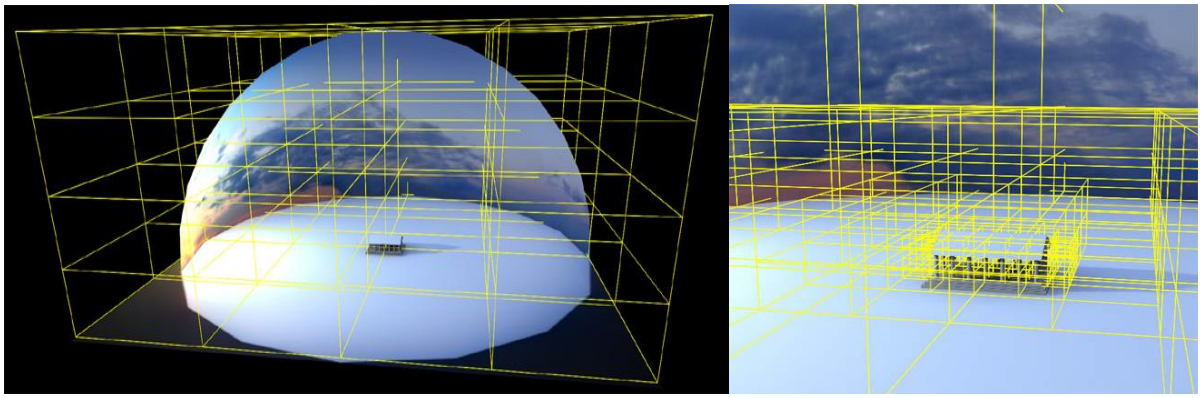
Figura 6-27 Captura del juego Rage de id Software. En el terreno se utilizan mapas de luces de gran tamaño administrados utilizando la técnica MegaTexture.

En cada cuadro de renderizado se determinan los mosaicos visibles desde el punto de vista actual. Esto se puede realizar renderizando la escena con un *shader* especial que señala los mosaicos visibles (esta pasada extra podría servir además para detectar oclusión). Con el objetivo de optimizar el proceso, este renderizado se puede realizar en baja resolución. Además, para capturar mosaicos próximos, el campo de visión se puede incrementar. Luego, esta información se procesa por el CPU, se realizan las tareas de *streaming*, la carga de texturas y la actualización de los índices. Para mantener el desempeño estable se carga solo un número arbitrario de mosaicos por cuadro. Si un mosaico no pudo ser cargado en memoria, la

tabla de indirección se ajusta para redireccionar esas coordenadas de textura al mosaico más cercano (probablemente un nivel de *Mipmap* inferior). Es posible que la calidad visual se reduzca, pero se puede evitar una degradación amplia de la calidad si se cargan primero los mosaicos correspondientes a niveles *Mipmaps* de menor precisión debido a que éstos suelen contener los mosaicos con mayor precisión de los sectores visibles.

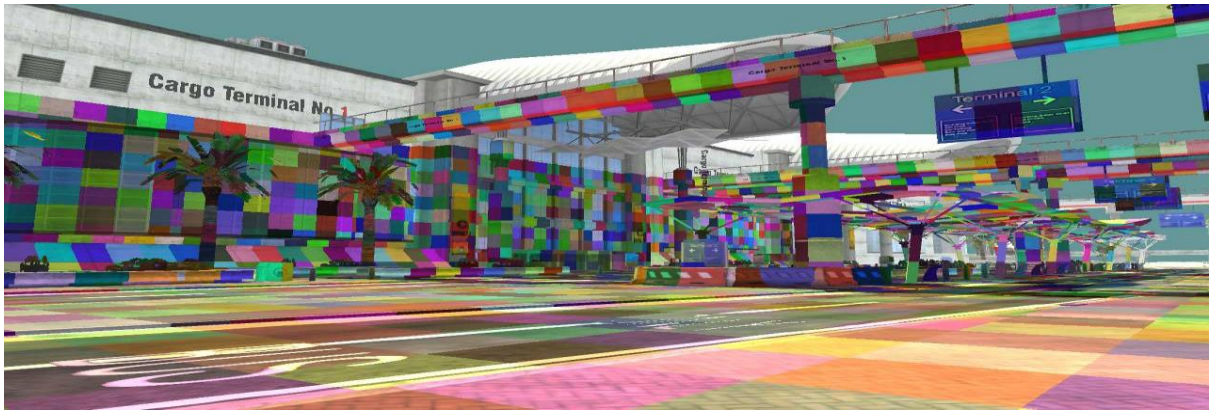
6.5 Volúmenes irradiantes

En (Greger, Shirley, Hubbard, & Greenberg, 1998) se propuso la técnica de volúmenes irradiantes (*irradiance volumes*) que aproxima la distribución de radiancia estática de la escena con un alto nivel de granularidad. La técnica consiste en dividir la escena en un conjunto de *voxels*, o volúmenes irradiantes, en el que en cada uno de sus vértices se calcula una aproximación de la distribución de radiancia en ese punto y que típicamente se almacena utilizando armónicos esféricos. Los voxels pueden estar distribuidos uniformemente, pero el costo de almacenamiento podría resultar muy alto. Alternativamente, es posible utilizar una estructura *octree* en la que se calculan y subdividen las áreas que necesitan mayor granularidad, es decir, los voxels en los que las magnitudes de irradiancia cambian significativamente (Figura 6-28). Debido a que la distribución de radiancia típicamente se calcula solo una vez en la etapa de producción, se podría utilizar un método de fuerza bruta que comienza con una estructura altamente subdivida y colapsa los voxels cuando la irradiancia que éstos reciben es similar según un umbral arbitrario. Este método no es perfecto porque se asume el nivel máximo de subdivisión necesario. Por lo tanto, se suele emplear una heurística que detecta voxels que pueden necesitar una mayor subdivisión. Por ejemplo, se podrían considerar áreas con una gran cantidad de vértices como posibles candidatos. En (Tatarchuk, 2005) se presenta un conjunto de heurísticas para la subdivisión de estos voxels.



(a) (b)

Figura 6-28 (a) Subdivisión Uniforme. (b) Subdivisión adaptativa utilizando un octree y heurísticas de subdivisión (Tatarchuk, 2005).



(a)

(b)

(c)

Figura 6-29 (a) Cada color denota el área de influencia de cada volumen irradiante (b) Iluminación calculada utilizando solamente volúmenes irradiantes (c) Escena iluminada utilizando iluminación directa junto con volumes irradiantes (Moore & Jefferies, 2009).

Cuando se renderiza, por cada punto de superficie evaluado se accede a la información de radiancia almacenada en los vértices del voxel en el que se encuentra el punto y se calcula una aproximación de la irradiancia recibida interpolando los valores correspondientes según su distancia al punto y a la orientación de su normal. Es posible que surjan artefactos cuando un objeto dinámico se mueve hacia voxels con

distinto nivel de subdivisión; en (Tatarchuk, 2005) se propone extender la estructura y muestrear también (si se requiere) los vértices de *voxels* de menor nivel de subdivisión. Además, se aconseja no utilizar una interpolación lineal de armónicos esféricos, sino utilizar la técnica de gradientes de armónicos esféricos propuesta en (Annen, Kautz, Durand, & Seidel, 2004) que permite reducir el costo computacional de la operación.

En un pipeline *deferred renderer* se hace dificultoso cargar la información de radiancia por objeto debido a que la iluminación no se calcula sobre los objetos sino sobre toda la escena tomando como entrada la información almacenada en el *G-Buffer*. Una posibilidad es ampliar el *G-Buffer* y almacenar la información de radiancia directamente en éste; sin embargo, de esta manera se aumenta el tamaño del *G-Buffer*, se hacen más complejos los *shaders* de esta etapa y se calcula información de iluminación sobre todos los objetos dentro del frustum, incluso los ocluidos, reduciendo así una de las ventajas más importantes de los pipelines *deferred renderer*. Para atenuarlo, se podría realizar una pasada de profundidad previa que procese los vértices de los objetos y solo escriba el mapa de profundidad, que luego será utilizado para descartar rápidamente los objetos ocluidos cuando éstos sean procesados en el *G-Buffer*.

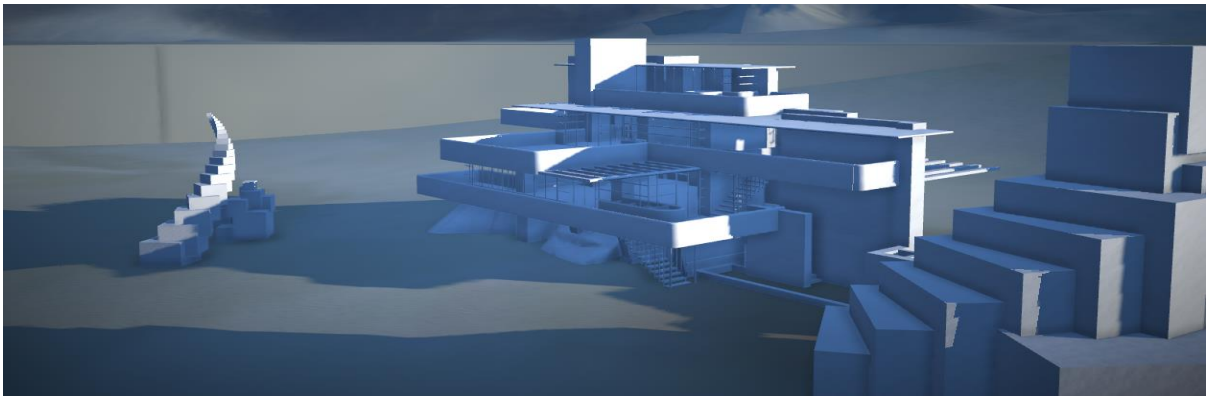
En (Moore & Jefferies, 2009) se utiliza una estructura *octree* y se subdivide la escena con el algoritmo de fuerza bruta agregando una heurística para subdividir voxels en áreas donde haya mucho detalle geométrico. Debido a que el pipeline de renderizado utilizado es diferido, por cada cuadro renderizado se crea una textura volumétrica con la información de radiancia que influye en el área que cubre el frustum de la cámara. La información es distribuida espacialmente (por lo que puede ser accedida utilizando el mapa de profundidad) y puede utilizarse una distribución no lineal con el objetivo de obtener mayor granularidad en las áreas cercanas al punto de visión. En esencia, se plantea utilizar una técnica de *streaming* sobre la información de radiancia de la escena similar a la utilizada en *MegaTexture* (sección 6.4) (Figura 6-29).

El método de volumes irradiantes es similar al de mapas de luces, pero se reduce el espacio utilizado en memoria de CPU y de GPU, manteniendo un alto volumen de información de iluminación global y es aplicable tanto a objetos estáticos como dinámicos; sin embargo, no se tienen en cuenta las repercusiones que los objetos dinámicos producen sobre la iluminación de la escena.

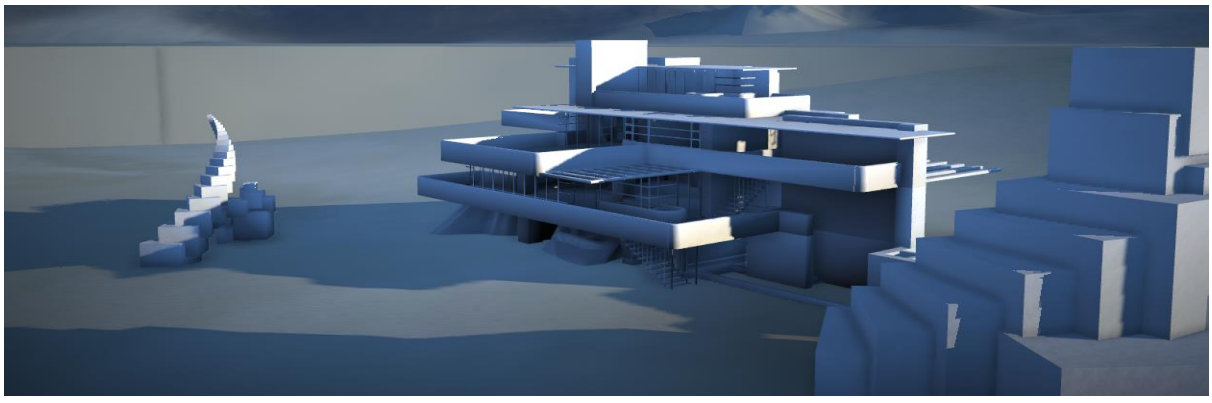
6.6 Precomputed Radiance Transfer

La técnica *Precomputed Radiance Transfer* (PRT) fue propuesta en (Sloan, Kautz, & Snyder, Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments, 2002) y al igual que la de volúmenes irradiantes ha logrado una alta aceptación por su costo computacional y la calidad visual de los resultados, a expensas de un cálculo preciso y adaptable al dinamismo de la escena.

Figura 6-30 Resultado de aplicar *Precomputed Radiance Transfer*. Fenómenos como occlusion ambiental, interreflexiones (especialmente notorios en la estructura metálica) y sombras suaves pueden ser modelados con bajo costo computacional utilizando esta técnica (Chen & Liu, Lighting and material of Halo 3, 2008).



(a)



(b)

Figura 6-31 (a) Iluminación directa y luz ambiental representada con armónicos esféricos (b) iluminación directa y *Precomputed Radiance Transfer* utilizando el mismo mapa ambiental. En el edificio central de la segunda figura se puede apreciar la influencia de los objetos de la escena en la iluminación resultante, incluyendo un efecto similar a oclusión ambiental (Chen & Tatarchuk, Lighting Research at Bungie, 2009).

En tiempo de producción, algoritmos computacionalmente costosos calculan las funciones de transferencia de la luz sobre los vértices de los objetos de la escena; esto permitirá que en tiempo de ejecución se mapee una radiancia incidente (solo conocida en tiempo de ejecución) a una radiancia de salida. Con estas funciones de transferencia se permite modelar un conjunto de fenómenos de la luz computacionalmente muy costosos, entre los que se incluyen *subsurface scattering*, interreflexiones, sombras suaves y oclusión ambiental (Figura 6-30, Figura 6-31).

Está técnica tiene restricciones significativas:

- Los objetos se asumen rígidos (no se ajusta a escenas con animaciones rígidas y *skinning*).
- A menos que los objetos se muevan juntos, las interacciones entre ellos no se adaptan correctamente.
- Solo se modelan fenómenos de bajo detalle.

Sin embargo, se han realizado investigaciones para solucionar o atenuar estas restricciones (Sloan, Luna, & Snyder, Local, Deformable Precomputed Radiance Transfer, 2005; Sloan, Luna, & Snyder, Local, Deformable Precomputed Radiance Transfer, 2005; Ng, Ramamoorthi, & Hanrahan, 2003). Una limitación importante que no puede ser sorteada es el hecho de que esta técnica solo tiene en cuenta la interacciones por consecuencia directa o indirecta de luces ambientales, es decir, infinitamente distantes; por esto no es posible, por ejemplo, representar correctamente sombras provenientes de lámparas, al menos que se asuma de antemano su presencia y su posición no sea alterada en tiempo de ejecución.

Las funciones de transferencia son funciones esféricas por lo que suelen ser almacenadas utilizando armónicos esféricos. Además, las propiedades de los armónicos esféricos permiten que la aplicación de las funciones de transferencia sobre la radiancia incidente se realicen por medio de un conjunto de operaciones simples (Sloan, Kautz, & Snyder, Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments, 2002). Debido a que es necesario almacenar estas funciones por vértice, ha sido realizado un gran esfuerzo para comprimir los coeficientes de los armónicos esféricos (Ko, Ko, & Zwicker, 2009).

Precomputed Radiance Transfer provee un método para representar interacciones complejas de la luz y BRDFs sobre escenas estáticas, permitiendo la utilización de luces ambientales dinámicas.

6.7 Radiosidad Instantánea

Lograr una aproximación de iluminación global completamente dinámica que se ejecute eficientemente en tiempo real es un tópico de gran interés en la actualidad. El método propuesto por Bunnell (Bunnell, 2005) que se analizó en la sección 6.2.1 puede ser extendido para incorporar una simulación sencilla de iluminación indirecta completamente dinámica. Éste y otros métodos propuestos que utilizan conceptos similares (Sloan, Govindaraju, Nowrouzezahrai, & Snyder, 2007; Guerrero, Jeschke, & Wimmer, 2008; Dachsbacher, Stamminger, Drettakis, & Durand, 2007; Evans A. , 2006) pueden ser utilizados como una aproximación sencilla del algoritmo de radiosidad, pero desafortunadamente, son ignorados debido a la

complejidad de implementación, requerimientos de memoria y desempeño (Kaplanyan A. , Light Propagation Volumes in CryEngine 3, 2009).

También se han realizado implementaciones (Shishkovtsov, 2005; McGuire & Luebke, 2009; Stamate, 2008) que simplifican el algoritmo de *Photon Mapping* introducido en la sección 4.5.6 y que utilizan la GPU para mejorar la eficiencia de alguna etapa del algoritmo. Sin embargo, estas técnicas todavía no han resultado ser viables en tiempo real, especialmente en escenas dinámicas, debido a que los algoritmos típicamente reúsan información de cuadros previos que es costosa de recalcular y que al momento de hacerlo (por ejemplo, cuando un objeto se mueve) se afecta notoriamente la distribución, idealmente uniforme, de los cuadros renderizados (Kaplanyan A. , Light Propagation Volumes in CryEngine 3, 2009).

La técnica *Screen-Space Directional Occlusion*, expuesta en la sección 6.3 es una técnica plausible, pero también tiene un conjunto de desventajas importantes debido a que no captura información fuera de la geometría visible, puede reproducir como máximo un solo rebote o interacción de luz, la información de iluminación de los vecinos del punto no incluye la influencia de la iluminación directa (aunque es posible extenderlo para lograrlo) y el radio de influencia típicamente es pequeño para reducir la penalidad de accesos a memoria. No obstante, esta técnica es atractiva debido a que realiza oclusión direccional y una aproximación simple de radiosidad de un solo rebote con un costo computacional relativamente bajo.

En (Keller, 1997) se introdujo el método de radiosidad instantánea (*instant radiosity*) para generar iluminación global dinámica potencialmente de tiempo real. Puntualmente se introdujo el concepto de luces puntuales virtuales (VPLs), también denominado *many lights*, que dio lugar a un conjunto de métodos similares (u optimizaciones del método anterior) que permiten generar iluminación global dinámica en tiempo real. El algoritmo simplificado consiste en trazar un conjunto de rayos desde cada luz de la escena y colocar, en los puntos de intersección con superficies, luces puntuales que simulan un rebote indirecto de esa luz sobre dichas superficies. Desde cada punto de intersección se pueden trazar nuevos rayos con el objetivo de considerar más de un rebote de luz. Al igual que una luz convencional, idealmente en cada VPL se debe calcular la información de oclusión desde ese punto, es decir, se debe calcular la sombra que produce (Figura 6-32). Este algoritmo es computacionalmente muy costoso debido, en parte, a que es necesario trazar rayos y detectar colisiones y debido a que también es necesario calcular sombras para un conjunto alto (al menos cientos) de luces puntuales. Sin embargo, este algoritmo sencillo tiene varias oportunidades de optimización, simplificación y también extensión. Además, es posible que en un futuro cercano este enfoque permita reproducir iluminación global dinámica de tiempo real con una calidad visual cuasi fotorrealista (Figura 6-33).

Entre las técnicas derivadas de radiosidad instantánea se destacan *Reflective Shadow Maps (RSM)*, *Incremental Instant Radiosity* e *Imperfect Shadow Maps* dado que permiten la ejecución en tiempo real de este algoritmo, pero con importantes limitaciones.

Reflective Shadow Maps (Dachsbacher & Stamminger, Reflective Shadow Maps, 2005) extiende la idea de mapas de sombra anexando a la información de profundidad, la información de normales, de albedo y las posiciones en espacio de mundo (Figura 6-34). De esta manera cada texel del *RSM* se convierte, en esencia, en una luz puntual virtual. No obstante, debido a la gran cantidad de puntos considerados es impráctico considerar más de un rebote de la luz. La ejecución de este esquema puede realizarse

completamente en GPU, lo que favorece al desempeño; sin embargo, el costo de crear cada *RSM* es alto. Además, se requiere acceder a un gran volumen de información por cada punto de superficie renderizado; sin embargo, es posible reducirlo utilizando técnicas de esténcil y otras optimizaciones relacionadas sobre un pipeline diferido, similares a las utilizadas en luces convencionales (Valient, The Rendering Technology of KillZone 2, 2009).

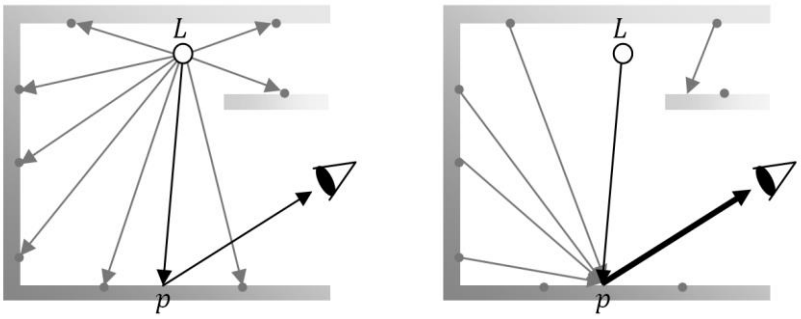
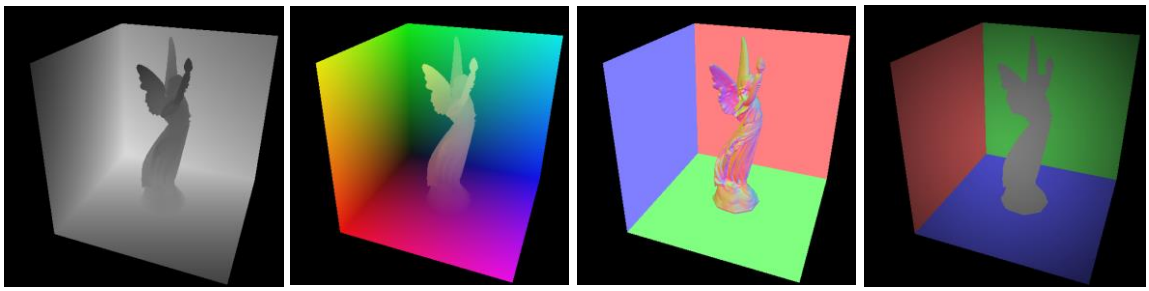


Figura 6-32 Aproximación de iluminación indirecta utilizando luces puntuales virtuales. (a) Se trazan rayos desde las luces de la escena y en los puntos de intersección se ubican luces puntuales (b) Al momento de renderizar el punto p se utilizan como fuentes de luz las luces puntuales ubicadas en el paso anterior más las luces directas, idealmente las luces puntuales virtuales deberían brindar información de oclusión para evitar que aporten luz incorrectamente (Preiner & Wimmer, 2010).

Figura 6-33 Escenas renderizadas utilizando técnicas que mejoran y optimizan los resultados del algoritmo de iluminación global dinámica de radiosidad instantánea. (a) Técnica *Bidirectional Lightcuts* (Walter, Khungurn, & Bala, 2012) (b) Técnica *Virtual Spherical Lights* (Hašan, Křivánek, Walter, & Bala, 2009). La obtención de estos resultados demora unos pocos minutos en hardware convencional contemporáneo (aproximadamente 30 y 4 minutos respectivamente), pero permiten modelar la interacción con superficies brillosas, entre otras consideraciones, con un costo computacional significativamente menor que los algoritmos introducidos en la sección 4.5.

Incremental Instant Radiosity (Laine, Saransaari, Kontkanen, Lehtinen, & Aila, 2007) reutiliza la información de los VPLs calculados en los cuadros previos actualizando solo un número de VPLs inválidos por cuadro, con el objetivo de mantener estable la distribución de cuadros por segundo. Uno de los aportes

más importantes de este trabajo, es la detección de VPLs inválidos cuando la iluminación directa cambia; sin embargo, este esquema solo funciona para geometría estática, lo que reduce su utilidad.



(a) (b) (c) (d)

Figura 6-34 Componentes de un Reflective Shadow Map. (a) Profundidad, (b) posiciones en espacio del mundo, (c) normales y (d) albedo (Dachsbacher & Stamminger, Reflective Shadow Maps, 2005)

Imperfect Shadow Maps (Ritschel, Grosch, Kim, Seidel, Dachsbacher, & Kautz, 2008) permite aproximar, con un costo computacional bajo, la información de oclusión de cada VPL utilizando para esto una representación de la escena basada en puntos. Este esquema funciona debido a que el aporte de cada VPL es pequeño, por lo que una aproximación muy sencilla de la información de oclusión típicamente es suficiente. Está técnica solo se enfoca en mejorar el tiempo de ejecución de la creación de los mapas de sombras de los VPLs, por lo que puede combinarse con *RSM* para generar la distribución de VPLs en la escena. Desafortunadamente, la precisión de cada mapa de sombras puede degradarse significativamente en escenas complejas y de gran tamaño; además, debido a la poca resolución y precisión de estos mapas, las sombras indirectas de contacto no suelen reproducirse adecuadamente.

6.8 Volúmenes de Propagación de la Luz

Los volúmenes de propagación de la luz (*light propagation volumes*) (Kaplanyan, Engel, & Dachsbacher, 2010; Kaplanyan A. , Light Propagation Volumes in CryEngine 3, 2009; Kaplanyan & Dachsbacher, Cascaded Light Propagation Volumes for Real-Time Indirect Illumination, 2010) es una técnica que se inspira en *radiosidad instantánea* y utiliza *Reflective Shadow Maps* para distribuir las luces puntuales virtuales. Esta técnica se ejecuta eficientemente en hardware contemporáneo, aún en consolas de actual generación, reproduciendo iluminación indirecta difusa producto de varios rebotes de luz (Figura 6-35).

El algoritmo puede dividirse en cuatro etapas (Figura 6-36). En la primera, se crean *RSM* por cada luz directa de la escena. Es posible aplicar técnicas desarrolladas para mapas de sombras convencionales; en escenas de exterior, por ejemplo, el sol suele modelarse como una luz direccional por lo que es posible utilizar la técnica de cascadas (Engel, Cascaded Shadow Maps, 2006) para mejorar la distribución de VPLs. También es posible utilizar técnicas como (Liao, 2010; Heidrich & Seidel, View-Independent

Environment Maps, 1998) para generar *RSM* para luces puntuales. Debido a la baja resolución de los *RSM*, es posible notar artefactos de *jitter* cuando la luz se mueve, pero pueden reducirse si éstas se mueven en unidades que coinciden con las dimensiones del texel del *RSM* en espacio del mundo.

Figura 6-35 *Light Propagation Volumes* en CryEngine 3. Notar los tonos verdes suaves reflejados en la pared (Kaplanyan A. , Light Propagation Volumes in CryEngine 3, 2009).

La segunda etapa consiste en inicializar, utilizando la información almacenada en cada *RSM*, una estructura de voxels que almacena la distribución de radiancia de la escena. La información de radiancia se representa utilizando armónicos esféricos y se almacena en la GPU utilizando una textura. Para acumular un nuevo valor sobre esta textura se utiliza simplemente *additive blending*, lo que simplifica el proceso. Esta estructura impone importantes limitaciones: la resolución de la grilla típicamente impide capturar detalles finos y podría provocar artefactos del tipo *light bleeding* y, debido a la utilización de armónicos esféricos y a la naturaleza del proceso de propagación, la única interacción de luz modelada es la difusa.

