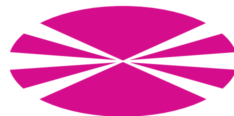
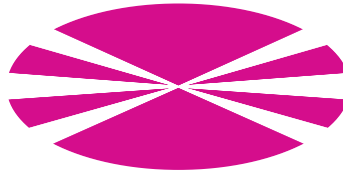
Técnicas de altas prestaciones para métodos de iluminación global

José Rodrigo Sanjurjo Amado



Departamento de Electrónica e Sistemas
Universidade da Coruña

Departamento de Electrónica e Sistemas
Universidade da Coruña



Tesis Doctoral

Técnicas de altas prestaciones para métodos de iluminación global

José Rodrigo Sanjurjo Amado

Diciembre de 2011

Dirigida por:
Margarita Amor López y
Montserrat Bóo Cepeda

Dra. Margarita Amor López, Profesora Titular de Universidad del área de Arquitectura y Tecnología de Computadores de la Universidade da Coruña.

Dra. Montserrat Bóo Cepeda, Profesora Titular de Universidad del área de Arquitectura y Tecnología de Computadores de la Universidade de Santiago de Compostela.

CERTIFICAN:

Que la memoria titulada "Técnicas de altas prestaciones para métodos de iluminación global" ha sido realizada por José Rodrigo Sanjurjo Amado bajo nuestra dirección en el Departamento de Electrónica e Sistemas de la Universidade da Coruña y concluye la Tesis que presenta para optar al grado de Doctor.

A Coruña, 13 de diciembre de 2011.

Fdo.: Dra. Margarita Amor López.
Codirectora de la tesis.

Fdo.: Dra. Montserrat Bóo Cepeda.
Codirectora de la tesis.

Fdo.: Dr. Juan Touriño Domínguez.
Director del Departamento de Electrónica e Sistemas.

Resumen

El gran interés en los métodos de iluminación global se debe a sus múltiples aplicaciones y al realismo de sus imágenes resultantes. La investigación presentada en esta memoria se centra en mejorar computacionalmente el algoritmo de radiosidad, planteando estrategias tanto para métodos determinísticos como estocásticos.

Respecto de los métodos determinísticos, se expondrán nuestras implementaciones en un sistema distribuido del algoritmo de radiosidad progresiva, utilizando el paradigma de paso de mensajes. Estas implementaciones están basadas en la división de la escena de una manera uniforme o no uniforme. Además, se usa la técnica de las máscaras de visibilidad para el cálculo de visibilidad entre elementos de distintos subescenas. También se demuestra que estas metodologías pueden reducir el tiempo de ejecución secuencial.

Relativo a las soluciones estocásticas, presentamos dos implementaciones del método de relajación estocástica de Monte Carlo para radiosidad: en un sistema distribuido y en una *Graphics Processing Unit* (GPU). La primera se basa en tres técnicas: partición de la escena, empaquetamiento de rayos y determinación distribuida del fin de iteración. En la implementación GPU, además de la partición de la escena se empleó la simplificación de la malla de elementos y una organización eficiente de la ejecución de las tareas.

Resumo

O grande interese nos métodos de iluminación global débese ás súas múltiples aplicacións e ao realismo das súas imaxes resultantes. A investigación presentada nesta memoria céntrase en mellorar computacionalmente o algoritmo de radiosidade, formulando estratexias tanto para métodos determinísticos como estocásticos.

Respecto dos métodos determinísticos, exporanse as nosas implementacións nun sistema distribuído do algoritmo de radiosidade progresiva, utilizando o paradigma de paso de mensaxes. Estas implementacións están baseadas na división da escena dunha maneira uniforme ou non uniforme. Ademais, úsase a técnica das máscaras de visibilidade para o cálculo de visibilidade entre elementos de distintas subescenas. Tamén se demostra que estas metodoloxías poden reducir o tempo de execución secuencial.

Relativo as solucións estocásticas, presentamos dúas implementacións do método de relaxación estocástica de Monte Carlo para radiosidade: nun sistema distribuído e nunha *Graphics Processing Unit* (GPU). A primeira baséase en tres técnicas: partición da escena, empaquetamento de raios e determinación distribuída do fin de iteración. Na implementación GPU, ademais da partición da escena empregouse a simplificación da malla de elementos e unha organización eficiente da execución das tarefas.

Summary

The great interest in global illumination methods is due to their multiple applications and the realism of the resulting images. The research presented in the present thesis focuses on computationally improving the radiosity algorithm, proposing strategies for both deterministic and stochastic approaches.

For deterministic approaches, our implementations of the progressive radiosity algorithm will be demonstrated in a distributed system , using the message passing paradigm. These implementations are based on the partitioning of the scene in a uniform or non uniform manner. Furthermore, the technique of visibility masks is employed to calculate the visibility between elements in different subscenes. It is also shown that these methods are capable of reducing the sequential execution time.

With regard to stochastic solutions, we present two implementations of the stochastic relaxation method for Monte Carlo radiosity: in a distributed system and in a Graphics Processing Unit (GPU). The first is based on three techniques: partition of the scene, ray packing strategy and distributed testing of the end of each iteration. In the GPU implementation, as well as the partition of the scene a simplified mesh of the elements was used along with an efficient thread scheduling.

Agradecimientos

Esta Tesis no habría sido posible sin la colaboración de muchas personas y me gustaría aprovechar estas líneas para agradecer su aportación, empezando por mis directoras, Montse y Marga. Gracias a Montse que desde la distancia ha realizado un gran esfuerzo para mejorar las publicaciones y esta memoria. Y muchísimas gracias a Marga por su total implicación, ya que sin su trabajo no habríamos llegado hasta aquí.

Agradecer también al Departamento de Electrónica y Sistemas de la Universidad da Coruña y, fundamentalmente, al Grupo de Arquitectura de Computadores, por acogerme, darme la oportunidad de desarrollar mi carrera investigadora y facilitarme el acceso a todos los recursos que me fueron necesarios. Quiero agradecer especialmente el trato recibido por Ramón, por integrarme en su grupo hace ya 21 años (¡que rápido pasa el tiempo!), darme una segunda oportunidad y mantener su interés en que este trabajo y mi carrera salieran adelante. Y también muchísimas gracias a Emilio, por su ayuda en la programación de los algoritmos y por la aportación de las escenas que hemos utilizado en las pruebas realizadas en la Tesis.

Debo agradecer además la financiación de mi labor investigadora a través de los proyectos TIC2001-3694-C02-02 y TIN2004-07797-C02-02 del Ministerio de Ciencia y Tecnología, TIN2007-67537-C03 del Ministerio de Ciencia e Innovación, TIN2010-16735 del Ministerio de Educación y Ciencia, y PIDT01PXI10501PR, PGIDIT03-TIC10502PR, 08TIC001206PR, INCITE08PXIB105161PR y Consolidación de Grupos de Investigación Competivivos ref. 2006/3 de la Xunta de Galicia, junto a las ayudas recibidas de la Red Gallega de Computación de Altas Prestaciones (G-HPC) y de la Universidade da Coruña. También debo dar las gracias al Centro de Supercomputación de Galicia (CESGA) por el acceso al supercomputador *Finis Terrae* que se han empleado en la elaboración de este trabajo, así como la ayuda prestada cuando surgieron complicaciones.

También me gustaría tener un recuerdo para todas aquellas personas que han contribuido en mayor o menor medida a que yo haya llegado hasta aquí, desde los profesores y compañeros de la Facultad de Físicas de la Universidad de Santiago de Compostela, pasando por los componentes del Departamento de Electrónica y Computación de la Universidad de Santiago de Compostela y los del Centro Nacional de Biotecnología de Madrid, hasta todos mis compañeros actuales en el Departamento de Electrónica y Sistemas de la Universidade da Coruña. Hay muchas personas a las que les debo algo, aunque algunas de ellas no sean conscientes, y sería injusto que me olvidara de alguna al enumerarlas. Y, por supuesto, gracias también a todos mis amigos y a mi familia.

A Adriana, o meu complemento,
e Rodrigo, o noso proxecto.

“El hombre nunca sabe de lo que es capaz
hasta que lo intenta”

Charles Dickens

Índice general

1. Iluminación global **1**

1.1. Introducción 2

1.2. Iluminación local 4

1.3. Métodos de iluminación global 5

1.4. Método de radiosidad 7

1.5. Soluciones determinísticas para el método de radiosidad 11

1.5.1. El método de radiosidad jerárquica 12

1.5.2. El método de radiosidad progresiva 15

1.6. Soluciones estocásticas para el método de radiosidad 17

1.6.1. Relajación estocástica del método de radiosidad 18

1.6.2. Métodos de caminos discretos aleatorios para radiosidad . . . 22

1.6.3. Métodos de estimación de la densidad de fotones 24

1.7. Técnicas de aceleración 26

1.8. Estructura de la memoria 31

2. Radiosidad progresiva paralela: partición uniforme **33**

2.1. El método de radiosidad progresiva 34

2.2. Método de partición uniforme de una escena 38
2.2.1. División en subespacios 42
2.2.2. Distribución de datos entre los procesadores 43
2.2.3. Recorte de los polígonos 45
2.3. Paralelización del algoritmo 50
2.3.1. Cálculo de la radiosidad local 51
2.3.2. Disparo y propagación de un polígono en el resto de la escena 55
2.3.3. Comprobación de la convergencia 57
2.4. La partición de la escena como método para incrementar la localidad de los datos 60
2.4.1. Análisis de la reutilización de datos en el algoritmo de radiosidad progresiva 61
2.4.2. Transformación por bloques del algoritmo para optimizar la localidad de los datos 63
2.5. Resultados experimentales 65
2.5.1. Método de recorte de polígonos 66
2.5.2. Rendimiento de la implementación paralela 67
2.5.3. Análisis de la partición de la escena como método para incrementar la localidad de los datos 69
2.5.4. Calidad de la escena resultante 72

3. Radiosidad progresiva paralela: partición no uniforme 75
3.1. Partición no uniforme 76
3.2. Paralelización del algoritmo 83
3.3. Resultados experimentales 86

4. Método distribuido de Monte Carlo para radiosidad 89

4.1. Método iterativo incremental estocástico de Jacobi 90

4.2. Método iterativo incremental estocástico de Jacobi distribuido 96

4.2.1. Métodos de partición convexa de la escena 101

4.2.2. Estrategia basada en el empaquetamiento de rayos 103

4.2.3. Procedimiento para comprobar el final de una iteración 107

4.3. Resultados experimentales 109

5. Implementación GPU del método de Monte Carlo para radiosidad 123

5.1. Introducción a las GPUs 124

5.1.1. Evolución de las GPUs 125

5.1.2. Comparación entre CPU y GPU 128

5.2. Arquitectura NVIDIA Tesla de la GPU 129

5.3. CUDA 132

5.4. Proyección GPU del MCR con CUDA 139

5.4.1. Partición de la escena 139

5.4.2. Simplificación de la malla de elementos 142

5.4.3. Organización de las tareas 144

5.5. Resultados experimentales 147

Conclusiones y principales aportaciones 155

Bibliografía 160

Índice de tablas

2.1. Campos de un vértice en la lista de vértices de recorte. 47

2.2. Tabla de comprobación de la convergencia para 4 procesadores. . . . 59

4.1. Ejemplo de paquetes de rayos intercambiados entre los procesadores. . 108

4.2. Distribución de la carga computacional para una configuración de 32 procesadores. 116

4.3. Comparación entre el tiempo de ejecución total y el de cada etapa del algoritmo distribuido para la escena *salón*. 117

5.1. Tiempos de ejecución en segundos para las implementaciones en CPU y GPU. 153

5.2. Parámetros de configuración: Nd y k. 154

Índice de figuras

1.1. Definición de radiancia $L(x, \Theta)$ 3

1.2. Representación geométrica de la *BRDF* 4

1.3. Componentes de la *BRDF* 6

1.4. Diferencial de ángulo sólido. 8

1.5. Cálculo del factor de forma. 10

1.6. Interpretación física de la ecuación clásica de la radiosidad. 11

1.7. Radiosidad jerárquica: refinamiento de las interacciones. 14

1.8. *Acumulación* de energía frente a *disparo* de energía 15

1.9. Método iterativo de Jacobi 20

2.1. Estructura del algoritmo de radiosidad progresiva. 36

2.2. Construcción de una lista de candidatos a obstáculos mediante un corredor (*shaft culling*) 39

2.3. Partición recursiva *vs.* cálculo *a priori* de las divisiones. 42

2.4. División uniforme de una escena en cuatro subescenas. 44

2.5. Cuatro posibles casos de pertenencia de un polígono a una subescena 45

2.6. Orden en el recorrido de los vértices de un polígono. 46

2.7. Ejemplos de vértices en el recorte de un polígono respecto de un subespacio 47

2.8. Resultados de la primera y segunda fase del método de recorte de los polígonos 49

2.9. Inserción de una intersección de arista durante la segunda fase el método de recorte. 50

2.10. Algoritmo de radiosidad progresiva distribuido. 52

2.11. Correspondencia de elementos en fronteras enfrentadas. 53

2.12. Cálculo de la máscara de visibilidad inicial. 54

2.13. Máscara intermedia de visibilidad. 56

2.14. Determinación de las máscaras de visibilidad de un polígono para cuatro procesadores. 58

2.15. Escenas iluminadas: (a) *habitación* y (b) *armadillo*. 65

2.16. Comparación del método de recorte usado con el algoritmo de Sutherland y Hodgman. 66

2.17. Resultados de las escenas de prueba: (a) *F. Ethernet Clúster* (b) *SCI Clúster*. 67

2.18. Tiempo de ejecución para la escena *armadillo*. 68

2.19. Tiempo de ejecución para *habitación*, *armadillo* y *habitación grande*. . 69

2.20. Fallos caché: (a) primer nivel; (b) segundo nivel. 70

2.21. Tasa de fallos de datos: (a) primer nivel; (b) segundo nivel. 70

3.1. Árbol de búsqueda binaria 79

3.2. Búsqueda de la posición del plano de división (coordenada x). 80

3.3. Ejemplo del método de división no uniforme para 8 particiones. . . . 81

3.4. División no uniforme de una escena en cuatro subescenas. 82

3.5. Jerarquía de procesadores. 84

3.6. Determinación de las máscaras de visibilidad de un triángulo para cuatro procesadores con la partición no uniforme. 85

3.7. Escenas iluminadas: (a) *habitación* y (b) *armadillos*. 86

3.8. Función de distribución para los métodos uniforme y non uniforme. . 87

3.9. Comparación en términos de *speedup* entre división uniforme y no uniforme 88

4.1. Método iterativo incremental estocástico de Jacobi en acción. 92

4.2. Algoritmo iterativo incremental estocástico de Jacobi. 95

4.3. Algoritmo iterativo incremental estocástico de Jacobi distribuido. . . 99

4.4. Partición realizada sobre una escena: (a) partición convexa, (b) partición no convexa. 101

4.5. Escenas iluminadas: (a) *edificio*, (b) *salón* y (c) *estudio*. 110

4.6. *Speedup* usando división uniforme. 111

4.8. *Speedup* y eficiencia para la división uniforme y no uniforme usando área y potencia. 115

4.9. Dependencia del *speedup* con el tamaño del paquete de rayos 118

4.10. Dependencia de la eficiencia con la frecuencia de chequeo de recepción de paquetes. 119

4.11. Comparación del *speedup* según cuando se hace el chequeo de paquetes recibidos: (a) escena *edificio* y (b) escena *salón*. 121

5.1. Estructura de una GPU según: (a)Direct3D10 y (b) Direct3D11. . . 127

5.2. Esquemas de las diferentes filosofías de diseño de CPUs y GPUs. . . 128

5.3. Arquitectura Tesla: (a)*Streaming Multiprocessor* (SM) y (b) *Texture Processor Cluster* (TPC). 129

5.4. Estructura de la serie 200 de la GPU GeForce GTX. 130
5.5. Arquitectura Fermi: (a) SM (b) Esquema. 132
5.6. Un *kernel* en CUDA. 133
5.7. Jerarquía de tareas. 134
5.8. Uso de la memoria de texturas. 138
5.9. Métodos de división de la escena: uniforme, no uniforme, híbrido . . . 141
5.10. Implementación CUDA del algoritmo Monte Carlo para radiosidad. . 144
5.11. Múltiples rayos alcanzan el mismo elemento. 147
5.12. Escenas iluminadas: (a) *edificio*, (b) *salón* y (c) *estudio*. 148
5.15. Tiempos de ejecución para GPU con las tres técnicas de división . . . 151

Capítulo 1

Iluminación global

Hoy en día muchos de los fenómenos que se pueden observar en la naturaleza, o muchas de las situaciones de la vida habitual pueden ser recreados de una manera artificial en un ordenador de una forma muy realista, debido al avance en el hardware gráfico y al desarrollo de un amplio conjunto de algoritmos de síntesis de imágenes realistas.

La síntesis de imágenes realistas es el proceso de crear imágenes sintéticas que sean indistinguibles de imágenes (por ejemplo fotografías) del mundo real. Las primeras técnicas para esta propuesta estaban limitadas por la tecnología disponible y no eran capaces de obtener los efectos deseados. No fue hasta los años 80 del siglo veinte que el avance de la tecnología permitió la introducción de nuevos algoritmos de síntesis de imágenes. Así aparecieron las técnicas de iluminación local y global procedentes de otros campos científicos como la óptica o la termodinámica. Con estos métodos comenzaron a mejorar los resultados debido a la utilización del conocimiento físico de la naturaleza de la luz para simular su comportamiento.

En los últimos años se ha vivido una explosión en el uso de la computación gráfica y en la creación de imágenes realistas. Por ejemplo, en la industria del entretenimiento se puede ver que las películas hacen un uso extensivo de efectos generados por computador, o que los juegos de ordenadores y consolas presentan mundos cada vez más cercanos a la realidad. Pero también este desarrollo se emplea en otros campos como en el diseño, la arquitectura, la sanidad, la educación o la publicidad.

En este capítulo se van a describir los principios básicos de los modelos de iluminación global, centrándose en el método de radiosidad que es el que de una manera

más realista permite modelar el equilibrio que se alcanza en la distribución de la luz en un entorno. Dentro del método de radiosidad se presentarán los dos algoritmos en los que se ha centrado este trabajo: el algoritmo de radiosidad progresiva, como solución determinística, y el método de Monte Carlo para radiosidad, como solución estocástica. De esta forma, se comienza con una introducción a conceptos de iluminación (Sección 1.1); a continuación se hablará sobre la iluminación local (Sección 1.2) y la iluminación global (Sección 1.3); en la Sección 1.4 se describe el método de radiosidad; en las dos siguientes secciones se comentarán las soluciones determinísticas del método de radiosidad (Sección 1.5) y las soluciones estocásticas (Sección 1.6); la Sección 1.7 está dedicada a hacer un repaso sobre las distintas técnicas de mejora del rendimiento computacional de los algoritmos explicados en las secciones anteriores; finalmente, en la Sección 1.8 se indica la estructura de esta memoria.

1.1. Introducción

La simulación de la dispersión de la luz en un modelo sintético basada en propiedades físicas se realiza con los modelos de iluminación global [2, 30]. La meta de los métodos de iluminación global es calcular la distribución de la energía luminosa en una escena en un estado estable, simulando todas las interacciones de la luz con los objetos de la escena y prediciendo con exactitud la intensidad de la luz en cualquier punto de esta. Los datos iniciales del algoritmo de iluminación global son la descripción de la geometría, de los materiales y de las fuentes de luz de la escena, y su cometido es calcular como interacciona la luz que sale de las fuentes luminosas con el resto de la escena.

El estudio de los métodos de iluminación global debe comenzarse explicando algunas magnitudes físicas relacionadas con la energía luminosa. Un concepto fundamental es la potencia radiante que representa la cantidad de energía que fluye desde, hacia o a través de una superficie por unidad de tiempo. Se denota por Φ y se expresa en *vatios* (*julios*/*segundo*). Considerando la potencia radiante que sale de una superficie, se define la radiosidad B como la potencia radiante de salida por unidad de área (expresada en $vatios/m^2$):

$$B = \frac{d\Phi}{dA} \tag{1.1}$$

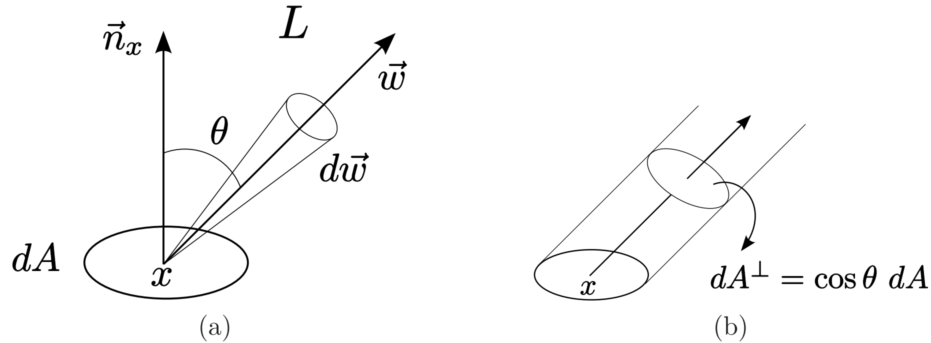


Figura 1.1: (a) Definición de radiancia $L(x, \vec{w})$; (b) definición de área proyectada $dA^{\perp}$.

Otro concepto es la radiancia L que se define como la potencia radiante por unidad de área proyectada y por unidad de ángulo sólido ($vatios/(estereorradianes{\cdot}m^2)$). En la Figura 1.1a se muestra una representación gráfica de la radiancia L en un punto x de la superficie dA en la dirección $\vec{w}$, donde θ es el ángulo que forma $\vec{w}$ con la normal a la superficie $\vec{n}_x$ en el punto x. El área proyectada $dA^{\perp}$ sería el resultado de proyectar la superficie dA en un plano perpendicular a la dirección $\vec{w}$ (ver Figura 1.1b). La radiancia es, por tanto, una magnitud de cinco dimensiones que varía con la posición y con la dirección y que puede expresarse como:

$$L(x, \vec{w}) = \frac{d^2\Phi}{d\vec{w} \; dA^{\perp}} = \frac{d^2\Phi}{d\vec{w} \; \cos\theta \; dA} \tag{1.2}$$

donde $L(x, \vec{w})$ representa la radiancia que sale del punto x en la dirección $\vec{w}$, y $d\vec{w}$ es el diferencial de ángulo sólido en esa dirección.

Intuitivamente, la radiancia expresa cuanta potencia sale de o llega a un cierto punto de una superficie, por unidad de ángulo sólido y por unidad de área proyectada. La radiancia es, probablemente, la variable física más importante en los algoritmos de iluminación global, ya que es el valor más relacionado con la “apariencia” de los objetos en la escena [30]. En el vacío, la radiancia es constante a lo largo de una línea recta de visión. Esta es una propiedad de gran importancia que se aplica en algoritmos que utilizan el trazado de rayos. La excepción a esta regla son los medios participativos (la niebla, por ejemplo) y los medios no-lineales (donde el índice de refracción varía, como en el fenómeno de los espejismos en el desierto). Se puede relacionar la potencia radiante y la radiosidad con la radiancia de la siguiente

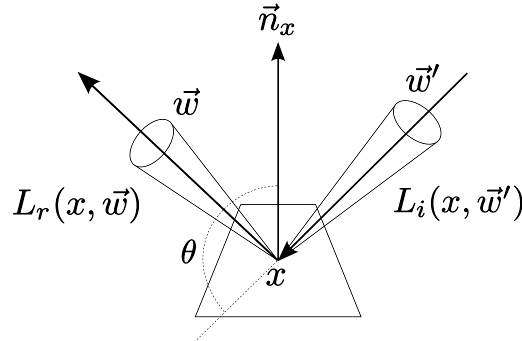


Figura 1.2: Representación geométrica de la *BRDF*.

forma:

$$\Phi = \int_A \int_\Omega L(x,\vec{w}) \cos\theta \; d\vec{w} \; dA_x \tag{1.3}$$

$$B(x) = \int_\Omega L(x,\vec{w}) \cos\theta \; d\vec{w} \tag{1.4}$$

1.2. Iluminación local

Normalmente cuando la luz alcanza un material penetra en él, es dispersada y abandona la superficie por un punto diferente. Lo que ocurre cuando un rayo de luz choca con una superficie se simula con los métodos de *iluminación local* en computación gráfica. Puede describirse esta iluminación local usando la función de distribución de la reflectancia bidireccional de superficie dispersante (*Bidirectional Scattering Surface Reflectance Distribution Function, BSSRDF*). Debido a la complejidad de esta función, se ha definido una aproximación a la BSSRDF que es la función de distribución de reflectancia bidireccional de un material (*Bidirectional Reflectance Distribution Function, BRDF*)[85], en la cual se asume que la luz que incide en un punto en una superficie es reflejada desde el mismo punto. La BRDF, f_r, (ver Figura 1.2) define la relación entre la radiancia reflejada, L_r, y la potencia radiante por unidad de superficie, L_i, que incide en un punto determinado x, según las direcciones de llegada $\vec{w}'$ y salida $\vec{w}$:

$$f_r(x,\vec{w}',\vec{w}) = \frac{dL_r(x,\vec{w})}{L_i(x,\vec{w}')(\vec{w}' \cdot \vec{n}_x) d\vec{w}'} \tag{1.5}$$

donde $\vec{n}_x$ es la normal en x.

Los modelos de iluminación local utilizan la BRDF [2, 68]. Conocida la radiancia incidente en un punto de la superficie, se puede calcular la radiancia reflejada en todas las direcciones mediante la siguiente integral:

$$L_r(x, \vec{w}) = \int_{\Omega_x} f_r(x, \vec{w}', \vec{w}) L_i(x, \vec{w}')(\vec{w}' \cdot \vec{n}_x) d\vec{w}' \tag{1.6}$$

donde Ω_x es el hemisferio de todas las direcciones de llegada a x, y $\vec{w}' \cdot \vec{n}_x = \cos\theta$, siendo θ el ángulo que forma la normal $\vec{n}_x$ con el vector de la dirección del rayo incidente $\vec{w}'$. Una propiedad importante de la BRDF es la ley de Helmholtz de la reciprocidad, que establece que la BRDF es independiente de la dirección en la que fluye la luz:

$$f_r(x, \vec{w}', \vec{w}) = f_r(x, \vec{w}, \vec{w}') \tag{1.7}$$

Otra importante propiedad física de la BRDF es la conservación de la energía, que indica que una superficie nunca puede reflejar más energía de la que le llega, con lo cual debe cumplirse la siguiente relación:

$$\int_{\Omega_x} f_r(x, \vec{w}', \vec{w})(\vec{w}' \cdot \vec{n}_x)\ d\vec{w}' \ < \ 1 \ , \ \forall\ \vec{w}' \tag{1.8}$$

1.3. Métodos de iluminación global

La base matemática para todos los métodos de iluminación global es la ecuación de *rendering* [60]. Esta ecuación establece las condiciones de equilibrio para el transporte o distribución de la luz en un entorno sin medios participativos. La esencia es la conservación de la energía, e indica que la radiancia de salida L_o en un punto x y en una dirección $\vec{w}$ es la suma de la radiancia emitida L_e (emitancia) y la reflejada L_r. Usando la Ecuación 1.6, se obtiene la expresión formal clásica de la ecuación de *rendering*:

$$L_o(x, \vec{w}) = L_e(x, \vec{w}) + \int_{\Omega_x} f_r(x, \vec{w}', \vec{w}) L_i(x, \vec{w}')(\vec{w}' \cdot \vec{n}_x) d\vec{w}' \tag{1.9}$$

En la práctica, el comportamiento de la luz cuando interacciona con un objeto, modelado por la BRDF, puede descomponerse en varios términos más sencillos, representando cada uno de ellos un aspecto determinado de la interacción de la luz con el material, el cual puede aparecer por ejemplo como una superficie difusa o un espe-

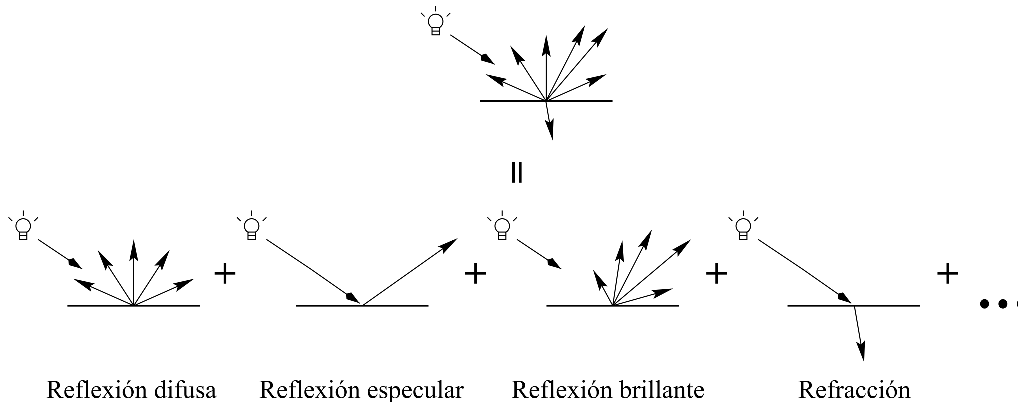


Figura 1.3: Componentes de la *BRDF*.

jo o un cristal (ver Figura 1.3). Los modelos de iluminación global suelen centrarse en uno o varios de estos efectos, simplificando la función que da el comportamiento físico real de la luz.

Se han desarrollado varios algoritmos de iluminación global para tratar de resolver, normalmente de forma aproximada mediante una serie de simplificaciones, la ecuación de *rendering*, pero la mayoría de ellos están basados en dos técnicas [2, 30, 59]:

- *Ray tracing* [41]: es un método de muestreo basado en el lanzamiento de rayos a través de la escena desde el punto de vista del observador, mediante los cuales se estima la iluminación.
- Radiosidad [20, 121]: asumiendo materiales difusos, este método está basado en el uso de elementos finitos que servirán de base para el cálculo de la distribución final de la luz mediante la resolución de un conjunto de ecuaciones lineales.

El método básico de *ray tracing* sólo puede usarse para el cálculo de reflexiones, refracciones e iluminación directa. Sin embargo, con la incorporación de métodos estocásticos, como Monte Carlo, se pueden simular todo tipo de efectos de la luz. El inconveniente es que para la eliminación de ruido en las imágenes generadas se requiere un gran número de rayos, por lo que se consume un elevado tiempo de ejecución.

El método de radiosidad se introdujo inicialmente para escenas con materiales difusos [21, 43] donde se asume que la reflexión se puede representar por una constante, independientemente de la dirección. Más tarde se introdujeron extensiones

para manejar modelos más complejos. El método básico calcula la iluminación global de forma totalmente independiente del punto de vista del observador, pero es costoso en cuanto a tiempo de computación y requerimientos de memoria. Para mejorar la eficiencia se han presentado propuestas como realizar los cálculos desde el punto de vista del observador [127], usar *clustering* para simplificar la complejidad de los objetos [126], o utilizar técnicas jerárquicas [51]. Por otro lado, el algoritmo de la radiosidad se basa en la utilización de una malla finita de elementos que ha de construirse cuidadosamente si se quiere obtener una buena solución. Además, el algoritmo tiene problemas cuando hay cambios abruptos de iluminación, como en la representación de las sombras. Para combinar lo mejor del método de la radiosidad con el de *ray tracing* han sido desarrolladas técnicas híbridas.

El método de radiosidad tiene una serie de propiedades que lo hacen interesante para un amplio número de aplicaciones, como por ejemplo:

- Proporciona una solución de la iluminación global totalmente independiente del punto de vista del observador.
- Simula especialmente bien el comportamiento difuso de la luz permitiendo obtener resultados realistas en la iluminación lograda. Por tanto, esto lo hace ideal para la creación de entornos pre–iluminados para sistemas de realidad virtual.
- Es muy adecuado para aplicaciones interactivas, ya que al ser independiente del punto de vista del observador, los resultados obtenidos pueden ser reutilizados para visitas virtuales (*walk–throughs*) a través de la escena.

El principal inconveniente de este método es la dependencia que establece entre la iluminación de una escena y su geometría, lo que limita, en principio, la complejidad de las escenas a utilizar. Debido a sus importantes ventajas se ha escogido este método para este trabajo. A continuación es descrito más detalladamente.

1.4. Método de radiosidad

El método de la radiosidad es una herramienta computacional que simula la distribución de la luz en una escena tridimensional. Para ello, primero se divide la escena en un conjunto de elementos, procediendo a computar a continuación la

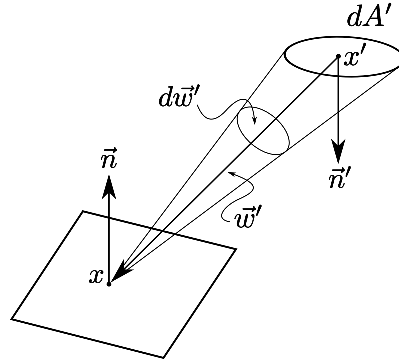


Figura 1.4: Diferencial de ángulo sólido.

solución de una ecuación de balance de energía con coeficientes acoplados llamados *factores de forma*. Realmente la idea del método, que se inició en 1984 [43] poco después de la aparición del método clásico de *ray-tracing*, proviene de la aplicación de la resolución de problemas de transferencia de calor mediante *métodos de elementos finitos* a la generación de imágenes sintéticas. Con este fin, la radiosidad clásica únicamente tiene en cuenta la componente difusa de la reflexión de la luz, necesitando el complemento de otras técnicas para añadir la simulación de efectos especulares de otro tipo.

La ecuación de *rendering* (Ecuación 1.9) normalmente se expresa como una integral sobre puntos de la superficie para la aplicación de métodos de elementos finitos:

$$L_o(x, \vec{w}) = L_e(x, \vec{w}) + \int_S f_r(x, x' \rightarrow x, \vec{w}) L_i(x' \rightarrow x) V(x, x') G(x, x') dA' \qquad (1.10)$$

donde $L_i(x' \rightarrow x)$ indica la radiancia que se emite desde el punto x' hacia x, S es el conjunto de todos los puntos de las superficies de la escena y $V(x, x')$ es una función de visibilidad que vale 1 si x y x' son mutuamente visibles y 0 si no lo son. $G(x, x')$ es un término geométrico que indica el porcentaje de energía que se transmite desde el punto x' hasta el punto x, en función de la distancia a la que se encuentran y la orientación relativa de las superficies a las que pertenecen dichos puntos:

$$G(x, x') = \frac{(\vec{w}' \cdot \vec{n}')(\vec{w}' \cdot \vec{n})}{\|x' - x\|^2} \qquad (1.11)$$

donde $\vec{n}$ y $\vec{n}'$ son las normales a las superficies en los puntos x y x', respectivamente.

Para pasar de la Ecuación 1.9 a la Ecuación 1.10 se usa la siguiente fórmula para

el diferencial de ángulo sólido (ver Figura 1.4):

$$d\vec{w}'(x) = \frac{(\vec{w}' \cdot \vec{n}')dA'}{\|x' - x\|^2} \tag{1.12}$$

Una estrategia para resolver la ecuación de *rendering* es simplificar el problema, asumiendo que todas las superficies son reflectores difusos ideales o *Lambertianas*, es decir, que la radiancia reflejada es constante en todas las direcciones. En este caso la BRDF sería:

$$f_r(x, x' \to x, \vec{w}) = \frac{\rho(x)}{\pi} \tag{1.13}$$

siendo $\rho(x)$ el coeficiente de reflectancia, que representa la fracción de energía incidente para cada longitud de onda que es reflejada en el punto x de la superficie. Teniendo en cuenta esto se puede reemplazar el cálculo de la radiancia por un término más simple de radiancia emitida, también llamada radiosidad, B. Esto simplifica la Ecuación 1.10:

$$B(x) = E(x) + \frac{\rho(x)}{\pi} \int_S B(x')V(x, x')G(x, x')dA', \tag{1.14}$$

donde E es la radiosidad autoemitida o emitancia.

El algoritmo de radiosidad de elementos finitos normalmente resuelve esta integral transformándola en un sistema aproximado de ecuaciones lineales por medio de un procedimiento llamado discretización de Galerkin [20, 121]. Esto se hace eligiendo *Ne* elementos con radiosidad constante, dando como resultado:

$$B_i = E_i + \rho_i \sum_{j=1}^{Ne} B_j F_{ij} \tag{1.15}$$

donde B_i es la radiosidad para el elemento S_i, ρ_i es el coeficiente de reflectancia difusa para este elemento y F_{ij} es el *factor de forma*. Este valor, F_{ij}, representa la fracción de radiosidad de S_i cuyo origen es S_j y se calcula como:

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{V(x, x')G(x, x')}{\pi} dA_j dA_i \tag{1.16}$$

En la Figura 1.5 se muestra una representación gráfica de los parámetros necesarios para el cálculo del factor de forma entre dos elementos S_i y S_j de áreas dA_i y dA_j, respectivamente.

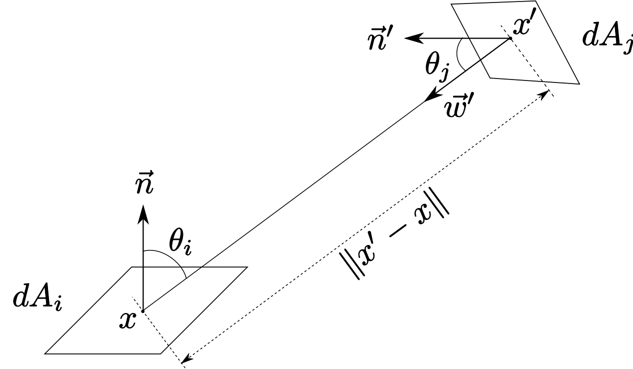


Figura 1.5: Cálculo del factor de forma.

La Ecuación 1.15 se puede reescribir en forma matricial de la siguiente manera:

$$\underbrace{\begin{pmatrix} 1-\rho_1 F_{11} & -\rho_1 F_{12} & \cdots & -\rho_1 F_{1Ne} \\ \rho_2 F_{21} & 1-\rho_1 F_{22} & \cdots & -\rho_1 F_{2Ne} \\ \vdots & \vdots & \vdots & \vdots \\ \rho_{Ne} F_{Ne1} & -\rho_{Ne} F_{Ne2} & \cdots & 1-\rho_{Ne} F_{NeNe} \end{pmatrix}}_{\mathbf{K}} \underbrace{\begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_{Ne} \end{pmatrix}}_{\mathbf{B}} = \underbrace{\begin{pmatrix} E_1 \\ E_2 \\ \vdots \\ E_{Ne} \end{pmatrix}}_{\mathbf{E}} \tag{1.17}$$

donde los coeficientes **B** y **E** son dos vectores con los valores de radiosidad y de emitancia, respectivamente, de todos los elementos de la escena, y **K** corresponde a la matriz de interacciones entre los distintos elementos.

La idea básica del método clásico de radiosidad es calcular la radiosidad promedio B_i en cada elemento S_i de la superficie de un modelo tridimensional. Habitualmente estos elementos son triángulos o cuadriláteros convexos, aunque se han explorado otras alternativas [3]. A cada elemento S_i se le asocia una emitancia E_i y una reflectancia ρ_i. La emitancia es la radiosidad que el elemento emite por sí mismo, incluso aunque no haya otros elementos en la escena o que el resto tenga un coeficiente de reflectancia nulo. Estos datos son suficientes para calcular la radiosidad total emitida B_i por cada elemento, que es la contribución tanto de la recibida del resto de la escena como de la autoemitida, como se indica en la Figura 1.6.

Las ecuaciones que relacionan a B_i con E_i y ρ_i para todos los elementos son resueltas y las radiosidades resultantes se transforman para visualizar los colores de las superficies. Debido a que sólo se calcula la iluminación difusa, los colores de las superficies son independientes del punto de vista de la escena.

Realmente el valor B_i que se obtiene después de resolver el sistema de ecuaciones

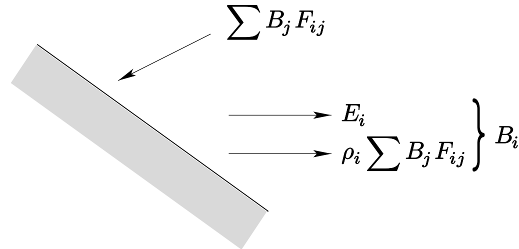


Figura 1.6: Interpretación física de la ecuación clásica de la radiosidad.

lineales es sólo una aproximación de la radiosidad promedio, ya que raramente la radiosidad es constante en cada elemento. Sin embargo, la diferencia entre B_i y el valor promedio real de la radiosidad en el elemento es difícilmente visible en la práctica.

Teniendo en cuenta que la radiosidad y la potencia radiante están relacionadas por la ecuación $P_i = A_i \cdot B_i$ y que el factor de forma satisface la relación de reciprocidad $A_i \cdot F_{ij} = A_j \cdot F_{ji}$, se puede reescribir la Ecuación 1.15 como:

$$P_i = P_{ei} + \rho_i \sum_{j=1}^{Ne} P_j F_{ji} \tag{1.18}$$

Este sistema de ecuaciones establece que la potencia P_i emitida por el elemento S_i consta de dos partes: una potencia autoemitida P_{ei} y la potencia recibida y reflejada desde otros elementos S_j. En este caso el factor de forma, F_{ji}, representa la fracción de potencia emitida por el elemento S_j que llega al elemento S_i.

Mucho esfuerzo investigador se ha desarrollado sobre la manera de resolver la ecuación de radiosidad eficientemente. En [20, 30, 121] se pueden consultar unas excelentes revisiones de estos métodos. Básicamente hay dos orientaciones principales: resolver las ecuaciones de una manera determinística o mediante métodos estocásticos. A continuación serán comentados los métodos más utilizados en la bibliografía.

1.5. Soluciones determinísticas para el método de radiosidad

La resolución del sistema de ecuaciones que propone el método de radiosidad se realiza, normalmente, mediante un proceso iterativo, ya que la complejidad tanto

computacional como de almacenamiento es $O(Ne^2)$, lo que hace inviable un cálculo directo de la solución para escenas grandes.

Además, debido a esa complejidad, es muy difícil utilizar las técnicas habituales para la resolución iterativa de sistemas de ecuaciones lineales y se han desarrollado distintas alternativas, entre las que destacan dos por ser las más empleadas: la aplicación de soluciones jerárquicas [51] análogas a las que han dado buenos resultados en el problema de los N-cuerpos en la física gravitacional [124], y el método de la radiosidad progresiva [19], en el que se aplica una variante de la relajación de Southwell [44, 81].

1.5.1. El método de radiosidad jerárquica

El método jerárquico [51] reduce la complejidad inicial del método clásico de radiosidad utilizando una subdivisión adaptativa de los elementos de la escena basada en las distribución de la energía en la misma. De esta forma, se tienen más elementos donde los cambios de luz son más bruscos, consiguiendo muy buenos resultados por ejemplo en las zonas próximas a las sombras. Sin embargo, este algoritmo tiene un coste computacional y de almacenamiento elevado cuando las escenas aumentan su complejidad, fundamentalmente por el cálculo de los factores de forma y el aumento del número de elementos en la escena. Para intentar resolver estos problemas se han propuesto varias soluciones, como la implementación paralela en sistemas de memoria distribuida [35, 76, 92, 94], la utilización de métodos de multirresolución [29, 96], o la aplicación de técnicas de *clustering* [52, 126], entre otras.

El método de radiosidad jerárquica parte de un mallado de la escena más grueso que el de otros métodos de radiosidad, refinando los elementos *a posteriori* durante el cálculo iterativo de la radiosidad. Este refinamiento se hace de forma adaptativa en función de la energía transportada por las distintas interacciones, creando una jerarquía de subdivisiones de los elementos iniciales de la escena, con la correspondiente jerarquía de interacciones asociada.

La aproximación jerárquica permite resolver el cálculo de la radiosidad de la escena rebajando drásticamente la complejidad $O(Ne^2)$ de la matriz de interacción clásica debido, en primer lugar, a la sustitución de interacciones finas innecesarias por otras más gruesas. En segundo lugar, la utilización de un criterio de refinado de los elementos de la escena, basado en la distribución de la luz en la misma, permite que el número final de elementos obtenido con la subdivisión adaptativa sea mucho

menor que el número de elementos de los que inicialmente se partiría en un esquema de radiosidad clásico, para la obtención de un resultado equivalente. De esta manera, la complejidad del método jerárquico de radiosidad, para un conjunto inicial de Ne_G elementos gruesos, y alcanzando un número final, tras la iluminación de la escena, de Ne elementos, con $Ne_G \ll Ne$, es del orden de $O(Ne + Ne_G^2)$, tendiendo a $O(Ne)$ a medida que aumenta la complejidad de la escena a procesar.

El método iterativo habitualmente aplicado en radiosidad jerárquica es Gauss–Seidel, lo que equivale a recorrer secuencialmente los elementos de la escena, acumulando en cada uno de ellos la energía recibida por parte de los elementos de la escena con los que interactúa. Los tres pasos típicos que se distinguen en una iteración del método de radiosidad jerárquica sobre cada elemento de la escena son: refinado (*refinement*), propagación de la energía recibida por el elemento (*gathering*) y sincronización de su jerarquía (*sweeping*).

El núcleo del método jerárquico es la etapa de refinado, ya que es la fase en la que se toma la decisión de considerar una interacción entre dos elementos suficientemente precisa o, por el contrario, de proceder a su refinamiento. La heurística más habitual para el refinado jerárquico es la que se conoce como refinado BF [51], llamado así porque se utiliza una estimación del producto de la radiosidad (B) por el factor de forma (F) para determinar cuando se realiza el refinamiento. Aplicando este método una interacción entre dos polígonos necesita ser refinada cuando:

1. Esa interacción transporta una cantidad excesiva de energía ($B_j \cdot F_{ij}$), de acuerdo a un determinado umbral previamente fijado para la escena. Influyen en el refinado, por tanto, el factor de forma de la interacción y la radiosidad que desprende la fuente de la interacción.

2. La relación de visibilidad entre los polígonos que están interactuando no está claramente definida, $0 < V_{ij} < 1$. Es decir, las interacciones correspondientes a elementos que no son completamente visibles, ni están totalmente obstaculizados, tienen que ser refinadas. Esto supondrá un mayor nivel de detalle en la iluminación de las zonas de penumbra.

Si se decide refinar una interacción uno o los dos elementos involucrados son subdivididos, creando nuevas interacciones entre los elementos resultantes de la subdivisión. En la Figura 1.7 se puede ver un ejemplo, en el cual la interacción entre los elementos S_1 y S_2 es refinada dividiendo primero S_1 en los elementos S_{11}, S_{12}, S_{13} y S_{14}, y S_2 en S_{21}, S_{22}, S_{23} y S_{24}. A continuación se hace una nueva subdivisión

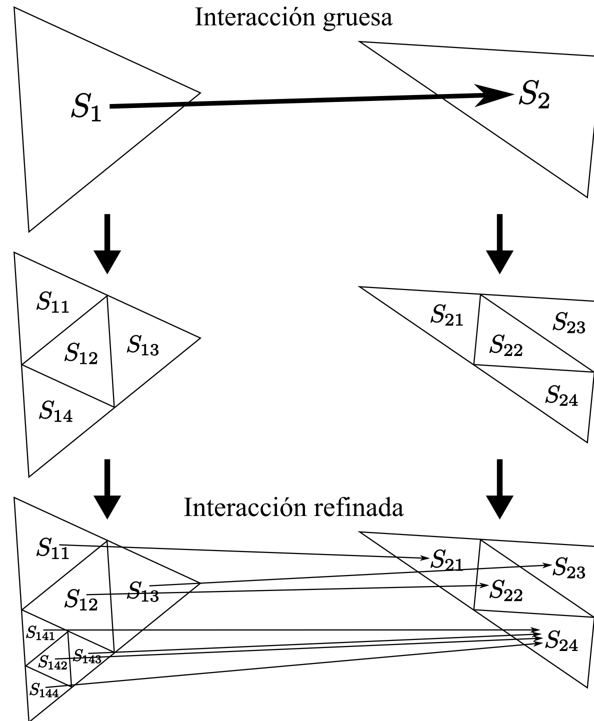


Figura 1.7: Radiosidad jerárquica: refinamiento de las interacciones.

del elemento S_{14} en los elementos S_{141}, S_{142}, S_{143} y S_{144}. Finalmente, la interacción gruesa inicial $I(S_1 - S_2)$ es sustituida por las interacciones refinadas $I(S_{11} - S_{21})$, $I(S_{12} - S_{22})$, $I(S_{13} - S23)$, $I(S_{141} - S24)$, $I(S_{142} - S_{24})$, $I(S_{143} - S_{24})$ e $I(S_{144} - S_{24})$.

Una vez que todas las interacciones de un elemento han sido evaluadas, el resultado es un árbol jerárquico de elementos surgidos de las sucesivas subdivisiones que recursivamente ha sido necesario realizar sobre el elemento grueso original, que será la raíz de esta jerarquía. A su vez, las interacciones gruesas iniciales, que componían una matriz simétrica, han sido sustituidas por una jerarquía de interacciones entre elementos de distintos niveles de refinado.

Finalizado el refinamiento de un elemento, se procede a la propagación por cada uno de los elementos de la jerarquía de la energía transmitida por los elementos con los que interactúa. De esta forma, la energía originalmente transportada por la interacciones gruesas del elemento raíz estará repartida entre los distintos elementos de su estructura jerárquica.

El tercer y último paso del proceso iterativo es la sincronización de la estructura jerárquica creada, que actualizará los valores de radiosidad en los distintos niveles del

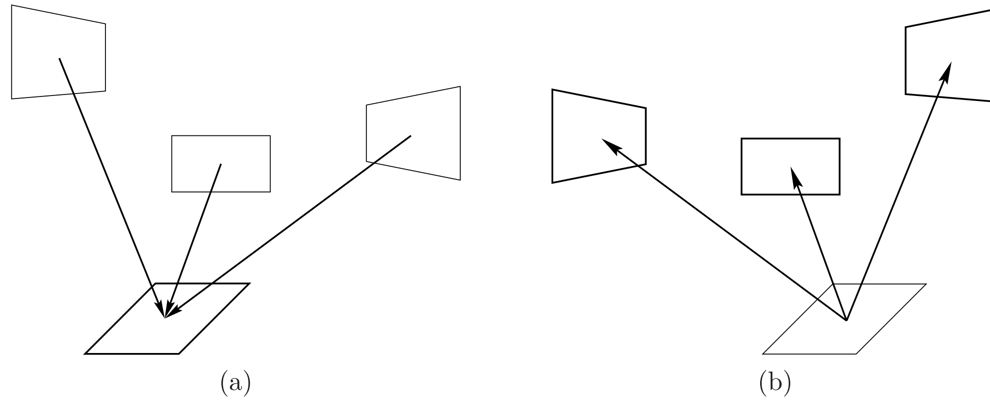


Figura 1.8: (a) *Acumulación* de energía en un elemento (*gathering*); (b) *disparo* de la energía de un elemento (*shooting*).

árbol, dando coherencia a toda la estructura. Para esto se recorrerá la estructura jerárquica y para cada elemento, por un lado, se añadirá la energía obtenida en ese nivel a sus descendientes, y por otro lado se le sumará a la suya propia una ponderación de la energía recibida por sus descendientes.

1.5.2. El método de radiosidad progresiva

En [19] se reformuló el algoritmo de radiosidad presentando el método de *refinamiento progresivo* o *radiosidad progresiva*. El objetivo de esta técnica es obtener una imagen útil después de cada iteración. En el método de radiosidad jerárquica, que utiliza la solución Gauss-Siedel, se produce un proceso de acumulación de energía recibida por cada uno desde todos los demás (*gathering*) como se muestra en la Figura 1.8a. De esta forma, una estimación de la radiosidad de todos los elementos no es disponible hasta que se completa una iteración recorriendo todos estos elementos en un orden fijo. En cambio, en el algoritmo progresivo en cada paso se escoge el elemento S_i que contribuye a la escena con más energía y su radiosidad es distribuida entre los otros elementos (*shooting*), como se puede ver en la Figura 1.8b. Esto hace que finalizada una iteración se tenga una estimación de la radiosidad de todos los elementos de la escena, sin la necesidad de haberlos recorrido todos. Específicamente, el algoritmo de radiosidad progresiva para resolver el sistema $\mathbf{K} \cdot \mathbf{B} = \mathbf{E}$ (Ecuación 1.17) define una variable adicional para cada elemento, ΔB_i, que representa la energía sin disparar. Una vez escogido el elemento a disparar como se ha comentado antes, el resto de elementos acumulan nueva energía ($\Delta B_j = \Delta B_j + \rho_j \cdot F_{ji} \cdot \Delta B_i$), que puede ser disparada en las siguientes iteraciones, en el caso de ser seleccionados.

Una vez finalizada la iteración, el elemento seleccionado pasa a no tener energía sin disparar ($\Delta B_i = 0$), y los valores de radiosidad se actualizan en todos los elementos con la radiosidad recibida ($B_j = B_j + \Delta B_j, \forall j$).

Como ha sido demostrado en [44] y [81], el algoritmo progresivo es un proceso iterativo equivalente a la combinación de dos técnicas de análisis numérico conocidas como el método de relajación de Southwell y el algoritmo iterativo de Jacobi. En concreto, en cada etapa del algoritmo progresivo se aplica primero el método de relajación de Southwell para los cálculos de los residuos que representan las nuevas energías que reciben los elementos desde el que ha sido disparado. A continuación, para finalizar cada paso, se realiza un barrido completo de una iteración del algoritmo de Jacobi para la actualización de la radiosidad de todos los elementos. A continuación se muestra brevemente como funcionan estos dos métodos.

Definiendo el vector del residuo $\mathbf{R} = \mathbf{E} - \mathbf{K} \cdot \mathbf{B}$ y la aproximación inicial $\mathbf{B}^0 = 0$, para obtener la solución del sistema el método de relajación de Southwell calcula en cada iteración la componente de valor máximo del vector del residuo, R_i^k. Con este valor se actualiza la componente correspondiente del vector B^{k+1}, $B_i^{k+1} = B_i^k + R_i^k$, y se recalcula el nuevo vector del residuo, $R_j^{k+1} = R_j^k - R_i^k \cdot K_{j\,i}$, $j = 1, \ldots, Ne$.

Por otro lado, se puede reescribir el sistema de ecuaciones anterior de la siguiente forma: $\mathbf{B} - (\mathbf{I} - \mathbf{K}) \cdot \mathbf{B} = \mathbf{E}$. Definiendo $\mathbf{C} = (\mathbf{I} - \mathbf{K})$, el sistema de ecuaciones se puede reformular como $\mathbf{B} - \mathbf{C} \cdot \mathbf{B} = \mathbf{E}$, o también como $\mathbf{B} = \mathbf{E} + \mathbf{C} \cdot \mathbf{B}$. Este tipo de sistemas de ecuaciones lineales puede ser resuelto mediante el método iterativo de Jacobi usando un sencillo esquema de iteración. La idea del método se basa en seleccionar primero un $\mathbf{B}^0$ arbitrario. Luego, durante cada iteración, $\mathbf{B}^k$ es transformado en $\mathbf{B}^{k+1}$ colocando a $\mathbf{B}^k$ en el lado derecho de las ecuaciones $\mathbf{B}^{k+1} = \mathbf{E} + \mathbf{C}\mathbf{B}^k$. Si la norma de $\mathbf{C}$ es menor que 1, entonces la secuencia de $\mathbf{B}^k$ siempre convergerá al mismo punto $\mathbf{B}$, la solución del sistema. Al punto $\mathbf{B}$ también se le llama punto fijo del esquema de iteración.

