Instituto de Computación
Facultad de Ingeniería Universidad de la República
Montevideo – Uruguay

ACELERACIÓN DEL CÁLCULO DE LA MATRIZ DE FACTORES DE FORMA UTILIZANDO VISIBILIDAD JERÁRQUICA

Joel Vázquez
Pablo Guartes

Tutores:
Eduardo Fernández
José Pedro Aguerre

Proyecto de Grado

Diciembre 2017

Resumen

Los algoritmos de iluminación global son una categoría de algoritmos que permiten computar la luz indirecta en escenas complejas. El método de radiosidad es un algoritmo que resuelve la ecuación de la luz mediante la aplicación de elementos finitos, donde cada elemento (o parche) es considerado una superficie de reflexión perfectamente difusa. La etapa del algoritmo más costosa desde el punto de vista computacional consiste en calcular la matriz de factores de forma. Esta matriz ($\mathbf{F}$) es cuadrada con respecto a la cantidad de parches en la escena. $\mathbf{F}_{ij}$ define la fracción de energía que sale desde el parche i e incide directamente en el parche j. Entre los métodos posibles para el cálculo de $\mathbf{F}$, se encuentra el basado en el método del hemicubo. Este método se basa en la proyección de la escena sobre medio cubo, lo que se usa para calcular una fila de $\mathbf{F}$. Para el cálculo de los hemicubos se utilizan algoritmos de rasterización (o pixelado) acelerados por GPU.

En este proyecto se estudia la aceleración del método del hemicubo para escenas con un alto factor de oclusión, como puede ser el modelo geométrico de una ciudad, donde un elemento de la escena en promedio "ve" una porción muy reducida de la escena. Escenas con estas características generan matrices $\mathbf{F}$ poco densas (con muchos valores iguales a 0). Para conseguir los objetivos planteados, se aplica un algoritmo de visibilidad jerárquica que permite determinar si una superficie está ocluida o no antes de ser dibujada. Esto se realiza mediante la utilización de una estructura jerárquica para agrupar los objetos de la escena. Los resultados obtenidos fueron comparados con otra implementación del método del hemicubo sin técnicas de aceleración, obteniéndose una aceleración de hasta 3,5× en escenas de más de cien mil parches.

Palabras clave: *Radiosidad, Factores de forma, OpenGL, Z-Buffer jerárquico.*

Índice

Capítulo 1

Introducción

Simular de forma realista la luz en una escena representa un problema no trivial. En términos físicos, la luz [1] es radiación electromagnética formada por partículas sin masa denominadas fotones. La complejidad de dicha simulación se debe a que estos fotones interactúan con los objetos de la escena de variadas formas. Esto se debe a que las diferentes superficies actúan de forma distinta ante frecuencias o espectros distintos de luz. A su vez, fenómenos como la reflexión y la refracción implican un cambio en la dirección del fotón al interactuar con las superficies, usualmente de forma no determinista.

Las superficies con las que la luz interactúa repercuten en el resultado de la imagen percibida. Para simplificar el modelado de la reflexión, se suele considerar que hay sólo dos tipos de superficies: las superficies especulares y las difusas [2]. Las superficies especulares son perfectamente lisas a un nivel microscópico. Una superficie de perfecta reflexión especular, refleja la luz de modo que el ángulo que forma el rayo incidente con la normal, es igual al ángulo que se forma entre el rayo reflejado y la normal. Las superficies difusas, en contrapartida, son superficies rugosas, y como resultado la luz se refleja en diferentes direcciones. Una superficie de reflexión difusa perfecta, refleja la luz en todas direcciones de forma determinista y fácilmente modelable (Figura 2.2).

Los algoritmos de iluminación global [2] son una categoría de algoritmos que permiten computar la luz indirecta en escenas complejas. Dentro de ellos la radiosidad [3] es un algoritmo de iluminación global para escenas con superficies difusas. Este algoritmo toma en cuenta no solo la fuente de luz, sino la interacción entre las superficies de la escena. Para esto se realiza del cálculo de factores de forma. El factor de forma desde una superficie i a una superficie j, $\mathbf{F}_{ij}$, es la fracción de la energía que sale de la superficie i e incide directamente en la superficie j. Se utiliza ampliamente para resolver problemas tanto lumínicos como térmicos [4], en áreas como la ingeniería industrial, la arquitectura, el diseño 3D [5] y diseño de videojuegos [6].

El método del hemicubo se utiliza para calcular los factores de forma. Esta técnica consiste en la construcción de la mitad de un cubo por parche o superficie de la escena, con centro en su baricentro. Dado un parche cualquiera i, se construye el hemicubo centrado en el baricentro de i, se proyecta la escena sobre él, y esa proyección se usa para el cálculo de la fila i de $\mathbf{F}$, es decir, se calculan todos los $\mathbf{F}_{ij}$ para todos los parches j de la escena. El cálculo de los factores de forma es computacionalmente costoso, y

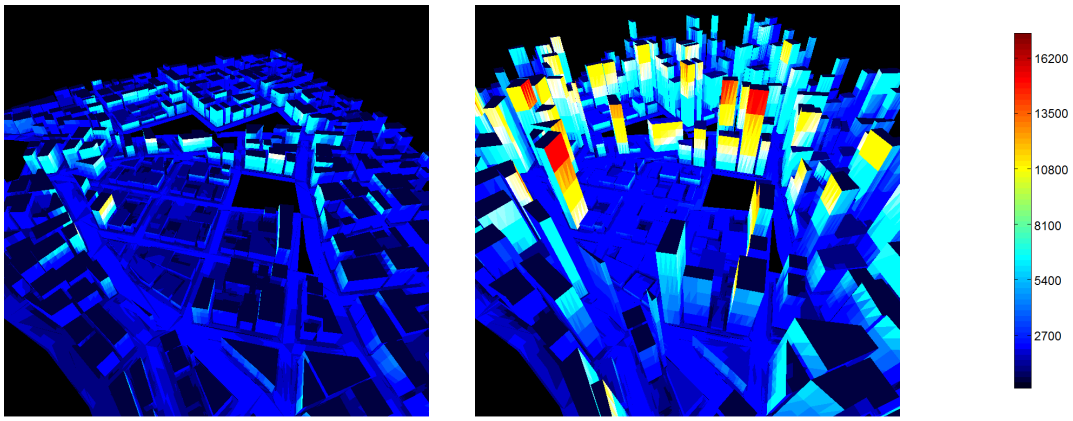


Figura 1.1: Dos ejemplos de escenas urbanas. El color de cada parche indica la cantidad de parches que son vistos desde este. Extraído de [9]

por esto se requiere el uso de técnicas que permitan utilizar la aceleración mediante el procesador gráfico (GPU), para poder trabajar con escenas complejas que contengan una gran cantidad de parches.

En el marco de este proyecto se maneja como objetivo principal estudiar la viabilidad de combinar las técnicas del hemicubo (Cohen *et al.* [7]) y del Z-Buffer jerárquico (Greene *et al.* [8]). Se pretende estudiar desde el punto de vista teórico y práctico las mismas y algunas variantes de su implementación. Se busca acelerar el cálculo de cada hemicubo en escenas de alta oclusión; un ambiente ocluido puede entenderse como aquel en el que si se coloca una cámara en la escena, la misma tendrá una alta probabilidad de captar una porción pequeña del modelo total. Teniendo esto en cuenta, se deduce que una ciudad es un ejemplo de una escena altamente ocluida, como se puede ver en la Figura 1.1. Un aspecto muy importante a explorar es que la alta oclusión de las ciudades tiene una coherencia espacial. Esto se puede definir de la siguiente manera: dados dos parches cercanos i_1 e i_2, y otro parche cualquiera j, es muy probable que si i_1 no ve a j, i_2 tampoco vea a j (por ejemplo, dos parches de la misma pared probablemente no ven a otro parche de un edificio que está a varias manzanas). Por tanto, puede resultar útil conocer la información de qué parches ve i_1, para computar cuáles son los parches vistos por i_2. Para esto se estudiará la utilidad de aprovechar las técnicas de filtrado por oclusión (*Occlusion Culling*).

También dentro de los principales objetivos de este trabajo se encuentra optimizar el uso de los recursos disponibles a la hora de resolver el problema en cuestión. Entre ellos se encuentran la optimización a la hora de operar con la tarjeta gráfica, minimizar la cantidad de transferencias entre CPU y GPU, así como también la cantidad de llamadas a la GPU a través de OpenGL, siempre priorizando el aumento del rendimiento y la reducción del tiempo de procesamiento de las escenas.

Se realizará un estudio comparativo entre la implementación del método del hemicubo sin utilizar una estructura jerárquica para el cálculo de profundidades y otra implementación que cuente con dicha optimización, en el que se demostrará la eficiencia de las optimizaciones propuestas, alcanzando aceleraciones de hasta $3,5\times$ para escenas

urbanas de mediano y gran porte.

El documento posee una primera parte donde se detallan los trabajos que forman la base del algoritmo en la sección de **Estado del arte**. A continuación de la misma se describe el **Alcance del proyecto** seguido de la **Solución propuesta** para culminar con el **Análisis experimental** y las **Conclusiones**.

Capítulo 2

Estado del Arte

En esta sección se realiza una breve introducción a los conceptos y técnicas más importantes que constituyen este trabajo, así como también se resumen trabajos previos que se utilizaron como referencia para construir las bases del algoritmo diseñado.

Las técnicas de iluminación global se utilizan para modelar la interacción de la luz con la escena. En particular, técnica de radiosidad se describe porque en ella se utiliza la matriz de factores de forma **F**. Por tanto, se describe el sistema lineal que contiene a **F** y el cálculo de los elementos de **F**, poniendo especial atención a la técnica del hemicubo. Luego se describen técnicas de procesamiento gráfico, basadas en el uso del pipeline gráfico, y aceleradas por las GPU. A continuación se describen las matrices dispersas, elemento clave para almacenar matrices **F** enormes debido a la gran cantidad de 0 que tiene en escenas con alta oclusión. Posteriormente, se describe la técnica de visibilidad jerarquiza del Z-Buffer (HZB), que permite acelerar la vista de las escenas de alta oclusión. Por último, se describe la Z-Curva, que se utiliza para ordenar los parches de la escena por cercanía y orientación. Esto último es clave para aprovechar la coherencia especial en relación a la oclusión.

2.1. Iluminación Global

Los algoritmos de iluminación [10] trabajan sobre un conjunto de modelos (Figura 2.1) tridimensionales de entrada que representan una escena. Un modelo de este tipo está definido por un conjunto de puntos denominados vértices, que determinan polígonos (o parches). Un polígono es un área formada por al menos tres vértices. Sobre cada polígono también suele incorporarse otra información como coordenadas de textura, información de color y vector normal.

Un modelo de iluminación global determina la iluminación de un punto en términos de la luz emitida por diversas fuentes de luz que llega de forma directa al punto, y también por luz que llega al mismo mediante reflejos o transmisiones a través de otras superficies [11]. La luz que es indirectamente reflejada y transmitida es frecuentemente llamada iluminación global. En contrapartida, la luz llega al punto de forma directa es llamada iluminación local. Gran parte de la luz en ambientes del mundo real no

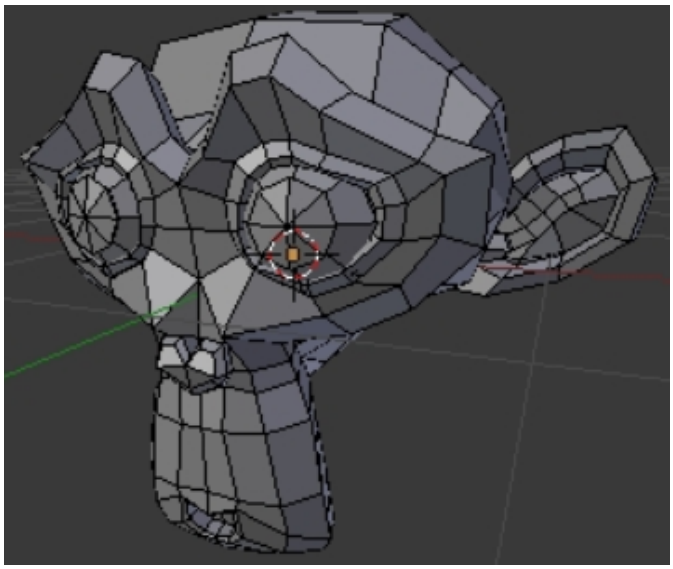


Figura 2.1: Ejemplo de modelo 3D formado por polígonos.

proviene de fuentes directas. Por lo tanto, un modelo de iluminación completo debe tener en cuenta las fuentes indirectas como son reflexiones especulares, incidencia de la luz difusa, luz refractada y luz ambiental.

La reflexión de la luz se comporta de manera distinta en diferentes tipos de superficies. En superficies totalmente lisas a un nivel microscópico la reflexión se conoce como reflexión especular (Figura 2.2 , izquierda), donde el ángulo incidente es igual al ángulo reflejado. En las superficies rugosas lo que sucede es que los rayos se reflejan en múltiples direcciones (Figura 2.2, derecha). Esto ocurre debido a las macro o micro rugosidades que desvían la luz en distintos ángulos. Una superficie de perfecta reflexión difusa, refleja la luz en todas direcciones con respecto a la ley de Lambert [12]. La cual establece que la intensidad de radiación percibida en una superficie perfectamente difusa es independiente del observador y su valor es directamente proporcional al coseno del ángulo entre el vector de la luz normalizado (con dirección desde la superficie hacia la fuente de luz) y la normal a la superficie.

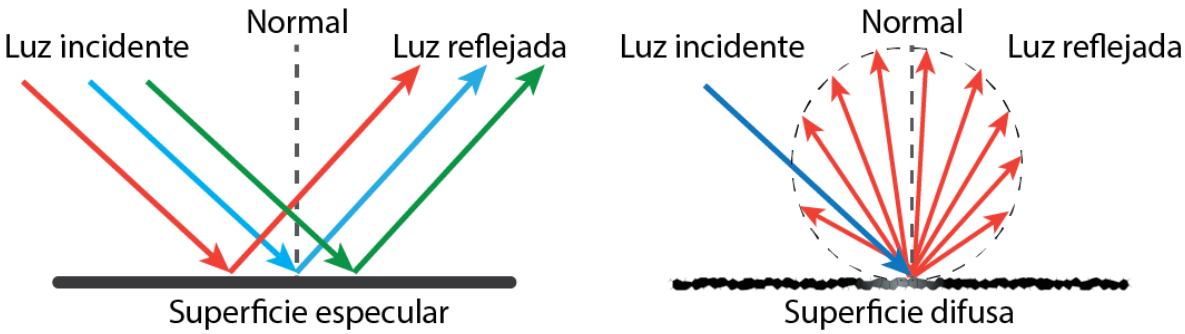


Figura 2.2: Reflexión en superficies difusas y especulares.

Entre los dos extremos de reflexiones especulares y difusas, existe un conjunto infi-

nito de posibilidades, porque cada superficie y material tiene propiedades de reflexión propias. Existen materiales opacos, brillosos, espejados, traslúcidos, coloreados, entre otros. También, los materiales presentan más de una propiedad a la vez. Por ejemplo, un material puede ser parcialmente difuso y parcialmente especular. Por otra parte, la reflexión difusa tiene la propiedad particular de que es independiente del punto de vista del observador: desde cualquier punto de vista se observa la misma intensidad de luz reflejada por una superficie. Esta propiedad facilita su modelado y simulación computacional. Al ser la reflexión difusa muy común en la naturaleza y en la arquitectura (el cemento y la cal se pueden modelar con reflexión difusa), se generaron dos tipos de algoritmos principales.