En la tercera etapa se ejecuta un algoritmo iterativo de propagación. Para esto se utiliza un algoritmo similar a (Evans K. F., 1998), con la diferencia principal de que en vez de transferir la energía a los 26 vecinos de la celda solo se propaga a 6 vecinos con el objetivo de mejorar el desempeño del algoritmo. Idealmente, el algoritmo se ejecuta hasta que la luz se propague por todo el volumen. Para poder incluir sombras indirectas es necesario expandir la estructura utilizada anexando una representación volumétrica (utilizando armónicos esféricos) de las superficies de la escena; además, con esta información, es posible calcular múltiples rebotes de la luz. Por último, la escena se ilumina utilizando esta información, de forma similar a *Irradiance Volumes*.

Este esquema puede ser extendido utilizando el concepto de cascadas, similar al utilizado en mapas de sombras, con el objetivo de obtener mayor precisión en áreas cercanas a la cámara; esta variación es conocida como *Cascaded Light Propagation Volumes*. Si bien eleva la calidad visual, introduce un

conjunto de complejidades en la etapa de creación de RSM y en la etapa de propagación. Además, aprovechando el hecho de que la iluminación es de bajo detalle y típicamente ésta varía lentamente, es posible utilizar un esquema que reutilice cálculos por varios cuadros. También, es posible extenderlo para incorporar interacciones especulares.

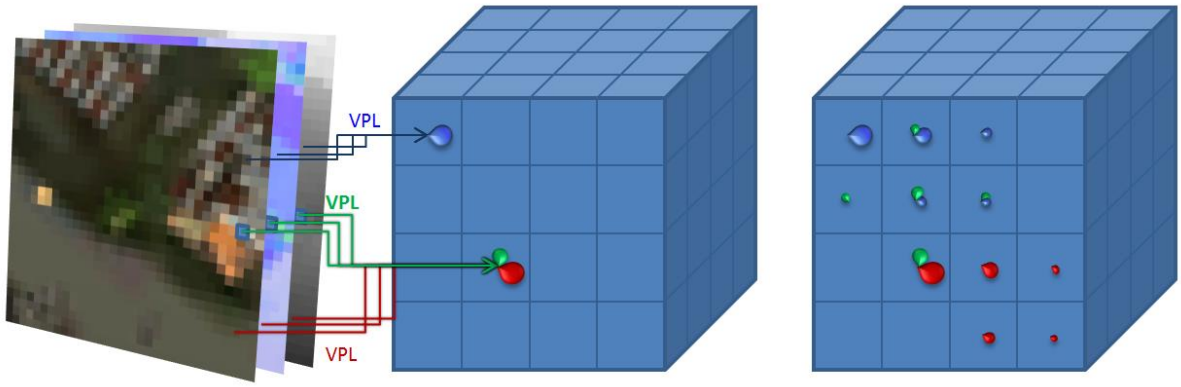


Figura 6-36 Esquema básico de *Light Propagation Volumes*. (a) El primer rebote de luz es capturado en la creación de los RSM de las luces de la escena; (b) una estructura con la distribución de radiancia de la escena es creada utilizando la información contenida en los RSM; (c) la luz se propaga utilizando un algoritmo de propagación iterativo. Por último, esta estructura se utiliza como parámetro de entrada para iluminar la escena (Kaplanyan, Engel, & Dachsbacher, 2010).

Capítulo 7 Pipelines Gráficos

Actualmente resulta impráctico realizar una implementación de tiempo real de cualquiera de los algoritmos de iluminación global introducidos en la (sección 4.5), ya sea porque requieren hardware dedicado todavía no disponible masivamente o porque las técnicas desarrolladas para ejecutarse eficientemente en las GPUs contemporáneas no son aún lo suficientemente rápidas para competir cara a cara con un pipeline de iluminación local extendido; esto se debe a que el hardware actual no se adecúa completamente a la tarea que debe desenvolver y a la falta de madurez de las implementaciones desarrolladas.

Teniendo en cuenta la estructura y las características generales del pipeline implementado en el hardware de las GPUs modernas, se han desarrollado un conjunto de pipeline gráficos de iluminación local, cada uno con ventajas y desventajas con respecto a las otras alternativas, que permiten renderizar gráficos fotorealistas eficientemente en tiempo real. Es necesario realizar desde el comienzo una selección adecuada de uno de estos debido a que podría afectar desde la selección de técnicas gráficas hasta la estructura de software que lo alimenta.

En este capítulo se describirá brevemente el pipeline general implementado en el hardware de las GPUs lo que permitirá entender las limitaciones y las posibilidades de acción en las implementaciones propuestas. Se realizará un relevamiento de los pipelines gráficos de iluminación local contemporáneos más importantes y utilizados que se ejecutan eficientemente bajo este hardware[7]. Se comenzará describiendo el modelo más simple, el *forward renderer* que se corresponde de forma natural con el pipeline implementado en hardware, para luego describir los actualmente populares *deferred renderers*, que modularizan el renderizado de la escena, realizando parte de los cálculos en espacio de pantalla. También se describirán esquemas de indexación de luces, que identifican qué luces afectan áreas en espacio de pantalla o mundo y que permiten utilizar *forward* y *deferred renderers* de forma más eficiente, pero introduciendo algunas complejidades adicionales.

7.1 Pipeline General de las GPUs

En la Figura 7-1 se muestra un diagrama esquemático simplificado de las etapas del pipeline implementado en el hardware de las GPUs modernas. El pipeline recibe como entrada un conjunto de vértices (posición, normal, coordenadas de texturas, etc.), la conectividad entre éstos, los valores globales y

[7] Todos los pipeline de renderizado se nombrarán utilizando su denominación en inglés. Esto se hará así para mantener uniformidad debido a que, si bien la traducción de algunos de ellos es clara, hay otros que se prestan a confusión.

las texturas y requiere el ajuste de un conjunto de estados. El resultado producido es luego almacenado en cero, una o más posiciones de un conjunto de *render targets*, es decir, en texturas o en el propio *frame buffer*.

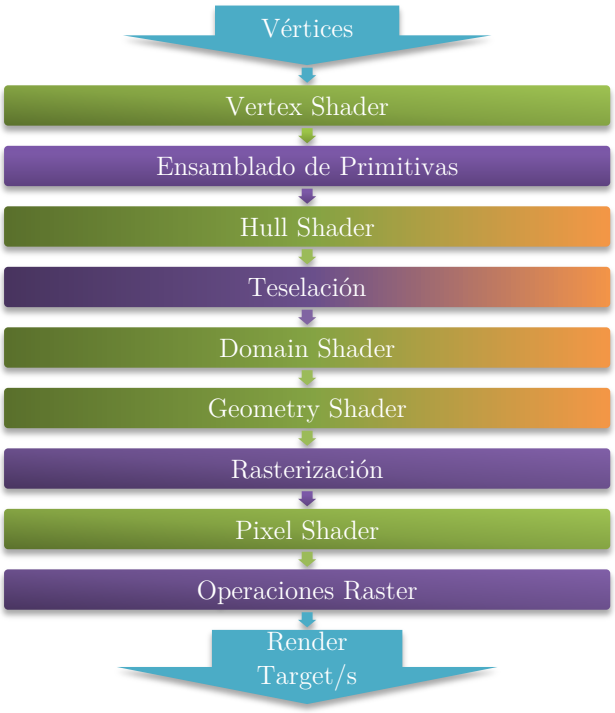


Figura 7-1 Diagrama esquemático simplificado de las etapas del pipeline de las GPUs modernas. Las etapas verdes son programables, las etapas violetas son de funcionalidad fija, y las etapas naranjas son opcionales. La teselación (*Hull Shader*, Teselación y *Domain Shader*) solo está disponible para GPUs con *shader model 5*, mientras que la etapa *Geometry Shader* está disponible para GPUs con soporte para *shader model 4* o superior.

Ciertas etapas son programables y ejecutan un programa suministrado por el usuario denominado *shader*, mientras que otras etapas son fijas y ejecutan un algoritmo implementado en hardware que solo varía su funcionalidad dependiendo del ajuste de un conjunto pequeño de estados. Actualmente se utiliza una arquitectura unificada en la que se utilizan las mismas unidades, denominadas *stream processors*, para ejecutar los distintos tipos de *shaders*; de esta manera se permite realizar un mejor aprovechamiento de las unidades disponibles y se simplifican las tareas de optimización. Todos los recursos internos de la GPU (registros, *stream processors*, etc.) son administrados transparentemente utilizando complejos planificadores; no obstante, existen ciertas limitaciones que podrían requerir la atención del desarrollador, como lo es la ejecución en lote y sincronizada de los *shaders* (Drone, 2007).

La primera etapa, *Vertex Shader* o programa de vértices, realiza una serie de operaciones definidas por el usuario sobre cada vértice de forma individual y produce un vértice transformado por cada uno de los vértices originales. En esta etapa se necesita producir como mínimo (o transformar desde la posición en espacio del objeto) la posición en espacio de *clipping* del vértice procesado. Luego, en la siguiente etapa, se

ensamblan los vértices en primitivas geométricas utilizando la topología suministrada, y se genera una representación de triángulos, puntos o líneas según la indicación dada al pipeline.

Las siguientes tres etapas (*Hull Shader*, Teselación, y *Domain Shader*) cubren la tarea de teselación, que es opcional y que está disponible de forma estándar a partir de las GPUs con soporte para *shader model 5*, aunque es una tecnología inicialmente implementada por AMD que se encuentra disponible desde las GPUs Xenos de Xbox 360 y las Radeon HD 2000, 3000 y 4000. Conceptualmente, esta etapa permite obtener una versión subdividida de las primitivas producidas en la etapa anterior, tarea que es controlada a través de dos *shaders*, el *hull shader* que opera sobre puntos de control y el *domain shader* que permite modificar la posición de los vértices luego de aplicar el algoritmo de teselación. Estas etapas permiten aplicar eficientemente técnicas como mapeo de desplazamiento, subdivisión poligonal y nivel de detalle dinámico; también reducen el espacio de almacenamiento requerido en la memoria de la GPU y mejoran el desempeño global al permitir aplicar cálculos costosos de procesamiento de vértices sobre una versión simplificada del objeto renderizado (Microsoft, 2012).

La etapa de *Geometry Shader* (programa geométrico) también es opcional y está disponible a partir de las GPUs con soporte para *shader model 4*. Ésta procesa cada primitiva, produciendo en cada ejecución cero, una o más primitivas. Alternativamente, es posible volcar el resultado a un *buffer* de vértices en memoria de GPU. Estos *shaders* permiten generar sombras cúbicas en una sola pasada, generar pelaje eficientemente, etc.

Luego se ejecuta la etapa de funcionalidad fija denominada habitualmente como Rasterización. Inicialmente se aplican técnicas de *clipping* y *culling* para reducir el número de polígonos a renderizar. Los polígonos que sobreviven se rasterizan, lo que convierte a las primitivas en un conjunto de fragmentos. En esta etapa además se suelen realizar los tests de profundidad y esténcil anticipados, que evitan la generación de fragmentos innecesarios, siempre y cuando el *pixel shader* a ejecutar en la siguiente etapa no modifique la profundidad. El test de *scissor* también se realiza de forma anticipada en esta etapa.

Una vez que la primitiva es rasterizada en un conjunto de fragmentos, y los parámetros de los fragmentos son calculados (típicamente interpolando los valores de los vértices involucrados y los atributos asociados al mismo), se ejecuta el *pixel shader* (o *fragment shader)*, un programa de usuario que produce uno o varios valores de color, y si es requerido, también puede alterar el valor de profundidad del *Z-Buffer* activo.

Posteriormente, por cada fragmento se realizan los tests *alpha*, de esténcil y de profundidad. Si alguno de estos test falla, se descarta el fragmento sin actualizar él o los render targets correspondientes. En esta etapa también se realiza la operación de *alpha blending* o *additive blending*.

Por último, una vez que el procesamiento de todos los objetos termina y la GPU reciba una orden de resolver los *render targets*, se ejecutan operaciones para mostrar la imagen en pantalla o para preparar los *render targets* para su uso en etapas posteriores; en este momento se pueden realizar las operaciones relacionadas con los métodos de *antialiasing* por hardware, como *Multisample Antialiasing* (*MSAA*) o *Fullscreen Antialiasing* (*FSAA*).

A partir de GPUs con soporte para *shader model 4*, se incorporó la *General-Purpose Computing on Graphics Processing Units* (GPGPU), una alternativa para ejecutar programas que brindan mayor flexibilidad y que se ejecutan sobre los mismos *stream processors* disponibles para *shaders* convencionales. En el contexto de pipelines gráficos, la GPGPU permite una alternativa eficiente para codificar etapas que requieren una comunicación entre diferentes ejecuciones del mismo programa, dado que éstas comparten un área de memoria que puede leerse y escribirse (Houston & Lefohn, 2011). Sin embargo, para tareas convencionales es aconsejable utilizar directamente la programación clásica, dado que los diseñadores de GPUs priorizan la optimización del hardware para esta operatoria. Es posible cambiar en tiempo real entre el modo clásico y la GPGPU; no obstante, es recomendable limitar estos cambios al mínimo posible (por ejemplo, un cambio por cuadro renderizado) debido a que es necesario reconfigurar diferentes unidades de hardware, manteniéndolas ociosas hasta que el cambio se completa.

7.2 Forward Renderers

Los *forward renderers* fueron los primeros pipelines de iluminación local implementados y son los que se adaptan con mayor naturalidad al *pipeline* por hardware de las GPUs. De forma simplificada, consisten en realizar en un solo paso y por cada objeto los cálculos geométricos y los de iluminación, acumulando los resultados parciales por medio de la información de profundidad almacenada en el *Z-Buffer*. El resultado de este proceso es la imagen final, aunque es posible realizar una pasada adicional en el espacio de pantalla para aplicar un operador tonal (si no fue aplicado al momento de producir el color de cada fragmento) y efectos de post-procesamiento.

Existen dos posibilidades de cómo implementar este algoritmo general, el *forward renderer* de una única pasada y el *forward renderer* de múltiples pasadas, que varían en cómo aplicar el cálculo del aporte de las fuentes de iluminación locales sobre la geometría de la escena.

7.2.1 *Forward Renderer* de Única Pasada

Inicialmente se desarrolló el pipeline *forward renderer* de una única pasada, que se resume en el siguiente pseudocódigo:

```
Por cada objeto
        Buscar las luces que afectan al objeto.
        Renderizar el objeto en una sola pasada utilizando las luces halladas en el paso anterior.
```

Por cuestiones de eficiencia el *shader* de fragmentos utilizado para renderizar el objeto deberá estar diseñado idealmente para procesar solo un número de luces que coinciden con la cantidad y tipo de luces

que afectan al objeto. Debido a que cada material necesita un código específico por cada tipo de luz, el número de combinaciones a mantener aumenta rápidamente (explosión combinatoria). Además, es necesario actualizar las variables globales relacionadas con las luces locales frecuentemente, lo que reduce considerablemente el desempeño del sistema. También es necesario mantener todos los mapas de sombras en memoria hasta que se rendericen todos los objetos. Para evitar las tareas que realiza la GPU al momento de preparar la ejecución de un nuevo *shader* (diferente al último usado) y para reducir el número de *shaders*, se pueden codificar solo los casos más frecuentes (algo que puede ser difícil de estimar) y desperdiciar operaciones en el cálculo de luces sin aporte lumínico para los casos que no se corresponden con los implementados

Asimismo, la flexibilidad se ve limitada dado que resulta inviable aplicar una cantidad alta de luces locales por objeto, debido a las propias limitaciones de longitud de los *shaders*, la cantidad de registros disponibles, su complejidad y su costo computacional, lo que podría dificultar la correcta reproducción de la iluminación de una escena compleja. Para resolver este problema se desarrolló el *forward renderer* de múltiples pasadas.

7.2.2 *Forward Renderer* de Múltiples Pasadas

Un *forward renderer* de múltiples pasadas permite reproducir un número arbitrario de luces, renderizando el objeto con una o varias luces por pasada y componiendo estos resultados intermedios a través de *additive blending*. Su pseudocódigo es el siguiente:

```
Por cada luz
        Por cada objeto
                Renderizar el objeto utilizando la luz actual. Si la profundidad del Z-Buffer coincide
                con la del fragmento que se está procesando, sumar el resultado, utilizando additive
                blending, con el de las anteriores pasadas.
```

En esencia el problema se transforma en uno de $O(luces * objetos)$, pero no es necesario renderizar los objetos que están fuera del área de influencia de una luz, por lo que se pueden detectar estos casos y reducir el número de pasadas considerablemente. Además, es posible utilizar un esquema híbrido en el que se agrupen varias luces y se renderice al objeto con éstas en una sola pasada, repitiendo el proceso hasta que no haya más luces que afecten al objeto.

Aunque el *forward renderer* de múltiples pasadas es un esquema más flexible y potente, requiere la repetición de varias operaciones costosas: procesamiento de vértices y primitivas, acceso a las texturas del objeto, etc. Asimismo, ambos esquemas de *forward renderers* tienden a desperdiciar ciclos de reloj en cálculos innecesarios, debido a que calculan la iluminación en áreas del objeto que podrían no estar afectadas por la luz. Esto puede atenuarse en un *forward renderer* de múltiples pasadas si se utiliza el

esténcil *buffer* para generar una máscara que cubra el área de influencia de la luz; dado que para esto se renderiza un volumen, éste puede ser comparado con el *Z-Buffer* (que debe precalcularse) y de esta manera mejorar aún más la discretización del área de influencia de la luz; sin embargo, esta optimización pierde gran parte de su efectividad cuando se la combina con el agrupamiento de varias luces en una sola pasada. Similarmente, la iluminación puede ser calculada innecesariamente en objetos que se encuentran ocluidos por otros, pero esto puede ser evitado fácilmente si se pre-renderiza la escena generando solo el *Z-Buffer* (*Z Pre-Pass*) y se aprovecha el test de profundidad anticipado que realizan las GPUs modernas.

7.3 Deferred Renderers

Los *deferred renderers* desacoplan el procesamiento de la geometría del cálculo de iluminación. Inicialmente se renderiza la geometría utilizando un *shader* especial que almacena la información de profundidad, normales, e información adicional de la superficie en lo que se conoce como *buffer* geométrico o *G-Buffer*. Este *buffer* es accedido en etapas posteriores junto con la información de iluminación local y global de la escena, con el objetivo de producir la imagen final de la escena.

El concepto básico de esta técnica surge del artículo presentado en (Deering, Winner, Schediwy, Duffy, & Hunt, 1988), pero fue introducida de forma más concreta en el artículo (Saito & Takahashi, 1990), debido a que proponen renderizar los objetos de la escena en un *G-Buffer*, almacenando las normales, la profundidad y otras propiedades geométricas y aplicando sobre éste técnicas de procesamiento de imagen en espacio de pantalla para extraer contornos y siluetas entre otras posibilidades.

7.3.1 Deferred Shading

En 2002, con el soporte por hardware de múltiples *render targets* (MRT), se comienza a plantear el uso de esta tecnología y posteriormente se da forma al primer *deferred renderer* eficiente en tiempo real, conocido como *deferred shading* (Hargreaves, Deferred Shading, 2004; Thibieroz N. , Deferred Shading with Multiple Render Targets, 2004; Shishkovtsov, 2005). Su seudocódigo es el siguiente:

```
Por cada objeto opaco
        Renderizar el objeto con un shader que almacena la información geométrica en el G-Buffer.
Por cada luz
       Renderizar el volumen de influencia de la luz, calcular un BRDF sobre los puntos de superficie que
       ésta afecta utilizando la información contenida en el G-Buffer y almacenar el resultado de forma
       aditiva.
Por cada objeto transparente
        Renderizar el objeto en modo forward.
```

Esta modularización transforma al problema en uno de $O(luces + objetos)$, simplificando los *shaders*, mejorando el procesamiento por lote, permitiendo la aplicación de un alto número de luces y reduciendo el costo de las mismas, entre otras ventajas. Sin embargo, también añade un conjunto de desventajas: la flexibilidad a la hora de representar distintos BRDFs se ve seriamente limitada debido a que se ejecuta un mismo algoritmo para todas las superficies de los objetos que varía en comportamiento solo según los parámetros almacenados en el *G-Buffer*. Los objetos transparentes no pueden ser procesados directamente en el *G-Buffer* debido a que se necesitarían varias capas de información; por lo tanto, estos objetos deben ser renderizados posteriormente en un *forward renderer*. No es posible la utilización de *MSAA* sobre múltiples *render targets* en presencia de hardware *shader model 3* o inferior y ésta debe realizarse por medio de un *shader* en hardware contemporáneo, por lo que típicamente se buscan alternativas para incorporar un sistema de *antialiasing* en espacio de pantalla que evite estos problemas.

En la generación del *G-Buffer* debe almacenarse, por cada punto de superficie del objeto, toda la información necesaria para iluminarlo que, al igual que para un *forward renderer*, incluye la posición en espacio de mundo o de vista (la profundidad es suficiente para reconstruirla), la normal, el color albedo y las propiedades del material. Cuantos más parámetros se almacenen mayor será la flexibilidad a la hora de representar distintos materiales; sin embargo, un exceso de información podría requerir más espacio de almacenamiento del que se dispone, un mayor costo computacional en la generación del propio *G-Buffer* y la ejecución de un *shader* de iluminación más complejo con muchos saltos condicionales.

La Figura 7-2 muestra el *G-Buffer* generado en Killzone 2 y el resultado de composición de esta información con la información de iluminación directa de la escena; este motor gráfico permite reproducir gráficos de alta calidad con una compleja iluminación directa dinámica a un costo computacional bajo aunque no permite reproducir una gran variedad de materiales, quedando confinado a un BRDF del tipo Blinn-Phong limitado en expresividad.

Varios algoritmos, entre los que se incluye algoritmos de iluminación global como oclusión ambiental en espacio de pantalla y oclusión direccional, mapas de sombras, *decals* y partículas suaves, se benefician de disponer de la información de profundidad, normales y albedo de los objetos, por lo que la creación de un *G-Buffer* podría requerirse aún en un *forward renderer*, haciendo más atractiva la utilización de un *deferred renderer*.

Si bien el procesamiento de luces es más económico que en un *forward renderer*, es aconsejable realizar optimizaciones para reducir el costo de la etapa de iluminación. En primera instancia se puede realizar una operación de *frustum culling* sobre luces puntuales y *spot* utilizando un volumen que contiene su área de influencia, lo que descarta luces que no aportan iluminación en el cuadro actual.

La luces se procesan en espacio de pantalla, renderizando su volumen de influencia. Para la luz ambiental y las luces direccionales se utiliza un plano que cubre toda la pantalla dado que su área de influencia abarca toda la escena; en cambio, para las luces puntuales y las luces *spot* se utilizan esferas y conos respectivamente, lo que permite evitar la creación de fragmentos innecesarios y reducir así considerablemente el costo de procesamiento de las luces. Sin embargo, se genera una pequeña complicación dado que si se renderizan las caras externas del volumen, cuando éste se encuentre cercano a

la cámara se deberá detectar esta situación y renderizar las caras internas. También es posible renderizar las caras externas, pero se debe tener una consideración similar cuando este volumen intercepta el plano lejano de la cámara.

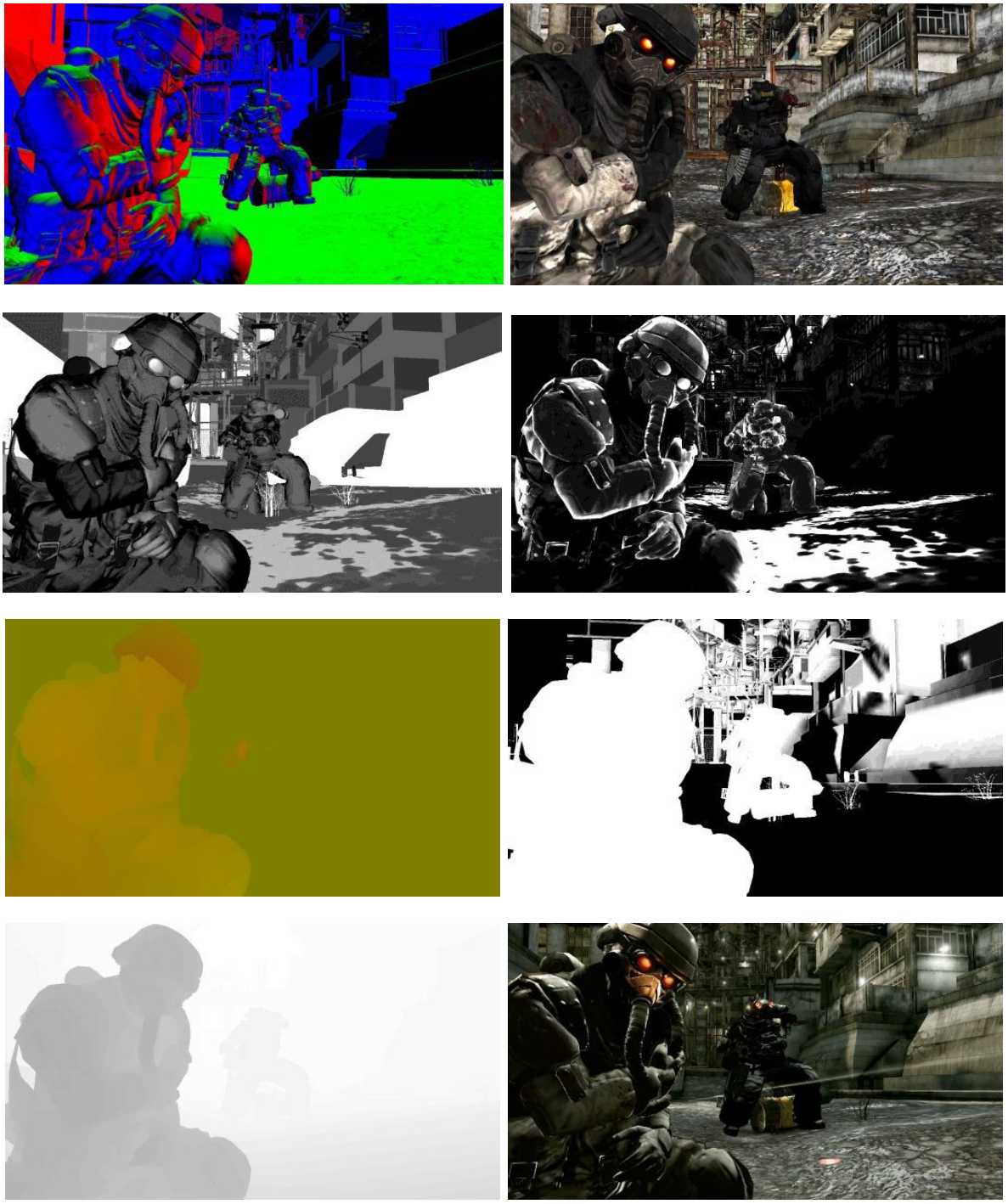


Figura 7-2 Información almacenada en el *G-Buffer* implementado en Killzone 2 (de izquierda a derecha y de arriba hacia abajo): normales, albedo, poder especular (*glossiness*), intensidad especular, vectores de movimiento (requeridos para *motion blur*), oclusión del sol (optimización para evitar ejecutar el *shader* de iluminación del sol en áreas que se aseguran están en sombras, los objetos dinámicos se renderizan siempre en 1 y el cielo siempre en 0) y profundidad. En la siguiente etapa (última imagen) se calcula y se acumula el aporte de cada luz utilizando la información del *G-Buffer* (Valient, The Rendering Technology of KillZone 2, 2009).