Las diferencias del algoritmo progresivo respecto del algoritmo de Southwell son:

- En el método de Southwell se actualiza una sola componente de $\mathbf{B}$ en cada paso, mientras que en el refinamiento progresivo se actualizan todas.
- En el método de Southwell se escoge en cada momento el elemento con el valor máximo del residuo, que en el algoritmo progresivo se correspondería con el elemento con más radiosidad sin disparar, ΔB_i. Sin embargo, en este último método la elección se hace a partir de la potencia sin disparar, $A_i \cdot \Delta B_i$, es decir, la radiosidad está ponderada por el área. Desde el punto de vista físico,

lo que se está haciendo es centrar la atención en aquellos elementos que emiten o reflejan la mayor cantidad de energía.

- Al terminar el algoritmo progresivo en vez de considerar el valor de $\mathbf{B}$, el resultado es $\mathbf{B}+\Delta\mathbf{B}$. La interpretación física es que cuando el algoritmo acaba se tiene la radiosidad sin disparar en $\Delta\mathbf{B}$, la cual hay que añadirla a la escena para dar un resultado más correcto. Desde el punto de vista numérico, también tiene sentido, ya que se está realizando un recorrido completo del método iterativo de Jacobi.

La elección del elemento a disparar es una ventaja respecto del método jerárquico, ya que eligiendo el elemento con la mayor potencia sin disparar se reduce en mayor medida la potencia total que queda por disparar. Conviene indicar que esto no significa que Southwell sea siempre mejor que Gauss-Seidel, pero en problemas de radiosidad la ventaja es más clara, ya que en las iteraciones iniciales la radiosidad sin disparar está concentrada en pocos elementos. De esta forma, Gauss-Seidel consume mucho tiempo en actualizar variables que no varían mucho, mientras que el método de Southwell de la radiosidad progresiva concentra su esfuerzo en elementos que sí lo hacen.

Otra ventaja respecto de Gauss-Seidel es el paso final de Jacobi, que no supone coste extra adicional por disponer del vector completo de radiosidades sin disparar. Esta operación final equivale a Ne iteraciones de “relajación”, consiguiendo evitar hacer muchas iteraciones asociadas a los algoritmos de Southwell o Gauss-Seidel.

El método progresivo finaliza cuando se alcanza un umbral mínimo de potencia sin disparar. En ese momento se habrán disparado Nk elementos, con lo cual la complejidad del algoritmo será de $O(Nk \times Ne)$, con $Nk \ll Ne$.

Para el estudio realizado en este trabajo, dentro de las soluciones determinísticas de la radiosidad se ha escogido el método de radiosidad progresiva. En los Capítulos 2 y 3 se presenta una descripción más detallada de este algoritmo, junto con la metodología realizada y los resultados obtenidos.

1.6. Soluciones estocásticas para el método de radiosidad

Otra posible aproximación para el cálculo de la radiosidad es el algoritmo de Monte Carlo. Los métodos de Monte Carlo [61] son de tipo estocástico y toman

muestras aleatorias del espacio asociado con el problema para obtener una solución aproximada. Estos métodos son utilizados para resolver una gran variedad de problemas en numerosas áreas como la física nuclear, la mecánica de fluidos, y son muy útiles en el análisis de riesgos, por ejemplo en las finanzas, la exploración espacial o la explotación de petróleo.

Los métodos de radiosidad que utilizan Monte Carlo (*MCR*, *Monte Carlo Radiosity*) comparten un hecho importante: no requieren el cálculo y almacenamiento de los factores de forma. Esto es posible porque los factores de forma pueden ser interpretados como probabilidades que pueden ser muestreadas eficientemente. La no necesidad de calcular y almacenar los factores de forma hace que estos métodos sean muy adecuados para trabajar con grandes escenas y también que sea más fácil su implementación [10, 30]. Un inconveniente es el coste computacional por la necesidad de generar un gran número de muestras para obtener un resultado aceptable, aunque se han analizado soluciones para reducir este coste mediante el reuso de muestras [16].

En esta sección se describen los tres tipos principales de algoritmos de radiosidad basados en muestreados estocásticos, utilizando métodos de Monte Carlo.

El primer tipo, llamados métodos de *relajación estocástica* [79, 80, 82], están basados en adaptaciones estocásticas de métodos clásicos iterativos de resolución de sistemas lineales, como los métodos de Jacobi, Gauss-Seidel o Southwell.

El segundo tipo, *caminos discretos aleatorios* para radiosidad [112, 113], surge de la aplicación de Monte Carlo a la resolución de sistemas lineales, basándose en la noción de camino discreto aleatorio.

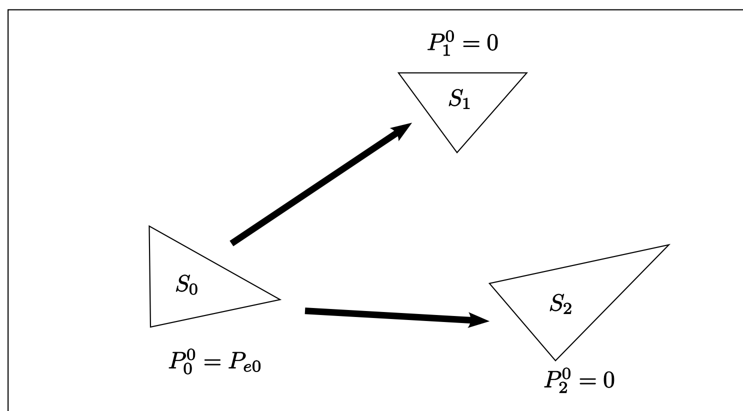
El tercer tipo, *estimación de densidad de fotones* [59, 70], también es una aplicación de los métodos de Monte Carlo y es muy similar al de caminos aleatorios, pero resuelven directamente la ecuación integral de radiosidad o de *rendering* en lugar de resolver un sistema lineal. Los caminos aleatorios de estos métodos no son otra cosa que trayectorias simuladas de fotones. La densidad de puntos de choque de estas trayectorias en una superficie es proporcional a la radiosidad.

1.6.1. Relajación estocástica del método de radiosidad

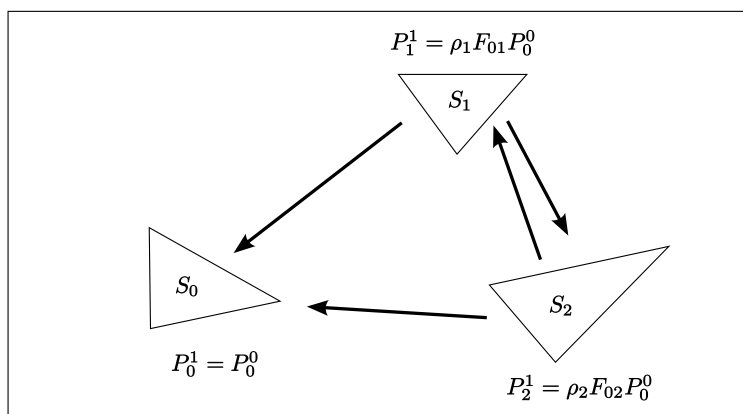
Los métodos de relajación estocástica surgen cuando se aplica el método de Monte Carlo para estimar las sumas que aparecen en cada iteración de métodos

iterativos tales como Jacobi, Gauss-Seidel o Southwell [20, 121]. Este trabajo se va a centrar en el método de relajación estocástica para radiosidad con Jacobi o, formalmente, radiosidad estocástica de Jacobi. En [117, 118, 119] se han propuesto algoritmos que se pueden interpretar como algoritmos estocásticos incrementales de Gauss-Seidel o Southwell, y en [9] se han estudiado adaptaciones estocásticas de sobrerrelajación, el método iterativo de Chebyshev y el método del gradiente conjugado. Estos métodos se han desarrollado para intentar reducir el número de iteraciones necesario para converger. Como las iteraciones determinísticas tienen un coste computacional fijo, estrechamente relacionado con el tamaño del sistema lineal, reducir el número de iteraciones claramente reduce el coste computacional total. Sin embargo, esto no es así en las variantes estocásticas. En estos métodos el coste computacional está dominado por el número de muestras que se tienen en cuenta. Así, en radiosidad, el método de radiosidad estocástica de Jacobi destaca por su simplicidad computacional y por su eficiencia igual o superior a los otros métodos de relajación estocástica.

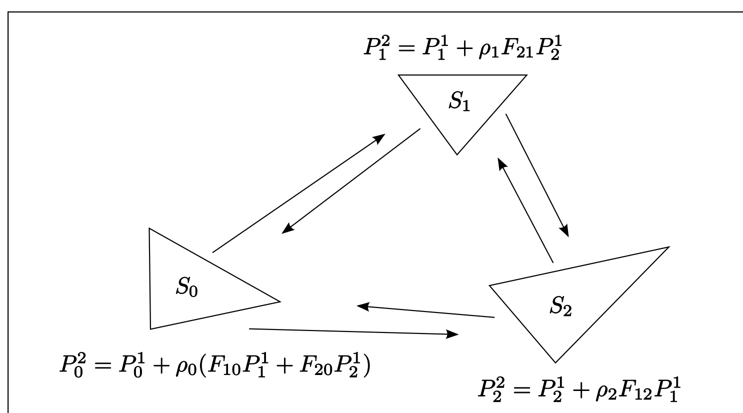
El método de radiosidad estocástica de Jacobi trabaja con valores de potencia. Si se reescribiera la Ecuación 1.18 en forma matricial, su matriz de coeficientes sería la misma $\mathbf{K}$ de la Ecuación 1.17. La norma de esta matriz $\mathbf{K}$ es menor que 1, requisito necesario para que el método converja. En el contexto de radiosidad, los vectores de incógnitas y de constantes del sistema de ecuaciones se corresponden con una distribución de potencia de luz sobre las superficies de la escena. La matriz del sistema de radiosidad modela un único nivel de interreflexión en la escena. Por ejemplo, cuando se multiplica por el vector de radiosidad propia emitida se obtiene la iluminación directa. Cuando se multiplica por el vector de iluminación directa, se obtiene iluminación indirecta con un nivel de reflexión. Cada iteración de Jacobi consiste en calcular un nuevo rebote de la luz interreflejada, y añadir ese valor a la radiosidad propia emitida. El equilibrio en la distribución de la iluminación en la escena es el punto fijo de este proceso. En la Figura 1.9 se muestra un ejemplo de como se aplica el método iterativo de Jacobi al cálculo de la radiosidad: en el estado inicial (Figura 1.9a) solo el elemento S_0 tiene potencia autoemitida; en la primera iteración los elemento S_1 y S_2 reflejan la potencia recibida de S_0 (Figura 1.9b); en la segunda iteración el elemento S_0 refleja la nueva potencia recibida de S_1 y S_2, y, a su vez, S_1 refleja la recibida de S_2 y S_2 la de S_1 (Figura 1.9c). Así en cada iteración cada elemento refleja la potencia recibida que los otros elementos emiten en la iteración anterior. Con el grosor de las flechas se indica que en cada paso la potencia luminosa reflejada es menor, debido a la multiplicación por los coeficientes de reflectancia y por los factores de forma.



(a) Estado inicial



(b) Primera iteración



(c) Segunda iteración

Figura 1.9: Método iterativo de Jacobi aplicado al cálculo de la radiosidad.

Las sumas de la formulación iterativa de Jacobi, $\mathbf{x}^{k+1} = \mathbf{b} + \mathbf{A}\mathbf{x}^{k}$, pueden ser estimadas usando métodos de Monte Carlo. Esto se puede realizar estocásticamente escogiendo aleatoriamente términos de estas sumas de acuerdo con alguna probabilidad. El valor promedio de los términos escogidos, teniendo en cuenta la probabilidad con la que han sido seleccionados, proporciona una estimación de las sumas.

Existen distintas aproximaciones para escoger los términos del sumatorio. Estas aproximaciones están basadas en líneas globales o locales. El procedimiento con líneas globales trabaja con líneas aleatoriamente trazadas a lo largo de la escena y la intersección de cada línea con las superficies de la escena define los tramos entre puntos mutuamente visibles a lo largo de la línea. En las aproximaciones basadas en líneas locales, las líneas se trazan seleccionando aleatoriamente el elemento origen S_i o el elemento destino S_j. En caso de que se escoja el elemento origen, se escoge un conjunto de elementos de origen S_i y se trazan líneas desde ellos. En el caso de escoger elemento destino, se traza un conjunto de rayos aleatorios siguiendo una distribución coseno desde cada polígono S_i y, consecuentemente, los polígonos destino S_j también son aleatoriamente elegidos. Esto permite una discretización más precisa de la potencia total en la iteración de disparo que la selección del polígono origen, pues en este caso solo se escoge una fracción de todos los polígonos en cada iteración [82].

La principal ventaja de las líneas globales sobre las locales es que se puede explotar la coherencia geométrica de la escena para generar eficientemente las líneas y, con el mismo coste computacional, crear más tramos que con las líneas locales. La principal limitación de las líneas globales es que su construcción no pueden adaptarse fácilmente para incrementar o reducir la densidad de líneas para un polígono dado [30]. En particular, cuando se usan para el cálculo del factor de forma, ya que se puede demostrar que la varianza del factor de forma es mayor cuanto menor sea el área del polígono.

Para el estudio del cálculo de la radiosidad mediante soluciones estocásticas, en este trabajo se ha escogido este método. En los Capítulos 4 y 5 se presenta una descripción más detallada, junto con las propuestas y los resultados obtenidos.

1.6.2. Métodos de caminos discretos aleatorios para radiosidad

Los métodos que se comentan a continuación también evitan el cálculo y almacenamiento de los factores de forma, permitiendo obtener la radiosidad en escenas grandes con menor coste computacional y menor consumo de memoria que las soluciones determinísticas. Estos métodos están basados en el concepto de camino aleatorio en un espacio discreto de estados y, a diferencia de los métodos de relajación estocástica, han sido bastante tratados en la bibliografía de métodos de Monte Carlo relacionada con la resolución de sistemas lineales [50, 103] y ha sido propuesta su aplicación al cálculo de la radiosidad en [112].

En un proceso de caminos aleatorios discretos existirán unas partículas que pueden encontrarse en un número discreto de estados, Nst. Inicialmente las partículas tendrán una probabilidad de *nacimiento* π_i de estar en un estado i. Estas probabilidades estarán normalizadas: $\sum_{i=1}^{Nst} \pi_i = 1$. Luego, esas partículas pueden cambiar de un estado i a otro j con una probabilidad de *transición*, p_{ij}. La suma de estas probabilidades no necesita ser uno. Estas partículas terminarán su camino en un estado con una probabilidad de *terminación* o de *absorción* $\alpha_i = 1 - \sum_{j=1}^{Nst} p_{ij}$.

Suponiendo que hay Npt partículas, el número esperado de veces que una partícula puede estar en el estado i será:

$$N_i = Npt \cdot \pi_i + \sum_{j=1}^{Nst} N_j \cdot p_{ji} \tag{1.19}$$

Dividiendo la ecuación anterior por Npt, se obtiene:

$$\chi_i = \pi_i + \sum_{j=1}^{Nst} \chi_j \cdot p_{ji}, \tag{1.20}$$

donde χ_i, que se llama *densidad de colisión*, es la solución de un sistema de ecuaciones lineales. χ_i puede ser mayor que uno y por eso no corresponde a una probabilidad. Resumiendo, sistemas de ecuaciones lineales como el anterior pueden ser resueltos generando caminos aleatorios y contando el número de veces que cada estado es visitado por una partícula. Cada estado se corresponde con una incógnita del sistema.

El sistema de la Ecuación 1.20 es similar al sistema de potencia emitida (Ecuación 1.18), aunque los términos P_{ei} no suman uno. Sin embargo, la solución es sencilla: se dividen ambos lados de la ecuación por la potencia total autoemitida $P_{eT} = \sum_{i=1}^{Ne} P_{ei}$. Por tanto, la Ecuación 1.18 se puede transformar en:

$$\frac{P_i}{P_{eT}} = \frac{P_{ei}}{P_{eT}} + \sum_{j=1}^{Ne} \frac{P_j}{P_{eT}} \; F_{ji} \; \rho_i \tag{1.21}$$

Este sistema sugiere un proceso de caminos aleatorios discretos con probabilidades de nacimiento $\pi_i = P_{ei}/P_{eT}$ y con probabilidades de transición $p_{ij} = F_{ij}\rho_j$. En el proceso, en primer lugar se muestrea un candidato a transición trazando, por ejemplo, una línea local. A continuación, se realiza sobre la partícula un test de aceptación/rechazo con una probabilidad de *supervivencia* igual a la reflectancia ρ_j. Si la partícula no sobrevive al test se dice que es *absorbida*.

Simulando *Npt* partículas que recorren caminos aleatorios de esta forma y contando el número N_i de partículas que llegan a un elemento S_i, la potencia luminosa P_i puede ser estimada como

$$\frac{N_i}{Npt} \approx \frac{P_i}{P_{eT}} \tag{1.22}$$

Debido a que las partículas se originan en fuentes de luz a este método se le llama de caminos aleatorios disparados (*shooting random walk method*). Al estimador se le llama de *supervivencia*, porque las partículas solo son contadas si sobreviven al test de rechazo. Normalmente las partículas en las fuentes de luz no son contadas, porque estiman la distribución de luz autoemitida, que es conocida. Esto se conoce como *supresión de la estimación de los términos fuente*.

El estimador anterior no es totalmente óptimo, porque cada candidato necesita una operación de disparo de rayo y si la transición no es aceptada esta operación ha sido realizada en vano. Es más eficiente contar todas las partículas que llegan a un elemento, sobrevivan o no. Así se tiene el estimador de *colisión* en el que es necesario reducir el número de estimaciones para compensar que se contabilizan partículas de más:

$$\rho_i \, \frac{N_{Ti}}{Npt} \approx \frac{P_i}{P_{eT}}, \tag{1.23}$$

donde N_{Ti} es el número total de partículas que llegan al elemento S_i.

Hay otro tercer estimador que solo cuenta las partículas si han sido absorbidas.

Así las estimaciones del estimador de *absorción* son

$$\frac{\rho_i}{1-\rho_i}\frac{N_{ai}}{Npt} \approx \frac{P_i}{P_{eT}}, \tag{1.24}$$

siendo Na_i el número total de partículas absorbidas en S_i.

Alternativamente al método de disparo, también se puede estimar la radiosidad en un elemento S_i dado utilizando las partículas que se originan en S_i y que llegan a una fuente de luz. Matemáticamente estos estimadores de acumulación (*gathering random walk estimators*) se corresponden con un estimador de disparo de caminos aleatorios que resuelve un sistema de ecuaciones adjunto [30].

1.6.3. Métodos de estimación de la densidad de fotones

Dentro de los método de Monte Carlo a continuación se va a comentar un conjunto de métodos de caminos aleatorios que están muy relacionados con los de la Subsección 1.6.2, pero que resuelven la ecuación integral de radiosidad (Ecuación 1.14), o la ecuación general de *rendering* (Ecuación 1.9), en lugar del sistema de ecuaciones de la radiosidad. Realmente, de la misma forma que los caminos aleatorios discretos se usan para resolver sistemas lineales, los caminos aleatorios en un espacio continuo de estados pueden ser utilizados para resolver ecuaciones integrales como la de la radiosidad o la ecuación de *rendering*. Así los métodos de estimación de la densidad de fotones [59] que se describen en esta sección también pueden ser llamados *métodos de caminos aleatorios continuos para radiosidad*.

Los caminos aleatorios no son más que trayectorias de fotones emitidos por fuentes de luz que rebotan en la escena según las leyes físicas de la luz. Los puntos de las superficies donde chocan estos fotones son almacenados en una estructura de datos para su uso posterior. Una propiedad esencial de dichos puntos es que su densidad en un determinado lugar (el número de choques por unidad de área) es proporcional a la radiosidad en ese lugar [59]. Esta densidad puede ser estimada en cualquier lugar de las superficies por medio de métodos de estimación de densidades de matemáticas estadísticas [122].

La principal ventaja de estos métodos es que se pueden tener en cuenta emisiones de luz no difusa y dispersiones, aunque, como en los métodos anteriores, no permiten resolver de forma exacta la ecuación de *rendering* en cualquier punto, con lo cual también pueden aparecer errores como límites de sombra borrosos o faltas de luz.

No obstante, los métodos de estimación de la densidad de fotones abren un camino a representaciones de la iluminación alternativo a la radiosidad media en polígonos. Por esta razón, han ganado considerable atención e importancia en los últimos años.

La simulación de la trayectoria de los fotones según las leyes de la física, llamada simulación *analógica* de la trayectoria de los fotones, consiste básicamente en comenzar en un punto de una fuente de luz e ir trazando rayos. A continuación, en puntos de choque en superficies de la escena se realiza un test de supervivencia, siguiendo muestreando nuevas reflexiones del fotón mientras este no sea reabsorbido. La dirección de salida se calculará según la distribución direccional de la emisión de luz de la fuente y las direcciones de reflexión dependerán de la BRDF de cada superficie.

Sea Nch_x el número esperado de choques por unidad de área de una partícula resultante de la simulación anterior, cerca de un punto x de una superficie. Esta densidad de choques tiene dos contribuciones: la densidad de partículas que nacen en x, Npn_x, y la densidad de partículas que llegan a x desde otro punto y. Esta densidad de partículas que vienen de y depende de Nch_y y de la densidad de las transiciones hacia x, $T(x|y)$. Por tanto se tiene:

$$Nch_x = Npn_x + \int_S Nch_y T(x|y) dA_y \tag{1.25}$$

Para un ambiente difuso, $Npn_x = B_{ex}/P_{eT}$ y $T(x|y) = G(y,x)V(y,x)\rho(x)/\pi$. Así se obtiene:

$$Nch_x = \frac{B_{ex}}{P_{eT}} + \frac{1}{\pi}\int_S Nch_y G(y,x)V(y,x)\rho(x) dA_y \;\Rightarrow\; Nch_x = \frac{B_x}{P_{eT}} \tag{1.26}$$

Por tanto, el número esperado de choques de partículas por unidad de área cerca de un punto x de una superficie es proporcional a la radiosidad B_x. Aunque este resultado se ha obtenido para ambientes difusos también es válido para emisiones de luz no difusas y para dispersiones.

De esta forma, el problema de calcular la radiosidad se ha reducido al problema de estimar la densidad de choques de partículas, esto es, estimar el número de choques de partículas por unidad de área en un punto dado de una superficie. El problema de estimar este tipo de densidades, dado un conjunto de localidades a muestrear ha sido muy estudiado en la matemática estadística [122]. A continuación, entre los métodos que han sido aplicados en el contexto de la iluminación de escenas, se comentarán

los métodos del vecino más cercano o algoritmos *photon mapping*, porque han sido los más populares y usados en los últimos años.

El algoritmo del mapa de fotones [59] (*photon mapping*) usa una técnica llamada *estimación de los vecinos más próximos*. Este método en lugar de fijar el área sobre la que se calcula el número de impactos de los fotones, lo que se fija es el número de impactos. Así, si el número de choques de los fotones se encuentran en un área pequeña la densidad será muy grande y al contrario, si es necesaria un área grande para abarcar el número de choques entonces la densidad será pequeña. La principal ventaja de este método es que no necesita hacer un mallado de la escena, solo almacenar el conjunto de puntos donde se producen los impactos de las partículas. El algoritmo consta de dos pasos. En el primero se construye la estructura del mapa de fotones emitiendo rayos desde las fuentes de luz de la escena, llamados ahora *fotones*, propagándolos a través de la escena y almacenando la información en la memoria del computador. Esta técnica permite la utilización de diversos tipos de fuentes de luz, así como simular todos los comportamientos de los fotones cuando chocan con un material (reflexión difusa o arbitraria, refracción o absorción). En una segunda etapa, una vez obtenido el mapa de fotones, la iluminación es estimada usando la técnica de los vecinos más próximos mencionada anteriormente.

En el método del mapa de fotones la representación de la iluminación es independiente de la geometría de la escena, lo que permite usar de una manera directa otro tipo de representaciones que no estén basadas en polígonos. Sin embargo existen dos principales inconvenientes: el gran consumo de memoria necesario para obtener resultados realistas en la iluminación indirecta en un entorno cerrado [42], y el problema potencial de la resolución eficiente de las relaciones de vecindad entre puntos arbitrarios del espacio, necesaria para el aprovechamiento de la estructura del mapa de fotones [73].

1.7. Técnicas de aceleración

Como ya ha sido comentado, todos los métodos de radiosidad tienen un alto coste computacional y grandes requerimientos de memoria, por lo que se han realizado grandes esfuerzos en la aplicación de técnicas de computación de alto rendimiento con el objetivo de reducir el tiempo de ejecución. Estos esfuerzos han ido encaminados, principalmente, a la implementación paralela de estos algoritmos en sistemas multiprocesador.

La paralelización del método clásico de radiosidad puede ser abordada, en una primera aproximación, como la resolución común de un sistema de ecuaciones lineales. Para ello, hay que tener en cuenta tanto las características concretas de la geometría del problema y de la iluminación global, como las relaciones de visibilidad entre los diferentes objetos de la escena y la coherencia de la transmisión de luz a través del medio.

Las implementaciones paralelas de los métodos de radiosidad se pueden clasificar según el tipo de acceso a la escena permitido a cada procesador. La necesidad de resolver la relación de visibilidad entre los distintos polígonos para el cálculo de los factores de forma obliga o a mantener almacenada la geometría total de la escena en cada memoria local, impidiendo su aplicación a escenas muy grandes y complejas, o a distribuir la escena aumentando y complicando las comunicaciones entre los procesadores. De esta forma, podemos considerar un primer grupo de soluciones paralelas en las que todos los procesadores tienen acceso a toda la escena, lo que simplifica notablemente la paralelización. Esto sucede en los sistemas de memoria compartida, como por ejemplo en [95, 120, 125] donde se implementa el algoritmo de radiosidad jerárquica. En [101] se presenta una implementación paralela del algoritmo de radiosidad progresiva también en una máquina de memoria compartida (SGI Origin 2000). En este caso la escena se divide en varias subescenas para hacer uso de las memorias caché locales y se utilizan máscaras de visibilidad e interfaces virtuales para el intercambio de energía entre subespacios. En [75] se expone una eficiente implementación paralela del algoritmo Monte Carlo de radiosidad, también en un computador SGI Origin 2000, utilizando el método de líneas globales.

Dentro de este grupo también hay implementaciones en sistemas paralelos de memoria distribuida. Para ello es necesaria la replicación de la escena y su almacenamiento en la memoria local de todos los procesadores, como en [130] donde se implementa el algoritmo de *ray tracing* en un *clúster* de PCs utilizando una aproximación de cliente–servidor y realizando las comunicaciones mediante llamadas estándar UNIX de TCP/IP. En [128] se hace una implementación del algoritmo de radiosidad progresiva utilizando una estructura maestro–esclavo. En este caso todos los nodos tienen la información de todos los elementos, pero las tareas de cálculo de visibilidad se reparten entre los procesadores dedicados a este trabajo. A su vez estos procesadores están asociados a otros que se encargan del cálculo y almacenamiento de la radiosidad, dependiendo todos de un procesador maestro. En cuanto al método de radiosidad jerárquica, en [37] se presenta una implementación con una división espacial dinámica muy simple que intenta mantener la carga computacional balan-

ceada, y en [92] se hace un reparto cíclico de los elementos entre los procesadores según su área con un posterior ajuste fino de este reparto para obtener un balanceo dinámico de la carga computacional.

Finalmente, en [17] se estudia la implementación de la radiosidad progresiva y la radiosidad jerárquica tanto en una máquina de memoria compartida como en una máquina de memoria distribuida. En el primer caso se balancea la carga computacional basándose en el área de los elementos. En el segundo caso, se minimizan las comunicaciones mediante la replicación parcial de los datos en cada memoria local.

Salvo [128], todas las implementaciones paralelas comentadas hasta aquí, en las que todos los procesadores tienen acceso a toda la escena, son de grano grueso. En ellas el objetivo principal es reducir el tiempo de ejecución repartiendo la carga de trabajo entre los procesadores, sin consideraciones adicionales que compliquen la paralelización y realizando el mínimo número de comunicaciones entre los procesadores. En estas implementaciones se realiza un reparto de los polígonos de la escena inicial entre los procesadores usando alguna métrica determinada, de manera que cada procesador actúa únicamente sobre un subconjunto de polígonos. Sin embargo, la geometría completa de la escena reside en la memoria de todos los procesadores, evitando las comunicaciones derivadas de la determinación de la visibilidad, pero limitando el tamaño de las escenas.

Hay un segundo grupo de implementaciones paralelas en las que la escena es distribuida entre las diferentes memorias locales de los procesadores, repartiendo no solo los cálculos a realizar sobre los polígonos, sino también la geometría de la propia escena. Aunque en alguna de las soluciones de este tipo la ejecución se dirige de forma centralizada desde un nodo usando una aproximación de grano fino [31, 133] que conlleva una gran sobrecarga de comunicaciones, la mayoría de los trabajos optan por una solución de grano grueso, obteniendo además mejores resultados. Por ejemplo en [76] se presenta una implementación paralela del método de radiosidad jerárquica, pero los datos son almacenados en una unidad de disco común a todos los procesadores, lo que limita el rendimiento de esta solución. Otra implementación paralela de este algoritmo la podemos encontrar en [94], donde se hace una división uniforme de la escena con una adaptación posterior a la geometría y donde se introduce una técnica de multitarea para mejorar la interacción entre los procesadores. En [7, 48] se muestran implementaciones paralelas del algoritmo de radiosidad progresivo utilizando el concepto de *máscaras de visibilidad* para manejar la interacción entre los distintos procesadores. La idea de repartir la geometría entre los procesadores presenta, a priori, el inconveniente de un aumento del número de

comunicaciones, como consecuencia de la necesidad de determinar la visibilidad entre objetos de subespacios almacenados en procesadores distintos, lo que conlleva a la necesidad de acceder a información que no está almacenada localmente. Para poder reducir el número de comunicaciones, se ha optado habitualmente por la simplificación o aproximación de la visibilidad entre subescenas [39].

En los últimos años ha surgido otra alternativa en el procesamiento de altas prestaciones que es el uso de las GPUs (*Graphics Processing Unit*). La mayoría de las aplicaciones de gráficos por computador actuales requieren cientos de gigaflops de potencia, lo que es satisfecho por la GPU presente en cualquier PC de usuario. Debido a la gran demanda del mercado de gráficos 3D de alta definición en tiempo real, la GPU ha evolucionado a un procesador paralelo programable, multitarea y con cientos de núcleos, obteniendo una alta potencia de cálculo y un gran ancho de banda de memoria. De esta manera las GPUs ofrecen una plataforma para computación de alto rendimiento, no solo para procesamiento gráfico sino también para procesamiento no gráfico (GPGPU, *General-Purpose Computation on Graphics Processing Unit*). Los primeros lenguajes de programación general para GPUs que aparecieron (por ejemplo *Cg* [32] o *Microsoft HLSL* [45]) eran APIs (*Application Programming Interface*) orientados a gráficos que requerían código de bajo nivel, donde el programador debía gestionar directamente ciertos recursos hardware y sus limitaciones. Hoy en día la programación de la GPU es todavía compleja y requiere el uso de lenguajes especiales, como *CUDA* de *NVIDIA* [88] o *Brook+* de *ATI* [1], que frecuentemente exponen características o limitaciones del hardware. Esto restringe la flexibilidad de los programas de la GPU y fuerza al programador a conocer el hardware para explotar eficientemente su rendimiento. Recientemente ha habido esfuerzos en estandarizar la programación de las modernas GPUs que han llevado a la creación de *DirectCompute* [14] y *OpenCL* [66], que son lenguajes estándar de programación para computación heterogénea. Aún así, la implementación requiere un análisis detallado y optimizaciones de los algoritmos para obtener un rendimiento que explote las características de la GPU.

Los primeros trabajos para el cálculo de la radiosidad con GPUs se centraron en la aceleración de determinadas partes de los algoritmos. Así en [15] se utilizó el formato de texturas para la matriz de radiosidad para acelerar las operaciones matriciales. En [65] se usó un API gráfico estándar para el cálculo de la acumulación de energía de la escena a partir de la definición de fuentes de luz virtuales (VPLs) en el algoritmo de *radiosidad instantánea*. A partir de una solución más burda del método de radiosidad jerárquica, en [74] se mostró la obtención de una iluminación

de alta definición mediante la GPU. La aceleración del cálculo de los factores de forma en el algoritmo de radiosidad progresiva mediante la GPU se presentó en [86], mientras que en [24] ya aparece una implementación del algoritmo completo de radiosidad progresiva en GPU, aunque por simplicidad solo emplean cuadriláteros como polígonos.

La utilización de la GPU puede permitir obtener resultados de los algoritmos de iluminación global en tiempo real y, por tanto, el empleo de estos algoritmos en aplicaciones interactivas, como por ejemplo en videojuegos. Por ejemplo, en [71] se presenta un método de iluminación basado en una función de transferencia de radiancia precalculada (PRT) interactiva para escenas estáticas usando una representación jerárquica de puntos. Otras implementaciones interactivas en GPU del cálculo de la iluminación global usan otros métodos como *antirradiancia* [26] o *visibilidad implícita* [28] con el objetivo de evitar el cálculo de la visibilidad entre elementos de la escena, aunque el número de polígonos utilizados es reducido para conseguir interacción en tiempo real. Además, el manejo implícito de la visibilidad hace necesario la obtención de una estructura de enlaces jerárquicos que requiere un trabajo adicional [77], y que crece desproporcionadamente con el incremento de la complejidad de la escena. Otras propuestas hacen aproximaciones a los cálculos de la iluminación indirecta usando *visibilidad imperfecta* [102], desarrollando esquemas simplificados para una iluminación indirecta plausible [62], o aproximando la iluminación de la escena mediante técnicas basadas en la idea de la *radiosidad instantánea* y la definición de un conjunto de VPLs [69]. Recientemente se ha popularizado operar en el espacio de la imagen, utilizando mapas de sombras de reflexión (*reflective shadow map*, RSM) que capturan directamente las superficies iluminadas, pero además almacenan información adicional que es necesaria para calcular la iluminación indirecta de estas superficies, de forma que cada píxel puede ser visto como una pequeña fuente de luz. En [83] se utilizan los RSMs para crear VPLs, calculando la iluminación indirecta a una baja resolución para superficies suaves y más exactamente donde haya más detalle en la geometría. Esta idea ha sido mejorada en [84] mediante una técnica de agrupación inteligente de los píxeles de los RSMs.

En este trabajo se presenta la implementación paralela de los algoritmos de radiosidad progresiva y del método de Monte Carlo para radiosidad, en sistemas de memoria distribuida y con el reparto de la escena entre las memorias locales de los procesadores, usando una aproximación de grano grueso. Además, también se ha realizado una implementación de grano fino en GPUs del algoritmo de Monte Carlo para radiosidad, empleando las técnicas de división de la escena desarrolladas para

las implementaciones distribuidas comentadas anteriormente, junto con soluciones adicionales específicas para estos sistemas.

1.8. Estructura de la memoria

Esta memoria se ha dividido en cinco capítulos, constituyendo este primero una visión general de los modelos de iluminación global, así como una introducción a métodos determinísticos y estocásticos de la resolución de la ecuación de radiosidad y una visión del estado actual de las técnicas de aceleración de estos algoritmos.

Este trabajo, dentro de los métodos determinísticos, como ya se ha comentado se centra en el método de radiosidad progresiva, porque frente al método jerárquico tiene la ventaja de que al elegir el elemento con mayor potencia radiante sin disparar se reduce en mayor medida la potencia total que queda sin disparar en la escena. Así al final de cada iteración se obtiene una aproximación de la radiosidad de toda la escena, que se va refinando progresivamente, sin la necesidad de recorrer todos los elementos.

Dos de los principales problemas para los métodos determinísticos es el cálculo y almacenamiento de los factores de forma. En el método progresivo se evita el almacenamiento de todos los factores de forma, porque se hace su cálculo bajo demanda, pero este cálculo se puede hacer inmanejable cuando aumenta la complejidad de la escena. En el Capítulo 2 se presenta nuestra propuesta de implementación paralela en sistemas de memoria distribuida utilizando el paradigma de paso de mensajes. Para ello la escena es dividida en subescenas más manejables, todas de igual tamaño y forma, mediante un método de partición convexo y uniforme, reduciendo así la complejidad y permitiendo una computación totalmente distribuida [5, 6, 47]. Para evitar la sobrecarga de comunicación entre procesadores se emplean comunicaciones no bloqueantes que permiten el solapamiento de comunicaciones y cálculos. Todo esto ha hecho posible la implementación paralela del algoritmo de radiosidad progresiva, pero también se demuestra que algunas de las técnicas utilizadas pueden reducir el coste computacional en entornos monoprocesador [105].

La división de la escena aplicada en las implementaciones del Capítulo 2 es una partición geométrica uniforme. Esta manera de dividir la escena no tiene en cuenta sus características, lo que hace que sea poco eficiente en cuanto al reparto de la carga de trabajo. En el Capítulo 3 se presenta otro método de división dirigido por la minimización de una función de distribución que describe el desbalanceo de la

carga computacional en términos de un factor específico. Este método lleva a obtener subescenas de distinto tamaño, pero con carga computacional más equilibrada [104].

En cuanto a las soluciones estocásticas para el cálculo de la radiosidad se han escogido los métodos de relajación estocástica que aplican Monte Carlo. Los métodos de caminos aleatorios discretos para radiosidad o los métodos de estimación de densidad de fotones son aproximadamente igual de eficientes que las técnicas de Monte Carlo de relajación estocástica. Sin embargo, cuando se intenta mejorar el rendimiento de estos métodos existen técnicas como la reducción de la varianza mediante variables de control que son más fáciles de implementar y más efectivas en métodos de relajación estocástica.

Los métodos de Monte Carlo no tienen el problema del cálculo y almacenamiento de los factores de forma, pero su implementación en escenas complejas también tiene un alto coste computacional por el gran número de rayos que hay que disparar para conseguir una buena estimación de la radiosidad. En el Capítulo 4 se presenta nuestra implementación paralela sobre un sistema de memoria distribuida del algoritmo Monte Carlo de radiosidad, utilizando el paradigma de paso de mensajes y basándose en la subdivisión de la escena, tanto uniforme como no uniforme, y en la distribución de los datos entre los procesadores, lo que permite la utilización de este algoritmo con escenas complejas [106, 108]. Para resolver los problemas derivados de estas implementaciones e incrementar su rendimiento se presentan diferentes propuestas, como por ejemplo distintas estrategias para dirigir la división no uniforme, el empaquetamiento de rayos para la minimización de las comunicaciones entre los procesadores o el cálculo distribuido de la determinación del final de las iteraciones.

El reciente interés en GPGPU nos ha llevado a estudiar la implementación del método de Monte Carlo para radiosidad en la GPU porque consideramos que se adapta bien a una implementación paralela de grano fino adecuada para este tipo de arquitectura. Partiendo de trabajos previos [107, 110], en el Capítulo 5 es presentada nuestra propuesta de implementación, con distintas estrategias para mejorar su rendimiento: la utilización de una malla simplificada de elementos de las superficies para reducir los requerimientos computacionales, la división de la escena para aumentar la localidad y una planificación eficiente de la ejecución de tareas para explotar las características de la GPU [109, 111].

Capítulo 2

Radiosidad progresiva paralela: partición uniforme

En este trabajo se ha escogido entre los métodos determinísticos el algoritmo progresivo de radiosidad. Este método no requiere el cálculo y almacenamiento de todos los factores de forma, ya que el cómputo de estos valores se realiza cada vez que son necesitados. A pesar de esto, el algoritmo sigue teniendo un alto coste computacional, por lo cual se ha explorado su implementación en sistemas paralelos.

Debido a los requerimientos de memoria, la implementación paralela del algoritmo de radiosidad progresiva puede limitar el tamaño de la escena en sistemas de memoria compartida, o en sistemas de memoria distribuida en el caso de replicar toda la información en todos los procesadores. Para evitar esto, la solución es utilizar un sistema de memoria distribuida y repartir los datos entre las memorias de los procesadores. Sin embargo, esto supone grandes retos en cuanto a la coordinación de la ejecución del algoritmo y también respecto a las comunicaciones entre los procesadores, si se quiere aprovechar la capacidad de cálculo simultáneo.

En este capítulo se presenta nuestra aproximación del algoritmo progresivo de radiosidad en sistemas de memoria distribuida mediante una paralelización de grano grueso. Para no tener limitado el tamaño de la escena, hemos desarrollado un método de división de la escena convexo y uniforme y hemos repartido los datos entre los procesadores. Además hemos utilizado el paradigma de paso de mensajes, minimizando la sobrecargar de comunicación mediante el uso de comunicaciones no bloqueantes para solapar cálculos y comunicaciones. Otro problema que tuvimos que resolver era el cálculo de la visibilidad entre subespacios. Hemos optado por uti-

lizar una modificación de la técnica de las *máscaras de visibilidad* [7]. Finalmente, la división del trabajo entre los procesadores también implica nuevas comunicaciones para determinar la finalización de cada iteración. Para ello, presentamos un método de envío de mensajes que evita el bloqueo de los procesadores que non han finalizado su tarea, mientras los otros están a la espera de recibir nuevos datos o de que finalicen los demás. Este trabajo ha sido descrito en [5, 6, 47].

Por otro lado hemos comprobado que la utilización del método de división uniforme de la escena también aporta beneficios en cuanto al aprovechamiento de la localidad de los datos. Así presentamos una transformación del algoritmo secuencial con la aplicación a bloques de datos, basada en la división uniforme de la escena, que optimiza el algoritmo y que logra reducir su tiempo de ejecución. Este trabajo ha sido publicado en [105].

En cuanto a la estructura de este capítulo, se comienza hablando sobre el método de radiosidad progresiva. A continuación, en la Sección 2.2, se desarrolla nuestro método de distribución de la escena entre los procesadores, dividiéndola en subescenas, lo que permite trabajar con escenas grandes. Para realizar esta distribución se utiliza un nuevo algoritmo de recorte de los polígonos de la escena que mejora los resultados de los algoritmos tradicionales. En la Sección 2.3 se explica como se realiza nuestra implementación paralela, en la cual a cada procesador se le asigna una subescena y es responsable de realizar todas las computaciones de la radiosidad del conjunto de datos de esa subescena. En la Sección 2.4 se presenta un análisis de la reutilización de datos en ejecuciones secuenciales del algoritmo progresivo basado en las ideas de división de la escena, demostrando que estas técnicas pueden ser empleadas para la optimización del algoritmo. El capítulo finaliza mostrando los resultados experimentales obtenidos.

2.1. El método de radiosidad progresiva

Como se comentó en el anterior capítulo, el método de radiosidad progresiva resuelve el sistema de ecuaciones de radiosidad combinando dos técnicas de resolución iterativa de sistemas lineales que son el método de relajación de Southwell y el algoritmo iterativo de Jacobi [44, 81].

La idea básica de la radiosidad progresiva es *disparar* en cada iteración la energía del elemento S_i con mayor energía no disparada, es decir, que tenga el producto $\Delta B_i \cdot A_i$ mayor, para conseguir que el algoritmo converja lo más rápido posible. In-

tuitivamente, los elementos con más energía luminosa tendrán un efecto mayor sobre la iluminación de la escena y, por lo tanto, deberían de ser considerados primero. De acuerdo a esta regla, la mayoría de las fuentes de luz son disparadas primero, ya que los otros elementos tienen la radiosidad a cero, y a continuación serán procesados aquellos polígonos que reciben más luz de estas fuentes, y así sucesivamente. Siguiendo este orden, desde las primeras iteraciones ya se obtienen soluciones bastante precisas, reduciendo sustancialmente los costes de computación.

El resultado final del *disparo* del elemento S_i es que los demás elementos de la escena, S_j, pueden recibir nueva radiosidad y que S_i queda sin radiosidad sin disparar ($\Delta B_i = 0$). Cada iteración puede interpretarse como una multiplicación del valor de radiosidad que tiene que ser *disparada* por la columna de la matriz de los factores de forma (Ecuación 1.17). Por consiguiente, los factores de forma entre el elemento que se dispara y el resto de elementos tienen que ser calculados. F_{ij} se puede expresar de la siguiente forma:

$$F_{ij} = G_{ij} \cdot V_{ij} \tag{2.1}$$

donde G_{ij} engloba a todos los términos geométricos y V_{ij} representa a la visibilidad que tiene valor 1 si S_i es visible desde S_j y 0 en otro caso.

Una vez que la radiosidad de un elemento es disparada comienza una nueva iteración escogiendo otro elemento. Durante la evolución del algoritmo la operación puede repetirse para el elemento S_i varias veces, pero en cada paso solo la cantidad de radiosidad que ese elemento ha recibido desde la última vez, ΔB_i, debe ser considerada. El algoritmo continúa hasta que se alcance una tolerancia deseada en la cantidad de energía a disparar.

El principal problema de la radiosidad progresiva es hacer el modelado de la escena, es decir, obtener una malla de polígonos. Los polígonos iniciales de la escena pueden ser de un tamaño bastante grande, y su utilización para el cálculo de la iluminación daría un resultado muy poco realista. Por lo tanto, estos elementos iniciales deben ser divididos. Cuanto más fina sea la malla mejor será la iluminación resultante, ya que tendremos mayor número de elementos y nuestra escena tendrá un mayor nivel de detalle. Sin embargo, un aumento de la densidad también supone una mayor carga computacional y el beneficio proporcionado depende de la superficie y no siempre aporta mejoras significativas a la iluminación. En nuestra implementación se utilizan como polígonos triángulos y se escoge inicialmente una densidad de malla fijando un valor máximo de área.

```
1   /* Inicializamos valores */
2   for (S_i = Escena→Elemento_inicial; S_i; S_i = S_i →siguiente) {
3         B_i = E_i;
4         ΔB_i = E_i;
5   }
6
7   /* Proceso iterativo */
8   while (no convergió) {
9           S_i = 0;
10          /* BUCLE1: seleccionamos elemento con mayor ΔB_i · A_i */
11          for (S_j = Escena→Elemento_inicial; S_j; S_j = S_j →siguiente)
12                S_i = MaxPot(S_i, S_j);
13
14          /* BUCLE2: Calculamos, solo primera vez que un elemento es
                      disparado, elementos que interacciona con S_i */
15          for (S_j = Escena→Elemento_inicial; S_j; S_j = S_j →siguiente)
16                Interacción(S_j, S_i);
17
18          /* BUCLE3: Disparamos radiosidad desde el elemento S_i */
19          for (S_j = S_i →Interacción→Elemento_inicial; S_j;
                    S_j = S_i →Interacción→ S_j →siguiente) {
20                V_ji = Visibilidad(S_i, S_j);
21                G_ji = Geometría(S_i, S_j);
22                F_ji = V_ji · G_ji;
23                ΔB_j = ΔB_j + ΔB_i · ρ_j · F_ji;
24                B_j = B_j + ΔB_i · ρ_j · F_ji;
25          }
26          ΔB_i = 0;
27  }
```

Figura 2.1: Estructura del algoritmo de radiosidad progresiva.

En la Figura 2.1 se muestra la estructura del algoritmo. En primer lugar, todas las radiosidades, B_i y ΔB_i, son inicializadas a cero para todos los polígonos que no sean fuentes de luz, y a los valores de radiosidad auto–emitida para los elementos emisores (líneas desde la 2 a la 5).

En el proceso iterativo, una vez escogido el elemento S_i con el mayor valor de $\Delta B_i \cdot A_i$ (líneas 11 y 12 de la Figura 2.1) se dispara su radiosidad a través de la escena (líneas desde la 15 a la 25). Como resultado de este disparo, los otros elementos S_j pueden recibir cierta radiosidad nueva (líneas 23 y 24) y S_i deja de tener radiosidad sin disparar (línea 26). Cada iteración se puede interpretar como la multiplicación del valor de radiosidad que va a ser disparado por una columna de la matriz de factores de forma del sistema de ecuaciones (ver Ecuación 1.17). Por tanto en este momento es necesario calcular los factores de forma requeridos y el valor de visibilidad entre el polígono escogido y el resto de polígonos (líneas desde la 20 a la 22). Respecto al cálculo del factor de forma, hemos usado el método analítico llamado *área diferencial a polígono convexo* [20], también conocido como *punto a polígono*. Es un método utilizado habitualmente cuando se manejan modelos poligonales.

Por otra parte, para la determinación de la visibilidad hemos empleado un algoritmo basado en técnicas direccionales, que intentan explotar la coherencia en la localización y dirección de los rayos lanzados. Un precursor de esta clase de métodos es el algoritmo de clasificación de rayos de Arvo y Kirk [8], que subdivide mediante volúmenes 5D el espacio de todos los rayos posibles entre dos polígonos, formando conjuntos de rayos a los que asocia listas de candidatos que pueden obstaculizarlos. Esta técnica ha sido escasamente implementada hasta ahora [123, 25] debido a su complejidad, a su alto coste tanto en tiempo de ejecución como de memoria requerida, y a que es menos robusta y predecible que otras que trabajan en el dominio de la escena. Algunos métodos más recientes [115] solucionan algunos problemas del método original, pero todavía sin llegar a alcanzar los resultados en cuanto a rendimiento de otras técnicas. No obstante, se pueden conseguir grandes mejoras en el rendimiento aplicando los conceptos de corredor (*shaft*) y listas de candidatos de Haines y Wallace [49] tanto en el dominio 5D usado por Arvo y Kirk como en el espacio 3D de la escena [97, 93], alternativa que se ha escogido para nuestra implementación.

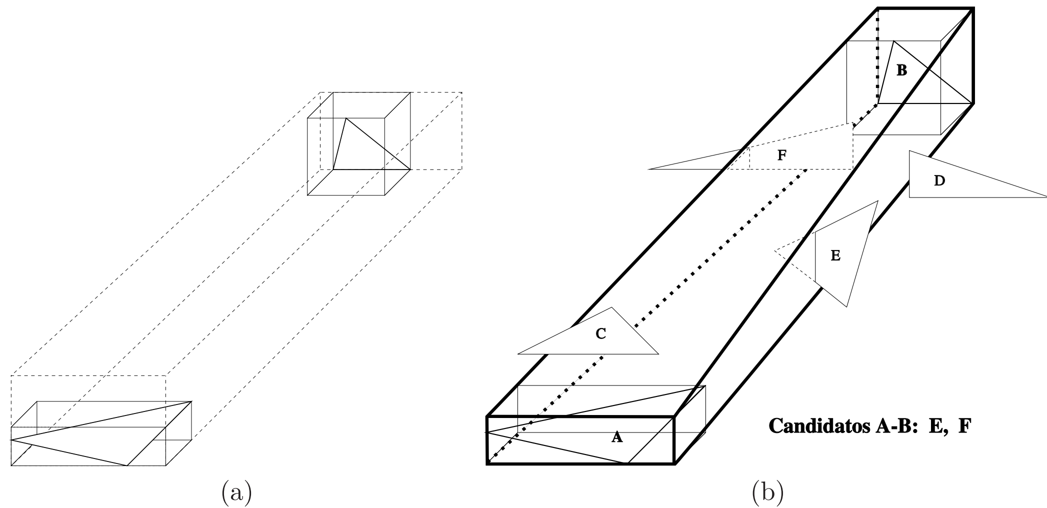
En nuestra propuesta se aprovecha la localidad de los rayos delimitando el volumen que contiene todos los posibles rayos que se pueden lanzar entre dos polígonos, construyendo a continuación una lista con los polígonos que atraviesan o se encuen-

tran por completo dentro de ese volumen, proceso conocido como *shaft–culling*. Esta lista contiene a todos los posibles candidatos a obstaculizar a alguno de los rayos lanzados entre ambos polígonos. En [49] se propuso la construcción de un corredor entre las cajas de contorno de los dos polígonos como un refinamiento del volumen de contorno alineado con los ejes que contiene a ambos polígonos, pero ajustado a las cajas de contorno de cada uno de ellos. Esta estructura es rápida y fácil de calcular y acelera la detección de intersecciones corredor–polígono respecto a una caja de contorno orientada.

En la Figura 2.2 se muestra un ejemplo de la creación de un corredor y la posterior construcción de la lista de candidatos para dos polígonos. Primero se forma la caja de contorno grueso y las cajas individuales de los dos polígonos (Figura 2.2a), usando planos alineados con los ejes. La caja gruesa se refina usando planos que unen las aristas apropiadas en las cajas de contorno individuales, obteniendo el corredor (Figura 2.2b). Para la búsqueda de candidatos primero se comparan los demás polígonos con la caja gruesa, para un descarte rápido de los polígonos más alejados, y después directamente con el corredor para un ajuste más fino con los polígonos más cercanos. En nuestro ejemplo (Figura 2.2b), la caja gruesa permite que el polígono C sea inmediatamente descartado como candidato a obstaculizar la interacción A-B. Luego mediante el corredor descartamos también el polígono D, mientras que E y F se confirman como candidatos a obstaculizar los rayos lanzados entre A y B.

2.2. Método de partición uniforme de una escena

Obtener una buena partición de los datos es un aspecto fundamental que debe ser estudiado con atención para obtener tiempos de ejecución eficientes en programas paralelos de alto rendimiento. Esta tarea es incluso más importante en sistemas de memoria distribuida. En estas plataformas la distribución de los datos entre los procesadores es muy importante para evitar la replicación de los mismos, por las limitaciones del tamaño de las memorias locales, para minimizar las comunicaciones entre los nodos y para alcanzar un buen balanceo de la carga computacional. En esta sección se presenta nuestro método distribuido de partición uniforme de la escena, para el cual hemos diseñado un algoritmo totalmente paralelo en el que cada procesador calcula su subespacio geométrico junto con la lista de polígonos sobre los que va a calcular la radiosidad, sin necesidad de establecer comunicaciones con el resto de procesadores. Además, como puede haber polígonos iniciales que



(a) (b)

Figura 2.2: Construcción de una lista de candidatos a obstáculos mediante un corredor (*shaft culling*): (a) Caja de contorno gruesa entre los dos polígonos (b) Corredor obtenido y proceso de descarte de polígonos (*shaft culling*).

pertenezcan a varios subespacios, se muestra un nuevo método de recorte de estos polígonos que mejora las técnicas más habitualmente utilizadas.

Normalmente, la distribución de los datos entre los procesadores se calcula resolviendo un problema de división de grafos [114], en el cual lo que se busca es distribuir los vértices del grafo en Nd subdominios disjuntos, asumiendo Nd procesadores, de tal modo que se optimice el tiempo de ejecución. La partición de un grafo es un problema NP-completo, pero se han desarrollado muchos algoritmos para buscar una solución razonablemente buena. Generalmente, en la división del grafo se intenta tener en cuenta los siguientes aspectos:

- La relación computación–comunicación dentro de un procesador debe ser maximizada.
- La carga computacional debe estar balanceada entre todos los procesadores.
- El coste temporal asociado al control de las comunicaciones y a la complejidad en los almacenamientos debe ser minimizado.

Siguiendo estos criterios, si se asigna un peso a los vértices y a las aristas, generalmente las técnicas de partición lo que buscan es repartir los vértices entre los subespacios para que cada uno tenga la misma cantidad de peso de los vértices y

para que se minimice la métrica que estima el número o volumen de comunicaciones entre subdominios. Una métrica empleada en algunos algoritmos es el número de aristas cortadas, que serán aquellas cuyos vértices pertenecen a distintos subespacios. Suponiendo que cada procesador necesitará ambos vértices de la arista cortada para realizar los cálculos necesarios por la aplicación, esta métrica da una medida aproximada del número de comunicaciones que serán necesarias.

Ejemplos de la utilización de estas técnicas se pueden encontrar en diferentes bibliotecas de funciones software, entre las que destacan Chaco [54] y METIS [63]. Chaco ha sido desarrollada en el contexto de la computación paralela, pero puede ser utilizada en otros campos. Resuelve la división de grafos ajustada a diversas topologías, constando de varios algoritmos basados en las estrategias inercial, espectral, Kerninghan-Lin y métodos multinivel combinados con otros más simples, permitiendo la utilización conjunta de varias. Cada una de estas estrategias puede ser usada para dividir el grafo en dos, cuatro u ocho partes en cada nivel de recurrencia. En METIS los algoritmos están basados en la división multinivel de grafos, usando técnicas para reducir el tamaño del grafo así como para refinar la partición en la fase en la que se usa el grafo sin simplificaciones para hacer la división del grafo original. Los algoritmos de METIS producen particiones de similar o mejor calidad que los algoritmos multinivel de bisección más utilizados y normalmente son dos órdenes de magnitud más rápidos [64].