Este problema abrió el paso a dos tipos de algoritmos principales. Los algoritmos del primer tipo son los dependientes del punto de vista del observador, en los que se calcula para cada punto la luz incidente por distintos fenómenos, como rayos reflejados, refractados , luz ambiental y también fuentes de luces directas, permitiendo representar realísticamente fenómenos que son muy dependientes de la posición del observador como reflejos en superficies especulares. Algunos ejemplos de este tipo de algoritmo son *ray tracing*, *path tracing* y *photon mapping*. La principal desventaja de estos métodos es que pueden requerir cierto trabajo extra al modelar los fenómenos difusos. Los algoritmos independientes del punto de vista del observador discretizan el ambiente y lo procesan de manera de proveer información suficiente para calcular la iluminación de la escena desde cualquier punto de vista. Este tipo de algoritmos modelan de una forma natural los fenómenos difusos pero requieren una mayor cantidad de memoria para poder almacenar información acerca de la interacción entre cada par de superficies en la escena. Ambos tipos de algoritmos convergen a resolver la ecuación de renderizado enunciada por Kajiya [11], que expresa la luz transferida de un punto a otro en términos de intensidad de la luz emitida de un punto al otro y la intensidad de la luz emitida por todos los otros puntos y que llegan al primero tras ser reflejada por el segundo. La luz transferida de todos los otros puntos al primero es expresada recursivamente por la ecuación de renderizado presentada por Kajiya, que puede ser observada en la Ecuación 2.1.

$$I\left(x,x'\right)=g\left(x,x'\right)\left[\varepsilon\left(x,x'\right)+\int_{s}\rho\left(x,x',x''\right)I\left(x',x''\right)dx''\right] \tag{2.1}$$

En esta ecuación, x, x' y x'' son puntos de la escena.a $I(x,x')$ representa la intensidad pasando desde x' a x. $g(x,x')$ es un término geométrico que se convierte en 0 cuando x y x' están completamente ocluidos uno del otro, y en $1/r^2$ cuando se ven totalmente, donde r es la distancia entre ellos. $\varepsilon\ (x,x')$ es la intensidad de luz que es emitida desde x' a x. La evaluación inicial de $g(x,x')\ \varepsilon\ (x,x')$ siendo x el punto de vista elegido da como resultado la determinación de las superficies visibles en la esfera de visión sobre x. La integral continua se realiza sobre todos los puntos en todas las superficies S. $\rho\ (x,x',x'')$ representa la intensidad de luz reflejada(incluyendo reflexión especular y difusa) de x'' a x desde la superficie desde la superficie x'. Así, la ecuación establece que la luz desde x' que llega a x consiste de la luz emitida por el mismo x' y la luz dispersada por x' hacia x proveniente de todas las demás superficies, que emiten luz

por si mismas y recursivamente dispersan luz a otras superficies.

La Ecuación 2.1 es la base de los algoritmos fotorrealistas. Lamentablemente, casi nunca es posible el cálculo de la solución exacta, salvo para escenas muy simples. Debido a este problema se han ideado una gran cantidad de métodos con el objetivo de aproximarse a la solución de formas mucho más prácticas. En la siguiente lista se describen algunos de los algoritmos más efectivos:

- **Ray tracing** [13] - Este método consiste en trazar rayos desde el observador e intersectar cada uno de ellos con la escena. Luego de que se intersecta el primer objeto, en la ecuación de renderizado 2.1 se tienen en cuenta el material del objeto y las posiciones de las luces para determinar el color final del píxel, luego se calcula su nueva trayectoria y el rayo continúa hasta una cota máxima de reflexiones o de distancia recorrida sin intersecciones.

- **Path tracing** [14] - Es una técnica muy efectiva y fiel a la Ecuación 2.1, que aproxima numéricamente utilizando métodos de Monte Carlo, en el cual se toman muestras aleatorias para cada píxel de los objetos. En el algoritmo se van creando caminos de luz a medida que los rayos rebotan en los objetos. Posee dos formas de generar los rayos, desde las fuentes de luz emitir rayos y crear caminos (*paths*) por la escena, o desde un punto de una superficie acumular rayos entrantes. Cuando se utilizan las dos técnicas se denomina path tracing bidireccional. Al igual que en ray tracing se tienen en cuenta los materiales de los objetos. Las imágenes generadas son fotorrealistas y se utilizan muchas veces como ejemplo para comparar con el resultado de otros algoritmos de iluminación global menos costosos.

- **Photon mapping** [15] - Al igual que el algoritmo de path tracing, es un método de Monte Carlo. Como primer paso se construye un mapa de fotones partiendo de las fuentes de luz. Cuando un fotón impacta con un objeto, es almacenado el punto de intersección junto con la dirección del fotón en un caché. Luego se determina aleatoriamente y con probabilidades que dependen del material del objeto intersectado, si el fotón es reflejado, refractado o absorbido. Una vez que el mapa de fotones ha sido construido, se procede a realizar ray tracing sobre todos los píxeles de la pantalla. Se halla la intersección con el objeto más cercano y se procede a calcular la Ecuación 2.1 de forma simplificada en: luz directa, luz indirecta, reflexiones especulares y cáusticas. Para la luz indirecta y las cáusticas se utiliza el mapa de fotones generado y para la luz directa y las reflexiones especulares se combina el mapa de fotones con *ray tracing*.

2.2. Radiosidad

El método de Radiosidad modela la interacción de luz entre superficies difusas [3]. Su estudio es de suma importancia en áreas como el diseño, la animación 3D, así como en el área de la radiación térmica [16]. Por ejemplo, Frostbite [6] es un motor gráfico muy potente que utiliza esta técnica (en conjunto con otras técnicas de iluminación),

al igual que el motor de renderizado Mental Ray, disponible para 3DS Max y Maya [5]. En las próximas secciones se revisarán las bases del algoritmo del hemicubo propuesto por Cohen *et al.* [7], una solución al problema de radiosidad para ambientes complejos, el cual toma como referencia el trabajo realizado por Goral *et al.* [3] previamente.

En esencia, cada objeto de la escena es tratado como una fuente de luz secundaria. Este método provee una representación de la luz difusa y ambiente muy precisa, contemplando fenómenos como cambios en el sombreado, penumbras en los bordes de sombra o sangrado.

Cabe destacar que los cálculos son independientes de la posición del observador por lo que permite un renderizado eficiente desde múltiples vistas de la misma escena en secuencias dinámicas.

2.2.1. Cálculo de radiosidad

El método de radiosidad propone un equilibrio de energía dentro de la escena. Como se mencionó anteriormente, se asume que todas las emisiones y procesos de reflexión son difusos ideales.

Radiosidad es el nombre de la técnica, pero también es la medida de la potencia de luz emitida o reflejada por unidad de superficie. Se suele utilizar como unidad de radiosidad al W/m^2. De la Ecuación 2.1 se deriva la Ecuación 2.2 asumiendo todas las superficies difusas.

$$B(x) = E(x) + p(x) \int_S B(x')G(x, x')dA' \tag{2.2}$$

- S: superficies en la escena
- x, x': dos puntos de S
- dA': diferencial del área en x'
- B (Radiosidad): el flujo total de energía que sale del punto. Suma de la energía emitida y la energía reflejada ($\frac{W}{m^2}$)
- E (Emisión): el flujo con el cual el punto emite energía (luz) ($\frac{W}{m^2}$)
- p (Reflectividad): la fracción de luz incidente que no es absorbida y es reflejada de nuevo hacia la escena (sin unidad)
- G (Término geométrico): es la relación geométrica entre dos puntos x e x' que representa la fracción de energía que sale de x e impacta en x'. G toma en cuenta la visibilidad entre los puntos y la distancia (sin unidad)

Aplicando el método de elementos finitos, la escena se subdivide en N elementos (de ahora en más "parches") para los cuales se asume una radiosidad constante. De esta manera se puede obtener la Ecuación 2.3 (una deducción detallada se encuentra en [17]). En la Figura 2.4 se puede observar la subdivisión de una escena en parches.

$$B_i = E_i + p_i \sum_{j=1}^{N} B_j F_{ij} \tag{2.3}$$

- B_i (Radiosidad): el flujo total de luz por unidad de área que abandona el parche i. Suma del flujo por unidad de área emitido y reflejado. ($\frac{W}{m^2}$)
- E_i (Emisión): es el flujo por unidad de área emitido ($\frac{W}{m^2}$)
- p (Reflectividad): la fracción de luz incidente que no es absorbida por la superficie y es reflejada de nuevo hacia la escena (sin unidad)
- F_{ij} (Factor de forma): La fracción de la energía que sale de una superficie i y que llega a otra superficie j de manera directa (sin unidad)

Esta ecuación establece que la cantidad de flujo de luz saliente por unidad de área (o sea radiosidad) es igual al flujo de luz por unidad de área emitido más el reflejado. Esta luz reflejada es igual a la luz saliente de todas las demás superficies multiplicadas por la fracción (Factor de Forma) que llega a la superficie en cuestión y por la reflectividad de la superficie receptora (véase la Figura 2.3).

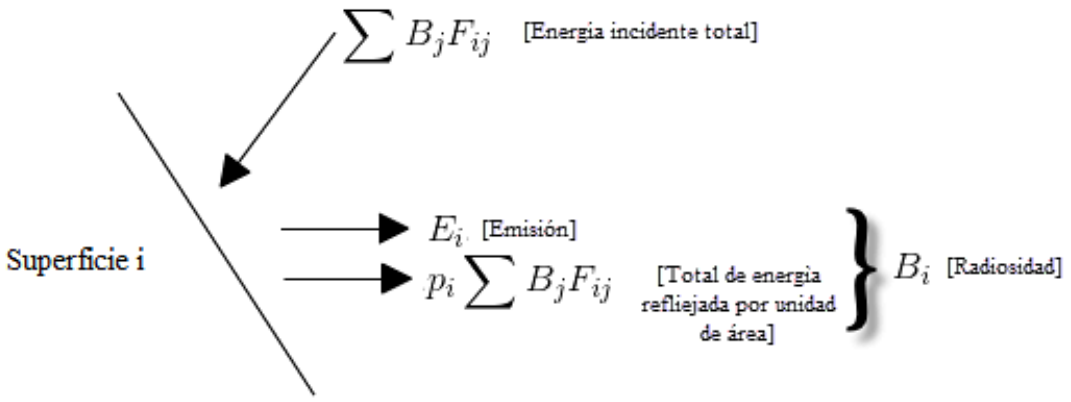


Figura 2.3: Ilustración radiosidad.

Esta discretización genera un sistema de ecuaciones lineales capaz de describir la interacción de la luz en la escena:

$$(I - RF)B = E \tag{2.4}$$

- I: es la matriz identidad
- R: es una matriz diagonal con $R(i,i) = p_i$
- **F**: es la matriz con los factores de forma

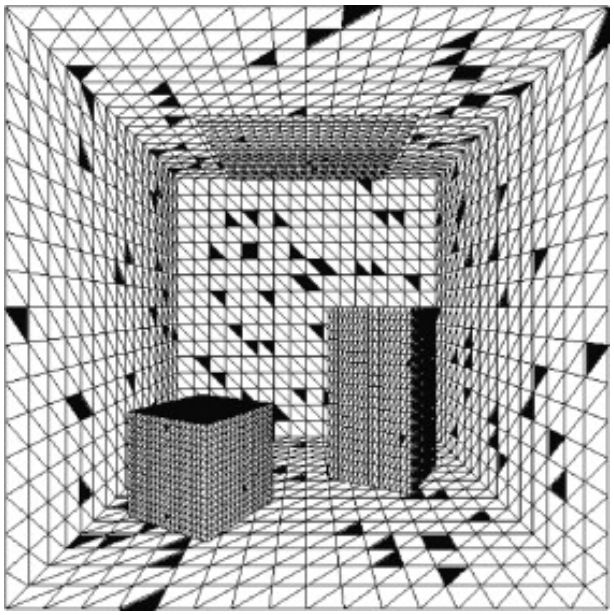


Figura 2.4: Cornell box discretizado. Extraído de [18].

- B: es el vector con la radiosidad de cada parche
- E: es el vector con las emisiones de los parches

Expandiendo cada término en la Ecuación 2.4:

$$\begin{bmatrix} 1-p_1F_{11} & -p_1F_{12} & . & . & -p_1F_{1N} \\ -p_2F_{21} & 1-p_2F_{22} & . & . & -p_2F_{2N} \\ . & . & & & . \\ . & . & & & . \\ -p_NF_{N1} & -p_NF_{N2} & . & . & 1-p_NF_{NN} \end{bmatrix} \begin{bmatrix} B1 \\ B2 \\ . \\ . \\ B_N \end{bmatrix} = \begin{bmatrix} E1 \\ E2 \\ . \\ . \\ E_N \end{bmatrix} \tag{2.5}$$

El color de un objeto es determinado por su reflectividad (o emisión en el caso de una fuente de luz) en cada longitud de onda del espectro visible, también llamadas bandas. La reflectividad y emisión en las ecuaciones es válida para una longitud de onda particular. Es necesario determinar y resolver la matriz para cada longitud de onda para poder determinar la radiosidad de cada parche. Es importante notar que se asume que cada parche tiene una radiosidad constante B, a pesar de que esta puede ser un conjunto de tres o más valores que representan la radiosidad en las diferentes bandas. También se destaca que los factores de forma son solamente una función geométrica y por lo tanto independientes de cualquier consideración de color.

La resolución de la serie de ecuaciones 2.5 se resuelve típicamente con un método iterativo como puede ser Gauss-Siedel o Jacobi. Se propone un valor inicial y se itera hasta satisfacer alguna de las condiciones de parada.

2.2.2. Factores de forma

El factor de forma define la fracción de energía que sale de una superficie e impacta contra otra. Por definición, la suma de todos los factores de forma para un parche particular es igual a la unidad. Los términos geométricos de los factores de forma están representados en la Figura 2.5, en la que r es la distancia entre el diferencial de área del parche i y el del parche j. Los vectores normales a los parches están dados por N_i y N_j mientras que ϕ_i y ϕ_j representan el ángulo entre la normal y el vector distancia r. Para ambientes sin oclusión el factor de forma entre áreas diferenciales viene dado por la Ecuación 2.6.

$$F_{dA_i dA_j} = \frac{\cos\phi_i \cos\phi_j}{\pi r^2} \tag{2.6}$$

Existen distintos tipos soluciones para el cálculo de factores de forma y se clasifican según su forma de calcularlos [17]. Se dividen en analíticos y numéricos. A su vez, dentro de los numéricos tenemos:

- **Área diferencial a área**. En este tipo de algoritmos se proyectan las áreas de los elementos j sobre un área diferencial para los elementos i, el resultado es evaluado en uno o más puntos de la superficie diferencial A_i y luego se hace un promedio. Esto dio lugar a distintas implementaciones como el método del plano singular o el método del hemicubo el cual es una de las bases de este proyecto. Son eficientes cuando se necesita calcular todos los factores de forma desde un mismo punto hacia todos los elementos al mismo tiempo.
- **Área a área**. Para poder computar los factores de forma de un área a otra completamente, sin realizar promedios como en los área diferencial a área, se tuvieron que idear aproximaciones eficientes para el cálculo de la integral. Métodos de Monte Carlo o de integración de contorno entran en esta clasificación [17].

En particular en el hemicubo el cálculo de estos factores de forma se realiza de área diferencial a área, sin embargo existen distintos métodos con distintos enfoques de cómo resolver la integral de la Ecuación 2.8.

Cuando se trabaja sin oclusión los factores de forma entre dos superficies pueden ser representados mediante la siguiente expresión:

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos\phi_i \cos\phi_j}{\pi r^2} dA_j dA_i \tag{2.7}$$

Si se quiere agregar la posibilidad de que un objeto pueda ocluir a otro entonces se debe agregar un término más a la ecuación:

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos\phi_i \cos\phi_j}{\pi r^2} \mathrm{HID(i,j)} dA_j dA_i \tag{2.8}$$

La función HID toma valor uno o cero dependiendo si el área diferencial i "puede ver" al área diferencial j. La solución analítica es compleja desde el punto de vista computacional debido al factor de oclusión (HID). En general se utilizan métodos numéricos

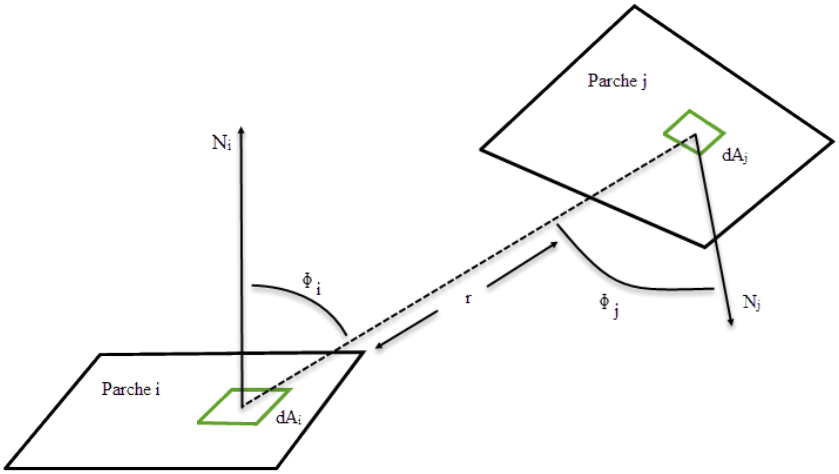


Figura 2.5: Geometría de factores de forma.

para aproximar los factores de forma de los parches. Nusselt obtuvo un análogo de la integral de factores de forma, la *Analogía de Nusselt* [16], que consiste en tomar una fracción del círculo cubierto proyectando el área en un hemisferio y luego ortográficamente hacia el círculo (Figura 2.6). Tanto crear elementos de igual tamaño sobre una esfera como crear un conjunto de coordenadas lineales para describir ubicaciones sobre su superficie resultan poco prácticos, es por esto que se creó un enfoque más apropiado que aproveche el hardware gráfico al utilizar superficies planas, el hemicubo.