Adicionalmente se puede utilizar el *Z-Buffer* junto con operaciones esténcil con el objetivo de reducir aún más la generación de fragmentos innecesarios. Por cada luz, se resetea el esténcil *buffer* en 0 y se renderiza el volumen de las luces en dos pasadas. La primera renderiza las caras interiores del volumen y no afecta al *buffer* de iluminación, solo ejecuta una operación esténcil que testea si la profundidad es igual o mayor a la almacenada en el *Z-Buffer*; si lo es, marca al pixel con 1. En la segunda pasada se renderizan las caras exteriores del volumen utilizando el *shader* de iluminación, pero solo se procesan los fragmentos con valor de esténcil 1 y que pasan un nuevo test de profundidad que controla que la profundidad del fragmento sea menor o igual al valor del *Z-Buffer* (Figura 7-3). De esta forma, solo se procesan los fragmentos que se encuentran dentro del volumen de influencia de la luz. Si la cámara se encuentra dentro del volumen sólo se debe renderizar una de las pasadas, realizando un solo test de profundidad.

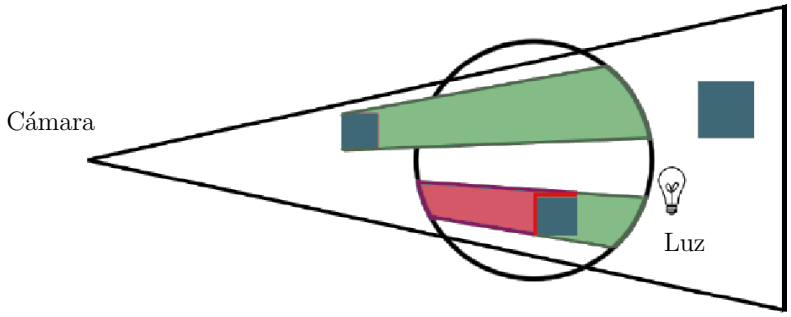


Figura 7-3. Esquema de renderizado de luces utilizado en Killzone 2. Se utiliza el *Z-Buffer*, las operaciones esténcil y un algoritmo de dos pasadas para reducir la cantidad de fragmentos generados. En la primera pasada se procesan las caras interiores y se detecta que solo las áreas verdes contienen fragmentos potencialmente útiles. En la segunda pasada se renderizan las caras exteriores y utilizando la información de la primera pasada y la información de profundidad de la escena se detecta que solo los fragmentos que abarcan el área roja son útiles (Valient, The Rendering Technology of KillZone 2, 2009).

Si el área de influencia de la luz en unidades de espacio de pantalla es pequeña, entonces es posible que este algoritmo sea más costoso que realizar solo una pasada, debido en parte a los cambios de estados que es necesario realizar en la GPU. En (Valient, The Rendering Technology of KillZone 2, 2009) propusieron calcular el diámetro angular del volumen de la luz y si éste es menor que un cierto umbral solo se renderiza el volumen en una pasada utilizando solo un test de profundidad.

Este esquema también permite la utilización de volúmenes definidos por el usuario (*clip volumes*), que pueden ser utilizados para controlar el área de influencia de la luz, lo que podría resultar además en una disminución del costo computacional de esta etapa (Figura 7-4).

El renderizado de objetos transparentes no puede realizarse de forma directa sobre el *G-Buffer* debido a que se necesitarían almacenar varias capas de información. Típicamente, estos objetos se renderizan posteriormente a la etapa de iluminación utilizando un *forward renderer*, lo que significa implementar dos pipeline gráficos; no obstante, los objetos transparentes habitualmente representan una fracción de los objetos de la escena por lo que una implementación simplificada de este segundo pipeline podría resultar suficiente en la mayoría de los casos (Koonce, 2007). Alternativamente, el hardware con soporte para *shader model 4* permite implementar técnicas como *reverse depth peeling* (Thibieroz N. , 2008) o *stencil-*

routed K-Buffer (Bavoil & Myers, Deferred Rendering using a Stencil Routed K-Buffer, 2008) que pueden utilizarse para almacenar varias capas de información en el *G-Buffer*, haciendo posible renderizar objetos transparentes con el mismo código que objetos opacos, pero aumentando los requerimientos de almacenamiento del *buffer* geométrico y aumentando la complejidad de creación y acceso a éste.

(a)

(b)

Figura 7-4 (a) La luz exterior influencia áreas interiores ocluidas. Esto puede ser solucionado con un mapa de sombras, pero a costa de ejecutar un algoritmo costoso. (b) Utilización de *clip volumes* que limitan el área de influencia de las luces con un costo computacional menor (Donzallaz & Sousa, 2011).

A partir de las GPU con soporte *shader model 4*, se puede activar *MSAA* conjuntamente con múltiples *render targets* y en un amplio espectro de formatos de color, inclusive en formatos de punto flotante de 16

bits. Con *shader model 4.1* se extendieron aún más los requerimientos mínimos, dando soporte a formatos de punto flotante de 64 bits. Sin embargo, aún si este hardware se encuentra disponible, resolver un mapa de profundidad o un mapa de normales con *MSAA* activado provocaría resultados erróneos, debido a que el promedio de estos valores no tiene sentido en este contexto.

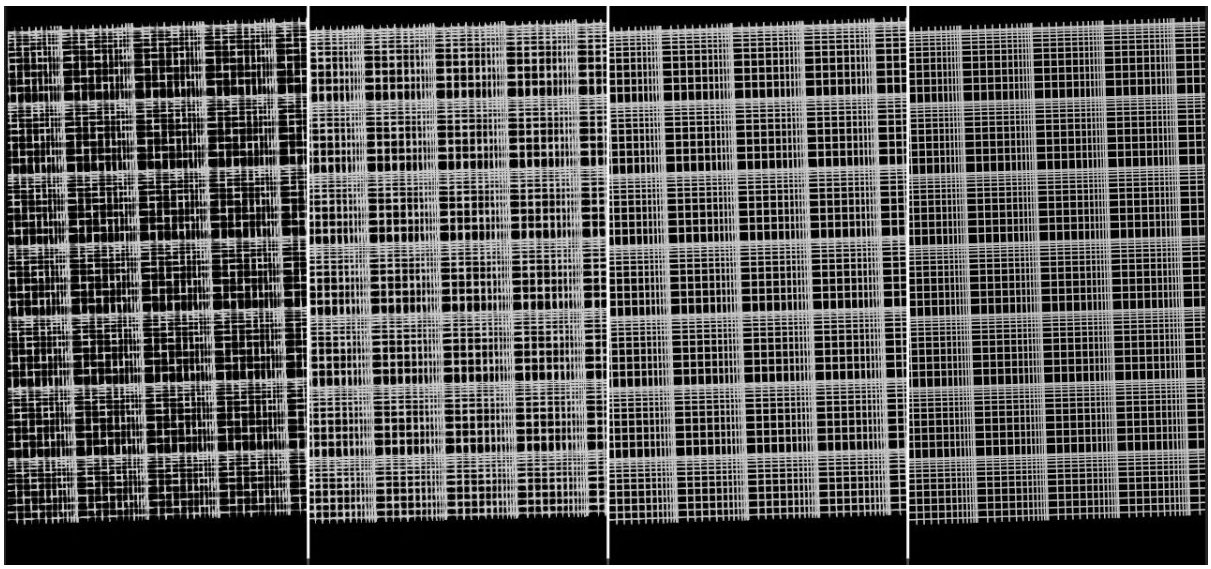
Afortunadamente, el hardware con soporte para *shader model 4.1* permite mantener un *buffer MSAA* sin resolver y acceder y renderizar por muestra en vez de por pixel. También es posible lograr el mismo resultado en hardware con soporte para *shader model 4* con un método de múltiples pasadas, pero a un costo computacional mayor. Si se genera el *G-Buffer* con *MSAA* y se calcula la iluminación por muestra, posteriormente se podrá generar un *buffer* de color *MSAA* que puede ser resuelto directamente por hardware y que produce un resultado equivalente a utilizar *MSAA* sobre un *forward renderer*. No obstante, ejecutar el algoritmo de iluminación por muestra es muy costoso, por lo que es importante utilizar un método de detección de bordes, como el propuesto en (Thibieroz N. , Deferred Shading with Multsampling Anti-Aliasing in DirectX 10, 2009), que detecta las áreas que deben ejecutarse por muestra y las áreas que deben ejecutarse por pixel. A pesar de que esta técnica efectivamente permite ejecutar *MSAA* en un *deferred renderer*, el costo de almacenar varios *buffers* con *MSAA* podría resultar muy alto.

Un enfoque alternativo y complementario, que puede ser implementado en hardware con soporte para *shader model 3*, es el utilizar algún tipo de filtro de *antialiasing* en espacio de pantalla que utilice como entrada la información de color, de profundidad y/o de normales. En S.T.A.L.K.E.R. (Shishkovtsov, 2005) se detectan los bordes comparando la diferencia de profundidad con la de los ochos vecinos del pixel y la diferencia de ángulo con las cuatro normales de los vecinos directos; luego, los bordes encontrados son difuminados promediando el valor de color del pixel con el de los cuatro vecinos directos. En Tabula Rasa (Koonce, 2007) se utiliza un concepto similar; en la detección de bordes se calcula la diferencia de profundidad con los ochos vecinos del pixel, pero se compara la máxima diferencia encontrada con la mínima diferencia, también se compara la normal con la de sus ochos vecinos. En Crysis (Sousa, Vegetation Procedural Animation and Shading in Crysis, 2007) se suavizan los bordes detectados por medio del mapa de profundidad y muestreando valores utilizando triángulos rotados y filtrado bilineal. Estos enfoques alivian el *aliasing* pero no son una solución efectiva.

Morphological antialiasing (*MLAA*) (Jiménez, Masia, Echevarría, Navarro, & Gutiérrez, 2010; Reshetov, 2009) es un método con gran aceptación que detecta bordes utilizando los mapas de profundidad, normales y/o color (permitiendo de esta manera también realizar *antialiasing* de texturas) buscando ciertos patrones que permiten encontrar potenciales líneas; luego se buscan líneas según este resultado y se aplica un filtro de difuminado que promedia valores según la ubicación relativa de los valores muestreados dentro de la línea en la que se encuentra, con lo que se obtienen resultados superiores a un costo computacional mayor que requiere 3 pasadas de procesamiento, pero que es posible ejecutar en hardware contemporáneo, inclusive en consolas de actual generación.

(a) (b) (c) (d)

Figura 7-5 Comparación entre los métodos de *antialiasing*: (a) *FXAA*, (b) *MLAA*, (c) *SMAA* 2X y (d) *SSAA* 16X (Jiménez, Echevarría, Sousa, & Gutiérrez, 2012).



(a) (b) (c) (d)

Figura 7-6 Comparación entre métodos de *antialiasing* y su tiempo de ejecución: (a) *FXAA* (0,62 ms.), (b) *MLAA* (0,98 ms.), (c) *SMAA* 2X (1,32 ms.) y (d) *CSAA* 16X (6,4 ms.), (Jiménez, Echevarría, Sousa, & Gutiérrez, 2012).

Fast approximate antialiasing (*FXAA*) (Lottes, 2009) es otro filtro similar en espacio de pantalla que detecta líneas comparando los valores de luminancia de la imagen y aplicando un filtro de difuminado que utiliza como entrada una clasificación de los patrones de contraste encontrados; esta técnica es actualmente una de las más utilizadas y suele combinarse con *MSAA*, aunque Timothy Lottes indicó informalmente que

la combinación de ambas técnicas puede producir artefactos en ciertos escenarios. También se han desarrollado métodos que utilizan el concepto de coherencia temporal (Yang, Nehab, Sander, Sitthi-amorn, Lawrence, & Hoppe, 2009) distribuyendo el costo de los cálculos a través de varios cuadros; entre éstos, se destaca la versión mejorada de *MLAA*, *Enhanced Subpixel Morphological Antialiasing* (*SMAA*) (Jiménez, Echevarría, Sousa, & Gutiérrez, 2012), que además mejora el proceso de detección de bordes, entre otros aspectos. La Figura 7-5 y la Figura 7-6 muestran una comparación entre *MLAA*, *FXAA*, *SMAA* y un método de *antialiasing* por hardware; no obstante, una comparación más profunda requiere el análisis de varios escenarios y de cómo estos métodos responden al efecto de *jitter*.

7.3.2 Deferred Lighting

Denominado *Deferred Lighting* o *Light-Pre Pass Renderer* (Balestra & Engstad, 2008; Engel, Designing a Renderer for Multiple Lights: The Light Pre-Pass Renderer, 2009), es un *deferred renderer* que solo almacena en el *G-Buffer* la mínima información necesaria para pre-calcular parcialmente la interacción de la luz con la geometría. Una segunda pasada, denominada *light pre-pass*, procesa las luces en espacio de pantalla de forma similar (y utilizando las mismas optimizaciones) a un *deferred shading*, pero calculando parcialmente el BRDF de las superficies, y obteniendo solo los cálculos directamente influenciados por las luces locales, que serán almacenados en un *buffer* especial, típicamente denominado *L-Buffer*. En una segunda pasada geométrica, se obtendrá el color final de la superficie procesando el *L-Buffer* con las propiedades del material no capturadas previamente, como el mapa de reflexión y su intensidad, el color albedo, la intensidad especular, etc. Su seudocódigo es el siguiente:

```
Por cada objeto opaco
        Renderizar el objeto con un shader que almacena parcialmente la información geométrica en el G-
      Buffer.
Por cada luz
        Renderizar el volumen de influencia de la luz, calcular parcialmente el BRDF sobre los puntos de
      superficie que ésta afecta, utilizando la información contenida en el G-Buffer y almacenar el
      resultado en un L-Buffer de forma aditiva.
Por cada objeto opaco
        Renderizar el objeto con un shader que componga la información del L-Buffer con las propiedades
      del material, obteniendo el color final de estas superficies.
Por cada objeto transparente
        Renderizar el objeto en modo forward.
```

El *G-Buffer* de un *deferred lighting* almacena el mapa de profundidad, normales y las propiedades de superficie necesarias para calcular el BRDF parcial en la etapa *light pre-pass*; por ejemplo, si se calcula un BRDF del tipo Blinn-Phong, se necesitará almacenar el poder especular. En presencia de tecnologías de

iluminación global, es posible que se almacenen otros canales de información, como por ejemplo el color albedo (para oclusión direccional en espacio de pantalla o *light propagation volumes*) o máscaras para mezclar luces ambientales; también es posible almacenar información para otras etapas, como por ejemplo los vectores de movimiento. No obstante, una de las razones para optar por este esquema es la reducción del tamaño del *G-Buffer*, por lo que suele almacenar solo un conjunto pequeño de datos. La consola Xbox 360 de Microsoft y la consola Wii U de Nintendo poseen una memoria caché, denominada EDRAM, que mantiene los *render targets* en uso y reduce casi completamente el costo de escritura en éstos; su tamaño es de 10 y 32 Mb respectivamente. Para estas consolas, particularmente la Xbox 360, el almacenamiento temporal de varios *render targets* de alta resolución podría sobrepasar el tamaño disponible, lo que provocaría la ejecución de una operación de *predicated tiling* (Microsoft, 2012), degradando levemente el desempeño. Alternativamente, es posible dividir la generación del *G-Buffer* en dos o más pasadas, pero esto también provocaría una caída del desempeño.

La pasada *light pre-pass* puede ejecutar cualquier BRDF, pero habitualmente se utiliza un modelo Blinn-Phong que es un modelo rápido de calcular y que permite representar una amplia gama de materiales. Similarmente al método *deferred shading*, es posible almacenar en el *G-Buffer* banderas qué indiquen que BRDF calcular; no obstante, este enfoque no resulta atractivo debido a que es necesario almacenar una bandera y posiblemente varios parámetros extras necesarios para cada BRDF. Además, el procesamiento de las luces se tornaría significativamente más costoso.

Debido a que la geometría se procesa por segunda vez, es posible renderizar un conjunto pequeño de objetos opacos en un *forward renderer* utilizando BRDFs alternativos; si bien se desperdician ciclos de procesamiento en el cálculo de luces, se obtiene aún mayor flexibilidad en el renderizado de objetos, aprovechando el primer procesamiento geométrico para técnicas de iluminación global y similares. Esta segunda pasada también permite completar la ecuación de iluminación calculada en la etapa previa con parámetros y cálculos bien diferenciados entre los distintos objetos, concediendo una mayor flexibilidad a la hora de renderizar.

En el caso de la adopción de un modelo de iluminación Blinn-Phong, la ecuación de iluminación simplificada para varias luces puntuales puede verse conceptualmente como:

$$I = \text{Color}_{Ambiental} + \sum_i Atenuación_i(N.L_i * (Color_i + (N.H_i)^n * Color_i)) \qquad \text{(Ecuación 7-1)}$$

Donde $Ambiental$ es el color de luz ambiental, $Atenuación$ es la función de atenuación de la luz, $Color_i$ es el color emitido por la fuente de luz y n es el poder especular de la superficie. Con el objetivo de armar la ecuación completa del BRDF, esta información se almacena en dos términos principales:

$$Difuso = \text{Color}_{Ambiental} + \sum_i Atenuación_i(N.L_i * Color_i)$$

$$Especular = \sum_i Atenuación_i(N.L_i * (N.H_i)^n * Color_i) \quad \text{(Ecuación 7-2)}$$

Debido a que gran cantidad de formatos de color brindan cuatro canales de información, en muchas implementaciones el término especular se almacena monocromáticamente en el canal alfa, es decir, se elimina la variable $Color_i$ de dicho término. En (Mittring, A Bit More Deferred - CryEngine 3, 2009) concluyeron que la diferencia perceptual entre el almacenamiento monocromático del término especular y el almacenamiento RGB puede ser marginal en la mayoría de los escenarios.

Deferred lighting es un esquema alternativo que permite reducir los requerimientos de almacenamiento en la generación del *G-Buffer* y de esta manera puede ejecutarse con mayor eficiencia en hardware antiguo. Consecuentemente, también se reducen los tiempos de generación del *G-Buffer* y se reduce el volumen de información accedida en la generación del *L-Buffer*, en la que se accede repetidamente al *G-Buffer* por cada luz procesada. Además, facilita la incorporación de una mayor variedad de materiales, pero para lograr una flexibilidad completa es necesario renderizar estos objetos en un *forward renderer*. Tampoco soluciona los problemas con objetos transparentes, ni permite ejecutar fácilmente un método de *antialiasing* por hardware. No obstante, ha demostrado ser un pipeline de renderizado eficiente que permite generar gráficos fotorrealistas; un claro ejemplo es el videojuego Crysis 2 (Sousa, Kasyan, & Schulz, CryEngine 3: Three Years of Work in Review, 2012), que complementándose con técnicas de iluminación global, permite generar escenas complejas con alta calidad visual y con iluminación completamente dinámica (Figura 7-7).

Figura 7-7 Captura de pantalla de Crysis 2, un videojuego que utiliza *deferred lighting* (Sousa, Kasyan, & Schulz, CryEngine 3: Three Years of Work in Review, 2012).

7.3.3 Inferred Lighting

Inferred Lighting (Kircher & Lawrance, 2009) es un *deferred renderer* que extiende la idea propuesta en *deferred lighting* permitiendo renderizar objetos transparentes, activar *MSAA* en la segunda pasada geométrica sin ningún tipo de consideración especial y reducir la resolución de los *buffers* temporales (*G-Buffer* y *L-Buffer*) sin, teóricamente, una pérdida considerable de calidad visual.

Similar a *deferred lighting*, en un *G-Buffer* de menor resolución que el *frame buffer* se almacenan la mínima información necesaria para precalcular parcialmente la interacción de la luz con la geometría, pero a ésta se le anexan los datos para aplicar, lo que ellos llaman, un *discontinuity sensitive filter (DSF)*; específicamente, utilizan 16 bits para almacenar la profundidad lineal del pixel, 8 bits para identificar al objeto (si hay más de 256 objetos deben compartirse identificadores) y 8 bits para identificar (casi unívocamente) las áreas donde las normales son continuas. El *L-Buffer* se calcula de la misma forma que en un *deferred lighting*, manteniendo la misma resolución que el *G-Buffer* previamente calculado. En la segunda pasada geométrica, al momento de renderizar un fragmento, se acceden a los 4 texels del *L-Buffer* que rodean al punto evaluado, se identifican los valores que corresponden con el objeto renderizado, descartando los valores que no coincidan con éste y promediando el resto. La identificación de áreas con normales continuas se utiliza para descartar valores de iluminación para áreas del mismo objeto donde posiblemente el color producido sea significativamente diferente. Pettineo (Pettineo, 2010) propuso alternativamente utilizar 16 bits para identificar los objetos eliminando toda información referente a la continuidad de las normales, con el objetivo de reducir el costo de generación y procesamiento del DSF y evitando los artefactos cuando objetos diferentes utilizan el mismo índice en escenas con más de 256 objetos.

Los objetos transparentes pueden ser procesados sobre este pipeline si éstos solo almacenan su información geométrica en un pixel de cada región de cuatro pixeles del *G-Buffer*, permitiendo de esta manera almacenar información de hasta tres objetos transparentes que solapan a uno opaco. De esta manera se calculan menos valores de iluminación por objeto, y es muy notorio en presencia de mapas de normales y sombras.

El seudocódigo de un *inferred lighting* es el siguiente:

```
Por cada objeto
        Renderizar el objeto con un shader que almacena parcialmente la información geométrica en el G-
      Buffer, anexando los datos necesarios para aplicar un discontinuity sensitive filter. Los objetos
      transparentes deben renderizarse en un patrón entrelazado. El G-Buffer está generado con una
      resolución levemente inferior a la del frame buffer.
Por cada luz
        Renderizar el volumen de influencia de la luz, calcular parcialmente el BRDF sobre los puntos de
      superficie que ésta afecta, utilizando la información contenida en el G-Buffer y almacenar el
      resultado en un L-Buffer de forma aditiva.
```

```
Por cada objeto
        Renderizar el objeto, acceder a la información del L-Buffer que rodee el fragmento procesado,
descartando los valores que no correspondan con el objeto renderizado o con el área del objeto en la que
se encuentra el fragmento. Componer el promedio pesado de estos valores con las propiedades del
material, obteniendo el color final de estas superficies.
```

Esta técnica soluciona los problemas relacionados con objetos transparentes y *antialiasing* por hardware. Sin embargo, el incremento de accesos a la información contenida en *L-Buffer*, la generación de los valores necesarios para aplicar el DSF y la menor precisión en detalles de alta frecuencia, en particular en presencia de objetos transparentes, reducen su atractivo. Hasta el momento se desconoce una implementación de este *pipeline* de renderizado en proyectos de envergadura.

7.4 Light-Indexed Renderers

Estos pipelines de renderizado almacenan los índices de las luces que afectan a un área en el espacio de pantalla o en el espacio de mundo. Luego, al momento de calcular la iluminación, ya sea en un *forward* o *deferred renderer,* la información de cada luz es accedida desde un arreglo utilizando estos índices. Alternativamente, es posible almacenar directamente en un *L-Buffer* la dirección, color y demás propiedades de cada luz, pero de esta forma se limitaría significativamente la cantidad de luces que pueden ser almacenadas por área debido al volumen de datos que debe almacenarse. El objetivo principal de estos pipelines de renderizado es permitir representar eficientemente un gran número de luces locales, sin pagar el costo computacional de los excesivos accesos al *G-Buffer* debido a que en los pipelines anteriormente introducidos se accede a estas propiedades repetidamente por cada luz procesada. Además, el *frame buffer* se accede solo una vez y la acumulación de los valores de iluminación se realiza en registros, con la mayor precisión de punto flotante disponible.

7.4.1 Light-Indexed Deferred Rendering

Light-indexed deferred rendering (Trebilco, 2009) almacena los índices de las luces que influyen en cada fragmento visible, para accederlas en una etapa posterior utilizando un *forward renderer* o un *deferred renderer.* En este trabajo no se discute la utilización de un *deferred renderer* en combinación con este esquema, pero se considera una alternativa válida, con ventajas y desventajas. Además, el nombre resulta confuso, por eso se propone llamarlo *light-indexed deferred renderer* y *light-indexed forward renderer* en el caso que se utilice un *deferred* renderer o un *forward renderer* respectivamente.

El seudocódigo de un *light-indexed forward renderer* es el siguiente:

```
Por cada objeto opaco
        Renderizar el objeto y obtener la información de profundidad (Z Pre-Pass).
Por cada luz
        Renderizar el volumen de influencia de la luz y almacenar el índice de ésta en el L-Buffer.
Por cada objeto
        Renderizar el objeto en modo forward indexando la información de iluminación almacenada en el L-
      Buffer.
```

El problema con este esquema surge cuando más de una luz afecta un punto de superficie y es necesario almacenar más de un índice por pixel. Idealmente, el primer índice se almacenaría en el canal rojo, el segundo en el verde, y así siguiendo. Pero para lograrlo se debe implementar un esquema de empaquetado, con la dificultad de que no es posible leer en el *shader* el valor previamente almacenado en el *L-Buffer*.

Una solución a este problema consiste en crear cuatro arreglos (asumiendo un máximo de 4 índices por pixel) y por cada luz directa, encontrar el arreglo en el que puede ser agregada sin que su volumen de influencia se solape con el de ninguna luz perteneciente a ese mismo arreglo. Luego se renderizan los volúmenes de influencia de las luces de cada arreglo, con los estados de renderizado habilitando solo la escritura del canal que corresponde con cada arreglo. Si existen luces que no pueden ser agregadas, puede realizarse el mismo proceso para una segunda pasada, requiriendo un *render target* adicional y la modificación de los *shaders* utilizados para renderizar los objetos; también es posible diferir las tareas y realizar una segunda pasada geométrica en forma aditiva que renderice los objetos utilizando la información de las luces que no pudieron agregarse en la etapa anterior.

Alternativamente, es posible empaquetar los índices utilizando las operaciones de *bit shifting* introducidas en *shader model 4*. Este esquema permite, además, balancear los requerimientos entre el número de luces que pueden influenciar un pixel y la cantidad de luces totales a indexar. Sin embargo, el desempaquetado es una operación levemente compleja, que requiere hardware moderno y conocer de antemano los requerimientos de la escena.

El arreglo que contiene la información de las luces puede almacenarse utilizando una textura, lo que permite un acceso fácil y eficiente, aunque es posible que la actualización de esta textura provoque una caída de desempeño. También es posible utilizar los *buffers* contantes introducidos en GPUs con soporte para *shader model 4*.