En nuestra implementación nos hemos centrado en algoritmos paralelos de división estática de grafos, ya que la estructura de los datos no cambia entre iteraciones. Dentro de las técnicas estáticas hemos escogido las *técnicas geométricas* [11, 40] que calculan las particiones basándose solamente en la información de las coordenadas de los nodos de la malla, sin tener en cuenta la conectividad entre los elementos. Normalmente estas técnicas dividen los elementos de la malla directamente y no los grafos que modelizan la malla, por lo que también se denominan técnicas de *partición de malla*. Habitualmente son técnicas extremadamente rápidas que proporcionan una buena localidad de datos.

A este tipo de métodos pertenece el algoritmo *disección anidada de coordenadas* [11], CND (*Coordinate Nested Dissection*), también conocido como *bisección recursiva de coordenadas*, RCB (*Recursive Coordinate Bisection*). En esta técnica primero se calcula el octaedro (paralelepípedo rectangular) que encierra a la malla y, a continuación, se divide la malla trazando un plano perpendicular en la mitad de las aristas con mayor longitud del octaedro. De esta manera lo que se busca es minimizar el tamaño de las fronteras entre dominios. Cada subdominio puede ser

recursivamente dividido por la misma técnica. Este esquema es extremadamente rápido, requiere poca memoria y es fácil de implementar.

En este capítulo se usa una versión del algoritmo CND que hace una partición geométrica uniforme y paralela de la escena, es decir, divide la escena en un conjunto de subescenas disjuntas del mismo tamaño y forma, asignando los elementos contenidos en cada una de ellas a un procesador. La partición es de tipo geométrico porque solo emplea la información de las coordenadas para realizar la distribución de la escena, y es paralela porque cada procesador realiza la parte de la partición correspondiente a su subescena, en lugar de ser un único procesador el encargado de dividir y distribuir los datos entre todos los demás. Aquellos elementos que pertenecen parcialmente a varias subescenas son divididos mediante un algoritmo de recorte en tantas partes como subescenas a las que pertenecen.

Esta clase de partición de la escena se calcula rápidamente y de una forma directa y el hecho de ser uniforme es muy conveniente para el uso de las *máscaras de visibilidad* [7] en la implementación paralela del algoritmo, que será explicada con detalle en la Sección 2.3. Una partición uniforme minimiza el número de "vecinos" que puede tener una subescena, simplificando de esta forma los cálculos que se realizan en la técnica de las máscaras de visibilidad y reduciendo el número de comunicaciones necesarias. También proporciona una alta localidad de los datos. Sin embargo, tiende a calcular particiones desbalanceadas, lo que podría suponer una penalización en el tiempo total de ejecución del algoritmo paralelo.

En nuestra propuesta se han tenido en cuenta las siguientes consideraciones:

- Hay tantos subespacios como procesadores y, debido a la división uniforme, deben ser iguales en tamaño. Esto significa que para un número impar de procesadores y, por consiguiente de subespacios, no podemos dividir la escena en dos mitades y luego, recursivamente, dividir una mitad en otras dos (ver Figura 2.3a), porque los subespacios resultantes no serían de igual tamaño. Por lo tanto, el algoritmo implementado no es recursivo y calcula *a priori* el número de planos divisores que es necesario trazar perpendiculares a cada eje de coordenadas.

- Al ser una partición uniforme, los planos que dividen la malla que sean perpendiculares al mismo eje de coordenadas han de ser paralelos y equidistantes. Además, cada plano divisor debe atravesar toda la escena y no solo una de las subescenas existentes, como hace el CND (ver Figura 2.3a).

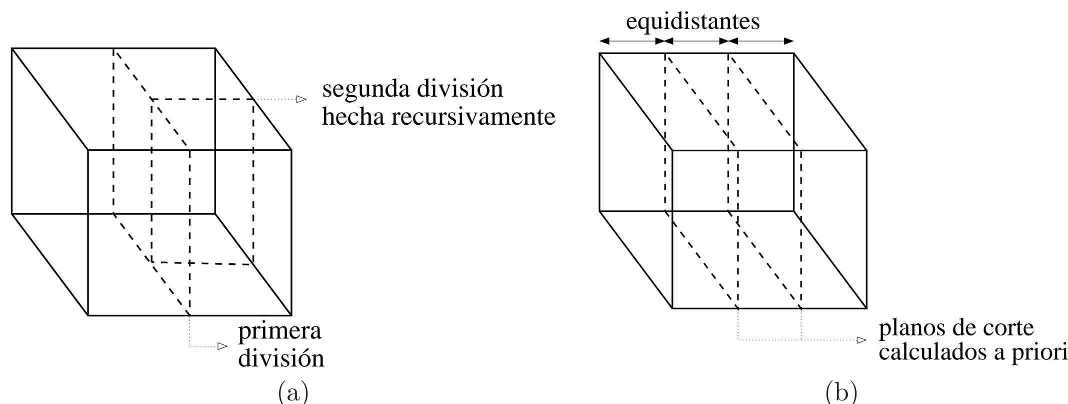


Figura 2.3: Partición recursiva *vs.* cálculo *a priori* de las divisiones: (a) Partición en tres subespacios de forma recursiva. (b) Partición en tres subespacios calculando las divisiones *a priori*.

El algoritmo de partición tiene dos fases. En la primera, que llamamos *división en subespacios* y que se comentará más detalladamente en la Subsección 2.2.1, la escena inicial sin modelar, es decir, sin haber sido refinado el mallado de los polígonos, se replica en todos los procesadores. Este hecho no supone ningún inconveniente por los bajos requerimientos de memoria que supone la escena sin dividir. A continuación cada procesador divide el espacio de la escena, determinando sus subespacios y calculando sus límites geométricos.

La segunda fase, que podemos denominar *distribución de datos entre los subespacios* y que se comentará en la Subsección 2.2.2, los objetos de la escena son repartidos entre los procesadores de acuerdo a la asignación de subespacios realizada en la fase anterior. Una vez que esto se ha hecho cada procesador tiene un conjunto disjunto de polígonos y solo realizará el mallado de éstos. Para poder llevar a cabo este reparto de los polígonos, algunos de ellos tienen que ser divididos por pertenecer parcialmente a varios subespacios.

2.2.1. División en subespacios

Cada procesador calcula el espacio de la escena, que es el paralelepípedo de menor tamaño que contiene todos los objetos de la escena. A continuación, cada procesador determina el número de divisiones Nd^x, Nd^y y Nd^z que se van a hacer en cada una de las dimensiones del espacio de la escena. Este espacio será dividido en cada

dimensión por tantos planos como número de divisiones menos una correspondan a esa dimensión. Dichos planos y las caras del paralelepípedo del espacio con las que son paralelos deben ser equidistantes, como se muestra en la Figura 2.3b, para obtener una partición uniforme.

El número de divisiones por dimensión se calcula en función de la lista de factores primos que resultan de descomponer el número total de procesadores Nd en un producto de sus divisores primos d_i:

$$Nd = \prod_{i=1}^{m} d_i \tag{2.2}$$

Para evitar subespacios alargados, que podría suceder si el número de divisiones en una dimensión es mucho mayor que en otras, los factores primos son asignados a cada dimensión de la siguiente forma: inicialmente $Nd^x = Nd^y = Nd^z = 1$; luego, los factores se ordenan en orden descendente. Esta lista de factores se va recorriendo y cada factor d_i se asigna al menor valor de $\{Nd^x, Nd^y, Nd^z\}$, de forma que:

$$(Nd^j)^{\text{nuevo}} = (Nd^j)^{\text{viejo}} \cdot d_i \tag{2.3}$$

Como ejemplo, si trabajamos con 36 procesadores, la lista ordenada de factores primos será 3, 3, 2, 2 y las divisiones por dimensión serían $Nd^x = 3$, $Nd^y = 3$ y $Nd^z = 2 \times 2 = 4$.

En la Figura 2.4 podemos ver un ejemplo de la partición uniforme de una escena en cuatro subescenas, donde $Nd^x = 2$, $Nd^y = 2$ y $Nd^z = 1$.

2.2.2. Distribución de datos entre los procesadores

La distribución de la escena se ejecuta, como su división uniforme, en paralelo por todos los procesadores. Primero, los polígonos son clasificados según su posición geométrica como totalmente dentro, parcialmente dentro o totalmente fuera. Un polígono está totalmente dentro de una subescena cuando todos sus vértices están dentro de ella (Figura 2.5a). Un polígono está parcialmente dentro si uno o más vértices están fuera y se cumple al menos una de las siguientes condiciones:

- Un vértice está dentro del subespacio. En la Figura 2.5b podemos ver un ejemplo de un polígono con un vértice dentro del subespacio y el resto fuera.

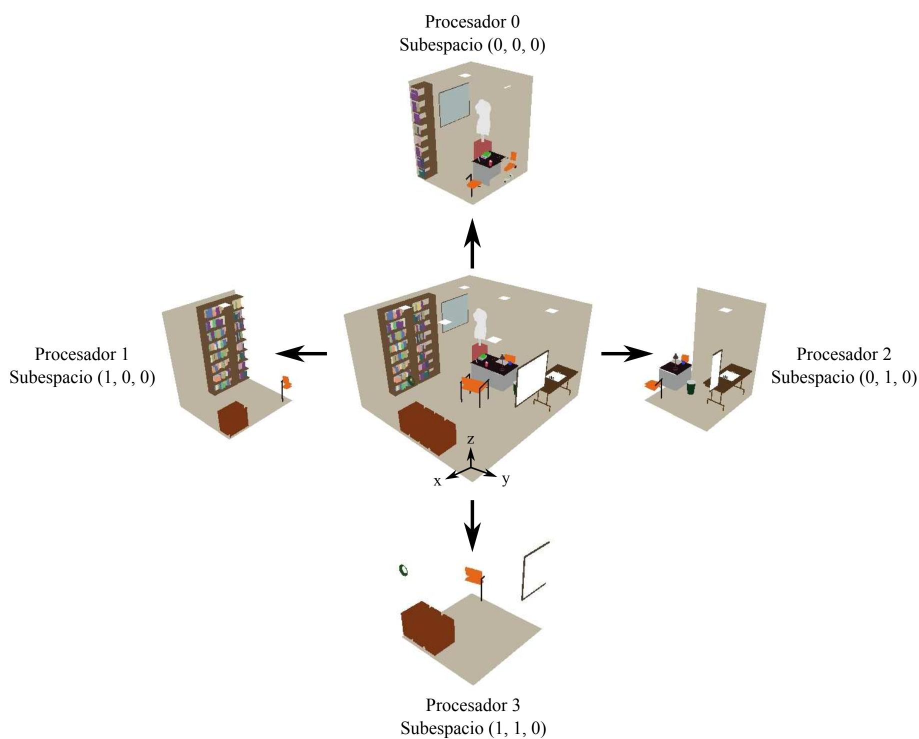


Figura 2.4: División uniforme de una escena en cuatro subescenas.

- Alguna arista del subespacio corta al polígono. En la Figura 2.5c se muestra como un polígono es atravesado por cuatro aristas de la subescena.

- Alguna arista del polígono cruza una cara del subespacio. En la Figura 2.5d vemos un polígono parcialmente dentro del subespacio, que no tiene ningún vértice dentro de él y cuyas aristas cortan las caras del subespacio.

Posteriormente, los polígonos son procesados de acuerdo a la clasificación anterior:

- Los polígonos que están totalmente dentro del subespacio del procesador son añadidos a su lista de polígonos.

- Los polígonos que están parcialmente dentro del subespacio son recortados por el procesador, añadiendo a su lista de polígonos la parte interna y descartando el resto. El método que se usa para el recorte será explicado en la siguiente subsección.

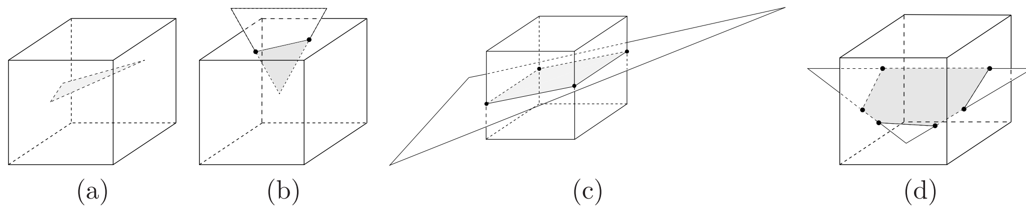


Figura 2.5: Cuatro posibles casos de pertenencia de un polígono a una subescena: (a) totalmente dentro; (b) un vértice dentro; (c) cortado por aristas del subespacio; (d) sus aristas cruzan las caras del subespacio.

- Si el polígono no pertenece al subespacio es totalmente descartado y no se añade a la lista.

Una vez construida la lista de polígonos básicos del subespacio, cada procesador realiza el mallado de su subespacio y comenzará el cálculo de la radiosidad.

2.2.3. Recorte de los polígonos

El recorte de un polígono en un dominio 2D o 3D [34, 46] consiste en eliminar del polígono aquellos trozos que están fuera de las fronteras de dicho dominio. En esta sección se describe un algoritmo que mejora el de Sutherland y Hodgman [129], el cual utiliza una estrategia de divide–y–vencerás. En lugar de esto, el algoritmo presentado en [5, 6, 47] elabora una lista ordenada de los vértices de la parte interior del polígono para, posteriormente, utilizarla para el cálculo del polígono o de los polígonos que se añaden a la lista del procesador. A continuación se mostrará este algoritmo, explicando previamente los distintos tipos de vértices que pueden surgir en el proceso de recorte.

Clasificación de los vértices de la parte interior del polígono

Antes de describir el algoritmo se han de tener en cuenta los posibles tipos de vértices que tendrá el polígono que se obtiene después de la operación de recorte. Si se sitúa un eje de coordenadas en la cara delantera (cara visible por otros polígonos) del polígono original con el eje z hacia donde puede emitir o reflejar luz, los vértices son ordenados siguiendo una rotación de eje z y girando en el sentido de eje x a eje y (ver Figura 2.6). De esta forma, partiendo de la lista de vértices originales, se

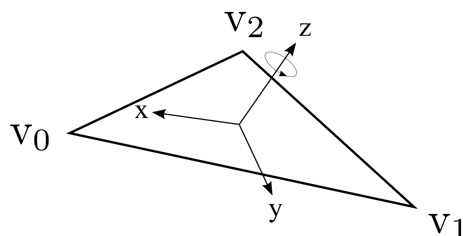


Figura 2.6: Orden en el recorrido de los vértices de un polígono.

obtienen en la parte interior del polígono los siguientes tipos de vértices:

- *Interiores*: son vértices del polígono original que están dentro del subespacio.
- *De intersecciones entrantes*: si el vértice v_i está fuera del subespacio y el vértice v_{i+1} está dentro, entonces la arista $v_i - v_{i+1}$ entra en el subespacio y produce una intersección entrante en una de las caras. Estas intersecciones están en las caras del subespacio que cruzan.
- *De intersecciones salientes*: si el vértice v_i está dentro y el vértice v_{i+1} está fuera, entonces la arista correspondiente sale del subespacio y produce una intersección saliente. Igual que las anteriores, también estas intersecciones están asociadas con las caras del subespacio que cruzan.
- *De intersecciones de arista*: son aquellas que son producidas por la intersección de una arista del subespacio con el polígono.

La Figura 2.7 muestra los cuatro tipos de vértices que se pueden obtener al recortar un polígono. En esta figura el vértice v_3 está dentro del subespacio y, por tanto, es un vértice interior; el siguiente vértice, v_0, está fuera del subespacio, así que la arista $v_3 - v_0$ cruza la cara del subespacio en is_0, produciendo un vértice de intersección saliente. Las aristas $v_0 - v_1$ y $v_1 - v_2$ no crean intersecciones porque todos sus vértices son exteriores. La arista $v_2 - v_3$ cruza una cara del subespacio produciendo un vértice de intersección entrante ie_0. Como el vértice de intersección saliente is_0 y el vértice de intersección entrante ie_0 pertenecen a caras distintas del subespacio, debe haber un vértice de intersección de arista entre ellas, que es ia_0.

Algoritmo de recorte del polígono

Este método consta de tres fases en las cuales se calculan los vértices del recorte del polígono y se almacenan ordenados en una lista. En la primera etapa, los vértices

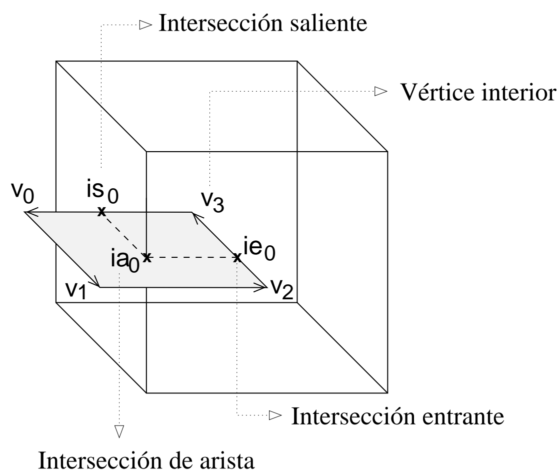


Figura 2.7: Ejemplos de vértices en el recorte de un polígono respecto de un subespacio

Vértice
Tipo
Cara asociada 1
Cara asociada 2
Cara de enlace

Tabla 2.1: Campos de un vértice en la lista de vértices de recorte.

interiores y las intersecciones salientes y entrantes son calculadas. En la segunda fase se calculan las intersecciones de arista y en la última etapa, si es necesario, el polígono recortado es dividido en triángulos (*tessellation*). Sin embargo, insertar los vértices en la posición correcta en la lista es una tarea complicada, porque, además de tener en cuenta el tipo de vértice, se ha de chequear si los vértices anterior y posterior en la lista cumplen las condiciones que sus posiciones requieren. Para esto se crea una estructura de datos que almacena toda la información relativa al vértice, como se muestra en la Tabla 2.1. El campo llamado *tipo* indica el tipo de vértice en la lista. Los campos llamados *Cara asociada 1* y *Cara asociada 2* indican las caras asociadas a los vértices, dejando el valor de *Cara asociada 2* en blanco si el vértice sólo está asociado a una cara. Si el vértice es interior los dos valores estarán en blanco. El campo *Cara de enlace* almacena la cara que es común con el siguiente vértice en la lista. A continuación se comenta más detalladamente el trabajo que se realiza en cada fase.

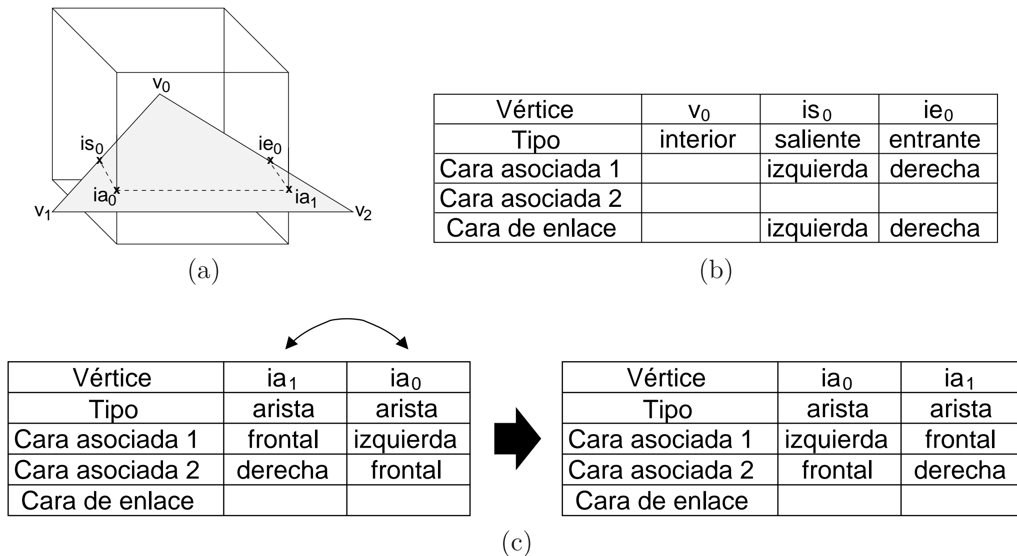
En la primera fase del método de recorte, se recorre la lista de vértices del polígono y para cada vértice v_i se realizan las siguientes tareas:

- Si el vértice está dentro del subespacio es almacenado en la primera posición libre de la lista de recorte.
- Se chequea si cualquiera de las seis caras del subespacio es atravesada por la arista $v_i - v_{i+1}$. Una arista corta como máximo dos caras del subespacio, en cuyo caso producirá una intersección saliente y una intersección entrante. Si cualquiera de las intersecciones coincide con v_i o con v_{i+1} son descartadas. Cuando hay dos intersecciones, la intersección entrante se mete en la primera posición libre de la lista de recorte y la saliente en el siguiente lugar. Si hay solo una intersección la almacenamos en la primera posición libre de la lista. Para cada intersección añadida a la lista los campos *Cara asociada 1* y *Cara de enlace* son inicializados al valor de la cara del subespacio atravesada.

La Figura 2.8a muestra el recorte de un polígono usado para ilustrar el método. El procedimiento comienza en el vértice v_0 que, al ser un vértice interior, se mete en la primera posición de la lista. A continuación, la arista $v_0 - v_1$ sale del subespacio y cruza la cara de la izquierda, que será puesta como *Cara asociada 1* y *Cara de enlace* del vértice generado is_0, que es una intersección de salida, y que es introducida en la lista. El vértice v_1 está fuera del subespacio y la arista $v_1 - v_2$ no cruza el subespacio. Finalmente, v_2 está fuera del subespacio y la arista $v_2 - v_0$ entra en el subespacio por la cara de la derecha, generando una intersección entrante ie_0. Esta intersección se mete en la lista con la cara derecha del subespacio como *Cara asociada 2* y como *Cara de enlace*. La Figura 2.8b muestra la lista de recorte al final de la primera fase.

En la segunda fase se comprueba si el polígono es atravesado por alguna de las aristas del subespacio. Si esto ocurre es necesario almacenar el punto de intersección en una lista secundaria similar a la lista de recorte, siendo los valores de las caras asociadas los correspondientes a las dos que tienen en común la arista del subespacio. Después de completar la lista secundaria se analiza cada uno de sus elementos, comparando sus datos con los de los vértices de la lista principal, para introducirlos en las posiciones correctas en esta lista. El procedimiento detallado es el siguiente:

- Cada punto de corte de la lista secundaria se compara con los puntos de la lista de recorte para ver si coinciden. Esto podría suceder si algún vértice del polígono pertenece a una arista del subespacio, o una arista del polígono se



Vértice	v_0	is_0	ie_0
Tipo	interior	saliente	entrante
Cara asociada 1		izquierda	derecha
Cara asociada 2			
Cara de enlace		izquierda	derecha

Vértice	ia_1	ia_0
Tipo	arista	arista
Cara asociada 1	frontal	izquierda
Cara asociada 2	derecha	frontal
Cara de enlace		

Vértice	ia_0	ia_1
Tipo	arista	arista
Cara asociada 1	izquierda	frontal
Cara asociada 2	frontal	derecha
Cara de enlace		

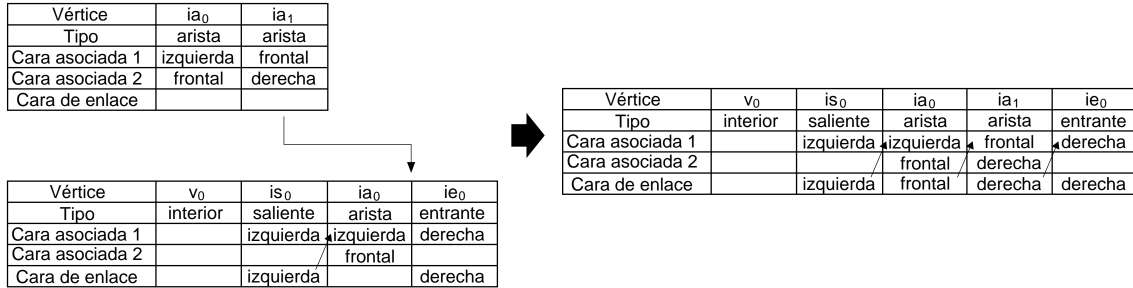
Figura 2.8: Resultados de la primera y segunda fase del método de recorte de los polígonos: (a) ejemplo de recorte; (b) lista de recorte después de la primera fase; (c) intercambio en la lista auxiliar (segunda etapa del método de recorte).

encuentra en el mismo plano que una de las caras del subespacio y corta a la vez una arista y una cara del subespacio. En estos casos el punto es desechado.

- Si no es descartada la intersección hay que buscar en que lugar de la lista de recorte deberíamos colocarla. Para ello se compara el campo de cara de enlace de cada elemento de la lista de recorte con los campos de caras asociadas de la intersección de la lista auxiliar. En caso de coincidencia la intersección de la lista auxiliar se coloca a continuación del elemento con el que ha coincidido.

- Si no se ha conseguido determinar la posición en la que debe ir la intersección de arista es porque, o bien la lista de recorte está vacía con lo cual se colocará en la primera posición, o bien porque debe de ir detrás de otra intersección de arista que está todavía en la lista secundaria. En este caso se intercambia en la lista secundaria la intersección que se está tratando con otra colocada n posiciones hacia adelante. Inicialmente y cada vez que se inserte una intersección en la lista de recorte eliminándola de la secundaria, el valor de n se pone a 1. En los demás casos n se incrementa en una unidad y el proceso se repite con el elemento con el que ha intercambiado la posición.

- Finalmente, la intersección de arista se inserta en la lista de recorte, moviendo el resto de elementos hacia adelante. En el campo de cara de enlace se pone el



Vértice	ia_0	ia_1
Tipo	arista	arista
Cara asociada 1	izquierda	frontal
Cara asociada 2	frontal	derecha
Cara de enlace		

Vértice	v_0	is_0	ia_0	ie_0
Tipo	interior	saliente	arista	entrante
Cara asociada 1		izquierda	izquierda	derecha
Cara asociada 2			frontal	
Cara de enlace		izquierda		derecha

Vértice	v_0	is_0	ia_0	ia_1	ie_0
Tipo	interior	saliente	arista	arista	entrante
Cara asociada 1		izquierda	izquierda	frontal	derecha
Cara asociada 2			frontal	derecha	
Cara de enlace		izquierda	frontal	derecha	derecha

Figura 2.9: Inserción de una intersección de arista durante la segunda fase el método de recorte.

valor de las caras asociadas que no se corresponde con el valor de la cara de enlace del elemento precedente en la lista de recorte.

En la tabla de la izquierda de la Figura 2.8c se muestra la lista auxiliar con las intersecciones de arista para el ejemplo de la Figura 2.8a. El orden de las intersecciones de arista en esta lista depende del orden de recorrido de las aristas del subespacio. Primero se procesa ia_1 y se busca su posición en la lista de recorte, pero no se encuentra porque su posición correcta es a continuación de ia_0, que todavía no ha sido insertada. Por este motivo se intercambia ia_1 con ia_0 que es la siguiente en la lista. La tabla de la derecha de la Figura 2.8c muestra la lista auxiliar después de hacer el intercambio. Luego se procesa ia_0 cuya posición se encuentra a continuación de is_0, pues el valor de la *Cara de enlace* de is_0 y el de la *Cara asociada 1* de ia_0 coinciden. Finalmente, se procesa ia_1 cuya posición se encuentra a continuación de ia_0. En la Figura 2.9 se muestra como queda la lista de recorte al final de la segunda fase del método de recorte, después de insertar ia_1 en su posición.

En la tercera fase, si el polígono resultante no es un triángulo se divide en triángulos (*tessellation*). Para ello, se forman los triángulos a partir del vértice v_0 de la lista, siendo el primero el formado por (v_0, v_1, v_2), el segundo por (v_0, v_2, v_3) y así sucesivamente hasta llegar al último vértice.

2.3. Paralelización del algoritmo

De entre las posibles soluciones para la paralelización del algoritmo de radiosidad comentadas en la Sección 1.7, para este trabajo se ha escogido la implementación en sistemas de memoria distribuida, repartiendo los datos de la escena entre las

memorias locales de cada procesador. Esta elección posibilita trabajar con escenas de mayor tamaño. Para reducir el número de comunicaciones necesarias para el intercambio de información entre los procesadores, la distribución de los datos se basa normalmente en una división de la escena en entornos locales.

Como se ha explicado en la sección anterior, la distribución de los datos se ha realizado mediante una partición uniforme y estática de la escena en subespacios. Cada subespacio es asignado a un procesador. La planificación de las tareas del algoritmo se ha llevado a cabo mediante una paralelización de *grano grueso*, es decir, cada procesador calcula la radiosidad del conjunto de elementos que pertenecen a su subescena. Por tanto, el algoritmo paralelo se ha implementado empleando un enfoque SPMD (*single–program, multiple–data*), de manera que todos los procesadores ejecutan el mismo código sobre datos diferentes. El modelo de programación paralela utilizado es el de paso de mensajes, para lo cual se han usado las funciones proporcionadas por la biblioteca *MPI* (*Message Passing Interface*) [91] para la comunicación entre los procesadores.

La comunicación entre los procesadores tiene lugar al final de cada iteración del cálculo de la radiosidad local. Cada comunicación incluye información sobre el polígono disparado y su visibilidad en su subespacio. Para la determinación de la visibilidad hemos utilizado una modificación de la técnica de las *máscaras de visibilidad* [7]. El esquema de nuestro algoritmo paralelo de radiosidad progresiva se muestra en la Figura 2.10. En esta figura y en la explicación que se realiza a continuación se indica con un superíndice l que el elemento o parámetro correspondiente pertenece a la subescena l. Cada fase del algoritmo se analiza en detalle en las siguientes subsecciones. En la Subsección 2.3.1 se expone cómo se calcula la radiosidad local y cómo se prepara la información que se ha de comunicar a los otros procesadores; en la Subsección 2.3.2 se explica cómo se propaga el disparo de un polígono por toda la escena; por último, en la Subsección 2.3.3 es presentado el método para realizar el chequeo distribuido del fin de cada iteración y de la convergencia del algoritmo.

2.3.1. Cálculo de la radiosidad local

Una vez finalizada la distribución de la escena, cada procesador dispone de un conjunto de polígonos locales sobre los que ha de calcular la radiosidad. En cada iteración, cada procesador escoge el polígono con mayor potencia radiante sin disparar S_i^l (líneas desde la 9 a la 12 de la Figura 2.10) entre todos los polígonos de

```
1   /* Inicializamos valores */
2   for (S_i^l = subescena^l→Elemento_inicial; S_i^l; S_i^l = S_i^l →siguiente)
                                                        : l = 1, ··· , Nd {
3        B_i^l = B_ei^l;
4        ΔB_i^l = B_ei^l;
5   }
6
7   /* Proceso iterativo */
8   while (no convergió) {
9          S_i^l = 0;
10         /* BUCLE1: seleccionamos elemento con mayor ΔB_i^l · A_i^l
                    para subescena l,  l = 1, ··· , Nd */
11         for (S_j^l = subescena^l→Elemento_inicial; S_j^l; S_j^l = S_j^l →siguiente)
12              S_i^l = MaxPot(S_i^l, S_j^l);
13
14         /* BUCLE2: Calculamos, solo primera vez que un elemento es
                    disparado, elementos que interacciona con S_i^l*/
15         for (S_j^l = subescena^l→Elemento_inicial; S_j^l; S_j^l = S_j^l →siguiente)
16              Interacción(S_j^l, S_i^l);
17
18         /* BUCLE3: Disparamos radiosidad desde el elemento S_i^l */
19         for (S_j^l = S_i^l →Interacción_inicial; S_j^l;
                  S_j^l l_i →Interacción→siguiente) {
20              if (S_i^l ∈ subescena^l)
21                   V_ji = Visibilidad_local(S_i^l, S_j^l);
22              else
23                   V_ji = Visibilidad_global(S_i^l, S_j^l);
24              G_ji = Geometría(S_i^l, S_j^l);
25              F_ji = V_ji · G_ji;
26              ΔB_j^l = ΔB_j^l + ΔB_i^l · ρ_j · F_ji;
27              B_j^l = B_j^l + ΔB_i^l · ρ_j · F_ji;
28         }
29         Propagación(S_i^l);
30         Recepción();
31         ΔB_i^l = 0;
32         Comprobación_convergencia();
33  }
```

Figura 2.10: Algoritmo de radiosidad progresiva distribuido.

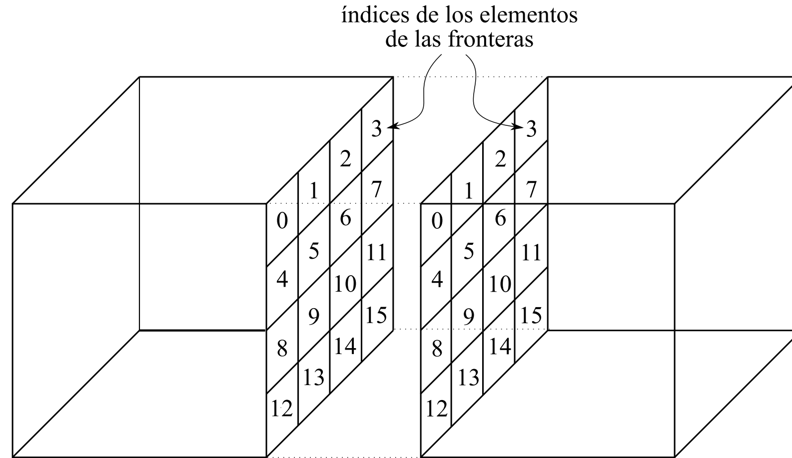


Figura 2.11: Correspondencia de elementos en fronteras enfrentadas.

su subespacio local l y los que ha recibido de otros procesadores. A continuación, se determinan los polígonos de la subescena que interaccionan con este (líneas desde la 14 a la 16). Después de esto, se calcula la acumulación local de la radiosidad siguiendo una versión modificada del algoritmo estándar de radiosidad (líneas desde la 18 a la 28).

Una vez se ha actualizado la radiosidad local en la subescena, la potencia de S_i^l se propaga por el resto de la escena (línea 29), con lo cual se ha de enviar cada polígono disparado junto con su información de visibilidad a los otros procesadores. La información de visibilidad se guarda en unas estructuras llamadas *máscaras de visibilidad* [7]. Esta técnica fue desarrollada para gestionar la transferencia de energía entre subespacios en la implementación paralela distribuida del algoritmo de radiosidad progresiva, utilizando una *interfaz virtual* que representa la frontera entre dos subespacios a través de la cual se transfiere energía entre los dos a nivel de polígono, es decir, se transfiere la energía de un polígono de cada vez. La cantidad de energía transmitida se regula mediante las máscaras de visibilidad que almacenan las oclusiones visuales de un polígono en su subespacio.

En nuestro trabajo se llama *frontera* a la cara que separa dos subespacios. Cada procesador divide esta frontera, $frontera_i$, en $Nm \times Nm$ elementos rectangulares, M_{ij}, que solo se emplean para la elaboración de las máscaras de visibilidad. Estos elementos se construyen para que sean visibles hacia el interior del subespacio. Además se crean de manera que su tamaño, forma y posición en el espacio sean los mismos que los de la frontera que se encuentra en su cara enfrentada, coincidiendo también sus índices j. Así las fronteras adyacentes de dos subespacios vecinos actúan como

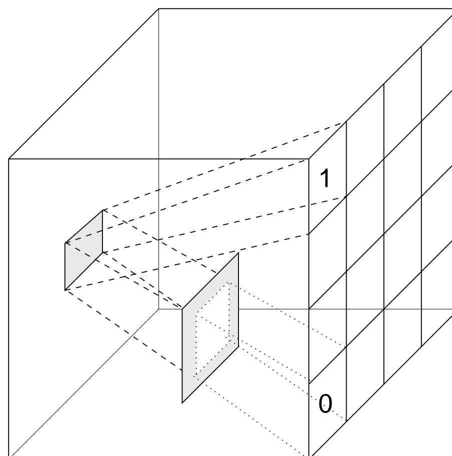


Figura 2.12: Cálculo de la máscara de visibilidad inicial.

una única frontera que los separa. En la Figura 2.11 se muestra un ejemplo de dos subespacios adyacentes y como los elementos de las caras enfrentadas coinciden, uno a uno, en tamaño, forma, posición e índice.

Cada uno de estos elementos tiene asociado un valor booleano. Este valor es 1 si el elemento M_{ij} de la frontera i es visible desde el polígono seleccionado y 0 en otro caso (ver Figura 2.12). La obtención de los valores de la máscara se realiza de igual forma que el cálculo de la visibilidad entre dos polígonos locales. Destacar que la estructura de visibilidad propuesta tiene unos requerimientos de almacenamiento reducidos. Finalmente, una vez que se tiene la máscara de visibilidad inicial, el procesador la envía junto con una copia del polígono al procesador vecino con el que tiene la frontera en común (línea 29 de la Figura 2.10).

Es conveniente indicar que los polígonos disparados de una subescena y sus correspondientes máscaras de visibilidad son copiados en las subescenas vecinas. Esto significa que estos subespacios tendrán que procesar polígonos locales y polígonos externos. Esto hay que tenerlo en cuenta cuando se calcula la acumulación de radiosidad dentro de cada subescena (líneas desde la 18 a la 28 de la Figura 2.10). En el caso de procesar un polígono externo la acumulación de radiosidad se calcula de una manera similar a un polígono local, pero la información de visibilidad (línea 23) se obtiene trazando rayos desde el polígono a la subescena a través de sus máscaras de visibilidad. El proceso continúa por el resto de los subespacios adyacentes con la misma estrategia, copiando los polígonos disparados en las subescenas ya procesadas y calculando y actualizando las máscara de visibilidad teniendo en cuenta la información local de los posibles obstáculos a los rayos. Todo esto se verá más

detalladamente a continuación.

2.3.2. Disparo y propagación de un polígono en el resto de la escena

Cuando un procesador recibe un polígono (línea 30 Figura 2.10) lo almacena en una lista de polígonos pendientes mientras no se disponga de toda la información de visibilidad relativa a ese polígono. Una vez que la información de la visibilidad ha sido completada, el polígono es eliminado de la lista de polígonos pendientes y pasa a la lista de polígonos con energía sin disparar. Además, este elemento puede ser enviado a otros procesadores vecinos.

Para poder determinar cuando la información de visibilidad está completa y evitar el envío de un polígono varias veces a un procesador, a cada polígono se le asocia una variable con dos campos: uno con la información que identifica al procesador que envió el polígono y otro que identifica al polígono en particular. Con esta información el procesador determina las fronteras desde las cuales se ha de recibir la información de visibilidad para ese elemento, por tanto cuántas máscaras de visibilidad ha de recibir, y también a través de las cuales se han de enviar las nuevas máscaras de visibilidad que hay que generar una vez actualizadas con la información de visibilidad local. Sean las coordenadas del subespacio o procesador local (c_x, c_y, c_z) y las del subespacio o procesador al que pertenece el polígono (s_x, s_y, s_z), el análisis para determinar cuando una frontera es de entrada o de salida respecto a ese elemento consiste en aplicar estas expresiones (se indica la coordenada x, siendo los casos de las coordenadas y y z análogos):

$$\begin{aligned} F(c_x+1) &= \mathrm{OUT}\cdot(s_x \le c_x) + \mathrm{IN}\cdot(s_x > c_x) \\ F(c_x-1) &= \mathrm{OUT}\cdot(s_x \ge c_x) + \mathrm{IN}\cdot(s_x < c_x) \end{aligned} \tag{2.4}$$

donde $F(c_x+1)$ y $F(c_x-1)$ son las fronteras entre el subespacio asignado al procesador considerado (c_x, c_y, c_z) y los subespacios (c_x+1, c_y, c_z) y (c_x-1, c_y, c_z) respectivamente, OUT indica si la frontera es de salida e IN si es de entrada.

Una vez que un polígono y toda su información de visibilidad ha sido recibida por las fronteras de entrada, ha de ser reenviado a los subespacios vecinos. Si el subespacio no tiene fronteras de salida el proceso de transmisión de la información del elemento ha finalizado. En otro caso han de ser calculadas para cada frontera de salida las *máscaras de visibilidad intermedias*. Estas máscaras almacenan la in-

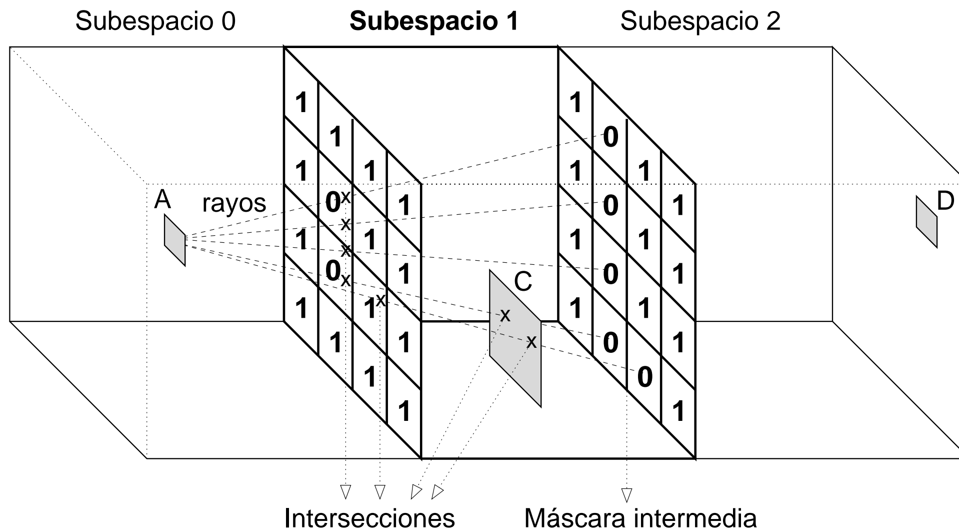


Figura 2.13: Máscara intermedia de visibilidad.

formación de visibilidad entre un elemento y una frontera de salida perteneciente a un subespacio distinto al del elemento.

La idea básica consiste en obtener las nuevas máscaras de visibilidad trazando rayos desde el polígono recibido a través de las fronteras de entrada y hacia las fronteras de salida. Los elementos de las fronteras de salida tendrán un valor 0 o 1 dependiendo de si son visibles desde el polígono original, teniendo en cuenta que los elementos de la frontera de entrada con valor 0 actúan como elementos que ocluyen a los rayos. En la Figura 2.13 se muestra el cálculo de la máscara intermedia para la frontera entre los subespacios 1 y 2 respecto del polígono A.

Las máscaras de visibilidad han de ser propagadas con cada polígono disparado a través de toda la escena. En la Figura 2.14 se ilustra un ejemplo en dos dimensiones de la creación y envío de las máscaras de visibilidad. El procesador 0 selecciona el polígono S_1, lo dispara en su subespacio y crea las máscaras de visibilidad iniciales, m_{10} (Figura 2.14a). A continuación, el polígono es enviado a los procesadores vecinos junto con las máscaras correspondientes (Figura 2.14b). Los procesadores 1 y 2, después de recibir la información del polígono S_1 junto con su máscara de visibilidad para la frontera de entrada, lo disparan en su subespacio y crean las máscaras intermedias para la frontera de salida, m_{11} y m_{12} (Figura 2.14c). Una vez hecho esto, los procesadores 1 y 2 envían al procesador 3 los datos del polígono junto con las máscaras creadas (Figura 2.14d). Este procesador, una vez recibidas estas dos máscaras, completará la información de visibilidad del polígono S_1 y lo disparará en

su subespacio, no continuando la propagación ya que no tiene fronteras de salida. En la Figura 2.14e se puede observar que el procesador 3 utilizando los datos recibidos (valores de S_1 y las máscaras m_{11} y m_{12}) determina que el polígono S_1 no es visible desde S_5.

Durante esta etapa las comunicaciones entre los procesadores se realizan empleando funciones no bloqueantes (*MPI_Isend*, *MPI_Irecv*) que permiten solapar la ejecución de otras instrucciones, ya que el emisor y el receptor no tienen que estar bloqueados desde que se inicia el envío hasta que se completa la recepción. Para ello, se han creado una serie de colas de envío y recepción, donde se guardan los datos que son enviados y recibidos. Estas colas son necesarias porque las rutinas de envío no bloqueantes deben disponer de la información hasta que el envío finalice, sin que sea alterada por tareas posteriores al inicio del envío.

2.3.3. Comprobación de la convergencia

Al final de cada iteración, después de la etapa de comunicaciones, la radiosidad sin disparar del polígono seleccionado, ΔB_i^l, se anula (línea 31 de la Figura 2.10) y a continuación cada procesador comprueba la convergencia del algoritmo (línea 32). Esta comprobación es un problema complejo en un sistema distribuido, porque hay que asegurar que los elementos disparados se han propagado por toda la escena. Además se debe tener en cuenta que estamos utilizando comunicaciones no bloqueantes para optimizar el rendimiento del algoritmo. Esto dificulta aún más que cada procesador determine la convergencia del algoritmo, ya que cada uno llegará a esta fase en diferentes momentos dependiendo de su carga de trabajo. En esta sección se muestra la solución que hemos propuesto para resolver todas estas dificultades que se presentan en el cálculo distribuido de la convergencia del algoritmo.

En la primera iteración y para que todos los procesadores tengan el mismo criterio de finalización, todos los procesadores buscan el polígono con el valor mayor de potencia radiante y después ejecutan una operación de reducción global (*MPI_Allreduce*) que calcula y guarda el máximo de entre todos los valores de las subescenas. De esta manera este valor es el mismo en todos los procesadores. En las siguientes iteraciones, cada procesador comprueba la convergencia del algoritmo en dos fases. En la primera el procesador compara un porcentaje del valor máximo de la potencia radiante disparada con el de la potencia disparada en esa iteración. Si esta potencia es mayor o la lista de polígonos recibidos pendientes de ser disparados no está vacía, el procesador pasa a ejecutar una nueva iteración. En caso contrario

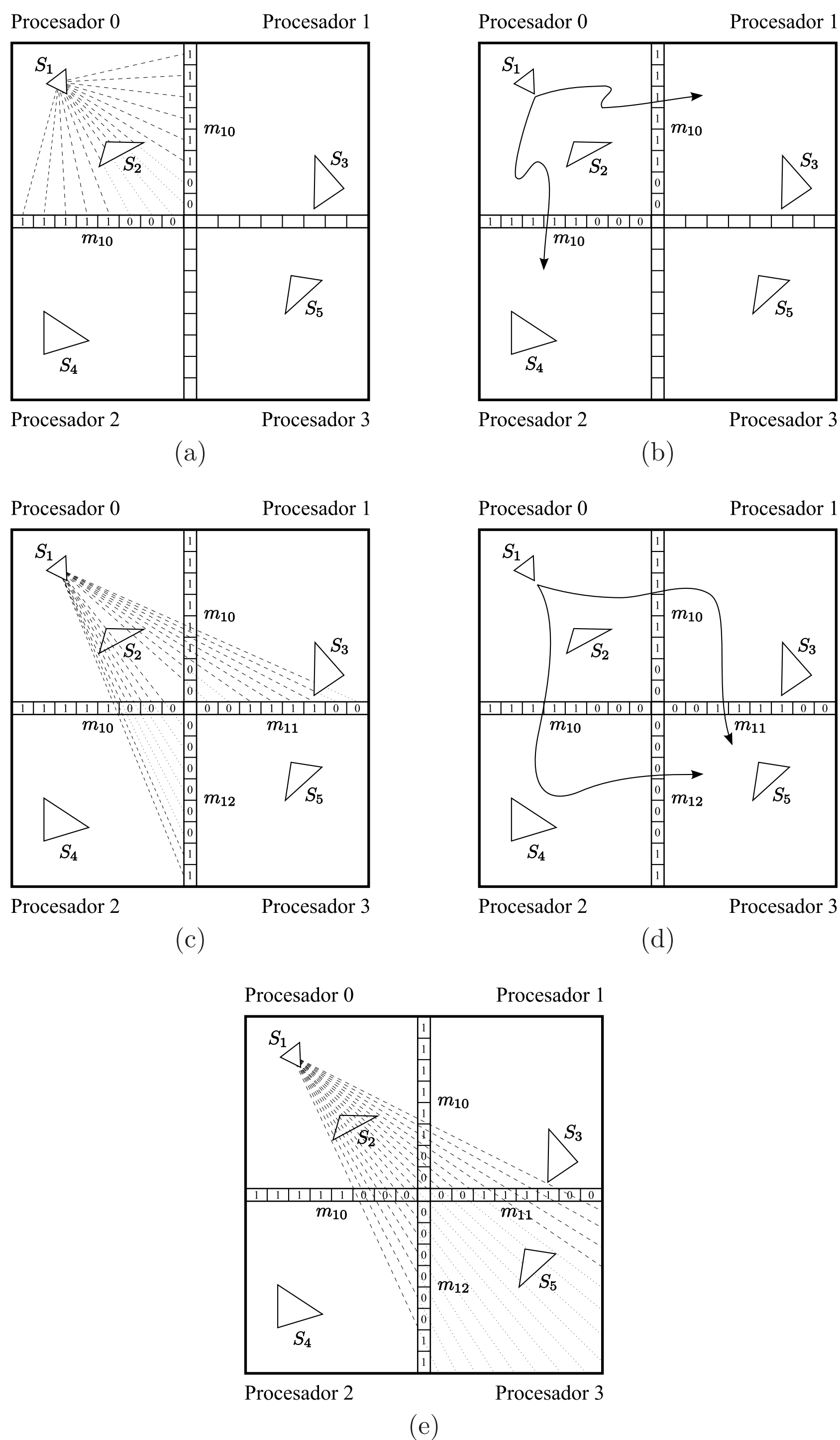


Figura 2.14: Determinación de las máscaras de visibilidad de un polígono para cuatro procesadores.

	Procesadores			
	1	2	3	4
NS_S	3	1	0	0
NS_R	0	2	3	2

Tabla 2.2: Tabla de comprobación de la convergencia para 4 procesadores.

el procesador comienza una segunda fase.

En la segunda fase, en primer lugar el procesador i envía un mensaje no bloqueante a los otros con la siguiente información:

$$NS_T^i = NS_S^i + NS_R^i \tag{2.5}$$

donde NS_S^i es el número de polígonos locales del procesador i enviados a otros procesadores, y NS_R^i es el número polígonos recibidos por el procesador i desde sus procesadores vecinos. A continuación entra en un estado de espera en el cual se pueden dar dos situaciones: o bien puede recibir polígonos disparados por otros procesadores, o bien mensajes de los procesadores que han llegado a esta fase con los valores NS_T^j, siendo $j = \{1, \ldots, Nd\}, j \neq i$. En el primer caso, saldrá de este estado, pasando a ejecutar otra iteración con los polígonos recibidos en cuanto tenga completa toda la información sobre ellos. En el segundo caso hace una comprobación de todos los valores NS_T^j recibidos. Si todos estos valores son iguales al suyo ($NS_T^j = NS_T^i, \forall j$) el algoritmo ha convergido y finaliza el proceso iterativo. Si no, permanece en el estado de espera porque algún polígono podría ser recibido por este procesador o por otros y tendría que ser procesado.

Para aclarar esto consideremos el ejemplo de la Tabla 2.2, donde se indica el número de polígonos locales enviados al resto de procesadores por cada procesador y el número de polígonos recibidos de otros procesadores. Por ejemplo, el Procesador 1 ha enviado 3 polígonos locales y no ha recibido ninguno, el Procesador 2 ha enviado 1 y ha recibido 2, el Procesador 3 no ha enviado ninguno y ha recibido 3 y el Procesador 4 tampoco ha enviado y ha recibido solo 2. Teniendo en cuenta esta información analicemos el estado de la comprobación de la convergencia en una fase específica de la computación, donde se verifican las siguientes condiciones:

- El Procesador 1 se encuentra disparando su tercer polígono local y aún no ha enviado ningún mensaje con el valor NS_T^1 a los otros polígonos.

- El Procesador 2 ya ha enviado el mensaje con el valor $NS_T^2 = 3$.
- El Procesador 3 ha recibido 3 polígonos, pero aún non ha procesado el último porque no ha completado toda su información.
- El Procesador 4 ha recibido 2 polígonos, los ha procesado y ha enviado el mensaje con el valor $NS_T^4 = 2$.

Según estas condiciones, tanto el Procesador 1 como el Procesador 3 se encuentran en la primera fase disparando un polígono local, en el caso del Procesador 1, o esperando información de un polígono recibido en el caso del Procesador 3. En cambio, el Procesador 2 y el Procesador 4 se encuentran en la segunda fase, después de enviar su mensaje. Al realizar la comprobación de los valores NS_T recibidos, determinarán que aún no ha convergido el algoritmo por ser distintos al suyo local y faltarles los valores NS_T^1 y NS_T^3, con lo cual entrarán en un estado de espera para recibir nuevos polígonos y nuevos valores de NS_T. En este ejemplo, si no son disparados nuevos polígonos locales en ningún subespacio, la convergencia se alcanzará cuando todos los valores NS_T sean 4 y estos hayan sido recibidos por todos los procesadores.

El empleo de comunicaciones no bloqueantes para la determinación de la convergencia complica el proceso, pero permite que unos procesadores puedan seguir trabajando mientras los otros están a la espera de recibir nuevos datos o de finalizar el algoritmo. Además evita la utilización de ineficientes sincronizaciones mediante comunicaciones bloqueantes.

2.4. La partición de la escena como método para incrementar la localidad de los datos

En general, el rendimiento de un código está influenciado, entre otros factores, por la reutilización de los datos y la explotación de su localidad, considerando que un dato es reusado cuando su siguiente utilización no implica un acceso a un nivel más bajo dentro de la jerarquía de memoria del computador. De hecho, puede haber una gran diferencia de rendimiento entre los programas que son diseñados para optimizar el uso de la caché frente a aquellos que no lo son. Por ejemplo, en el contexto de los gráficos por computador, en [18] se propone una arquitectura para visualización

gráfica con prebúsquedas de datos para la caché de píxeles que alcanza ganancias de un 32 % comparada con las arquitecturas convencionales.

Al comienzo de la Sección 2.3 se comentó que la división de la escena permite aumentar la localidad, lo cual beneficia el funcionamiento del algoritmo paralelo. Sin embargo, no se ha justificado ni cuantificado este beneficio. En esta sección se demostrará que este beneficio es real y que incluso permite mejorar el rendimiento de la versión secuencial del algoritmo.

Para hacer el estudio se utilizará el algoritmo secuencial, modificado para su aplicación a bloques de datos mediante la división uniforme de la escena. Esta técnica tiene asociado un coste computacional que hay que considerar, pero se verá que, teniendo en cuenta la tasa de fallos caché, la reutilización de datos puede ser empleada para la optimización del algoritmo, y con ello se logra reducir su tiempo de ejecución [105].

A continuación, primero se analizará la reutilización de los datos en el algoritmo y después será presentada nuestra estrategia de transformación del algoritmo con la aplicación a bloques de datos. Esta modificación del algoritmo está basada en nuestra metodología de división uniforme de la escena que se ha explicado anteriormente para la implementación paralela del algoritmo.

2.4.1. Análisis de la reutilización de datos en el algoritmo de radiosidad progresiva

En este apartado se analizará el comportamiento de la reutilización de los datos para el algoritmo de radiosidad progresiva. Como se verá a continuación, el algoritmo de radiosidad progresiva requiere que, para escenas de tamaño estándar, el mismo polígono sea accedido varias veces, pudiendo provocar múltiples accesos muy espaciados y, por consiguiente, varias referencias a memoria desde los niveles más bajos de la jerarquía, ralentizando la ejecución del programa.