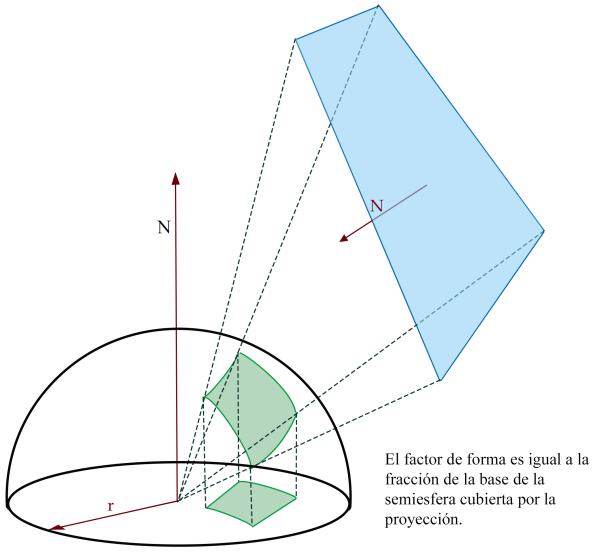


Figura 2.6: Analogía de Nusselt - Aproximación de cálculo de factores de forma.

2.2.3. El hemicubo

De la definición de factor de forma, se puede ver que si dos parches del ambiente que son proyectados en la semiesfera ocupan la misma área y ubicación, entonces tendrán el mismo factor de forma. Esto también es cierto para proyecciones sobre cualquier otro tipo de superficie.

En vez de proyectar sobre una esfera, un cubo imaginario es construido alrededor del centro del parche receptor (Figura 2.7). Esto permite aprovechar mejor las bibliotecas gráficas al tratarse de caras planas. Una de las técnicas ya implementadas en la GPU que se aprovecha es la del Z-Buffer (ver Sección 2.3.1), un algoritmo ampliamente utilizado en el área de computación gráfica que permite determinar cuáles objetos de la escena son visibles y cuáles están ocultos. Se coloca la cámara en el centro del parche en el origen con la normal del mismo coincidiendo con el eje Z positivo. En esta orientación, la semiesfera descrita anteriormente es reemplazada por la mitad de la superficie de un cubo. En la Figura 2.8 se aprecia un sistema de coordenadas alineado al hemicubo, el cual se divide en píxeles cuadrados, donde el eje Z es paralelo a la normal al parche, y los ejes X e Y están en el mismo plano del parche. Por tanto, el hemicubo se construye con una cara completa en el plano Z=1, y 4 medias caras en los planos X=1, X=-1, Y=1 e Y=-1.

En la Figura 2.9 se muestra el resultado de proyectar sobre el hemicubo de un parche la escena correspondiente a una de las ciudades que se utilizarán en el análisis experimental.

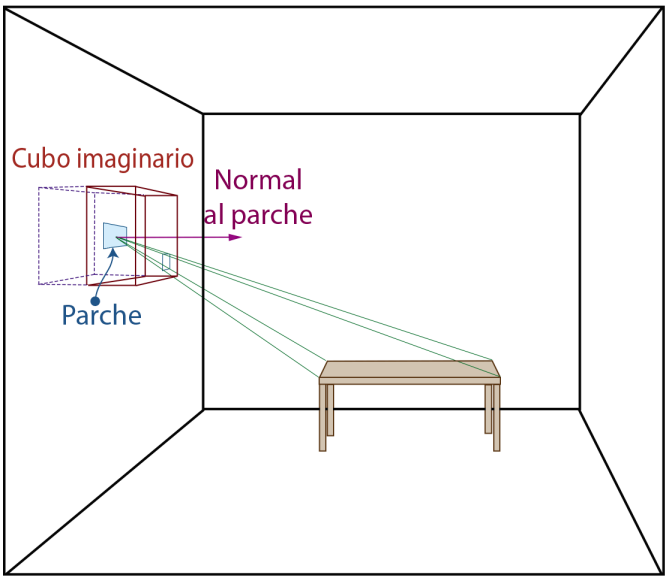


Figura 2.7: Representación visual del hemicubo. Un cubo imaginario es creado alrededor del centro del parche. Todos los demás parches del ambiente son proyectados en este cubo.

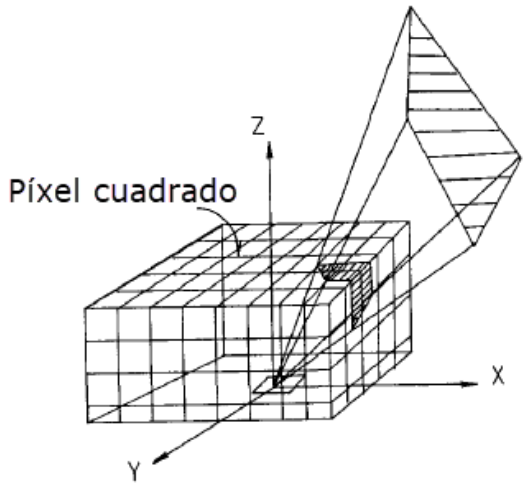


Figura 2.8: Subdivisión en píxeles del hemicubo. Extraído de [7].

Figura 2.9: Proyección de la escena de una ciudad en un hemicubo.

2.2.4. Cálculo de los factores de forma utilizando el hemicubo

Como se aprecia en la Figura 2.8, el hemicubo está formado por píxeles. Cada píxel tiene un valor de factor de forma, llamado diferencial de factor de forma o ΔF. La contribución de cada píxel de la superficie del cubo al valor de factores de forma, varía y es dependiente de la ubicación y orientación del píxel. En la Figura 2.11 se muestra la deducción de las ecuaciones del hemicubo a partir de la Ecuación 2.6. Con esta ecuación se obtiene un valor específico de los factores de forma de área diferencial a área diferencial para cada píxel del cubo, luego estos se guardan en una estructura de

hash[1] para fácil acceso. En la Figura 2.10 se encuentra representado un hemicubo con las caras laterales llevadas al mismo plano que la cara superior. Se observa en verde la cara superior y en azul las caras laterales. El verde oscuro representa la porción que debe mantenerse y que luego puede ser simetrizada en el resto de la cara superior. El azul oscuro representa la porción que debe mantenerse y que luego puede ser simetrizada en el resto de las caras laterales. Los píxeles simétricos se deducen fácilmente de las ecuaciones del hemicubo, mostradas en la Figura 2.11. En la cara superior por ejemplo, al aplicar la función a todo píxel (x,y), el resultado es el mismo que con (-x,y) , (x,-y),(-x,-y),(y,x),(-y,x),(y,-x) y (-y,-x). De la misma manera se pueden deducir los simétricos de los píxeles laterales. Dadas estas propiedades de simetría, se observa que alcanza con calcular 1/8 de todos los píxeles, ya que el resto poseen los mismos valores.

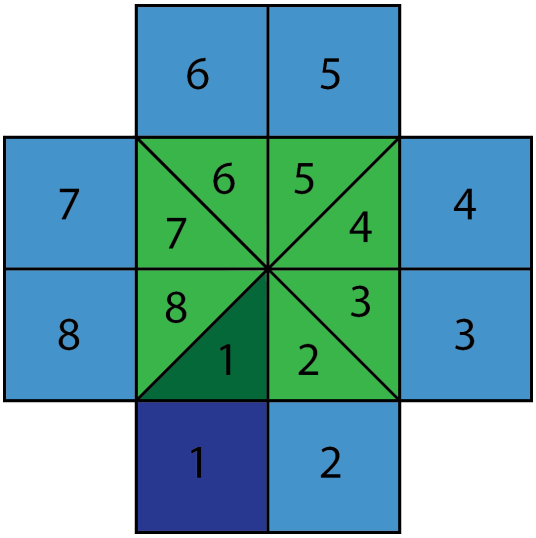


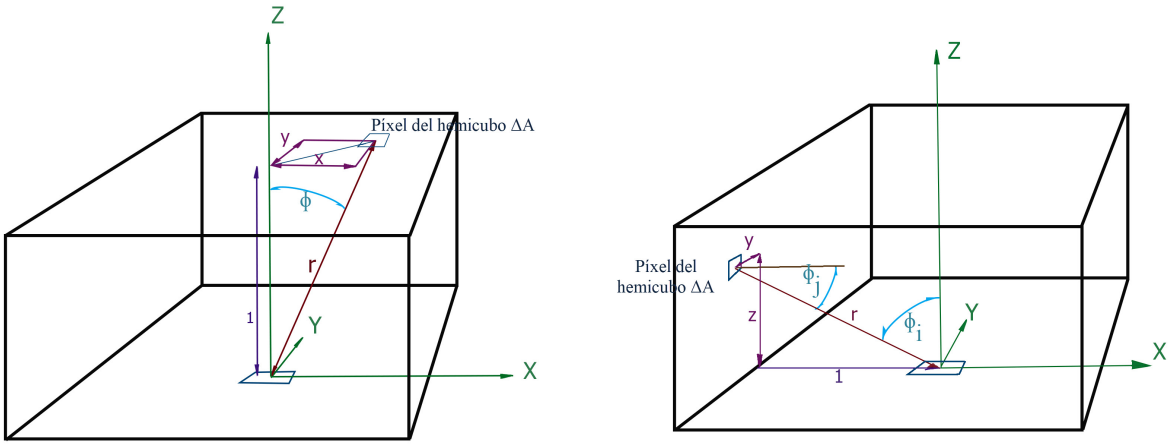
Figura 2.10: Simetría en las caras del hemicubo.

Luego de determinar qué parches son proyectados en cada píxel del hemicubo (correspondiente al parche i). Dado un píxel del hemicubo se determina si algún parche j fue proyectado en el mismo, en ese caso se suma el valor precalculado o peso del factor de forma asociado al píxel del hemicubo en la fila i, columna j de la matriz de factores de forma. Esta suma es realizada para cada píxel en el hemicubo. Este proceso se repite para todos los parches de la escena, obteniendo así la matriz de factores de forma completa.

$$F_{ij} = \sum_{q=1}^{R} \Delta F_q \tag{2.9}$$

ΔF_q = Delta factor de forma asociado con el píxel q del hemicubo.
R = Número de píxeles cuadrados del hemicubo que son cubiertos.
por la proyección de un parche sobre el hemicubo

[1]Un hash es una tabla de búsqueda a la cual se puede acceder a los elementos en O(1) promedio, asignándole a cada elemento guardado una clave de búsqueda.



$$r = \sqrt{x^2 + y^2 + 1}$$
$$\cos\phi_i = \cos\phi_j$$
$$\cos\phi = \frac{1}{\sqrt{x^2+y^2+1}}$$
$$\Delta F = \frac{\cos\phi_i \cos\phi_j}{\pi r^2}\Delta A$$
$$\Delta F = \frac{1}{\pi(x^2+y^2+1)^2}$$

$$r = \sqrt{y^2 + z^2 + 1}$$
$$\cos\phi_i = \frac{Z}{\sqrt{y^2+z^2+1}}$$
$$\cos\phi_j = \frac{1}{\sqrt{y^2+z^2+1}}$$
$$\Delta F = \frac{\cos\phi_i \cos\phi_j}{\pi r^2}\Delta A$$
$$\Delta F = \frac{Z}{\pi(y^2+z^2+1)^2}$$

Figura 2.11: Deducción de las ecuaciones del hemicubo.

2.2.5. Representacion matricial del sistema lineal

En las secciones anteriores se describió en detalle el método del hemicubo para el cálculo de factores de forma. Como resultado se obtiene el sistema de ecuaciones lineal que al resolverlo da como solución valores que determinan un aproximado de la función de radiosidad representado en la Ecuación 2.5 A continuación se analizan las propiedades de dicha matriz (**F**) teniendo en cuenta que el sistema del que forma parte debe resolverse utilizando algún método de resolución de sistemas lineales como puede ser Gauss-Seidel o Jacobi.

- Tamaño: En general la matriz será cuadrada $n \times n$, donde n es el número de parches de la escena.
- Dispersión: El grado de dispersión de la matriz depende de la oclusión de la escena, en el caso particular de las ciudades con alta oclusión, la matriz será más dispersa que completa.
- Simetría: Una matriz **A** es simétrica si sus elementos cumplen que $a_{ij} = a_{ji}$ para todo i, j . La matriz **F** no es simétrica en su forma actual. Sin embargo, una

simple transformación puede producir un sistema equivalente en que la matriz es simétrica. Si cada fila i es multiplicada por el área del elemento i, entonces la matriz de factores de forma se vuelve simétrica, debido a la reciprocidad de los factores de forma $\mathbf{F}_{ij}A_i = \mathbf{F}_{ji}A_j$.

- Diagonal dominante: Una matriz es diagonal dominante si para cada fila, el valor absoluto de la suma de los términos fuera de la diagonal es menor que el término de la diagonal. Esto es estrictamente cierto para el problema de radiosidad. Esto es una condición deseada para asegurarse que los métodos iterativos para resolver el sistema de ecuaciones 2.5 converjan.

- Condición: El número de condición de una matriz describe que tan sensible es a pequeñas perturbaciones en la entrada. En general la matriz **I-PF** está bien condicionada, indicando que la mayoría de los métodos de resolución son aplicables.

En [9] se muestra un análisis de cómo las propiedades geométricas de la escena inciden en las propiedades numéricas de la matriz.

2.3. Procesamiento Gráfico

En esta sección se tratan diferentes temas relacionados con el hardware y software involucrados en el procesamiento del *pipeline gráfico*.

2.3.1. Z-Buffering

El Z-Buffering [19] es el manejo de la profundidad de los objetos en una escena en 3 dimensiones con respecto a la ubicación de la cámara, usualmente se resuelve por hardware, aunque a veces por software. Es una de las soluciones al problema de visibilidad, en el cual se debe decidir qué objetos de una escena son visibles y cuáles están ocultos.

Usualmente se representa como un arreglo bidimensional, con cada entrada relacionada a un píxel de la pantalla. Cuando un elemento debe ser renderizado en un píxel, se compara el valor anterior de profundidad con el del objeto, en caso de ser más cercano al observador se sustituye el valor. Este filtrado es denominado *z-culling*.

La idea básica es verificar la profundidad z de cada superficie para determinar la superficie más cercana a la cámara. En el Algoritmo 1 se brinda un pseudocódigo.

La complejidad del algoritmo del Z-Buffer es O(Px), donde P es el número de polígonos y x es el número promedio de píxeles de cada polígono [20]. Como el producto Px no es sensible al número de polígonos (ya que si la cantidad de polígonos aumenta, el número promedio de píxeles por polígono es usualmente reducido), el rendimiento del algoritmo no es sensible al número de polígonos de la escena. Además, se aprovecha el paralelismo de las operaciones en GPU para mejorar el rendimiento del algoritmo.

A su vez, se definen dos planos para determinar los objetos que se van a renderizar. El plano *near* y el *far*. Si un elemento no se encuentra entre esos dos planos no es tomado

Algoritmo 1 Z-Buffer

```
Inicializar zbuffer(x,y) con una entrada para cada píxel de la pantalla con el valor 1
for cada poligono P do // En paralelo
    for cada pixel(x,y) de P do // En paralelo
        Computar profundidad_z en x,y
        if profundidad_z < zbuffer(x,y) then
            set_pixel(x,y,color)
            z_buffer(x,y) = profundidad_z
        end if
    end for
end for
```

en cuenta para ser dibujado. El frustum es el volumen tridimensional que determina cómo se proyectan los modelos desde un espacio de cámara en un espacio de proyección. Para una proyección en perspectiva (ver Figura 2.12), el frustum se puede ver como una pirámide recortada por los planos de corte cercano y lejano.

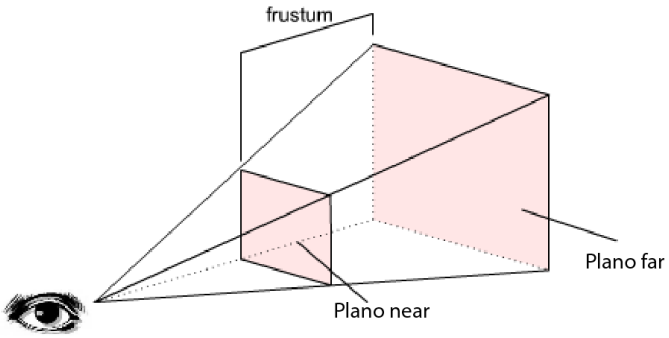


Figura 2.12: Ejemplo de frustum para proyección en perspectiva. Extraído de [21].