Las limitaciones referentes a *antialiasing* por hardware y objetos transparentes pueden solventarse si se eliminan las optimizaciones sobre el renderizado de los volúmenes de influencia que utilizan el mapa de profundidad, capturando todas las luces que interceptan el rayo que parte desde el punto de vista hasta el plano lejano de la cámara. Sin embargo, no se puede utilizar la técnica de *clip volumes y* combinar esta técnica con sombras suaves es costoso, hasta posiblemente inviable debido a que se debe disponer de todos los mapas de sombras al mismo tiempo; aún en presencia de sombras duras, utilizarlas para descartar fragmentos en el proceso de generación de índices significaría no poder renderizar objetos transparentes bajo este esquema o no poder activar *MSAA*. Si se renderizan los objetos en modo *deferred*, para solventar

las limitaciones de *antialiasing* y renderizado de objetos transparentes se debe recurrir a los mismos métodos propuestos en la sección 7.3.1, pero se evita procesar la geometría una segunda vez.

Además, se necesitaría almacenar un gran volumen de información para representar todos los posibles parámetros para un amplio espectro de tipos de luces; aún si se almacenan estas propiedades en arreglos personalizados, el costo computacional de representar diferentes tipos de luces es alto. Hasta el momento, este esquema no ha sido utilizado en ningún proyecto de envergadura.

7.4.2 Tile-Based Renderers

Similar al anterior, *Tile-Based Rendering* (Balestra & Engstad, 2008; Andersson, 2009; Olsson & Assarsson, 2011) almacena los índices de las luces que influyen en cada grilla definida en espacio de pantalla, para accederlas en una etapa posterior utilizando un *forward renderer* o un *deferred renderer.* Aunque es una diferencia sutil, este esquema es de mayor atractivo debido a que permite utilizar eficientemente una solución de GPGPU con el objetivo de mejorar significativamente la eficiencia de ciertas etapas.

Tile-based deferred shading es el pipeline de renderizado utilizado en Frostbite 2 de DICE, uno de los motores de tiempo real que logra mayor fotorrealismo (Figura 7-8). En esta implementación, se genera el *G-Buffer* utilizando técnicas convencionales y se realiza el cálculo de todas las demás etapas utilizando GPGPU., dado que estos cálculos pueden codificarse y ejecutarse con mayor eficiencia bajo este esquema de procesamiento, particularmente para GPUs con soporte para *shader model 5* (Andersson, 2009).

Figura 7-8 Captura de pantalla de Battlefield 3, videojuego desarrollado sobre Frostbite 2, un motor gráfico que utiliza *tiled-based deferred shading* programado parcialmente con GPGPU (Andersson, 2009).

Similarmente, *tile-based forward rendering* puede ejecutarse parcialmente utilizando GPGPU. En Leo Demo de AMD se realizan las tareas de *culling* de luces y la creación del *L-Buffer* con GPGPU, realizando el resto de las tareas de manera convencional (AMD, 2012).

7.4.3 World Space Light-Indexed Forward Rendering

Para el videojuego Just Cause 2 se introdujo una técnica (sin nombre) que se basa en *light-indexed deferred rendering* pero que almacena los índices de las luces que influyen en determinadas áreas en el espacio del mundo de la escena (Persson, Making it Large, Beatiful, Fast, and Consistent: Lessons Learned Developing Just Cause 2, 2010). Esta técnica surge debido a la necesidad de renderizar rutas y otros escenarios exteriores, en los que un gran número de luces se solapan en espacio de pantalla. De esta manera, se desborda rápidamente la cantidad de índices que se pueden almacenar para unas pocas áreas en espacio de pantalla, y al mismo tiempo, solo unos pocos índices son almacenados para la mayoría de las áreas restantes (Figura 7-9).

Los índices se almacenan en una textura 2D, aunque es posible utilizar una 3D si se considera conveniente dividir las áreas también en altura. Típicamente, la mayor parte de las luces no suelen trasladarse, por lo que es posible actualizar esta textura cada una cierta cantidad de cuadros, y actualizar solo las áreas de mundo donde se detectaron movimientos de luces. Además, la identificación de las luces que afectan cada área de mundo típicamente es más sencilla que en otros métodos. Sin embargo, esta técnica no permite la utilización de *clip volumes* para luces, ni permite descartar cálculos innecesarios en fragmentos que no son influenciados por la luz.

Figura 7-9 Captura de pantalla del videojuego Just Cause 2. En esta escena se aprecia la representación de un gran número de luces locales dinámicas en los asentamientos de la costa y de la montaña (Persson, Making it Large, Beatiful, Fast, and Consistent: Lessons Learned Developing Just Cause 2, 2010).

Capítulo 8 Caso de Estudio: XNA Final Engine

Paralelamente al desarrollo de esta tesis se diseñó e implementó un *framework* para la rápida generación de aplicaciones gráficas fotorrealistas. Este proyecto se denominó XNA Final Engine y su código fuente se encuentra disponible en el sitio de proyectos abiertos CodePlex (Schneider & Schefer, XNA Final Engine, 2010).

La creación de este *framework* la motivó el poder analizar los resultados visuales de un conjunto de técnicas gráficas fotorrealistas y evaluar su desempeño y viabilidad en hardware contemporáneo. Posteriormente, el desarrollo fue extendido con el objetivo de generar una herramienta que facilite la creación de videojuegos y aplicaciones gráficas en general y que también pueda utilizarse tanto como una herramienta de aprendizaje como de desarrollo a mediana escala.

En este *framework* se soporta hardware comercial contemporáneo, particularmente computadoras personales que disponen de GPUs con *shader model 4* o superiores y la consola Xbox 360 de Microsoft.

La elección de la plataforma de desarrollo es de vital importancia para un proyecto de esta envergadura, por lo que fue necesario realizar un minucioso análisis de las alternativas disponibles. De este análisis se concluyó que era necesario crear un *framework* desde sus cimientos, por lo que se realizó una evaluación de los lenguajes y APIs gráficas que derivó en la selección de C# y XNA. El siguiente paso consistió en instaurar una arquitectura de software simple, mantenible, flexible y eficiente; se seleccionó la programación orientada a datos y el desarrollo basado en componentes como los principios a seguir para lograr este cometido.

En este capítulo se discutirá el pipeline gráfico en el contexto del *framework* y las tecnologías gráficas implementadas, se introducirá el editor del *framework* y se mostrarán distintos proyectos que utilizan actualmente este *framework*. Por último, se realizará un análisis de desempeño del *framework*.

8.1 Plataforma de Desarrollo

La elección de la plataforma de desarrollo se realizó minuciosamente y analizando en profundidad distintas alternativas dado que era clave debido a la complejidad del proyecto. Se planteó utilizar una plataforma que permitiera el rápido y sencillo desarrollo, que redujera el riesgo de la aparición de potenciales errores de codificación y que brindara flexibilidad a la hora de implementar o adaptar tecnologías gráficas fotorrealistas. Se consideraron plataformas tales como *Unity3D* y se determinó que son una opción viable dado que proveen un marco de trabajo robusto para el desarrollo de estas aplicaciones, en particular porque permiten anexar (con ciertas restricciones) distintos tipos de *shaders*, evitando la necesidad de codificar todos los elementos de un *framework*, brindando un soporte robusto y la capacidad de portar el trabajo a varias plataformas de interés (Goldstone, 2011). Sin embargo, se decidió

implementar un *framework* interactuando directamente sobre una API gráfica debido a que este enfoque permite examinar, controlar e implementar el pipeline gráfico subyacente; particularmente permite experimentar con las representaciones de color y los operadores tonales, permite evaluar con mayor precisión el rendimiento de las técnicas y brinda la posibilidad de estudiar aspectos que de otra manera permanecerían abstraídos en una capa intermedia.

Por esta razón, fue necesario seleccionar un lenguaje de programación y una API gráfica para desarrollar un *framework* desde sus cimientos. Desafortunadamente no es posible realizar la elección de ambos de manera totalmente independiente debido a que no todas las APIs se encuentran disponibles en todos los lenguajes y, aún en presencia de *wrappers*[8], éstos no siempre disponen de una buena interfaz o funcionan correctamente.

8.1.1 Lenguaje de Programación

Debido a la necesidad de disponer de una API gráfica flexible, rápida y fácil de utilizar se decidió preseleccionar dos lenguajes, el lenguaje C++ y el lenguaje C#, dado que disponen de APIs nativas que mostraron ser aptas para este tipo de desarrollo. Históricamente, al momento de seleccionar un lenguaje de programación para desarrollar una aplicación gráfica compleja sólo surgía una única alternativa, el lenguaje C++. Con el correr de los años y con una creciente maduración del mismo, el lenguaje C# comenzó a perfilarse como una alternativa válida al lenguaje C++ para la creación de este tipo de aplicaciones. En la actualidad, en la industria del videojuego se lo suele utilizar para desarrollar herramientas de producción y ya se han realizado juegos comerciales exitosos completamente en este lenguaje, como lo son *Bastion* y *Magikca*, entre otros (Figura 8-1).

Ambos lenguajes tienen ciertas ventajas que podrían ser determinantes en el desarrollo de una aplicación gráfica, por lo que resultó importante una correcta evaluación de las características, funcionalidades y ventajas de ambos. A continuación se describirán las ventajas y desventajas más significativas de ambos y las razones por la que se optó por C#, razones que serán complementadas en la siguiente sección cuando se discuta la selección de la API gráfica. Serán ignoradas características que no afectan a este tipo de desarrollo, como por ejemplo, las expresiones lambda y LINQ de C#, que son tecnologías útiles en varias ramas de computación, pero que por problemas de desempeño no es deseable su utilización en aplicaciones gráficas.

[8] Un *wrapper* consiste en una capa de código que traslada la interfaz de una librería a una interfaz compatible. Esto se realiza por distintas razones, tales como refinar un diseño deficiente o una interfaz complicada, permitir que diferentes códigos se puedan intercomunicar (por ejemplo, incompatibilidades en el formato de los datos) o permitir la intercomunicación de código entre diferentes lenguajes.

(a) (b)

Figura 8-1 Captura de pantalla de pantalla de (a) *Magicka* y de (b) *Bastion*, dos videojuegos comerciales exitosos realizados en C# utilizando la API gráfica *XNA*.

Entre las ventajas de C# sobre C++ se destacan (Skeet, 2010):

- ***Lenguaje moderno / paradigma de rápido desarrollo***. Actualmente un porcentaje importante de las aplicaciones comerciales se desarrollan en lenguajes como Visual Basic o Java que funcionan con un paradigma de rápido desarrollo. C# puede ser visto como una evolución o modernización de C++ (como lo fue C++ de C) que incorpora el paradigma de rápido desarrollo. Además, C# está en fase de evolución y en períodos reducidos de tiempo, normalmente de dos a cuatro años, surge una nueva especificación con nuevas características y funcionalidades, que permiten potenciar el rápido desarrollo.
- ***Administración automática de memoria***. C# libera al desarrollador de asignar y liberar manualmente la memoria ocupada por los objetos, dejando estas tareas a un recolector de basura. Además, de esta manera, los punteros se vuelven transparentes al desarrollador. La administración automática de memoria incrementa la productividad y calidad del código sin impactar significativamente a la expresividad ni al desempeño.
- ***Fuertemente tipado y tipado seguro***. C# define mayores restricciones sobre conversiones implícitas y previene con mayor recelo errores de tipos. C# inicializa las variables por valor e inicializa con el identificar null a los tipos por referencia, la sentencia *if* solo acepta operadores booleanos, se chequea automáticamente el rango en los accesos a los arreglos y, a diferencia de C++, no se puede sobrescribir memoria no asignada. No obstante, C# puede acceder a punteros sin tipos, pero esto debe realizarse utilizando la palabra clave *unsafe* que podría llegar prohibirse su utilización por el compilador. Similarmente, es posible crear regiones de código *unsafe context* que permiten deshabilitar el chequeo dinámico de tipos. Además, C# tiene soporte para operaciones de *casting* (conversión de tipo explicitas) en tiempo de ejecución, pero si el *casting* es inválido se retornará un puntero nulo (si se usa un as) o una excepción (si se utiliza una operación de *casting* similar a las usadas en C).
- ***Uniformidad***. C++ es orientado a objetos, pero C# va más allá dado que hasta el tipo de dato más simple es tratado como un objeto. Esto y una cuidada planificación hacen de C# un lenguaje más uniforme.

- ***Sintaxis simplificada / programación simplificada***. C++ es un lenguaje extremadamente flexible y completo, pero esto va en detrimento de su simplicidad. C# intenta simplificar la sintaxis al ser más consistente y lógico mientras remueve algunas de las características más complejas de C++. Por ejemplo, C# no utiliza el concepto de punteros, los archivos de cabecera (.h) han sido eliminados y los operadores de referencia y de acceso a espacio de nombres, :: y -> respectivamente, se han remplazado por un único operador, el punto (.). Asimismo, C# no tiene herencia múltiple (pero si herencia múltiple de interfaces), no importa el orden en que hayan sido definidas las clases ni los métodos, no existen dependencias circulares y no existen las variables, funciones ni constantes globales (se suplantan con variables, métodos y clases estáticas). Al mismo tiempo, C# incorpora conceptos como propiedades, parámetros opcionales, parámetros nombrados, métodos de extensión, tipos implícitos, inicialización simplificada, tipos anónimos, etc.
- ***Librerías / herramientas de framework .NET***. El ambiente .NET brinda un conjunto robusto y documentado de librerías y herramientas; sin embargo, puede resultar muy difícil adaptar o agregar nueva funcionalidad no disponible dentro del marco .NET.
- ***Políticas sobre versiones de librerías***. El denominado DLL Hell (infierno de DLLs) es un problema constante tanto para usuarios como para desarrolladores. C# está diseñado para facilitar el manejo de las distintas versiones de las librerías utilizadas, permitiendo que éstas evolucionen y al mismo tiempo mantengan la compatibilidad con versiones anteriores.
- ***Comentarios XML***. C# soporta comentarios en XML que pueden convertirse automáticamente en documentación y que tiene como efecto colateral fomentar la documentación del código.
- ***Regiones***. Al codificar puede ser útil ocultar o agrupar visualmente un conjunto de sentencias de código. Las regiones permiten realizar esto elevando la legibilidad y productividad.
- ***Genericidad***. Esta característica permite definir estructuras de datos con tipado seguro sin especificar exactamente cuál es ese tipo, simplificando la codificación, evitando en el proceso algunos errores de codificación y mejorando la calidad/legibilidad del código fuente. C++ por su parte, utiliza el concepto de plantillas (*templates*), un concepto similar que permite mayor flexibilidad y posibilidades. Los argumentos de una plantilla, por ejemplo, no necesariamente tienen que ser tipos, pueden usarse variables, funciones y expresiones constantes, lo cual abre un nuevo conjunto de posibilidades. Sin embargo, las plantillas no son fuertemente tipadas como lo son *generics* de C# y producen un código levemente más difícil de entender. A su vez, un mal uso de las plantillas podría producir errores en tiempo de ejecución con mensajes poco significativos que podrían provocar una lenta identificación del problema.

Entre las ventajas de C++ sobre C# se destacan:

- ***Soporte de APIs y librerías***. Las dos APIs gráficas más utilizadas, DirectX y OpenGL funcionan nativamente en C++ y, como es lógico, gran cantidad de la documentación, tutoriales y libros están enfocados en estas APIs. Asimismo, en C++ se encuentran disponibles una gran cantidad de librerías y herramientas que no solo facilitan la realización de tareas, sino que también podrían ser la única alternativa viable si se buscan anexar ciertas tecnologías específicas. Pocos *wrappers* han

sido desarrollados para C# y típicamente son limitados en funcionalidad, presentan errores, no suelen actualizarse de manera continua y además en ciertos casos no es posible adaptar fácilmente el paradigma de funcionamiento de una librería de C++ a C#, por las restricciones que impone este último.

No obstante, C# tiene soporte para la API gráfica XNA, una API simple y potente que además brinda un conjunto importante de funcionalidad adicional para realizar otro tipo de tareas. El resto de las librerías o funcionalidades complementarias necesarias para la realización de este *framework* tienen una aceptable calidad y soporte en C#.

- ***Flexibilidad y control***. A pesar de que C# ha incorporado nueva funcionalidad en cada nueva versión, C++ incluye un conjunto más grande y flexible de constructores del lenguaje, a costa de aumentar la complejidad del lenguaje y aumentar la probabilidad de generar de errores de codificación. El desarrollo de aplicaciones gráficas es una tarea compleja en la que la expresividad y el control podrían resultar importantes o determinantes; sin embargo, la funcionalidad y flexibilidad que brinda C# es amplia.
- ***Velocidad de ejecución***. Existe un consenso de que C++ es más rápido en ejecución que C#. No obstante, es difícil concluir cuál es la diferencia real de velocidad entre ambos, en especial porque no existen buenas métricas para evaluar esta diferencia. Es posible analizar distintos factores que afectan a la velocidad de ejecución de ambos lenguajes, pero hasta el momento solo hay (según el relevamiento realizado) estimaciones sin rigor académico, como la realizada en (Morin, 2002).

  C# o cualquier programa compilado en la plataforma .NET corre en un ambiente *sand-box* y por esta razón muchas instrucciones tienen que controlarse por seguridad. Si un determinado fragmento de código necesita ejecutarse muchas veces y la velocidad de ejecución de ese código es crítica entonces C# podría resultar inviable. C# permite ejecutar código inseguro (*unsafe*) reduciendo esta penalidad, pero eliminando de esta manera algunas de las ventajas de C# y reduciendo la portabilidad. También se podría codificar ese código en C++ y llamarlo en C# a través de las herramientas de inter-operatividad, produciendo similares consecuencias.

  C# compila el código en dos pasadas, la primera es realizada por el compilador de C# en la máquina del desarrollador y la segunda es realizada por el *framework* .NET de ejecución en la máquina del usuario. Por consiguiente, existe la posibilidad de que un código compilado de esta manera sea más rápido que un código similar escrito en C++, debido a que en la segunda compilación se conoce perfectamente el ambiente de ejecución y el tipo de hardware disponible y de esta manera se pueden generar instrucciones que sean ideales al procesador específico subyacente. Además, C# permite potencialmente ejecutar el código eficientemente en arquitecturas todavía no desarrolladas al momento de producir la aplicación.

  Un compilador clásico en C++ genera un código nativo que usualmente es el menor común denominador de todos los procesadores disponibles, lo que significa que un programa C++ normalmente no toma partido de las ventajas incorporadas en nuevos procesadores. Cada nueva arquitectura de procesadores normalmente incluye un nuevo conjunto de instrucciones, pero si el desarrollador quiere aprovechar estas tecnologías debe, o bien generar distintos ejecutables para cada uno de los conjuntos de instrucciones que quiera aprovechar, o bien generar uno solo y dejar

de lado a los usuarios con procesadores más antiguos. No obstante, dados los altos requerimientos de hardware de las aplicaciones gráficas, ignorar una arquitectura antigua de procesadores no suele ser un problema de consideración.

En algunos escenarios puntuales, C# podría brindar la posibilidad de producir código más eficiente; sin embargo, típicamente las aplicaciones bien diseñadas se ejecutan con mayor eficiencia en C++, aunque es más difícil crear código bien diseñado en C++ que en C#. Para una aplicación típica de pequeña o mediana escala, la diferencia de velocidad de ejecución podría no ser suficiente para compensar las desventajas de C++ debido a que es beneficioso disponer de un lenguaje más simple, menos propenso a errores, con un código resultante de menor longitud y más legible, lo que típicamente resulta en código más rápido y estable. Todavía no está claro si esto también podría ser cierto para un proyecto de gran escala dado que no parece existir ninguna iniciativa que tenga como objetivo la utilización de este lenguaje.

- ***Portabilidad***. C++ suele ser el lenguaje utilizado en la mayoría de las plataformas, lo que lo hace atractivo para aprovechar el código producido, no solo en computadoras personales, sino también en consolas de última generación u otros dispositivos. C# puede ser utilizado en Xbox 360 con XNA como API gráfica y sin necesidad de comprar costosos SDK, pero deben tenerse en cuenta las limitaciones del ambiente de ejecución implementado en dicha consola para atenuar la reducción de desempeño que esta implementación produce.

8.1.2 API Gráfica

Se seleccionó XNA porque funciona de forma nativa en C# y ha evolucionado con el correr de los años en una librería estable y funcional. Debido a su modernidad, ésta se encuentra organizada siguiendo un esquema orientado a objetos (organización que se reconoce superior al antiguo esquema de máquina de estados), su interfaz es uniforme y limpia y típicamente con una cantidad menor de líneas se pueden realizar las mismas tareas con respecto a APIs como OpenGL y DirectX. En algunos casos, la sencillez de su interfaz va en detrimento de la flexibilidad, pero en la mayoría de los casos estas limitaciones han sido impuestas para garantizar un comportamiento uniforme en las distintas plataformas que soporta. Esto es posible dado que XNA funciona sobre la consola Xbox 360 y computadoras personales con GPUs con *shader model 4* o superiores. A partir de *shader model 4*, los fabricantes deben garantizar cubrir un mínimo común denominador de funcionalidad; bajo este pretexto y el hecho de que Xbox 360 es una plataforma cerrada, el equipo diseñador de XNA limitó ciertas funcionalidades y por consiguiente se reduce la probabilidad de aparición de errores en plataformas no testeadas (Hargreaves, Shawn Hargreaves Blog). Un aspecto negativo de lograr un comportamiento uniforme en todas las plataformas, es el hecho de que

debieron limitarse a la tecnología *shader model 3*[9]. Se consideró que esta limitación no imponía restricciones importantes, en particular, porque la mayoría de los desarrollos tecnológicos viables en tiempo real están enfocados en soportar las características de las consolas de actual generación y no el hardware más moderno.

XNA soporta GPUs con *shader model 2* y *3* y los teléfonos móviles *Windows Phone*, pero para mantener la uniformidad de funcionamiento la API los separa en una categoría adicional en la que se restringe aún más la funcionalidad, lo que hace imposible implementar tecnologías fotorrealistas y por esta razón no se dio soporte a estas plataformas de hardware.

XNA puede ser vista como una API de más alto nivel que OpenGL y DirectX. Soporta una amplia variedad de recursos nativamente: modelos (FBX y X), texturas (DDS, JPG, PNG y BMP), *shaders* (HLSL), fuentes *bitmap* (generadas a partir de fuentes TTF), sonidos (WAV y MP3), música (MP3 y WMA) y videos (WMV). Estos son preprocesados y precompilados y pueden cargarse utilizando simplemente una sentencia del lenguaje. Además se administran para evitar la carga repetida de recursos y se provee una interfaz para manejar los recursos como datos administrados por el recolector de basura, lo que reduce la posibilidad de fuga de memoria (*memory leak*). También provee clases y métodos para reproducir sonido posicional multicanal, música y videos, operaciones de *networking*, soporte básico para animación *skinning*, lectura de dispositivos de entrada (teclado, mouse y gamepads), administración del ciclo de actualización de la aplicación, operaciones matemáticas, etc. El SDK también incluye la herramienta PIX que permite encontrar problemas de desempeño y malas decisiones de diseño y que puede evaluar tanto una aplicación para computadoras personales como para Xbox 360.

El SDK de XNA, denominado *XNA Game Studio*, es gratuito para su uso en computadoras personales y requiere una cuota anual de aproximadamente 100 dólares para su uso en Xbox 360. Sin embargo, para uso académico se ofrece un plan en el que no es necesario el pago de la primera cuota anual.

8.2 Arquitectura de Software del *Framework*

El diseño de una buena arquitectura de software es de vital importancia para proyectos de mediana y gran complejidad. Se planteó como objetivo diseñar una interfaz uniforme, simple y que permita una gran potencialidad de acción. También se planteó la necesidad de producir código mantenible y extensible, en parte por la complejidad del problema y en parte porque cuando se diagrama un *framework* de estas características, típicamente no es posible ni deseable, definir con precisión cada uno de sus elementos. Debido a que el desempeño computacional de una aplicación gráfica es crítico, en especial en presencia de

[9] El requerimiento de una GPU *shader model 4* o superior surge por la necesidad de asumir la presencia de ciertas características, como el soporte para ciertos formatos de *render targets*, soporte de filtrado en éstos, etc. No obstante, el *shader model 3* es el máximo común denominador entre estas GPUs y la consola Xbox 360 y es por esta razón que funcionalmente está limitado a éste.

tecnologías fotorrealistas, es importante producir software que sea eficiente en ejecución en todas las plataformas soportadas.

En el 2010 se realizó una prueba temprana con una versión alfa del *framework* (sección 8.5.2) y ésta demostró que la interfaz no era lo suficiente uniforme ni simple, que realizar extensiones no era una tarea sencilla y que existían una gran cantidad de problemas de desempeño, entre otros, problemas de diseño. Por esta razón se realizó un relevamiento de arquitecturas de software en aplicaciones gráficas complejas, particularmente videojuegos y esto permitió llegar a la conclusión de que el paradigma orientado a objetos (el usado hasta ese momento en la realización del *framework*) no era el ideal para una aplicación en la que se maneja una gran cantidad de datos uniformes y en lote. En contrapartida, la programación orientada a datos junto con el desarrollo basado en componentes (Apéndice A) proveen una estructura de software que permite producir código mucho más eficiente, mantenible, extensible y paralelizable para este tipo de aplicaciones. Por consiguiente se rediseñó completamente el *framework* siguiendo estos principios. Se utilizó como referencia la organización de clases e interfaz de Unity3D, que es uniforme, simple y que tiene buena aceptación por parte de usuarios profesionales y aficionados.

El *framework* puede ser visto como un conjunto de elementos que se agrupan en las siguientes categorías:

- **Entidades de la escena**. Llamadas *GameObject* en el *framework*, representan cada entidad presente en el mundo y abarcan desde objetos visibles hasta objetos de lógica que controlan a otras entidades o subsistemas del *framework*. Las entidades no tienen ningún comportamiento por sí solas; solo agrupan componentes que son los encargados de darle la identidad.

  En este proyecto se decidió brindar dos tipos de entidades, las que operan en 2 dimensiones (espacio de pantalla) y las que operan en 3 dimensiones (espacio de mundo). Cada una tiene un tipo particular de componente de transformación, un componente de transformación para 2 dimensiones y uno para 3 dimensiones respectivamente. Además, algunos componentes pueden operar en uno de los espacios o en ambos. Con este enfoque se dispuso una forma clara y sencilla de manipular objetos de juego como lo son los elementos de la interfaz de usuario. La parte negativa es que el desarrollador podría encontrar poco intuitivo saber qué componente funciona en cada tipo de objeto de juego. Además, los componentes que operan en ambos espacios podrían llegar a ser más complejos.
- **Componentes**. Determinan el comportamiento de las diferentes entidades de la escena. Cada componente representa datos y un conjunto de operaciones sencillas para operarlos, pero no para procesarlos. Estos se procesan automáticamente y en lote por un subsistema adecuado del *framework*.

  Entre los tipos de componentes se encuentran, entre otros, los componentes de:
  - Transformación, que posicionan a la entidad en el mundo.
  - Renderizado, que contienen información para renderizar a la entidad como una partícula, una malla, un texto, etc.
  - Audio, que permiten reproducir sonido posicional en la posición de la entidad.
  - Animación, que brindan la posibilidad de reproducir animaciones rígidas y *skinning*.