Como se puede observar en el pseudocódigo de la Figura 2.1, el algoritmo consta de tres bucles principales donde se realizan operaciones de lectura y escritura sobre la lista de los Ne polígonos de la escena. Sea el número de referencias de memoria, Nr, el número de veces que un polígono es accedido. Teniendo en cuenta esta definición y asumiendo tamaños usuales de escenas, el número de referencias de memoria, Nr, requerido por las diferentes estructuras de datos para realizar el cálculo de la radiosidad en cada bucle es:

$$\begin{aligned} Nr_{\text{BUCLE1}} &= Ne \times t \\ Nr_{\text{BUCLE2}} &= Ne \times Nk \\ Nr_{\text{BUCLE3}} &= \overline{M} \times t \end{aligned} \tag{2.6}$$

donde t es el número de iteraciones, Nk el número de polígonos distintos que son disparados, con $Nk \leq t$, y $\overline{M}$ el número medio de objetos accedidos por iteración. BUCLE1 (líneas de la 9 a la 12 de la Figura 2.1) selecciona el polígono S_i con el valor máximo de potencia radiante y BUCLE2 (líneas de la 14 a la 16) los polígonos S_j que interaccionan con S_i. En ambos bucles los accesos son de tipo secuencial. Por otro lado, BUCLE3 (líneas de la 18 a la 25) presenta un comportamiento que es difícil de predecir. Esto es debido a la estructura irregular del algoritmo que se emplea para calcular la visibilidad. En nuestro caso, como ya se ha comentado, se emplean técnicas direccionales que implican el acceso a los candidatos potenciales que están incluidos en la lista de interacciones del polígono seleccionado S_i. Teniendo en cuenta esta información, el número total de referencias es:

$$Nr = (Ne + \overline{M}) \times t + Nk \times Ne \tag{2.7}$$

Como resultado de esto, cada polígono tiene que ser accedido en múltiples ocasiones, lo cual deriva en un gran ancho de banda de la memoria. Como Ne es un número muy grande para los tamaños habituales de escenas, la caché no puede contener toda la información necesaria durante la ejecución del algoritmo de radiosidad progresiva estándar (pseudocódigo de la Figura 2.1). Por tanto, los fallos de reemplazo ocurrirán en todos los bucles (siguiendo la notación de [38] llamaremos fallos de reemplazo al conjunto de fallos de capacidad y de conflicto [98]).

El rendimiento de un código está muy influenciado por la reutilización de los datos y por los recursos de memoria disponibles. El número de veces que se accede a un polígono durante toda la ejecución del algoritmo constituye una herramienta para evaluar la transferencia de datos entre los niveles de la jerarquía de memoria para escenas grandes. Específicamente, la redundancia de accesos para el algoritmo progresivo puede ser medida usando la siguiente ecuación:

$$Ra = \frac{Nr}{Ne} = \frac{(Ne + \overline{M}) \times t + Nk \times Ne}{Ne} = t + \frac{\overline{M}}{Ne} \times t + Nk \tag{2.8}$$

Para las escenas que se han utilizado en este trabajo y para el criterio de convergencia empleado, se ha observado que $\overline{M} \simeq Ne^2/4$ y $Nk \simeq t$. Debido a que las

escenas usadas en nuestras pruebas tienen diferentes estructuras y tamaños, consideramos que estas aproximaciones son representativas de un caso genérico. Por consiguiente:

$$Ra = \left(2 + \frac{Ne}{4}\right) \times t \tag{2.9}$$

Consecuentemente, cada polígono es accedido un promedio de $[(2 + Ne/4) \times t]$ veces. Teniendo en cuenta el elevado número de referencias, es importante maximizar el número de veces que se accede a un polígono mientras esté en la caché. Reducir el número de fallos caché va a permitir reducir el número de veces que el mismo polígono tiene que ser almacenado en la caché y, como consecuencia, disminuir el tiempo de ejecución del algoritmo.

2.4.2. Transformación por bloques del algoritmo para optimizar la localidad de los datos

Trasformar los procedimientos de un programa puede mejorar la localidad temporal y espacial de los datos. Para analizar como influye el método de partición de la escena en la mejora de la localidad hemos adaptado el algoritmo de radiosidad progresiva distribuido (Figura 2.10) a una plataforma monoprocesador. Para ello se ha dividido el volumen en Nd subespacios iguales y disjuntos, donde Nd es el número de particiones, utilizando la misma estrategia que en la versión paralela ya explicada. Los subespacios serán procesados secuencialmente, es decir, se comienza por $subespacio^1$ y se continúa hasta $subespacio^{Nd}$. Para el cálculo de la visibilidad entre polígonos de diferentes subespacios se ha usado de nuevo la técnica de las máscaras de visibilidad. Así la información de visibilidad de cada subespacio se puede emplear para el cálculo en las otras subescenas. Esto implica que cada subescena no tiene que acceder a la información asociada a los polígonos de otras subescenas sino solo a sus máscaras de visibilidad. Esto permite la reducción de los accesos a datos con baja localidad.

La división de la escena y las técnicas de las máscaras de visibilidad han de ser empleadas con cautela, porque ambos algoritmos tienen un coste asociado que podría producir un incremento en el tiempo de ejecución del algoritmo. No obstante, como se mostrará en los resultados experimentales, para un número específico de particiones estos costes son superados por los beneficios asociados con la explotación de la localidad de los datos.

En esta versión secuencial, la propagación de la potencia de un polígono por el resto de la escena se realiza copiando los datos de este polígono junto con sus máscaras de visibilidad en las subescenas vecinas. Esto hay que tenerlo en cuenta, igual que en el algoritmo distribuido, cuando se calcula la acumulación de radiosidad dentro de cada subescena (líneas desde la 18 a la 28 de la Figura 2.10).

Mediante esta técnica se puede ejecutar el algoritmo progresivo utilizando la estrategia de partición de la escena y, como se verá a continuación, se pueden obtener beneficios en cuanto a la reducción del número de fallos caché.

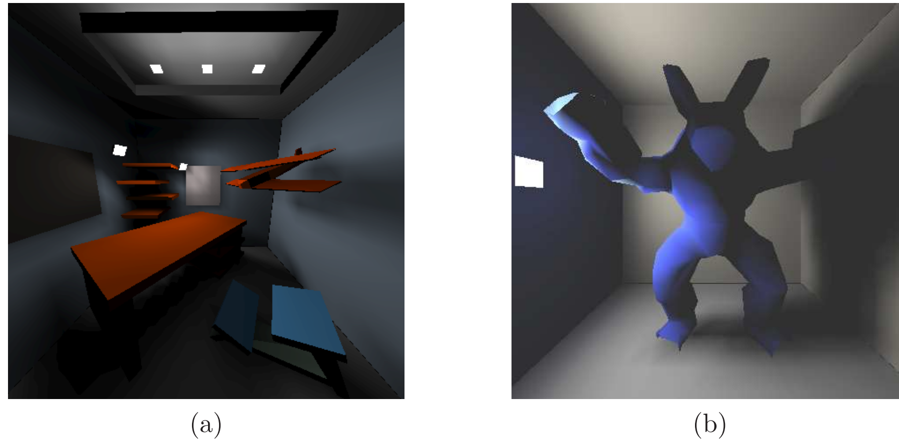
Si se pudiera almacenar completamente cada subescena en la caché, los cálculos locales podrían realizarse sin acceder a los niveles más bajos de la jerarquía de memoria. Así, el número total de referencias de memoria para el algoritmo modificado es:

$$Nr = \sum_{l=1}^{Nd} \left(\underbrace{Ne^l \times t}_{\text{BUCLE1}} + \underbrace{Ne^l \times Nk}_{\text{BUCLE2}} + \underbrace{\overline{M^l} \times t}_{\text{BUCLE3}} \right) + (Nd - 1) \times Nm^2 \times t \tag{2.10}$$

donde Nd es el número de subescenas, Ne^l el número de polígonos en cada subescena, $\overline{M^l}$ el promedio por iteración del número de objetos accedidos en cada subescena, y cada máscara está compuesta por $(Nm \times Nm)$ polígonos. Indicar que el último término de la ecuación está asociado con el cálculo de las máscaras. Teniendo en cuenta la definición de Nr este valor será menor cuando los polígonos Ne^l estén todos en la caché. En este caso, una vez que la subescena esté almacenada en la caché después de BUCLE1 (líneas desde la 9 a la 12 de la Figura 2.10), la información es reutilizada por BUCLE2 (líneas desde la 14 a la 16) y BUCLE3 (líneas desde la 18 a la 28) sin requerir más información. Esto significa que los términos relativos a BUCLE2 y BUCLE3 pueden ser eliminados de la ecuación de Nr. Por consiguiente, la redundancia de accesos es

$$Ra = \sum_{l=1}^{Nd} \frac{Ne^l \times t}{Ne} = t \tag{2.11}$$

Comparando este resultado con el obtenido para el algoritmo original (ver Ecuación 2.9), se tienen potencialmente $(2 + Ne/4)$ veces menos fallos caché usando la estrategia de partición de la escena. Hay que indicar que se ha omitido el término asociado al cálculo de las máscaras porque se está analizando solo la redundancia de accesos para los polígonos.



(a) (b)

Figura 2.15: Escenas iluminadas: (a) *habitación* y (b) *armadillo.*

2.5. Resultados experimentales

En esta sección, en primer lugar, es presentado un análisis del método utilizado para el recorte de los polígonos para la distribución de la escena. A continuación se muestran los resultados de rendimiento de nuestra implementación paralela de la radiosidad progresiva. Después se pasa a estudiar la implementación secuencial, analizando los beneficios obtenidos por la estrategia de división de la escena en términos de tiempos de ejecución y reducción de los fallos caché. Finalmente, también se ha verificado que la calidad de las imágenes resultantes no se ha visto afectada por el empleo de estas técnicas.

Para evaluar nuestro algoritmo paralelo de radiosidad progresiva se han usado dos *clústeres* diferentes de PC's: uno con procesadores Pentium III a 800 MHz conectados a través de una red Fast Ethernet de 100 Mbps, que denominaremos *F. Ethernet Clúster*, y otro con procesadores Intel XEON IV a 1.8 GHz conectados a través de una red SCI (*Scalable Coherent Interface*) de 250 Mbps, al que llamaremos *SCI Clúster*. Como estos dos *clústeres* tienen dos redes de conexión diferentes que proporcionan las comunicaciones de datos entre los nodos, lo que tratamos de probar es que nuestro método obtiene un buen rendimiento independientemente de la tecnología empleada en la conexión de la red de computadores.

En esta sección se presentan los resultados obtenidos al aplicar nuestro método al cálculo de la iluminación global de tres escenas. En la Figura 2.15 se muestran dos de ellas: la clásica habitación de Hanrahan, que denominaremos simplemente *habitación* (Figura 2.15a), y otra en la que se incluye un modelo poligonal complejo

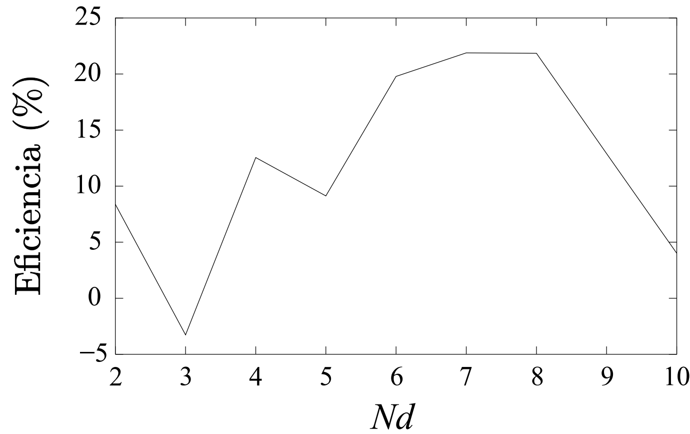


Figura 2.16: Comparación del método de recorte usado con el algoritmo de Sutherland y Hodgman.

en una habitación vacía y que llamaremos *armadillo* (Figura 2.15b). La tercera escena, que denominamos *habitación grande*, es básicamente la escena *habitación* duplicada en una especie de efecto espejo, eliminando en la parte añadida las luces y algunos objetos como el avión o la estantería. Sin haber realizado la partición en subespacios, estas escenas tienen, respectivamente, 5306, 3999 y 7070 triángulos. La escena *habitación* ha sido utilizada habitualmente para evaluar los métodos de iluminación y la escena *armadillo* fue elegida para medir el rendimiento de nuestro método aplicado a escenas no uniformes.

2.5.1. Método de recorte de polígonos

En primer lugar se ha evaluado el algoritmo de recorte de los polígonos. Para ello se han comparado los tiempos de ejecución del recorte de esta propuesta (T_c) [5, 6, 47] con los del algoritmo de Sutherland y Hodgman (T_{SH}) [129]. En este algoritmo el recorte se realiza progresivamente, analizando la situación del polígono respecto a cada cara del subespacio y recortándolo si es necesario. Para la comparación se ha utilizando la siguiente medida de eficiencia:

$$\text{Eficiencia} = \frac{T_{SH} - T_c}{T_{SH}} \tag{2.12}$$

En la Figura 2.16 se muestran los valores de eficiencia obtenidos en el *SCI Clúster* para distintos números de subespacios (Nd) y para la escena *armadillo*. Se puede

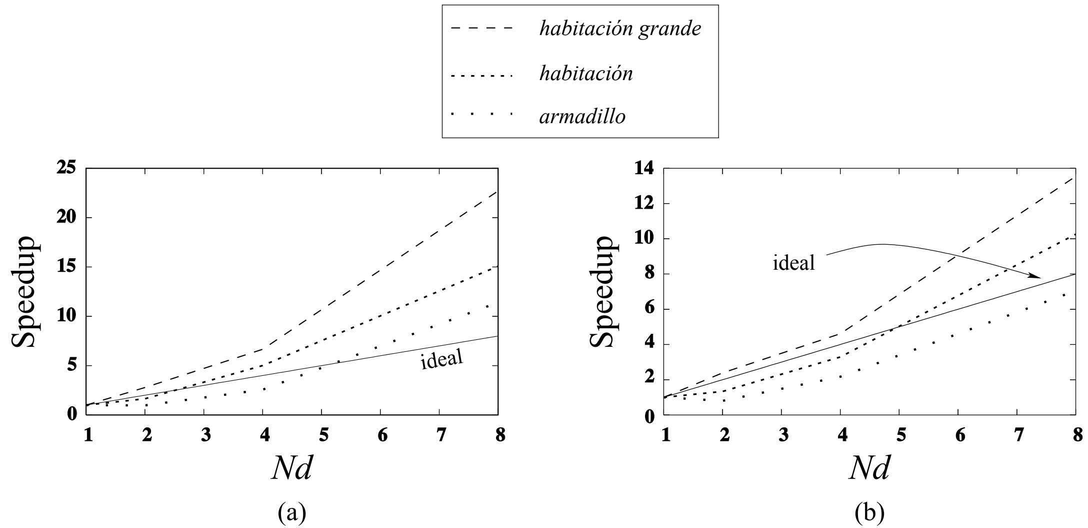


Figura 2.17: Resultados de las escenas de prueba: (a) *F. Ethernet Clúster* (b) *SCI Clúster*.

observar que hay un incremento de la eficiencia de más del 20 % para algunas configuraciones. Es de destacar que la eficiencia mejora cuando aumenta el número de subespacios, ya que se incrementa también el número de intersecciones de arista. Esto es debido a que el algoritmo de Sutherland y Hodgman recorta los polígonos contra las fronteras del subespacio y en cada iteración se podrían añadir vértices que deben ser testeados y que en algunos casos serán descartados. Sin embargo, el algoritmo de recorte usado solo calcula los vértices de recorte debido a la utilización de las intersecciones de arista. También hay que comentar que para las configuraciones de 3 y 10 procesadores el rendimiento del algoritmo analizado decrece. Esto es debido a que el número de polígonos recortados que son cruzados por las aristas de los subespacios disminuye. En general, podemos concluir que estos resultados demuestran que el algoritmo de recorte que usamos permite obtener mejores resultados en cuanto al tiempo de ejecución que el algoritmo estándar de recorte, beneficiando al rendimiento de la implementación paralela en su conjunto.

2.5.2. Rendimiento de la implementación paralela

Para evaluar el algoritmo paralelo, en la Figura 2.17 se presentan los resultados en términos de *speedup* para ambos *clústeres*. Todos los resultados han sido medidos respecto al algoritmo secuencial sin la división de la escena y muestran la baja sobrecarga asociada a nuestro método de partición, ya que el aumento del número de

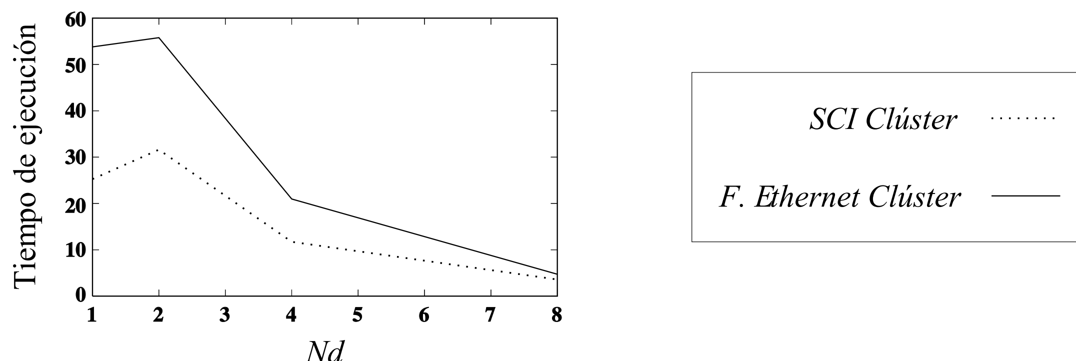


Figura 2.18: Tiempo de ejecución para la escena *armadillo*.

subespacios no implica una disminución del rendimiento, sino todo lo contrario. Los tiempos de ejecución para el algoritmo secuencial en el *F. Ethernet Clúster* y el *SCI Clúster* para la escena *habitación grande* fueron 344,44 segundos y 142,88 segundos, respectivamente. Los tiempos correspondientes para la ejecución paralela en 8 procesadores fueron 15,17 y 10,51 segundos, correspondiendo a unos valores de *speedup* de 22,71 y 13,59. Como se puede ver, los resultados muestran que se obtiene un *speedup* superlineal para estas escenas. Esto es debido principalmente a dos razones: la división de la escena conduce a una gran mejora en la localidad de los datos, que implica una reducción en los fallos de la caché y una mejora en la explotación de la jerarquía de la memoria en cada procesador; y el uso de las máscaras de visibilidad introduce unas aproximaciones en el cálculo de la visibilidad, produciendo una gran reducción en la complejidad de la determinación de la visibilidad con poca pérdida en la calidad del resultado.

En la Figura 2.18 se muestra el tiempo de ejecución del algoritmo para la escena *armadillo*. En esta gráfica se observa que el tiempo de ejecución paralela para dos procesadores se incrementa respecto al tiempo de ejecución secuencial (en la figura corresponde a $Nd = 1$). La razón de este comportamiento, correspondiente a escenas con pocos triángulos cuando se utilizan pocos procesadores, es la sobrecarga de trabajo asociada a nuestro método: la división de la escena y el cálculo de las máscaras de visibilidad. A partir de cuatro procesadores esta sobrecarga se compensa con las ventajas de la paralelización. También se debe indicar que el *speedup* que se obtiene en el *F. Ethernet Clúster* es más alto que en el *SCI Clúster* porque la reducción de los cálculos por la utilización de las máscaras de visibilidad tiene un mayor impacto en el *clúster* con un procesador más lento.

En resumen, podemos decir que nuestro método ofrece un buen rendimiento en

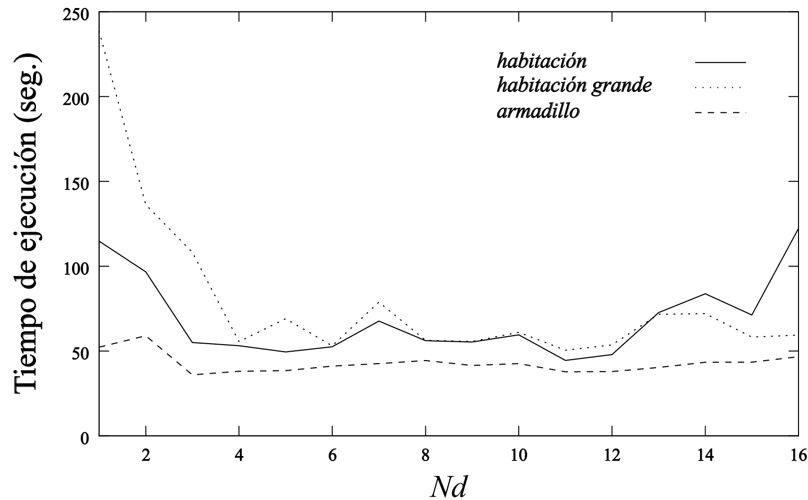


Figura 2.19: Tiempo de ejecución para *habitación*, *armadillo* y *habitación grande*.

dos sistemas diferentes, incluso para escenas con diferentes grados de uniformidad.

2.5.3. Análisis de la partición de la escena como método para incrementar la localidad de los datos

En este apartado se analizarán las mejoras de rendimiento que proporciona la estrategia de división de la escena en términos de tiempo de ejecución y de reducción de los fallos caché. También se ha verificado que la calidad de las imágenes resultantes no se ha visto afectada por el uso de esta técnica.

Para evaluar la propuesta se ha usado un procesador Athlon 64 a 3,2 GHz con una memoria cache L1 de 64+64 KB, una memoria caché L2 de 512 KB y una memoria RAM de 1 GB, y se ha aplicado el algoritmo modificado a las mismas escenas que en la implementación paralela: *habitación*, *habitación grande* y *armadillo*.

En primer lugar en la Figura 2.19 se presentan los resultados en términos de tiempo de ejecución, variando el número de particiones, Nd. No se incluyen datos para $Nd > 16$ porque en todos los casos se produce un incremento en los tiempos de ejecución. Se observa que cuando se utilizan valores bajos de Nd se reduce el tiempo requerido, pero aumenta para valores altos. Por ejemplo, para la escena *habitación grande* el tiempo de ejecución original es 238,63 segundos, mientras que para la versión con partición con $Nd = 4$ este tiempo disminuye a 55,54. Por consiguiente,

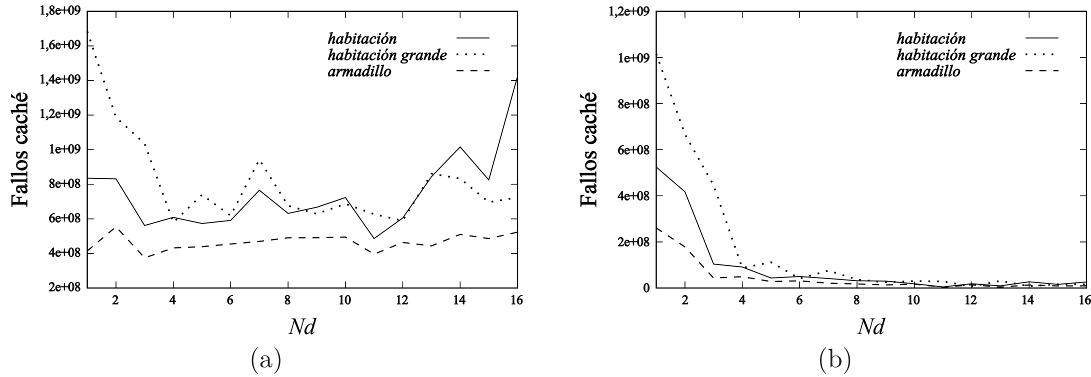


Figura 2.20: Fallos caché: (a) primer nivel; (b) segundo nivel.

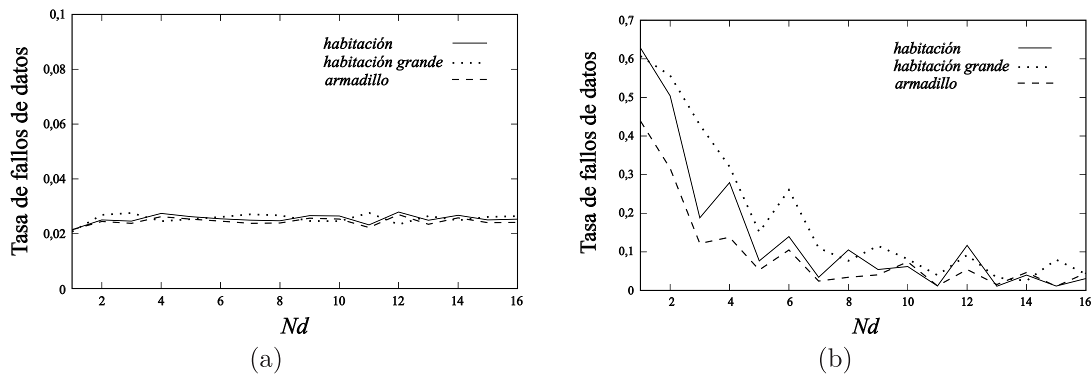


Figura 2.21: Tasa de fallos de datos: (a) primer nivel; (b) segundo nivel.

se obtiene un *speedup* de 4,30 para este ejemplo. Esta importante reducción en los tiempos de ejecución es debido principalmente a dos motivos: primero, la reducción de los fallos caché por la explotación de la localidad de los datos asociada con la estrategia de partición de la escena, y, en segundo lugar, la técnica de las máscaras de visibilidad, necesaria para la viabilidad de la aplicación del método de división, también produce una reducción del tiempo de computación, porque implica unas aproximaciones en la determinación de la visibilidad y, como consecuencia, una reducción en la complejidad computacional.

La medición de los fallos caché se puede realizar principalmente de dos formas: o por simulación o mediante los contadores hardware [4]. En este trabajo se ha elegido esta última, ya que puede dar con bastante precisión un amplio rango de informaciones de la ejecución de un programa. Específicamente, se ha usado la biblioteca PAPI (*Performance Application Programming Interface*) [78], que es una interfaz a los contadores hardware que permite mantener un código común para una amplia

variedad de arquitecturas. La Figura 2.20 muestra el número de fallos caché para el sistema Athlon para nuestras escenas, para el primer nivel (ver Figura 2.20a) y para el segundo nivel (ver Figura 2.20b). Por razones de claridad también mostramos las tasas de fallos de datos: para la caché de primer nivel en la Figura 2.21a y para la de segundo nivel en la Figura 2.21b.

Como se puede observar en la Figura 2.20b y la Figura 2.21b, la caché de segundo nivel puede contener a las subescenas incluso para valores pequeños de Nd. Esto se puede deducir de las figuras, ya que incrementando el valor de Nd se reduce el número de fallos caché y la tasa de fallos. Así, subescenas más pequeñas caben en la memoria y los datos pueden ser reutilizados, reduciendo el tráfico de datos entre los niveles de la jerarquía de memoria. También se debe señalar que, como muestran las figuras, aunque sigamos aumentando el valor de Nd los fallos caché no se eliminan, quedando los fallos de tipo forzoso o de arranque en frío, es decir, fallos asociados con la carga inicial de la subescena cada vez que una de ellas tiene que ser procesada.

Respecto a la caché de primer nivel, la Figura 2.21a indica que la tasa de fallos es prácticamente constante, mientras que la Figura 2.20a muestra que el número de fallos caché disminuye para valores intermedios de Nd. Esto significa que hay un descenso en el número de referencias a memoria para estos valores de Nd. Esta reducción de operaciones de carga y almacenamiento en el programa es debida a dos razones: la utilización de bloques de datos ayuda al uso de registros y el empleo de las máscaras de visibilidad minimiza los accesos en BUCLE2.

El uso de las máscaras de visibilidad tiene una influencia importante en el comportamiento del algoritmo. Esto es debido al hecho de que cuando la máscara de visibilidad de un polígono sólo tiene ceros (línea 32 de la Figura 2.10), el polígono ya no es propagado al resto de subescenas. Esto contrasta con el algoritmo original, en el cual las interacciones de este polígono con los otros de la escena deben ser comprobadas, con el consecuente incremento de los fallos caché. Por otro lado, la utilización de las máscaras de visibilidad reduce el número de accesos para determinar la visibilidad entre dos polígonos. En nuestro caso son usadas técnicas direccionales que calculan una lista de potenciales candidatos a obstaculizar las interacciones entre dos polígonos. La idea es eliminar de la lista aquellos polígonos que no pueden ser alcanzados por ningún rayo trazado entre ambos polígonos. Entonces para determinar la visibilidad entre los polígonos no es necesario acceder a las listas de elementos de otros subespacios y esto permite una notable reducción de accesos. Al mismo tiempo, los requerimientos computacionales asociados con la técnica de las máscaras de visibilidad dependen del valor de Nd. Para valores bajos, la sobrecarga asociada con

la técnica de las máscaras de visibilidad es clara, principalmente para escenas con un número pequeño de elementos, como se puede observar, por ejemplo, en el tiempo de ejecución para dos particiones de la escena *armadillo* (Figura 2.19). Para valores grandes de Nd los requerimientos computacionales asociados con el gran número de polígonos que tienen que ser "copiados" en otras subescenas pueden ser prohibitivos. Esto es así porque un incremento en el número de subescenas implica un incremento en el número de polígonos que tienen que ser propagados a otras subescenas, y cada vez que un triángulo tiene que ser enviado a otra subescena se ha de calcular o actualizar su máscara de visibilidad.

Resumiendo, el tiempo de ejecución del algoritmo modificado es mínimo para un valor intermedio de Nd. La influencia del valor de Nd se puede deducir también de la Ecuación 2.10. Para grandes valores de Nd el último término de la ecuación es mayor, mientras que término de BUCLE3 es más pequeño ($\overline{M^l}$ es menor). En consecuencia, el valor óptimo de Nd depende de la escena, aunque en nuestras pruebas se ha encontrado que un valor de Nd entre 4 y 6 es un buen valor de partición de la escena. Escenas mayores deberían necesitar valores más grandes. Con estos valores óptimos de Nd se explota la localidad de los datos porque las subescenas pueden ser almacenadas completamente en el segundo nivel de la caché. No obstante, valores mayores de Nd implican un incremento de la complejidad computacional asociada con el cálculo masivo de máscaras de visibilidad y, como consecuencia, un incremento en el tiempo de procesamiento. Esto también se traduce en un mayor número de fallos caché en la caché de primer nivel, como se observa en la Figura 2.20a.

2.5.4. Calidad de la escena resultante

Finalmente y respecto a la calidad de la imagen final, no se han detectado diferencias apreciables entre las imágenes generadas con el algoritmo estándar y las obtenidas con el algoritmo modificado. Adicionalmente, para verificar la calidad de las imágenes resultantes se ha usado como medida el error residual medio, definido como:

$$\text{error residual medio} = \frac{\sum_{i=1}^{Ne} \left| B_i - E_i - \rho_i \sum_{j=1}^{Ne} B_j F_{ij} \right|}{\sum_{i=1}^{Ne} E_i} \tag{2.13}$$

Los resultados numéricos indican que la calidad de las imágenes obtenidas por

ambos métodos es muy próxima. Por ejemplo, el resultado para la escena *habitación grande* para el algoritmo original es 2,09e–04, mientras que para la versión con cuatro particiones este error es 2,40e–04. En ambos casos los errores residuales son muy pequeños.

En nuestro método las máscaras de visibilidad permiten la eliminación de cálculos y evitan comprobar las interacciones entre muchos pares de polígonos. Por ejemplo, y como ya se ha mencionado anteriormente, una máscara de visibilidad con todos ceros descarta la interacción del polígono asociado con los otros polígonos en las siguientes subescenas. Esta utilización de las máscaras implica ciertas simplificaciones en el cálculo de visibilidad, reduciendo el número de comprobaciones. No obstante, cuando el número de elementos en la frontera es de un valor suficiente, prácticamente no se realizan modificaciones en la naturaleza del algoritmo y, como consecuencia, no se deben esperar cambios en la calidad de la imagen resultante. Esto lo confirman los resultados obtenidos.

Capítulo 3

Radiosidad progresiva paralela: partición no uniforme

En el capítulo anterior se ha empleado una división uniforme de la escena para la implementación paralela del algoritmo de radiosidad progresiva. Esta división produce subescenas de igual tamaño y forma. Este método de división es rápido, pero su principal inconveniente es que suele producir un pobre balanceo de la carga computacional. Esto es debido a la naturaleza del proceso de partición de la escena, ya que la división se realiza en función de consideraciones geométricas sin tener en cuenta el contenido de la escena.

Para lograr balancear la carga computacional en sistemas de memoria distribuida, en este capítulo se propone un nuevo método de partición no uniforme de la escena para su distribución en los procesadores. Este método está basado en la minimización de una función de distribución. Además, para poder calcular la visibilidad entre los objetos de diferentes procesadores hemos desarrollado una modificación de la técnica de las *máscaras de visibilidad* [7]. En el capítulo anterior, para división uniforme, las caras que separan dos subescenas vecinas eran iguales en forma y tamaño. Sin embargo, en una división no uniforme estas caras pueden ser de diferentes tamaños, no coincidir, o ser adyacentes a varias caras de otros subespacios. Debido a esto y para poder usar las máscaras de visibilidad se ha diseñado una planificación de las comunicaciones de las máscaras de acuerdo a una jerarquía de los procesadores. Esta jerarquía va a permitir reconstruir las máscaras de visibilidad a partir de máscaras parciales.

Nuestra propuesta consume poco tiempo en la división de la escena y evita la

replicación de ella en cada procesador. El método ha sido aplicado a un implementación paralela del algoritmo de radiosidad progresiva, alcanzando valores elevados de *speedup* en un *clúster* de PC's. Esto es debido a la explotación de la localidad de los datos y al buen balanceo de la carga computacional de nuestro método. Tanto una descripción de la metodología desarrollada como los resultados experimentales han sido publicados en [104].

En el resto del capítulo, en primer lugar, se presenta el método de división no uniforme que hemos desarrollado. A continuación, en la Sección 3.2 se explica la implementación paralela del algoritmo de radiosidad progresiva utilizando la división no uniforme. El capítulo finaliza mostrando los resultados experimentales obtenidos con esta implementación.

3.1. Partición no uniforme

En esta sección se va a presentar el método de división no uniforme que hemos desarrollado para dividir la escena. El método propuesto está basado en la minimización de una función de distribución asociada con la operación de partición. Por consiguiente, el objetivo es escoger una operación de partición que minimice esa función de distribución. A continuación, en primer lugar se harán algunas consideraciones iniciales para, después, explicar como se realiza la división de la escena.

En lo sucesivo se considerará que se tienen Nd procesadores y que la escena se va a dividir en Nd subescenas, cada una de las cuales se asignará a un procesador. Debido a la metodología utilizada en la partición de la escena, se asumirá que Nd es una potencia de dos. Sea M el conjunto de polígonos de las escena total y $D_{Nd}(M)$ la operación de partición que divide la escena en Nd partes $\{M^1, \ldots, M^{Nd}\}$. Se puede ilustrar esta operación de la siguiente manera:

$$M \xrightarrow{D_{Nd}} \begin{bmatrix} M^1 \\ \vdots \\ M^{Nd} \end{bmatrix} \tag{3.1}$$

Para dirigir la operación de partición hay que definir una función de distribución que describa el desbalanceo de la carga en términos de un factor X. Este factor es una propiedad específica de cada elemento de la escena que se puede emplear para la caracterización de las particiones resultantes. De esta forma, la función de

distribución se define como:

$$\sigma(D_{X,Nd}) = \frac{1}{\bar{X}}\sqrt{\sum_{d=1}^{Nd}(\bar{X} - X^d)^2/Nd}\ , \tag{3.2}$$

donde $D_{X,Nd}$ es la división de la escena en Nd subescenas y X es el factor empleado para guiar esta partición. $\bar{X}$ es el valor promedio del factor X para la escena y se define como:

$$\bar{X} = \frac{1}{Nd}\sum_{i=1}^{Ne} X_i, \tag{3.3}$$

donde X_i es el factor X del elemento S_i. Adicionalmente, para cada subescena M^d se puede definir:

$$X^d = \sum_{i=1}^{Ne^d} X_i, \quad S_i \in M^d, \tag{3.4}$$

siendo Ne^d el número de polígonos de las subescena M^d.

Para obtener un buen balanceo de la carga es necesario calcular la partición $D_{X,Nd}$ óptima que minimice esta función de distribución. Pero esto es un problema NP–completo. Por consiguiente, para resolverlo se ha optado por usar una estrategia *divide–y–vencerás*. Esta estrategia consiste en dividir un problema en varias operaciones más sencillas, solucionando cada una de estas operaciones básicas independientemente y combinando los resultados obtenidos para construir la solución del problema original. En nuestra propuesta se ha trabajado con el problema más simple de buscar la partición óptima de una escena en dos subespacios $D_{X,2}$. Por tanto, el objetivo es encontrar la división en dos subescenas $D_{X,2}$ que minimice la función $\sigma(D_{X,2})$:

$$\sigma(D_{X,2}) = \frac{|X^1 - X^2|}{X^1 + X^2} \tag{3.5}$$

donde X^1 y X^2 son los factores de las dos subescenas candidatas y $\sigma(D_{X,2}) = 0$ implica que la partición realizada tiene un balanceo de carga perfecto ($X^1 = X^2$). En la práctica, no se llega a obtener $\sigma(D_{X,2}) = 0$, sino que se establece un umbral *thr*, de manera que cuando $\sigma(D_{X,2}) < thr$ finaliza el proceso de minimización. Por tanto, la búsqueda de una operación óptima de partición $D_{X,Nd}$ se puede descomponer en

$\log_2(Nd)$ etapas de partición simplificadas, donde en cada una de ellas se aplica $D_{X,2}$ para dividir la escena o subescena en dos partes. Por ejemplo, el esquema de dividir M en 8 partes es el siguiente:

$$M \xrightarrow{D_{X,2}} \left[\begin{array}{l} M^0 \xrightarrow{D_{X,2}} \left[\begin{array}{l} M^{00} \xrightarrow{D_{X,2}} \left[\begin{array}{l} M^{000} \\ M^{001} \end{array} \right. \\ M^{01} \xrightarrow{D_{X,2}} \left[\begin{array}{l} M^{010} \\ M^{011} \end{array} \right. \end{array} \right. \\ M^1 \xrightarrow{D_{X,2}} \left[\begin{array}{l} M^{10} \xrightarrow{D_{X,2}} \left[\begin{array}{l} M^{100} \\ M^{101} \end{array} \right. \\ M^{11} \xrightarrow{D_{X,2}} \left[\begin{array}{l} M^{110} \\ M^{111} \end{array} \right. \end{array} \right. \end{array} \right.$$

Debe ser indicado que el índice i que etiqueta a cada partición M^i está representado por un número binario porque va a permitir simplificar el proceso de distribución de las subescenas entre los procesadores.

En nuestra propuesta, inicialmente para cada operación de división simplificada $D_{X,2}$ se elige la dirección del plano de partición. Para ello se calculan tres particiones diferentes asociadas a cada una de las tres coordenadas espaciales $(D^x_{X,2}, D^y_{X,2}, D^z_{X,2})$ colocando el plano de partición en la mitad de la caja que delimita el espacio mínimo que contiene a todos los elementos de la escena que está siendo dividida. A continuación, la partición que tenga el valor menor de $\sigma(D_{X,2})$ es seleccionada $(D_{X,2} = \min_{\sigma}(D^x_{X,2}, D^y_{X,2}, D^z_{X,2}))$. Una vez escogida la dirección del plano de partición, hay que calcular la posición óptima de este plano. Específicamente, se analizan $2^C - 1$ posiciones uniformemente distribuidas a lo largo de la coordenada seleccionada. Esto significa que el plano de división puede estar situado en los puntos $\{1/2^C, 2/2^C, \cdots, (2^C-1)/2^C\}$. Como se verá a continuación, el valor de C se determina dinámicamente con el menor número de iteraciones necesarias para alcanzar el umbral *thr* en el valor de $\sigma(D_{X,2})$.

La búsqueda la posición óptima se realiza utilizando un árbol de búsqueda binaria. Esto permite la optimización del proceso de búsqueda, no necesitando analizar todas las posibles posiciones candidatas. Como ejemplo la Figura 3.1 muestra un árbol de búsqueda binaria para $C = 4$. Cada nodo representa un plano de divi-

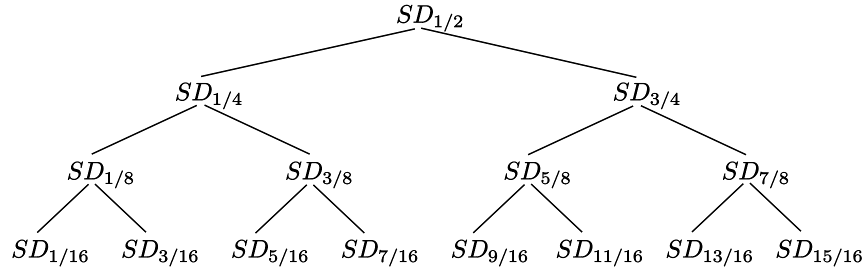


Figura 3.1: Árbol de búsqueda binaria empleado para la determinación de la posición de división con $C = 4$.

sión, donde el nodo raíz (etiquetado por el subíndice 1/2) es el plano que divide la escena en dos partes iguales. Cada nodo tiene dos hijos que dividen el espacio correspondiente en dos partes iguales:

$$\begin{array}{ccl} & & SD_{((i-1)2+1)/(2j)} \\ & \nearrow & \\ SD_{i/j} & & \\ & \searrow & \\ & & SD_{((i-1)2+3)/(2j)} \end{array} \tag{3.6}$$

El árbol resultante se emplea para identificar la mejor posición del plano de división. Primero se analiza el nodo raíz. Si el valor de $\sigma(D_{X,2})$ es superior que el umbral seleccionado *thr* el proceso continúa con los nodos hijos. Concretamente, asumiendo que la subescena^1 es la subescena de coordenadas inferiores, si $X^1 > X^2$ se analiza el nodo izquierdo y si no el nodo derecho. Los nodos de la búsqueda binaria se calculan en el tiempo de ejecución y el proceso finaliza cuando $\sigma(D_{X,2})$ es menor que el valor límite que se ha establecido, *thr*. La Figura 3.2 ilustra este método en dos dimensiones y para la coordenada x. Primero, $SD^x_{1/2}$ divide el espacio en dos mitades. Como $X^1 > X^2$ se elige el nodo hijo $SD^x_{1/4}$. El proceso continúa seleccionando $SD^x_{3/8}$ y finalmente $SD^x_{7/16}$. Por tanto, el árbol de búsqueda binaria tiene una profundidad de $C = 4$.

Para clarificar el proceso de división de la escena se va a considerar el ejemplo de la Figura 3.3, donde se divide una escena en 8 subescenas. En la primera columna de la izquierda se muestra la división ideal de la escena etiquetada con D_8. El mismo resultado se obtiene empleando el operador D_2 como se observa en la figura. Específicamente, se ha dibujado cada paso del proceso de división en columnas consecutivas. En cada caso, se marca la posición del plano de partición que minimiza

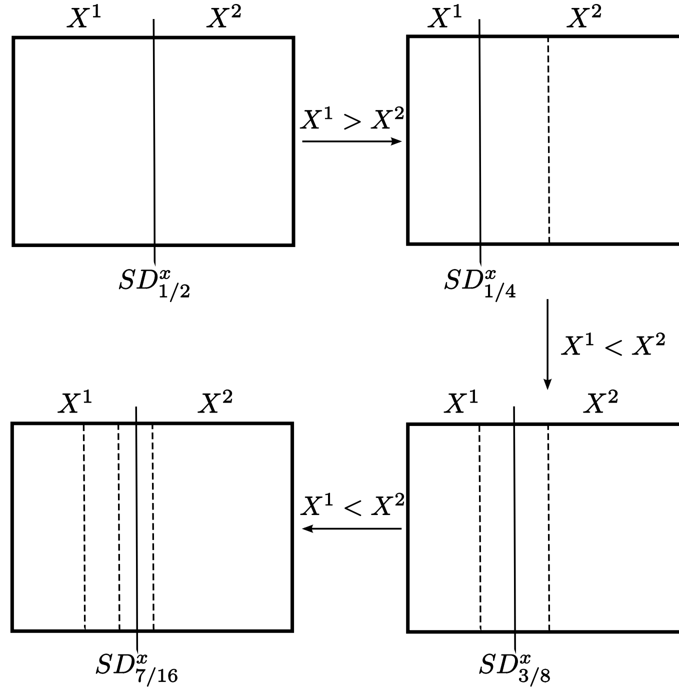


Figura 3.2: Búsqueda de la posición del plano de división (coordenada x).

la Ecuación 3.5 con una línea de trazos.

Hay diferentes opciones para seleccionar el factor X que guía el proceso de partición (área, potencia sin disparar, ...). En nuestro caso se ha escogido como función una que describe el desbalanceo de la carga en términos del área distribuida entre las particiones, ya que se ha considerado que el área es una buena medida de la carga computacional. La razón de esto es que la complejidad del algoritmo de radiosidad progresiva es proporcional al número de polígonos y este número depende fundamentalmente del área de la escena. Esto es debido a que en este algoritmo todos los elementos de la escena son divididos hasta que su área alcance un valor igual o inferior a un umbral dado. En este caso, llamando A_i al área del polígono S_i, la función de distribución sería:

$$\sigma(D_{A,Nd}) = \frac{1}{\bar{A}} \sqrt{\sum_{d=1}^{Nd} (\bar{A} - A^d)^2 / Nd} \ , \tag{3.7}$$

donde

$$\bar{A} = \frac{1}{Nd} \sum_{i=1}^{Ne} A_i \ , \tag{3.8}$$

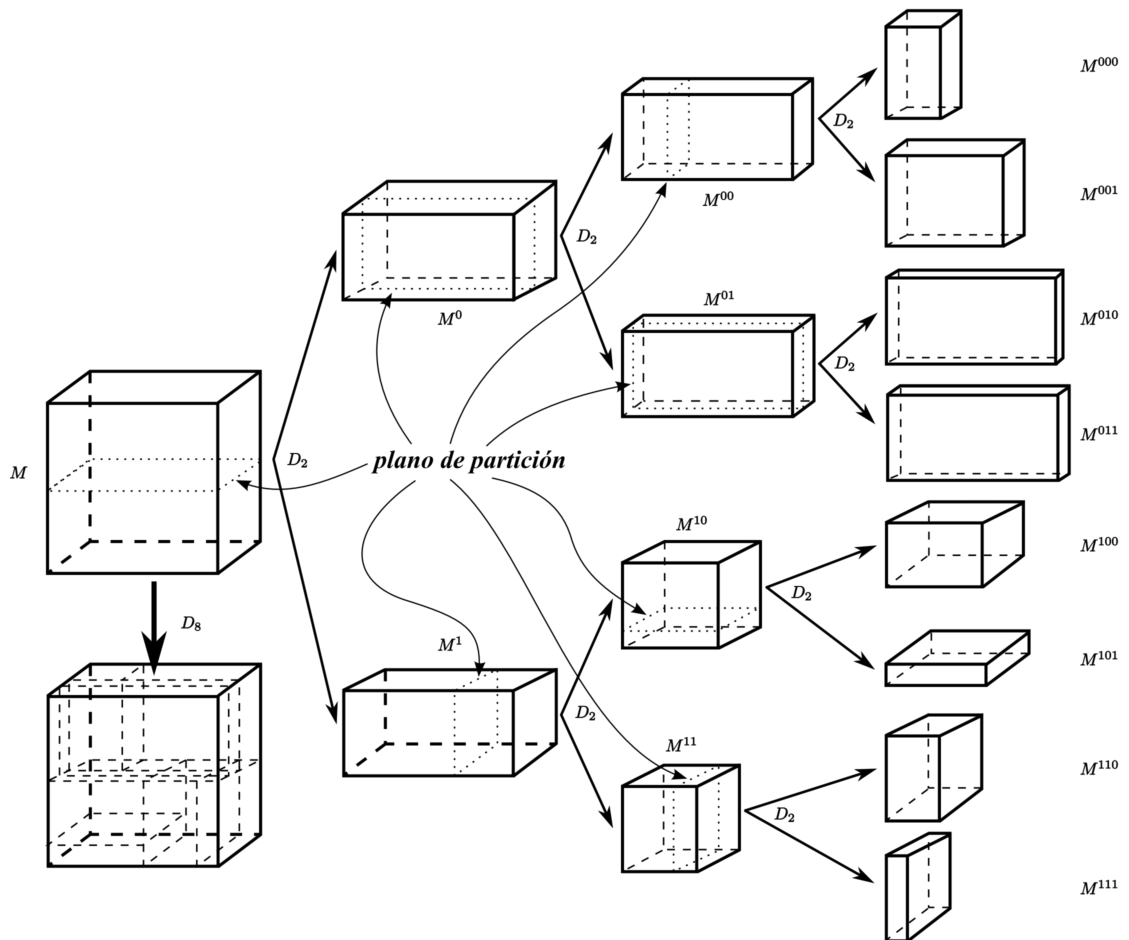


Figura 3.3: Ejemplo del método de división no uniforme para 8 particiones.

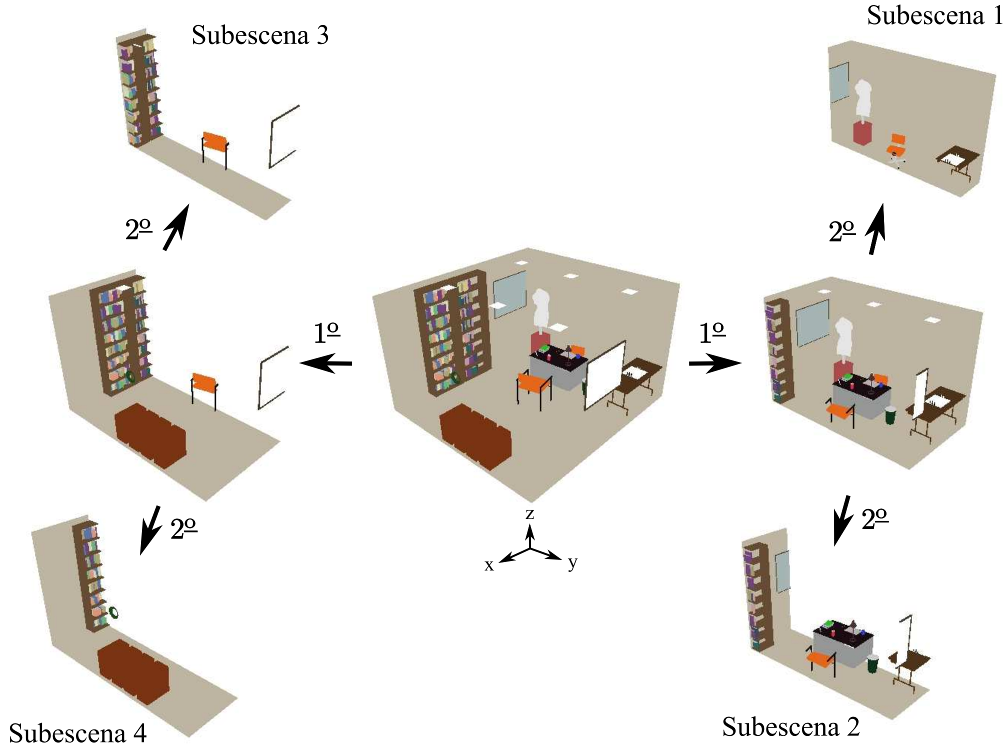


Figura 3.4: División no uniforme de una escena en cuatro subescenas.

y

$$A^d = \sum_{i=1}^{Ne^d} A_i \ , \quad S_i \in M^d \tag{3.9}$$

En este caso para la operación de división simplificada la función de distribución para dos subescenas es la siguiente:

$$\sigma(D_{A,2}) = \frac{|A^1 - A^2|}{A^1 + A^2} \tag{3.10}$$

En la Figura 3.4 se muestra la división de una escena en cuatro subescenas empleando el área de los polígonos como guía del proceso de partición no uniforme. En el primer paso la escena es dividida en dos partes con el valor más pequeño de $\sigma(D_{A,2})$, que en este caso corresponde a $D^x_{A,2}$. En el segundo paso cada partición divide las subescenas de nuevo en dos partes siendo otra vez $D^x_{A,2}$ la mejor opción para ambas subescenas. Se puede observar que se aprecia una mejor distribución de la carga (número de elementos) en esta figura comparada con la obtenida en el

caso de división uniforme. Por ejemplo, en la división uniforme el procesador 3 tiene menos elementos que los otros procesadores (ver Figura 2.4), situación que no ocurre en el ejemplo presentado de la división no uniforme.

3.2. Paralelización del algoritmo

El esquema de la implementación paralela realizada para el algoritmo con división no uniforme es semejante al utilizado para la división uniforme (ver algoritmo de la Figura 2.10), pero se han tenido que realizar importantes modificaciones en cuanto al cálculo de la visibilidad y la propagación del disparo de elementos a lo largo de la escena distribuida. En este caso también se utilizan las *máscaras de visibilidad* para el cálculo de la visibilidad entre polígonos de distintos subespacios, pero con una particularidad: las fronteras entre dos subescenas vecinas no tienen por qué coincidir en tamaño. En el caso de división uniforme dos subescenas vecinas comparten una cara, denominada frontera, que es de igual tamaño para los dos subespacios. Estas fronteras son definidas como *idénticas*. Sin embargo, en la partición no uniforme las únicas particiones con fronteras idénticas son aquellas cuyos índices binarios difieren exclusivamente en el último bit de la derecha. Como un ejemplo considérese el caso de la Figura 3.3. En esta figura la subescena M^{000} tiene una frontera idéntica con M^{001}, pero no con M^{010}. Como consecuencia, la información de visibilidad de un polígono de M^{000} que tenga que ser enviada a M^{010} también necesita la información de visibilidad de M^{001}. Combinando las caras externas del conjunto M^{000} y M^{001}, es decir considerando las caras externas de M^{00}, se puede construir una frontera idéntica con el grupo M^{01}.

Para poder llevar a cabo la implementación paralela de la radiosidad progresiva con división no uniforme de la subescena, en primer lugar se ha tenido que diseñar un nuevo método para calcular las máscaras de visibilidad asociadas a cada grupo, partiendo de definir una jerarquía de procesadores. Nuestra propuesta está basada en la estructura de la división binaria del algoritmo. Vamos a llamar *procesadores hermanos* a los que tienen asignadas particiones con fronteras idénticas, y *procesador padre* al que recoge el grupo de dos procesadores hermanos. Con esto cada frontera es asignada a un procesador. Por ejemplo, en la Figura 3.3, las particiones de dos procesadores hermanos, etiquetadas como M^{00} y M^{01}, son agrupadas en M^{0} que se asigna al procesador padre.

En la Figura 3.5 se muestra la jerarquía de procesadores. Los grupos finales se

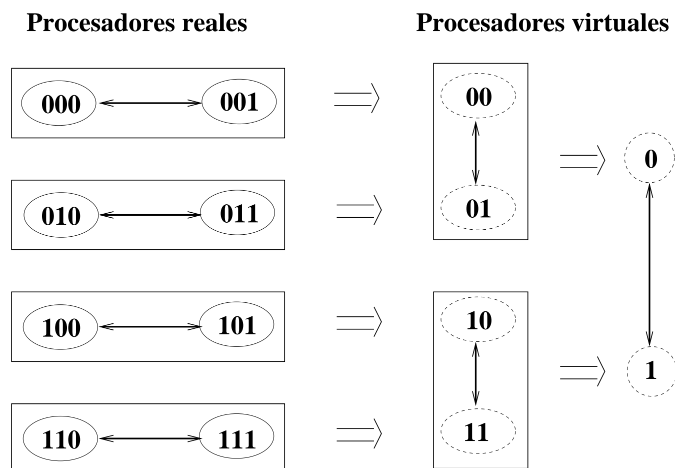


Figura 3.5: Jerarquía de procesadores.

asignan a procesadores reales, dibujados en la figura rodeados por una elipse de trazo completo. El resto de grupos son asignados a procesadores virtuales que tiene una frontera asociada, indicados en la figura mediante una elipse o un círculo con línea a trazos. Un procesador virtual con índice i es asignado a un procesador real con índice j, donde j es el número binario resultante de extender i con ceros por la derecha. Por ejemplo, en la Figura 3.5, el procesador virtual 0 es asignado al procesador real 000 y el procesador virtual 1 es asignado al procesador real 100.