2.3.2. Pipeline gráfico

Un *pipeline* es un proceso que consiste en varias etapas en la que el resultado de la ejecución de algunas es la entrada de otras. Ademas varias etapas pueden ser ejecutadas de forma concurrente. El pipeline de renderizado en tiempo real se puede dividir en tres etapas conceptuales: *Application*, *Geometry* y *Rasterizer*, representadas en la Figura 2.13. Esta estructura es la base de lo que se utiliza para las aplicaciones de computación gráfica en tiempo real [10]. La etapa más lenta del pipeline es la que determina la velocidad de renderizado, el tiempo de refresco de las imágenes, que usualmente se representa como cuadros por segundo (fps) o *Hertz* (Hz).

La etapa de *Application*, es manejada por la aplicación misma, y por lo tanto está implementada en software corriendo en CPUs. Algunas de las tareas que son tradicionalmente implementadas aquí incluyen: detección de colisiones, algoritmos de aceleración de la aplicación, animaciones, simulaciones físicas y muchas más. La etapa *geometry*,

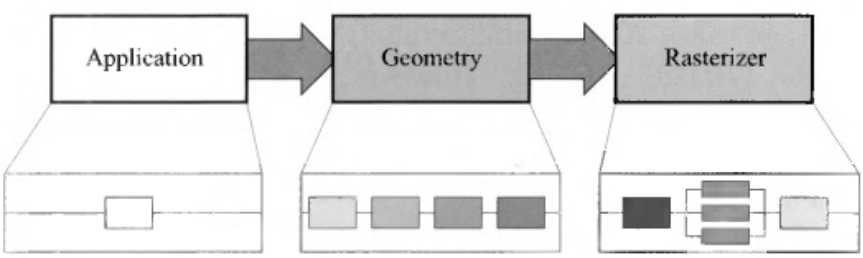


Figura 2.13: La estructura del pipeline gráfico en tres etapas: *Application*,*Geometry* y *Rasterizer*. Las últimas dos se dividen en subetapas funcionales. Extraído de [10].

se encarga de las transformaciones y proyecciones. Esta etapa computa qué se debe dibujar, cómo se debe dibujar y dónde se debe dibujar. Típicamente se ejecuta sobre la GPU (*Graphics Processing Unit*) que contiene una gran cantidad de núcleos programables así como un conjunto de operaciones aritméticas en hardware. En la Figura 2.14 se muestran las principales etapas funcionales de la etapa geometry.

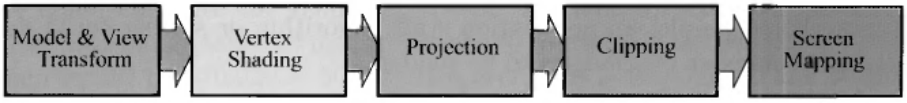


Figura 2.14: Las subetapas pertenecientes a la etapa del pipeline *geometry*. Extraído de [10].

Finalmente la etapa *Rasterizer* o de rasterización, dibuja una imagen con los datos que generó la etapa previa (*Geometry*), y también computa cualquier cálculo a nivel de píxel que sea necesario. Esta etapa es procesada completamente en GPU y al igual que *Geometry*, se subdivide en más etapas funcionales. Estas etapas se muestran en la Figura 2.15, y en particular la etapa funcional de *Pixel Shading* es una de las que le dará más importancia en este proyecto gracias a la versatilidad que ofrece para manipular datos en GPU.

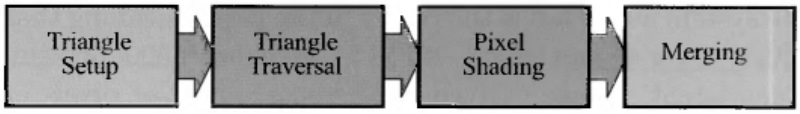


Figura 2.15: Las subetapas pertenecientes a la etapa del pipeline *rasterizer*. Extraído de [10].

2.3.3. Pipeline de la GPU

La GPU implementa las etapas del pipeline conceptual de geometría y rasterización introducidas anteriormente. Estas están divididas en varias etapas de hardware con diferentes grados de configurabilidad y programabilidad. En la Figura 2.13 se muestran

varias etapas pintadas con color de acuerdo a qué tan programables o configurables son, siendo las más oscuras las más programables, las más claras configurables pero no completamente programables y por último las grises siendo operaciones preprogramadas.

El *Vertex Shader* es una etapa totalmente programable que típicamente es utilizada para implementar las etapas funcionales de *Model and View Transform*, *Vertex Shading* y *Projection*. La etapa del *Geometry Shader* es opcional y opera en los vértices de las primitivas (puntos, líneas o triángulos). Puede ser usada para realizar operaciones de sombreado por primitiva, destruir primitivas o crear nuevas. Las etapas de *Clipping*, *Screen Mapping*, *Triangle Setup*, y *Triangle Traversal* son etapas preprogramadas que implementan las etapas funcionales con los mismos nombres. En la etapa *Rasterizer* se determina el correcto valor de cada píxel resultante. Los píxeles finales pueden ser mostrados en la pantalla, o como textura. Al igual que el *Vertex Shader*, el *Pixel Shader* correspondiente a la etapa de *Rasterizer* es totalmente programable y realiza la función de *Pixel Shading*. Finalmente, la etapa llamada *Merger* está entre la total programabilidad de los shaders y las operaciones preprogramadas de las otras etapas. Aunque no es programable, es altamente configurable y puede realizar una gran variedad de operaciones. Por supuesto, implementa la etapa funcional de *Merging*, a cargo de modificar el color, Z-Buffer, *blend*, *stencil* y otros buffers relacionados.

A través del tiempo, el pipeline de GPU ha evolucionado lejos de operaciones *hard-coded* hacia incrementar la flexibilidad y el control. La introducción de etapas de shaders programables fue un paso muy importante en su evolución.

Las GPUs actuales, en la que la mayoría de implementaciones de OpenGL están basadas, son capaces de ejecutar varios *teraflops* de operaciones, tienen varios *gigabytes* de memoria de acceso (con una velocidad de acceso superior a las CPUs tradicionales) y pueden manejar múltiples dispositivos de vídeo de alta resolución con enormes tasas de refresco.

2.3.4. OpenGL

OpenGL (Biblioteca de gráficos abierta) es una especificación abstracta de APIs para el renderizado gráfico. Diferentes implementaciones de OpenGL son provistas para diferentes GPUs.

La ventaja del diseño abstracto es la capacidad de ejecutar programas que utilizan esta API sin necesidad de lidiar con características propias de la plataforma escogida. De esta forma los comandos de un programa son enviados a una implementación de OpenGL y esta es la que se encarga de la comunicación con el hardware gráfico.

2.3.5. OpenGL Shading Language

El pipeline gráfico es la secuencia de pasos que OpenGL ejecuta para renderizar un modelo. Como ya se mencionó en 2.3.2 algunos de estos pasos son programables. En los pasos se recibe una escena tridimensional como entrada y se genera una imagen bidimensional como salida.

OpenGL Shading Language (GLSL o GLslang), es un lenguaje de programación de alto nivel para la programación de shaders con una sintaxis del estilo de C. Fue creado por el OpenGL ARB (*OpenGL Architecture Review Board*) para dar a los programadores mayor abstracción sobre el hardware gráfico y así no tener que utilizar *ARB assembly language* o lenguajes específicos del hardware.

2.3.6. Occlusion Queries

Las *occlusion queries* son una funcionalidad que brinda el hardware de NVIDIA para comprobar si una geometría será visible o no desde el punto de vista del observador.

Esto permite determinar si un objeto con una gran cantidad de vértices es visible o no, pudiendo acelerar el tiempo de renderizado. La aceleración se debe a que una llamada para determinar visibilidad no toma en cuenta la iluminación ni las texturas, que consumen la mayor parte del tiempo en el proceso de renderizado. Sin embargo presentan dos grandes problemas:

1. El *overhead* generado: las occlusion queries pueden agregar una llamada de dibujado adicional. Si un objeto pasa el test de visibilidad debe hacerse otra llamada para ser dibujado (es un proceso secuencial).

2. La latencia causada por esperar los resultados de las consultas.

En general, las occlusion queries se utilizan cuando se tiene una escena con muchos objetos complejos. Si los objetos son simples se pueden producir pérdidas en el rendimiento debido a los problemas mencionados anteriormente.

2.3.7. Frustum Culling

Esta técnica llamada *Frustum Culling* acelera los tiempos de rendering, eliminando aquellos elementos que no entran en el volumen de vista de la cámara (Figura 2.16). De esta forma se evita su dibujado y el consumo inútil de recursos de cómputo. Es explotable en todas las escenas, en algunas más que en otras y se ve beneficiado por el uso de estructuras jerárquicas como el octree.

2.4. Matrices dispersas

En todo algoritmo que haga uso de grandes matrices con una gran cantidad de valores iguales a cero, es de una gran utilidad trabajar con representaciones dispersas. En contrapartida de las matrices completas que guardan cada una de las entradas de la matriz en memoria, las matrices dispersas solamente guardan las entradas que no son cero. Dentro de los formatos disponibles, se eligió el formato *Dictionary of Keys* (DOK) por su fácil implementación y compatibilidad con Matlab [22] facilitando la integración de las matrices generadas por el algoritmo junto con Matlab. El formato se caracteriza por indicar en cada fila del archivo correspondiente a la matriz la terna

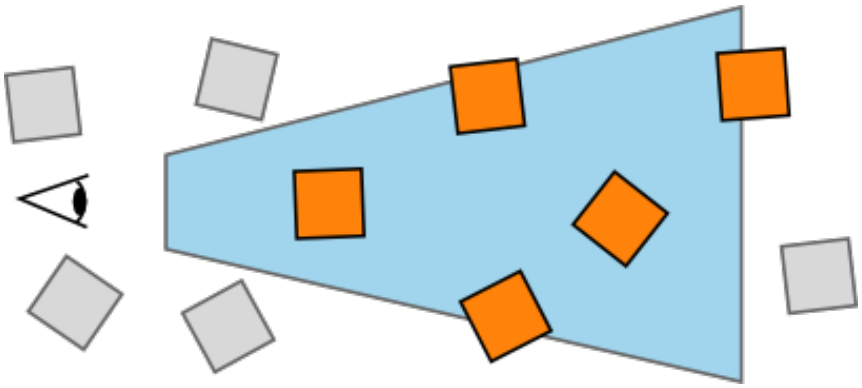


Figura 2.16: Ejemplo de filtrado de objetos fuera del cono de visión (Frustum Culling).

(Fila, Columna, Valor), con lo que se cargan todos los valores y el resto de la misma es poblado con ceros. Uno de los parámetros fundamentales para realizar estimaciones sobre el espacio de almacenamiento requerido y la complejidad de operaciones es el número de entradas no nulas, esto es, el total de números distintos de 0 de la matriz **S**. El almacenamiento computacional es proporcional al número de entradas no nulas, así como también la complejidad de operaciones simples sobre arreglos. La complejidad de las operaciones involucra también a la cantidad de filas (m) y cantidad de columnas (n), o el producto de ambos factores, pudiendo existir operaciones más complejas que involucren ordenamientos u otras técnicas. Con este tipo de representación es práctico manejar matrices conteniendo decenas de miles de elementos no nulos. De manera de ejemplo, se crearon dos matrices de prueba en Matlab, una completa y otra dispersa con una densidad del 5 %, luego se midió el espacio en disco y el tiempo que lleva calcular la multiplicación por sí misma. En la Tabla 2.1 se puede observar como ambos casos fueron favorables para la matriz de forma dispersa. Estas ganancias solo mejoran cuando se trabaja con matrices más grandes, sin embargo, se debe tener mucha cautela ya que el uso de matrices dispersas puede ser perjudicial en algunas circunstancias, sobre todo si la matrices con las que se va a trabajar son muy densas.

	Dispersa	Completa	Dispersa Grande	Completa Grande
Tamaño	1.100x1.100	1.100x1.100	11.000x11.000	11.000x11.000
Memoria (MB)	0,97	9,68	96,9	968
Tiempo $\mathbf{M} \times \mathbf{M}$ (s)	0,0471 s	0,171s	7,789s	87,4s

Tabla 2.1: Eficiencia de matrices dispersas.

2.5. Visibilidad jerárquica del Z-Buffer

En esta sección se analizará el algoritmo propuesto por Greene *et al.* [8], otra de las bases de este proyecto de grado. En él, se presentan las bases para la construcción de

un algoritmo de visibilidad eficiente, aprovechando la capacidad de la tarjeta gráfica y reduciendo los tiempos de renderizado. Los autores describen tres técnicas principales, llamadas de coherencia inherente, que buscan obtener mejoras frente al algoritmo del Z-Buffer convencional.

La primera es la coherencia de espacio-objeto: en muchos casos un simple cálculo puede determinar la visibilidad de una colección de objetos cercanos dentro del espacio. La segunda es la coherencia imagen-espacio, de una manera similar resuelve la visibilidad de un objeto obstruyendo una colección de píxeles. La tercera es la coherencia temporal, en la que se especifica un método para reutilizar la información del cuadro dibujado anteriormente para estimar qué se dibujará en el cuadro actual.

Para explotar la coherencia espacial se utiliza una subdivisión espacial en octrees, usados comúnmente para acelerar algoritmos de ray-tracing. Para explotar la coherencia imagen-espacio, se mejora el algoritmo tradicional de escaneo y conversión del Z-Buffer con una *Z-Pirámide* que permite rechazar rápidamente la geometría oculta. Finalmente para explotar la coherencia temporal, los autores seleccionan la geometría que fue visible en el cuadro anterior. Todo esto resulta en un algoritmo más eficiente que algoritmos tradicionales de Z-Buffering para algunos modelos.

2.5.1. Octree

La subdivisión espacial del tipo octree es una técnica de aceleración usada frecuentemente en algoritmos complejos de renderizado 3D como ray-tracing. Consiste en dividir la escena en Cubos de igual tamaño. Al principio se tiene un cubo grande conteniendo toda la escena, que luego se puede subdividir en 8 cubos de igual tamaño más pequeños y así sucesivamente (Figura 2.17), de aquí el nombre octree. Por lo general se establece una cantidad máxima de niveles. Se debe de tomar la decisión de si los objetos que pertenecen a más de un cubo del mismo nivel se asocian a sus nodos padres o a ambos nodos hermanos. Tampoco se deben generar todos los hijos posibles de todos los niveles, sino que se divide un cubo siempre que el cubo contenga una cantidad de primitivas mayor a cierta cantidad. En el momento que se tiene menos primitivas que la cantidad mínima establecida se corta la recursión y se asocian todas las primitivas a esa hoja. También se aprovechan los octrees para realizar el Frustum Culling y el algoritmo del Z-Buffer de forma inteligente, en caso de que no se vea un cubo, todas las geometrías dentro de él tampoco se verán, con lo que se logra ahorrar poder de procesamiento y por lo tanto mejora el rendimiento del algoritmo.

2.5.2. Z-Pirámide

El octree de objetos permite recortar grandes porciones del modelo al costo de la conversión de escaneo (*scan converting*) de los octrees. Este proceso puede resultar un poco costoso. Para reducir el costo de determinar la visibilidad de un cubo, se implementa una Z-Pirámide que en muchos casos puede concluir que un polígono grande está oculto evitando examinar el polígono píxel por píxel.

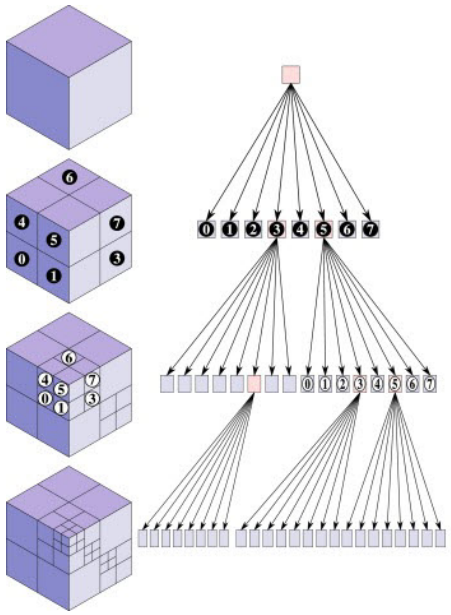


Figura 2.17: Estructura octree. Extraído de [23].

La idea básica de esta pirámide es utilizar el Z buffer original como el nivel base de la pirámide y luego combinar cuatro Z valores en cada nivel en un Z valor para el nivel más fino, eligiendo el Z más lejano al observador de los 4. Cada entrada de la Z-Pirámide representa el z más lejano para un área cuadrada del Z buffer. En el nivel más fino de la pirámide hay un sólo Z valor que es el más lejano para el observador en toda la imagen. En la Figura 2.18 se muestran los niveles de pirámide construida.