  - Física, que establecen un nexo entre el mundo físico y el de la escena.
  - Cámara, que contienen la información necesaria para visualizar la escena en pantalla.
  - Luces, que representan luces puntuales, direccionales y *spot*.
  - Script, que permiten anexar código para controlar la lógica de la entidad o de la propia escena.
- **Recursos**. Llamados *assets* en el *framework*, representan los recursos artísticos usados en la escena, entre los que se encuentran audio, video, modelos, texturas, animaciones, fuentes, *shaders* y materiales. Se administran utilizando las unidades llamadas *ContentManager*, que permiten agrupar y eliminar de memoria rápidamente los recursos cargados.
- **Escenas**. Especifican tareas lógicas a realizar al inicio de la aplicación o en determinados puntos de la ejecución de la misma y permiten agrupar fácilmente entidades y recursos.
- **Subsistemas**. Son las unidades funcionales que se encargan de procesar los componentes y típicamente el usuario no interactúa con éstos. Sin embargo, existen subsistemas especiales que no procesan componentes y son habitualmente accedidos por el usuario para obtener información; éstos incluyen el sistema de entrada, utilizado por el usuario para leer los valores procesados de los dispositivos de entrada y el sistema de tiempo, que da información de la longitud de los intervalos de actualización y renderizado, tiempo transcurrido desde el inicio de la ejecución de la aplicación, etc.
- **Elementos auxiliares**. Clases que contienen métodos y tipos que proveen funcionalidad útil para el usuario y para el propio *framework*.

De forma simplificada, el usuario crea una escena y en ésta carga recursos, los configura, crea entidades y le agrega componentes, que configura con recursos y otras datos. Probablemente el usuario necesite información del sistema, pero típicamente esto se reduce a obtener información de tiempo y los valores de los dispositivos de entrada. También, es posible que necesite funciones o tipos auxiliares como un *pool* de datos optimizado u operaciones sobre cadenas de caracteres. De esta forma el usuario interactúa con un conjunto de elementos bien identificados y con una interfaz uniforme y predecible. Asimismo, el *framework* se encuentra dividido en datos y unidades funcionales, lo que eleva significativamente la calidad y desempeño del código. Además, por ejemplo, esta modularización permite reemplazar de forma más sencilla la API gráfica sobre la que se apoya el *framework*, dado que las llamadas a esta librería típicamente solo se deben realizar en las unidades funcionales y el área de recursos.

Para mejorar la legibilidad del código, simplificar la interfaz y evitar que el usuario acceda a áreas relacionadas con la operatoria interna del *framework*, se estructuró la solución en capas. La capa de base del *framework*, donde se encuentra toda su funcionalidad y la capa de aplicación donde el usuario codifica su aplicación. Los recursos que forman parte de la operatoria del *framework*, se encuentran en un proyecto de contenido asociado al proyecto de la capa base, mientras que el usuario puede definir su propio proyecto de contenidos asociado al proyecto de la capa de aplicación. Existe una capa intermedia, la de editor, que es opcional y brinda al desarrollador la posibilidad de realizar cambios de los parámetros de su aplicación en tiempo real y bajo una interfaz gráfica.

8.2.1 Administración de Memoria

C# dispone de un manejado de memoria automático que incluye un recolector de basura. El manejo de memoria en C# es eficiente y rápido. Sin embargo, la programación orientada a datos se centra en reducir la penalidad de los accesos a memoria por lo que debieron tomarse ciertas consideraciones en el diseño del software para mejorar la localidad de los datos, en especial cuando se utiliza información por referencia. Además, es de vital importancia mantener estabilidad en la frecuencia de los cuadros por segundo. El ojo humano percibe si en un intervalo de tiempo considerable la imagen en pantalla no es actualizada, aún si la cantidad de cuadros por segundo sigue siendo significativamente alta. La ejecución del recolector de basura podría generar fluctuaciones considerables, por lo que se consideró necesario tomar medidas para evitarlas. En particular, la Xbox 360 sufre considerablemente la recolección de basura dado que no se implementó un esquema con generaciones, lo que significa que al momento de realizar una recolección, el entorno de ejecución inspecciona toda la memoria reservada para la CPU (Schneider & Larrea, Programación Orientada a Datos y Desarrollo Basado en Componentes Utilizando el Lenguaje C#, 2011).

El recolector de basura se ejecuta cuando se crea una cierta cantidad de información nueva en el *heap* desde la última recolección; en Xbox 360 ese valor es de solo un 1Mb. Existen dos enfoques para reducir el costo de su ejecución; se puede reducir la frecuencia o la latencia de ejecución del proceso de recolección. Para reducir la frecuencia se necesita reducir la cantidad de memoria asignada. Si no se aloja información en el *heap* desde la última recolección, el recolector de basura nunca se ejecutará. La otra posibilidad es reducir la complejidad de *heap*. Si el *heap* tiene pocas unidades de datos la recolección se realizará en menor tiempo aún si esos datos contienen mucha información. En la práctica se debe perseguir uno u otro enfoque, pero no se puede obtener un buen desempeño buscando un punto medio entre ambos debido a que las recolecciones en una aplicación que genera poca basura en un *heap* medianamente complejo seguirán siendo notorias (Hargreaves, Shawn Hargreaves Blog).

En este proyecto se decidió reducir la frecuencia de recolección y para lograrlo se diseñó un *pool* de datos con énfasis en la localidad de sus datos, se redujo el uso de los tipos que generan basura, como el tipo *string*, se cambió la interfaz estándar y recomendada para pasaje de mensajes por eventos (tipos por referencia cambiados por tipos por valor) y se evitó recorrer estructuras por referencia usando la interfaz *IEnumerable*.

Se implementó un *pool* de datos que permite guardar y acceder efectivamente a las propiedades de los componentes y otros datos. Esta estructura persigue dos objetivos principales. El primer objetivo es garantizar que los datos estén juntos en memoria. La mejor estructura base para lograr este objetivo es la utilización de un arreglo que solo almacene los datos requeridos. Si un dato se elimina, el último dato activo de la colección pasará a ocupar el lugar dejado por el dato eliminado y de esta manera se mantiene la localidad en memoria. Como segundo objetivo se pretende reducir la frecuencia de ejecución del recolector de basura. Si todos los datos por referencia requeridos están en memoria de antemano pocas creaciones de datos por referencia serán necesarias y por ende el recolector de basura se mantendrá ocioso.

Esta estructura también debe permitir acceder a la información guardada, pedir un nuevo elemento y eliminar uno existente en orden constante. Si los datos almacenados son por referencia, se podría devolver directamente el puntero al repositorio, pero las eliminaciones no serían constantes. En cambio, si los datos son por valor, retornar los datos significa la replicación de los mismos, lo cual presupone un gasto innecesario de memoria y la imposibilidad de modificación de los mismos. Por esta razón se implementaron dos versiones de esta estructura, una que devuelve la referencia al dato y otra que usa una estructura de datos diseñada para acceder a la información almacenada. Esta estructura de datos se modeló utilizando un tipo por referencia. De esta manera, si un objeto cambia de posición por una operación de eliminación, su información de referenciación se mantendrá actualizada automáticamente. También se permite incrementar el tamaño de la estructura dinámicamente. Sin embargo, esta operación es de *O(n).*

En el entorno de ejecución .NET no se puede elegir dónde se almacenarán los datos por referencia. Sin embargo, si estos datos se crean al principio de la ejecución cuando hay poca o nula fragmentación y los elementos existentes hasta ese momento no se eliminan, entonces se podría estimar que estos datos estarán juntos en memoria. Por el contrario, sí se puede asegurar la localidad de los datos por valor.

8.3 Tecnologías Gráficas Implementadas

En esta sección se detallarán las decisiones de diseño y tecnologías gráficas implementadas que dan forma al pipeline gráfico desarrollado para el *framework*. Se comenzará describiendo brevemente el tipo de pipeline gráfico implementado y se explicarán las razones que llevaron a su adopción, se describirán las representaciones de color seleccionadas y el tratamiento que se realiza sobre el mismo, se enumerarán las tecnologías de iluminación global presentes y, por último, se describirán las distintas etapas del pipeline gráfico y el resto de las tecnologías gráficas implementadas que permiten conformar un pipeline fotorrealista.

8.3.1 Pipeline Gráfico

La primera versión del motor incorporaba un simple pipeline *forward renderer*, pero inmediatamente se planteó la necesidad de implementar un pipeline *deferred renderer* dado que permite calcular el aporte de una cantidad arbitraria de luces a un costo computacional reducido, separar en etapas el proceso, lo que simplifica la adopción de tecnologías y organiza mejor el código y elevar el desempeño dado que gran parte de los cálculos se realizan en el espacio de pantalla; de esta manera, por ejemplo, solo se renderizan los BRDF de los materiales en los puntos de superficie visibles desde la cámara.

Específicamente se seleccionó un pipeline *deferred lighting* debido a que tiene un menor requerimiento de memoria de GPU en la generación del *G-Buffer*; esto se adapta mejor al reducido tamaño de la EDRAM de la consola Xbox 360, que por haber sido diseñada previamente a la era de los *deferred*

renderer, su tamaño es reducido (de tan solo 10 Mb) e insuficiente para mantener *G-Buffers* de alta resolución que almacenan gran cantidad de información. Si bien el sistema de *predicated tiling* implementado es transparente y su utilización equivale típicamente a solo una pequeña (pero sensible) reducción de performance, es deseable evitar la ejecución de éste. Además, este pipeline simplifica el agregado de nuevos materiales a costa de procesar por segunda vez los vértices de los objetos de la escena; sin embargo, este costo extra se consideró tolerable, en especial, porque es posible descartar los fragmentos ocluidos de antemano utilizando el mapa de profundidad generado en la etapa de *G-Buffer*.

8.3.2 Color

Los cálculos de iluminación se realizan en espacio lineal y en alto rango dinámico. La información de color de las etapas *light pre-pass* y el renderizado de objetos (previo al operador tonal) se almacena utilizando el formato de *render target HDRBlendable* de XNA que en PC es un formato de punto flotante de 16 bits por canal y en Xbox 360 es un formato de punto flotante de 32 bits, 10 bits para cada uno de los tres primeros canales (7 bits para la mantisa y 3 bits para el exponente) y 2 bits para el canal alfa. En PC, la información de color especular producida en la etapa *light pre-pass* puede almacenarse monocromáticamente en el canal alfa del *render target* donde se almacena la información de color difuso; sin embargo, en Xbox 360 solo se dispone de 2 bits para almacenar esta información; esto resulta insuficiente, por lo que se optó por utilizar una segunda textura con el mismo formato. Para mantener la uniformidad entre plataformas, también se dispuso utilizar en PC una segunda textura para almacenar la información de color especular.

Además, se analizó el utilizar el formato de color RGBM debido a que provee un esquema eficiente de codificación que permite almacenar esta información en un formato sRGB de 32 bits. Sin embargo, dado a que se debe realizar *additive blending* sobre estos *render targets* y a que no es posible realizar la mezcla de dos valores RGBM con las operaciones de *blending* provistas por el hardware, la única solución viable para su uso era utilizar una configuración de dos *render targets* en modo *ping-pong*, lo que duplica el costo de memoria de GPU y produce muchos cambios de estado en la GPU; esto resulta en un mayor costo que el de los formatos de punto flotante. El hardware de PlayStation 3 es muy lento cuando opera sobre formatos de punto flotante, pero permite utilizar el formato RGBM sin un costo adicional dado que se puede leer el contenido del *render target* en el *shader* de fragmentos; por esto es el formato ideal si se desea realizar una translación del *framework* a esta plataforma. No obstante, el formato RGBM resultó ser un formato atractivo para almacenar texturas de alto rango dinámico y se dio soporte para éste en texturas cúbicas.

Se agregó un conjunto de operadores tonales y éstos están a disposición del desarrollador por medio de parámetros a nivel de código de C#. Los operadores tonales agregados son: exponencial, logarítmico, logarítmico de Drago, exponencial de Reinhard, fílmico, Uncharted 2 y Duiker. La exposición de luz de la cámara puede realizarse manualmente o por medio de una función automática de auto-exposición que utiliza como entrada el valor de luminancia promedio de los últimos cuadros. También se agregó la técnica

bloom que permite simular sobre-exposición, en la que la luz muy brillosa parece desbordarse y ocupar el espacio de objetos que se encuentran más cercanos al punto de vista.

La curva de transformación a espacio sRGB implementada en hardware es diferente en computadoras personales (Ecuación 5-5) que en Xbox 360, en la que se realiza una aproximación lineal de 4 segmentos. Por esto se decidió implementar una transformación por software que permite mantener la uniformidad de los resultados. Aunque es recomendable utilizar la transformación especificada en (Ecuación 5-5), se decidió implementar una aproximación que produce resultados muy similares a un costo computacional inferior:

```
return pow(color, 1 / 2.2);
```

Alternativamente se puede usar una aproximación aún más agresiva, que es muy utilizada en la práctica porque provee un balance atractivo entre costo computacional y fiabilidad de los resultados:

```
return sqrt(color); // Equivalente a return pow(color, 1 / 2);
```

Para convertir información en espacio sRGB (colores y texturas) a espacio lineal se utiliza una formulación similar.

Con el objetivo de obtener un mayor control sobre el color producido se agregó un sistema de *color grading* basado en *lookup tables*, que permite mezclar hasta dos de estas texturas, facilitando la transición entre secciones de una escena con distinta iluminación o para escenas de exteriores con variación de clima o transición día-noche. Además, se pueden ajustar los niveles de brillo de la imagen resultante, similar a la herramienta *levels* de Photoshop, aunque se aconseja incluir este mapeo directamente en una *lookup table* y realizar una operación de *color grading*. Es posible, a través de un *shader* creado para este *framework*, simular el ruido conocido como *film grain*, que se produce por la presencia de pequeñas imperfecciones en las películas fotográficas de antiguas cámaras analógicas.

8.3.3 Iluminación Global

Se incorporó una luz ambiental basada en imagen representada con armónicos esféricos de segundo orden, cuyos coeficientes pueden ser calculados automáticamente a partir de un mapa cúbico o mapa latitud-longitud ambiental de alto rango dinámico (Figura 8-2).

También se adaptaron tecnologías de iluminación global en el espacio de la pantalla. Se comenzó implementado un algoritmo sencillo de oclusión ambiental, que luego fue remplazado por uno de mayor calidad visual que utiliza la distribución y propagación de rayos *ray marching*. Se incorporó además el algoritmo *Horizon-Based Ambient Occlusion* que produce muy buenos resultados visuales a un costo computacional ligeramente menor al anterior (Figura 8-2). También se realizaron pruebas con el algoritmo

de oclusión ambiental direccional en espacio de pantalla y su extensión que permite aproximar un algoritmo de radiosidad simple de un rebote de luz.

Estos algoritmos en el espacio de la pantalla, pueden ejecutarse utilizando los mapas de profundidad y de normales de un cuarto de resolución, con el objetivo de mejorar sustancialmente su desempeño y apoyándose en el hecho de que el resultado es información de color de bajo detalle. Debido a que no es correcto promediar los valores de profundidad, se desarrolló un *shader* que selecciona el mayor valor entre los cuatro posibles. Aunque para las normales se requiere un enfoque similar, dado que estos mapas son utilizados para cálculos de bajo detalle y dado que los artefactos que provoca son poco notorios, simplemente se utiliza un promediado por hardware.

8.3.4 Etapas del Pipeline Gráfico

Inicialmente se realiza una operación de *frustum culling* sobre todos los objetos de la escena. Debido a que la arquitectura de software se apoya en la programación orientada a datos resulta más eficiente realizar una técnica de fuerza bruta (Collin, 2011) que complejos algoritmos como el propuesto en (Bittner, Mattausch, & Wimmer, 2009). Además, se consideró implementar un esquema eficiente de *occlusion culling* (Collin, 2011; Valient, Practical Occlusion Culling in Killzone 3, 2011), pero esta tarea no se llevó a cabo porque era muy compleja debido a que el *frustum culling* realiza gran parte del descarte de objetos no visibles, el *Z-Buffer* previamente generado permite evitar el procesamiento de los fragmentos no visibles y las técnicas de *occlusion culling* no son completamente automáticas y pueden requerir tareas de ajuste manuales en la etapa de producción.

La etapa de *G-Buffer* calcula y almacena el mapa de profundidad, el mapa de normales y el valor de poder especular (*glossiness*) del material del objeto. XNA 4.0 estipula que un *Z-Buffer* debe estar asociado a un *render target* (o varios en configuración múltiple) y esta asociación no puede ser alterada, por lo que resulta necesario copiar la información de profundidad del *Z-Buffer* en un *render target* dedicado que puede ser accedido en otras etapas del pipeline; esto impide aprovechar la posibilidad de utilizar el contenido de un *Z-buffer* en etapas posteriores, una funcionalidad implementada a partir del hardware con soporte para *shader model 4* o superior (Thibieroz N. , Ultimate Graphics Performance for DirectX 10 Hardware, 2008). Si bien con este esquema es posible almacenar la información de profundidad en el espacio lineal, lo que simplifica la manipulación de estos valores, se debe utilizar un *buffer* extra que típicamente requiere entre 24 y 32 bits por pixel; a esto se le debe sumar una pérdida de eficiencia por escribir un *buffer* extra y la posibilidad de prescindir de alguna optimización por hardware relacionada con el almacenamiento u operación del *Z-Buffer*. El tamaño de la EDRAM de la Xbox 360 impone indirectamente restricciones en la cantidad de información que puede ser almacenada en el *G-Buffer*, por lo que, por ejemplo, anexar la información de albedo de los objetos visibles por la cámara podría provocar la ejecución de una operación de *predicated tiling* y, por consiguiente, una disminución de la performance.

Figura 8-2 Capturas de pantalla utilizando el *framework*. La primera imagen calcula sólo iluminación local, la segunda incorpora iluminación ambiental por armónicos esféricos y la última incluye además oclusión ambiental en espacio de la pantalla.

En la implementación realizada se almacena la información de profundidad en un canal de 32 bits, pero se analizó almacenar la información de profundidad en tres canales de 8 bits y utilizar los 8 restante para una máscara que permita mantener y mezclar dos luces ambientales. Típicamente esta información se almacena en el *Z-Buffer*, el cual tiene operaciones por hardware para almacenar y leer eficientemente los valores de profundidad y operaciones esténcil que permiten la generación de la máscara de forma sencilla y eficiente. Asimismo se planteó como alternativa realizar una *Z-Pre Pass*, para generar el mapa de profundidad por separado y disponer información temprana para descartar fragmentos en el *G-Buffer*, reduciendo el tamaño de éste a expensas de procesar una vez más cada uno de los vértices de los objetos opacos renderizados. Se generó un *shader* que lee información de un mapa de profundidad y a partir de éste genera un *Z-Buffer*; hasta el momento no se pudo realizar una prueba que evalúe el costo de separar en dos etapas la generación del *G-Buffer*.

El mapa de normales se almacena utilizando el método de compresión *best-fit normals* (Kaplanyan A. , CryENGINE 3: Reaching the Speed of Light, 2010) que permite almacenar esta información en tan solo 24 bits con una precisión aceptable y que solo requiere una operación de normalización para su descompresión. Los 8 bits restantes se utilizan para almacenar el poder especular y se comprimen utilizando el método propuesto en (Valient, Deferred Rendering in Killzone 2, 2007).

Como trabajo futuro se plantea almacenar adicionalmente en el *G-Buffer* el color albedo de las superficies, lo que permitiría adaptar algoritmos de iluminación global dinámicos; debido a que es deseable evitar el *predicated-tiling* que realiza la Xbox 360 cuando la EDRAM no puede contener los *buffers*, se deberá desacoplar previamente la generación del mapa de profundidad del *G-Buffer*, aprovechando en el proceso la posibilidad de hacer un *Z Pre-pass*.

En la etapa *light pre-pass*, se renderizan las luces utilizando volúmenes apropiados y con la posibilidad de utilizar volúmenes definidos por el usuario para evitar *light bleeding*. Además, se crean máscaras utilizando el *Z-Buffer* junto con operaciones esténcil para evitar la generación de fragmentos innecesarios. Como se introdujo anteriormente, XNA 4.0 no permite desasociar el *Z-Buffer* del *G-Buffer*, pero es posible recrearlo utilizando un *shader* de espacio de pantalla que utiliza la información de profundidad del *G-Buffer*. En la implementación realizada en el *framework* se decidió usar la versión inversa del esquema de renderizado de luces (sección 7.3.1), en la que el caso especial se encuentra cuando se intercepta el plano lejano de la cámara. De esta manera se eliminan los casos especiales, a costa de realizar dos pasadas por luz cuando la luz intercepta la cámara, momento en el cual típicamente el radio de influencia de la luz cubre la mayor parte del espacio de pantalla visible.

El *framework* soporta mapas de sombras para luces direccionales y *spot*, *cascaded shadow maps* para luces direccionales y mapas de sombras cúbicas para luces puntuales. Las sombras se calculan de forma diferida, lo que permite aplicar filtros en espacio de pantalla a un costo computacional menor. Es posible que la técnica de reducción de fragmentos antes propuesta haga atractiva el realizar el filtrado directamente en el *shader* de iluminación de la luz; sin embargo, dado que la generación del mapa de sombras se descarta para luces puntuales y *spot* lejanas y dado que el resto de las luces cubren un área significativa en espacio de pantalla se decidió mantener el esquema diferido por simplicidad y eficiencia. En esta etapa también se ejecutan, de forma diferida, los algoritmos de iluminación global de tiempo real.

En la etapa de renderizado de objetos se renderizan los objetos opacos utilizando la información de iluminación de la etapa anterior y ejecutando uno de los BRDFs implementados. La variedad de BRDF está limitada por la cantidad de información de iluminación disponible derivada de la etapa anterior, aunque puede ser sorteada realizando el cálculo del BRDF en esquema *forward*. Luego se renderiza el cielo (este orden de renderizado permite descartar fragmentos innecesarios), los sistemas de partículas, los objetos transparentes (en esquema *forward*), las líneas, los textos y los *sprites* en el espacio del mundo y en el espacio lineal. El sistema de partículas tiene varios parámetros de control (duración, dirección de gravedad, tamaño inicial y final, etc.), permite renderizar partículas duras o suaves (Rosado, 2009) y texturas animadas (en esquema de mosaicos).

La etapa final de post procesamiento realiza la calibración del color, la adaptación dinámica de la exposición, el mapeo tonal, *bloom*, *color grading*, ajuste de niveles de brillo y *film grain*. También ejecuta un filtro *MLAA* en espacio de pantalla que reduce el *aliasing* de la imagen producida y que puede utilizar la información de profundidad, la información de color o ambas. Por último se renderizan las líneas, los textos y los *sprites* en el espacio de mundo y en el espacio de color gamma y se renderizan las líneas, los textos y los *sprites* en el espacio de la pantalla y en el espacio de color gamma.

8.4 Editor

A partir de la restructuración del *framework* y paralelo al desarrollo de éste, se implementó un editor gráfico inspirado en Unity3D. Este editor soporta, entre otras cosas:

- Selección múltiple de objetos.
- Transformaciones de escalado, rotación y posición (local y global) por medio de gizmos gráficos o la introducción manual de los valores.
- Creación, modificación o eliminación de recursos artísticos, componentes y entidades.
- Visualización de la escena en vistas ortográficas, en perspectiva o la visión final de la escena.
- Manipulación rápida de estas cámaras.

Todo esto puede hacerse a través de una potente interfaz de usuario, que soporta operaciones de deshacer y rehacer, entre otras facilidades (Figura 8-3).

Con el objetivo de evitar la reducción de la calidad del código fuente del *framework*, el editor se encuentra en una capa superior a éste y solo puede interactuar con las mismas interfaces disponibles para el usuario. Asimismo, el editor ayuda a comprobar la calidad de la interfaz, además de ser una excelente herramienta para encontrar errores de programación.

8.5 Desarrollos en esta Plataforma

En este *framework* se han desarrollado un conjunto de pequeños proyectos con diferentes objetivos. Paralelo al desarrollo del motor se generó una escena de prueba que permite analizar en un escenario con alta carga poligonal el desempeño y resultado visual de las tecnologías incorporadas. También se seleccionaron alumnos de la materia Computación Gráfica que mostraron gran entusiasmo en aprender nuevas tecnologías, se los dividió en equipos y así realizaron el trabajo final de la misma sobre este *framework*. Además, se realizó un prototipo comercial para un videojuego multiplataforma.

Figura 8-3 Capturas del editor gráfico del *framework*.

Estos proyectos permitieron analizar y mejorar los distintos elementos del *framework*, mejorando la usabilidad, el desempeño computacional y la funcionalidad brindada. También fueron un medio útil para realizar una mejor depuración de errores y encontrar las principales limitaciones que impone.

8.5.1 Escena de Prueba

La escena de prueba consiste en un modelo de un vehículo de aproximadamente 400.000 vértices y un modelo sencillo de una construcción que contiene 14.000 vértices; se aplicaron las técnicas desarrolladas, evaluando su calidad visual y su desempeño (Figura 8-4). Sin embargo, esta escena no permite evaluar con precisión el comportamiento de las tecnologías para escenas de gran distancia.

Figura 8-4 Capturas de pantalla de la escena de prueba diseñada para comprobar la calidad de los distintos elementos del *framework*. El modelo del vehículo contiene aproximadamente 400.000 vértices, y la construcción contiene aproximadamente 14.000 vértices. En esta escena se pueden apreciar *Cascaded Shadow Maps*, *Horizon Based Ambient Occlusion*, mapeo tonal fílmico, adaptación automática de la exposición, iluminación ambiental por medio de armónicos esféricos de segundo orden, entre otras tecnologías gráficas.

Se han ejecutado herramientas de evaluación de desempeño sobre esta escena; particularmente se ejecutó la herramienta PIX y la herramienta GPU View, que permiten encontrar primordialmente un exceso de llamadas al sistema y de cambios de estados en la GPU, y la creación excesiva de recursos. Una parte de estos problemas se solucionan por la propia plataforma de desarrollo, gracias al manejo de estados implementado por hardware en las GPUs con *shader model 4* o superiores y a la administración de estados brindada por *XNA 4.0*, que evita de forma transparente los cambios de estados innecesarios y que permite usar el esquema de estados de estas GPUs. No obstante, fue necesario ordenar el renderizado de objetos por *shader,* es decir, renderizar en lote (*batching*). También se fusionaron diferentes variantes de un mismo *shader* en un único *shader* (modelo *uber shader*) dado que se concluyó que para éstos, el costo computacional de ejecutar una pequeña cantidad de saltos condicionales es menor que el costo de preparar

la GPU para la ejecución de otra versión del *shader;* además, simplifica las tareas de ordenamiento y *batching*.

8.5.2 Trabajos Finales de Materia

En el primer cuatrimestre del año 2010, dos grupos de alumnos de la materia Computación Gráfica, perteneciente al plan de Ingeniería en Sistemas de Computación del DCIC, Universidad Nacional del Sur, realizaron el trabajo final de la materia utilizando este *framework*. Éste consistió en la creación de un juego utilizando los conceptos básicos aprendidos en la materia, los que debían complementar incluyendo además el uso de dispositivos no convencionales, *shaders* no vistos previamente, sonido, música, etc. y estando el tiempo de desarrollo acotado a 3 semanas y con fecha fija de entrega.