En segundo lugar, una vez que las máscaras iniciales son calculadas, se ha tenido que desarrollar un nuevo método de propagación de la información entre los procesadores. Para realizar esta propagación, cada procesador envía las máscaras junto con una copia del polígono al procesador asociado a cada frontera. Si la frontera del grupo asignado al procesador virtual es idéntica a la frontera del grupo asignado al procesador real, entonces el polígono es almacenado en la cola de polígonos con potencia sin disparar de este procesador real. En otro caso, la información es enviada por el procesador real para que la distribuya entre los procesadores hermanos e hijos que le correspondan. Además, estos procesadores tienen que enviar el polígono a los procesadores virtuales con fronteras idénticas. El proceso continúa hasta que la información de la frontera llegue al procesador real final correspondiente. Una vez que la información de la máscara de visibilidad de entrada se completa, se actualiza la información de visibilidad trazando rayos desde el polígono recibido a través de las máscaras de visibilidad de entrada y hacia las fronteras de salida.

Como ejemplo considérese el cálculo de las máscaras de visibilidad para el triángulo S_1 de la Figura 3.6. Después de disparar la radiosidad del triángulo en su

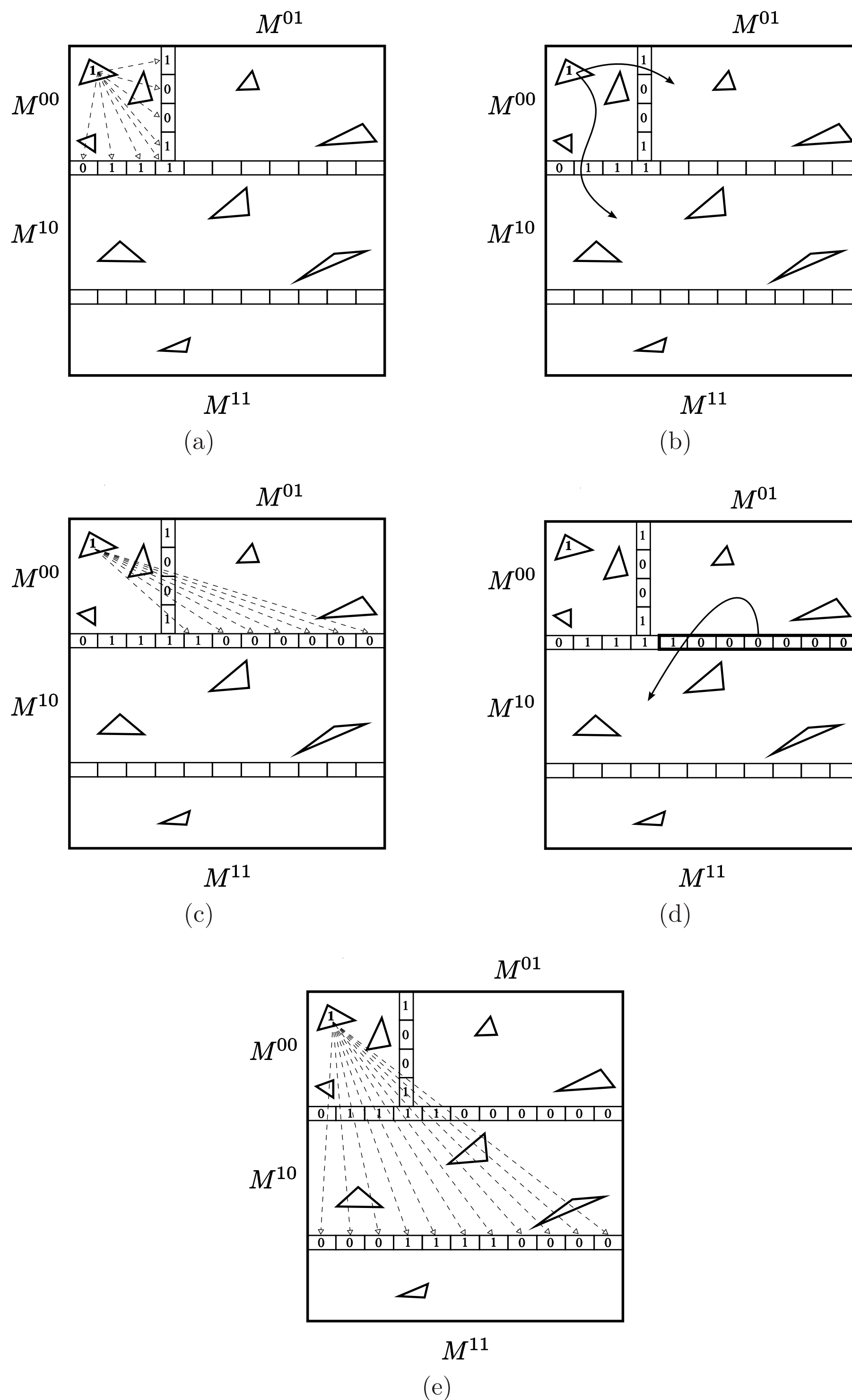
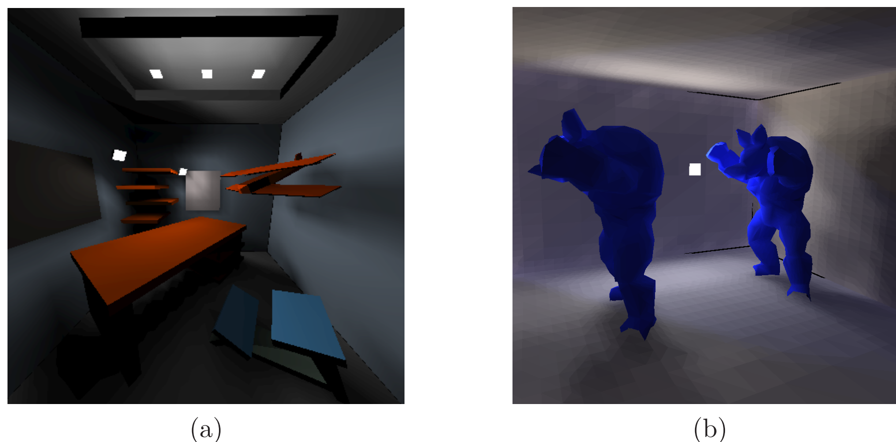


Figura 3.6: Determinación de las máscaras de visibilidad de un triángulo para cuatro procesadores con la partición no uniforme.



(a) (b)

Figura 3.7: Escenas iluminadas: (a) *habitación* y (b) *armadillos*.

subescena se calculan sus máscaras de visibilidad iniciales (ver la Figura 3.6a). En la Figura 3.6b el triángulo y las máscaras de visibilidad son enviadas al procesador M^{01} y al procesador padre de M^{10}, que en este caso es el mismo procesador. A continuación, el procesador hermano, M^{01}, calcula la nueva máscara de visibilidad usando la máscara recibida (ver la Figura 3.6c). Este procesador envía esta máscara de visibilidad al procesador padre correspondiente, pudiendo ahora este procesador completar el cálculo de la visibilidad uniendo esta máscara con la recibida previamente (ver la Figura 3.6d). Finalmente, antes de enviar la información de visibilidad al procesador hermano, el procesador M^{10} tiene que calcular la nueva máscara de visibilidad (ver la Figura 3.6e).

3.3. Resultados experimentales

En esta sección se van a presentar los resultados de rendimiento y un análisis del método de partición no uniforme gestionando el cálculo de la visibilidad mediante la jerarquía de procesadores. Para la implementación paralela de nuevo se ha utilizado la metodología de paso de mensajes empleando la biblioteca de funciones MPI. En este caso las pruebas se han realizado sobre el *SCI Clúster*, ya mencionado en las Sección 2.5. Como ya se explicó, este *clúster* está formado por procesadores Intel XEON IV a 1.8 GHz conectados a través de la red SCI de 250 Mbps.

Los resultados que se van a mostrar son los obtenidos con la escena *habitación* (ver Figura 3.7a) y con otra escena construida a partir de la escena *armadillo* en la

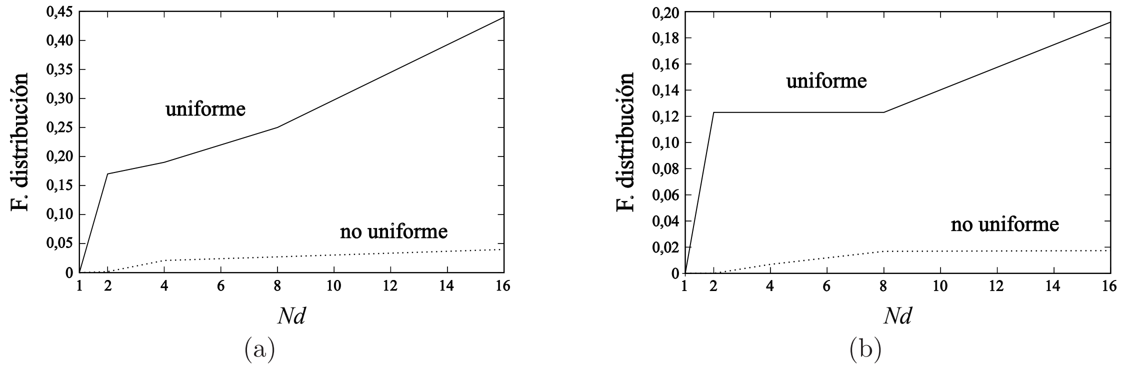


Figura 3.8: Función de distribución para los métodos uniforme y non uniforme: (a) *habitación* y (b) *armadillos*.

que se ha replicado el modelo del armadillo. Esta escena será denominada como *armadillos* (ver Figura 3.7b). Se ha querido emplear esta escena para tener un ejemplo de una escena poco regular y con un número mayor de polígonos.

En primer lugar se comparará el método de división no uniforme frente al uniforme presentado en el capítulo anterior (ver Figura 3.8). El balanceo de la carga computacional fue chequeado experimentalmente midiendo la función de distribución (ver la Ecuación 3.2) para cada método. Como se puede observar, los valores de σ son mejores usando la partición no uniforme que usando la uniforme y la diferencia entre ambos métodos se incrementa cuando aumentamos el número de procesadores. Más concretamente, el desbalanceo de la carga del algoritmo uniforme es 0,440 y 0,192, y del algoritmo no uniforme 0,040 y 0,017, para la escena *habitación* y la escena *armadillos*, respectivamente, utilizando 16 procesadores.

Aunque la escena *armadillos* contiene dos modelos de armadillo idénticos, solo se obtiene un balanceo perfecto para dos procesadores con el método no uniforme. Esto se debe a la localización de los dos armadillos en la escena. Específicamente, en la partición uniforme de la escena la aplicación no elige la mejor coordenada para colocar el plano de división, sino que escoge la primera dentro del orden que tenga implementado (en este caso la coordenada x). De esta manera, ambos armadillos se encuentran en la misma partición, mientras que la otra corresponde a una habitación vacía, con una carga computacional muy baja. Cuando se realiza la partición no uniforme, el algoritmo escoge la mejor dimensión para el plano de corte, resultando que cada armadillo es asignado a particiones distintas, lo que produce un sistema mucho mejor balanceado.

Por otra parte, en la Figura 3.9 se muestra el rendimiento en términos de *speedup*.

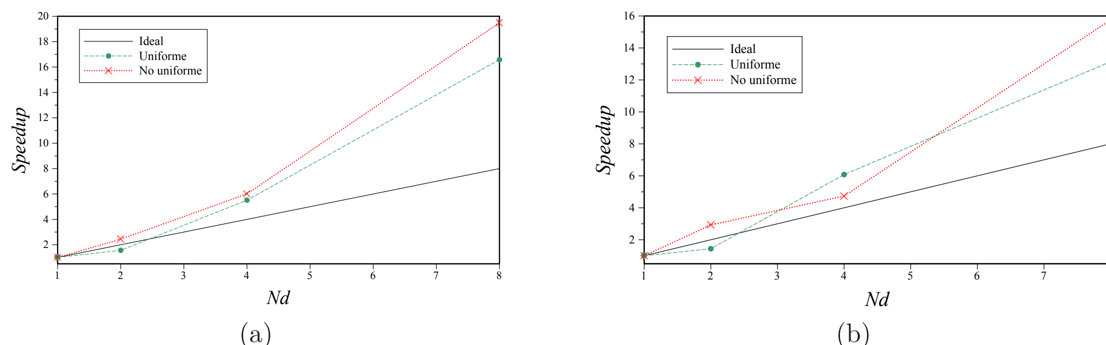


Figura 3.9: Resultados en términos de *speedup* de las escenas de prueba: (a) *habitación* y (b) *armadillos*.

Todos los resultados han sido medidos respecto al algoritmo secuencial estándar de radiosidad progresiva ejecutado en un procesador Intel XEON IV a 1.8 GHz, sin aplicar el método de partición. Estos resultados revelan una baja sobrecarga de trabajo asociada a nuestro método de partición. El tiempo de ejecución para el algoritmo secuencial para la escena *habitación* fue de 77,54 segundos, y para la escena *armadillos* fue de 23,49 segundos. Como ejemplo, los correspondientes valores de tiempo de ejecución en la versión paralela para 8 procesadores fueron, respectivamente, de 4,68 y 1,79 para el método de partición uniforme, y de 3,98 y 1,49 segundos para el método de partición no uniforme. Como puede observarse, se obtienen buenos valores de *speedup* para estas escenas, siendo mejores los del método de partición no uniforme que los de la partición uniforme en casi todos los casos. Por tanto, estos resultados prueban que la partición no uniforme empleada para paralelizar el algoritmo permite la obtención de un excelente rendimiento en sistemas de memoria distribuida.

Capítulo 4

Método distribuido de Monte Carlo para radiosidad

En el Capítulo 1 ya se comentó que en este trabajo se ha escogido, dentro de los métodos probabilísticos que utilizan Monte Carlo, un método de relajación estocástica basado en el método iterativo de Jacobi. Como ya se ha indicado, los métodos de relajación estocástica son tan eficientes como otros métodos probabilísticos tales como los métodos de caminos aleatorios discretos para radiosidad [112, 113] o los métodos de estimación de densidad de fotones [59, 70]. Sin embargo, las técnicas escogidas presentan algunas ventajas adicionales, como por ejemplo su facilidad y efectividad para la implementación de sistemas de reducción de varianza mediante variables de control. Por otro lado, el método iterativo incremental estocástico de Jacobi elegido destaca frente a otros métodos de relajación estocástica, que utilizan Gauss-Seidel o Southwell [117, 118, 119], por su simplicidad computacional y por su igual o superior eficiencia.

En los métodos de Monte Carlo para radiosidad (MCR), que pueden manejar escenas muy grandes y complejas, los requerimientos computacionales del algoritmo son altos y es conveniente buscar formas de optimizar el tiempo de ejecución. La propuesta que se presenta en este capítulo es una implementación paralela del algoritmo en sistemas de memoria distribuida, dividiendo la escena en particiones convexas. Se han empleado los dos métodos de división vistos en la paralelización del algoritmo de radiosidad progresiva: el uniforme, que divide la escena en particiones de igual tamaño siguiendo solo criterios geométricos, y el no uniforme que persigue una distribución homogénea de la carga de trabajo entre todos los procesadores.

Nuestra propuesta sigue una estrategia de grano grueso y usa un paradigma de paso de mensajes. Así, cada procesador realiza los cálculos de la radiosidad relacionada con su subescena, permitiendo trabajar con escenas de mayor tamaño. Además, la partición convexa de la escena permite la explotación de la localidad de los datos y la optimización del proceso de disparo de rayos debido a la minimización del número de objetos que tienen que ser chequeados en el cálculo de la intersección.

En la implementación paralela presentada, cada procesador calcula la radiosidad para su subescena según los rayos locales y los que proceden de las subescenas vecinas. Los rayos son enviados a las escenas vecinas si salen de la subescena local, dependiendo de por donde atraviesen las fronteras. Este proceso de comunicación de información entre los procesadores es crucial debido a la gran cantidad de rayos que se intercambian. En nuestro caso se ha propuesto una técnica basada en la generación y transmisión de paquetes de rayos que reduce los requerimientos de comunicación.

Debido a estar utilizando un método iterativo, en la implementación distribuida surge el problema de la comprobación del fin de cada iteración. La solución propuesta está basada en tres fases: la primera es un chequeo local, la segunda es una fase de comunicaciones y en la tercera se hace una comprobación de los datos locales y los recibidos. El método desarrollado evita la utilización de barreras entre los procesadores, permitiendo solapar este proceso de comprobación con cálculos útiles.

Con nuestra propuesta se han obtenido buenos resultados en términos de tiempo de ejecución incluso para modelos complejos. Adicionalmente, no se realizan aproximaciones por lo que se calcula exactamente la iluminación, obteniendo imágenes de alta calidad. Este trabajo ha sido publicado en [106].

En este capítulo inicialmente se hablará sobre el método iterativo incremental estocástico de Jacobi. A continuación, en la Sección 4.2 se describirá el método distribuido de Monte Carlo para radiosidad. Finaliza el capítulo con un análisis de los resultados experimentales obtenidos.

4.1. Método iterativo incremental estocástico de Jacobi

Como ya se ha comentado en el Capítulo 1, el método iterativo de Jacobi es un método para resolver sistemas de ecuaciones lineales que usa un esquema sencillo de iteración. Aplicando este método a la siguiente ecuación (ver Sección 1.4):

$$P_i = P_{ei} + \rho_i \sum_{j=1}^{Ne} P_j F_{ji} \tag{4.1}$$

se puede obtener en una iteración k una aproximación de la potencia total P_i^k del elemento S_i, como la suma de incrementos ΔP^l calculados en cada iteración l $(l = 0, \ldots, k)$:

$$\begin{aligned} \Delta P_i^{k+1} = & \ \rho_i \sum_{j=1}^{Ne} \Delta P_j^k \, F_{ji} \\ P_i^k = & \ \sum_{l=0}^{k} \Delta P_i^l \end{aligned} \tag{4.2}$$

donde $\Delta P_i^0 = P_{ei}$, siendo P_{ei} la potencia autoemitida por el elemento S_i. De una manera similar al algoritmo de radiosidad progresiva, en cada iteración el algoritmo incremental de Jacobi propaga la potencia sin disparar en lugar de la potencia total.

Las sumas de la Ecuación 4.2 se pueden estimar estocásticamente mediante el método de Monte Carlo escogiendo términos de esas sumas aleatoriamente de acuerdo a una cierta probabilidad. Promediando el valor de los términos escogidos de acuerdo con la probabilidad con que han sido elegidos se obtienen estimaciones no sesgadas de las sumas. Cuando se aplica esta formulación a la radiosidad, el proceso se corresponde a una simulación directa de las reflexiones de un único rebote de la luz según los caminos seguidos por fotones. En la Figura 4.1 se muestra un ejemplo de como actúa el algoritmo. En la primera fila, inicialmente la potencia autoemitida es propagada, resultando una primera aproximación de la iluminación directa. En las siguientes filas (de (1b) a (1d)) se muestran los rayos disparados en cada iteración y el resultado incremental del algoritmo.

Para la aplicación de Monte Carlo a las iteraciones del disparo de potencia incremental, por razones puramente técnicas, la suma de la Ecuación 4.2, $\sum_{j=1}^{Ne} \Delta P_j^k \, F_{ji}$, se reescribe como una doble suma introduciendo una función delta de Kronecker δ_{li} ($\delta_{li} = 1$ si $l = i$ y 0 si $l \neq i$):

$$\Delta P_i^{k+1} = \sum_{j=1}^{Ne} \sum_{l=1}^{Ne} \Delta P_j^k F_{jl} \, \rho_l \, \delta_{li} \tag{4.3}$$

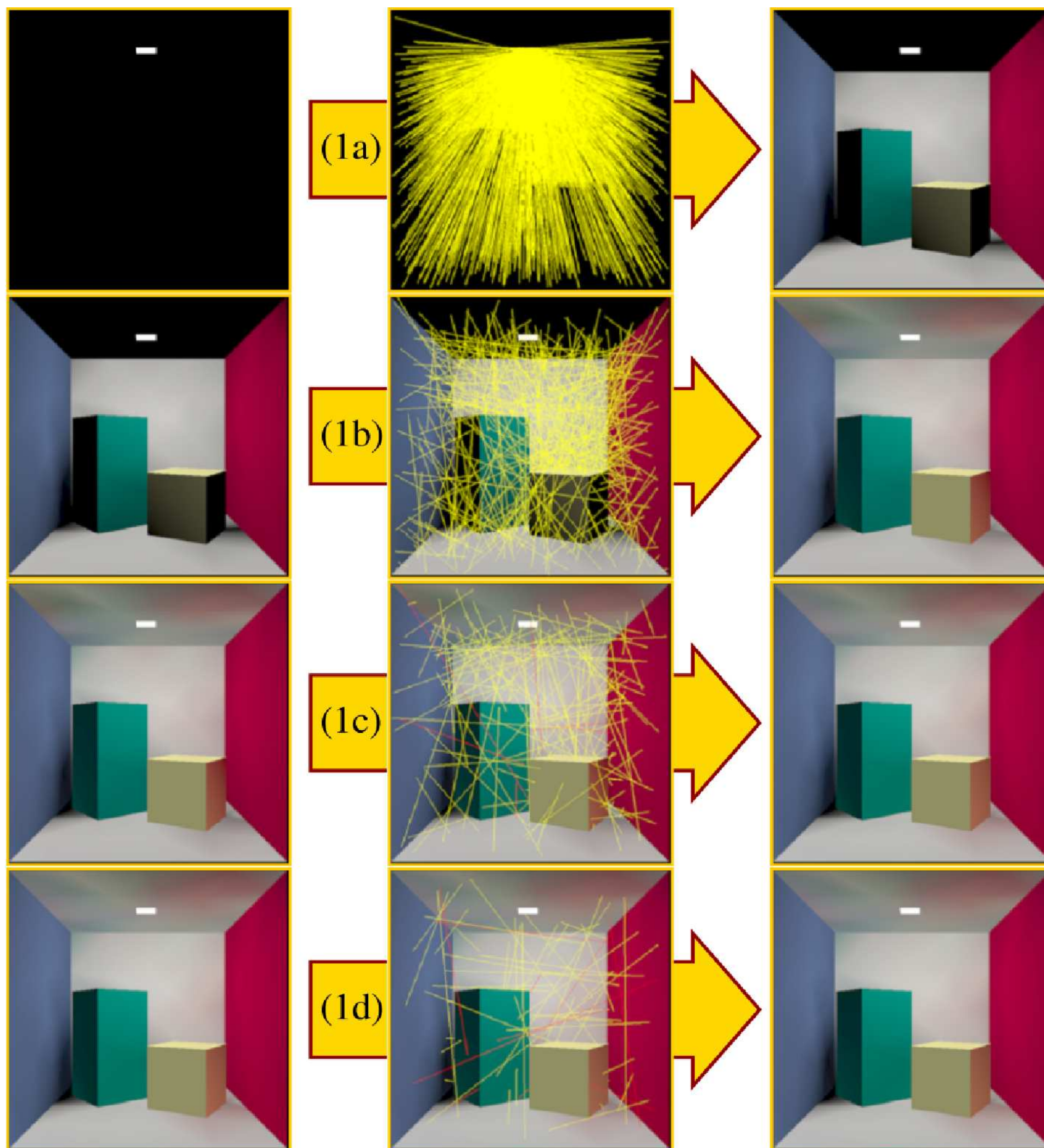


Figura 4.1: Método iterativo incremental estocástico de Jacobi en acción. Figura sacada de [30].

Teniendo en cuenta lo comentado en el Capítulo 1, se ha decidido trabajar con líneas locales seleccionando aleatoriamente el elemento destino S_j. Según esto, las sumas dobles de la Ecuación 4.3, una por cada elemento S_i, se pueden estimar de una manera simultánea y eficiente de la siguiente manera:

1. Selección de pares de elementos (S_j, S_l) mediante líneas locales:

   - Se escoge un elemento origen S_j con una probabilidad p_j proporcional a su potencia sin disparar, de forma que desde este elemento partirán un número de líneas locales proporcional a esta potencia:

$$p_j = \frac{\Delta P_j^k}{\Delta P_T^k} \ , \ \Delta P_T^k = \sum_{j=1}^{Ne} \Delta P_j^k \tag{4.4}$$

   - Se selecciona un elemento destino S_l con la probabilidad condicional $p_{l|j} = F_{jl}$. Para ello se trazan líneas locales escogidas aleatoriamente mediante una distribución uniforme un punto x_j sobre el elemento S_j, y a continuación se elige una dirección D_j para disparar el rayo desde x_j según una distribución coseno respecto de la normal a la superficie de S_j en el punto x_j. El elemento destino S_l será la superficie más cercana a S_j cruzada por el rayo (x_j, D_j). Si se disparan Nr_j rayos desde el elemento S_j según este procedimiento, el factor de forma F_{jl} se puede aproximar según la siguiente relación:

$$F_{jl} \approx \frac{Nr_{jl}}{Nr_j} \tag{4.5}$$

     donde Nr_{jl} es el número de rayos que parten de S_j y que alcanza la superficie de S_l. De esta manera, implícitamente se están obteniendo unos valores aproximados de la matriz de los factores de forma.

   La probabilidad combinada de la elección de la pareja (S_j, S_l) es:

$$p_{jl} = p_j \, p_{l|j} = F_{jl} \, \frac{\Delta P_j^k}{\Delta P_T^k} \tag{4.6}$$

2. Cada término escogido da una puntuación igual al valor del término dividido por su probabilidad p_{jl}. El valor medio de esa puntuación es una estimación sin sesgo para ΔP_i^{k+1}. Por ejemplo, si el elemento origen de una línea local t

es S_{j_t} y el elemento destino S_{l_t}, la estimación de ΔP_i^{k+1} para Nr líneas locales sería:

$$\frac{1}{Nr}\sum_{t=1}^{Nr}\frac{\Delta P_{j_t}^k\, F_{j_t l_t}\, \rho_{l_t}\, \delta_{l_t i}}{\Delta P_{j_t}^k\, F_{j_t l_t}/\Delta P_T^k} = \rho_i\, \Delta P_T^k\, \frac{Nr_i}{Nr} \approx \Delta P_i^{k+1} \tag{4.7}$$

donde $Nr_i = \sum_{t=1}^{Nr} \delta_{l_t i}$ es el número de líneas locales o rayos que llegan a S_i.

Este procedimiento puede ser utilizado para estimar ΔP_i^{k+1} para todos los elementos S_i simultáneamente. Básicamente, solo es necesario contar el número de rayos que llegan a cada elemento. Globalmente, el número de rayos aleatoriamente lanzados en la escena es Nr. En cada iteración el número de rayos lanzados es proporcional a la cantidad de potencia propagada, ΔP_T^k. Normalmente se asume que el algoritmo de Monte Carlo para radiosidad tiene una complejidad lineal-logarítmica, aunque es complicado determinarla exactamente. En cualquier caso, es mucho menor que cuadrática.

La heurística más habitual para determinar el número total de rayos a lanzar Nr, la cual busca la convergencia del proceso iterativo, es la siguiente [30]:

$$Nr \approx 9 \cdot \max_i \frac{\rho_i A_T}{A_i}, \quad i = 1, \ldots, Ne \tag{4.8}$$

donde A_T es el área total de los elementos de la escena. En la práctica, es razonable no tener en cuenta los elementos más pequeños. En nuestra aproximación se ha escogido un valor mínimo de área para hacer el cálculo de la ecuación anterior y se ha establecido el valor obtenido de Nr como el número de rayos que se lanzan en la primera iteración. Como la potencia se distribuye uniformemente entre todos los rayos, para el cálculo de la potencia por rayo utilizamos la siguiente expresión:

$$P_r = \frac{1}{Nr}\sum_{i=1}^{Ne} P_{ei} = \frac{1}{Nr} \cdot \Delta P_T^0 \tag{4.9}$$

donde ΔP_T^0 es el total de potencia autoemitida de la escena. Este valor, P_r, va a ser constante durante todo el proceso.

La Figura 4.2 muestra el algoritmo iterativo incremental estocástico de Jacobi que se ha usado. Este algoritmo se obtiene aplicando el método descrito para la estimación de ΔP_i^{k+1}, y utilizando la relación entre la potencia sin disparar de cada elemento, ΔP_i^k, y la potencia total sin disparar de toda la escena, ΔP_T^k, para

1 /* Inicializamos valores */
2 $\Delta P_T^0 = 0$; k = 0;
3 **for** (S_i = Escena→Elemento_inicial; S_i; $S_i = S_i$ →siguiente) {
4 $\quad P_i = \Delta P_i^0 = P_{ei}$; $\Delta P_i^1 = 0$;
5 $\quad \Delta P_T^0 = \Delta P_T^0 + \Delta P_i^0$;
6 }
7 P_r=Potencia_rayo();
8 **while** ($\Delta P_T^k > \varepsilon$) {
9 $\quad$ Número_muestras(Nr); $Nr_{prev} = 0$; $q = 0$;
10 $\quad$ **for** (S_i = Escena→Elemento_inicial; S_i; $S_i = S_i$ →siguiente) {
11 $\quad\quad$ /* Calculamos el número de rayos que se dispararán desde S_i */
12 $\quad\quad q = q + \frac{\Delta P_i^k}{\Delta P_T^k}$;
13 $\quad\quad Nr_i = \lfloor Nr \cdot q + \xi \rfloor - Nr_{prev}$;
14 $\quad\quad Nr_{prev} = Nr_{prev} + Nr_i$;
15 $\quad\quad$ /* Calculamos el rayo (x_t, D_t) y lo disparamos */
16 $\quad\quad$ **for** ($t = 0$; $t < Nr_i$; $t++$) {
17 $\quad\quad\quad$ Selección_aleatoria_punto(x_t);
18 $\quad\quad\quad$ Selección_aleatoria_dirección(D_t);
19 $\quad\quad\quad S_j$ = Elemento_más_cercano(x_t, D_t);
20 $\quad\quad\quad \Delta P_j^{k+1} = \Delta P_j^{k+1} + \rho_j \cdot P_r$;
21 $\quad\quad$ }
22 $\quad$ }
23 $\quad$ /* Actualizamos los valores de las potencias */
24 $\quad$ **for** (S_i = Escena→Elemento_inicial; S_i; $S_i = S_i$ →siguiente) {
25 $\quad\quad P_i = P_i + \Delta P_i^{k+1}$;
26 $\quad\quad \Delta P_i^{k+2} = 0$;
27 $\quad\quad \Delta P_T^{k+1} = \Delta P_T^{k+1} + \Delta P_i^{k+1}$;
28 $\quad$ }
29 $\quad k++$;
30 }

Figura 4.2: Algoritmo iterativo incremental estocástico de Jacobi.

calcular el número de rayos que se deben disparar desde cada elemento S_i. En primer lugar se inicializa la potencia total sin disparar ΔP_T^0 y la potencia sin disparar de cada elemento $P_i = \Delta P_i^0$ (desde la línea 1 hasta la 6). En la línea 7 se calcula la potencia por rayo P_r. Las iteraciones se repetirán hasta que se alcance el criterio de convergencia (línea 8), es decir, mientras que la potencia que reste por disparar sea mayor que un umbral ε. En la línea 9 se calcula el número de muestras o rayos que se dispararán en la iteración, que será proporcional a la cantidad de potencia ΔP_T^k que se propagará en esa iteración. Teniendo en cuenta que todos los rayos deben transportar la misma cantidad de potencia, el número de rayos lanzados en cada iteración será:

$$Nr = \frac{\Delta P_T^k}{P_r} \tag{4.10}$$

Una vez se obtiene el valor de Nr, se calcula el número de rayos Nr_i que serán disparados desde cada elemento S_i (líneas 12 y 13). Para disparar un rayo t desde el elemento S_i, primero se escoge aleatoriamente con una distribución uniforme un punto x_t sobre la superficie de S_i y luego se determina aleatoriamente una dirección D_t con una distribución coseno respecto de la normal de la superficie en x_t (líneas 17 y 18). Para cada rayo disparado, se determina el elemento más cercano S_j que es cruzado por él (línea 19). Después de esto, se actualiza la potencia ΔP_j^{k+1} que ha recibido el elemento S_j, que será la potencia sin disparar para la siguiente iteración (línea 20). Finalmente, para cada elemento se incrementa su potencia P_i con la recibida y se calcula el nuevo total de potencia sin disparar ΔP_T^{k+1} (desde la línea 23 hasta la línea 28).

4.2. Método iterativo incremental estocástico de Jacobi distribuido

La implementación paralela del método de Monte Carlo para la radiosidad que se presenta a continuación, está basada en una estrategia de paralelismo de grano grueso, en la cual cada procesador calcula la radiosidad de la subescena que tiene asignada. Además, también se ha usado la biblioteca de paso de mensajes MPI para las comunicaciones entre los procesadores.

El método iterativo incremental estocástico distribuido de Jacobi propuesto está basado en tres técnicas:

- Una partición convexa de la escena para permitir procesar escenas complejas y simplificar el proceso de disparo de rayos, sin realizar ningún tipo de aproximación en el cálculo de la iluminación.
- La utilización de paquetes de rayos para reducir las comunicaciones entre los procesadores.
- Un proceso distribuido para determinar la finalización de cada iteración en un sistema de procesamiento paralelo, sin la utilización de sincronizaciones o barreras que degraden el rendimiento.

Las tres técnicas han implicado grandes retos no solo desde el punto de vista algorítmico, sino también desde la perspectiva computacional debido a las dificultades asociadas con la programación en un entorno de comunicaciones de paso de mensajes. Además, se han usado comunicaciones no bloqueantes para permitir el solapamiento de computaciones y comunicaciones con el objetivo de incrementar el rendimiento de la aplicación, lo cual también aumenta la complejidad de la programación. A continuación se comentan estas tres propuestas.

La primera clave de nuestra propuesta es la partición de la escena. El objetivo es evitar la replicación de la escena en la memorias locales de cada procesador, lo que limitaría el tamaño de las escenas procesables por el método, y, al mismo tiempo, reducir la comunicación entre los procesadores a través de la explotación de la localidad de los datos. Además, esto produce una optimización en la utilización de la caché de los procesadores [105]. Una de nuestras principales metas de diseño fue la flexibilidad, que se alcanza con la utilización de componentes modulares. Por consiguiente, nuestra implementación paralela permite la utilización de cualquier algoritmo que proporcione una partición convexa y disjunta de la escena, de forma que siempre se establezcan comunicaciones solo entre los procesadores encargados de subescenas vecinas. Entre los posibles algoritmos de división se han empleado los esquemas de división uniforme y no uniforme ya presentados en los Capítulos 2 y 3. En la Subsección 4.2.1 se comentan brevemente estas técnicas de división convexa de la escena, indicando las diferencias respecto a lo que ya se ha visto en la implementación paralela del método de radiosidad progresiva. Como se ha comprobado estas técnicas permiten una división simple de la escena e importantes beneficios en términos de explotación de la localidad de los datos y balanceo de la carga. En la sección de resultados se mostrará un análisis detallado de estos beneficios y de la influencia del factor que se haya escogido para guiar la división no uniforme.

Nuestra segunda contribución está relacionada con la cantidad de rayos asociada con el algoritmo. Uno de los principales retos de la paralelización del algoritmo de Monte Carlo aplicado al cálculo de la radiosidad en un contexto de escena distribuida es el número elevado de rayos que tienen que ser comunicados entre los procesadores. Por ejemplo, en los estudios realizados de nuestra aplicación sobre nuestras escenas de prueba se ha visto que cientos de miles de rayos son comunicados entre los procesadores. Para minimizar y optimizar las comunicaciones se ha empleado una técnica basada en el empaquetamiento de rayos. Esta técnica permite reducir el número de comunicaciones y, además, incrementar la coherencia de los rayos [131]. Esto es un factor clave para obtener un procesamiento rápido del disparo de rayos, ya que los rayos con el mismo origen y una dirección similar se agrupan en el mismo paquete. Como resultado esto redunda en una optimización en el uso de la memoria caché. Se ha analizado exhaustivamente el tamaño de los paquetes de rayos para la optimización del rendimiento en términos de tiempo de ejecución. En la Sección 4.3 se pueden encontrar más detalles sobre este punto.

El último reto de la implementación paralela es el procedimiento para determinar el fin de una iteración. El algoritmo de radiosidad basado en el método iterativo incremental de Jacobi recorre todas las superficies de la escena y dispara Nr_i rayos desde cada elemento S_i en cada iteración. El siguiente paso iterativo no comienza hasta que haya terminado el procesamiento de todos los rayos del paso actual. Además, en un sistema de memoria distribuida varios rayos son disparados simultáneamente y el objeto más cercano que atraviesen puede estar en una subescena asignada a otro procesador. De esta forma, un procesador tiene que considerar no solo que ya haya procesado sus rayos, sino que no le va a llegar ningún otro rayo antes de comenzar la siguiente iteración. Nuestra propuesta está basada en un algoritmo de tres etapas, con una fase local, otra de comunicaciones y una final de comprobación global, y asegura un proceso de comprobación rápido, manteniendo un número reducido de comunicaciones y evitando barreras o sincronizaciones bloqueantes. Este método será explicado con más detalle en la Subsección 4.2.3.

En la Figura 4.3 se muestra la estructura detallada de nuestro algoritmo paralelo. En primer lugar, se calcula la potencia sin disparar $\Delta P^0_{T,l}$ de cada subescena l, $1 \leq l \leq Nd$ (línea 2), siendo Nd el número de subescenas en que se ha dividido la escena. La potencia sin disparar de toda la escena ΔP^0_T se obtiene con una operación de reducción, esto es, se recogen los resultados de todos los procesadores y se combinan con una operación de suma (línea 3). La potencia por rayo P_r se obtiene utilizando la misma estimación que se empleó en el algoritmo secuencial de la Figura 4.2. Para

```
1   /* Calculamos valores iniciales */
2   ΔP^0_{T,l} = Potencia_no_disparada(subescena^l) :  l = 1, ··· , Nd ;
3   ΔP^0_T = Reducción_potencia(ΔP^0_{T,l});
4   P_r=Reducción_potencia_rayo();
5
6   /* Proceso iterativo */
7   while (ΔP^k_T > ε) {
8          for (S_i = subescena^l →Elemento_inicial; S_i; S_i = S_i →siguiente)
                                                                 : l = 1, ··· , Nd {
9                /* Calculamos el número de rayos que se dispararán desde S_i */
10               Calcula(Nr_i);
11               /* Calculamos los rayos r_t(x_t, D_t) y los disparamos */
12               for (t = 1; t < Nr_i; t + +) {
13                    Selección_aleatoria_punto(x_t);
14                    Selección_aleatoria_dirección(D_t);
15                    S_j = Elemento_más_cercano(x_t, D_t);
16                    if ( S_j == NULL ) {
17                       v = Procesador_vecino(r_t(x_t, D_t));
18                       Buffer_{l,v} ← r_t(x_t, D_t);
19                       if ( Tamaño(Buffer_{l,v}) = Nb )
20                          Enviar(Buffer_{l,v});
21                    }
22                    else
23                         ΔP^{k+1}_j = ΔP^{k+1}_j + ρ_j P_r;
24               }
25               Recibir(Rayo_l);
26         }
27         while ( Fin_iteración_distribuida_( ) ) {
28                Recibir(Rayo_l);
29                Procesar_Rayo(Rayo_l);
30         }
31         k + +;
32         /* Actualizamos los valores de las potencias */
33         ΔP^k_{T,l} = Potencia_no_disparada(subescena^l): l = 1, ··· , Nd ;
34         ΔP^k_T = Reducción_potencia(ΔP^k_{T,l});
35  }
```

Figura 4.3: Algoritmo iterativo incremental estocástico de Jacobi distribuido.

ello es necesario realizar otra operación de reducción que calcula el área total de la escena.

Después de lo anterior se inicia el proceso iterativo que finalizará cuando la potencia total sin disparar ΔP_T^k sea menor que un cierto umbral ε escogido por el usuario (línea 7). Dentro de este proceso iterativo, en primer lugar se calcula el número de rayos Nr_i que serán disparados desde cada elemento S_i (línea 10). Como en el algoritmo secuencial, estos rayos son disparados seleccionando aleatoriamente un punto sobre la superficie de S_i mediante una distribución uniforme, y una dirección mediante una distribución coseno (líneas 13 y 14). A continuación hay que buscar el elemento más cercano a S_i cruzado por el rayo. Para ello, en primer lugar se analizan los elementos de la subescena local l (línea 15). En el caso de que haya alguna intersección local, se determina el elemento más cercano S_j que es alcanzado por el rayo y se incrementa su potencia asociada ΔP_j^{k+1} (línea 23). Si no hay intersecciones locales (desde la línea 16 hasta la línea 21), el rayo cruzará una frontera que es un elemento virtual común a dos subescenas vecinas. Una vez que el procesador vecino v es determinado, el rayo es añadido al paquete de rayos asociado con esa frontera, $\text{Buffer}_{l,v}$. Como se ha indicado anteriormente y para optimizar el rendimiento de las comunicaciones, se establece un tamaño máximo Nb para los paquetes de rayos. Cuando el paquete de rayos $\text{Buffer}_{l,v}$ alcanza este tamaño es enviado al procesador vecino v (línea 20). Adicionalmente al procesamiento de los rayos locales, el procesador debe chequear la llegada de otros rayos procedentes de los procesadores vecinos. Los rayos recibidos de otros subespacios son incluidos en una cola de rayos activos locales (Rayo_l) para su procesamiento (línea 25).

Después de que todos los rayos asignados a una subescena hayan sido procesados, el procesador correspondiente tiene que chequear si todos los rayos enviados en el sistema han alcanzado algún elemento (línea 27). En el caso de que haya rayos que aún no hayan alcanzado un objetivo, su procesador asignado deberá detectar el elemento más cercano que atraviesan en su subescena o, en otro caso, reenviar el rayo a un procesador vecino (línea 29).

Una vez que se ha comprobado que todos los rayos han alcanzado un destino en la escena, en cada subescena l se calcula la potencia total sin disparar $\Delta P_{T,l}^k$ (línea 33). Finalmente, la potencia total sin disparar de toda la escena ΔP_T^k se actualiza con otra operación de reducción (línea 34).

A continuación se detallarán las técnicas empleadas en la implementación del algoritmo iterativo incremental estocástico de Jacobi distribuido: la partición con-

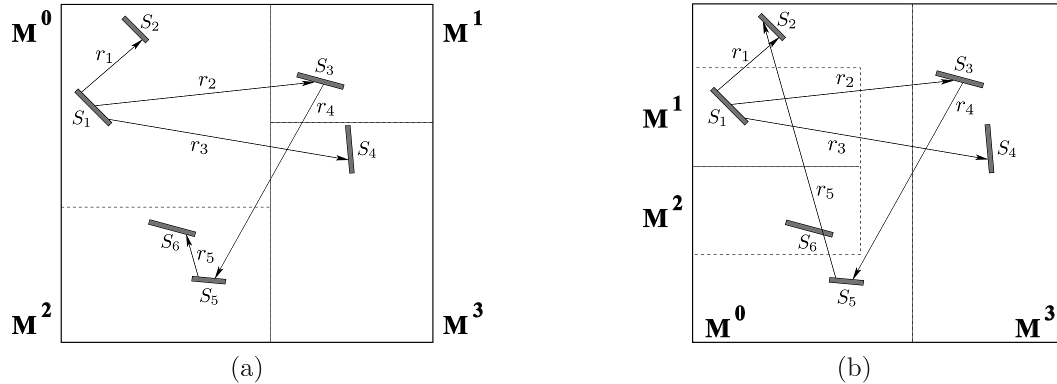


Figura 4.4: Partición realizada sobre una escena: (a) partición convexa, (b) partición no convexa.

vexa de la escena, el empaquetamiento de rayos y la comprobación del final de cada iteración.

4.2.1. Métodos de partición convexa de la escena

En la división de la escena se han utilizado las dos técnicas ya empleadas en la implementación paralela del método de radiosidad progresiva: la división uniforme que divide la escena en una malla regular de subescenas de igual forma y tamaño (ver Sección 2.2), y la división no uniforme que hace la partición basándose en la minimización de una función de distribución con el objetivo de balancear la carga computacional (ver Sección 3.1).

Las estrategias de división empleadas generan subescenas convexas, lo cual permite la simplificación del procedimiento de disparo de rayos que busca el objeto más cercano que es alcanzado por un rayo. Con una partición convexa, si un rayo alcanza a un objeto en la misma subescena del elemento emisor, entonces no es necesario comprobar el resto de particiones. Además, en el caso de que el rayo no alcance ningún objeto en la escena local, solo será necesario enviarlo a una de las escenas vecinas. En una división no convexa todas las particiones tienen que ser chequeadas para asegurar que el objeto más cercano ha sido realmente alcanzado, como se muestra en la Figura 4.4. Así en el ejemplo de la Figura 4.4a se aplica una partición convexa a la escena, con lo que el rayo r_5, lanzado a partir de un punto de la superficie S_5, no necesita comunicación entre los procesadores, al chocar con otro elemento, S_6, en el mismo subespacio que el del origen del mismo. Por otra parte,

en la Figura 4.4b el mismo rayo r_5 tiene que ser comunicado a varios procesadores para poder saber con seguridad cuál es el elemento más cercano que alcanza.

La técnica de división uniforme empleada con el algoritmo de Monte Carlo para radiosidad es exactamente la misma que la ya comentada en la Sección 2.2. Sin embargo, en el método de partición no uniforme visto en la Sección 3.1 se han introducido varias modificaciones. Se sigue empleando un algoritmo de partición estática debido a que el número de iteraciones, en general, en el algoritmo de Monte Carlo para radiosidad es bajo (entre 9 y 13 para nuestras escenas de prueba).

Una primer cambio que se ha hecho en la partición no uniforme es respecto al método de escoger el plano de división. Se ha comprobado que una mala elección en la coordenada donde se va a situar el plano de corte puede provocar un descenso significativo en el rendimiento de la implementación. En nuestra propuesta para el método de radiosidad progresiva paralela, inicialmente para la elección de la coordenada del plano de corte se calculan tres particiones diferentes asociadas a cada una de las tres coordenadas espaciales. Para ello, se coloca el plano de partición en la mitad de la caja que delimita el espacio mínimo que contiene a todos los elementos de la escena que estamos dividiendo. A continuación, de las tres posibilidades se selecciona la partición que tenga el valor menor de $\sigma(D_{X,2})$. Una vez escogida la dirección del plano de partición, se calcula la posición óptima de este plano. En cambio, en esta nueva aproximación en lugar de seleccionar la coordenada con el menor valor de $\sigma(D_{X,2})$ con el plano de partición en la mitad del subespacio, lo que se hace es calcular el mejor plano de corte en las tres coordenadas y después escoger uno de ellos. Para hacer esto, el criterio empleado es el análisis de los volúmenes de las subescenas resultantes. Específicamente, y para obtener las mejores particiones en las siguientes etapas, el plano de división final SD es aquel que produce volúmenes más equilibrados. Es decir, la partición que genere las subescenas con volúmenes similares es la preferida. Con esta elección se persigue la minimización del número de rayos que tienen que ser comunicados entre los procesadores.

Otra diferencia respecto a la aplicación de la partición no uniforme en el método de radiosidad progresiva, es que se ha escogido como factor guía de la división un valor que incluye no solo el área de la partición, sino también su potencia radiante. Esta elección fue debida a la naturaleza del método de Monte Carlo para radiosidad, en el cual se disparan rayos individuales en el lugar de elementos. El número de rayos depende de la potencia sin disparar y por eso es razonable pensar que se obtendría un mejor balanceo de la carga computacional si se tiene en cuenta la potencia a la hora de hacer la división de la escena. La expresión del factor utilizado es:

$$X = k \cdot P + A; \quad k \geq 0 \tag{4.11}$$

siendo P y A la potencia radiante y el área de una subescena, respectivamente, y k un coeficiente de peso para la potencia respecto del área. Destacar que con $k = 0$ el factor sería exactamente el mismo que el empleado con la radiosidad progresiva ($X = A$), y que cuanto mayor sea el valor de k mayor importancia tendrá la potencia en la partición de la escena.

En la Sección 4.3 se presenta un análisis más detallado de los beneficios de estas técnicas. Como se verá en esa sección, se han obtenido excelentes resultados en términos de *speedup* para los diferentes métodos de partición empleados. Esto demuestra que nuestra propuesta es una buena solución para la implementación del método de Monte Carlo para radiosidad en un sistema distribuido.

4.2.2. Estrategia basada en el empaquetamiento de rayos

Una de las características de cualquier implementación paralela del método de radiosidad es la gran cantidad de información que tiene que ser transmitida entre los procesadores. A diferencia de la implementación paralela del método de radiosidad progresiva, el algoritmo de Monte Carlo para radiosidad no utiliza ningún tipo de máscara para la comunicación entre procesadores, ya que no se realiza un reparto de radiosidad o potencia entre elementos visibles, sino lo que se hace es un disparo de rayos. Esto implica que solo es necesario seguir el camino del rayo hasta que intercepte al elemento más próximo. Como ya se ha comentado, si ese elemento está dentro de la subescena a la que pertenece el elemento origen del rayo, ya no será necesario enviar la información a otros procesadores. Sin embargo, si un rayo originado en un elemento no alcanza a ningún otro elemento de la subescena a la que pertenece, el rayo continua propagándose hacia una subescena vecina. En lo que respecta a la implementación paralela, esto implica la comunicación de información entre procesadores. Debido al gran número de rayos a procesar y a comunicar entre procesadores en este tipo de aplicaciones, el coste asociado a las comunicaciones puede ser considerable. En esta sección se describe el esquema que proponemos para la minimización y optimización de las comunicaciones entre procesadores asociadas a la transmisión de rayos entre subescenas.

Como se comentó en la subsección anterior, los métodos de división desarrollados se basan en la generación de subescenas de naturaleza convexa. Una de las ventajas

de esta estructura convexa es la inherente simplificación del proceso de comunicación debido a que se puede identificar de forma directa la subescena a analizar para cada rayo saliente de un subespacio específico. En este caso cada procesador para cada rayo saliente r_t identifica la subescena que lo va a recibir y el procesador encargado de su procesamiento. Para el cálculo del procesador vecino, en la división uniforme no hay ninguna variación respecto de la implementación realizada en el caso del algoritmo de radiosidad progresiva, pero sí se ha hecho una optimización en la división no uniforme debido a las diferencias entre los dos métodos de radiosidad. En la Sección 3.2 se definía una jerarquía de procesadores que se utilizaba para calcular el procesador vecino y reconstruir las máscaras de visibilidad cada vez que se enviaba un elemento a otro subespacio. En el caso del algoritmo de Monte Carlo no se emplean las máscaras, solo se necesita saber el procesador vecino de cada paquete de rayos. Por ello, en lugar de utilizar la jerarquía de los procesadores, lo que se hace es, previamente a la ejecución del algoritmo, determinar para cada frontera el procesador vecino, es decir, el procesador correspondiente al subespacio que comparte esa frontera. Este dato es almacenado con cada frontera. De esta forma cuando un rayo sale de un subespacio se calcula primero la frontera por la que cruza para pasar a otro subespacio. Determinada la frontera se tendrá también el subespacio vecino y su procesador asociado. Así no se necesita usar la jerarquía de procesadores cada vez que un rayo sale de un subespacio, ahorrando las comunicaciones asociadas a la utilización de esa jerarquía.

Una vez se ha identificado la subescena que va a recibir el rayo saliente, se realiza el envío de un mensaje con la siguiente información:

$$Mess_t = \{x_t, D_t, rgb_t\} \tag{4.12}$$

donde se incluyen las coordenadas del punto origen del rayo, x_t, los valores del vector de dirección del rayo, D_t, así como la contribución por color de ese rayo a la potencia en formato RGB. Así, un rayo que únicamente contribuye en el color rojo tiene el siguiente valor en el campo rgb_t, $Mess_t \rightarrow rgb = \{1, 0, 0\}$.

Cuando el procesador destino recibe ese mensaje procesa el rayo y busca el elemento local que puede alcanzar. De no producirse la intersección con los elementos asociados a la subescena en procesamiento, el rayo atravesará la subescena y penetrará en otra subescena vecina. De nuevo es tarea del procesador identificar la subescena implicada y el procesador al que está asignada, y realizar el envío de la información. Aunque la cantidad de rayos que no pueden ser procesados de forma local a cada subescena es totalmente dependiente de la naturaleza de la escena y

la partición realizada, las pruebas realizadas indican un elevado número de comunicaciones entre los procesadores. A modo de ejemplo, para dar una visión de la magnitud del número de comunicaciones involucradas en el procesamiento de una escena típica, para la escena *estudio* (ver Sección 4.3) y para una configuración de 32 procesadores y una partición no uniforme, se han generado unos 739.000 rayos y se han producido algo más de 1,2 millones de mensajes.

Siguiendo una estrategia semejante a la utilizada en los algoritmos de *ray tracing*, nuestra propuesta para minimizar las comunicaciones entre procesadores se basa en la utilización de paquetes de rayos. Tradicionalmente la búsqueda de la reducción del tiempo de ejecución de este tipo de algoritmos se basaba en la aceleración del procesamiento de rayos individuales, usando estructuras de indexado espacial o jerárquico para minimizar el número de pasos transversales y de intersecciones primitivas que había que realizar por cada rayo que se procesaba. Más recientemente las estrategias de aceleración se han centrado en el desarrollo de propuestas basadas en el procesamiento de grupos de rayos. Con este tipo de estrategias se consigue minimizar el número de accesos a memoria, compartir operaciones comunes y explotar la utilización de unidades SIMD [131]. La clave para obtener beneficios de esta estrategia es el agrupamiento de rayos con el mismo origen y dirección similar. Para ello en nuestra propuesta se crea un paquete de rayos $\mathrm{Buffer}_{l,v}$ para cada procesador l y para cada una de las fronteras v con sus subescenas vecinas. En este paquete, al ser enviado al procesador que tiene asignada dicha subescena, se incluyen todos los mensajes asociados a los rayos salientes de la subescena a través de la frontera v:

$$\mathrm{Buffer}_{l,v} \leftarrow Mess_t \tag{4.13}$$

El número de rayos por paquete, Nb, es constante para la escena que se está procesando. Este factor es determinante al no producirse el envío del paquete hasta estar completo, es decir, hasta haber detectado Nb rayos salientes a través de dicha frontera. Una vez procesados todos los rayos asignados al procesador se realiza el envío de los paquetes incompletos a los procesadores vecinos y se entra en la fase de comprobación de fin de iteración, que será detallada en la Subsección 4.2.3. Los paquetes incompletos son marcados como tal para que puedan ser identificados de forma correcta por los procesadores receptores.

El tamaño de los paquetes es un factor con una influencia importante en el rendimiento de la aplicación. Un tamaño pequeño puede implicar demasiados paquetes y, por consiguiente, podría producirse un gran número de comunicaciones. Cuanto

mayor sea el tamaño del paquete, mayor es el número de rayos involucrados en el envío de paquetes incompletos. Esto deriva en un mayor tiempo de espera de cada procesador receptor. Además la utilización de paquetes de mayor tamaño puede llegar a provocar contradictoriamente la generación de un mayor número de paquetes. Esto se debe a que los procesadores al entrar en la fase de comprobación del final de la iteración pueden realizar el envío de sus mensajes y recibir, a posteriori, paquetes incompletos procedentes de otros procesadores vecinos. Tamaños grandes de paquetes implican mayores requerimientos temporales para la generación y comunicación de paquetes de rayos. Por tanto, la comunicación de paquetes entre los procesadores se ralentiza y, consecuentemente, los tiempos de espera de los procesadores para recibir paquetes de rayos también se incrementa.