Mantener la Z-Pirámide es sencillo. Cada vez que se modifica el Z buffer, se propaga el nuevo Z valor a los niveles más gruesos de la pirámide. El proceso se detiene cuando se llega al nivel de entrada de la pirámide que contiene un valor igual o más lejano que el nuevo Z.

Para testear la visibilidad de un polígono se toma la muestra más fina de la imagen cuya imagen correspondiente tapa el espacio de pantalla donde se encuentra el octree del objeto. Si el valor Z más cercano del polígono se encuentra más lejos que la muestra en la Z-Pirámide, se sabe inmediatamente que el polígono está oculto. Esto se utiliza para conocer la visibilidad de cada cara de los octrees y los polígonos de los modelos.

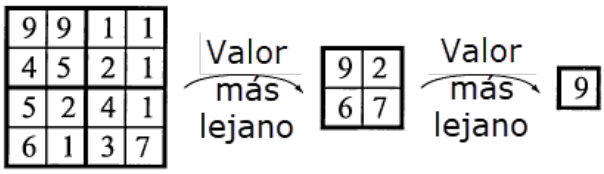


Figura 2.18: Niveles de la Z-Pirámide. Extraído de [10].

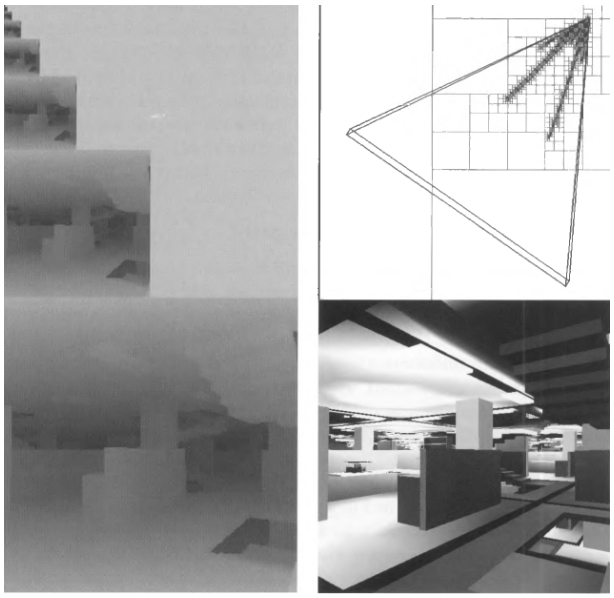


Figura 2.19: Escena compleja (abajo-derecha) con su correspondiente Z-Pirámide (izquierda) y octree (derecha-arriba). Extraído de [10].

En la Figura 2.19 se muestra un ejemplo en donde se puede observar una escena compleja (Parte inferior derecha) a la que se le aplica el algoritmo de la Z-Pirámide (Parte izquierda). También se observa la subdivisión de la misma en un octree vista desde arriba (Parte superior derecha).

2.5.3. Lista de coherencia temporal

Frecuentemente cuando se renderiza una imagen de un modelo complejo utilizando octrees, solo una pequeña porción de los cubos es visible. Por ejemplo, si se renderiza el próximo cuadro de una animación, la mayoría de esos cubos probablemente sean visibles nuevamente. Para aprovechar esto, se mantiene una lista de cubos visibles de un cuadro al siguiente, y se renderizan antes de comenzar el algoritmo, esto hace que el uso de la Z-Pirámide sea mucho más efectivo además de aprovechar más el poder de la GPU.

En este proyecto, se toma la idea de la lista de coherencia temporal pero se aplica de otra manera. En su lugar se utiliza una lista de coherencia espacial utilizando la Z-Curva (2.6) para el ordenamiento de parches. Si dos parches son cercanos y tienen normales similares, existe una alta probabilidad que los objetos que ve uno también los ve el otro, por lo que se asemeja al caso de coherencia temporal que describe una relación entre el mismo parche en cuadros distintos de la ejecución. Como en principio interesa solamente renderizar un cuadro para cada hemicubo, la alternativa de la coherencia espacial resulta adecuada.

2.6. Z-Curva

El algoritmo de la Z-Curva, también conocido como Z-Order u Ordenamiento de Morton, tiene como propósito mapear datos multidimensionales a una sola dimensión, manteniendo la localidad de los puntos.

El algoritmo parte de un espacio de n dimensiones acotado. Este espacio acotado se subdivide en potencias de dos secciones equiespaciadas. Estas divisiones en cada coordenada se numeran y representan en binario. Luego se genera un código binario que se obtiene intercalando cada componente del punto (dado como un vector). El código es interpretado como una representación en binario del valor de la Z-Curva para este fragmento del espacio n-dimensional. Cuando se quiere ordenar una serie de puntos que pertenecen a este espacio se les asigna un valor de Z-Curva determinado por la porción del espacio en el que se encuentran. Los z-valores son escalares que se asignan a cada porción del espacio, si se ordenan estos escalares de menor a mayor se obtiene una curva fractal que llena el espacio. Esto es particularmente atractivo por ejemplo cuando se quiere ordenar objetos del espacio de acuerdo a alguna propiedad vectorial de las superficies, por ejemplo las normales o las posiciones.

Este método fue propuesto por Guy Mcdonald's Morton [24]. Son funciones que mapean un vector de datos en un escalar. El interés en estas funciones se debe a la preservación de localidad de la entrada (un ordenamiento en el cual el criterio es la distancia entre los datos de entrada).

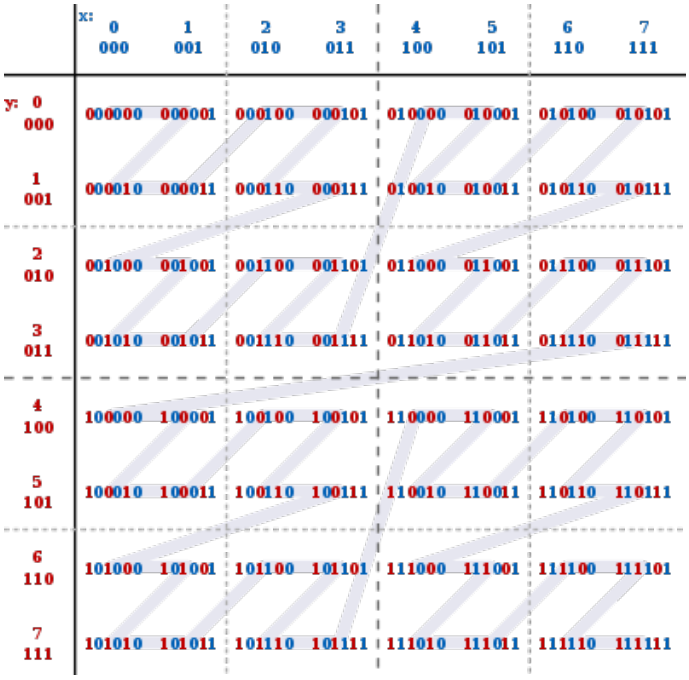


Figura 2.20: Ejemplo Z-Order.

La Figura 2.20 muestra los z-valores para el caso de dos dimensiones con coordenadas enteras con $0 <= x <= 7$ y $0 <= y <= 7$. Entrelazando los códigos binarios de los

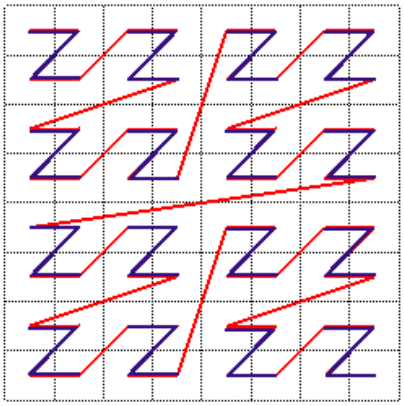


Figura 2.21: Ejemplo Z-Curva fractal extraída de [25].

valores de las coordenadas se obtiene el valor Z. Si se conectan los z-valores en su orden numérico, se ve que se produce una curva con forma Z fractal (Figura 2.21), que es la que le da el nombre a este algoritmo.

Utilizando la Z-Curva, se puede ordenar las superficies de acuerdo a su posición y su normal, maximizando la coherencia espacial y temporal del orden de procesamiento.

Capítulo 3

Solución propuesta

En este capítulo se detallan las propuestas realizadas para desarrollar el algoritmo. A su vez, se desarrollan las decisiones y los detalles tanto de diseño como de implementación de la solución. Se explicará cómo se relacionan los componentes del sistema implementado y que función cumple cada uno de ellos, teniendo en cuenta las bases del algoritmo utilizado explicadas en capítulos anteriores.

3.1. Objetivos y Alcance del proyecto

Para optimizar el cálculo de la matriz de factores de forma, se busca aplicar el concepto de visibilidad jerárquica al método del hemicubo en escenas altamente ocluidas.

En este trabajo sólo se toma en cuenta la información geométrica de los polígonos. La información correspondiente a colores y texturas de los polígonos no es procesada y por lo tanto no influye en el resultado final. Dentro de los objetivos propuestos en el capítulo 1, interesa construir una propuesta de implementación a modo de tener como referencia los hitos más relevantes dentro del proyecto. A continuación se detalla cada uno de ellos.

- **Implementar la técnica del hemicubo para el cálculo de factores de forma**. Esta implementación puede realizarse en CPU o GPU. En la sección 3.2.1 se entrará en detalle de la elección y su justificación.
- **Implementar la estructura de octree.** Esta técnica de volúmenes acotantes será de mucha utilidad para administrar el renderizado de la escena de una forma más eficiente.
- **Frustum Culling.** Se aplicará esta técnica en el cálculo de la visibilidad de cada cara de cada hemicubo. Al filtrar una gran cantidad de objetos de la escena se evitarán muchos cálculos innecesarios.
- **Occlusion Culling**. Para implementar esta técnica se aprovechan las Occlusion Queries que ofrece OpenGL. Así al realizar tests de profundidad (también llamados Z-Tests) sobre el octree, si se rechaza un nodo, también se rechazan todos los nodos hijos. Las tarjetas gráficas modernas en su mayoría implementan en hard-

ware técnicas eficientes para hacer el test de profundidad. Por esta razón no se implementa la Z-Pirámide recomendada por Greene *et al.* [8] (ver Sección 2.5.2).

- **Procesar de acuerdo a la coherencia espacial**. Se construirá el algoritmo de manera que se realice una recorrida por la escena de forma ordenada, pudiendo utilizar la información del parche anterior para estimar la visibilidad en el parche actual.

3.2. Diseño de la solución

El diseño propuesto se basa en una distribución de cómputo entre CPU y GPU en la que se intenta mantener ambas unidades ocupadas el mayor tiempo posible de forma concurrente. Se renderiza la geometría para cada parche de la escena en las cinco caras del hemicubo correspondiente. Esta información es posteriormente utilizada para obtener la fracción de energía de cada parche en la escena que alcanza al primero, es decir una fila completa correspondiente a la matriz de factores de forma.

Con el propósito de optimizar el renderizado, la escena es almacenada en un octree; este permite la organización jerárquica de la misma y algunas técnicas de filtrado que se describirán a continuación.

Se aplica una estrategia de *Frustum Culling*, que consiste en evitar el procesamiento de partes de la escena que no entran en el volumen de vista y por lo tanto no aportan información alguna al hemicubo.

Otra de las técnicas utilizadas es *Occlusion Culling*, la cual consiste en evitar la carga de procesamiento dada por geometrías que no aportan información en la escena y que no siempre pueden ser filtradas por el método de frustum culling. La técnica intenta reducir el procesamiento de geometrías eliminando objetos ocluidos (detrás de otros en la escena) y que por ende no aportan información a la sección calculada de la matriz de factores de forma. Esta técnica es la más importante en escenas de alta oclusión como es el caso de las ciudades. La solución hace uso de las *Occlusion Queries*, que son una implementación que provee OpenGL para usar la estrategia de *Occlusion Culling*.

El orden de procesamiento tiene un papel fundamental en el diseño. Tanto es así que forma parte del pre-procesamiento de la escena. La razón principal de su importancia es el aprovechamiento de la coherencia espacial. Dado un parche y una de las caras de su hemicubo, otro parche cercano (tanto en posición como en orientación) tendrá altas probabilidades de tener una interacción directa con la escena muy similar. Si bien hay excepciones, la mayoría de los casos son aprovechables. Esta información es usada para estimar parches visibles de un hemicubo teniendo en cuenta la visibilidad desde el hemicubo anterior.

En la Figura 3.1 se observa un estructura de árbol que puede ser generalizada como la estructura de un octree agregando ocho nodos hijos en lugar de los dos que tiene actualmente. En esta estructura se observa la información obtenida en el procesamiento de un cierto hemicubo con anterioridad (los nodos de trazo discontinuo). Si el nodo fue visible para el hemicubo se realiza una Occlusion Query pero no se espera a obtener el resultado (aunque se guarda para el próximo hemicubo). Este nodo se asume visible y

se procesan las geometrías contenidas. En el caso de que un nodo en el frame anterior estuviese oculto se realiza la Occlusion Query y se espera el resultado para determinar su visibilidad. En este caso los nodos hijos de un nodo invisible no son procesados y la información de visibilidad se propaga desde el nodo padre. Al no procesar la geometría contenida en los nodos hijos, se reduce el costo computacional de dibujar la escena.

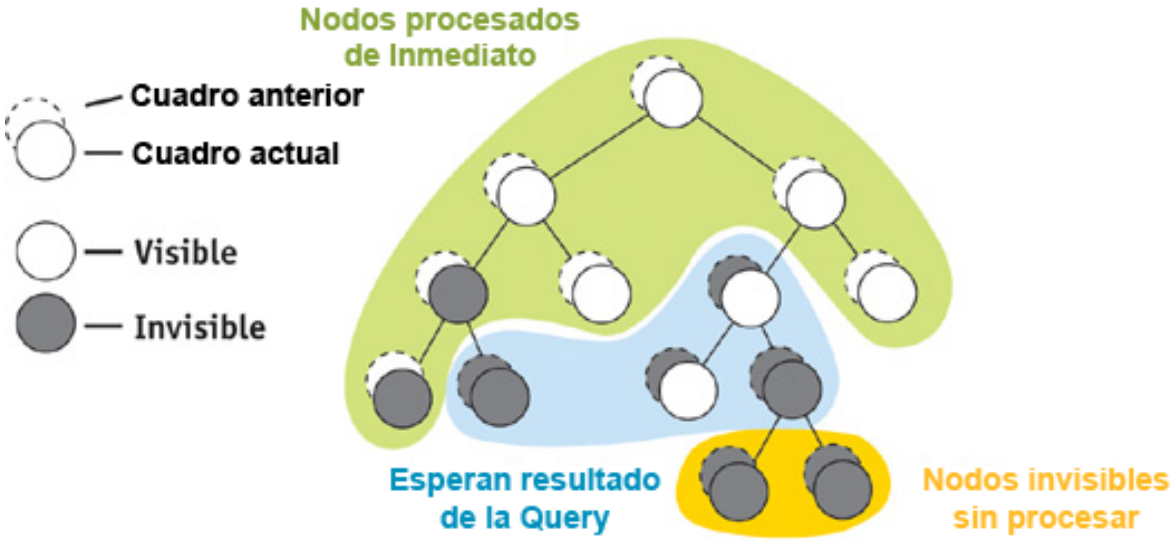


Figura 3.1: Occlusion Culling. Extraído de [26].

3.2.1. Decisiones de diseño

En el transcurso del proyecto se han tenido que tomar decisiones entre distintas alternativas en las que siempre se priorizó el rendimiento computacional del algoritmo.

- La primera decisión de diseño consiste en la elección del hardware en donde calcular los factores de forma. Una decisión más que clara es que se utiliza la GPU en vez de CPU por todos los beneficios que implica esto en cuanto a paralelismo y desempeño. Dentro de la GPU esto se puede lograr tanto utilizando CUDA[1] como *Shaders*. En este caso se opta por utilizar shaders ya que posibilita explotar una técnica para proyectar los vértices de la escena sobre una textura en la que luego es posible ejecutar el cálculo de píxel por píxel de una forma eficiente. Una vez que se obtiene la textura en la GPU correspondiente a la proyección sobre el hemicubo de un parche, esta es procesada directamente por otro conjunto de shaders también en GPU que retorna una fila completa de la matriz de factores de forma, evitando así la pérdida de recursos por tiempo de transferencia a CPU.
- La utilización de octrees para jerarquías de la escena es una elección tomada para seguir los líneamientos de [8], además de que a lo largo de los años ha probado ser una estructura de subdivisión espacial muy eficiente [27] [28].

[1]CUDA es una tecnología propietaria de las tarjetas gráficas NVIDIA que permite implementar programas que utilicen la arquitectura paralela de las tarjetas.