Ambas comisiones estaban integradas por tres alumnos y los juegos elegidos fueron un juego de disparos llamado *Red Fire* y un juego de autos llamado *Threepwood Wii Circuit 2010* (Figura 8-5). En ambos proyectos se utilizó el Wiimote como dispositivo de entrada. El primero utilizó el Nunchuck para los movimientos de traslación del personaje y la cámara infrarroja para apuntar; el segundo utilizó los acelerómetros para direccionar el volante. Se decidió suministrar la mayor parte de los recursos gráficos utilizados en estos proyectos (modelos y texturas) para evaluar mejor el desempeño en programación de los grupos sobre el *framework*. Además, en ese momento los alumnos contaron con una versión alfa del motor estable pero significativamente limitada funcionalmente. No incluía, por ejemplo, sistemas de física y de partículas.

Esta prueba temprana mostró la existencia de problemas de interfaz y desempeño, que derivaron en la restructuración del *framework*. Además, permitió analizar las tecnologías gráficas incorporadas hasta ese momento, lo que derivó en la modificación y mejora de varias de éstas.

En el primer cuatrimestre del año 2011 se repitió la experiencia con un solo grupo de alumnos, pero esta vez se suministró una versión del *framework* que contaba con la nueva estructura de software y nuevas tecnologías gráficas. El juego creado fue un juego de aviones que denominaron *Cóndor Flight* y para éste se desarrolló un sistema de terrenos inspirado en el juego de aviones H.A.W.X. 2 de Ubisoft (Figura 8-6).

La interacción de los alumnos con el *framework* fue más fluida y menos problemáticay sólo fue necesario realizar posteriormente un conjunto de ajustes menores. El desempeño global también fue superior; aunque se encontraron caídas de desempeño (particularmente en el sistema de terrenos). Se utilizaron las herramientas PIX y GPU View que al ejecutarse sobre este proyecto y la escena de prueba del *framework* permitieron encontrar una ineficiencia producto de la falta de ordenamiento del renderizado de los objetos.

(a) (b)

Figura 8-5 Capturas de pantalla de (a) *Threepwood Wii Circuit 2010* y (b) *Red Fire*.

Figura 8-6 Sistema de terrenos implementado para el proyecto *Cóndor Flight*.

8.5.3 Prototipo Comercial

En 2012 se realizó un pequeño prototipo comercial (Figura 8-7) denominado *Extraction* a cargo de una empresa canadiense, que buscaba analizar la viabilidad de un sistema de juego de suspenso y disparos en tercera persona, y evaluar al mismo tiempo XNA y C# y el propio *framework* como herramienta de desarrollo multiplataforma para un futuro juego de estas características.

Este proyecto brindó una realimentación que permitió continuar mejorando detalles en la interfaz y realizar una mayor depuración del código, entre otras mejoras. El proyecto requirió además extensiones en el sistema de animación para incluir un sistema avanzado de mezcla de animaciones; desafortunadamente, el sistema de animación no pudo completarse y solo se realizaron las tareas necesarias para este proyecto. Para este proyecto se analizó además optimizar la compresión de la información de *skinning animation* (Frey & Herzeg, 2011), brindando la posibilidad de utilizar un alto número de *bones* y aumentando al mismo tiempo la eficiencia de los *shaders*, pero estas tareas fueron descartadas en favor del sistema de mezcla de animaciones y debido a que una simple compresión de la información resultó suficiente para los requerimientos del prototipo.

También fue necesario representar un mundo físico, por lo que se creó una sencilla interfaz con el objetivo de interactuar con la librería Bepu, una librería de física desarrollada sobre XNA que brinda un funcionamiento robusto y completo.

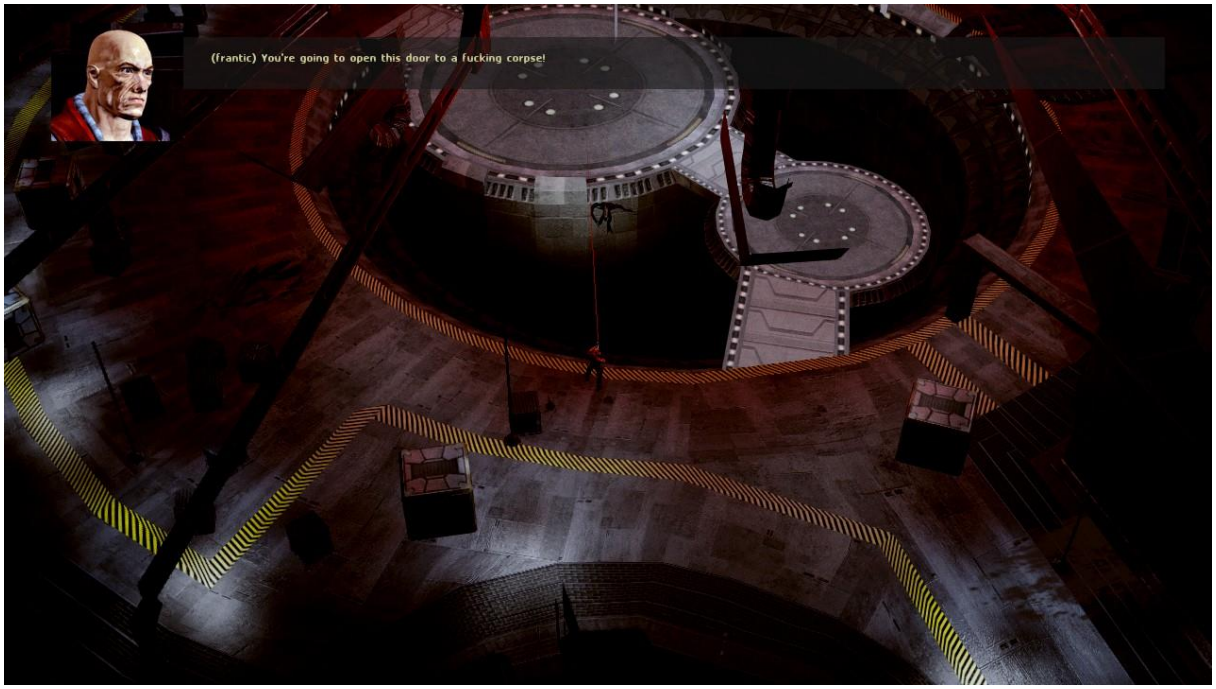


Figura 8-7 Captura de pantalla de *Extraction*, un prototipo comercial realizado sobre el *framework*.

8.6 Análisis de Desempeño

El análisis de desempeño realizado se enfocó en evaluar el desempeño de las distintas etapas del pipeline gráfico implementado, dejando de lado el análisis de los sistemas de audio, de animaciones, de física y de lógica. Éste se realizó sobre la plataforma Xbox 360 y sobre la plataforma PC utilizando una GPU GeForce 560Ti de 384 *stream processors* y un procesador AMD Phenom II X4 965 de 4 núcleos (3.6Ghz). La resolución elegida fue 1120x630 pixeles debido a que no genera *predicated tiling* en la consola Xbox 360.

Dado que se desea medir principalmente el desempeño gráfico, se utiliza como medida los cuadros por segundo, si bien ciertas etapas, como *frustum culling*, hacen uso intensivo de la CPU.

Para los distintos test realizados se tomó como base la escena de prueba anteriormente descripta, pero se hicieron modificaciones con el objetivo de medir con mayor precisión el costo computacional de ciertos algoritmos o técnicas. Además, para obtener medidas más precisas se realizaron ejecuciones parciales del pipeline gráfico; por ejemplo, al evaluar el desempeño de la pasada *light pre-pass*, se ejecuta el algoritmo de *frustum culling*, la generación del *G-Buffer*, la etapa *light pre-pass* y la etapa que renderiza los elementos 2D, pero no las etapas de renderizado de objetos y de post procesado. La renderización de los elementos 2D presupone una sobrecarga necesaria para mostrar información estadística entre la que se incluyen los cuadros por segundo.

Frustum Culling

En la primera etapa del pipeline gráfico se realiza una operación de *frustum culling* que ejecuta, por cada objeto activo, una operación de intersección entre el *frustum* de la cámara y el *bounding sphere* del objeto. Este algoritmo se diseñó en modo *multithreading* sin necesidad de sincronización entre los hilos, se reemplazó la operación de intersección provista por *XNA* por una implementación altamente optimizada, se evitó toda generación de basura y el algoritmo y los datos suministrados se organizaron de manera de reducir a un mínimo los accesos a memoria. En este caso se obtienen los siguientes resultados para una escena que contiene 6708 objetos activos:

Descripción del test	PC	Xbox 360
Frustum Culling sobre 6708 objetos activos	1010 cuadros por segundos	145 cuadros por segundo

La cantidad de objetos presentes en esta escena de prueba es muy alta y la mayoría de las aplicaciones gráficas no alcanzan valores tan altos. Estos resultados muestran que la plataforma PC no se ve afectada significativamente por esta tarea, sin embargo, la consola Xbox 360 en escenarios similares requerirá la ejecución previa de un sistema de niveles de detalle para descartar con mayor rapidez objetos no visibles, en especial, si la aplicación necesita realizar complejas tareas de física o lógica.

Generación del *G-Buffer*

Con el objetivo de reducir las llamadas al sistema y la cantidad de veces que la GPU cambia de programa de vértices y de fragmentos la generación del *G-Buffer* se realiza utilizando solo 5 técnicas de *shaders*. Para esto es necesario calificar los objetos visibles en sus respectivas técnicas para ejecutar todos

los *shaders* en lote. Ejecutando la aplicación de prueba y ejecutando el pipeline gráfico hasta esta etapa se obtiene (Figura 8-8):

Descripción del test	PC	Xbox 360
Generación del *G-Buffer*	1132 cuadros por segundos	218 cuadros por segundo

Estos resultados muestran que se alcanzó un muy buen desempeño en esta etapa. Dado que los objetos se renderizan en lote por técnica y los *shaders* son relativamente simples, la principal limitación de esta etapa se encuentra en la cantidad de vértices a procesar y la cantidad de objetos presentes.

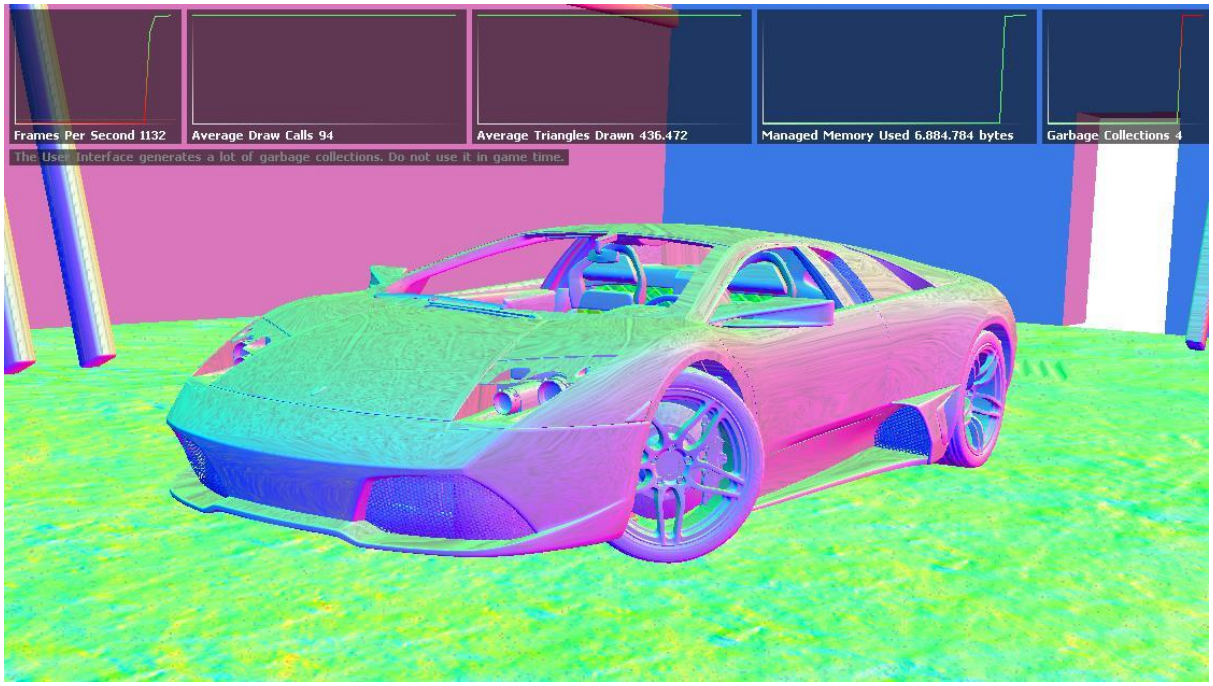


Figura 8-8 Ejecución en la plataforma PC del pipeline gráfico hasta la etapa G-Buffer. El mapa de normales es mostrado en pantalla.

Light Pre-Pass

Esta etapa típicamente es la que más tiempo de procesamiento consume debido a que se procesan todas las luces, las sombras y los algoritmos de iluminación global. Para obtener una mejor lectura del costo computacional de esta etapa se realizaron diferentes test con diferentes configuraciones de iluminación. Ejecutando la aplicación de prueba y ejecutando el pipeline gráfico hasta ésta etapa se obtiene:

Descripción del test	PC	Xbox 360
Luz ambiental (utilizando armónicos esféricos)	823 cuadros por segundos	139 cuadros por segundo
Luz ambiental (utilizando armónicos esféricos) + oclusión ambiental (HBAO)	189 cuadros por segundo	16 cuadros por segundo
Luz direccional	830 cuadros por segundo	139 cuadros por segundo
Luz direccional + sombras cascaded (4 cascadas)	614 cuadros por segundo	98 cuadros por segundo
Luz puntual	1010 cuadros por segundo	170 cuadros por segundo
Luz puntual + sombras cúbicas	660 cuadros por segundo	80 cuadros por segundo
Luz ambiental (utilizando armónicos esféricos) + oclusión ambiental (solo PC) + luz direccional (con sombras cascaded) + 2 luces puntuales + 1 luz spot	151 cuadros por segundo	66 cuadros por segundo

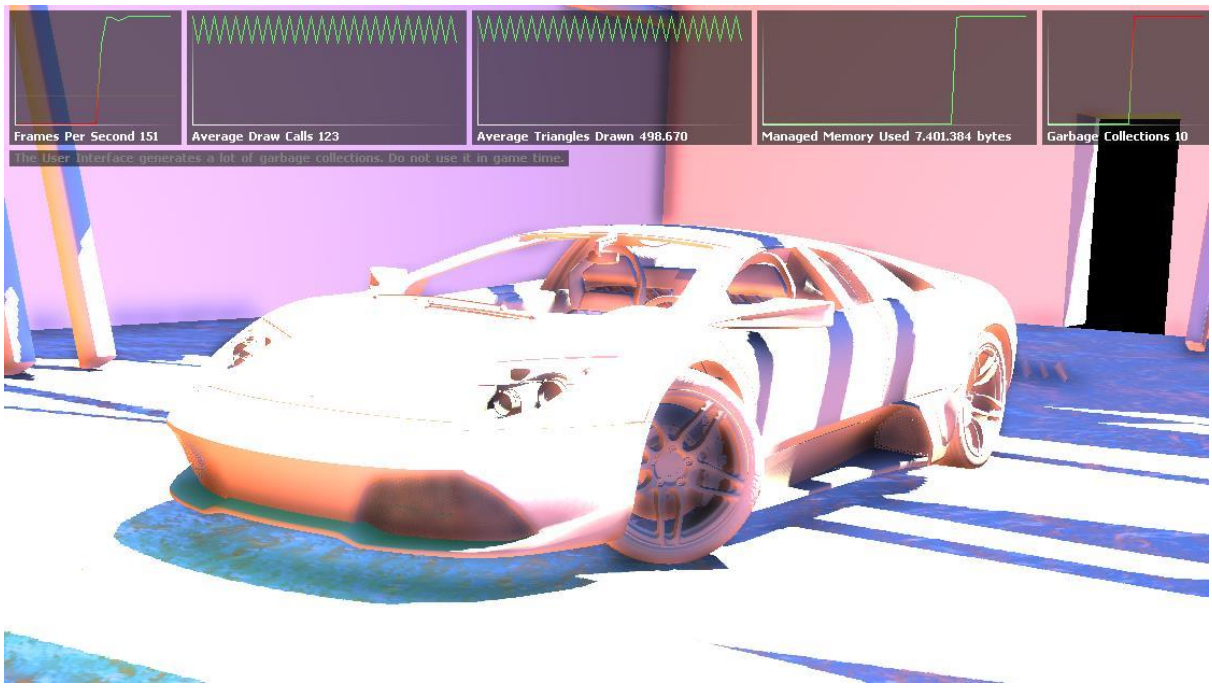


Figura 8-9 Ejecución en la plataforma PC del pipeline gráfico hasta la etapa *light pre-pass*. En este test se procesan 1 luz ambiental (utilizando armónicos esféricos y oclusión ambiental), 1 luz direccional (con sombras cascaded), 2 luces puntuales y 1 luz *spot*. La contribución difusa es mostrada en pantalla.

La luz ambiental es computacionalmente costosa por ser un *shader* que se ejecuta sobre toda la pantalla, no obstante, incrementa significativamente la calidad visual debido a que genera una aproximación sencilla de iluminación global. La oclusión ambiental, por el contrario, reduce

significativamente los cuadros por segundo al punto de resultar prohibitiva para la consola Xbox 360 y GPUs de bajo rendimiento, por lo que se aconseja implementar un esquema de coherencia temporal o prescindir de esta técnica.

La luz direccional también requiere la ejecución de un *shader* que abarca toda la pantalla y por lo tanto se recomienda utilizarla solo para representar la luz solar. Las sombras *cascaded* permiten representar efectivamente sombras en escenas de exteriores a un costo computacional razonable. Las luces puntuales no representan una sobrecarga significativa y es posible representar una gran cantidad de estas luces; sin embargo, las sombras cúbicas reducen significativamente los cuadros por segundo y se recomienda activarlas solo cuando se considere estrictamente necesario.

El último test (Figura 8-9) muestra que una iluminación de mediana o alta complejidad es viable aún en la consola Xbox 360, pero se recomienda ignorar algoritmos de iluminación global en el espacio de la pantalla para hardware de bajo rendimiento.

Renderizado de Objetos

Similarmente a la generación del *G-Buffer*, se intenta reducir las llamadas al sistema y la cantidad de veces que la GPU cambia de programa de vértices y de fragmentos; para esto se califican nuevamente los objetos visibles en técnicas y se renderiza por lote. Para obtener una mejor lectura del costo computacional de esta etapa se realizaron diferentes test cada uno renderizando diferentes tipos de objetos. Ejecutando la aplicación de prueba (con la misma iluminación que el último test de la etapa anterior) y ejecutando el pipeline gráfico hasta ésta etapa se obtiene:

Descripción del test	PC	Xbox 360
Renderizado de objetos opacos	138 cuadros por segundos	47 cuadros por segundo
Renderizado de objetos opacos y de 150 partículas suaves	96 cuadros por segundo	29 cuadros por segundo
Renderizado de objetos opacos, de 150 partículas suaves y de objetos transparentes	96 cuadros por segundo	29 cuadros por segundo

El renderizado de objetos opacos tiene una sobrecarga que se ve influenciada principalmente por la complejidad poligonal y la cantidad de objetos a renderizar debido a que los objetos se renderizan en lote por técnica y los *shaders* son relativamente simples. Las partículas implementadas generan una gran cantidad de fragmentos y si varias de éstas cubren un área extensa de la pantalla, el desempeño se verá afectado significativamente. El renderizado de objetos transparentes (solo unos pocos polígonos en esta escena) es eficiente pero está influenciado sólo por un conjunto reducido de luces locales.

Post Procesado

Para obtener un mejor desempeño computacional se fusionaron varios algoritmos (calibración, exposición, mapeo tonal, *film grain* y *color grading*) en un único *shader*. Ejecutando la aplicación de prueba y ejecutando el pipeline gráfico completo se obtiene (Figura 8-10):

Descripción del test	PC	Xbox 360
Mapeo tonal + *bloom* + *film grain* + *color grading*	94 cuadros por segundos	26 cuadros por segundo
Mapeo tonal + *bloom* + *film grain* + *color grading* + *MLAA*	93 cuadros por segundo	24 cuadros por segundo

La etapa de post procesado tiene un costo computacional razonable, aportando técnicas gráficas que son muy valiosas a la hora de incrementar el fotorrealismo de la escena.

Figura 8-10 Ejecución en la plataforma PC del pipeline gráfico completo utilizando la escena de prueba; este renderizado incluye *MLAA*.

Todos estos tests muestran que es posible reproducir gráficos fotorrealistas en *XNA Final Engine* aún en plataformas como la consola Xbox 360; sin embargo, los algoritmos de iluminación global en espacio de pantalla son prohibitivos en la mayoría de las escenas y se recomienda, con el objetivo de aumentar el fotorrealismo, utilizar sólo luz ambiental basada en imagen en conjunto con un gran número de luces locales.

Capítulo 9 Conclusiones y Trabajo Futuro

El creciente poder de procesamiento de las GPUs y el gran número y potencialidad de las tecnologías gráficas de tiempo real contemporáneas permiten representar escenas complejas con una calidad gráfica cada más cercana al fotorrealismo. Los avances recientes en representación y tratamiento de color, algoritmos de iluminación global de tiempo real y pipelines de iluminación locales permiten representar una gran cantidad de fenómenos visuales, que al complementarse con otras tecnologías gráficas tales como el renderizado de volúmenes y la dispersión atmosférica, permiten renderizar gráficos en tiempo real que simulan convincentemente ser capturas del mundo real.

El relevamiento realizado en este trabajo introdujo un conjunto completo de representaciones y algoritmos que permiten establecer efectivamente las bases de un motor gráfico fotorrealista de tiempo real para el renderizado de superficies. Específicamente se relevó:

- **Color**: Se describió el espacio de color de color RGB debido a que es el formato de color utilizado en pantallas. Además, se introdujeron las transformaciones gamma, el espacio de color sRGB y se discutieron las técnicas más populares de compresión para formatos de color RGB de bajo rango dinámico.

  Dado que representar el color en alto rango dinámico es muy importante si se tiene como objetivo lograr un renderizado fotorrealista, se introdujeron las principales opciones a la hora de adoptar un formato de color de alto rango dinámico, considerando la precisión, rango y costo computacional de estos formatos. Dado el bajo rango dinámico de los dispositivos de display, una representación de color de alto rango dinámico debe ser convertida a un sistema de color RGB de bajo rango dinámico. Dado que esta conversión es compleja e influencia significativamente la calidad de los resultados, se introdujo un conjunto de técnicas para llevar a cabo dicha tarea de manera satisfactoria.
- **Iluminación Global de Tiempo Real**: Se introdujeron los principales algoritmos de iluminación global de tiempo real, desde los enfoques más simples hasta los más complejos, incluso aquellos que consideran el dinamismo de los objetos y la iluminación. Se comenzó describiendo la luz ambiental y los métodos más eficientes para almacenar información de bajo detalle, entre los que se destacan los armónicos esféricos. Luego se introdujo la oclusión ambiental, la oclusión direccional y un conjunto de técnicas de iluminación global más complejas que van desde la técnica menos adaptable, mapas de luces, pasando por técnicas semidinámicas, volúmenes irradiantes y transferencia de radiancia precomputada, hasta abarcar las técnicas que consideran el dinamismo de los objetos y la iluminación derivadas del algoritmo general introducido en radiosidad instantánea.

- **Pipelines Gráficos**: Se describió brevemente el pipeline general implementado en el hardware de las GPUs, para luego realizar un relevamiento de los pipelines gráficos de iluminación local contemporáneos más importantes y utilizados que se ejecutan eficientemente bajo este hardware.

Las tecnologías presentadas pueden ser mutuamente excluyentes o complementarias, supliendo distintas necesidades; por lo tanto, el desarrollador de la aplicación gráfica debe analizar qué fenómenos visuales o características tienen mayor importancia al momento de representar las escenas y, de esta manera, poder seleccionar correctamente las tecnologías más adecuadas. En este trabajo se brinda al desarrollador una descripción de las técnicas gráficas más importantes referentes a representación y tratamiento de color, algoritmos de iluminación global de tiempo real y pipelines de iluminación locales. Esto lo ayudará a obtener una perspectiva de estas en cuanto a características como la precisión de los resultados, la completitud de los fenómenos visuales que intentan modelar, la capacidad de adaptarse a cambios en la escena, la complejidad de los algoritmos, el costo de procesamiento y el consumo de memoria.

Adicionalmente se desarrolló un caso de estudio, el *framework* XNA Final Engine, con el objetivo de contar con una herramienta que permitiera analizar y evaluar prácticamente varias de las tecnologías presentadas, así como también observar cómo éstas se interrelacionan con el objetivo de generar fotorrealismo. Este *framework* también reveló ser una herramienta valiosa para el aprendizaje, como se pudo apreciar en los trabajos finales de la materia Computación Gráfica, permitiendo a grupos de alumnos desarrollar juegos gráficamente atractivos en un tiempo reducido y utilizando tecnologías innovadoras. Además, dada su arquitectura de software, este sistema brinda una organización flexible y mantenible que permite incorporar y evaluar nuevas tecnologías y que podría ser de vital importancia en futuras investigaciones desarrolladas en el VyGLab.

Adicionalmente, el *framework* desarrollado fue alojado en el sitio de proyectos abiertos CodePlex con el objetivo de compartirlo con la comunidad científica y de desarrolladores independientes. Esta experiencia demostró ser valiosa para el desarrollo del mismo dado que se obtuvo una importante realimentación que permitió mejorar aspectos puntuales de diseño y de funcionamiento.

El desarrollo de esta tesis dio lugar a múltiples publicaciones: (Schneider & Larrea, Programación Orientada a Datos y Desarrollo Basado en Componentes Utilizando el Lenguaje C#, 2011; Schneider, Martig, & Castro, Framework XNA Final Engine, Análisis de su Aplicación para el Desarrollo de Juegos y Aplicaciones Gráficas Innovadoras, 2010; Schneider, Martig, & Castro, Framework de Realidad Aumentada Integrando Gráficos de Alta Complejidad, 2010; Schneider, Martig, & Castro, Framework para la Generación de Aplicaciones Gráficas Integrando Realidad Aumentada y Gráficos de Alta Complejidad, 2010; Schneider, Martig, & Castro, Realidad Aumentada en un Entorno Gráfico de Alta Performance, 2009).

9.1 Trabajo Futuro

A partir del relevamiento realizado surgen potenciales líneas de trabajo. A continuación se mencionan las más relevantes:

Mapeos Tonales. En tiempo real, hasta el momento, sólo se utilizan mapeos tonales globales sencillos o de mediana complejidad. En este contexto, sería de gran utilidad lograr implementaciones eficientes de mapeos tonales más complejos que modelen con mayor exactitud el comportamiento del sistema visual humano. Como un primer paso, se podrían realizar implementaciones de mapeos tonales locales derivados de trabajos como el operador local de variación espacial de Chiu (Chiu, Herf, Shirley, Swamy, Wang, & Zimmerman, 1993), el operador retinex de Rahman y Johnson o el operador local desarrollado por Pattanaik denominado modelo del observador multiescala (Pattanaik, Ferwerda, Fairchild, & Greenberg, 1998).