Siguiendo con el ejemplo anterior (escena *estudio*, configuración de 32 procesadores y división no uniforme) escogiendo un tamaño de paquete de rayos de $Nb = 100$ se generaron 22K paquetes, de los cuales 11K fueron paquetes completos. Sin embargo, para $Nb = 1000$ se crearon 36K paquetes, de los cuales solo 586 estaban completos. En la Sección 4.3 se incluye un análisis detallado del tamaño del paquete y su influencia en el rendimiento.

Nuestra propuesta, como se ha comentado anteriormente, usa la estrategia de comunicaciones no bloqueantes, puesto que, las comunicaciones bloqueantes suponen un lastre importante para el programa, al introducir esperas innecesarias hasta que un determinado mensaje llega a o es enviado por uno de los procesadores. La librería MPI proporciona una serie de funciones no bloqueantes que permiten solapar en lo posible cálculo y comunicaciones. En este caso es necesario una serie de puntos de muestreo a lo largo del código con el que se solapan las comunicaciones para comprobar si se han recibido nuevos mensajes.

El número de puntos de muestreo utilizado es un parámetro importante para la obtención de un buen rendimiento: un exceso de chequeos innecesarios supondrá una sobrecarga extra de instrucciones a ejecutar, mientras que comprobaciones demasiado espaciadas en el tiempo implican prolongar las esperas. El reto se encuentra, por tanto, en encontrar el equilibrio entre el trabajo local propio de cada procesador y su atención a peticiones remotas del resto de procesadores del sistema. Nuestra aproximación implementa un muestreo periódico, en el que cada procesador hace comprobaciones regulares de si ha recibido algún paquete nuevo.

4.2.3. Procedimiento para comprobar el final de una iteración

Comprobar que ha finalizado cada iteración del algoritmo es un problema complejo en un sistema de memoria distribuida, ya que hay que asegurar que todos los rayos disparados por todos los procesadores han alcanzado un elemento de la escena, teniendo en cuenta además que se usan comunicaciones no bloqueantes para optimizar el rendimiento del algoritmo. La solución implementada en nuestro algoritmo paralelo está basada en tres fases: una primera con un chequeo local en cada procesador, una segunda de comunicaciones entre todos los procesadores y una tercera de detección del fin de la iteración. Nuestro procedimiento evita el establecimiento de una barrera entre los procesadores y solo requiere comunicaciones no bloqueantes que se solapan con la ejecución de cálculos útiles.

En la primera etapa, cada procesador chequea un criterio local. Este comprobación evalúa si el procesador tiene rayos recibidos sin disparar en su subespacio y si tiene paquetes de rayos incompletos sin enviar. En el caso de que no existan rayos sin disparar y paquetes sin enviar, el procesador pasaría a la segunda etapa. En otro caso, se procesan los rayos recibidos y se envían los paquetes.

En la segunda fase, cada procesador l envía mensajes no bloqueantes a todos los otros procesadores con la siguiente información:

$$IF^l = \{N_{S,l}, N_{R,l}, N_S^l, N_R^l\}$$

donde $N_{S,l}$ y $N_{R,l}$ son, respectivamente, el número total de paquete enviados y recibidos por el procesador l, mientras que N_S^l y N_R^l son el número total de paquetes enviados y recibidos, respectivamente, por todos los procesadores del sistema según la información actualizada de la que dispone el procesador l.

En la tercera frase los procesadores utilizan la información recibida para detectar si la iteración ha terminado. Para ello, cada procesador actualiza los valores N_S^l y N_R^l con todos los valores $N_{S,m}$ y $N_{R,m}$ de los que dispone ($m = 1, \ldots, Nd$) y compara estos valores con los recibidos de los otros procesadores. La iteración habrá finalizado cuando $N_S^l = N_R^m$, para $1 \leq l, m \leq Nd$, es decir, cuando todos los procesadores hayan comprobado que el número total de paquetes de rayos recibidos en todo el sistema es igual al número total de paquetes de rayos enviados y que esta cantidad es la misma para todos ellos. En el caso de que esto no suceda se volverá a la primera fase porque aún quedará algún rayo en el sistema que no ha alcanzado su objetivo.

		Paquetes enviados			
	Procesadores	1	2	3	4
Paquetes recibidos	1	-	2	4	-
	2	3	-	-	1
	3	2	-	-	1
	4	-	1	3	-

Tabla 4.1: Ejemplo de paquetes de rayos intercambiados entre los procesadores.

Para aclarar esto vamos a considerar el ejemplo de la Tabla 4.1, donde se indica el número total de paquetes intercambiados entre cada procesador y sus vecinos. Por ejemplo, el Procesador 1 envía tres paquetes al Procesador 2 y dos al Procesador 3, mientras que recibe 2 del Procesador 2 y 4 del Procesador 3. Teniendo en cuenta esta información analicemos el estado de la iteración en una fase específica de la computación, donde se verifican las siguientes condiciones:

- El Procesador 1 se encuentra en la segunda fase del proceso de determinación del fin de la iteración, no ha recibido ninguna información de los otros procesadores y ha enviado su IF^1 a los otros procesadores.
- El Procesador 2 no ha finalizado la primera fase del proceso de determinación del fin de la iteración.
- El Procesador 3 y el Procesador 4 han finalizado la primera fase de la determinación del fin de la iteración y comienzan la segunda fase, pero aún no han enviado su IF a los otros procesadores.

Según estas condiciones, la información que se ha intercambiado es la siguiente:

$$\begin{aligned} IF^1 &= \{5, 6, 5, 6\} \\ IF^2 &= \{-, -, -, -\} \\ IF^3 &= \{7, 3, 12, 9\} \\ IF^4 &= \{2, 4, 7, 10\} \end{aligned}$$

Es decir, IF^1 indica que el Procesador 1 tiene solo información local y ha enviado cinco paquetes de rayos y recibido seis. El Procesador 3 y el Procesador 4 tienen su información local junto con la información recibida del Procesador 1, esto es, IF^1. Por

ejemplo, el Procesador 3 ha enviado siete paquetes de rayos y ha recibido tres, pero le ha comunicado el Procesador 1 que ha enviado cinco y recibido seis. En consecuencia el número total de paquetes intercambiados de acuerdo con la información de la que dispone el Procesador 3 es de doce paquetes enviados y de 9 paquetes recibidos en el conjunto del sistema.

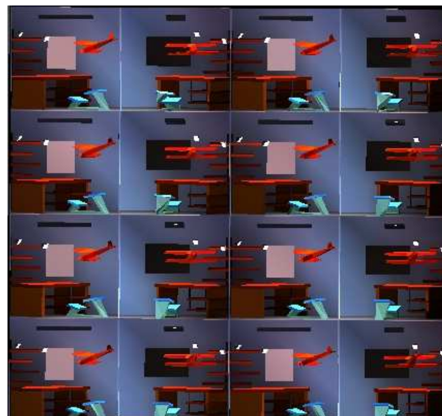
Los procesadores irán llegando a la tercera fase y comprobarán que los totales de paquetes recibidos y enviados no son los mismos en todos los procesadores, con lo cual volverán a la primera fase. La iteración habrá finalizado cuando estos totales tengan los mismos valores en todos los procesadores.

4.3. Resultados experimentales

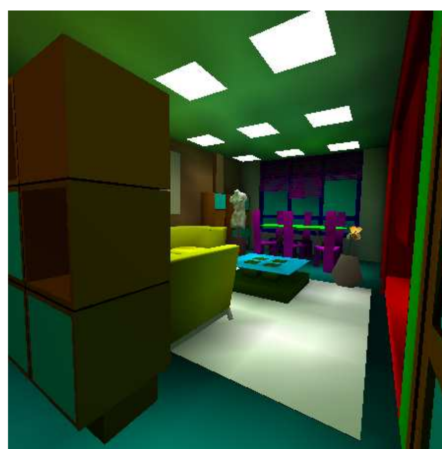
En esta sección se van a presentar los resultados de rendimiento y un análisis de nuestras propuestas. Como ya se ha comentado anteriormente, la paralelización del método iterativo estocástico incremental de Jacobi ha sido implementada con una aproximación de grano grueso mediante el paradigma de paso de mensajes. Específicamente, se ha utilizado el entorno de programación MPI [91].

En las pruebas que se han hecho para evaluar nuestra propuesta se ha empleado el supercomputador *Finis Terrae* [33], compuesto por varios nodos cada uno de los cuales consta de 16 núcleos físicos Itanium 2 Montvale a 1,6 GHz que pueden acceder a una memoria compartida de 128 GB. Estos nodos están conectados a través de una red Infiniband con un ancho de banda de interconexión de 16 Gbps y una baja latencia, que para el usuario es de 6 microsegundos. La política de ocupación del *Finis Terrae* está basada en la maximización del número de procesos MPI por nodo. Como consecuencia, los procesos MPI de una aplicación pueden finalmente ser ejecutados en el mismo nodo o en diferentes nodos. Adicionalmente, la carga de trabajos del sistema solo nos ha permitido ejecutar nuestra aplicación hasta con 32 procesadores. Este sistema es un DSM (*Distributed Shared Memory*), pero se ha usado como un sistema de memoria distribuida.

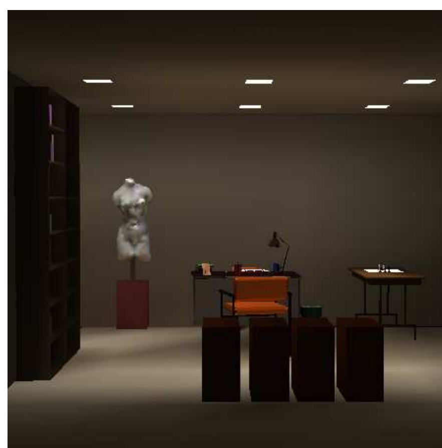
Para comprobar el rendimiento de nuestras propuestas se han utilizado las siguientes escenas: *edificio* (ver la Figura 4.5a), *salón* (ver la Figura 4.5b) y *estudio* (ver la Figura 4.5c). Estas tres escenas han sido seleccionadas debido a sus características específicas en términos de regularidad y distribución de los objetos y de las luces. Esto permitió, como se verá a continuación, la caracterización del rendimiento de nuestros algoritmos y de su dependencia de la naturaleza de la escena.



(a)



(b)



(c)

Figura 4.5: Escenas iluminadas: (a) *edificio*, (b) *salón* y (c) *estudio*.

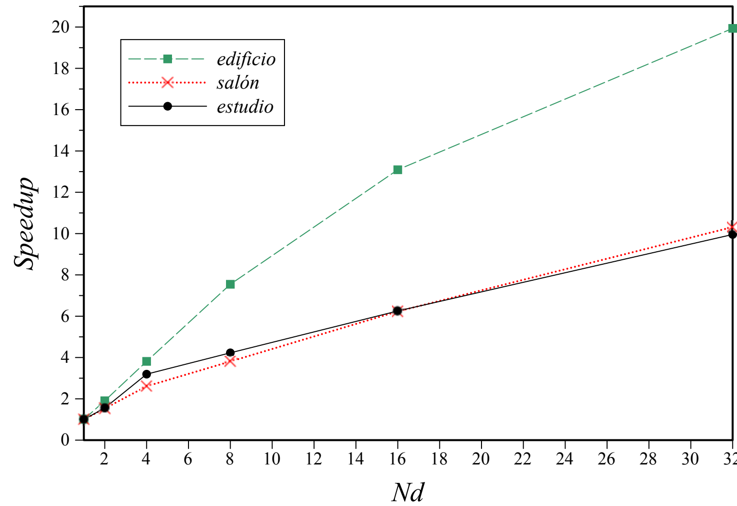


Figura 4.6: *Speedup* usando división uniforme.

La escena *edificio* fue construida replicando $4 \times 4 \times 4$ veces una escena de una habitación, abriendo dos puertas laterales entre ellas que permiten el paso de rayos de unas habitaciones a otras en la misma planta. Debido a su estructura, esta escena presenta una distribución muy regular de objetos y luces. La escena *salón* tiene una distribución irregular de objetos junto con un área sin iluminación (un pasillo sin luces). Finalmente, la escena *estudio* tiene una distribución irregular de objetos y una colocación uniforme de las luces.

En nuestro trabajo se analizó el rendimiento de nuestra propuesta de grano grueso. En primer lugar, se verificó, en términos de *speedup* y calidad, que el algoritmo de división convexa es una buena solución para la partición y distribución de la escena entre los procesadores en un sistema de memoria distribuida. Además, se compararon las dos aproximaciones: partición uniforme y no uniforme. Para simplificar, se ha considerado que cada subescena se asigna a un procesador, es decir, *Nd* es el número de procesadores. Después de comprobar los beneficios de la estrategia de partición no uniforme, se analizó la influencia de los factores que se pueden emplear para guiar el proceso de división. También fueron analizados los costes en tiempo de ejecución asociados a las técnicas de división de la escena. Una vez seleccionado el mejor procedimiento de partición, se estudió la influencia del tamaño máximo del paquete de rayos, *Nb*. Para finalizar, también se estudió la influencia de la frecuencia de chequeo de recepción de paquetes.

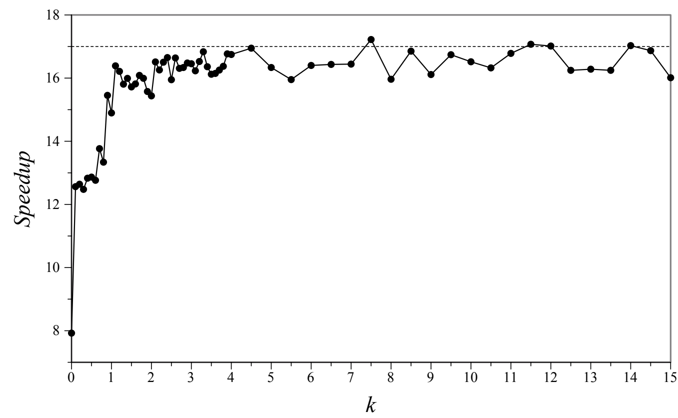
En primer lugar se ha realizado un análisis de nuestro método de división. La Figura 4.6 muestra el *speedup* de nuestro algoritmo paralelo con el método de divi-

sión uniforme para diferentes números de procesadores. Estos datos se han medido respecto de los resultados obtenidos con el algoritmo puramente secuencial sin división de la escena. Para el algoritmo secuencial los tiempos de ejecución fueron 19,21 segundos, 23,48 segundos y 12,50 segundos para las escenas *edificio*, *estudio* y *salón*, respectivamente. En términos de *speedup*, para la escena *edificio* se han obtenido buenos rendimientos con nuestra propuesta. Por ejemplo, para 32 procesadores el tiempo de ejecución para esta escena fue 0,96 segundos, correspondiendo a un *speedup* de 19,93. Para las otras escenas y con 32 procesadores, los tiempos de ejecución fueron 2,28 segundos para la escena *salón* y 1,26 segundos para la escena *estudio*, que equivalen a unos valores de *speedup* de 10,32 y 9,96, respectivamente. Los buenos resultados de la escena *edificio* están asociados con la regularidad de la escena que posee una distribución uniforme de los objetos y de las luces, características ideales para el método de división uniforme, porque produce una carga computacional balanceada, una reducción en el tiempo de búsqueda del elemento destino de un rayo y una reducción en los fallos caché [105]. Bajos *speedups* se han obtenido para las otras dos escenas debido a sus distribuciones no uniformes de objetos y luces, que implican un desbalanceo de la carga computacional.

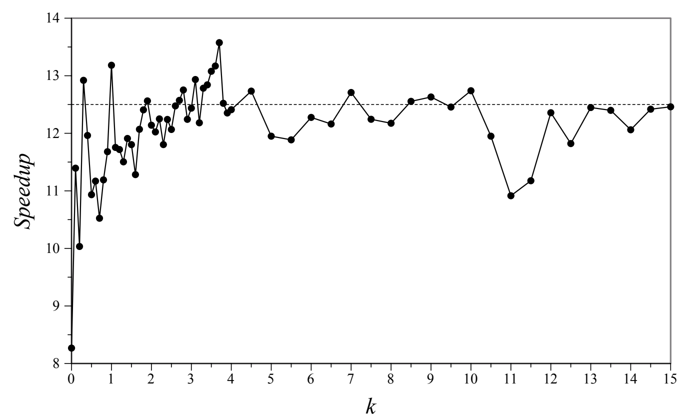
Para poder obtener un mejor balanceo de la carga computacional para escenas irregulares se ha analizado la partición no uniforme. Como ya se ha visto, nuestro método se basa en la selección de un factor X adecuado y en la minimización de una función de distribución, dependiente de este factor, que guíe la división de la escena (ver la Ecuación 3.2). Como resultado de nuestro estudio, finalmente se ha elegido y posteriormente analizado el siguiente factor:

$$X = k \cdot P + A; \quad 0,0 \leq k \leq 15,0 \tag{4.14}$$

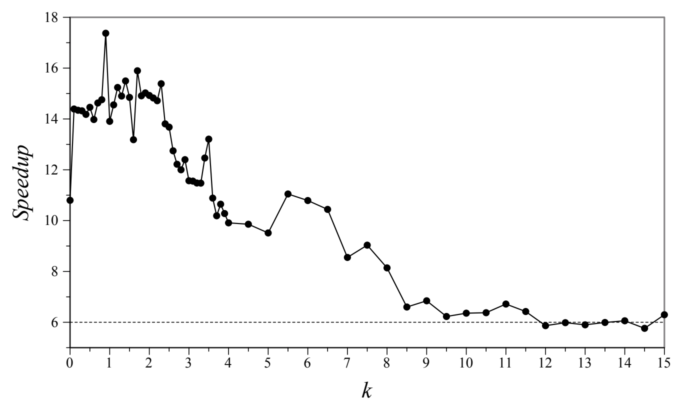
siendo P y A valores normalizados de la potencia radiante y el área de una subescena, respectivamente. Con $k = 0$ la división es guiada por el parámetro del área, mientras que $k > 0$ mejora la metodología de partición por la inclusión de la potencia radiante en la conducción del proceso de división. La Figura 4.7 muestra el *speedup* obtenido para todas las escenas de prueba para la división no uniforme y con $Nd = 32$ para diferentes valores de k. Como se puede observar en las gráficas, el tiempo de ejecución para cada escena depende de k para valores bajos ($k < 1$ para la escena *edificio*, $k < 4$ para la escena *estudio* y $k < 9$ para la escena *salón*), y es casi independiente de k para valores altos. Concretamente, para las escenas *salón* y *estudio* hay unos valores bajos específicos de k que proporcionan *speedups* muy elevados. Se debe hacer notar que estas escenas están caracterizadas por una distribución no uniforme de la



(a)



(b)



(c)

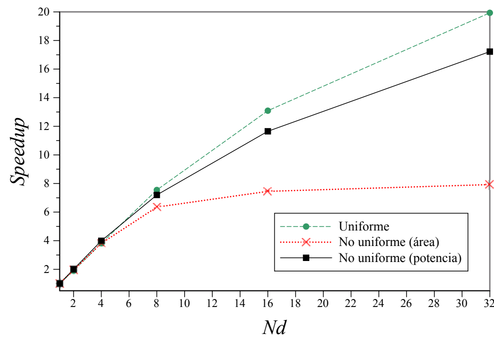
Figura 4.7: *Speedup* usando división no uniforme con diferentes valores de k y $Nd = 32$: (a) *edificio*, (b) *salón*, (c) *estudio*.

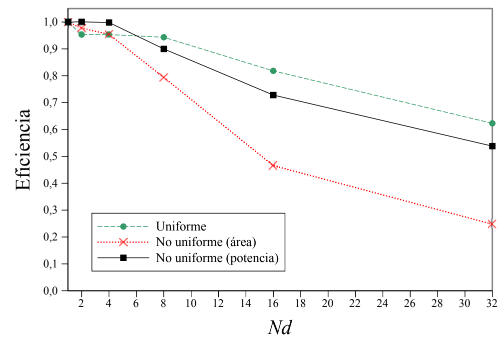
geometría y de las luces. A continuación se va a analizar este comportamiento en detalle.

Emplear el área como único parámetro para dirigir el proceso de división de la escena, es decir usar $k = 0$, puede generar subescenas sin fuentes de luz, con la consecuente degradación del balanceo de la carga computacional, ya que estas subescenas sin iluminación tendrá una pobre o nula carga de trabajo. Se debe indicar que esta falta de carga computacional en algunos nodos se puede resolver después de algunas iteraciones iniciales, cuando la potencia radiante que llegue de otras subescenas se detecte y se procese. Por esta razón, parece razonable pensar que mayores valores de k den valores de *speedup* iguales o superiores, ya que la utilización de la potencia como guía resolvería estas situaciones asociadas con subescenas sin fuentes de luz.

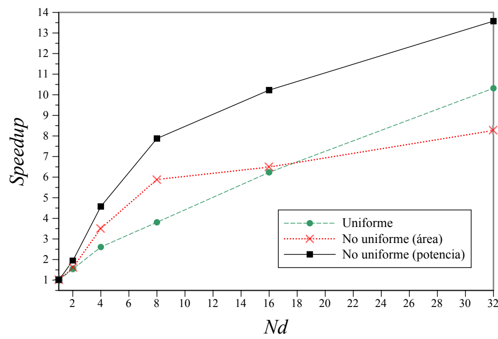
Teniendo en cuenta esto, parece que sería deseable utilizar $k > 0$ para las primeras iteraciones para evitar particiones sin iluminación. No obstante, en las siguientes iteraciones podría dar mejores resultados una división diferente que priorizara el parámetro del área. Como se puede observar en la Figura 4.7 hay valores específicos de k para los que se obtienen *speedups* muy altos para las escenas *salón* y *estudio*. Estas son escenas con una distribución irregular de luces y objetos. Estos valores específicos de k combinan las ventajas de la utilización de la potencia como parámetro para las primeras iteraciones y la inclusión del área en el factor que guía el proceso de partición para las siguientes iteraciones. Destacar también que para la escena *estudio* (ver la Figura 4.7c) hay una fuerte degradación en el rendimiento obtenido para valores grandes de k, debido a que en estos casos se producen subescenas casi vacías.

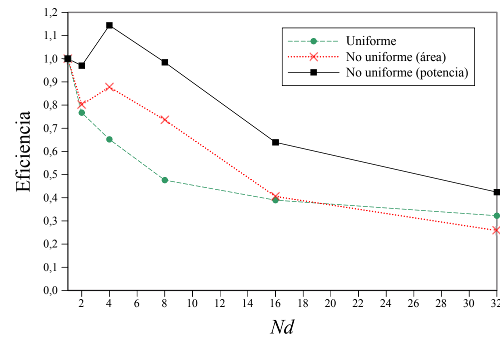
El siguiente paso en el análisis fue comparar los rendimientos de la división uniforme con los mejores rendimientos obtenidos con la división no uniforme. La Figura 4.8 muestra los resultados del algoritmo obtenidos para todas las escenas para los métodos de división uniforme y no uniforme, en términos de *speedup* y eficiencia. Específicamente, se han incluido los resultados obtenidos para $k = 0$ (etiquetados en la figura como “área”) y los mejores resultados obtenidos con $k > 0$ (etiquetados en la figura como “potencia”). Para escenas muy regulares los mejores rendimientos se obtienen con la partición uniforme, siendo ligeramente superiores a la versión “potencia” de la división no uniforme. No obstante, para las escenas con distribución irregular de objetos y fuentes de luz, los mejores resultados se han obtenido con versión “potencia” de la división no uniforme, consiguiéndose superlinealidad en el *speedup* en algunos casos. Por ejemplo, para la escena *estudio* los valores de



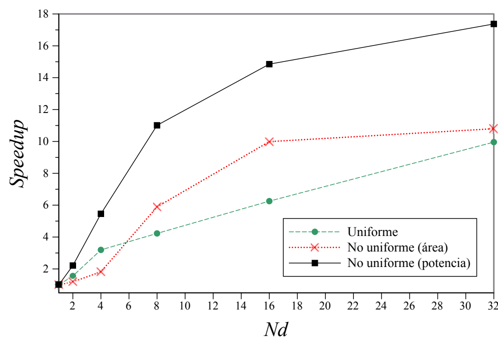


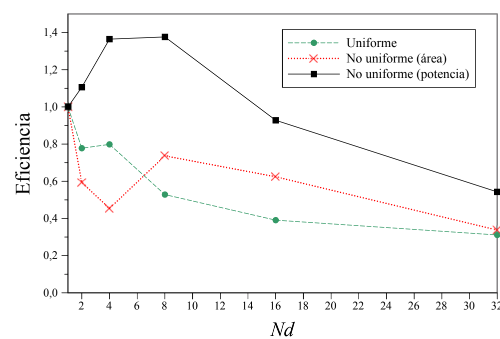
(a)





(b)





(c)

Figura 4.8: *Speedup* y eficiencia para la división uniforme y no uniforme usando área y potencia: (a) *edificio*, (b) *salón* y (c) *estudio*.

	$\sigma(D_{X,32})$	
	Uniforme	No uniforme
Edificio	0,0084	0,0065
Salón	1,8684	0,0640
Estudio	1,4109	0,1770

Tabla 4.2: Distribución de la carga computacional para una configuración de 32 procesadores.

speedup fueron 2,21, 5,46 y 11,01 para 2, 4 y 8 procesadores, respectivamente. Por consiguiente, estos resultados prueban que las técnicas que hemos propuesto para la paralelización del algoritmo de Monte Carlo para radiosidad permiten lograr un excelente rendimiento en sistemas de memoria distribuida.

Como justificación del buen funcionamiento de la división no uniforme, se ha chequeado el balanceo de la carga computacional midiendo la función de distribución de cada método (ver la Ecuación 3.2). Concretamente, la Tabla 4.2 muestra los valores de $\sigma(D_{X,32})$ para una configuración de 32 procesadores aplicando las estrategias de división uniforme y no uniforme, donde se han utilizado para el factor X los valores de k correspondientes a los mejores resultados obtenidos para la versión no uniforme. Como se puede observar, se obtienen los valores más bajos y, en consecuencia, aplicaciones más balanceadas para la versión no uniforme. De todas formas hay que indicar que para la escena *edificio* los mayores *speedups* se consiguen con la partición uniforme aunque el valor de $\sigma(D_{X,32})$ es más alto. Esto se debe a las características especiales de la escena y a la configuración de los procesadores. Para este caso especial, en la versión uniforme a cada procesador se le asignan exactamente dos habitaciones completas, resultando una óptima explotación de la localidad y un número más bajo de comunicaciones.

Por otro lado, se ha realizado una comparación entre el tiempo computacional requerido por cada etapa del algoritmo paralelo y el tiempo computacional total. La Tabla 4.3 muestra las medidas en segundos obtenidas para la escena *salón* para cada una de las técnicas de división, uniforme y no uniforme. El primer valor para cada técnica representa el tiempo de inicialización, el segundo es el tiempo requerido para la partición de la escena, el tercero es el tiempo necesario para calcular el *kd–tree* y las iteraciones, y el cuarto valor es el tiempo total de ejecución. Destacar que para ambos métodos de partición el tiempo requerido para la división es razonable respecto del tiempo computacional total. Por ejemplo, para $Nd = 16$ y para la

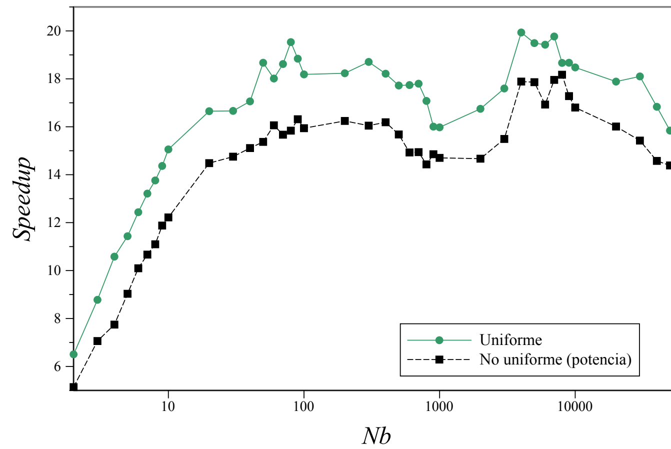
Nd	uniforme				no uniforme			
	inicio	partición	kd+iter	total	inicio	partición	kd+iter	total
2	0,1166	0,0109	15,273	15,305	0,1407	0,0309	14,583	14,654
4	0,1051	0,0120	8,981	9,003	0,1119	0,0628	6,610	6,686
8	0,0987	0,0113	6,150	6,165	0,1056	0,0771	3,906	3,989
16	0,0982	0,0116	3,753	3,767	0,1002	0,0945	3,522	3,619
32	0,0991	0,0172	2,256	2,276	0,0995	0,1744	1,553	1,729

Tabla 4.3: Comparación entre el tiempo de ejecución total y el de cada etapa del algoritmo distribuido para la escena *salón*.

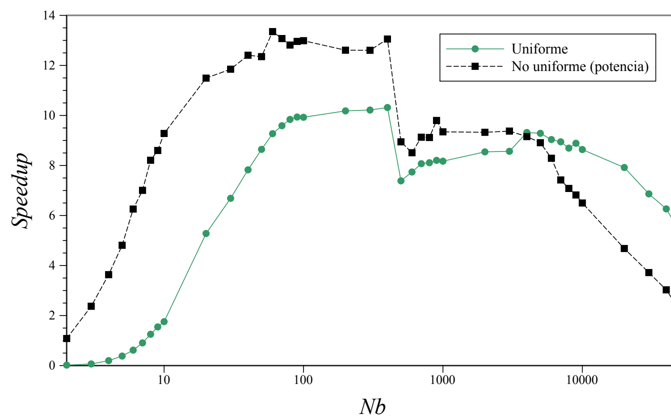
versión de división no uniforme, la técnica de partición requiere un 2,6 % del tiempo computacional total.

Como otro paso en la evaluación de nuestra propuesta, también se ha analizado experimentalmente el tamaño máximo óptimo del paquete de rayos. Los resultados obtenidos están resumidos en la Figura 4.9, donde se ha utilizado en la división no uniforme el valor de k óptimo para cada escena (ver Figura 4.7). Estas gráficas muestran la dependencia del *speedup* con el tamaño del paquete de rayos, Nb, para una configuración de 32 procesadores. Como se puede observar, los mejores rendimientos se obtienen para un tamaño máximo de paquete de rayos entre, aproximadamente, 0,71 y 14,1 KiB, que corresponde a valores de Nb entre 20 y 400, para las escenas *salón* y *estudio*, y entre 0,71 y 351,6 KiB, correspondientes a valores de Nb entre 20 y 10000, para la escena *edificio*. En estos rangos, hay un número reducido de comunicaciones, una eficiente explotación del ancho de banda de interconexión y una más eficiente planificación de la computación, ya que los ciclos de espera de los procesadores son reducidos. El rango amplio de la escena *edificio* es debido a su diferente naturaleza. En este caso y para $Nd = 32$ se genera una situación muy adecuada, ya que dos habitaciones, cada una con dos puertas abiertas, son asignadas a cada procesador.

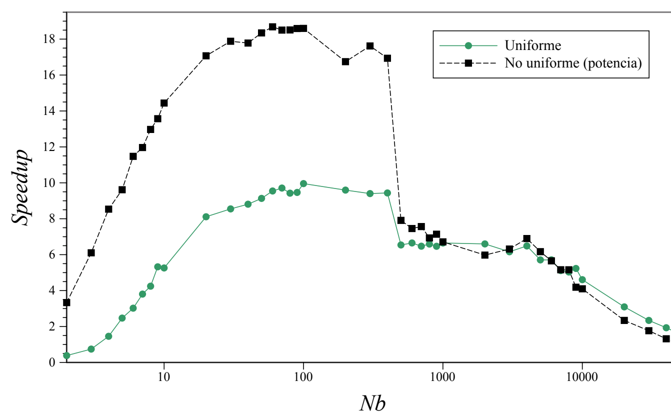
Hay que hacer notar que para un procesador dado, los paquetes incompletos de rayos que llegan desde los procesadores vecinos pueden generar más comunicaciones si los rayos no alcanzan ningún objeto en la subescena asociada. En esta situación, los rayos correspondientes tienen que ser enviados de nuevo a otros procesadores. Para valores de Nb dentro de los rangos especificados, estos rayos pueden ser eficientemente procesados y agrupados en los paquetes de rayos locales en construcción, reduciendo de esta forma el número global de comunicaciones. Para valores de Nb mayores el comportamiento cambia: el periodo de espera de los procesadores se in-



(a)



(b)



(c)

Figura 4.9: Dependencia del *speedup* con el tamaño del paquete de rayos para 32 procesadores: (a) *edificio*, (b) *salón y (c) estudio.*

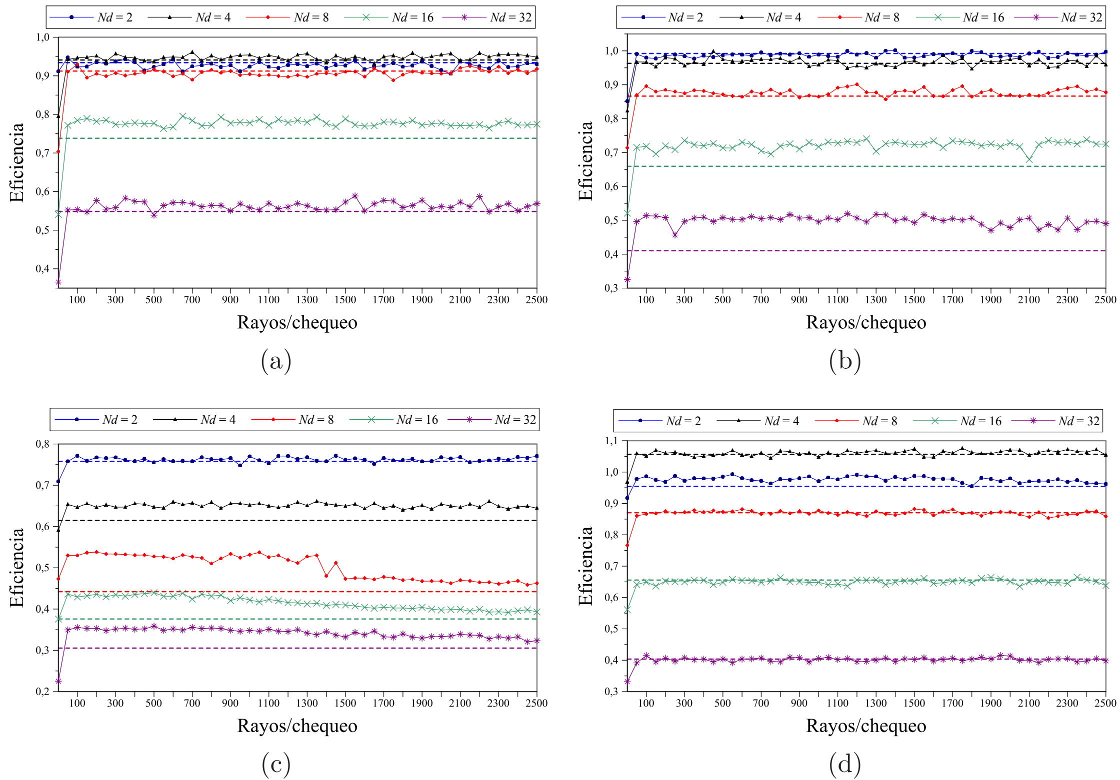


Figura 4.10: Dependencia de la eficiencia con la frecuencia de chequeo de recepción de paquetes: (a) escena *edificio* partición uniforme, (b) escena *edificio* partición no uniforme, (c) escena *salón* partición uniforme y (d) escena *salón* partición no uniforme.

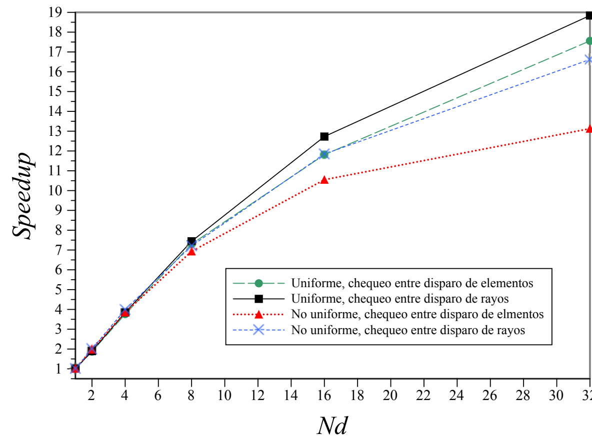
crementa debido a la necesidad de empaquetar un mayor número de rayos. Además, la comunicación de los paquetes finales incompletos es más costosa debido a que, en promedio, tiene que ser enviado un número mayor paquetes incompletos con el consecuente incremento en los requerimientos de comunicaciones.

Finalmente, se ha analizado la influencia de la frecuencia del chequeo de paquetes recibidos en el rendimiento del algoritmo distribuido. En los resultados presentados hasta ahora el chequeo de la recepción de paquetes se realiza finalizado el disparo de los rayos de cada elemento (línea 25 del algoritmo de la Figura 4.3) y cuando se comprueba el fin de la iteración (línea 28 de la Figura 4.3). Para este último análisis, lo que se ha hecho es poner un chequeo adicional en el bucle del disparo de rayos (líneas de la 12 a la 24) que se realice cada cierto número de rayos disparados. De esta manera se puede comprobar con mayor frecuencia la recepción de paquetes, sin

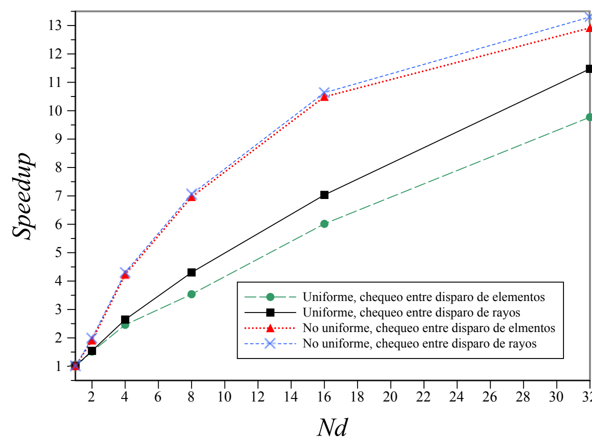
tener que esperar a que sean disparados todos los rayos de un elemento.

En la Figura 4.10 mostramos la dependencia del rendimiento con la frecuencia de chequeo para las escenas *salón* y *edificio*, empleando un valor de tamaño de paquete de $Nb = 100$. Para la partición no uniforme se ha usado $k = 3,7$ con la escena *salón* y $k = 7,5$ con la escena *edificio*, que son los valores de k que mejores rendimientos dieron en las pruebas presentadas anteriormente (ver Figura 4.7). Para poder comparar mejor en una sola figura los resultados de los distintos valores de Nd, se ha optado por mostrar los datos de eficiencia. Además se han marcado con líneas discontinuas los mejores valores de eficiencia obtenidos con las pruebas realizadas anteriormente para cada valor de Nd. Como se puede observar para la escena *salón*, en el caso de la división uniforme un chequeo de recepción de paquetes dentro de un rango de un número de rayos lanzados (entre 100 y 1300 rayos) puede beneficiar el rendimiento (ver Figura 4.10c). Sin embargo, en el caso no uniforme el rendimiento es prácticamente el mismo y no se obtiene beneficio de un aumento de la frecuencia de chequeo (ver Figura 4.10d). Por otro lado, para la escena *edificio* el comportamiento difiere, ya que prácticamente no hay mejora de rendimiento en el caso de la partición uniforme, mientras que sí lo hay para la división no uniforme para $Nd = 16$ y, sobre todo, para $Nd = 32$.

Este comportamiento es debido al desbalanceo de la carga computacional: para la escena *salón* en la división uniforme, en general, hay unos pocos procesadores que disparan muchos rayos, mientras que otros alcanzan muy pronto la fase de comprobación del fin de la iteración (líneas de la 27 a la 30 en el algoritmo de la Figura 4.3). Estos últimos procesadores entran en un bucle de inactividad, hasta que reciban paquetes de rayos. Por otro lado, los primeros pueden actuar como barreras que retarden el paso de rayos a través de su subescena a subescenas vecinas. Todo esto provoca una reducción del rendimiento del algoritmo distribuido. Así, un chequeo más frecuente de los paquetes recibidos evita la acumulación de estos sin ser procesados, beneficiando la ejecución del algoritmo. En cambio, en la división no uniforme con una carga computacional más equilibrada, los procesadores tienen mejor distribuido el disparo de rayos, con lo cual el envío y recepción de paquetes está más balanceado entre ellos, no necesitándose una frecuencia mayor de chequeo que la establecida en las pruebas realizadas anteriormente. El caso de la escena *edificio* es singular, como ya se ha comentado, sobre todo para $Nd = 32$. Concretamente, en esta configuración, en la división no uniforme se alcanza el mejor balanceo en términos del factor X (ver la Tabla 4.2). Sin embargo, el balanceo ideal se logra para 32 procesadores si se asignan dos habitaciones por procesador. Esto es lo que



(a)



(b)

Figura 4.11: Comparación del *speedup* según cuando se hace el chequeo de paquetes recibidos: (a) escena *edificio* y (b) escena *salón*.

sucede para la división uniforme, y así en este caso no es necesario aumentar la frecuencia de chequeo. Pero esta asignación no ocurre en la división no uniforme, y, por tanto, para este método de partición se logra un mejor rendimiento con un chequeo más frecuente de los paquetes recibidos.

En resumen, hemos presentado una implementación distribuida del algoritmo de Monte Carlo para radiosidad empleando cuatro técnicas: la partición convexa de la escena, uniforme o no uniforme, para la distribución del procesamiento; el empaquetamiento de rayos para mejorar las comunicaciones; el chequeo periódico de la recepción de rayos para evitar procesadores ociosos y otros que actúen como barreras; y, finalmente, una comprobación distribuida del fin de iteración, sin

comunicaciones bloqueantes que actúen como barreras de sincronización. En la Figura 4.11 se pueden ver los mejores valores de *speedup* de nuestra implementación para las escenas *salón* y *edificio*. Se muestran las gráficas para los dos métodos de partición y para el chequeo de recepción de paquetes entre disparo de elementos o entre disparo de rayos. Como puede verse, en todos los casos hay una mejoría cuando se realiza un chequeo entre un número de rayos disparados, siendo menor cuanto mejor esté distribuida la carga computacional.

Capítulo 5

Implementación GPU del método de Monte Carlo para radiosidad

La evolución de las GPUs ha permitido su explotación para computaciones de propósito general, *General Purpose Programming using Graphics Processing Unit* (GPGPU). Sin embargo, hasta bien entrada la primera década de este siglo, la programación de las GPUs para GPGPU era compleja por la necesidad de utilizar lenguajes como NVIDIA Cg [32], o Microsoft HLSL [45], que están optimizados para cálculos gráficos y requieren APIs gráficas como OpenGL [90] o Direct3D [45]. Desde 2006 han aparecido nuevos lenguajes de programación que buscan facilitar el proceso de paralelización de las aplicaciones de propósito general para GPUs: Brook+ para las GPUs de ATI-AMD [1], CUDA para las GPUs de NVIDIA [88] y OpenCL [66] para todo tipo de sistemas. Brook+ es una implementación del lenguaje Brook optimizada para el hardware AMD. CUDA fue presentado por NVIDIA en el 2006 y tiene soporte para varios lenguajes de programación como C o FORTRAN. OpenCL fue desarrollado inicialmente por Apple para ser traspasado más tarde al Khronos Group que lo mantiene como un estándar abierto que pretende ser independiente de la plataforma. Hoy en día también puede ser utilizado en las GPUs de NVIDIA, aunque su rendimiento es inferior a CUDA. Esto es debido a que el hardware de NVIDIA ha sido desarrollado estrechamente ligado a su propio API CUDA y el entorno de CUDA está en un estado más avanzado de evolución.

Con respecto a los algoritmos de iluminación global sobre GPU, han aparecido implementaciones usando *antirradiancia* [26] o *visibilidad implícita* [28] con el objetivo de evitar el cálculo de la visibilidad entre las superficies, aunque requieren la

generación de estructuras que suponen un coste computacional adicional [77]. Otras implementaciones para el cálculo de la iluminación indirecta aplican aproximaciones usando *visibilidad imperfecta* [102] o desarrollan esquemas simplificados para una iluminación indirecta plausible [62].

En el Capítulo 4 se ha visto que el método de Monte Carlo para radiosidad (MCR) está basado en el disparo de rayos y en la búsqueda del primer objeto que es alcanzado por cada rayo. Por tanto, debido a que los rayos no requieren información de otros rayos, el cálculo relativo a cada rayo puede ser hecho de forma independiente. De esto se deduce que el proceso de disparo de rayos es extremadamente paralelo a nivel de tarea (*thread*). Esta naturaleza paralela nos ha llevado a explorar la implementación del algoritmo sobre GPUs con el cálculo explícito de la visibilidad, es decir, sin la utilización de aproximaciones para este cálculo.

Específicamente, en este capítulo se presenta una implementación de grano fino sobre una GPU NVIDIA utilizando el lenguaje de programación CUDA C, demostrando que nuestra propuesta es una alternativa viable a las implementaciones previas. Nuestro trabajo se ha basado en la utilización de la división de la escena como método para la aceleración del chequeo de la intersección de rayos. Por otro lado, las técnicas de partición de la escena permiten la optimización de la localidad de los datos y la explotación de la jerarquía de memoria de la GPU. Además hemos incluido otras técnicas como la utilización de una versión simplificada de la malla de elementos para reducir los costes computacionales del proceso de disparo de rayos, y la propuesta de una planificación eficiente de la ejecución de tareas que maximiza la utilización de los recursos computacionales disponibles en la GPU.

El resto del capítulo comienza con una introducción sobre las GPUs. Sigue la Sección 5.2 con un análisis de la estructura de una GPU de NVIDIA. Posteriormente, se describe el entorno de programación CUDA en la Sección 5.3. A continuación, en la Sección 5.4 se detalla nuestra implementación del algoritmo de Monte Carlo para radiosidad sobre GPU usando CUDA. Finaliza el capítulo con la Sección 5.5 mostrando y analizando los resultados experimentales obtenidos, los cuales han sido publicados en [109, 111].

5.1. Introducción a las GPUs

Desde el año 2003, la industria de los semiconductores se ha orientado hacia dos caminos principales para el diseño de los microprocesadores [57]. Por un lado están

los procesadores multinúcleo, que buscan aumentar la velocidad de ejecución añadiendo múltiples núcleos que realizan un procesamiento paralelo y ejecutan múltiples tareas al mismo tiempo. Estos sistemas multinúcleo comenzaron con procesadores de doble núcleo y, aproximadamente, se fue doblando el número de núcleos con cada generación de procesadores. Un ejemplo actual es el microprocesador Intel Core i7, con cuatro núcleos, cada uno de los cuales implementa el conjunto completo de instrucciones de la familia x86 de Intel [67]. Siguiendo esta tendencia, Intel ya ha creado el *Single-chip Cloud Computer* (SCC), un procesador experimental que incluye 48 núcleos integrados en un único chip [58].

La otra alternativa son los procesadores *many-core* que están orientados a la ejecución de aplicaciones paralelas. Estos procesadores empezaron con un gran número de pequeños núcleos y, de nuevo, este número se fue doblando con cada generación. Los procesadores *many-core*, especialmente las GPUs, han liderado la carrera del rendimiento de las operaciones en punto flotante desde el 2003. Mientras que la mejora del rendimiento en los procesadores de propósito general se ha ido ralentizando significativamente, las GPUs han continuado mejorando sin pausa. Considerando la velocidad máxima que potencialmente se puede obtener en estos chips para operaciones en punto flotante, en el 2009 se conseguían unos valores de 1 teraflop para las GPUs frente a 100 gigaflops para los procesadores multinúcleo, lo que supone una relación de 10 a 1 [67]. Este hecho ha motivado que muchos desarrolladores de aplicaciones hayan trasladado a las GPUs la ejecución de las partes de su software que requieren un cálculo intensivo. Un ejemplo actual de un procesador *many-core* es la GPU GeForce GTX 580 de NVIDIA con 512 núcleos [89].

5.1.1. Evolución de las GPUs

Las modernas GPUs de gráficos 3D han evolucionado desde procesadores de funciones gráficas fijas a procesadores paralelos programables con capacidades superiores a las CPUs de varios núcleos. Así, la primera GPU fue la GeForce 256 que apareció en 1999 y que permitía el procesamiento en punto flotante de los vértices en una unidad (*vertex shader*), mientras que en otras unidades se procesaban los píxeles (*pixel shader*). Era programable mediante OpenGL y la API DX7 de Microsoft. En 2001, la GeForce 3 presentó el primer *vertex shader* programable con DX8 y OpenGL. La Radeon 9700, aparecida en 2002, procesaba los píxeles en punto flotante con 24 bits con DX9 y OpenGL. Posteriormente, la GeForce FX pasó a procesar los píxeles en punto flotante con 32 bits. En 2005 la Xbox 360 presentó

una primera GPU unificada, permitiendo que la computación de píxeles y vértices se realizara en el mismo procesador.

En noviembre de 2006 apareció la arquitectura Tesla de NVIDIA [72] en la GPU GeForce 8800, la cual unificó el *pixel shader* y el *vertex shader*. Por otra parte, esta nueva arquitectura permite ejecutar aplicaciones paralelas de propósito general de alto rendimiento usando CUDA (*Compute Unified Device Architecture*) [88]. La arquitectura unificada de computación y gráficos Tesla está disponible en la familia de GPUs de la serie 8 de GeForce y en las GPUs Quadro para portátiles, equipos de sobremesa, estaciones de trabajo y servidores. También proporciona la arquitectura de procesamiento para las plataformas de computación de altas prestaciones constituidas por GPUs tipo Tesla que aparecieron en 2007.

En el 2009 se presentó la arquitectura Fermi de NVIDIA [87]. Respecto a su predecesora, la arquitectura Fermi mejora el rendimiento de las operaciones en punto flotante sobre todo en doble precisión, permite usar memorias ECC (memorias con corrección de errores), utiliza memorias caché, amplia la memoria compartida y realiza las operaciones atómicas más eficientemente. La arquitectura Fermi es la última generación de GPUs de NVIDIA en el mercado, pero en el congreso *GPU Technology Conference* de 2010, Jen–Hsun Huang, director ejecutivo y cofundador de NVIDIA, ya anunció los próximos proyectos: Kepler y Maxwell.

En cuanto a la programación, hasta hace poco las unidades programables para *pixel shader* y *vertex shader* de las GPUs solo podían operar con datos existentes, es decir, no podían ser empleadas para generar o destruir geometría de una forma directa [2]. Normalmente el *vertex shader* se emplea para transformaciones de los vértices y para cálculos en cada vértice, no siendo accesible la información de cada vértice al resto. Por otro lado, el *pixel shader* calcula básicamente el color de cada fragmento de la escena. Este panorama cambió con la introducción con Direct3D10 (ver Figura 5.1a) del *geometry shader* [12]. Este *shader* permite la creación y destrucción de datos geométricos en la GPU abriendo con esta característica un amplio abanico de posibilidades. Así, algoritmos de procesamiento gráfico tradicionalmente limitados en su implementación en GPU, fueron reconsiderados para una implementación más eficiente, como por ejemplo los de simplificación de mallas [27] y los de *tessellation* de superficies de subdivisión y superficies paramétricas [13, 22]. El *geometry shader* procesa primitivas (puntos, segmentos de líneas o triángulos) y el número de primitivas en la salida puede ser distinto del número de primitivas en la entrada. Además, para cada triángulo se puede acceder a la información de los tres triángulos vecinos. La principal restricción es el número de valores de salida (actual-

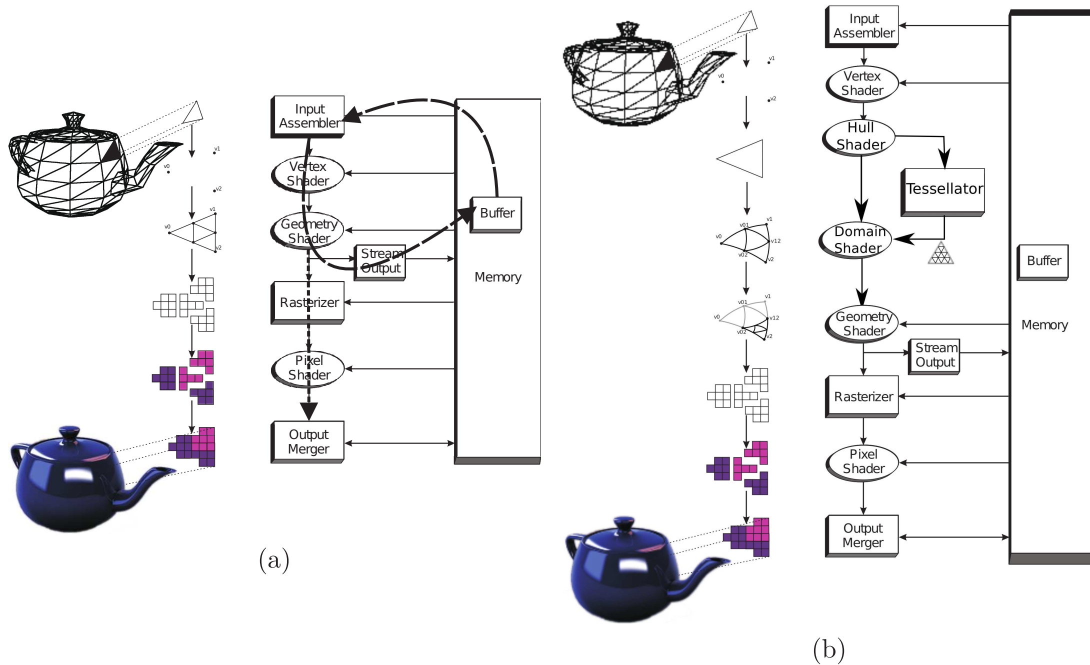


Figura 5.1: Estructura de una GPU según: (a)Direct3D10 y (b) Direct3D11.

mente 1024 valores de 32 bits). Los resultados intermedios producidos por el *vertex shader* o por el *geometry shader* se pueden enviar hacia atrás en la estructura de la GPU, permitiendo el procesamiento iterativo, o se pueden mandar directamente a la etapa de generación de los píxeles de la imagen.

Direct3D11 introdujo tres nuevas etapas para poder realizar la *tessellation* (ver la Figura 5.1b): el *hull shader*, la unidad *tessellator* y el *domain shader*. Los dos *shaders* son programables, mientras que la unidad de *tessellation* es una etapa con una función fija. Además de estas tres nuevas etapas, se añadió el *patch* como nueva primitiva. El *hull shader* tiene dos funciones: calcular los factores de formación de polígonos que se pasarán a la unidad de *tessellation* y, opcionalmente, realizar una conversión de los puntos de control de una representación a otra. La unidad de *tessellation* recibe los factores y bordes calculados en la etapa anterior y produce una serie de pesos correspondientes a las primitivas declaradas en esa etapa previa (líneas, triángulos o cuadriláteros). Finalmente, el *domain shader* utiliza las coordenadas paramétricas producidas en el constructor de polígonos y los puntos de control producidos en el procesador de superficie para evaluar cada uno de los puntos de la superficie.