- Codificación de los parches en GPU. Es necesario idear un sistema de identificación de parches en GPU para conocer que columna de la matriz es la que debe ser actualizada. Para esto se utiliza una variación de una técnica llamada *Color Picking* [29] la cual se elige por su eficiencia y fácil implementación.
- No implementar la Z-Pirámide a nivel de aplicación ya que es redundante con el Z-Buffer ya presente en las tarjetas de hoy en día, lo que causaría una reducción en el desempeño. En su lugar se utilizan *oclusion queries* como una técnica de aceleración más eficiente.
- Se busca crear una solución parametrizable, cuyos parámetros variables sean modificables a través de un archivo de configuración con el objetivo de no recompilar la aplicación con cada cambio.
- Las matrices se almacenan de forma dispersa para soportar matrices de gran tamaño, siempre y cuando la escena presente una alta oclusión entre los parches. Como se ve en la Figura 1.1, las escenas de ciudades típicamente cumplen esta propiedad. Cuando dos parches no se ven entre ellos esto genera dos ceros en la matriz de factores de forma, uno por cada fila. Por lo tanto, en estos casos la matriz de factores de forma será poco densa, con lo que se ahorra una gran cantidad de memoria.
- Utilizando la Z-Curva, se ordenan los parches de la escena primero de acuerdo a sus normales y luego de acuerdo a su posición. El procesamiento se realiza en este orden, completando todas las caras del hemicubo asociado. Durante el procesamiento se mantiene una estructura por cada cara del hemicubo, que permite determinar por cada otro parche en la escena su visibilidad. Al momento de procesar el siguiente parche se aprovecha la información del hemicubo anterior haciendo uso de la coherencia espacial. Esto se asemeja a la coherencia temporal vista en [8]. De este modo cada cara del hemicubo actual utiliza la cara más cercana (en términos de vectores normales) del hemicubo anterior para estimar si una cara será visible.

3.3. Propiedades de la escena

Es importante conocer el tipo de escena que se está manejando para evaluar el desempeño del algoritmo. Una de las propiedades que influyen mayormente es la coherencia espacial. Una habitación normal, compuesta por 4 paredes y algunos objetos, es una escena con alta coherencia espacial, porque parches cercanos de la pared o el techo ven casi lo mismo, desde la misma perspectiva. Por el contrario, una cámara anecoica (Figura 3.2) es una escena con baja coherencia espacial, porque parches cercanos ven partes de la escena muy diferentes. Cuanto mayor sea la coherencia espacial, más efectividad tendrán los algoritmos predictivos que utilizan información de parches cercanos para determinar la visibilidad.

El segundo parámetro que se puede identificar es el factor de oclusión. En las escenas con un factor de oclusión alto (como las ciudades), se puede aprovechar esta propiedad para acelerar el procesamiento, descartando la mayor parte de los parches no visibles.

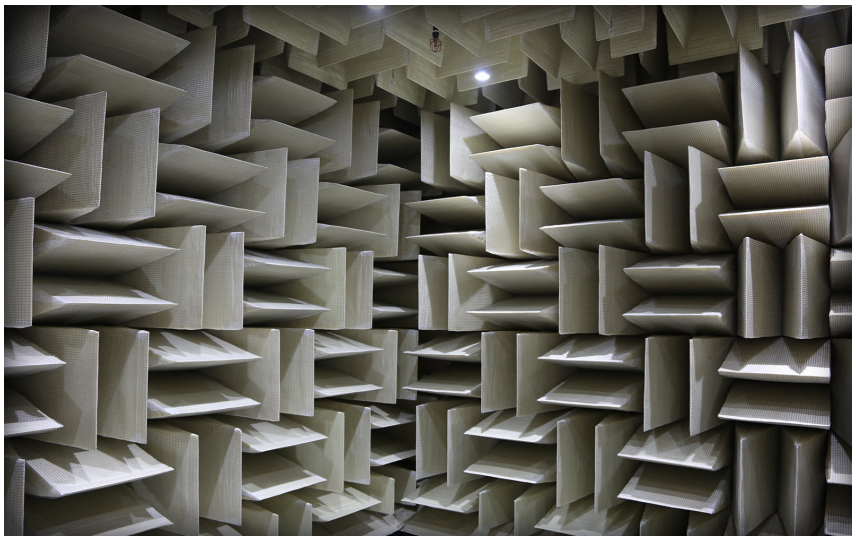


Figura 3.2: Ejemplo de baja coherencia espacial: Cámara anecoica.

Asumiendo igual cantidad de objetos, las escenas con un factor bajo de oclusión poseen un tiempo de renderizado mayor en proporción a las escenas con alta oclusión, debido a que las técnicas de filtrado no serán muy efectivas y se debe renderizar la mayor parte de la escena, incrementando los tiempos de ejecución. En la Figura 3.3 se observa el factor de oclusión de dos de las ciudades de prueba utilizadas en este proyecto. Para cada uno de los píxeles se muestra la cantidad de parches ocluidos. En cada píxel sólo se ve un parche y el resto se "dibuja" sin aparecer al final del proceso. Evitando el renderizado de los parches ocluidos se obtienen mejoras significativas en el rendimiento del algoritmo. La oclusión puede tener coherencia espacial, esto es: dos parches cercanos "no ven" prácticamente lo mismo. Por tanto, esa información disponible para un parche, puede ser aprovechada por los parches cercanos para acelerar los cálculos relativos a la escena.

3.4. Diagrama de flujo

La Figura 3.4 muestra el flujo de ejecución del programa y refleja el orden en que se realizan las distintas operaciones para construir la matriz de factores de forma resultante.

1. El programa recibe un archivo de entrada que presenta la descripción de la escena codificada a nivel de superficies y vértices.

2. Se indica si se desea subdividir los triángulos en triángulos más pequeños para aumentar el número de parches.

3. Se ordenan los parches por normales y luego por posición haciendo uso de la Z-Curva.

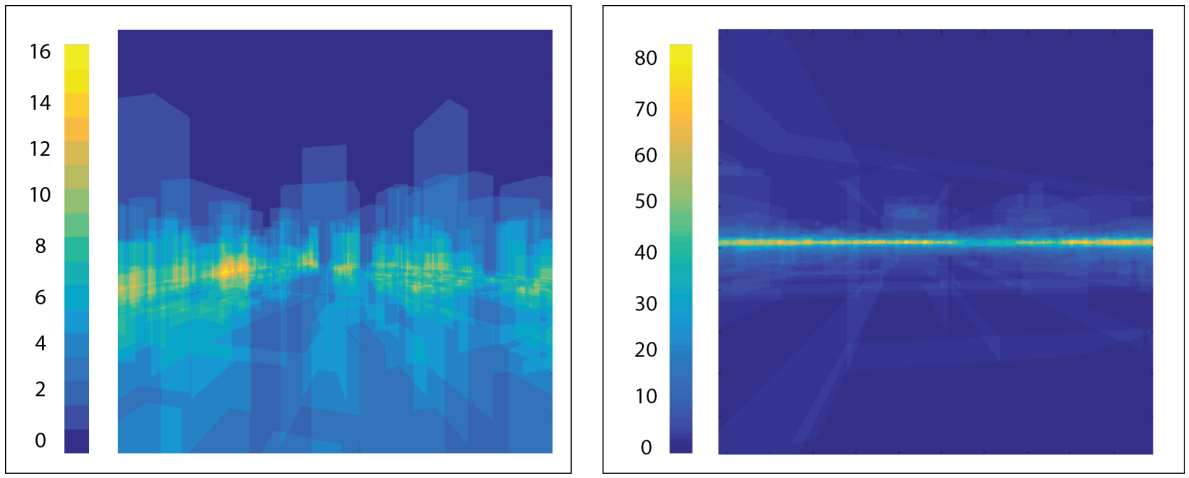


Figura 3.3: Oclusión en dos escenas urbanas. A la izquierda City (8.800 parches) y a la derecha Venecia (1.495.068 parches). Para cada píxel se representa el número de parches ocluidos.

4. Se asigna un color único a cada parche.

5. Se procede con la construcción del octree de la cantidad de niveles deseada.

6. Para cada parche se realiza el frustum culling de los demás cubos del octree para que no participen en el renderizado.

7. Para cada parche se realizan las Occlusion Queries para determinar de una forma más eficiente los nodos visibles del octree. Se procede con el renderizado a través de los shaders, obteniendo una textura como resultado.

8. Se procesa la textura asociada a cada parche y se crea otra que contiene los resultados de un fila de la matriz **F**. Como **F** tiene una fila y columna por parche, al procesar todas las texturas se calcula la matriz **F** completa.

9. Opcionalmente se renderizan los resultados parciales en la pantalla para obtener un idea del trabajo que se está realizando.

10. Se traduce la textura a la matriz **F** de salida que contiene los factores de forma de cada parche, pudiendo ser utilizada posteriormente en el algoritmo de Radiosidad.

3.5. Implementación

El algoritmo fue implementado en el lenguaje C++ para la ejecución en CPU junto con el GLSL para la implementación de los shaders de GPU. Las decisiones de implementación se pueden dividir en cinco grandes grupos que se definen a continuación.

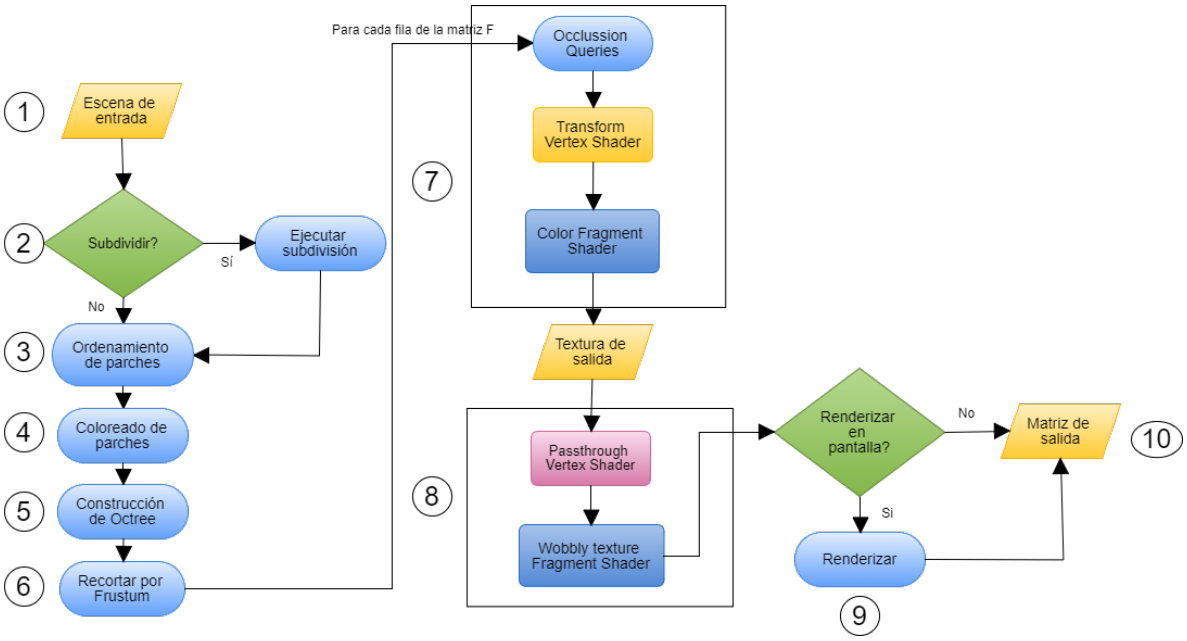


Figura 3.4: Diagrama de flujo del algoritmo.

3.5.1. Codificación de parches con colores

Para conseguir pre cargar la información de identificación de cada parche en memoria gráfica se utilizó el color buffer para asignar un color distinto en cada parche de la escena. Este color puede ser decodificado en el entero correspondiente al índice de la fila de la matriz de factores de forma que debe ser afectada. Con este enfoque se limita el número de llamadas a GPU por lo que se reduce la tasa de transferencia CPU-GPU[30] y por tanto se mejora en eficiencia en el uso de bus del sistema y más tarde en tiempos totales de ejecución.

Para realizar esto se le asigna a cada parche de la escena un identificador único de tipo entero, y luego se convierte en un código de color RGB. Este color no tiene un significado extra. La codificación de los parches en la escena implica utilizar los bits menos significativos del entero original en el canal rojo(R) y los más significativos en el canal azul (B).

Esto puede entenderse mejor con el siguiente fragmento de código donde i representa el índice que identifica el parche en la escena:

Dada la representación de un entero sin signo como un código de 4 bytes,

$$\begin{aligned} \text{byte r} &= (\text{i} \ \& \ \text{0x000000FF}) >> 0 \\ \text{byte g} &= (\text{i} \ \& \ \text{0x0000FF00}) >> 8 \\ \text{byte b} &= (\text{i} \ \& \ \text{0x00FF0000}) >> 16 \end{aligned}$$

Básicamente, se codifican los primeros 8 bits como un entero entre 0 - 255 que

representan el canal R (rojo) los próximos 8 bits representan el canal G (verde) y los últimos 8 bits representan B (azul), en la Figura 3.5 se muestran los parches de una escena codificados usando este procedimiento.

Con esta representación es muy sencillo decodificar el índice en la tarjeta gráfica con el siguiente procedimiento:

$$\text{int i} = \text{r} + 256*\text{g} + 256*256*\text{b}$$

Este índice representa la fila del parche p en la matriz de factores de forma.

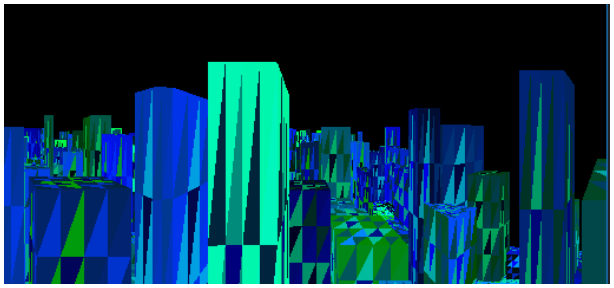


Figura 3.5: Codificación de parches con colores.

3.5.2. Algoritmo optimizado

A continuación se describen los detalles de implementación de cada etapa del algoritmo (Figura 3.4).

1. Carga de la escena en memoria: en este paso se carga la escena iterando sobre el archivo y de esta forma se generará una estructura de escena en memoria que consiste de parches, vértices y normales.

2. La subdivisión de la escena es configurable. Este parámetro indica los niveles de subdivisión. Para el primer nivel se obtienen todos los parches de la escena (se asumen triángulos) y se dividen en cuatro nuevos triángulos correspondientes al segundo nivel. Estas divisiones se pueden continuar, multiplicando por 4 la cantidad la cantidad total de parches cada vez que se agrega un nivel de división.

3. La asignación de colores a los parches se realiza con el índice de la posición luego del ordenamiento de los mismos en la escena. Para esto se toma el índice y se codifica como fue descrito en la Sección 3.5.1.

4. Ordenamiento de los parches en la escena utilizando la Z-Curva [31] por normales y posición. En esta etapa se utilizó una implementación basada en la codificación y decodificación de códigos de Morton para mapear los parches de la escena a un código. Este es interpretado como su representación entera y ordenado. De esta forma se obtiene un criterio de cercanía utilizado a posteriori para decidir el orden en que se procesan las filas de la matriz de salida.

5. La carga en el octree se realiza de forma de llevar los parches a el nivel más bajo posible, sin embargo es difícil dejar escapar el hecho por el cual algunos de los parches no encajan perfectamente en los niveles inferiores. Por esta razón el criterio elegido implica asignar este parche al nodo padre.

   En esta etapa la información referente a la escena ya se encuentra totalmente en memoria principal con la estructura del octree completa. Posteriormente a esto, el algoritmo procede con la carga de la información de todos los parches de la escena junto con los cubos correspondientes a los nodos del octree en GPU.

   Para el traspaso de esta información a GPU se hace uso de la abstracción de buffers brindada por OpenGL. Por cada nodo se reserva memoria para dos buffers, el primero contiene los parches correspondientes al octree, y el segundo contiene los propios vértices del octree. Esto sucede porque en las occlusion queries se hace uso de dicha información [30].

6. Se implementó dentro de la clase octree información con los seis planos definidos por la pirámide visual [32] junto con los ocho vértices de las esquinas de cada nodo y la definición de la cámara. Los planos definidos por la pirámide visual permiten realizar un filtrado de geometrías en la escena. Para esto, se recorre de forma recursiva la estructura del octree mientras se califican los posibles escenarios:

   - Completamente contenida. En este caso la recursión es terminada y se determina que no se puede filtrar ningún nodo de esta rama.
   - Completamente fuera. En esta rama ningún nodo es visible y por lo tanto no hay geometrías para renderizar.
   - Parcialmente contenida. La recursión debe continuar pero la geometría en el nodo parcialmente contenido no es filtrada.