Radiosidad Instantánea. El modelo general de radiosidad instantánea es una técnica que podría utilizarse como base para buscar mejores implementaciones de algoritmos de iluminación global completamente dinámicos. Particularmente, la técnica volúmenes de propagación de la luz provee un punto de partida atractivo dado que es una solución de tiempo real que demostró ser efectiva, aún en escenas de gran complejidad. Es posible analizar e implementar extensiones con el objetivo de mejorar la resolución y precisión de los cálculos e incorporar interacciones especulares.

Pipelines Gráficos. Ciertamente es posible analizar nuevas alternativas o extensiones a los pipelines de iluminación local ya presentados. No obstante, podría resultar de mayor interés realizar implementaciones innovadoras de pipelines de iluminación global como los presentados en la sección 4.5, analizando si las ventajas que brinda la GPGPU podrían ser beneficiosas a la hora de generar implementaciones más eficientes.

Apéndice A Programación Orientada a Datos y Desarrollo Basado en Componentes

Una de las tareas más importante en computación gráfica de tiempo real es la de optimizar el sistema computacional subyacente. Es típico encontrar trabajos que únicamente optimizan o simplifican las técnicas anteriormente presentadas, con el objetivo de hacerlas viables en tiempo real. El desempeño de una aplicación gráfica no siempre está ligado con la optimización de los algoritmos de forma aislada. Uno de los principales cuellos de botella de estas aplicaciones se encuentra sumamente influenciado por la organización y estructura del software de la aplicación. Además, conocer estos cuellos de botella permite realizar una evaluación más precisa del desempeño de las técnicas individuales.

La industria del videojuego produce productos cada vez más complejos, por lo que se tornó una necesidad el encontrar formas de organizar el código que permitan producir código más modular, mantenible, legible y paralelizable. Por esta razón se comenzó a definir y adoptar la programación orientada a datos y el desarrollo basado en componentes, que no solo proveen un paradigma de desarrollo con estas ventajas, sino que también permiten mejorar significativamente el desempeño del sistema computacional para aplicaciones gráficas complejas.

Por esto se propone identificar y aislar las tareas (o procesamientos de alto nivel) y los datos que se acceden y manipulan en estas tareas, es decir, organizar en términos de datos no de entidades, y utilizar un modelo de agregación en vez de uno de especialización para definir el comportamiento de las entidades de la escena.

En este capítulo se analizará por qué la programación orientada a objetos no es un paradigma adecuado para el desarrollo de aplicaciones gráficas complejas de tiempo real y se describirán las principales limitaciones del hardware que condicionan estas aplicaciones. Se presentará la programación orientada a datos que se centra en los datos y su flujo y que permite sortear estas limitaciones y se introducirá el desarrollo basado en componentes que brinda una estructura clara para agrupar un conjunto de datos en entidades. Debido a que las bases de este paradigma no están claramente definidas se introducirá un conjunto de presentaciones y artículos que ayudaron a establecer los lineamientos de este paradigma (Kuznetsov; Fermier, 2002). Por último, se describirán las tareas de alto nivel a realizar con el objetivo de producir una aplicación gráfica siguiendo estos principios.

A.1 Programación Orientada a Objetos y Aplicaciones Gráficas

La programación orientada a objetos (POO) permite, y en parte condiciona, pensar un problema en términos de objetos y sus interacciones. Cada objeto encapsula su funcionalidad y sus datos, y además provee una interfaz bien definida para acceder a ambos. En esencia, cada objeto puede percibirse como una caja negra. Con la POO se favorece el reuso, la mantenibilidad y la usabilidad dado que se abstraen los detalles de implementación de un objeto hacia al resto.

En aplicaciones gráficas se accede a un gran volumen de datos y con gran frecuencia, por lo que resulta necesario centrarse en los datos y en su flujo. El uso de una estrategia de POO en este marco puede llegar a ser contraproducente. Una de las razones radica en que se realiza indirectamente un mal manejo de la memoria *caché* y actualmente la memoria principal (RAM) es muy lenta en comparación al CPU; en particular, la latencia de acceso a la RAM es muy lenta (Figura A-1). En 1980 la latencia de acceso era de 1 ciclo de procesador, pero en la actualidad ronda los 400 ciclos o más (Hennessy & Patterson, 2007). En consolas, por ejemplo, la latencia de acceso podría alcanzar los 600 ciclos, tiempo en el cual se pueden realizar, por ejemplo, 20 multiplicaciones de matrices (Torp, 2010).

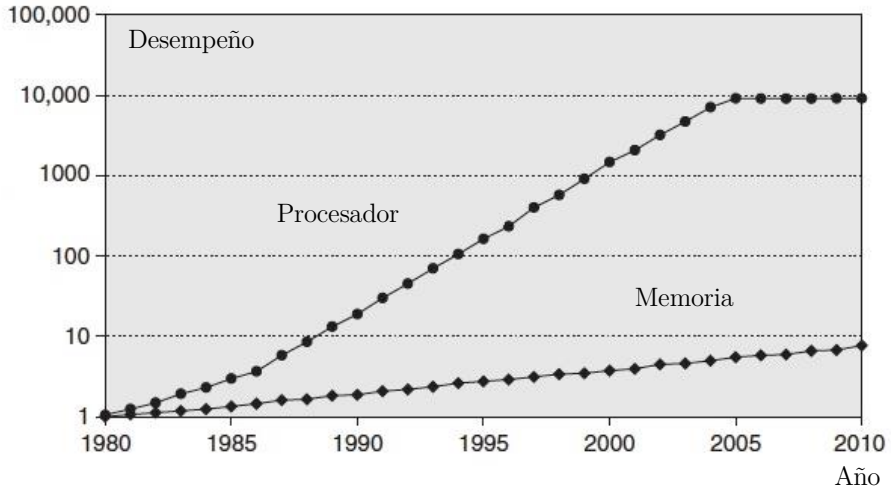


Figura A-1 Comparación del crecimiento (en escala logarítmica) de la capacidad de procesamiento del procesador central contra la reducción de latencia de la memoria RAM. En los últimos años la velocidad de procesamiento por *core* se estancó; sin embargo, la brecha entre ambos todavía es significativamente alta (Hennessy & Patterson, 2007).

Por lo tanto es imperativo reducir los accesos a la RAM, concretamente, reducir los *misses* de *caché*. Para lograrlo se debe intentar tener en *caché* solo los datos que se necesiten y buscarlos en la RAM la menor cantidad de veces posible. En la práctica, esto se traduce en procesar la mayor parte de la información en bloques de datos homogéneos y contiguos en memoria. De esta forma no solo los datos se necesitarán llevar a *caché* una o pocas veces por ciclo de procesamiento (actualización o renderizado de un cuadro), sino que también el código que los procesa se cargará en memoria pocas veces por ciclo.

A continuación se presenta un ejemplo en el cual se intenta optimizar una función; sin embargo, esto resulta en una caída importante del desempeño debido a las características del hardware sobre el que se ejecuta. El código se ejecuta hipotéticamente en una PlayStation 3.

```
float3 BoundingSphereEnEspacioDeMundo(Matrix matrixMundo)
{
   if (actualizacionNecesaria)
      boundingSphereEnEspacioDeMundo = boundingSphereEnEspacioDeObjeto.Transform(matrixMundo);
   return boundingSphereEnEspacioDeMundo;
}
```

Este método o función retorna la *bounding sphere* de la entidad transformada del espacio del objeto al espacio del mundo. Realizar esta transformación es una operación que requiere 12 ciclos de reloj, por lo que se recurre a una bandera que indica si el objeto sufrió alguna transformación; si no es así, simplemente se devuelve la matriz calculada en una llamada anterior. Debido a las características del procesador Cell de la PlayStation 3, un *miss prediction* en esta plataforma equivale a 23 ciclos perdidos. Eso quiere decir, que en una escena mayoritariamente dinámica esta optimización conlleva a una degradación del desempeño de la aplicación, particularmente porque es una operación que suele llamarse al menos una vez por objeto de la escena por cuadro de renderizado. Pero aún más importante, la variable *actualizacionNecesaria* podría no encontrase en el mismo bloque de memoria que el resto de la información procesada, por lo que se podría originar un *caché miss* extra para traerla de memoria, que en este hardware equivale a 400 ciclos de procesador (Albrecht, 2009).

En este ejemplo se realizó una evaluación de desempeño local a la función, pero también es posible que el desempeño de la función esté afectado por factores externos conjuntamente con factores internos todavía no discutidos. A continuación se comparan dos procedimientos equivalentes en funcionalidad que permiten analizar esta situación:

```
RenderizarObjetos(List<> objetos)
{
   foreach objeto en objetos
      if (DentroFrustumCamara(objeto.BoundingSphereEnEspacioDeMundo(objeto.MatrixMundo))
         Objeto.Renderizar();
}
```

```
RenderizarObjetos(List<> objetos)
{  bool[] renderizar;
   for (i = 0; i < objetos.Longitud; i++)
      renderizar[i] = DentroFrustumCamara(objeto.BoundingSphereEnEspacioDeMundo(objeto.MatrixMundo);
   for (i = 0; i < objetos.Longitud; i++)
```

```
      if (renderizar[i])
         Objeto[i].Renderizar();
}
```

La operación Renderizar es típicamente compleja y probablemente reemplace los datos de la *caché* de programa con su propio código, por lo que en cada iteración del primer procedimiento expuesto, el código de las operaciones *DentroFrustumCamara* y *BoundingSphereEnEspacioDeMundo* será buscado en memoria e insertado nuevamente en caché. Esto puede evitarse si se procesa la información en lote, como se muestra en el segundo procedimiento. Además, esto da cabida a otra posible optimización, organizar los propios datos en lote. Si las matrices de mundo y *bounding spheres* de todos los objetos se encuentran almacenadas contiguas en memoria se evitarán varios *caché misses* debido a que solo se traerá de memoria la información útil en la etapa actual, lo que permitirá incrementar significativamente la velocidad de ejecución.

Ignorar cómo funciona el hardware puede significar la diferencia entre una aplicación de tiempo real y una que no puede cumplir con este requerimiento. En particular, las aplicaciones gráficas brindan una gran oportunidad de optimización debido a la gran cantidad de información homogénea que procesan en lote, similar al comportamiento de una GPU. Por ejemplo, el motor Frostbite de la empresa DICE fue desarrollado utilizando la POO e incorporaba un complejo, pero optimizado, algoritmo de *frustum culling*. En cambio, el motor Frostbite 2 fue desarrollado teniendo en cuenta los datos procesados y su flujo; para éste se incorporó un algoritmo simple de *frustum culling* de fuerza bruta que se ejecuta en lote y que transfiere a caché solo la información sensible para dicha operación. De esta manera se pudo triplicar la velocidad de la ejecución de esta operación en escenas complejas de 15.000 objetos, utilizando un código 5 veces más pequeño que a su vez permite mayores oportunidades de optimización debido a su tamaño y sencillez (Collin, 2011).

A.2 Programación Orientada a Datos y Desarrollo Basado en Componentes

Las limitaciones del hardware contemporáneo dan lugar a la adopción de la Programación Orientada a Datos (POD) que se centra en identificar y aislar las tareas (o procesamientos de alto nivel) y los datos que se acceden y manipulan en estas tareas. De esta manera, la organización del código no está guiada por las entidades como ocurre en la POO (Frykholm, 2010; Joshi, 2007). Una de las características más visibles de este paradigma es que se procesa la mayor parte de la información en bloques de datos homogéneos y contiguos en memoria.

Desafortunadamente, con la POD la visualización de las entidades podría resultar dificultosa. Sin embargo existe una solución para este problema, y es la de utilizar un modelo de agregación en el que cada entidad posea un conjunto de componentes (aislados unos de otros) que coinciden con los datos identificados. Los componentes del mismo tipo son agrupados en la RAM sin importar a qué objeto pertenezcan, y de esta manera se mejora la localidad de los datos. Además, son los datos de entrada de

unidades funcionales los que producen los resultados de los distintos sistemas internos de la aplicación, es decir, el renderizado, la física, el audio, etc. Este modelo se conoce como Desarrollo Basado en Componentes (DBC).

Las ganancias que se obtienen a través de la POD y el DBC no solo quedan confinadas en disminuir los accesos innecesarios a memoria. Muchas aplicaciones se modelan naturalmente con este paradigma. En general, el código resultante es simple (aunque se introducen ciertas complejidades), mejor modularizado y más paralelizado. Además, trabajar directamente con los datos facilita crear unidades de testeo (Albrecht, 2009; Llopis, 2011).

La POD y la POO pueden ser vistas como enfoques complementarios. Mientras las entidades de la aplicación son visualizadas con la POD y el DBC, ciertos subsistemas podrían modelarse utilizando la POO. Además, la POD, como la POO, puede implementarse en lenguajes enfocados en otros paradigmas. Por ejemplo, la POD puede implementarse en lenguajes orientados a objetos como C++ o C# (Gregory, 2009).

A.3 Antecedentes

Kuznetsov definió los principios en lo que se basa la POD (Kuznetsov). Estos principios comenzaron a adoptarse en la industria del videojuego a principio de la década pasada. Uno de los primeros en utilizar la POD exitosamente fue *Ensemble Studio* con su juego *Ages of Mithology* (Fermier, 2002). Años más tarde, y con mayor desarrollo e investigación, varias compañías comenzaron a desarrollar sus productos bajo la POD, y se comenzó a utilizar el concepto de DBC. Empresas como *Naughty Dog* y *Radical Entertainment* desarrollaron exitosamente sus videojuegos bajo este modelo (Gregory, 2009).

Actualmente, no todas las empresas desarrolladoras de videojuegos utilizan la POD; además, muchos programadores que han sido formados en la POO tienen cierto recelo de utilizarla. Los líderes de proyecto de *Radical Entertainment* con su juego *Prototype* concluyeron que fue un gran acierto seguir este modelo, y sus programadores también, a pesar de que en un principio manifestaron cierta resistencia en su adopción (Chady, 2009).

Desafortunadamente no es un tópico de relevancia en investigaciones científicas y por lo tanto la información relacionada proviene principalmente de presentaciones y cursos realizados por empresas de videojuegos, que por necesidad buscaron alternativas que les permitieran producir código de mayor calidad, aumentando al mismo tiempo el desempeño de la aplicación.

A.4 Creación de una Aplicación Gráfica bajo estos Principios

En esta sección se describirán las principales tareas de alto nivel que permiten la creación de una aplicación gráfica siguiendo los principios antes descriptos.

La primera tarea realizada en el proceso de diseño consiste en identificar las tareas y los datos, organizar estos datos en entidades o componentes y planificar de antemano su flujo y comunicación. En el diseño basado en la POO los tipos de entidades forman una jerarquía normalmente profunda en donde la funcionalidad se va especializando a medida que se desciende en la jerarquía de herencia. Sin embargo, y para evitar la herencia múltiple, gran parte de la definición de la funcionalidad se encuentra cerca de la raíz, y es probable que esta funcionalidad no sea utilizada por todos sus hijos. Es posible implementar la funcionalidad cerca de las hojas de la jerarquía, pero hacerlo de esta manera supondría la replicación de código, afectando severamente la mantenibilidad. En cambio, con el DBC las entidades se organizan de manera diferente; se pasa de un modelo de especialización a un modelo de agregación. De esta forma, toda entidad es de un mismo tipo común, normalmente denominado objeto de juego (o *game object*) debido a que la mayor parte de la bibliografía fue desarrollada por empresas de desarrollo de videojuegos. Un objeto de juego tendrá un conjunto de componentes que determinarán el comportamiento del mismo. Los componentes pueden ser de renderizado (de mallas, de partículas, etc.), de física, de audio, etc.

Para favorecer un buen diseño, los componentes idealmente no deben conocer la interfaz ni el funcionamiento del resto de los componentes; sin embargo, en ciertas circunstancias, la comunicación entre ellos es necesaria. En estos casos se recomienda realizar la comunicación por medio de mensajes. Es posible utilizar el sistema de eventos como medio de comunicación, dado que suelen estar implementados en el lenguaje de programación utilizado y proveen un mecanismo eficiente para tal objetivo. Normalmente la comunicación tiene como destino u origen al componente de transformación; dicho componente es el responsable de ubicar al objeto en el mundo 2D o 3D de la escena. Por razones de practicidad este componente siempre está asociado con un objeto de juego y se permite al resto de los componentes asumir su existencia. Pero esto no significa que los componentes puedan comunicarse directamente con él; siempre es recomendable usar el sistema de mensajes.

Una vez definidos los datos se definen las unidades funcionales (o *managers*). Estas unidades normalmente operan en lote sobre uno o pocos tipos de componentes a la vez y producen un resultado global (renderizado, sonido, IA, física). De esta manera no es necesario tener en *caché* todos los datos del objeto de juego al mismo tiempo.

La información traída a *caché* por estas unidades funcionales puede reducirse aún más. Por ejemplo, al realizar el renderizado de mallas se necesita, entre otras cosas, la información del componente de renderizado de mallas y la posición del objeto de juego en el mundo, esto es, es necesario conocer la matriz del mundo del componente de transformación. La matriz del mundo no cambia para todos los objetos de juego ni en todos los cuadros de renderizado, por lo que el componente de renderizado de mallas podría almacenar una copia de este valor y por medio de mensajes actualizarla solo cuando éste cambia. De esta forma se evita acceder al resto de los datos del componente de transformación en el renderizado de mallas, reduciendo los *caché misses* y mejorando el desempeño.

Las unidades funcionales típicamente son modeladas utilizando complementariamente los principios de la POO y de la POD. Esto no significa, de ninguna manera, una reducción de la calidad o performance del código resultante.

Referencias

Abrash, M. (1996). Quake's Lighting Model: Surface Caching. *Dr. Dobb's Sourcebook #260*.

Albrecht, T. (2009). Pitfalls of Object Oriented Programming. *Game Connect Asia Pacific '09.*

AMD. (2012). *Radeon™ HD 7900 Series Graphics Real-Time Demos.* Obtenido de developer.amd.com: http://developer.amd.com/resources/documentation-articles/samples-demos/gpu-demos/amd-radeon-hd-7900-series-graphics-real-time-demos/

Andersson, J. (2009). Parallel Graphics in Frostbite - Current & Future. *SIGGRAPH '09 Course: Beyond Programmable Shading.*

Andrews, L. C., & Phillips, R. L. (2005). *Laser beam propagation through random media, Second Edition.* Washington: SPIE Publications.

Annen, T., Kautz, J., Durand, F., & Seidel, H.-P. (2004). Spherical Harmonic Gradients for Mid-Range Illumination. *EGSR'04 Proceedings of the Fifteenth Eurographics Conference on Rendering Techniques* (págs. 331-336). Aire-la-Ville: Eurographics Association.

Balestra, C., & Engstad, P.-K. (2008). The Technology of Uncharted: Drake's Fortune. *Game Developer's Conference '08.*

Barret, S. (2008). *Sparse Virtual Textures*. Obtenido de silverspaceship.com/src/svt.

Bartleson, C. J., & Breneman, E. J. (1967). Brightness Reproduction in the Photographic Process. *Photographic Science and Engineering, 11*(4), 254–262.

Bavoil, L., & Andersson, J. (2012). Stable SSAO in Battlefield 3 with Selective Temporal Filtering. *Game Developer's Conference '12.*

Bavoil, L., & Myers, K. (2008). Deferred Rendering using a Stencil Routed K-Buffer. En W. Engel, *ShaderX6* (págs. 189-198). Boston: Charles River Media.

Bavoil, L., & Sainz, M. (2009). Image-Space Horizon-Based Ambient Occlusion. En W. Engel, *ShaderX7: Advanced Rendering Techniques* (págs. 425-443). Boston: Charles River Media.

Beers, A., Agrawala, M., & Chaddha, N. (1996). Rendering from Compressed Textures. *SIGGRAPH '96 Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (págs. 373-378). New York: ACM.

Bittner, J., Mattausch, O., & Wimmer, M. (2009). Game-Engine-Friendly Occlusion Culling. En W. Engel, *ShaderX7* (págs. 637-653). Boston: Charles River Media.

Bjorke, K. (2005). *Let's Get Small Understanding MIP Mapping.* Obtenido de developer.nvidia.com.

Blinn, J. F. (1977). Models of Light Reflection for Computer Synthesized Pictures. *SIGGRAPH '77 Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques* (págs. 192–198). New York: ACM.

Borb. (2008). Obtenido de wikimedia.org.

Bunnell, M. (2005). Dynamic Ambient Occlusion and Indirect Lighting. En M. Pharr, & F. Randima, *GPU Gems 2: Programming Techniques For High-Performance Graphics And General-Purpose Computation.* Boston: Addison-Wesley Professional.

Campbell, G., DeFanti, T. A., Frederiksen, J., Joyce, S. A., & Leske, L. A. (1986). Two Bit/Pixel Full Color Encoding. *SIGGRAPH '86 Proceedings of the 13th annual conference on Computer graphics and interactive techniques* (págs. 215-223). New York: ACM.

Chady, M. (2009). Theory and Practice of Game Object Component Architecture. *Game Developer's Conference '09.*

Chajdas, M. G., Eisenacher, C., Stamminger, M., & Lefebvre, S. (2010). Virtual Texture Mapping. En E. Wofgang, *GPU Pro 1* (págs. 185-195). Natick: A.K. Peters.

Chapman , N. (2007). *indigorenderer.com.*

Chen, H., & Liu, X. (2008). Lighting and material of Halo 3. *SIGGRAPH '08 ACM SIGGRAPH 2008 Games* (págs. 1-22). New York: ACM.

Chen, H., & Tatarchuk, N. (2009). Lighting Research at Bungie. *SIGGRAPH '09 Advances in Real-Time Rendering in 3D Graphics and Games Course.*

Chiu, K., Herf, M., Shirley, P., Swamy, S., Wang, C., & Zimmerman, K. (1993). Spatially Nonuniform Scaling Functions for High Contrast Images. *Proceedings of Graphics Interface '93*, (págs. 245-244).

Christensen, P. H. (2003). Global Illumination and All That. *SIGGRAPH '03 Course 9.*

Collin, D. (2011). Culling the Battlefield: Data Oriented Design in Practice. *Game Developer's Conference.*

Cook, R. L., & Torrance, K. E. (1982). A Reflectance Model for Computer Graphics. *ACM Transactions on Graphics, 1*(1), 7-24.

Cook, R., Porter, T., & Carpenter, L. (1984). Distributed Ray Tracing. *SIGGRAPH '84 Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques* (págs. 137-145). New York: ACM.

Dachsbacher, C., & Stamminger, M. (2005). Reflective Shadow Maps. *I3D '05 Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games* (págs. 203-231). New York: ACM.

Dachsbacher, C., Stamminger, M., Drettakis, G., & Durand, F. (2007). Implicit Visibility and Antiradiance for Interactive Global Illumination. *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2007, 26*(3), Articulo No. 61.

Deering, M., Winner, S., Schediwy, B., Duffy, C., & Hunt, N. (1988). The Triangle Processor and Normal Vector Shader: a VLSI System for High Performance Graphics. *SIGGRAPH '88 Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques* (págs. 21-30). New York: ACM.

Delp, E. J., & Mitchell, O. R. (1979). Image Compression Using Block Truncation Coding. *IEEE Transactions on Communications, COM-27*(9), 1335-1342.

Dimitrov, R., Bavoil, L., & Sainz, M. (2008). Horizon-Split Ambient Occlusion. *I3D '08 Proceedings of the 2008 symposium on Interactive 3D graphics and games.* New York: ACM.

Donzallaz, P.-Y., & Sousa, T. (2011). Lighting in Crysis 2. *Game Developer's Conference '11.*

Drago, F., Myszkowski, K., Annen, T., & Chiba, N. (2003). Adaptive Logarithmic Mapping for Displaying High Contrast Scenes. *Computer Graphics Forum, 22*, 419-426.

Drone, S. (2007). Under the Hood: Revving Up Shader Performance. *Gamefest '07.*

Duiker, H.-P. (2003). Lighting reconstruction for "The Matrix Reloaded". *SIGGRAPH '03 Sketches & Applications* (págs. 1-1). New York: ACM.

Durand, F., & Dorsey, J. (2000). Interactive Tone Mapping. *Proceedings of the Eurographics Workshop on Rendering Techniques 2000* (págs. 219-230). London: Springer-Verlag.

Dutré, P., Bala, K., & Behaert, P. (2006). *Advanced Global Illumination.* Wellesley: A. K. Peters.

Eisemann, E., & Durand, F. (2004). Flash Photography Enhancement via Intrinsic Relighting. *SIGGRAPH '04 ACM SIGGRAPH 2004 Papers* (págs. 673-678). New York: ACM.

EIZO Library. (2008). *The Ability to Display Color Correctly Is Vital: Understanding the Color Gamut of an LCD Monitor.* Obtenido de eizo.com.

Engel, W. (2006). Cascaded Shadow Maps. En W. Engel, *ShaderX5* (págs. 197-206). Boston: Charles River Media.

Engel, W. (2009). Designing a Renderer for Multiple Lights: The Light Pre-Pass Renderer. En W. Engel, *ShaderX7* (págs. 655-666). Boston: Charles River Media.

Evans, A. (2006). Fast Approximations for Global Illumination on Dynamic Scenes. *SIGGRAPH '06 ACM SIGGRAPH 2006 Courses* (págs. 153-171). New York: ACM.

Evans, K. F. (1998). The Spherical Harmonics Discrete Ordinate Method for Three-Dimensional Atmospheric Radiative Transfer. *Journal of Atmospheric Sciences*, 429-446.

Fairchild, M. D., & Reniff, L. (1995). Time Course of Chromatic Adaptation for Color-Appearance Judgments. *Journal of the Optical Society of America A, 12*(5), 824-833.

Fairman, H. S., Brill, M. H., & Hemmendinger, H. (1997). How the CIE 1931 Color-Matching Functions were Derived from Wright-Guild Data. *Color Research & Application, 22*(1), 11-23.

Farrar, T. (2007). *Output Data Packing.* Obtenido de opengl.org: http://www.opengl.org/discussion_boards/showthread.php/163470-Output-data-packing

Fermier, R. (2002). Creating a Data Driven Engine: Case Study: The Age Of Mythology. *Game Developer's Conference '02.*

Ferrer Baquero, J. J. (2007). Obtenido de wikipedia.org.

Ferwerda, J. A., Pattanaik, S. N., Shirley, P., & Greenberg, D. P. (1996). A Model of Visual Adaptation for Realistic Image Synthesis. *SIGGRAPH '96 Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques* (págs. 249-258). New York: ACM.

Feynman, R. P. (1998). *Six Easy Pieces: Fundamentals of Physics Explained.* London: Penguin Books.

Flagstaffotos. (2005). Obtenido de flagstaffotos.com.au.

Flannery, R. (s.f.). Obtenido de rebeccaflannery.com.

Frey, I. Z., & Herzeg, I. (2011). Spherical Skinning with Dual Quaternions and QTangents. *SIGGRAPH '11 ACM SIGGRAPH 2011 Talks* (pág. Articulo No. 11). New York: ACM.