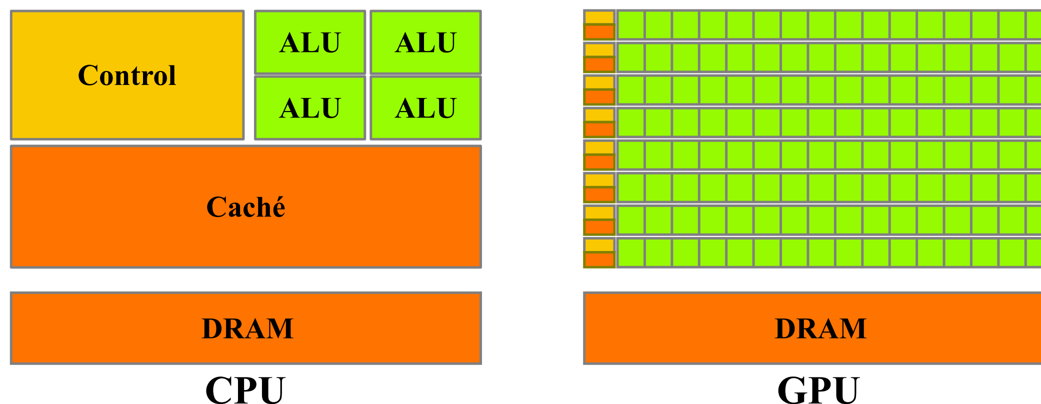


Figura 5.2: Esquemas de las diferentes filosofías de diseño de CPUs y GPUs.

5.1.2. Comparación entre CPU y GPU

La gran diferencia de rendimiento entre los procesadores multinúcleo de propósito general y las GPUs radica en la diferencia de filosofía de diseño de los dos sistemas (ver la Figura 5.2). El diseño de una CPU está optimizado para el código secuencial. Así utiliza una sofisticada lógica de control para poder ejecutar tareas en paralelo o incluso fuera de orden manteniendo la apariencia de una ejecución secuencial. Además emplea grandes memorias caché para reducir la latencia de acceso a las instrucciones y a los datos. Ni esta lógica de control ni las memorias caché contribuyen a la mejora de la velocidad máxima de cálculo. Otro punto importante es el ancho de banda de la memoria. Las GPUs han estado operando con un ancho de banda 10 veces superior al de las CPUs. Esto es debido a que es difícil mejorar el ancho de banda de los procesadores de propósito general. La razón de esta dificultad es que las CPUs tienen que satisfacer una serie de requerimientos heredados de como el software, las aplicaciones y los dispositivos de entrada/salida esperan que funcione el acceso a memoria. Sin embargo, los diseñadores de GPUs pueden alcanzar anchos de banda más grandes ya que trabajan con modelos de memoria más simples y con menos restricciones.

La filosofía de diseño de las GPUs está supeditada al rápido crecimiento de la industria de los videojuegos, la cual ejerce una gran presión económica para conseguir un enorme número de cálculos en punto flotante por cada fotograma de vídeo en los juegos avanzados. Esta demanda hace que los fabricantes de GPUs busquen la forma de maximizar el área de chip y el gasto de potencia dedicados a los cálculos en punto flotante. La solución imperante hoy en día es optimizar la ejecución a través del uso de un número muy grande de tareas. El hardware se beneficia del gran número de

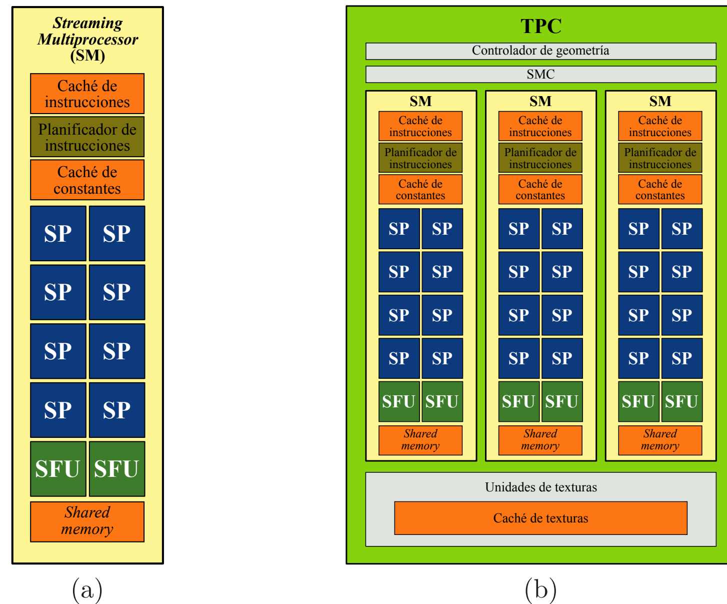


Figura 5.3: Arquitectura Tesla: (a)*Streaming Multiprocessor* (SM) y (b) *Texture Processor Cluster* (TPC).

tareas en ejecución, ya que puede pasar a ejecutar otra tarea cuando alguna de ellas esté esperando por accesos a memoria de gran latencia, minimizando de esta forma la lógica de control requerida para cada tarea en ejecución. También se utilizan pequeñas memorias locales como ayuda a los requerimientos de un gran ancho de banda de esas aplicaciones, de forma que cuando varias tareas requieren los mismos datos de memoria no necesitan acceder todas a la memoria global. El resultado es que una gran parte del área del chip se dedica a los cálculos en punto flotante.

Hay que resaltar que las GPUs están diseñadas para cálculo numérico y no realizan de una manera eficaz algunas tareas para las que las CPUs están mejor diseñadas. Por tanto, una implementación eficiente de una aplicación debería usar la CPU como procesador central y las GPUs como un coprocesador, ejecutando las partes de cálculo numérico intensivo en ellas.

5.2. Arquitectura NVIDIA Tesla de la GPU

La arquitectura Tesla está basada en un conjunto escalable de multiprocesadores SM (*Streaming Multiprocessors*) (ver Figura 5.3a). Cada SM cuenta con una serie de procesadores SP (*Streaming Processors*), concretamente ocho en esta arquitec-

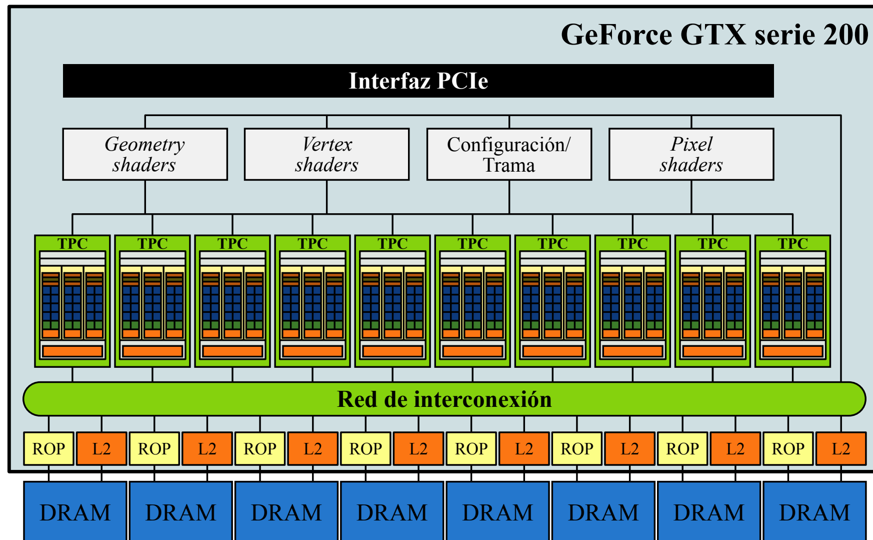


Figura 5.4: Estructura de la serie 200 de la GPU GeForce GTX.

tura, y dos unidades de funciones especiales (SFU) encargadas de la ejecución de instrucciones relacionadas con las funciones trascendentales como seno o coseno. Adicionalmente, cada SM contiene una pequeña caché de instrucciones, una caché de constantes y una memoria de lectura/escritura de 16 KB de baja latencia llamada *shared memory*. Los SM están agrupados en TPCs (*Texture Processor Cluster*) (ver Figura 5.3b). El número de SMs en cada TPC depende del modelo de GPU. Cada TPC tiene una caché de texturas de solo lectura que es compartida por todos los SMs de ese TPC. En este capítulo se comentará más adelante que cuando un programa CUDA ejecuta código en la GPU, los bloques independientes de tareas, en los que el programador ha dividido el problema, se distribuyen entre los SMs siguiendo un criterio de capacidad de ejecución. Los bloques se ejecutarán en cualquier orden en cualquier SM que esté disponible en el sistema.

El sistema tiene cuatro tipos de memoria: memoria compartida (*shared memory*), memoria global, memoria de constantes y memoria de texturas. Mientras que la *shared memory* se encuentra en cada SM, los otros cuatro tipos de memoria residen en la memoria del dispositivo, siendo común a cualquier SM (ver Figura 5.4).

Debido a su ubicación la **shared memory** es más rápida que los otros tipos. Su información solo es accesible por las tareas del bloque que se ejecuta en el SM correspondiente y su tiempo de vida es el del bloque. Esta memoria está dividida en varios módulos, de forma que las palabras adyacentes de 4 bytes se almacenan en módulos adyacentes [134]. Una lectura de varias palabras que residan en módulos

distintos se puede realizar simultáneamente, pero cuando los accesos coinciden en el mismo módulo se tienen que realizar en serie. Teóricamente, si non hay conflictos de acceso, esta memoria es tan rápida como los registros.

La **memoria global** reside en la memoria del dispositivo y es compartida por todos los SMs. Es una memoria de mayor latencia que la memoria de la CPU, pero este inconveniente se ve compensado en aplicaciones masivamente paralelas por su elevado ancho de banda.

La **memoria de constantes** se encuentra en la memoria del dispositivo y tiene asociada una caché en los SMs. En este espacio residen las variables declaradas como constantes, que tendrán un tiempo de vida igual a la duración de la aplicación y son accesibles por todas las tareas y por la CPU. Las peticiones de memoria son servidas desde la caché si se produce un acierto caché, o desde la memoria del dispositivo en caso contrario.

Finalmente, también en la memoria del dispositivo se encuentra la **memoria de texturas**, asociada con otra caché en los SMs. Es una memoria de solo lectura que presenta algunos beneficios frente a la memoria global o a la memoria de constantes. Por ejemplo, puede lograrse un mayor ancho de banda cuando los patrones de acceso a memoria no son adecuados para memoria global, pero sí tienen localidad espacial. Además, los cálculos de direccionamiento se realizan por unidades dedicadas. Igual que en la memoria de constantes, un acceso a la memoria de texturas se sirve desde la caché de texturas en caso de acierto en ésta, o desde la memoria del dispositivo en caso contrario.

Los chips que implementan la arquitectura Tesla son altamente multitarea, pudiendo ejecutar miles de tareas por aplicación. Por ejemplo, el chip G80 con 128 núcleos organizados en 16 multiprocesadores puede ejecutar hasta 768 tareas por multiprocesador, lo cual suma algo más de 12000 tareas, y el GTX 295 llega a las 1024 tareas por multiprocesador superando las 30000 para el chip [67]. Cada multiprocesador crea, manipula y ejecuta tareas concurrentemente sin conste adicional en la planificación. Además las tareas concurrentes pueden ser sincronizadas en un punto usando una única instrucción del multiprocesador. El bajo coste computacional en la creación de tareas, sin sobrecoste en su planificación, junto con la rápida sincronización mediante barreras hacen de esta arquitectura un sistema paralelo de grano fino muy eficiente.

Nuestro trabajo se ha realizado sobre GPUs de arquitectura Tesla, aunque desde el año 2009, como se ha comentado, NVIDIA ya tiene en el mercado una nueva

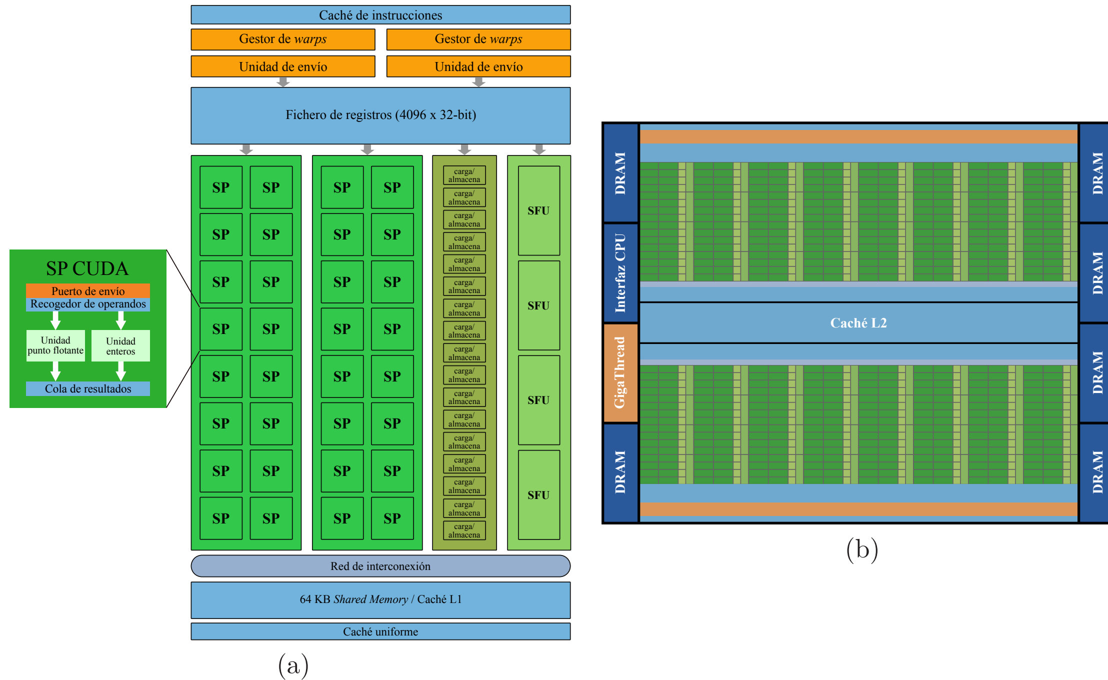


Figura 5.5: Arquitectura Fermi: (a) SM (b) Esquema.

arquitectura de GPUs: **Fermi** [87]. En la Figura 5.5 podemos ver alguna de las diferencias de la arquitectura Fermi respecto de la Tesla. Por ejemplo, los SMs están compuestos por 32 núcleos y tienen cuatro unidades SFU. Además poseen un espacio de memoria de 64 KB configurable, es decir, se pueden utilizar 48 KB como *shared memory* y 16 KB como caché de primer nivel, o, al contrario, 16 KB como *shared memory* y 48 KB como caché. Por otro lado, cada SM posee 16 unidades de carga/almacenamiento, que permiten que las direcciones de destino y origen de los datos puedan ser calculadas simultáneamente para 16 tareas. El sistema tiene también una caché de segundo nivel común para todos los SMS (ver Figura 5.5b). Por otra parte, esta arquitectura mejora el rendimiento de las operaciones en punto flotante de doble precisión, está optimizada para OpenCL y es la primera GPU que puede utilizar memorias ECC.

5.3. CUDA

El modelo de programación de CUDA conduce a dividir el problema en subproblemas que se puedan resolver en paralelo, y estos subproblemas, a su vez, en piezas

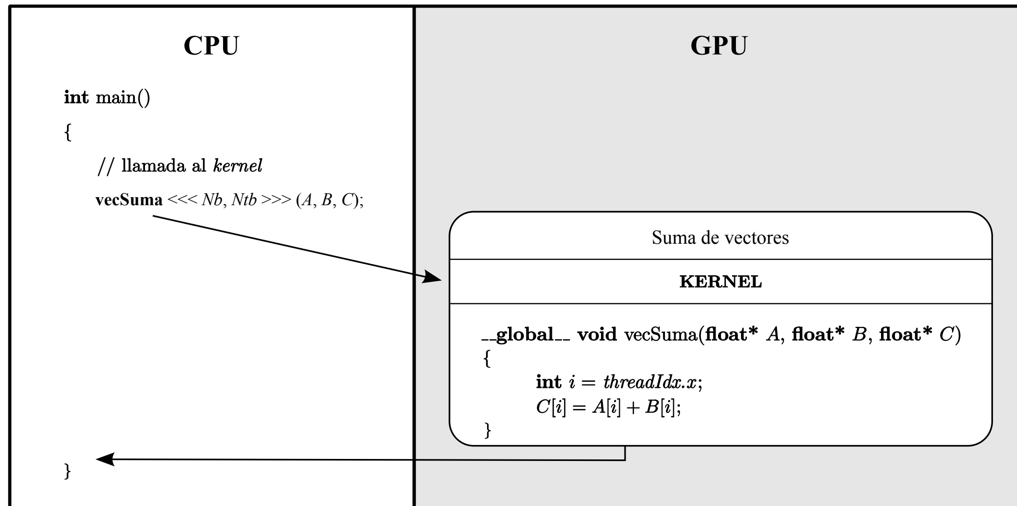


Figura 5.6: Un *kernel* en CUDA.

más pequeñas que puedan solucionarse cooperativamente. Así, un programa CUDA compilado podrá ejecutarse con cualquier número de núcleos, y solo el sistema en tiempo de ejecución necesitará conocer el número de núcleos físicos.

Para aplicaciones CUDA C, la GPU actúa como coprocesador que ejecuta funciones C, llamadas *kernels*, que son programas invocados por la CPU, que actúa como procesador central (ver el ejemplo de la Figura 5.6). Estas funciones se ejecutan Nt veces en paralelo, es decir, se generan Nt tareas CUDA diferentes. De esta forma, un *kernel* crea muchas tareas paralelas organizadas en una malla de Nb bloques de tareas. Los Nb bloques son conjuntos de Ntb tareas, todos de igual tamaño, siendo el número total de tareas igual al producto del número de tareas por bloque y el número de bloques, $Nt = Nb \times Ntb$. Los bloques se organizan en una malla (*grid*) unidimensional o bidimensional como se muestra en la Figura 5.7.

Los bloques de tareas han de poderse ejecutar independientemente: debe ser posible ejecutarlos en cualquier orden, en paralelo o secuencialmente. Esta independencia permite planificar la distribución y la ejecución de los bloques entre cualquier número de núcleos, lo que facilita la escritura de códigos escalables. Cuando un programa CUDA en la CPU invoca a un *kernel*, los bloques de la malla se enumeran y distribuyen entre los SMs, siendo ejecutadas las tareas de un bloque en los SPs de un SM. Cuando estas tareas terminan se lanzan nuevos bloques en los SMs libres.

Para poder manejar y ejecutar cientos de tareas computando diferentes programas, el multiprocesador Tesla usa el modelo de arquitectura una–instrucción,

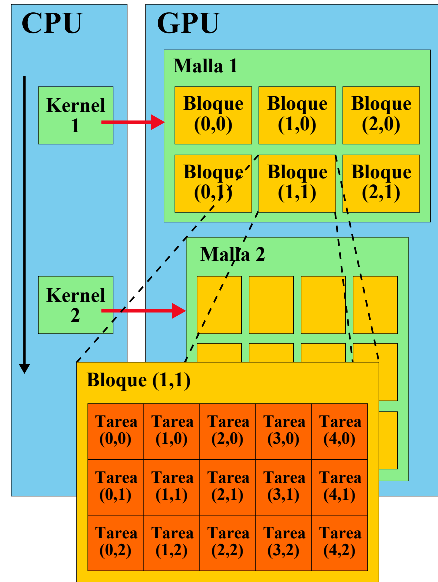


Figura 5.7: Jerarquía de tareas.

múltiples–tareas (SIMT, *single–instruction, multiple–thread*). Cuando un multiprocesador recibe uno o más bloques, crea, maneja, planifica y ejecuta tareas en grupos de 32 tareas paralelas llamados *warps*. La manera en que un bloque es dividido en *warps* siempre es la misma: cada *warp* contiene tareas con identificadores consecutivos y crecientes, con el primer *warp* conteniendo la tarea 0. Las tareas que componen un *warp* son del mismo tipo y comienzan juntas en la misma dirección del programa, pero después son ejecutadas independientemente. La máxima eficiencia se obtiene cuando las 32 tareas siguen el mismo camino de ejecución. Más adelante se comentará que esta eficiencia se puede ver degradada en la ejecución de los saltos condicionales.

La arquitectura SIMT es similar al modelo una–instrucción, múltiples–datos (SIMD, *single–instruction, multiple–data*). La diferencia es que SIMT aplica una instrucción a múltiples tareas independientes en paralelo, no solo a múltiples datos. En contraste con las arquitecturas vectoriales SIMD, SIMT les permite a los programadores no solo escribir código paralelo a nivel de tarea para tareas independientes sino que también pueden escribir código paralelo a nivel de datos para tareas coordinadas. Un programador puede escribir un código correcto ignorando los atributos del modelo SIMT, como los *warps*; sin embargo, se pueden obtener mejoras importantes de rendimiento escribiendo códigos en los cuales las tareas de un *warp* diverjan en raras ocasiones.

El número de bloques que puede procesar un SM a la vez depende de los recursos

compartidos utilizados por cada tarea; es decir, del número de registros y de la cantidad de *shared memory* usados por el *kernel*, ya que el número de registros y la *shared memory* de un SM se reparten entre las tareas de los bloques.

En general, se requiere un gran número de tareas para ocupar todos los núcleos. La arquitectura permite que las tareas dentro de un bloque puedan cooperar eficientemente entre ellas compartiendo datos a través de la *shared memory*. Para coordinar los accesos a memoria, la sincronización de la ejecución de las tareas se puede llevar a cabo mediante la función intrínseca **__syncthreads( )**. Esta función actúa como una barrera realizando una sincronización entre todos los *warps* de un bloque [132].

Para optimizar el rendimiento de una implementación CUDA las distintas estrategias se basan en tres ideas básicas: maximizar la ejecución en paralelo, optimizar el uso de la memoria y optimizar el uso de las instrucciones.

Para maximizar la ejecución paralela, dado un número total de tareas por malla, *Nt*, el número de tareas por bloque, *Ntb*, equivalentemente el número de bloques, *Nb*, debería ser escogido con el fin de maximizar la utilización de los recursos de cálculo disponibles. Esto significa que debería haber al menos tantos bloques como SMs en la GPU. La máxima eficiencia se conseguiría cuando todas las tareas de un *warp* ejecuten una instrucción común y simultáneamente. Por ello, *Ntb* tendría que ser elegido como un múltiplo del tamaño de un *warp* para evitar la infrautilización de los recursos disponibles por la existencia de *warps* incompletos. En las GPUs actuales *Ntb* debería tener un valor entre 32 y 512, y, en general, ser un múltiplo de 64 [88]. Este número se selecciona como un compromiso entre tener muchas tareas por bloque, para maximizar la utilización de los recursos de cálculo, y el número de registros disponibles por tarea.

Para optimizar el uso de la memoria se debe tratar de minimizar la transferencia de datos entre la GPU y la CPU ya que el ancho de banda entre la memoria de la GPU y la del sistema es mucho menor que el ancho de banda en las transferencias entre zonas de la memoria de la GPU. También, debido al coste asociado a cada transferencia, agrupar muchas transferencias pequeñas en una grande siempre ofrece mejores rendimientos que realizar cada transferencia por separado. De todos modos, el ancho de banda efectivo de cada espacio de memoria depende significativamente del patrón de acceso a memoria. A continuación se describe brevemente que sucede con los accesos a cada tipo de memoria de la GPU.

Respecto a la memoria global, no tiene una caché asociada en arquitectura Tesla,

con lo cual es muy importante seguir el patrón de acceso correcto con coalescencia (agrupamiento) para obtener el máximo ancho de banda, especialmente dado lo costoso que resultan los accesos a este tipo de memoria. Esta memoria es accesible mediante transacciones de 32, 64 o 128 bytes, las cuales deberían estar alineadas para obtener mejor rendimiento. Cuando un *warp* ejecuta una instrucción que accede a memoria global agrupa los accesos de un *halfwarp* (las 16 primeras tareas de un *warp* o las 16 últimas) en tantas transacciones como sean necesarias para obtener los datos que requiere ese *halfwarp*. El tamaño de las transacciones es función del tamaño de la palabra accedida y del grado de coalescencia de las transacciones. El grado de coalescencia depende de las características de la GPU, pero, básicamente, será mayor cuanto mayor sea el número de referencias de un *halfwarp* que se estén en el segmento de memoria al que se va a acceder en una transacción. Esto quiere decir que el grado sería máximo si las tareas de un *halfwarp* accedieran a palabras de posiciones contiguas en memoria. Cabe destacar que cuanto mayor sea el grado de no–coalescencia, mayor será el número de transacciones por acceso a memoria y, por tanto, mayor cantidad de bytes serán desperdiciados.

El espacio de memoria constante está ubicado también en la memoria del dispositivo, pero tiene una memoria caché asociada en el multiprocesador, con lo cual las peticiones de memoria son servidas desde esta caché en caso de acierto, o desde la memoria del dispositivo en caso contrario. Un acceso a memoria constante para un *warp* se divide en dos accesos, uno para cada medio *warp*. Esta memoria se comporta bien para accesos en los que un *warp* accede a la misma dirección para todas sus tareas.

Igual que la memoria constante, la memoria de texturas reside en la memoria del dispositivo y tiene una memoria caché asociada en cada TPC, sirviéndose el dato accedido desde esta caché cuando hay un acierto o desde la memoria del dispositivo en caso contrario. Una de las característica más importantes de esta memoria es que el almacenamiento de los datos en la caché de texturas intenta respetar al máximo la localidad espacial. Leer de memoria de texturas tiene algunos beneficios respecto de hacerlo de memoria global o de memoria de constantes, entre los que destacamos:

- Si las lecturas de memoria no siguen los patrones que deben cumplir los accesos a memoria global o constante, se puede lograr más ancho de banda si se logra localidad espacial en la búsqueda de texturas.
- Los cálculos de direccionamiento son realizados fuera del *kernel* por unidades especializadas.

Como se explicará más adelante, nuestra propuesta utiliza la memoria de texturas para incrementar el rendimiento aprovechando los beneficios comentados. Debido a que el uso de la memoria de texturas es menos habitual, a continuación se va a detallar un poco más como es su empleo.

Las funciones que proporciona CUDA para leer de memoria de texturas se conocen como *texture fetches*. En la Figura 5.8 vemos un ejemplo de una función de este tipo: **tex1Dfetch( )**. El primer parámetro y más importante de estas funciones es la *texture reference*, el cual define la parte de memoria global que se busca que debe pertenecer a una región de memoria llamada *textura* (*texRefVertexs* en la Figura 5.8). Una textura puede haber sido reservada como memoria lineal o como una estructura matricial de CUDA. Antes de que un *kernel* pueda usar una *texture reference*, esta debe estar vinculada a una textura (una región de memoria global). En otras palabras, el programador reserva una zona de memoria global y luego ese espacio se vincula a una *texture reference* para poder ser accedida desde el *kernel*. En el ejemplo de la Figura 5.8 se reserva con **cudaMalloc( )** un espacio de memoria global para el vector *vertexs_d* en el que se copian los datos del vector *vertexs* de la CPU mediante la función **cudaMemcpy( )**. A continuación, se vincula el vector *vertexs_d* con la textura *texRefVertexs* con la función **cudaBindTexture( )**. Una vez que haya finalizado la ejecución del *kernel* termina el uso de la textura, ya que los datos que pueda escribir el *kernel* irán a la memoria global. Conviene recordar que la memoria de texturas es de solo lectura. Finalmente, se puede desvincular la textura y liberar el espacio de memoria correspondiente (en el ejemplo de la Figura 5.8 se realiza con las funciones **cudaUnbindTexture( )** y **cudaFree( )**).

El último tipo de memoria, la *shared memory*, se encuentra integrada en el SM y está dividida en módulos de igual tamaño, llamados *bancos*, a los que se puede acceder simultáneamente. Debido a su ubicación en la GPU, los accesos a la *shared memory* son más rápidos que los que se realizan a la memoria del dispositivo. De hecho, para todas las tareas de un *warp*, acceder a la *shared memory* teóricamente es casi tan rápido como acceder a un registro siempre y cuando no existan conflictos de bancos entre las tareas. Un dato almacenado en esta memoria tiene las siguientes características:

- Reside en el espacio de *shared memory* de un bloque de tareas.
- Tiene el tiempo de vida de un bloque.
- Es solo accesible por las tareas de un bloque.

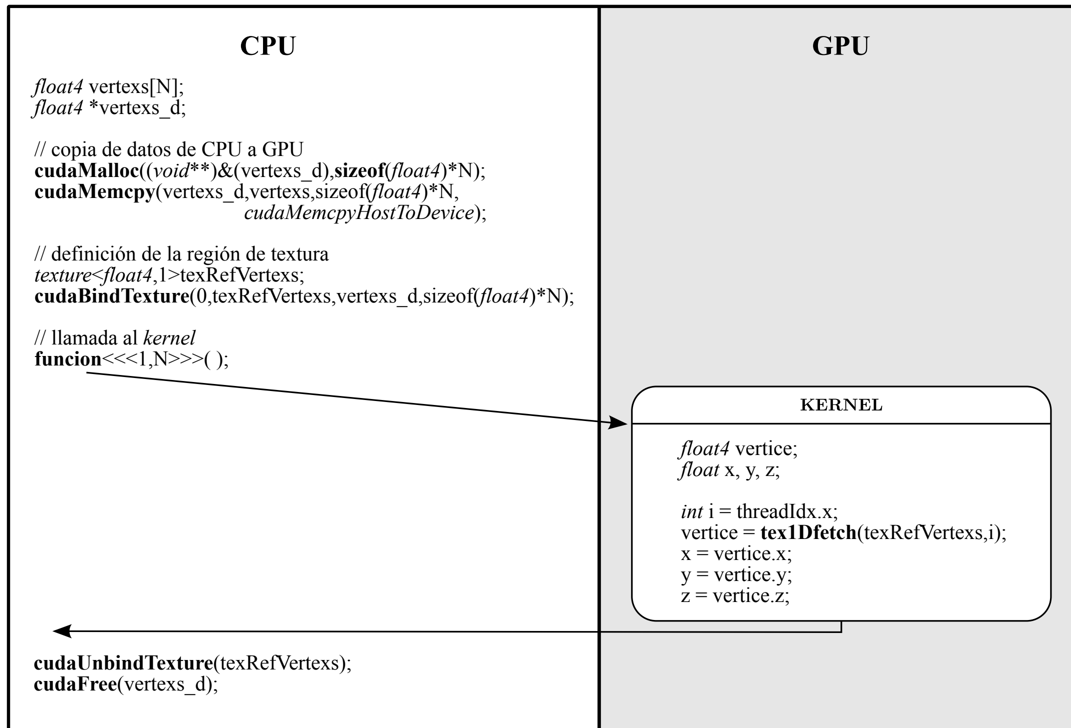


Figura 5.8: Uso de la memoria de texturas.

Finalmente, en la optimización del rendimiento de una aplicación en CUDA debemos tener en cuenta el efecto de las instrucciones de control de flujo (**if**, **switch**, **do**, **for**, **while**). Estas instrucciones pueden impactar significativamente en el rendimiento causando que tareas del mismo *warp* sigan diferentes caminos de ejecución. Por ejemplo, cuando la instrucción a ejecutar es un **if** o un **switch** la condición a evaluar puede ser verdadera, con lo que la siguiente instrucción será la instrucción posterior al salto, o falsa, siendo en este caso ejecutada la primera instrucción posterior al bloque condicional. Si todas las tareas de un *warp* tienen el mismo resultado al evaluar la condición de salto, el SM ejecutará una única instrucción para las 32 tareas. Si no es así, los diferentes caminos han de ser ejecutados en serie, incrementando el número total de instrucciones ejecutadas por este *warp*. Cuando todos los caminos de ejecución se hayan completado, las tareas convergen al mismo camino. La divergencia de salto solo ocurre dentro de un *warp*, es decir, un salto en un bloque de 64 tareas donde el primer *warp* evalúa el salto como verdadero y el segundo como falso no sufrirá ningún tipo de ejecución en serie, pues todos las tareas de cada *warp* compartirán el mismo camino.

5.4. Proyección GPU del MCR con CUDA

En esta sección se presenta nuestra propuesta para la proyección del MCR sobre una GPU usando CUDA. Una implementación directa del algoritmo utilizando CUDA no produce el rendimiento esperado. Debido a esto, nuestra propuesta modifica la estructura original del algoritmo para explotar eficientemente las capacidades hardware de la GPU. Este objetivo lo logramos mediante tres técnicas diferentes:

- la división de la escena en subescenas,
- la utilización de una versión simplificada de la malla de elementos basada en un criterio de planitud,
- la organización eficiente de las tareas.

A continuación se van a explicar con detalle estas tres técnicas.

5.4.1. Partición de la escena

En el Capítulo 4 se empleó la partición de la escena para poder implementar el MCR en un sistema distribuido. En este capítulo se va a utilizar una estrategia similar en la proyección del MCR sobre GPU con dos objetivos fundamentales:

1. Acelerar el cálculo de la intersección de los rayos con los elementos, sin tener que usar estructuras como el *kd–tree*.
2. Aprovechar la localidad de los datos para optimizar el uso de la memoria de la GPU.

Nuestra propuesta está basada en la partición de la escena en Nd subescenas disjuntas con una estructura convexa. Esta partición es utilizada como único procedimiento para acelerar el chequeo de la intersección de los rayos con los elementos, sin necesidad de utilizar una estructura *kd–tree* [53]. Esta estructura es ampliamente utilizada en las implementaciones de CPU debido a su simplicidad (tanto conceptualmente como por su implementación), versatilidad y porque es una estructura poderosa para clasificación y ordenación [56, 116]. En cuanto a la GPU, también han aparecido implementaciones de *kd–tree* que compiten en cuanto a rendimiento con las de CPU. Debido a las características de la arquitectura de la GPU, estas

implementaciones han tenido que resolver problemas que no se tienen en la CPU, como la imposibilidad de utilizar recursividad. Por ello algunas utilizan un esquema convencional de pila para el recorrido de los nodos del árbol, con el fin de conseguir una implementación CUDA eficiente mediante un almacenamiento de una estructura matricial en la memoria local de cada tarea [135]. Otras evitan la utilización de estas pilas, aunque requieren calcular información adicional en la elaboración de la estructura del *kd–tree* y cálculos extra en el recorrido del árbol [55, 100]. En nuestro caso, hemos optado por emplear nuestro método de división convexo de la escena como única guía para acelerar las comprobaciones de las intersecciones entre los rayos y los elementos. Una de las razones de esto es que la estructura convexa permite una simplificación del proceso de disparo de rayos. Esto es debido a que cuando se busca el elemento más cercano que es cruzado por el rayo, si se encuentra en la misma subescena que contiene al elemento origen del rayo ya no es necesario seguir comprobando el resto de subescenas. En las particiones no convexas habría que examinar todas las subescenas, como se comentó en la Subsección 4.2.1. De esta forma se reduce el número de elementos que tienen que ser analizados y se minimiza el número de accesos a memoria.

Por otro lado, para alcanzar un rendimiento computacional elevado en la arquitectura de una GPU es crítico un balanceo apropiado del uso de los recursos compartidos. Una estrategia básica es mejorar el reuso de los datos y reducir los accesos a la memoria global. La estrategia de división de la escena permite optimizar el patrón de accesos a la memoria global, ya que facilita los accesos coalescentes, es decir, las referencias a memoria cercanas son agrupadas en un único acceso con la consiguiente reducción en el número de accesos a la memoria global. En el caso de la utilización de una estructura *kd–tree* para cada subescena, debido a las limitaciones de almacenamiento de la GPU, la mayor parte de esta estructura se almacena en la memoria global y es ineficiente en la utilización de los accesos coalescentes.

Por otro lado, el empleo de subescenas también incrementa la reutilización de los datos. Teniendo en cuenta esto, los valores geométricos, que no se modifican durante el proceso, en nuestra propuesta son almacenados en la memoria de texturas. De esta forma, durante la ejecución en cada subescena estos datos geométricos pueden caber en la memoria caché de texturas de cada multiprocesador, reduciendo los accesos a memoria global y mejorando el rendimiento.

En nuestra implementación se han usado los dos métodos de división ya comentados: división uniforme (ver Sección 2.2) y división no uniforme (ver Sección 3.1) con la utilización de los parámetros explicados en la Subsección 4.2.1. Además se ha

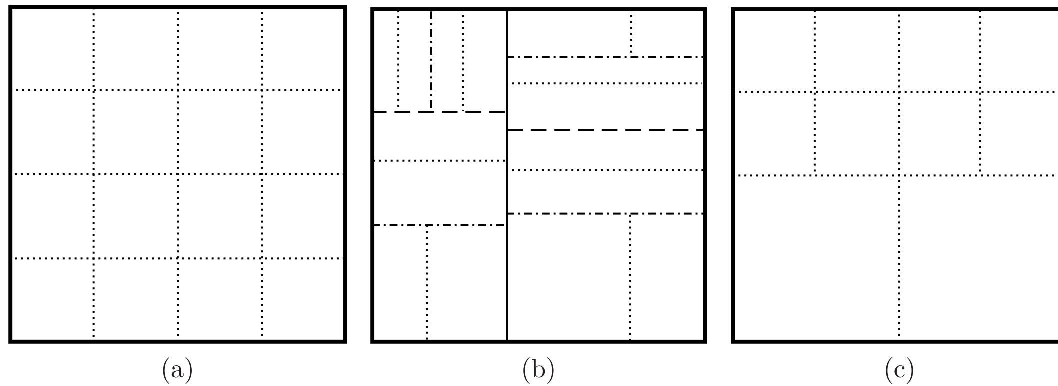


Figura 5.9: Métodos de división de la escena: (a) uniforme, (b) no uniforme, (c) híbrido.

desarrollado un tercer método, que llamamos división híbrida, que combina las dos estrategias anteriores. Con este método se pretende conseguir el balanceo de la carga computacional del método no uniforme, manteniendo cierta simplicidad del método uniforme. Cuando se habla de simplicidad se hace referencia, fundamentalmente, a un número reducido de fronteras, tanto en el conjunto de la escena como en cada subescena particular, y a una simplificación del paso de rayos entre subescenas. A continuación se comenta detalladamente esta tercera técnica de división de la escena.

La estrategia de división híbrida consiste en una partición adaptativa de la escena usando como base el método de división uniforme. Inicialmente, para que la partición sea igual en las tres dimensiones, se divide la escena en ocho subescenas de igual tamaño según el método de división uniforme. En los siguientes pasos de división, solo algunas subescenas son divididas siguiendo de nuevo el método de división uniforme. El objetivo de este método es obtener un conjunto de subescenas con un valor similar del factor X que se use para medir el desbalanceo de la carga computacional, esto es, $X^d \approx \overline{X}$. Así, solo las subescenas con un valor $X^d > \overline{X}$ son divididas. De esta forma se combinan las ventajas de los dos métodos anteriores: potencial balanceo de la carga para la partición no uniforme y simplicidad para la división uniforme.

En la Figura 5.9 se muestra un ejemplo en dos dimensiones donde la escena es dividida en hasta 16 subescenas siguiendo las tres estrategias comentadas anteriormente. Específicamente para los métodos uniforme y no uniforme obtenemos 16 subescenas, mientras que la división híbrida genera, en este ejemplo, 10 subescenas. En la división uniforme (ver la Figura 5.9a) las 16 subescenas tiene la misma forma

y tamaño. Esto puede producir un grave problema de desbalanceo de carga computacional en aquellas escenas con una distribución muy irregular de objetos, como por ejemplo en escenas sin objetos en alguna zona. En el caso del ejemplo se supone que la parte superior de la escena tiene una mayor complejidad que la parte inferior. Para obtener un balanceo de la carga computacional se puede utilizar la estrategia de división no uniforme. Así en la Figura 5.9b la escena se divide primero en dos mitades con el valor más pequeño de $\sigma(P_{A,2})$ (tomando como factor X el área) que en este caso se corresponde con $P^{x}_{A,2}$. En el segundo paso cada partición se divide en otras dos subescenas siendo $P^{y}_{A,2}$ la mejor opción para las dos subescenas. El proceso continúa de esta forma hasta completar las 16 subescenas (los diferentes tipos de líneas en la Figura 5.9b representan las diferentes etapas del proceso de partición). La división híbrida también persigue balancear la carga computacional. Como se puede observar en la Figura 5.9c, la mayor complejidad de la parte superior de la escena conlleva a obtener una partición más fina en esta zona.

5.4.2. Simplificación de la malla de elementos

En esta subsección se presenta un nuevo método para minimizar el número de elementos que tienen que ser comprobados en el proceso de disparo de rayos. El método propuesto está basado en una utilización de una versión simplificada de la malla de elementos de la escena. Esta malla “gruesa” almacena la información asociada con una representación menos detallada de la escena. Como se verá en la sección de resultados, la utilización de esta técnica permite una mayor reducción en los tiempos de ejecución obtenidos en la GPU porque permite una explotación más eficiente de la memoria de texturas.

En los algoritmos de radiosidad, cuanto más detallada sea la escena mejores resultados de iluminación se obtienen. Incluso en zonas planas se consiguen mejores resultados cuando estas zonas son representadas mediante un gran número de elementos. Sin embargo, las escenas detalladas implican un gran número de operaciones para determinar la intersección entre objetos y rayos. La idea de nuestra propuesta es emplear una versión simplificada de las áreas planas para comprobar las intersecciones, mientras que mantenemos el detalle para el cálculo de la radiosidad.

Hay diferentes métodos para simplificar la malla de elementos, los cuales construyen una versión simplificada de la malla detallada original, mientras intentan preservar su apariencia [36]. En nuestra propuesta se emplea una técnica de simplificación restrictiva para mantener la calidad de la iluminación. Específicamente, la

simplificación solo se realiza en elementos vecinos que son aproximadamente coplanares. El proceso es el siguiente:

1. La escena detallada se reduce a una más simple de acuerdo a un criterio de planitud. Comenzando en un elemento, se chequean los elementos vecinos y se aplica la siguiente comprobación:

$$\|\overrightarrow{n_i} - \overrightarrow{n_j}\| < \delta \tag{5.1}$$

   donde $\overrightarrow{n_i}$ y $\overrightarrow{n_j}$ son las normales de los elementos y δ es un umbral definido por el usuario. Si la relación anterior se cumple, los elementos son unidos en una nueva superficie. Luego el proceso continúa comprobando la relación entre la nueva superficie y el elemento o superficie contiguo. Esta operación solo se realiza cuando la superficie resultante tiene una estructura convexa. Destacar que esto permite un chequeo más simple de la intersección entre un rayo y un objeto. Al final de este proceso, cada superficie "grande" resultante estará compuesta por un conjunto de superficies más finas que estarán en el mismo plano.

2. Los métodos de partición explicados en la Subsección 5.4.1 son aplicados a la escena simplificada. Por consiguiente, las subescenas resultantes tienen un número más pequeño de elementos que el que les correspondería según la representación original de la escena.

Nuestra propuesta de implementación del algoritmo de radiosidad emplea la versión simplificada de las subescenas para las comprobaciones de las intersecciones entre rayos y objetos. Para cada rayo que se dispara se determina la superficie "grande" más cercana que atraviesa. Después de esto, se calcula la superficie específica más fina que contiene el punto de intersección.

La simplificación de la malla mejora el rendimiento porque así la malla "gruesa" de elementos de cada subescena puede ser almacenada en la memoria caché de texturas de la GPU cuya latencia de acceso es similar a la de un acceso a un registro. Este almacenamiento puede ser eficientemente explotado debido a que las tareas de un mismo bloque, como se mostrará en la siguiente subsección, en general procesan rayos de la misma subescena.

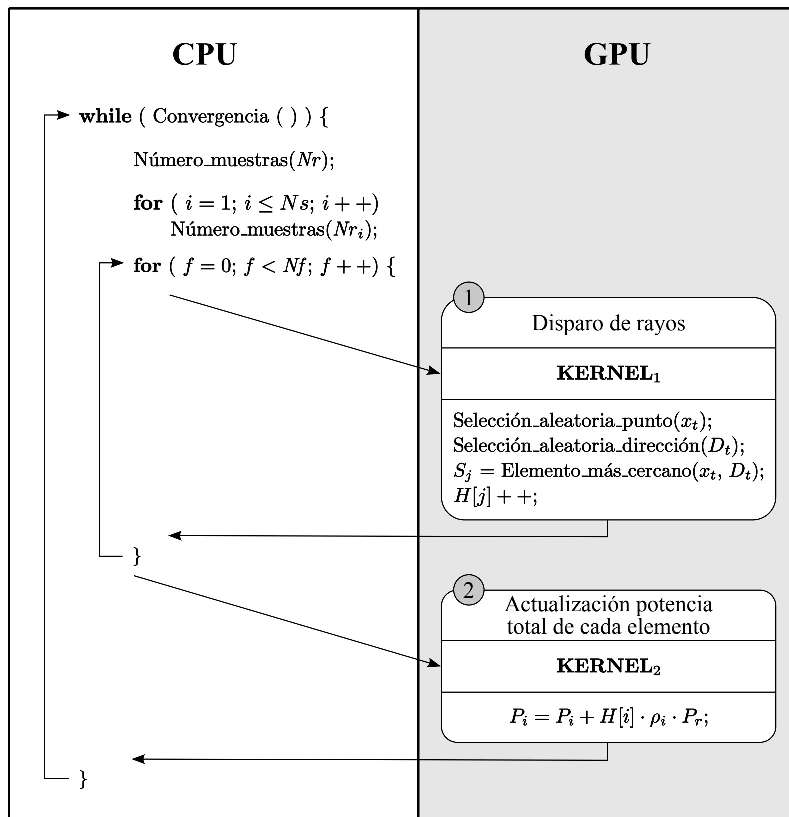


Figura 5.10: Implementación CUDA del algoritmo Monte Carlo para radiosidad.

5.4.3. Organización de las tareas

La disponibilidad de un número alto de tareas independientes para mantener ocupados los núcleos, junto con una explotación óptima de la jerarquía de memoria es una estrategia esencial para una eficiente implementación en CUDA. Para conseguir esto la clave de nuestra propuesta es realizar una planificación eficiente de las tareas asociada con la estrategia de partición de la escena.

La estructura de nuestro código CUDA se muestra en la Figura 5.10. Como ya se ha comentado, un programa en CUDA consiste en segmentos de código que son ejecutados en la CPU y otros segmentos de código que son ejecutados como funciones *kernel* en la GPU. Específicamente, nuestra propuesta está basada en dos *kernels*: $\mathbf{KERNEL}_1$ y $\mathbf{KERNEL}_2$. El proceso se repite iterativamente mientras no haya convergencia. En la CPU se calculan el número de muestras Nr y el número de rayos Nr_i que tienen que ser disparados desde el elemento S_i. La información sobre la radiosidad y la geometría de cada elemento se almacena en unas listas que se ordenan

de acuerdo a la partición de la escena que se ha comentado en la Subsección 5.4.1. De esta forma, los elementos de una misma subescena están en posiciones de memoria consecutivas. Posteriormente, el $\mathbf{KERNEL}_1$ se repite Nf etapas para optimizar la utilización de la memoria. En este *kernel* se ejecuta el disparo de rayos, es decir, el cálculo del objeto más cercano a un elemento para cada rayo que se dispara desde ese elemento. Finalmente, la potencia total P_i de cada elemento es incrementada en $\mathbf{KERNEL}_2$. A continuación se explicará detalladamente el algoritmo.

En la invocación de $\mathbf{KERNEL}_1$, que calcula el proceso de disparo de rayos, cada tarea procesa un rayo diferente porque nuestra implementación paralela en la GPU sigue una metodología de grano fino. El objetivo es la utilización de un gran número de tareas para mantener los núcleos ocupados. Además, no todos los elementos se tienen que disparar al mismo tiempo porque esto podría degradar la utilización de las memorias [23]. Específicamente, las memorias de textura, que tienen una caché asociada de 8 KB por multiprocesador, se usan para almacenar la información de los elementos que son disparados. Por consiguiente, $\mathbf{KERNEL}_1$ se invoca en Nf etapas consecutivas, donde Nf tiene el siguiente valor:

$$Nf = \max_{l=1,\cdots,Nd} \left\lceil \frac{Ns^l}{Nps} \right\rceil \tag{5.2}$$

siendo Nps el número de elementos por subescena que son agrupados en cada llamada a $\mathbf{KERNEL}_1$ y Ns^l el número de elementos en la subescena l.

En cuanto a la organización de los bloques de tareas en $\mathbf{KERNEL}_1$, en trabajos previos se optó por que los rayos disparados desde un elemento se procesaran en el mismo bloque, esto es, todas las tareas de un bloque correspondían a rayos cuyo origen es el mismo elemento [107, 110]. Sin embargo, se debe tener en cuenta que, en general, cada elemento va a disparar menos rayos en cada iteración, especialmente después de la segunda iteración. Como consecuencia, el número de rayos por bloque en el primer *kernel*, siguiendo la estrategia de agrupar solo rayos que salen de un único elemento, va a ser más pequeño que el tamaño de medio *warp*, comprometiendo seriamente la eficiencia de la implementación. De esta forma, en este trabajo se ha escogido una estrategia [109, 111] que va a perseguir la utilización de un valor óptimo del número de tareas por bloque Ntb, de manera que todos los núcleos de un multiprocesador estén ocupados cuando ejecutan las tareas de un *warp*. Para ello se permite que cada bloque procese rayos de distintos elementos, pero, normalmente, de la misma subescena con el objetivo de minimizar la divergencia del *warp*. Esto hace que los rayos de un mismo bloque puedan no partir desde el mismo elemento

y que, por tanto, no compartan esta información. Además, se ha decidido utilizar la memoria de texturas para almacenar la información de la subescena y de los elementos origen de los rayos (se debe recordar que la técnica de malla simplificada permite que las subescenas puedan estar almacenadas en la memoria de texturas). Esto posibilita que los rayos pueden acceder a esta información a través de la memoria caché de texturas de una manera muy rápida. Por consiguiente, el número de bloques se calcula con la siguiente ecuación:

$$Nb = \left\lceil \frac{Nr^f}{Ntb} \right\rceil \tag{5.3}$$

siendo Nr^f el número de rayos disparados en la etapa f, es decir,

$$Nr^f = \sum_{l=1}^{Nd} \sum_{i=B}^{E} Nr_i, \tag{5.4}$$

con $B = Is_0^l + Nps \cdot f$ y $E = Is_0^l + Nps \cdot (f+1)$, siendo Is_0^l el índice del primer elemento de la subescena l y Nr_i el número de rayos del elemento S_i.

El gran número de rayos que se intercambian entre subescenas adyacentes hace que su gestión sea una tarea crítica. Si un rayo no cruza un elemento en el subespacio en el que es disparado, este chocará contra una de las fronteras. Esta fronteras son modeladas como un conjunto de superficies virtuales compartidas por un subescena y su vecina adyacente. Hay indicar que en el caso de la partición uniforme, cada frontera corresponde a un único elemento virtual compartido por las dos subescenas adyacentes. Sin embargo, en las particiones no uniforme e híbrida la estructura adyacente a una subescena en una frontera podría ser más compleja y consistir de un conjunto de subescenas (ver las Figuras 5.9b y 5.9c).

Cada rayo disparado en una subescena que no alcanza a ningún elemento en esa subescena debe ser enviado a una única subescena adyacente. Esta subescena específica tiene que ser identificada y esto requiere determinar no solo la frontera de la subescena por la que sale el rayo, sino también el elemento virtual de esa frontera con el que choca. El código de la GPU que implementa la búsqueda de esta subescena adyacente a la que ha de ser enviado el rayo saliente, está caracterizado por utilizar lazos condicionales y, consecuentemente, por un grado de divergencia. Esta irregularidad, y por tanto la divergencia, es mayor para las particiones más complejas cuanto mayor sea el número de elementos virtuales por frontera que tienen que ser analizados.

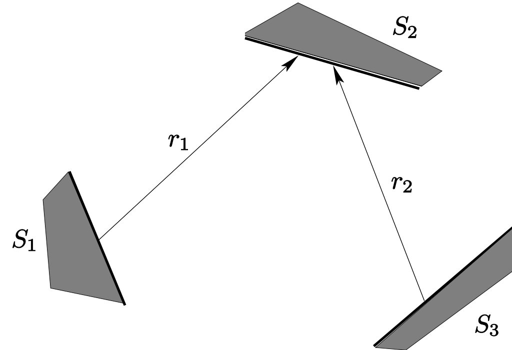


Figura 5.11: Múltiples rayos alcanzan el mismo elemento.

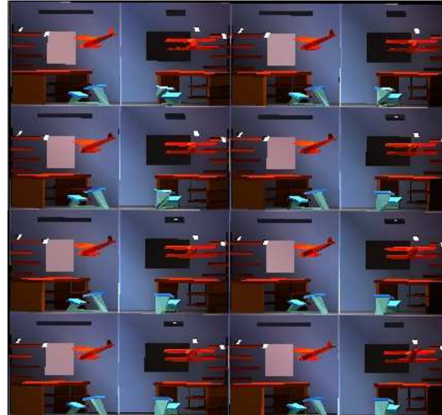
Otro elemento clave en la planificación de las tareas es la estrategia multitarea independiente que se ha utilizado. El procesamiento de cada rayo requiere accesos a memoria de solo lectura para procesar la geometría de la escena. Sin embargo, junto con estas operaciones de lectura hay una operación asociada de escritura para almacenar el incremento de potencia del objeto más cercano al elemento origen del rayo que es alcanzado por este, $\Delta P_j^{(k+1)}$. De esta manera, varias tareas podrían intentar incrementar la misma potencia al mismo tiempo. El resultado sería un conflicto debido a múltiples accesos de memoria para escritura. En la Figura 5.11 se muestra un ejemplo donde dos rayos llegan a un mismo elemento.

Para obtener un algoritmo eficiente de multitarea independiente, se ha diseñado una sencilla estrategia basada en la utilización de la función **atomicAdd( )**, una eficiente primitiva atómica de lectura–modificación–escritura proporcionada por CUDA. Así se ha empleado un vector auxiliar llamado $H(j)$ (ver la Figura 5.10) y cada elemento S_j que es seleccionado como el más cercano al origen del rayo que lo alcanza, incrementa la posición j de ese vector mediante la función **atomicAdd( )** en el **KERNEL**$_1$.

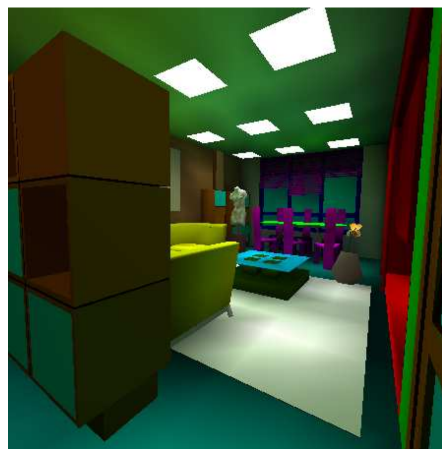
Para finalizar la iteración, la potencia de cada elemento se actualiza con el segundo *kernel*, **KERNEL**$_2$, utilizando el vector auxiliar $H(j)$, que indica el número de rayos que han alcanzado a cada elemento S_j (ver Figura 5.10).

5.5. Resultados experimentales

Para el estudio de nuestra segunda propuesta de paralelización hemos escogido las escenas *edificio*, *salón* y *estudio* (ver la Figura 5.12). Los resultados que serán



(a)



(b)

(c)

Figura 5.12: Escenas iluminadas: (a) *edificio*, (b) *salón* y (c) *estudio*.

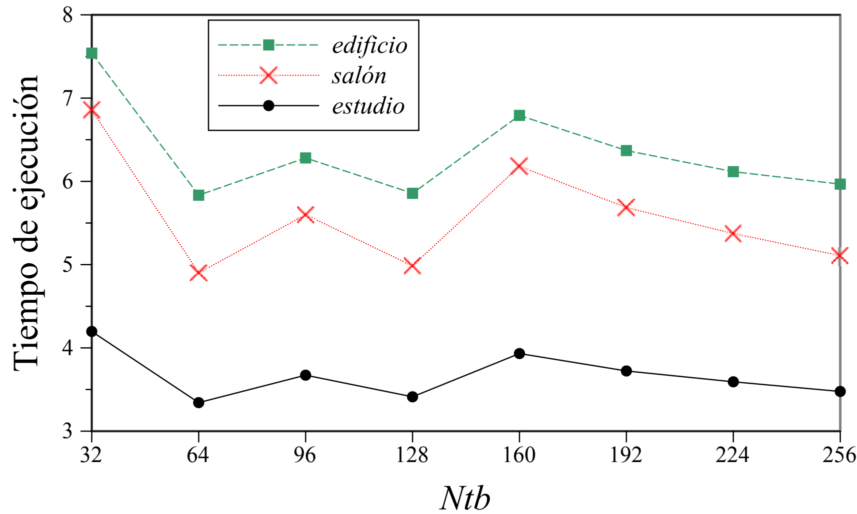


Figura 5.13: Número de tareas por bloque.

comentados a continuación han sido obtenidos en un sistema Intel Core 2 6600 a 2,40 GHz con una tarjeta gráfica NVIDIA GeForce GTX 295.