7. Se utilizó la implementación de las Occlusion queries de Nvidia ARB occlusion query [30]. En un principio se estudia el comportamiento de esta funcionalidad que permite determinar la visibilidad de forma más eficiente que el pasaje completo por el pipeline gráfico de ciertos vértices a procesar. El uso directo de occlusion queries provoca que, mientras la CPU espera el resultado del test en GPU, la primera se encuentre desocupada. Esto significa que mientras la GPU trabaja en determinar la visibilidad, la CPU no puede seguir avanzando, lo que repercutirá en el tiempo total del algoritmo. Cuando el resultado llega a CPU la GPU puede haberse quedado sin geometrías a procesar lo que ocasiona que también exista tiempo ocioso en la GPU. Con el objetivo de aprovechar la coherencia espacial, se utiliza la información del hemicubo inmediatamente anterior al que se está procesando. Se mantienen dos colas, una de ellas para los parches que fueron visibles por cada cara del hemicubo anterior y otra para los no visibles. Si fueron visibles en el anterior, entonces se renderizan todos sin esperar el resultado de respuesta de la *occlusion query* (igualmente se realiza la misma para actualizar su visibilidad). En el caso de que no fueran visibles en el hemicubo anterior, no

puede determinarse directamente su visibilidad y hay que esperar el resultado de la *occlusion query*. Este método asincrónico permite que no haya cuellos de botella esperando por el resultado de las *occlusion queries*.

Una decisión de implementación implicó no hacer occlusion queries sobre un volumen acotante que contenga menos de un número de parches (configurable) ya que en algunos casos la oclusión query del cubo puede costar más trabajo que el renderizado de su geometría contenida. Ya que el cubo que acota los parches se representa mediante 2 triángulos por cara, es muy directo determinar que el número de parches en el interior de este cubo debe ser por lo menos mayor a 2.

Para el renderizado se utilizan los *Pixel Buffer Object* de OpenGL. Estos evitan la sincronización explícita antes del acceso a la información de un buffer. Lo que significa que el programa puede estar haciendo otras cosas mientras el controlador está descargando o cargando datos de píxeles. Utilizando la información de vértices previamente cargada en la GPU y las matrices de transformación correspondientes, que son matrices de modelo, vista y proyección (MVP) [32]. Esta información es utilizada como entrada para el primer programa compuesto por el conjunto de shaders en GPU que permiten el renderizado en una textura las cinco caras de un hemicubo, la cual es accedida por el próximo conjunto de shaders.

8. El segundo programa que se ejecuta en GPU está compuesto por un Vertex Shader el cual procesa dos triángulos formados por los vértices de las esquinas de la textura generada anteriormente. La razón para esto es que así se cubre toda la textura con varias instancias del Fragment Shader (uno por píxel en la textura). Dentro de cada Fragment Shader se obtiene la posición a procesar de la textura, el color en tres canales (RGB) y se decodifica el índice. El resultado de este programa es una textura que representa una fila completa de la matriz de factores de forma. Para poder realizar las sumas atómicas de diferentes instancias del Fragment Shader en la misma posición de la textura resultante, se utilizó una extensión de OpenGL que NVIDIA proporciona en las tarjetas gráficas modernas que se puede encontrar en [33].

9. Uno de los pasos opcionales del algoritmo permite elegir si se debe renderizar en pantalla las vistas desde los hemicubos a medida que se van generando. Este parámetro se define en el archivo de configuración.

10. Retorno de la matriz de factores de forma tanto en formato completo como disperso. La matriz no es mantenida completamente en memoria principal. En lugar de esto se mantiene una fila completa de la misma mientras se escribe un archivo con su información.

3.5.3. Parámetros

Para poder adaptar el algoritmo a diferentes escenas y así mejorar los tiempos de procesamiento se utilizan 2 parámetros:

- Máximo de niveles del octree:

  La profundidad del octree influye de muchas formas el rendimiento del algoritmo. Si se decide un máximo muy chico, no se saca provecho de la oclusión presente en la escena. Es decir, no queda definido un orden de llamadas para renderizar aprovechando características como la cercanía a la cámara.

- *MinParches* - Cantidad mínima de parches para determinar el uso de occlusion queries:

  La cota del mínimo de parches para realizar occlusion queries debe ser elegida de forma en que estas tomen mucho menos tiempo que la orden de renderizado. La occlusion query se realiza directamente sobre un nodo del octree, es decir en una estructura cúbica seleccionada de forma artificial con independencia de los parches de la escena pero que provee información general sobre la posición de los parches (todos los parches contenidos están ocluidos siempre que el nodo al que pertenecen también lo esté). Esto se debe a que realizar una occlusion query [30] para pocos parches puede ser contraproducente.

3.5.4. Interfaz de prueba

Con el objetivo de facilitar las pruebas funcionales del proyecto, se desarrolló una interfaz simple de prueba por línea de comandos (Figura 3.6). La misma permite seleccionar cuál de las escenas se quiere utilizar, cambiar parámetros de ejecución y calcular el error generado en las matrices de salida. Esto permite tener mucha más flexibilidad en el código al no tener que compilar la solución nuevamente cada vez que se quieren probar parámetros nuevos.

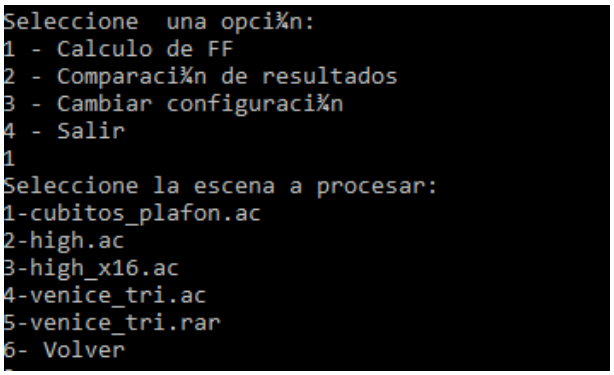


Figura 3.6: Interfaz de prueba.

3.5.5. Dificultades encontradas

- El diagnóstico de errores de implementación es difícil de visualizar. Encontrar errores a nivel de GPU no es una tarea sencilla. OpenGL no posee un sistema de

detección de errores comparable al de CUDA por lo que se tuvieron que realizar pruebas ingeniosas sobre las nuevas funcionalidades desarrolladas para asegurar que no tuvieran errores, previo a su integración al algoritmo.

- Aislar el programa del resto de la ejecución de tareas en el PC de prueba fue uno de los principales problemas que se tuvo que resolver. Obtener resultados certeros en las pruebas es fundamental para conocer el verdadero rendimiento del algoritmo. Si el proceso no era priorizado frente a otras tareas corriendo dentro del PC entonces se podrían llegar a obtener resultados erróneos que condujeran a una implementación menos eficiente. Esto fue solucionado ejecutando las pruebas con la menor cantidad posible de programas ejecutándose paralelamente y modificando la prioridad del programa en el administrador de tareas del sistema.

- La bajada de velocidad de la CPU cuando se alcanza una determinada temperatura (*throttling*) fue otro de los inconvenientes que degradaban la performance, afortunadamente se pudo solucionar con drivers específicos del procesador que permitían evitar el *throttling* pero con el cuidado de no sobrecalentar demasiado el CPU.

- Los largos tiempos de ejecución en escenas grandes dificultaron las tareas de *testing*. Para reducir las dificultades, se idearon tests alternativos, como por ejemplo tomar muestras de varios conjuntos de parches consecutivos, y detener la ejecución al haber recorrido 10 % de todos los parches de la escena.

Capítulo 4

Análisis experimental

En esta sección se realiza un análisis del rendimiento del algoritmo implementado. Se comparan mediciones de tiempos de ejecución, consumo de memoria y de procesamiento. Se estudia la variación del tiempo conforme se varían los distintos parámetros de configuración. También se realizan tablas comparativas con una implementación alternativa del algoritmo del hemicubo propuesta en [9] desarrollado utilizando CUDA y C++, la cual no utiliza las técnicas de la visibilidad jerárquica del Z-Buffer. La convención utilizada fue nombrar al algoritmo propuesto en [9] "algoritmo originalz al propuesto en este proyecto de grado "algoritmo optimizado".

4.1. Descripción de las pruebas

4.1.1. Hardware y software

Las pruebas experimentales fueron realizadas en un PC con las características especificadas en la Tabla 4.1.

Tabla 4.1: Tabla especificaciones PC de prueba.

SO	Windows 10 Home Edition 64 bits
GPU	NVIDIA 740m
Máx GPU Mem.	2048 MB
CPU	Intel Core i7-4700MQ
CPU RAM	8GB de RAM
CUDA Version	8.0

Para realizar las pruebas se utilizó la herramienta Profiler del IDE Visual Studio 2015, la cual proporciona métricas tales como el uso del CPU, GPU y RAM. También se realizaron mediciones de tiempo dentro del programa en múltiples sectores que se consideraron importantes. Por último también se implementó un método de comparación de normas entre matrices utilizando la herramienta Matlab 2013 [34].

4.1.2. Escenas

Las escenas de las prueba fueron elegidas con el propósito de resolver el problema de radiosidad para grandes ciudades.

Tabla 4.2: Tabla de escenas de prueba.

Escena	Cantidad de parches	Tamaño en memoria
Cornell box	40	0,004 MB
Cornell box×256	10.240	0,97 MB
City	8.897	0,80 MB
City×16	142.352	10 MB
Venecia	1.495.068	147 MB

La primera escena es una escena simple que consta de 40 polígonos o "parches" formando unos pocos objetos tridimensionales. El objetivo de utilizar esta escena es tener un ambiente de carga mínima en el cual probar rápidamente los cambios en el algoritmo. A esta escena se le conoce como *Cornell box* [35] (Figura 4.1). También se cuenta con una versión subdivida de esta escena en 4 niveles de 10.240 parches ($10240 = 40 \times 4^4$). Esta se obtuvo utilizando el algoritmo de subdivisión mencionado en la Sección 3.5.2 y es utilizada como referencia de escena con poca oclusión.

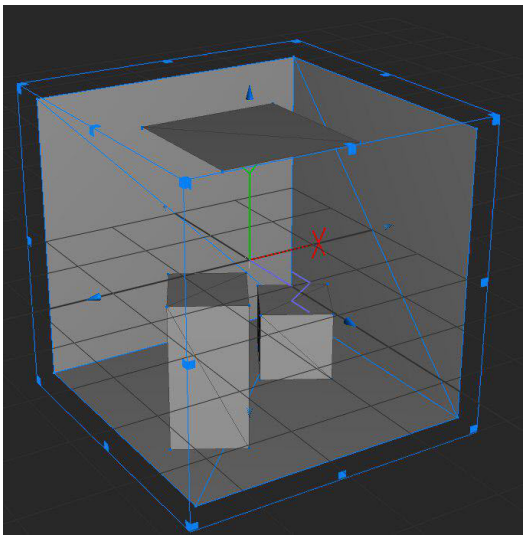


Figura 4.1: Escena Cornell box.

En segundo lugar se tiene una escena mediana compuesta por 8.897 parches, los cuales componen diversos rascacielos de una ciudad. Probando los algoritmos con esta escena ya se pueden encontrar diferencias de tiempo entre los mismos ya que se requiere mucho más poder de cómputo que en la escena simple. Para obtener una escena de tamaño más grande lo que se hizo fue realizar una división de cada uno de los parches en 16 parches más pequeños para obtener una escena de 142.352 parches. Visualmente

sigue siendo la misma ciudad, pero con superficies más discretizadas. A estas escenas se les dio el nombre de *City* y *City×16* respectivamente (Figura 4.2).

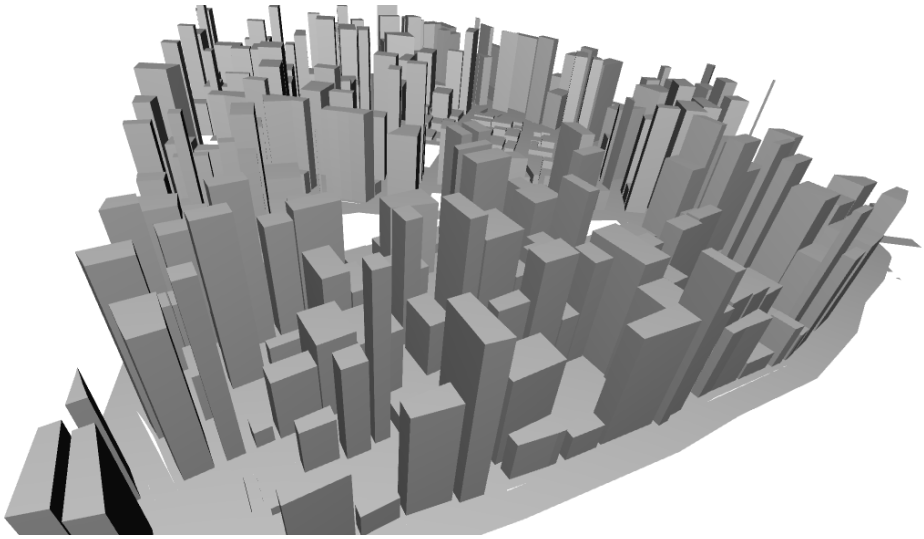


Figura 4.2: Escenas mediana y grande. City(8.852) y City×16 (8.852 x 16 = 142.352)

Por último se tiene una ciudad con una cantidad de parches 10 veces superior a la ciudad grande (1.495.068) y es un modelo tridimensional de la ciudad de Venecia (Figura 4.3). Para poder trabajar con la matriz de factores de forma de Venecia y de City × 16 se debe hacer pura y exclusivamente de forma dispersa, debido a su gran tamaño. Esto es posible gracias a que estas escenas generan matrices que poseen baja densidad de elementos no nulos.

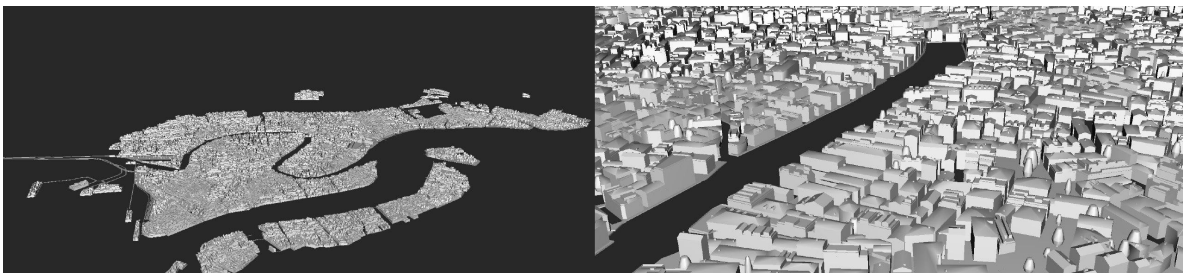


Figura 4.3: Modelo de ciudad de Venecia (1.495.068).

4.2. Análisis de tiempos de ejecución

Otro de los estudios realizados fue un análisis de tiempos de ejecución del algoritmo variando los parámetros configurables dentro del programa. A partir de estas mediciones es posible deducir formas de optimizar estos parámetros para una escena

dada. También se realizó una comparación de tiempos con el algoritmo original sobre las mismas escenas.

4.2.1. Estudio paramétrico

Para este estudio paramétrico lo que se tomó en cuenta fueron dos variables configurables, que ya fueron detalladas en la Sección 3.5.3:

- Máximo nivel del octree.
- *MinParches* - Mínima cantidad de parches en una hoja del octree.

Las escenas de prueba que se utilizaron son la mediana (City) y grande (City×16). La escena pequeña (Cornell box) no fue tenida en cuenta debido a su baja cantidad de parches. Cada configuración fue ejecutada 10 veces y luego se realizó un promedio del tiempo total.

Pruebas parámetricas en City

En esta escena los resultados son de bastante importancia. Al contar con cerca de nueve mil parches los experimentos resultan adecuados para validar diversos aspectos del algoritmo. En la Tabla 4.3 se pueden observar los tiempos obtenidos para cada uno de los casos de prueba. Con ellos se generó la Figura 4.4 en donde se presenta una gráfica con las mejoras obtenidas en cuanto a velocidad del algoritmo según el tiempo reportado para el algoritmo original (25,31 segundos). Algo interesante que sucedió fue que en la ejecución con el octree de 3 niveles se redujo el tiempo al realizar una menor cantidad de occlusion queries debido al mínimo de parches incrementado. Por otro lado, en la ejecución del octree de 5 niveles, a medida que se incrementa la variable *MinParches* se puede ver como el tiempo de ejecución también es cada vez más grande. El menor tiempo se vio con el octree configurado en 5 niveles y 300 parches, pero hay otras configuraciones que dan resultados similares. Para el octree de 10 niveles, es lógico que los tiempos sean muy similares ya que es muy poco probable encontrar un nodo que posea más de 300 parches debido a la cantidad de nodos resultantes. Se puede ir apreciando en estos resultados lo costoso que es realizar occlusion queries y que para las escenas de prueba involucradas, si se realizan filtros de 300 parches o menos, el costo de los mismos es superior al ahorro que generan.