Frykholm, N. (2010). Practical Examples in Data-Oriented Design. *Sthlm Game Developer Forum.*

García, A., Ávila, F., Murguía, S., & Reyes, L. (2012). Interactive Ray Tracing Using The Compute Shader in DirectX 11. En W. Engel, *GPU Pro 3* (págs. 353-376). Natick: A K Peters.

Glassner, A. S. (1995). *Principles of Digital Image Synthesis.* Burlington: Morgan Kaufmann.

Goldstone, W. (2011). *Unity 3.x Game Development Essentials.* Birmingham: Packt Publishing.

Gómez-Esteban, P. (2007). *Cuántica sin formulas.* Madrid: El Tamiz.

Goodnight, N., Wang, R., Woolley, C., & Humphreys, G. (2003). Interactive Time-Dependent Tone Mapping using Programmable Graphics Hardware. *EGRW '03 Proceedings of the 14th Eurographics Workshop on Rendering* (págs. 26 - 37). Switzerland: The Eurographics Association.

Goral, C. M., Torrance, K. E., Greenberg, D. P., & Battaile, B. (1984). Modeling the Interaction of Light Between Diffuse Surfaces. *SIGGRAPH '84 Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques* (págs. 213-222). New York: ACM.

Green, C. (2007). Efficient Self-Shadowed Radiosity Normal Mapping. *SIGGRAPH '07 Courses* (págs. 1-8). New York: ACM.

Green, C., & McTaggart, G. (2006). High Performance HDR Rendering on DX9-Class Hardware. *ACM Symposium on Interactive 3D Graphics and Games.*

Green, R. (2003). *Spherical Harmonic Lighting: The Gritty Details.* Sony Computer Entertainment America.

Greger, G., Shirley, P., Hubbard, P. M., & Greenberg, D. P. (1998). The Irradiance Volume. *Journal IEEE Computer Graphics and Applications, 18*(2), 32-43.

Gregory, J. (2009). *Game Engine Architecture.* Natick: A K Peters.

Gritz, L., & d'Eon, E. (2007). The Importance of Being Linear. En H. Nguyen, *GPU Gems 3* (págs. 529–542). Boston: Addison-Wesley Professional.

Grosch, T., & Ritschel, T. (2010). Screen-Space Directional Occlusion. En E. Wofgang, *GPU Pro 1* (págs. 215-230). Natick: A.K. Peters.

Guerrero, P., Jeschke, S., & Wimmer, M. (2008). Real-Time Indirect Illumination and Soft Shadows in Dynamic Scenes Using Spherical Lights. *Computer Graphics Forum, 27*(8), 2154-2168.

Guyton, A. C., & Hall, J. E. (2001). *Tratado de Fisiología Médica.* Madrid: McGraw-Hill Interamericana.

Hable, J. (2010). *Gamma: 360 vs. PS3.* Obtenido de filmicgames.com.

Hargreaves, S. (2004). Deferred Shading. *Game Developer's Conference '04.*

Hargreaves, S. (2009). *Texture Filtering: Mipmaps.* Obtenido de blogs.msdn.com/b/shawnhar/.

Hargreaves, S. (s.f.). *Shawn Hargreaves Blog.* Obtenido de http://www.shawnhargreaves.com/blogindex.html

Hašan, M., Křivánek, J., Walter, B., & Bala, K. (2009). Virtual Spherical Lights for Many-Light Rendering of Glossy Scenes. *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH Asia 2009, 28*(5), Articulo No. 143.

He, X. D., Torrance, K. E., Sillion, F. X., & Greenberg, D. P. (1991). A Comprehensive Physical Model for Light Reflection. *SIGGRAPH '91 In Proceedings of the 18th Annual Cconference on Computer Graphics and Interactive Techniques* (págs. 175–186). New York: ACM Press.

Heidrich, W., & Seidel, H.-P. (1998). View-Independent Environment Maps. *HWWS '98 Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware* (págs. 39-49). New York: ACM.

Heidrich, W., & Seidel, H.-P. (1999). Realistic, Hardware-accelerated Shading and Lighting. *SIGGRAPH '99 Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques* (págs. 171-178). New York: ACM.

Hennessy, J. L., & Patterson, D. A. (2007). *Computer Architecture - A Quantitative Approach.* San Francisco: Morgan Kaufmann.

Holzer, B. (2007). *High Dynamic Range Image Formats.* Vienna: Institute of Computer Graphics & Algorithms TU Vienna.

Hood, D. C., & Finkelstein, M. A. (1986). Sensitivity to Light. En K. R. Boff, J. P. Thomas, & L. R. Kaufman, *Handbook of Perception & Human Performance* (págs. 5-1-5-66). New York: Wiley-Interscience.

Horst, F. (2007). Obtenido de zeitmaschinisten.com: http://www.zeitmaschinisten.com/svg/Spektrum.svg

Houston, M., & Lefohn, A. (2011). Graphics with GPU Compute APIs. *SIGGRAPH '11 Courses.* New York: ACM.

Hunt, R. W. (2004). *The Reproduction of Colour.* New York: Fountain Press.

Immel, D. S., Cohen, M. F., & Greenberg, D. P. (1986). A Radiosity Method for Non-Diffuse Environments. *SIGGRAPH '86 Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques* (págs. 133-142). New York: ACM.

Jarosz, W. (2008). *Efficient Monte Carlo Methods for Light Transport in Scattering Media.* San Diego: Ph.D. dissertation.

Jensen, H. W. (1996). Global Illumination using Photon Maps. *Rendering Techniques '96 Proceedings of the Seventh Eurographics Workshop on Rendering* (págs. 21-30). New York: Springer-Verlag.

Jiménez, J., Echevarría, J. I., Sousa, T., & Gutiérrez, D. (2012). SMAA: Enhanced Subpixel Morphological Antialiasing. *Computer Graphics Forum, 31*(2), 355-364.

Jiménez, J., Masia, B., Echevarría, J. I., Navarro, F., & Gutiérrez, D. (2010). Practical Morphological Antialiasing. En W. Engel, *GPU Pro 2* (págs. 95-113). Natick: A K Peters.

Jones, L. A., & Condit, H. R. (1941). The Brightness Scale of Exterior Scenes and the Computation of Correct Photographic Exposure. *Journal of the Optical Society of America, 31*(11), 651-678.

Joshi, R. (2007). *Data-Oriented Architecture: A Loosely-Coupled Real-Time SOA.* Real-Time Innovations, Inc.

Kajalin, V. (2009). Screen-Space Ambient Occlusion. En E. Wolfgang, *ShaderX7: Advanced Rendering Techniques* (págs. 413-424). Boston: Charles River Media.

Kajiya, J. T. (1985). Anisotropic Reflection Models. *SIGGRAPH '85 Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques* (págs. 15–21). New York: ACM Press.

Kajiya, J. T. (1986). The Rendering Equation. *SIGGRAPH '86 Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques* (págs. 143-150). New York: ACM.

Kaplanyan, A. (2009). Light Propagation Volumes in CryEngine 3. *SIGGRAPH '09 Advances in Real-Time Rendering in 3D Graphics and Games Course.*

Kaplanyan, A. (2010). CryENGINE 3: Reaching the Speed of Light. *Advances In Real-Time Rendering, SIGGRAPH '10 Course*.

Kaplanyan, A. S., Engel, W., & Dachsbacher, C. (2010). Diffuse Global Illumination with Temporally Coherent Light Propagation Volumes. En W. Engel, *GPU Pro 2* (págs. 185-203). Natick: A K Peters.

Kaplanyan, A., & Dachsbacher, C. (2010). Cascaded Light Propagation Volumes for Real-Time Indirect Illumination. *I3D '10 Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (págs. 99-107). New York: ACM.

Keller, A. (1997). Instant Radiosity. *SIGGRAPH '97 Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques* (págs. 49-56). New York: ACM.

Kircher, S., & Lawrance, A. (2009). Inferred Lighting: Fast Dynamic Lighting and Shadows for Opaque and Translucent Objects. *Sandbox '09 Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games* (págs. 39-45). New York: ACM.

Klehm, O., Ritschel, T., Eisemann, E., & Seidel, H.-P. (2012). Screen-Space Bent Cones: A Practical Approach. En E. Wolfgang, *GPU Pro 3* (págs. 191-207). Natick: A.K. Peters.

Ko, J., Ko, M., & Zwicker, M. (2009). Practical Methods for a PRT-Based Shader Using Spherical Harmonics. En W. Engel, *ShaderX7* (págs. 355-379). Boston: Charles River Media.

Kodak. (1998). *KODAK VISION Premier Color Print Film 2393.* Obtenido de motion.kodak.com.

Koonce, R. (2007). Deferred Shading in Tabula Rasa. En H. Nguyen, *GPU Gems 3.* Boston: Addison-Wesley Professional.

Krawczyk, G., Myszkowski, K., & Seidel, H.-P. (2005). Perceptual Effects in Real Time Tone Mapping. *SCCG '05 Proceedings of the 21st Spring Conference on Computer Graphics* (págs. 195-202). New York: ACM.

Kuznetsov, E. (s.f.). *Data Oriented Programming.* DataPower.

Lafortune, E. P., Foo, S.-C., Torrance, K. E., & Greenberg, D. P. (1997). Non-linear Approximation of Reflectance Functions. *SIGGRAPH '97 Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques* (págs. 117-126). New York: ACM.

Lafortune, L., & Willems, Y. (1993). Bi-Directional Path Tracing. *Compugraphics '93 Proceedings of the Third International Conference on Computational Graphics and Visualization Techniques*, (págs. 145-153). Alvor.

Laine, S., Saransaari, H., Kontkanen, J., Lehtinen, J., & Aila, T. (2007). Incremental Instant Radiosity for Real-Time Indirect Illumination. *EGSR'07 Proceedings of the 18th Eurographics conference on Rendering Techniques* (págs. 277-286). Aire-la-Ville: Eurographics Association.

Landis, H. (2002). Production-Ready Global Illumination. *SIGGRAPH '02 Course 16* (págs. 87–102). New York: ACM.

Ledda, P., Chalmers, A., Troscianko, T., & Seetzen, H. (2005). Evaluation of Tone Mapping Operators Using a High Dynamic Range Display. *ACM Transactions on Graphics (TOG), 24*(3), 640-648.

Liao, H.-C. (2010). Shadow Mapping for Omnidirectional Light Using Tetrahedron Mapping. En W. Engel, *GPU Pro 1* (págs. 185-195). Natick: A.K. Peters.

Linde, R. (2005). *Making Quality Game Textures.* Obtenido de gamasutra.com.

Llopis, N. (2011). High-Performance Programming with Data-Oriented Design. En E. Lengyel, *Game Engine Gems 2* (págs. 251-261). Natick: A K Peters.

Lottes, T. (2009). *FXAA.* NVIDIA: Santa Clara.

Lyon, D. (2009). Obtenido de dicklyon.com.

Magnusson, K. (2011). Lighting you up in Battlefield 3. *GDC '11 Game Developer's Conference.*

Mardia, K. V., & Jupp, P. E. (2009). *Directional Statistics.* New Jersey: John Wiley and Sons.

Marshall, T. W. (2006). *Wave-particle duality in the 17th century.* Obtenido de physic.philica.com.

Mattausch, O., Scherzer, D., & Wimmer, M. (2010). Temporal Screen-Space Ambient Occlusion. En E. Wolfgang, *GPU Pro 2* (págs. 123-141). Natick: A K Peters.

Max, N. L. (1986). Horizon Mapping: Shadows for Bump-Mapped Surfaces. *Advanced Computer Graphics Proceedings of Computer Graphics Tokyo '86* (págs. 145-156). Tokyo: Springer Verlag.

McGuire, M., & Luebke, D. (2009). Hardware-Accelerated Global Illumination by Image Space Photon Mapping. *HPG '09 Proceedings of the Conference on High Performance Graphics 2009* (págs. 77-89). New York: ACM.

McTaggart, G. (2004). Half-Life 2 / Valve Source Shading. *GDC '04 Game Developer's Conference - Direct3D Tutorial Day.* San Diego.

Mead, C. (2002). *Collective Electrodynamics: Quantum Foundations of Electromagnetism.* Cambridge: The MIT Press.

Microsoft. (2012). *Predicated Tiling.* Obtenido de msdn.microsoft.com: http://msdn.microsoft.com/en-us/library/bb464139(v=xnagamestudio.40).aspx

Microsoft. (2012). *Tessellation Overview.* Obtenido de msdn.microsoft.com: http://msdn.microsoft.com/en-us/library/windows/desktop/ff476340(v=vs.85).aspx

Miller, G. S., & Hoffman, R. (1984). Illumination and Reflection Maps: Simulated Objects in Simulated and Real Environments. *SIGGRAPH '84 Course Notes for Advanced Computer Graphics Animation.* New York: ACM.

Miller, P. (2012). GPU Ray Tracing. *GPU Technology Conference 2012.*

Mitchell, J., McTaggart, G., & Green, C. (2006). Shading in Valve's Source Engine. *SIGGRAPH '06 Course on Advanced Real-Time Rendering in 3D Graphics and Games* (págs. 129-142). New York: ACM.

Mittring, M. (2007). Finding Next Gen - CryEngine 2. *SIGGRAPH '07 Courses* (págs. 97-121). New York: ACM.

Mittring, M. (2009). A Bit More Deferred - CryEngine 3. *Triangle Game Conference '09.*

Mittring, M., & Dudash, B. (2011). The Technology Behind the DirectX 11 Unreal Engine "Samaritan" Demo. *Game Developer's Conference '11.*

Moore, J., & Jefferies, D. (2009). Rendering Techniques in Split/Second. *SIGGRAPH '09 Advanced Real-Time Rendering in 3D Graphics and Games Course.*

Moorthy, S. (2006). Obtenido de wikimedia.org.

Morin, R. C. (2002). *Managed C# versus Unmanaged C++.* Obtenido de kbcafe.com: http://www.kbcafe.com/articles/CSharp.Performance.pdf

Naka, K. I., & Rushton, W. A. (1966). S-potentials from Luminosity Units in the Retina of Fish (Cyprinidae). *Journal of Physiology, 192*(2), 437–461.

Namek, P. (2006). Obtenido de en.wikipedia.

Nehab, D., Sander, P. V., Lawrence, J., Tatarchuk, N., & Isidoro, J. R. (2007). Accelerating Real-Time Shading with Reverse Reprojection Caching. *Proceedings of the Eurographics Symposium on Graphics Hardware 2007* (págs. 23-35). Aire-la-Ville: Eurographics Association.

Ng, R., Ramamoorthi, R., & Hanrahan, P. (2003). All-Frequency Shadows Using Non-Linear Wavelet Lighting Approximation. *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2003, 22*(3), 376-381.

Ngan, A., Durand, F., & Matusik, W. (2004). Experimental Validation of Analytical BRDF Models. *SIGGRAPH '04 Sketches and Applications* (pág. 90). New York: ACM.

Nicodemus, F. E., Richmond, J. C., Hsia, J. J., Ginsberg, W., & Limperis., T. (1977). *Geometric Considerations and Nomenclature for Reflectance.* Washington: National Bureau of Standards.

Obert, J., van Waveren, J., & Sellers, G. (2012). Virtual Texturing in Software and Hardware. *SIGGRAPH '12 Courses* (pág. Articulo 5). New York: ACM.

Olsson, O., & Assarsson, U. (2011). Tiled Shading. *Journal of Graphics, GPU and Game Tools, 15*(4), 235-251.

Ownby, J.-P., Hall, R., & Hall, C. (2010). Rendering techniques in Toy Story 3. *SIGGRAPH '10 Advances in Real Time Rendering in 3D Graphics and Games Course.*

Oyster, C. W. (1999). *The Human Eye: Structure and Function.* Sunderland: Sinauer Associates Incorporated.

Pattanaik, S. N., Ferwerda, J. A., Fairchild, M. D., & Greenberg, D. P. (1998). A Multiscale Model of Adaptation and Spatial Vision for Realistic Image Display. *SIGGRAPH '98 Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques* (págs. 287–298). New York: ACM.

Pattanaik, S. N., Tumblin, J., Yee, H., & Greenberg, D. P. (2000). Time-Dependent Visual Adaptation For Fast Realistic Image Display. *SIGGRAPH '00 Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques* (págs. 47 - 54). New York: ACM Press/Addison-Wesley Publishing Co.

Perlin, K., & Hoffert, E. M. (1989). Hypertexture. *SIGGRAPH '89 Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques* (págs. 253-262). New York: ACM.

Persson, E. (2006). *HDR Rendering.* Obtenido de ATI March 2006 SDK.

Persson, E. (2010). Making it Large, Beatiful, Fast, and Consistent: Lessons Learned Developing Just Cause 2. En E. Wolfgang, *GPU Pro 2* (págs. 571-596). Natick: A K Peters.

Pettineo, M. (2010). *Inferred Rendering.* Obtenido de mynameismjp.wordpress.com: http://mynameismjp.wordpress.com/2010/01/10/inferred-rendering/

Phong, B. T. (1975). Illumination for Computer Generated Pictures. *Communications of the ACM, 18*(6), 311-317.

Pratt, W. K. (2001). *Digital Image Processing* (3th ed.). New York: John Wiley & Sons.

Preiner, R., & Wimmer, M. (2010). Real-Time Global Illumination for Point Cloud Scenes. *Computer Graphics & Geometry, 12*(1), 2--16.

Ramamoorthi, R., & Hanrahan, P. (2001). An Efficient Representation for Irradiance Environment Maps. *SIGGRAPH '01 Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques* (págs. 497-500). New York: ACM.

Reinhard, E. (2003). Parameter Estimation for Photographic Tone Reproduction. *Journal of Graphics Tools, 7*(1), 45-51.

Reinhard, E., & Devlin, K. (2005). Dynamic Range Reduction inspired by Photoreceptor Physiology. *IEEE Transactions on Visualization and Computer Graphics, 11*(1), 13-24.

Reinhard, E., Ward, G., Pattanaik, S., & Debevec, P. (2005). *High Dynamic Range Imaging Acquisition, Display, and Image-Based Lighting*. Burlington: Morgan Kaufmann Publishers.

Ren, Z., Wang, R., Snyder, J., Zhou, K., Liu, X., Sun, B., y otros. (2006). Real-Time Soft Shadows in Dynamic Scenes using Spherical Harmonic Exponentiation. *ACM Transactions on Graphics - Proceedings of ACM SIGGRAPH 2006, 25*(3), 977-986.

Reshetov, A. (2009). Morphological Antialiasing. *HPG '09 Proceedings of the Conference on High Performance Graphics 2009* (págs. 109-116). New York: ACM.

Ritschel, T., Grosch, T., & Seidel, H.-P. (2009). Approximating Dynamic Global Illumination in Image Space. *I3D '09 Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games* (págs. 75-82). New York: ACM.

Ritschel, T., Grosch, T., Kim, M. H., Seidel, H.-P., Dachsbacher, C., & Kautz, J. (2008). Imperfect Shadow Maps for Efficient Computation of Indirect Illumination. *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH Asia 2008, 27*(5), Articulo No. 129.

Rosado, G. (2009). Efficient Soft Particles. En W. Engel, *ShaderX7* (págs. 143-147). Boston: Charles River Media.

Rosen, D. (2010). *Image-Based Ambient Lighting*. gamasutra.com.

Saito, T., & Takahashi, T. (1990). Comprehensible Rendering of 3-D Shapes. *SIGGRAPH '90 Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques* (págs. 197-206). New York: ACM.

Sándor, Z. (2005). Obtenido de fizkapu.hu: http://www.fizkapu.hu/fizfoto/fizfoto6.html

Scherzer, D., Jeschke, S., & Wimmer, M. (2007). Pixel Correct Shadow Maps with Temporal Reprojection and Shadow Test Confidence. *Proceedings of the Eurographics Symposium on Rendering 2007* (págs. 45-50). Aire-la-Ville: Eurographics Association.

Schlick, C. (1994). Quantization Techniques for the Visualization of High Dynamic Range Pictures. *Photorealistic Rendering Techniques* (págs. 7-20). New York: Springer-Verlag.

Schmittler, J., Woop, S., Wagner, D., Paul, W. J., & Slusallek, P. (2004). Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip. *HWWS '04 Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (págs. 95-106). New York: ACM.

Schneider, J. I., & Larrea, M. L. (2011). Programación Orientada a Datos y Desarrollo Basado en Componentes Utilizando el Lenguaje C#. *WAVi 2011* (págs. 109-116). Bahía Blanca: Ediuns.

Schneider, J. I., & Schefer, G. (2010). *XNA Final Engine*. Obtenido de http://xnafinalengine.codeplex.com/

Schneider, J. I., Martig, S., & Castro, S. (2009). Realidad Aumentada en un Entorno Gráfico de Alta Performance. *WICC 2009* (págs. 177-181). Rosario: Editorial de la Universidad Nacional de Rosario.

Schneider, J. I., Martig, S., & Castro, S. (2010). Framework de Realidad Aumentada Integrando Gráficos de Alta Complejidad. *WICC 2010* (págs. 302-306). Calafate: Red de Universidades Nacionales con Carreras en Informática.

Schneider, J. I., Martig, S., & Castro, S. (2010). Framework para la Generación de Aplicaciones Gráficas Integrando Realidad Aumentada y Gráficos de Alta Complejidad. *CACIC 2010* (págs. 455-464). La Plata: Edulp Editorial.

Schneider, J. I., Martig, S., & Castro, S. (2010). Framework XNA Final Engine, Análisis de su Aplicación para el Desarrollo de Juegos y Aplicaciones Gráficas Innovadoras. *WAVi 2010* (págs. 50-57). Bahía Blanca: Ediuns.

Sekuler, R., & Blake, R. (1994). *Perception* (3th ed.). New York: McGraw-Hill.

Shirley, P., & Morley, K. (2008). *Realistic Ray Tracing.* Natick: A K Peters.

Shishkovtsov, O. (2005). Deferred Shading in S.T.A.L.K.E.R. En M. Pharr, & F. Randima, *GPU Gems 2.* Boston: Addison-Wesley Professional.

Skeet, J. (2010). *C# in Depth, Second Edition.* Sebastopol: O'Reilly Media.

Sloan, P.-P., Govindaraju, N. K., Nowrouzezahrai, D., & Snyder, J. (2007). Image-Based Proxy Accumulation for Real-Time Soft Global Illumination. *PG '07 Proceedings of the 15th Pacific Conference on Computer Graphics and Applications* (págs. 97-105). Washington: IEEE Computer Society.

Sloan, P.-P., Kautz, J., & Snyder, J. (2002). Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments. *SIGGRAPH '02 Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques* (págs. 527-536). New York: ACM.

Sloan, P.-P., Luna, B., & Snyder, J. (2005). Local, Deformable Precomputed Radiance Transfer. *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2005, 24*(3), 1216-1224.

Smith, S. (2007). Picture Perfect Gamma Through the Rendering Pipeline. *GameFest.*

Sousa, T. (2007). Vegetation Procedural Animation and Shading in Crysis. En H. Nguyen, *GPU Gems 3* (págs. 373-385). Boston: Addison-Wesley Professional.

Sousa, T. (2011). *Crysis 2 & CryENGINE 3: Key Rendering Features.* Obtenido de crytek.com/cryengine/presentations/.

Sousa, T., Kasyan, N., & Schulz, N. (2012). CryEngine 3: Three Years of Work in Review. En E. Wolfgang, *GPU Pro 3* (págs. 133-168). Natick: A K Peters.

Stamate, V. (2008). Real-Time Photon Mapping Approximation on the GPU. En W. Engel, *ShaderX6* (págs. 393-400). Boston: Charles River Media.

Stokes, M., Anderson, M., Chandrasekar, S., & Motta, R. (1996). *A Standard Default Color Space for the Internet - sRGB - Version 1.10.* Obtenido de w3.org.

Szirmay-Kalos, L. (2000). *Photorealistic Image Synthesis.* Budapest: Technical University of Budapest.

Tatarchuk, N. (2005). Irradiance Volumes for Games. *Game Developers' Conference '05.*

Thibieroz, N. (2004). Deferred Shading with Multiple Render Targets. En W. Engel, *ShaderX2 Shader Programming Tips and Tricks with DirectX 9* (págs. 251-269). Plano: Wordware Publishing.

Thibieroz, N. (2008). Robust Order-Independent Transparency via Reverse Depth Peeling in DirectX 10. En W. Engel, *ShaderX6* (págs. 251-269). Boston: Charles River Media.

Thibieroz, N. (2008). Ultimate Graphics Performance for DirectX 10 Hardware. *Game Developer's Conference '08.*

Thibieroz, N. (2009). Deferred Shading with Multsampling Anti-Aliasing in DirectX 10. En W. Engel, *ShaderX7* (págs. 225-242). Boston: Charles River Media.

Torp, J. (2010). A Step Towards Data Orientation. *DICE Coders Day.*

Trebilco, D. (2009). Light-Indexed Deferred Rendering. En W. Engel, *ShaderX7* (págs. 243-258). Boston: Charles River Media.

Tumblin, J., & Rushmeier, H. (1993). Tone Reproduction for Computer Generated Images. *IEEE Computer Graphics and Applications, 13*(6), 42-48.

Valient, M. (2007). Deferred Rendering in Killzone 2. *Develop Conference.*

Valient, M. (2009). The Rendering Technology of KillZone 2. *Game Developer's Conference 2009.* San Francisco.

Valient, M. (2011). Practical Occlusion Culling in Killzone 3. *SIGGRAPH '11.*

Van den Bergh, S. (2004). Obtenido de planetperplex.com.

van Waveren, J. (2009). id Tech 5 Challenges: From Texture Virtualization to Massive Parallelization. *SIGGRAPH '09: Beyond Programmable Shading.*

Vlachos, A. (2008). Post Processing in The Orange Box. *Game Developer's Conference.*

Wallace, J. R., Cohen, M. F., & Greenberg, D. P. (1987). A Two-Pass Solution to the Rendering Equation: A Synthesis of Ray Tracing and Radiosity Methods. *SIGGRAPH '87 Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques* (págs. 311-320). New York: ACM.

Walter, B., Khungurn, P., & Bala, K. (2012). Bidirectional Lightcuts. *ACM Transactions on Graphics (TOG) - SIGGRAPH 2012 Conference, 31*(4), Articulo No. 59.

Ward Larson, G. (1991). Real Pixels. En J. Arvo, *Graphic Gems II* (págs. 80-83). Salt Lake: Academic Press.

Ward Larson, G. (1998). The LogLuv Encoding for Full Gamut, High Dynamic Range Images. *Silicon Graphics, Inc.*

Ward Larson, G., & Simmons, M. (2004). APGV '04 Proceedings of the 1st Symposium on Applied Perception in Graphics and Visualization. (págs. 83-90). New York: ACM.

Ward, G. (1992). Measuring and Modeling Anisotropic Reflection. *SIGGRAPH '92 Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques* (págs. 265-272). New York: ACM.

Watt, A., & Policarpo, F. (1999). *The Computer Image.* Boston: Addison-Wesley.

Whitted, T. (1979). *An Improved Illumination Model for Shaded Display.* New Jersey: Bell Laboratories.

Yang, L., Nehab, D., Sander, P. V., Sitthi-amorn, P., Lawrence, J., & Hoppe, H. (2009). Amortized Supersampling. *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH Asia 2009, 28*(5).

Zander, J. (2010). Obtenido de grand-illusions.com.