En primer lugar se ha analizado el valor óptimo del número de tareas por bloque, *Ntb*. La Figura 5.13 muestra los tiempos de ejecución medidos en segundos, para una división uniforme con 256 subescenas, siendo *Ntb* un múltiplo de 32 [88]. En nuestra implementación los mejores resultados han sido obtenidos con un valor bajo de $Ntb = 64$. Similares resultados se han obtenido con las otras estrategias de división. Como cada tarea utiliza un gran número de registros y bastante espacio de memoria de texturas el número de tareas por bloque debe ser reducido. Este valor de $Ntb = 64$ será el utilizado en el resto de pruebas realizadas.

Por otro lado se ha analizado la influencia del valor de *Nps* en el rendimiento de nuestra implementación. En la Figura 5.14 se muestra la variación en el tiempo de ejecución en segundos para la división uniforme con $Nd = 64$. Resultados semejantes se han obtenido para otros valores de *Nd*. Puede verse que para las escenas con menos elementos y menos luces no es necesario limitar el número de elementos que se disparan por *kernel*. Sin embargo, para la escena *edificio* sí se encuentra que hay un mínimo cuando se utiliza $Nps = 64$. Esto es debido a que esta escena consume muchos más recursos de memoria, con lo cual es más conveniente limitarlos mediante el valor de *Nps*, a costa de invocar más veces $\textbf{KERNEL}_1$. Teniendo en cuenta esto, en los siguientes estudios solo se ha utilizado el valor de *Nps* para la escena *edificio*, fijándolo a un valor de 64.

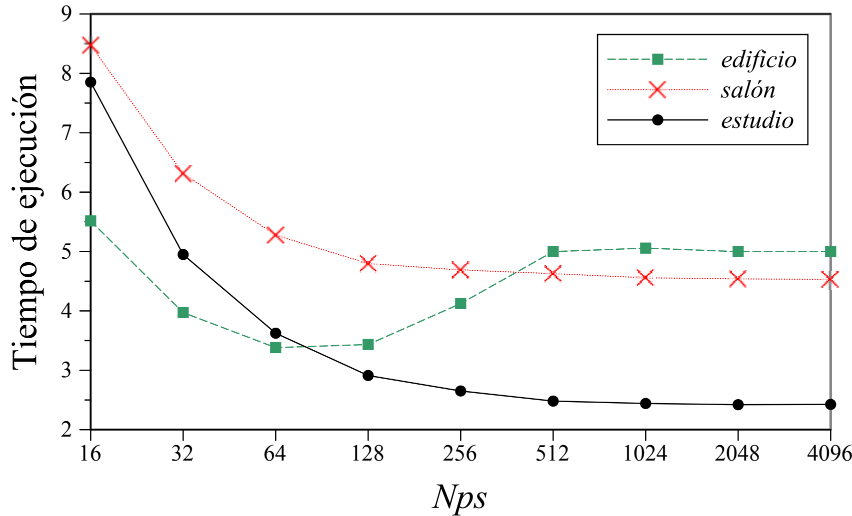
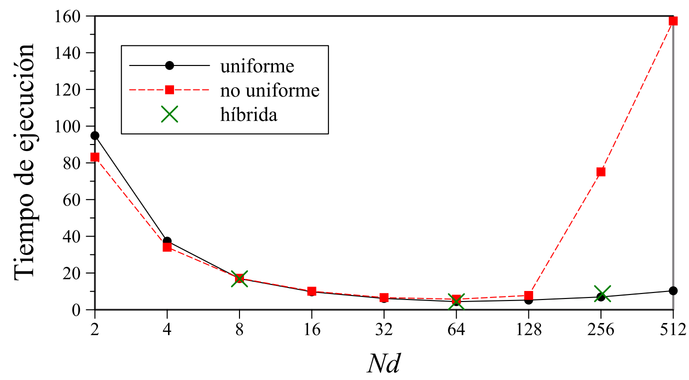


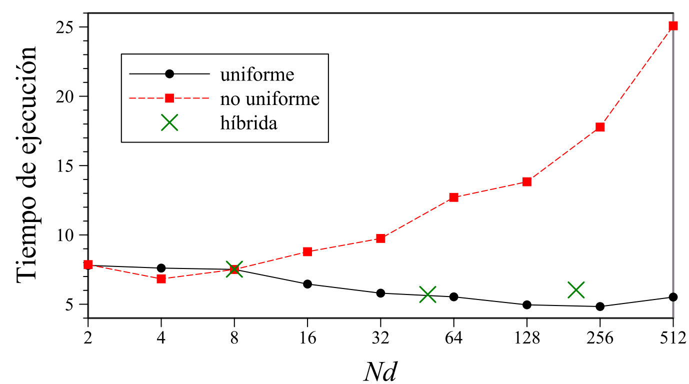
Figura 5.14: Análisis de Nps para $Nd = 64$.

También se ha estudiado el rendimiento según el número de subescenas Nd y los métodos de división. La Figura 5.15 muestra los tiempos de ejecución medidos en segundos para las tres estrategias de división comentadas en la Subsección 5.4.1, aplicadas a las tres escenas bajo análisis y para diferentes valores de Nd. En el caso particular de la división híbrida, como cada subescena cuando se divide lo hace en 8 subescenas, se han ejecutado las pruebas con los valores máximos de Nd de 8, 64 y 512, marcando en las gráficas los valores de Nd finales que se han obtenido. Como se puede observar, la estrategia de división permite la reducción del tiempo de ejecución para valores pequeños de Nd. Esta reducción es debida, principalmente, al menor número de accesos a memoria global y a la utilización eficiente de los bloques de tareas. Esta optimización es posible debido a la explotación de la localidad de los datos asociada a la estrategia de división de la escena empleada.

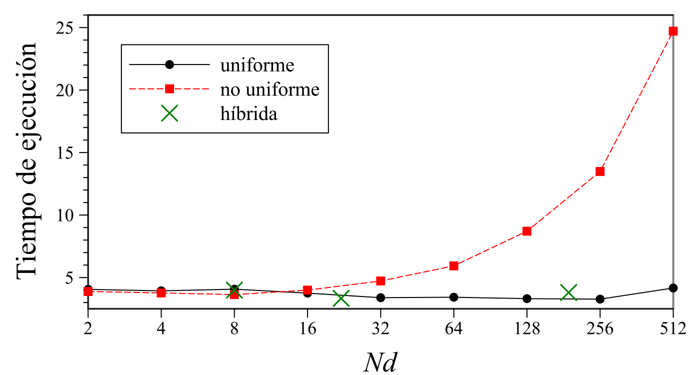
Sin embargo, para valores mayores de Nd el comportamiento es diferente. Para la partición no uniforme el tiempo de ejecución aumenta para $Nd > 4$ con la escena *salón*, siendo para $Nd > 64$ con la escena *edificio* y para $Nd > 8$ con la escena *estudio*. En cuanto a la partición híbrida el tiempo de ejecución se incrementa ligeramente cuando pasamos de $Nd_{\text{máx}} = 64$ a $Nd_{\text{máx}} = 256$. Por último, para la partición uniforme también se aprecia un pequeño incremento con $Nd = 512$ respecto de $Nd = 256$ para las tres escenas. Es necesario indicar que el método de partición uniforme es la mejor solución, en contraste con lo podríamos esperar teniendo en cuenta el desbalanceo de la carga en las escenas irregulares. Este comportamiento está relacionado



(a)

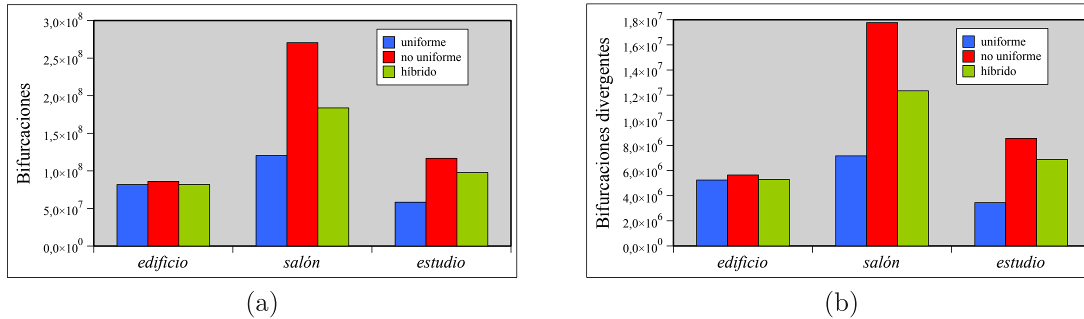


(b)



(c)

Figura 5.15: Tiempos de ejecución para nuestra implementación en GPU con las tres técnicas de división de la escena y diferentes Nd: (a) *edificio*, (b) *salón* y (c) *estudio*.



(a) (b)

Figura 5.16: (a) Número de bifurcaciones; (b) número de bifurcaciones divergentes.

con las diferentes estructuras adyacentes entre las subescenas generadas por los diferentes métodos de partición. Como se comentó en la Subsección 5.4.3, cuando un rayo se envía de una subescena a otra adyacente, la identificación de la subescena destino es más sencilla cuanto más sencilla sea la estructura de subescenas adyacentes. Mientras que para la división uniforme cada frontera de una subescena está modelada por un único elemento virtual compartido por las dos subescenas adyacentes, para los otros dos métodos de partición la estructura de vecindad es más compleja y el número de elementos virtuales es variable. De esta forma, para una implementación GPU, los rayos procesados en el mismo bloque de tareas pueden ir a diferentes subescenas vecinas. Esto implica que el número de comprobaciones realizadas para determinar la subescena destino es diferente para cada rayo y, consecuentemente, puede aparecer un alto grado de divergencia. Debido a la diferente complejidad de las estructuras de vecindad generadas por los distintos métodos de partición, la divergencia es mayor para la partición no uniforme que para la partición híbrida, y es mínima para la partición uniforme.

La Figura 5.16a muestra el número de bifurcaciones del algoritmo para cada tipo de partición con $Nd = 256$. En general, el número de bifurcaciones y el número de bifurcaciones divergentes (ver la Figura 5.16b) es mayor para la partición no uniforme que para las otras. La divergencia del *warp* puede conducir a una gran pérdida de eficiencia en el paralelismo porque, como ya se ha comentado, las tareas de un *warp* se ejecutan concurrentemente si corresponden a la misma instrucción, mientras que las que se desvían lo hacen secuencialmente. Esto explica el comportamiento pobre de la técnica de división no uniforme. En este caso, los buenos resultados que se esperaban como consecuencia de la búsqueda de un mejor balanceo de la carga, no compensan los problemas de divergencia de los *warps* que se generan.

Con el objetivo de estudiar el grado de actividad de los núcleos con esta imple-

Escenas	CPU	GPU uniforme	GPU no uniforme	GPU híbrida
edificio	10,62	3,38 (3,14)	4,75 (2,24)	3,30 (3,22)
salón	12,64	3,81 (3,32)	5,75 (2,20)	4,66 (2,71)
estudio	8,11	2,24 (3,62)	2,54 (3,19)	2,34 (3,46)

Tabla 5.1: Tiempos de ejecución en segundos para las implementaciones en CPU y GPU.

mentación en GPU, se ha calculado la desviación estándar normalizada de tareas útiles por bloque con la siguiente expresión:

$$\sigma_n = \sqrt{\frac{1}{Nb}\left(\sum_{i=1}^{Nb} \frac{Ntb - Ntb^i}{Ntb - 1}\right)^2} \tag{5.5}$$

siendo Ntb^i el número de tareas útiles en el bloque i. Por ejemplo, para la división uniforme y para Nd óptimos los valores de σ_n obtenidos fueron 0,017, 0,011 y 0,019 para las escenas *edificio*, *salón* y *estudio*, respectivamente. Se puede ver que la pequeña desviación confirma el alto grado de utilización de los núcleos.

Adicionalmente, se ha analizado en detalle la eficiencia de esta implementación en GPU respecto a la implementación convencional en la CPU. La Tabla 5.1 muestra los tiempos de ejecución más bajos obtenidos para las tres escenas que se han utilizado en esta comparación. Los datos de cada columna corresponden de izquierda a derecha a una implementación en CPU y a la implementación en GPU con las tres técnicas de partición de la escena. Los valores de *speedup* están indicados entre paréntesis. La implementación en CPU utilizada como referencia es una versión secuencial optimizada del algoritmo de Monte Carlo para radiosidad con la técnica de aceleración *kd–tree*. En concreto se ha empleado el acelerador *kd–tree* de Pharr e Humphreys [99]. El número de iteraciones para la implementación CPU y para todas las implementaciones GPU fue de nueve para la escena *edificio* y de trece para la escena *estudio*. La escena *salón* requirió diez iteraciones en la implementación CPU, pero solo nueve para las implementaciones GPU.

En las implementaciones GPU de las últimas tres columnas para cada caso se han elegido los mejores valores de Nd y k (ver la Ecuación 4.11). La Tabla 5.2 muestra los valores de Nd utilizados y, para las técnicas de partición no uniforme e híbrida, los valores de k están indicados entre paréntesis.

Escenas	uniforme	no uniforme	híbrida
edificio	64	64 (1,1)	64 (1,3)
salón	256	4 (0,0)	50 (0,5)
estudio	256	8 (1,2)	22 (1,1)

Tabla 5.2: Parámetros de configuración: Nd y k.

Se puede observar que con nuestra propuesta se obtienen mejores resultados que con la implementación en CPU, siendo superiores los obtenidos para las implementaciones con las particiones uniforme e híbrida que los de la implementación con la partición no uniforme. Como ejemplo, los mejores resultados de speedup se obtuvieron con la escena estudio, siendo 3,62 para la partición uniforme y 3,46 para la propuesta híbrida. En el caso más desfavorable, que ha sido para la escena salón con la división no uniforme, se ha logrado un speedup de 2,20 sobre la versión optimizada de CPU. De estos resultados se puede deducir que, aunque se ha logrado disminuir el tiempo de ejecución respecto de la versión optimizada de CPU y teniendo en cuenta la irregularidad del algoritmo, aún queda trabajo por hacer. Una de las posibles mejoras sería intentar reducir las divergencias para buscar aprovechar mejor el reparto de la carga computacional, sobre todo en el empleo de la división no uniforme, que ha sido el método que peores resultados ha dado, a diferencia de lo que se había obtenido en la implementación distribuida de MCR.

Conclusiones y principales aportaciones

Durante los últimos años la demanda en la producción de imágenes sintéticas cada vez más realistas ha crecido en multitud de campos, tales como la visualización científica, la industria del cine, la publicidad, los videojuegos, las aplicaciones médicas o el diseño asistido por ordenador. Satisfacer esta demanda ha sido posible con la aplicación de modelos de iluminación global basados en la radiosidad ya que simulan de forma fiel el comportamiento físico real de la luz. Concretamente, en la actualidad ningún otro modelo de iluminación global puede obtener, con un coste razonable, la simulación efectiva de la componente difusa del reflejo de la luz en su interacción con una superficie.

La solución del sistema de ecuaciones del método de radiosidad normalmente se obtiene mediante métodos iterativos. Dentro de estos, básicamente hay dos orientaciones principales: los algoritmos iterativos determinísticos y los métodos estocásticos. Entre los primeros se ha seleccionado el método de radiosidad progresiva que consigue al final de cada iteración una aproximación de la radiosidad de toda la escena, sin la necesidad de recorrer todos los elementos. Dentro de los métodos probabilísticos se ha escogido la solución estocástica de Jacobi, que destaca por su simplicidad computacional y por su eficiencia igual o superior a otros métodos estocásticos.

Las principales aportaciones de esta memoria son:

- Se ha implementado el algoritmo de radiosidad progresiva en un sistema paralelo de memoria distribuida utilizando el paradigma de paso de mensajes. Este método está basado en cuatro estrategias: la partición convexa de la escena, que evita la replicación de la escena en la memorias locales y permite la optimización de la localidad de los datos; la utilización de comunicaciones

no bloqueantes para solapar cálculos y minimizar la sobrecarga de las comunicaciones; el empleo de una modificación de la técnica de las máscaras de visibilidad para realizar el cálculo de la visibilidad entre elementos de subescenas distintas; un método para determinar la finalización de cada iteración que evita la sincronización de los procesadores. En cuanto a la división de la escena se han desarrollado dos técnicas: partición uniforme y no uniforme.

- Se ha analizado un método de partición geométrica uniforme que genera subescenas convexas de igual forma y tamaño. Este método produce una división de la escena extremadamente rápida. Además esta partición minimiza el número de subescenas vecinas que puede tener una subescena, simplificando los cálculos relacionados con las máscaras de visibilidad y reduciendo el número de comunicaciones necesarias. La implementación se ha probado con varias escenas en dos *clústeres* de procesadores con diferentes características, tanto en términos de procesadores como en cuanto a su red de interconexión. Por otro lado, también se ha examinado la aplicación del método de partición uniforme al algoritmo secuencial. Se ha verificado que la aplicación de la estrategia de partición mejora el rendimiento del algoritmo secuencial mediante el incremento de la localidad de los datos. En esta evaluación se han obteniendo importantes reducciones en el tiempo de ejecución, alcanzando un *speedup* de 4,30 respecto del algoritmo original. Además se ha comprobado que no hay pérdida en la calidad del resultado final.
- Se ha propuesto un algoritmo de división no uniforme para resolver el problema del balanceo de la carga que presenta la división uniforme. Este algoritmo está basado en la minimización de una función de distribución que describe el desbalanceo de la carga en términos de un factor. Este factor es una propiedad de cada elemento de la escena que se emplea para caracterizar las particiones resultantes. Para la utilización de la técnica de las máscaras de visibilidad se ha propuesto una jerarquía de procesadores, basada en la utilización de procesadores virtuales que permite la reconstrucción de las máscaras de visibilidad. Con este método y usando como factor de balanceo el área, sobre *SCI Clúster* se han logrado mejorar los valores de *speedup* del método de división uniforme, llegando a alcanzar un *speedup* de 19,49 para 8 procesadores para la división no uniforme frente a 16,58 de la división uniforme.

- Se ha desarrollado una implementación paralela en un sistema de memoria dis-

tribuida para el algoritmo de Monte Carlo de radiosidad. Nuestra propuesta resuelve los retos asociados con la implementación distribuida del algoritmo. Además, como los datos son comunicados eficientemente entre los procesadores, no es necesaria ninguna aproximación y se obtienen imágenes de alta calidad. Nuestra propuesta está basado en tres técnicas: la partición convexa de la escena para permitir procesar escenas complejas y simplificar el proceso de disparo de rayos; la utilización del empaquetamiento de los rayos que se han de enviar a otros procesadores para minimizar el número de comunicaciones; y una metodología para determinar el fin de una iteración en un sistema distribuido sin usar sincronizaciones o barreras que degraden el rendimiento.

- Se ha modificado la función de distribución propuesta para el algoritmo de radiosidad progresiva para la división no uniforme, utilizando como factor guía de la división de la escena un valor que incluye no solo el área sino también la potencia radiante. Esta elección es debida a la propia naturaleza del método de Monte Carlo para radiosidad, en el cual se realiza el disparo de rayos individuales en el lugar de elementos. El número de rayos depende de la potencia sin disparar, por lo que se obtiene un mejor balanceo de la carga computacional si se tiene en cuenta la potencia a la hora de dividir la escena. Con esta función de distribución se ha logrado mejorar los resultados para escenas con una distribución irregular de objetos y luces, alcanzando en el supercomputador *Finis Terrae* un valor de *speedup* de 17,37 para 32 procesadores y obteniendo superlinealidad en algunas configuraciones.

- Se ha desarrollado una estrategia de empaquetamiento de los rayos que han de ser comunicados entre los procesadores. Se ha analizado la influencia del tamaño del paquete de rayos en en rendimiento del algoritmo distribuido. Nuestra propuesta utiliza comunicaciones no bloqueantes para permitir el solapamiento con los cálculos. Esto hace necesario introducir una serie de puntos de muestreo a lo largo del código para comprobar la recepción de nuevos mensajes. También se ha estudiado la influencia de la frecuencia del chequeo de paquetes recibidos. Este esquema ha permitido la minimización y optimización de las comunicaciones entre los procesadores, cuando se utilizan tamaños de paquetes en un rango específico y para determinados intervalos de frecuencias de chequeo.

- Teniendo en cuenta el empleo de comunicaciones no bloqueantes, la complejidad de la comprobación del fin de iteración del algoritmo MCR nos

ha llevado a desarrollar una solución basada en tres fases: una primera de un chequeo local en cada procesador donde cada procesador evalúa si tiene rayos recibidos sin disparar y si tiene paquetes de rayos incompletos sin enviar; una segunda de comunicaciones entre procesadores donde se envía y recibe información relativa al número de paquetes de rayos enviados y recibidos; y una tercera de detección final de la iteración, en la que se procesa la información recibida sobre el número de paquetes comunicados para chequear si ha finalizado la iteración, o si hay que seguir esperando la recepción y envío de nuevos paquetes de rayos.

- Se ha realizado y analizado una implementación de grano fino del método de Monte Carlo para radiosidad en una GPU NVIDIA utilizando CUDA. Como resultado se ha logrado alcanzar un valor de *speedup* de 3,62 respecto de una implementación secuencial optimizada en CPU. Nuestra propuesta está basada en tres técnicas: la división de la escena en subescenas, la utilización de una versión simplificada de la malla de elementos basada en un criterio de planitud y la organización eficiente de las tareas.

  - Se ha utilizado la división de la escena para optimizar la localidad de los datos, analizando las técnicas de división uniforme y no uniforme. Además se ha desarrollado y estudiado un tercer método de división, llamado división híbrida, que aproxima el balanceo de la carga de la partición no uniforme, manteniendo cierta simplicidad de la división uniforme. Para ello se utiliza una división adaptativa de la escena usando como base el método de división uniforme. Como resultado de este análisis, podemos concluir que los potenciales beneficios de las técnicas no uniforme e híbrida en términos del balanceo de la carga computacional no compensan el incremento del grado de divergencia en los *warps*. Este incremento es debido a la mayor complejidad de las estructuras adyacentes de las subescenas en estos dos métodos de división.
  - Se ha propuesto la utilización de una versión simplificada de la malla de elementos de la escena, basada en un criterio de planitud. Esta malla simplificada almacena la información de una representación menos detallada de la escena. Esto reduce los requerimientos de almacenamiento posibilitando la explotación de la memoria de texturas de la GPU. La idea de nuestra propuesta es emplear la versión simplificada de las áreas planas para comprobar las intersecciones de los rayos con las superficies, mientras que se mantiene el detalle para el cálculo de la radiosidad. Esto

permite acelerar la búsqueda del elemento destino de un rayo sin perder calidad en el resultado final.

- Se ha utilizado un esquema de planificación de las tareas que mantiene tareas activas en todos los *warps*, proporcionando un alto grado de utilización de los núcleos de la GPU. La gestión de tareas utilizada logra, en general, que las tareas del mismo bloque procesen rayos de la misma subescena, consiguiendo accesos a la caché de la memoria de texturas con la consiguiente aceleración de la aplicación.

De los análisis realizados se deduce que aún queda trabajo por realizar para optimizar las ejecuciones, tanto en los sistemas distribuidos como en las GPUs. Por ejemplo, en el caso de la GPU se ha comprobado que existen un gran número de bifurcaciones divergentes que degradan el rendimiento del algoritmo, con lo cual sería importante lograr su disminución. Como trabajo futuro, una primera tarea que nos planteamos es realizar una implementación eficiente de una estructura de aceleración *kd–tree* en la GPU. También nos planteamos adaptar las técnicas presentadas a un entorno híbrido, es decir, a sistemas multiprocesadores con nodos multinúcleo y coprocesadores GPUs. En concreto, pretendemos diseñar un método eficiente en un clúster de GPUs. En este tipo de sistemas es fundamental explotar la localidad de los datos en cada GPU, ya que las transferencias de datos entre CPU y GPU y entre los diferentes nodos de GPUs son un cuello de botella importante para obtener buenos rendimientos.

Bibliografía

[1] Advanced Micro Devices, Inc. *ATI Stream Computing rev. 1.4.0a*, 2009. Citado en p. 29, 123

[2] T. Akenine-Möller, E. Haines, y N. Hoffman. *Real-Time Rendering*. AK Peters, 3ª edición, 2008. Citado en p. 2, 5, 6, 126

[3] L. Alonso, F. Cuny, S. Petit Jean, J.-C. Paul, S. Lazard, y E. Wies. The Virtual Mesh: A Geometric Abstraction for Efficiently Computing Radiosity. *ACM Transactions on Graphics (TOG)*, 20(3):169–201, 2001. Citado en p. 10

[4] G. Ammons, T. Ball, y J. R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. En *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI'97)*, páginas 85–96, New York, USA, 1997. ACM. Citado en p. 70

[5] M. Amor, J. R. Sanjurjo, E. J. Padrón, y R. Doallo. Progressive Radiosity Method on Clusters Using a New Clipping Algorithm. *International Journal of High Performance Computing and Networking (IJHPCN)*, 1(1/2/3):55–63, 2004. Citado en p. 31, 34, 45, 66

[6] M. Amor, J. R. Sanjurjo, E. J. Padrón, A. P. Guerra, y R. Doallo. A Scene Partitioning Method for Global Illumination Simulations on Clusters. En *Proceedings of 2nd Workshop on Hardware/Software Support for High Performance Scientific and Engineering Computing (SHPSEC 2003)*, páginas S3/1–S3/17, New Orleans, USA, septiembre 2003. Citado en p. 31, 34, 45, 66

[7] B. Arnaldi, T. Priol, L. Renambot, y X. Pueyo. Visibility Masks for Solving Complex Radiosity Computations on Multiprocessors. En *Proceedings of the First Eurographics Workshop on Parallel Graphics and Visualization*, páginas 219–232, 1996. Citado en p. 28, 34, 41, 51, 53, 75

[8] J. Arvo y D. Kirk. Fast Ray Tracing by Ray Classification. *ACM SIGGRAPH Computer Graphics (Proc. of SIGGRAPH '87)*, 21(4):55–64, 1987. Citado en p. 37

[9] P. Bekaert. *Hierarchical and Stochastic Algorithms for Radiosity.* Tesis doctoral, Katholieke Universiteit Leuven, Lovaina, Bélgica, 1999. Citado en p. 19

[10] P. Bekaert, L. Neumann, A. Neumann, M. Sbert, y Y. D. Willems. Hierarchical Monte Carlo Radiosity. En *Rendering Techniques'98, Proceedings of the Eurographics Workshop*, páginas 259–268, Viena, Austria, junio 1998. Springer. Citado en p. 18

[11] M. J. Berger y S. H. Bokhari. A Partitioning Strategy for Nonuniform Problems on Multiprocessors. *IEEE Transactions on Computers*, 36(5):570–580, 1987. Citado en p. 40

[12] D. Blythe. The Direct3D 10 System. *ACM Transactions on Graphics (TOG)*, 25:724–734, julio 2006. Citado en p. 126

[13] T. Boubekeur y C. Schlick. Generic Mesh Refinement on GPU. En *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS'05)*, páginas 99–104. Eurographics Association, julio 2005. Citado en p. 126

[14] C. Boyd y M. Schmit. *DirectX 11 Compute Shader*, 2008. Citado en p. 29

[15] N. A. Carr, J. D. Hall, y J. C. Hart. GPU Algorithms for Radiosity and Subsurface Scattering. En *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS'03)*, páginas 51–59, Aire-la-Ville, Suiza, 2003. Eurographics Association. Citado en p. 29

[16] F. Castro, M. Sbert, y J. H. Halton. Technical Section: Efficient Reuse of Paths for Random Walk Radiosity. *Computers and Graphics*, 32(1):65–81, 2008. Citado en p. 18

[17] G. Cerruela. *Radiosidad en Multiprocesadores.* Tesis doctoral, Universidad de Málaga, Málaga, España, 1998. Citado en p. 28

[18] M.-H. Choi, W.-C. Park, F. Neelamkavil, T.-D. Han, y S.-D. Kim. An Effective Visibility Culling Method Based on Cache Block. *IEEE Transactions on Computer*, 55(8):1024–1031, 2006. Citado en p. 60

[19] M. Cohen, S. E. Chen, J. R. Wallace, y D. P. Greenberg. A Progressive Refinement Approach to Fast Radiosity Image Generation. *ACM SIGGRAPH Computer Graphics (Proc. of SIGGRAPH '88)*, 22(4):31–40, 1988. Citado en p. 12, 15

[20] M. Cohen y J. R. Wallace. *Radiosity and Realistic Image Synthesis*. Academic Press Professional, Boston, USA, 1993. Citado en p. 6, 9, 11, 19, 37

[21] M. F. Cohen y D. P. Greenberg. The Hemi-Cube: a Radiosity Solution for Complex Environments. *ACM SIGGRAPH Computer Graphics (Proc. of SIGGRAPH '85)*, 19(3):31–40, agosto 1985. Citado en p. 6

[22] R. Concheiro, M. Amor, y M. Bóo. Synthesis of Bézier Surfaces on the GPU. En *Proceedings of the International Conference on Computer Graphics Theory and Applications (GRAPP 2010)*, páginas 110–115. INSTICC Press, mayo 2010. Citado en p. 126

[23] R. Concheiro, M. Amor, y M. Bóo. Synthesis of Bézier Surfaces on the GPU. En *Proceedings of the International Conference on Computer Graphics Theory and Applications (GRAPP 2010)*, páginas 110–115, Angers, France, mayo 2010. INSTICC Press. Citado en p. 145

[24] G. Coombe, M. J. Harris, y A. Lastra. Radiosity on Graphics Hardware. En *Proceedings of Graphics Interface 2004 (GI'04)*, páginas 161–168, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canadá, 2004. Canadian Human-Computer Communications Society. Citado en p. 30

[25] N. D. Cuong. An Exploration of Coherence-Based Acceleration Methods Using the Ray Tracing Kernel G/GX. Informe técnico, TU-Dresden, Alemania, 1997. Citado en p. 37

[26] C. Dachsbacher, M. Stamminger, G. Drettakis, y F. Durand. Implicit Visibility and Antiradiance for Interactive Global Illumination. *ACM Transactions on Graphics (TOG)*, 26(3):artículo 61, 10 páginas, 2007. Citado en p. 30, 123

[27] C. DeCoro y N. Tatarchuk. Real-time Mesh Simplification using the GPU. En *Proceedings of the 2007 Symposium on Interactive 3D Graphics (I3D'07)*, páginas 161–166, New York, USA, 2007. ACM. Citado en p. 126

[28] Z. Dong, J. Kautz, C. Theobalt, y H.-P. Seidel. Interactive Global Illumination Using Implicit Visibility. En *Proceedings of 15th Pacific Conference*

on Computer Graphics and Applications (PG'07), páginas 77–86, Washington D.C., USA, 2007. IEEE Computer Society. Citado en p. 30, 123

[29] R. Dumont. *Optimizing Global Illumination Computation for Indoor and Outdoor Scenes: LODs, Partitioning and Arbitrary Reflectances*. Tesis doctoral, Institut National de Recherche en Informatique (IRISIA), Rennes, Francia, 1999. Citado en p. 12

[30] P. Dutré, P. Bekaert, y K. Bala. *Advanced Global Illumination*. A K Peters, Ltd., 2006. Citado en p. 2, 3, 6, 11, 18, 21, 24, 92, 94

[31] C.-C. Feng y S.-N. Yang. A Parallel Hierarchical Radiosity Algorithm for Complex Scenes. En *Proceedings of the IEEE Symposium on Parallel Rendering (PRS'97)*, páginas 71–78, New York, USA, 1997. ACM. Citado en p. 28

[32] R. Fernando y M. J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, USA, 2003. Citado en p. 29, 123

[33] Web *Top500 Supercomputer Sites*: página del supercomputador *Finis Terrae*. http://www. top500.org/system/9500. Citado en p. 109

[34] J. D. Foley, A. van Dam, S. K. Feiner, y J. F. Hughes. *Computer Graphics: Principles and Practice in C*. Addison-Wesley Professional, segunda edición, agosto 1995. Citado en p. 45

[35] T. A. Funkhouser. Coarse-Grained Parallelism for Hierarchical Radiosity using Group Iterative Methods. En *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'96)*, páginas 343–352, New York, USA, 1996. ACM. Citado en p. 12

[36] M. Garland y E. Shaffer. A Multiphase Approach to Efficient Surface Simplification. En *Proceedings of the Conference on Visualization '02*, VIS '02, páginas 117–124, Washington, USA, 2002. IEEE Computer Society. Citado en p. 142

[37] R. Garmann. Spatial Partitioning for Parallel Hierarchical Radiosity on Distributed Memory Architectures. En *Proceedings of the Third Eurographics Workshop on Parallel Graphics and Visualization (EGPGV'00)*, páginas 13–23, Girona, España, septiembre 2000. Springer-Verlag. Citado en p. 27

[38] S. Ghosh, M. Martonosi, y S. Malik. Cache Miss Equation: A Compiler Framework for Analyzing and Tuning Memory Behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999. Citado en p. 62

[39] S. Gibson y R. J. Hubbold. A Perceptually-Driven Parallel Algorithm for Efficient Radiosity Simulation. *IEEE Transactions on Visualization and Computer Graphics*, 6(3):220–235, 2000. Citado en p. 29

[40] J. R. Gilbert, G. L. Miller, y S.-H. Teng. Geometric Mesh Partitioning: Implementation and Experiments. En *Proceedings of International Parallel Processing Symposium*, páginas 418–427, 1995. Citado en p. 40

[41] A. S. Glassner, editor. *An Introduction to Ray Tracing*. Academic Press Ltd., London, UK, 1989. Citado en p. 6

[42] O. Good y Z. Taylor. Optimized Photon Tracing Using Spherical Harmonic Light Maps. En *SIGGRAPH '05: ACM SIGGRAPH 2005 Sketches*, página 53, New York, USA, 2005. ACM. Citado en p. 26

[43] C. M. Goral, K. E. Torrance, D. P. Greenberg, y B. Battaile. Modeling the Interaction of Light between Diffuse Surfaces. *ACM SIGGRAPH Computer Graphics (Proc. of SIGGRAPH '84)*, 18(3):213–222, julio 1984. Citado en p. 6, 8

[44] S. Gortler, M. F. Cohen, y P. Slusallek. Radiosity and Relaxation Methods. *IEEE Computer Graphics and Applications*, 14(6):48–58, 1994. Citado en p. 12, 16, 34

[45] K. Gray. *Microsoft DirectX 9 Programmable Graphics Pipeline*. Microsoft Press, Redmond, USA, 2003. Citado en p. 29, 123

[46] G. Greiner y K. Hormann. Efficient Clipping of Arbitrary Polygons. *ACM Transactions on Graphics (TOG)*, 17(2):71–83, abril 1998. Citado en p. 45

[47] A. P. Guerra, M. Amor, J. Eiroa, E. J. Padrón, J. R. Sanjurjo, y R. Doallo. Método de Radiosidad Progresiva de Alto Rendimiento Basado en el Particionamiento de la Escena. En *Actas de las XIII Jornadas de Paralelismo*, páginas 251–256, Lleida, España, septiembre 2002. Citado en p. 31, 34, 45, 66

[48] A. P. Guerra, M. Amor, E. J. Padrón, y R. Doallo. A High–Performance Progressive Radiosity Method Based on Scene Partitioning. En *High Performance*

Computing for Computational Science - VECPAR 2002, volumen 2565 de *Lecture Notes in Computer Science*, páginas 537–548. Springer, 2003. Citado en p. 28

[49] E. A. Haines y J. R. Wallace. Shaft Culling for Efficient Ray-Traced Radiosity. En *Photorealistic Rendering in Computer Graphics (Proceedings of the Second Eurographics Workshop on Rendering)*, páginas 122–138, 1994. Citado en p. 37, 38

[50] J. Halton. A Retrospective and Prospective Survey of the Monte Carlo Method. *SIAM Review*, 12:1–63, 1970. Citado en p. 22

[51] P. Hanrahan, D. Salzman, y L. Aupperle. A Rapid Hierarchical Radiosity Algorithm. *ACM SIGGRAPH Computer Graphics (Proc. of SIGGRAPH'91)*, 25(4):197–206, 1991. Citado en p. 7, 12, 13

[52] J.-M. Hasenfratz, C. Damez, F. Sillion, y G. Drettakis. A Practical Analysis of Clustering Strategies for Hierarchical Radiosity. *Computer Graphics Forum (Proc. of Eurographics '99)*, 18(3):221–232, septiembre 1999. Citado en p. 12

[53] V. Havran. *Heuristic Ray Shooting Algorithms*. Tesis doctoral, Departamento de Computer Science and Engineering, Facultad de Electrical Engineering, Czech Technical University, Praga, República Checa, noviembre 2000. Citado en p. 139

[54] B. Hendrickson y R. Leland. The CHACO User's Guide, versión 2.0. Informe técnico SAND95-2344, Sandia National Laboratories, Alburquerque, USA, julio 1995. Citado en p. 40

[55] D. R. Horn, J. Sugerman, M. Houston, y P. Hanrahan. Interactive k-D tree GPU Raytracing. En *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games (I3D'07)*, I3D '07, páginas 167–174, New York, USA, 2007. ACM. Citado en p. 140

[56] W. Hunt, W. R. Mark, y G. Stoll. Fast KD-tree Construction with an Adaptive Error-Bounded Heuristic. En *IEEE Symposium on Interactive Ray Tracing*, páginas 81–88. IEEE, septiembre 2006. Citado en p. 139

[57] W. Hwu, K. Keutzer, y T. G. Mattson. The Concurrency Challenge. *IEEE Design & Test of Computers*, páginas 312–320, 2008. Citado en p. 124

[58] Intel®. Proyecto de investigación *Single-Chip Cloud Computer*. http://techresearch.intel.com/ProjectDetails.aspx?Id=1. Citado en p. 125

[59] H. W. Jensen. *Realistic Image Synthesis Using Photon Mapping*. A K Peters, Ltd., Natick, USA, 2001. Citado en p. 6, 18, 24, 26, 89

[60] J. T. Kajiya. The Rendering Equation. *ACM SIGGRAPH Computer Graphics (Proc. of SIGGRAPH '86)*, 20(4):143–150, 1986. Citado en p. 5

[61] M. H. Kalos y P. A. Whitlock. *Monte Carlo methods. Vol. 1: basics*. Wiley-Interscience, New York, USA, 1986. Citado en p. 17

[62] A. Kaplanyan y C. Dachsbacher. Cascaded Light Propagation Volumes for Real-Time Indirect Illumination. En *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games (I3D'10)*, páginas 99–107, New York, USA, 2010. ACM. Citado en p. 30, 124

[63] G. Karypis y V. Kumar. METIS: a Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill–Reducing Orderings of Sparse Matrices, versión 4.0. Informe técnico, University of Minnesota, Minneapolis, USA, septiembre 1998. Citado en p. 40

[64] G. Karypis y V. Kumar. Multilevel k-way Partitioning Scheme for Irregular Graphs. *Journal of Parallel and Distributed Computing*, 48:96–129, 1998. Citado en p. 40

[65] A. Keller. Instant Radiosity. En *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '97)*, páginas 49–56, New York, USA, 1997. ACM Press/Addison-Wesley Publishing Co. Citado en p. 29

[66] Khronos OpenCL Working Group. *The OpenCL Specification*, 2008. Citado en p. 29, 123

[67] D. B. Kirk y W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, febrero 2010. Citado en p. 125, 131

[68] M. Kurt y D. Edwards. A Survey of BRDF Models for Computer Graphics. *ACM SIGGRAPH Computer Graphics Quarterly*, 43(2), 2009. Citado en p. 5

[69] S. Laine, H. Saransaari, J. Kontkanen, J. Lehtinen, y T. Aila. Incremental Instant Radiosity for Real-Time Indirect Illumination. En *Proceedings of Eurographics Symposium on Rendering 2007*, páginas 277–286. Eurographics Association, 2007. Citado en p. 30

[70] B. D. Larsen y N. J. Christensen. Simulating Photon Mapping for Real-time Applications. En *Rendering Techniques '04*, páginas 123–132, 2004. Citado en p. 18, 89

[71] J. Lehtinen, M. Zwicker, E. Turquin, J. Kontkanen, F. Durand, F. X. Sillion, y T. Aila. A Meshless Hierarchical Representation for Light Transport. *ACM Transactions on Graphics (TOG)*, 27:artículo 37, 9 páginas, agosto 2008. Citado en p. 30

[72] E. Lindholm, J. Nickolls, S. Oberman, y J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28:39–55, marzo 2008. Citado en p. 126

[73] V. C. H. Ma y M. D. McCool. Low Latency Photon Mapping Using Block Hashing. En *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS'02)*, páginas 89–99, Aire-la-Ville, Suiza, 2002. Eurographics Association. Citado en p. 26

[74] I. Martin, X. Pueyo, y D. Tost. A Two-Pass Hardware-Based Method Hierarchical Radiosity. *Computer Graphics Forum*, 17(3):159–164, 1998. Citado en p. 29

[75] R. E. Martínez Ramírez. *Adaptive and Depth Buffer Solutions with Bundles of Parallel Rays for Global Line Monte Carlo Radiosity.* Tesis doctoral, Universitat Politècnica de Catalunya, Barcelona, España, 2004. Citado en p. 27

[76] D. Meneveaux y K. Bouatouch. Synchronization and Load Balancing for Parallel Hierarchical Radiosity of Complex Scenes on a Heterogeneous Computer Network. *Computer Graphics Forum*, 18(4):201–212, 1999. Citado en p. 12, 28

[77] Q. Meyer, C. Eisenacher, M. Stamminger, y C. Dachsbacher. Data-Parallel Hierarchical Link Creation for Radiosity. En *Proceedings of Nineth Eurographics Workshop on Parallel Graphics and Visualization*, páginas 65–70, 2009. Citado en p. 30, 124

[78] S. Moore y N. Smeds. Performance Tuning Using Hardware Counter Data. En *SuperComputing 2001*, 2001. Citado en p. 70

[79] L. Neumann. Monte Carlo Radiosity. *Computing*, 55(1):23–42, 1995. Citado en p. 18

[80] L. Neumann, M. Feda, M. Kopp, y W. Purgathofer. A New Stochastic Radiosity Method for Highly Complex Scenes. En *Fifth Eurographics Workshop on Rendering*, páginas 195–206, junio 1994. Citado en p. 18

[81] L. Neumann y A. Neumann. Radiosity and Hybrid Methods. *ACM Transactions on Graphics (TOG)*, 14(3):233–265, 1995. Citado en p. 12, 16, 34

[82] L. Neumann, W. Purgathofer, R. F. Tobler, A. Neumann, P. Elias, M. Feda, y X. Pueyo. The Stochastic Ray Method for Radiosity. En *Rendering Techniques '95*, páginas 206–218, Wien, Austria, 1995. Citado en p. 18, 21

[83] G. Nichols, J. Shopf, y C. Wyman. Hierarchical Image-Space Radiosity for Interactive Global Illumination. *Computer Graphics Forum*, 28(4):1141–1149, 2009. Citado en p. 30

[84] G. Nichols y C. Wyman. Multiresolution Splatting for Indirect Illumination. En *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games (I3D'09)*, páginas 83–90, New York, USA, 2009. ACM. Citado en p. 30

[85] F. E. Nicodemus, J. C. Richmond, J. J. Hsia, I. W. Ginsberg, y T. Limperis. Geometrical Considerations and Nomenclature for Reflectance. NBS Monograph 160, National Bureau of Standards, Washington D.C., USA, 1977. Citado en p. 4

[86] K. H. Nielsen y N. J. Christensen. Fast Texture-Based Form Factor Calculations for Radiosity Using Graphics Hardware. *Journal of Graphics Tools*, 6(4):1–12, 2002. Citado en p. 30

[87] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, 2010. Citado en p. 126, 132

[88] NVIDIA. *NVIDIA CUDA. Programming Guide 3.0*, 2010. Citado en p. 29, 123, 126, 135, 149

[89] NVIDIA®. NVIDIA GeForce GTX 580 GPU Datasheet. http://www.nvidia.com/docs/IO/100940/GeForce_GTX_580_Datasheet.pdf. Citado en p. 125

[90] OpenGL, D. Shreiner, M. Woo, J. Neider, y T. Davis. *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. Addison-Wesley Professional, agosto 2005. Citado en p. 123

[91] P. S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, Inc., 1997. Citado en p. 51, 109

[92] E. Padrón, M. Amor, J. Touriño, y R. Doallo. Hierarchical Radiosity on Multicomputers: a Load–Balanced Approach. En *Proceedings of the Tenth SIAM Conference on Parallel Processing for Scientific Computing*, 2001. Citado en p. 12, 28

[93] E. J. Padrón. *Técnicas de Aceleración para el Método de Radiosidad Jerárquica*. Tesis doctoral, Universidade da Coruña, A Coruña, España, febrero 2006. Citado en p. 37

[94] E. J. Padrón, M. Amor, M. Bóo, y R. Doallo. A Hierarchical Radiosity Method with Scene Distribution. En *Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP'07)*, páginas 134–138, Washington D.C., USA, 2007. IEEE Computer Society. Citado en p. 12, 28

[95] E. J. Padrón, M. Amor, M. Bóo, y R. Doallo. High Performance Global Illumination on Multi-core Architectures. En *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP'09)*, páginas 93–100, Washington D.C., USA, 2009. IEEE Computer Society. Citado en p. 27

[96] E. J. Padrón, M. Amor, M. Bóo, y R. Doallo. Hierarchical Radiosity for Multiresolution Systems Based on Normal Tests. *The Computer Journal*, 53(6), 2010. Citado en p. 12

[97] E. J. Padrón, R. Troncoso, M. Amor, J. R. Sanjurjo, y R. Doallo. Estudio Comparativo de Algoritmos de Visibilidad en los Modelos de Iluminación Global. En *Actas del XIII Congreso Español de Informática Gráfica (CEIG 2003)*, páginas 141–154, A Coruña, España, 2003. Citado en p. 37

[98] D. A. Patterson y J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, segunda edición, 1997. Citado en p. 62

[99] M. Pharr y G. Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, USA, 2004. Citado en p. 153

[100] S. Popov, J. Günther, H.-P. Seidel, y P. Slusallek. Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum*, 26(3):415–424, septiembre 2007. Citado en p. 140

[101] L. Renambot, B. Arnaldi, T. Priol, y X. Pueyo. Towards Efficient Parallel Radiosity for DSM-Based Parallel Computers Using Virtual Interfaces. En *Proceedings of the IEEE Symposium on Parallel Rendering (PRS'97)*, páginas 79–86, New York, USA, 1997. ACM. Citado en p. 27

[102] T. Ritschel, T. Grosch, M. H. Kim, H.-P. Seidel, C. Dachsbacher, y J. Kautz. Imperfect Shadow Maps for Efficient Computation of Indirect Illumination. *ACM Transactions on Graphics (TOG)*, 27(5):artículo 129, 8 páginas, diciembre 2008. Citado en p. 30, 124

[103] R. Y. Rubinstein. *Simulation and the Monte Carlo Method*. John Wiley, New York, USA, 1981. Citado en p. 22

[104] J. R. Sanjurjo, M. Amor, M. Bóo, y R. Doallo. Parallel Global Illumination Method based on a Non-Uniform Partitioning of the Scene. En *Proceedings of the 13th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP'05)*, páginas 251–257, Lugano, Suiza, febrero 2005. EUROMICRO. Citado en p. 32, 76

[105] J. R. Sanjurjo, M. Amor, M. Bóo, y R. Doallo. Improving Locality for Progressive Radiosity Algorithm: A Study based on the Blocking Transformation of the Scene. En *Proceedings of the Fifteenth International Conference in Central Europe on Computer Graphics, Visualization, and Computer Vision (WSCG 2007)*, páginas 113–120, Plzen, República Checa, enero 2007. University of West Bohemia. Citado en p. 31, 34, 61, 97, 112

[106] J. R. Sanjurjo, M. Amor, M. Bóo, y R. Doallo. Parallel Monte Carlo Radiosity using Scene Partitioning. *International Journal of High Performance Computing Applications*, 2011. En prensa. Citado en p. 32, 90

[107] J. R. Sanjurjo, M. Amor, M. Bóo, R. Doallo, y J. Casares. High-Performance Monte Carlo Radiosity on the GPU using CUDA. En *Proceedings of the International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE 2009)*, páginas 965–976, Gijón, España, 2009. Citado en p. 32, 145

[108] J. R. Sanjurjo, M. Amor, M. Bóo, E. J. Padrón, y R. Doallo. Método de Radiosidad de Monte Carlo Basado en la Distribución de la Escena. En *Actas de las XIX Jornadas de Paralelismo*, páginas 133–138, Castellón, España, septiembre 2008. Publicaciones de la Universitat Jaume I. Citado en p. 32

[109] J. R. Sanjurjo, M. Amor, M. Bóo, y R. Doallo. High-Performance Monte Carlo Radiosity on GPU based on Scene Partitioning. *Microprocessors and Microsystems: Embedded Hardware Design (MICPRO)*, 2011. En prensa. Citado en p. 32, 124, 145

[110] J. R. Sanjurjo, M. Amor, M. Bóo, R. Doallo, y J. Casares. Optimizing Monte Carlo Radiosity on Graphics Hardware. *The Journal of Supercomputing*, 58(2):177–185, 2011. Citado en p. 32, 145

[111] J. R. Sanjurjo, M. Amor, E. J. Padrón, R. Doallo, y M. Bóo. Uniform Partitioning of Monte Carlo Radiosity on GPUs. En *Proceedings of the 2010 International Conference on High Performance Computing and Simulation (HPCS 2010)*, páginas 477–483, Caen, Francia, 2010. IEEE. Citado en p. 32, 124, 145

[112] M. Sbert. *The Use of Global Random Directions to Compute Radiosity. Global Monte Carlo Tecniques*. Tesis doctoral, Universitat Politècnica de Catalunya, Barcelona, España, noviembre 1996. Citado en p. 18, 22, 89

[113] M. Sbert. Error and Complexity of Random Walk Monte Carlo Radiosity. *IEEE Transactions on Visualization and Computer Graphics*, 3(1):23–38, 1997. Citado en p. 18, 89

[114] K. Schloegel, G. Karypis, y V. Kumar. *Sourcebook of Parallel Computing*, capítulo Graph Partitioning for High-Performance Scientific Simulations, páginas 491–541. Morgan Kaufmann Publishers Inc., San Francisco, USA, 2003. Citado en p. 39

[115] A. Sharpe, M. Hampton, S. Nirenstein, J. Gain, y E. Blake. Accelerating Ray Shooting Through Aggressive 5D Visibility Preprocessing. En *Proceedings*

of the 2nd International Conference on Computer Graphics, Virtual Reality, Visualization and Interaction in Africa (AFRIGRAPH 2003), volumen 30, páginas 95–100, 2003. Citado en p. 37

[116] M. Shevtsov, A. Soupikov, y A. Kapustin. Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of Dynamic Scenes. *Computer Graphics Forum*, 26(3):395–404, 2007. Citado en p. 139

[117] P. Shirley. A Ray Tracing Method for Illumination Calculation in Diffuse-Specular Scenes. En *Proceedings of Graphics Interface '90*, páginas 205–212, Toronto, Canadá, 1990. Canadian Information Processing Society. Citado en p. 19, 89

[118] P. Shirley. Radiosity via Ray Tracing. En *Graphics Gems II*, páginas 306–310. Academic Press Professional, 1991. Citado en p. 19, 89

[119] P. Shirley. Time Complexity of Monte Carlo Radiosity. *Computers and Graphics*, 16(1):117–120, 1992. Citado en p. 19, 89

[120] F. Sillion y J.-M. Hasenfratz. Efficient Parallel Refinement for Hierarchical Radiosity on a DSM Computer. En *Proceedings of Third Eurographics Workshop on Parallel Graphics and Visualization*, páginas 61–74, 2000. Citado en p. 27

[121] F. X. Sillion y C. Puech, editores. *Radiosity and Global Illumination*. Morgan Kaufmann Publishers, Inc., San Francisco, U.S.A., 1994. Citado en p. 6, 9, 11, 19

[122] B. W. Silverman. *Density Estimation for Statistics and Data Analysis*. Chapman & Hall/CRC, abril 1986. Citado en p. 24, 25

[123] G. Simiakakis. *Accelerating Ray Tracing with Directional Subdivision and Parallel Processing*. Tesis doctoral, University of East Anglia, Norwich, UK, 1995. Citado en p. 37

[124] J. P. Singh. *Parallel Hierarchical N-body Methods and their Implications for Multiprocessors*. Tesis doctoral, Stanford University, Stanford, USA, 1993. Citado en p. 12

[125] J. P. Singh, A. Gupta, y M. Levoy. Parallel Visualization Algorithms: Performance and Architectural Implications. *Computer*, 27(7):45–55, 1994. Citado en p. 27

[126] B. Smits, J. Arvo, y D. Greenberg. A Clustering Algorithm for Radiosity in Complex Environments. En *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'94)*, páginas 435–442, New York, USA, julio 1994. ACM. Citado en p. 7, 12

[127] B. E. Smits, J. R. Arvo, y D. H. Salesin. An Importance-Driven Radiosity Algorithm. *ACM SIGGRAPH Computer Graphics (Proc. of SIGGRAPH '92)*, 26(2):273–282, julio 1992. Citado en p. 7

[128] W. Stürzlinger y C. Wild. Parallel Progressive Radiosity with Parallel Visibility Computations. En *Proceeding of The Second International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG 1994)*, páginas 66–74, Plzen, República Checa, enero 1994. University of West Bohemia. Citado en p. 27, 28

[129] I. E. Sutherland y G. W. Hodgman. Reentrant Polygon Clipping. *Communications of the ACM*, 17(1):32–42, enero 1974. Citado en p. 45, 66

[130] I. Wald, C. Benthin, A. Dietrich, y P. Slusallek. Interactive Distributed Ray Tracing on Commodity PC Clusters – State of the Art and Practical Applications. *Lecture Notes on Computer Science*, 2790:499–508, 2003. (Proceedings of EuroPar 2003). Citado en p. 27

[131] I. Wald, C. Benthin, M. Wagner, y P. Slusallek. Interactive rendering with coherent ray tracing. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001)*, 20(3):153–164, 2001. Citado en p. 98, 105

[132] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, y A. Moshovos. Demystifying GPU Microarchitecture through Microbenchmarking. En *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2010)*, páginas 235–246, White Plains, USA, marzo 2010. Citado en p. 135

[133] D. Zareski, B. Wade, P. Hubbard, y P. Shirley. Efficient Parallel Global Illumination Using Density Estimation. En *Proceedings of the IEEE Symposium on Parallel Rendering (PRS'95)*, páginas 47–54, New York, USA, 1995. ACM. Citado en p. 28

[134] Y. Zhang y J. D. Owens. A Quantitative Performance Analysis Model for GPU Architectures. En *Proceedings of the 17th IEEE International Symposium*

on High-Performance Computer Architecture (HPCA 17), páginas 382–393, febrero 2011. Citado en p. 130

[135] K. Zhou, Q. Hou, R. Wang, y B. Guo. Real-time KD-tree Construction on Graphics Hardware. *ACM Transactions on Graphics (TOG)*, 27:artículo 126, 11 páginas, diciembre 2008. Citado en p. 140