Pruebas parámetricas en City ×16

Por último se estudió la escena grande, que cuenta con 142.352 parches. De la misma manera que para la escena City, se construyó la Tabla 4.4 con los tiempos obtenidos en las pruebas, y luego se graficaron los datos con respecto al tiempo obtenido en el algoritmo original (2.840 segundos). Lo primero que salta a la vista en la Figura 4.5 es que entre la peor configuración y la mejor hay más de 100 segundos de diferencia. Se puede decir entonces que es posible obtener hasta un 40 % de mejora de rendimiento sobre una escena variando estos dos parámetros. El octree de 10 niveles fue el que

Tabla 4.3: Pruebas paramétricas - City.

Máximo nivel del octree	Mínimo de parches	Tiempo (s)	Aceleración
3	300	13,07	1,94×
3	600	12,07	2,10×
3	1.000	11,9	2,13×
5	**300**	**11,24**	**2,25×**
5	600	11,79	2,15×
5	1.000	11,93	2,12×
10	300	11,38	2,22×
10	600	11,29	2,24×
10	1.000	11,34	2,23×

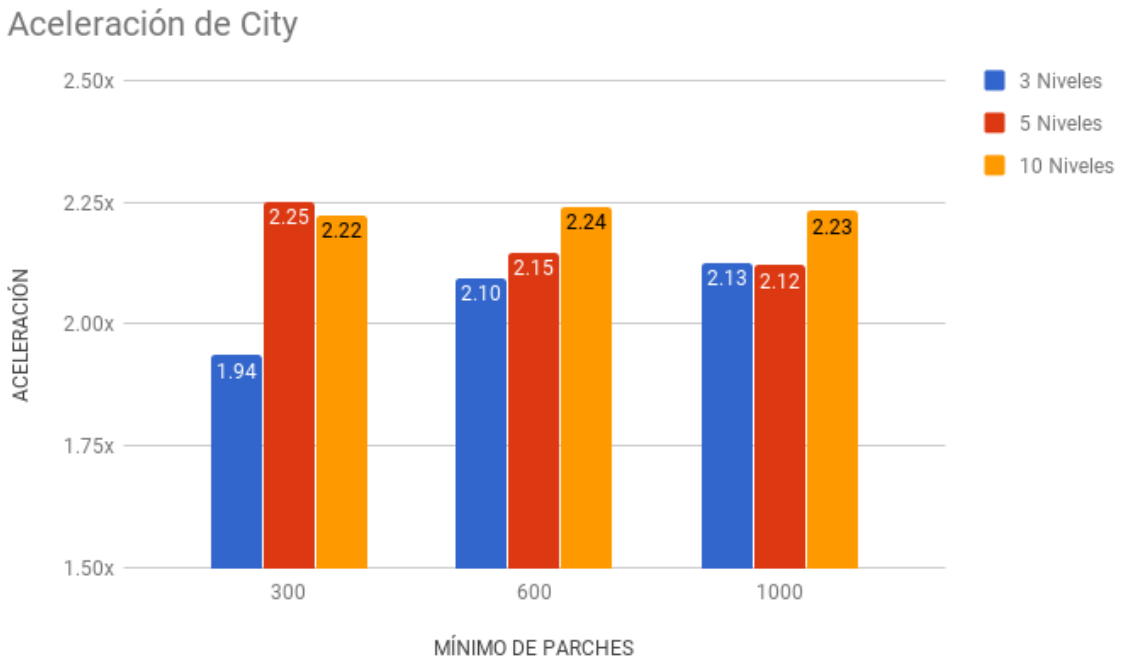


Figura 4.4: Aceleración de City con respecto al algoritmo original.

consiguió la mejor performance para esta escena, y su configuración de *MinParches* en 600 logró el mejor tiempo, por lo que se logró un buen equilibrio entre la cantidad de occlusion queries realizadas y los renderizados ahorrados en esta configuración. No es el caso para el octree de 3 niveles, que empeoró rotundamente su tiempo al decrementar la cantidad de occlusion queries realizadas y luego volvió a obtener un tiempo más adecuado al decrementarlas incluso más. Para el caso del octree de 5 niveles se ve cómo fue mejorando a medida que se incrementaba la variable *MinParches*, lo cual parece un comportamiento razonable como se vio en las escenas anteriores. El costo de las

occlusion queries puede llegar a ser muy alto, y al tener la escena subdividida con pocos niveles en el octree, se puede aprovechar para descartar solamente aquellos nodos realmente pesados y dibujar todos los nodos que posean poca cantidad de parches. De esta manera se puede mejorar el desempeño del algoritmo.

Tabla 4.4: Pruebas paramétricas - City×16.

Máximo nivel del octree	Mínimo de parches	Tiempo (s)	Aceleración
3	300	810,4	3,50×
3	600	894,7	3,17×
3	1.000	832,9	3,41×
5	300	848,4	3,35×
5	600	831,1	3,42×
5	1.000	816,3	3,48×
10	300	821,2	3,46×
10	**600**	**794,5**	**3,57×**
10	1.000	802,2	3,54×

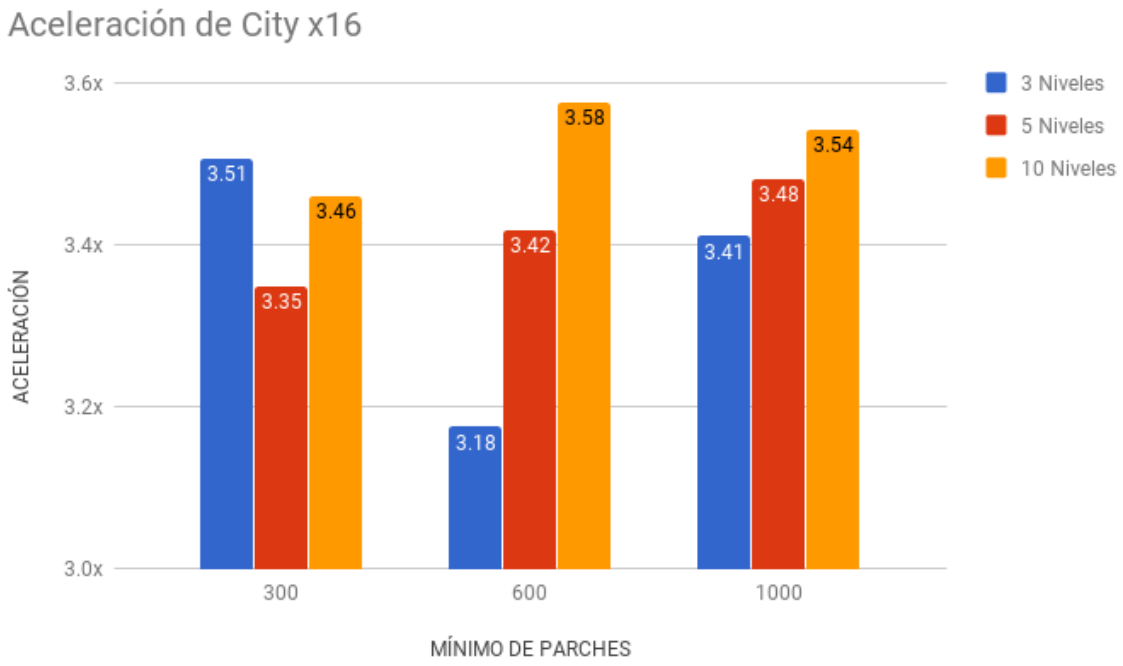


Figura 4.5: Aceleración de la escena City×16 con respecto al algoritmo original.

4.2.2. Optimización de parámetros sobre Venecia

Además del estudio realizado con las escenas City y City×16, se buscó optimizar los parámetros configurables para la escena de Venecia. El objetivo es validar la escalabilidad del algoritmo y que se saque el mayor provecho posible de las técnicas implementadas en él. Para esto se realizaron pruebas variando el nivel del octree entre 1 y 5 niveles, a su vez también se varió el *MinParches* entre 500, 1.000 y 2.000 para cada nivel. Los resultados de las pruebas se pueden observar en la Tabla 4.5.

Tabla 4.5: Pruebas parámetricas - Venecia.

Máximo nivel del octree	Mínimo de parches	Tiempo (s)	Aceleración
1	500	67.259,36	1,10×
1	1.000	66.384,29	1,12×
1	2.000	68.482,88	1,09×
2	500	58.592.27	1,27×
2	1.000	61.231,62	1,21×
2	2.000	63.993,00	1,16×
3	500	39.337.17	1,89×
3	1.000	34.545,03	2,15×
3	2.000	42.437,51	1,75×
4	500	33.462.03	2,22×
4	**1.000**	**26.073,71**	**2,85×**
4	2.000	36.248,42	2,05×
5	500	36.376,05	2,04×
5	1.000	39.386,09	1,89×
5	2.000	48.708,52	1,53×

Como se puede observar, la amplia gama de valores probados permite distinguir con qué parámetros se alcanza un mejor rendimiento. En este caso se ve que el mejor de los valores se dio cuando el nivel del octree se establece en 4 y *MinParches* en 1.000, alcanzando una reducción de tiempo de un 62 %. Se muestran los resultados hasta el quinto nivel debido a que a partir del mismo los tiempos empiezan a empeorar y no resultan relevantes para el análisis. En la Figura 4.6 se presentan los resultados de acuerdo a la mejora que significan en cuanto a la velocidad de ejecución del algoritmo, tomando el tiempo del algoritmo original como base de la comparación.

4.2.3. Comparación con algoritmo original

Con el objetivo de validar las virtudes del algoritmo optimizado, se realizó una comparación de desempeño con el algoritmo original, el cual no posee los dos componentes en los que se basa este proyecto, esto es: el uso de shaders en vez de CUDA en la implementación del Hemicubo (menor transferencia GPU/CPU) y el Z-Buffer jerárquico. Al igual que en las pruebas anteriores se utilizaron las escenas con los distintos tamaños,

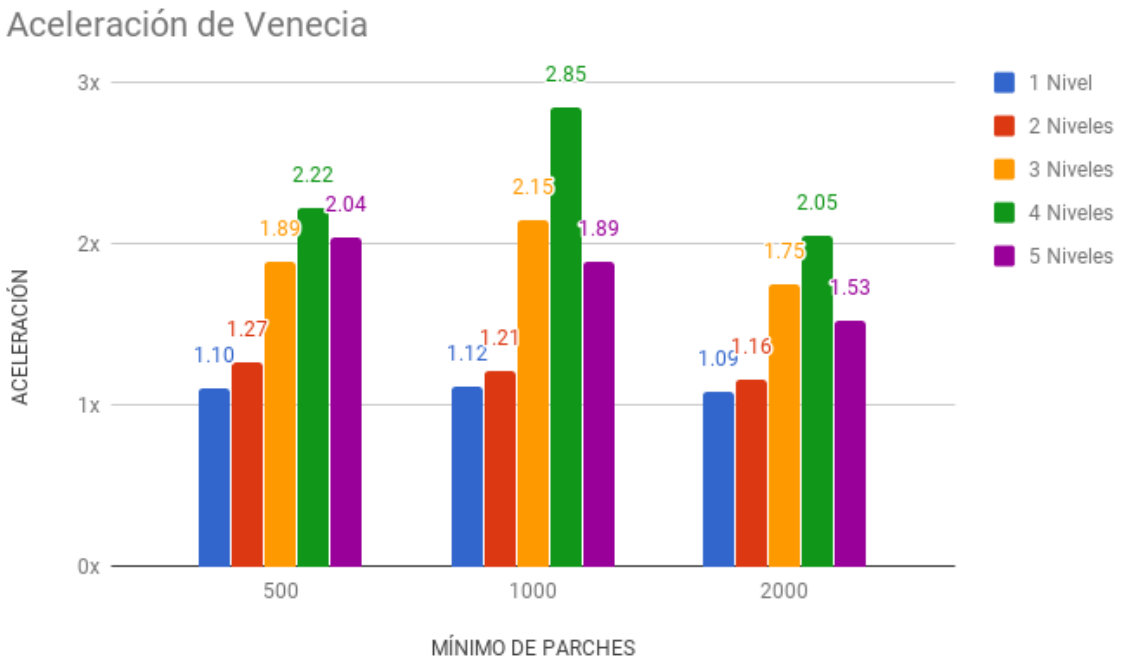


Figura 4.6: Aceleración del algoritmo con respecto al algoritmo original variando parámetros para Venecia.

en las cuales se ejecutó una muestra de 10 corridas de ambos algoritmos con el fin de obtener un tiempo promedio para cada una de las escenas. En la Tabla 4.6 se pueden apreciar las mejoras de los tiempos para cada escena, se logra reducir el tiempo de ejecución entre un 50 y un 62 por ciento para cada una de ellas, lo que se traduce en una mejora en la velocidad del algoritmo de entre 2× y 3,57× además de lograr soportar archivos más grandes como es el caso del modelo de Venecia con casi 1 millón y medio de parches. Cabe destacar que en el caso del algoritmo optimizado que posee parámetros configurables, se utilizó la mejor configuración encontrada.

Tabla 4.6: Tabla comparativa de algoritmos.

Escena	Algoritmo original(s)	Algoritmo optimizado(s)	Aceleración
Cornell box×256	28,73	11,93	2,41×
City	25,31	11,96	2,12×
City×16	2.840	795,5	3,57×
Venecia	74.306	26.073	2,85×

4.2.4. Estudio de aceleración de shaders

En estas pruebas se apunta a obtener resultados acerca de la mejoras obtenidas gracias al uso de *shaders*. Para esto se realizó una comparación del algoritmo original contra el algoritmo optimizado sin la estructura octree y sin realizar *frustum culling* ni *occlusion culling*. La Tabla 4.7 presenta los resultados obtenidos, en los que se puede observar magnitudes de aceleración muy altas para las escenas con menos cantidad de parches. Tienen sentido estas diferencias ya que el overhead que se generaba con las transferencias GPU-CPU disminuyó al utilizar shaders. En Venecia este ahorro es menos perceptible que en el resto de las escenas porque el overhead de la transferencia es mucho menor con respecto al tiempo necesario para el procesamiento de geometrías no visibles.

Tabla 4.7: Pruebas paramétricas - octree.

Escena	Algoritmo original(s)	Optimizado (sin octree)(s)	Aceleración
Cornell box×256	28,73	11,93	2,41×
City	25,31	13,41	1,89×
City×16	2.840	837,6	3,39×
Venecia	74.306	62.401	1,19×

4.2.5. Estudio de aceleración del octree

El objetivo de estas pruebas es determinar las mejoras en rendimiento por hacer uso del octree, junto con las operaciones de filtrado de geometrías sobre este. En la Tabla 4.8 se observa la comparación del algoritmo optimizado aprovechando la estructura del octree contra el tiempo obtenido sin aprovechar esta estructura. Los resultados son favorables en todas las escenas salvo en Cornell box ×256, y en la que mayor mejora se puede observar es en la escena con mayor cantidad de parches, con una aceleración de ×2,39, lo que valida las virtudes de los algoritmos de filtrado. La razón por las que Cornell box ×256 no presentó mejoras es debido a que es una escena con baja oclusión, por lo que el occlusion culling no es aprovechable.

Tabla 4.8: Pruebas parámetricas - octree.

Escena	Optimizado (sin octree) (s)	Optimizado(s)	Aceleración
Cornell box×256	11,93	13,66	0,87×
City	13,41	11,96	1,12×
City×16	837,6	795,5	1,05×
Venecia	62.401	26.073	2,39×

4.3. Análisis de rendimiento

Las gráficas generadas por las herramientas de diagnóstico durante la ejecución del algoritmo presentado aportan información muy interesante para entender el comportamiento del mismo. A continuación se analiza la utilización de recursos bajo distintas cargas por ambos algoritmos, el propuesto en este proyecto al que se le llamó optimizado y el propuesto en [9] al que se le denominó original.

4.3.1. Uso de CPU

A continuación se analiza el rendimiento del CPU para ambos algoritmos en las distintas escenas de prueba.

CPU en Cornell box

Esta escena es la de menor carga dentro de las de prueba. En su corto tiempo de procesamiento se puede apreciar que el uso de GPU es mínimo, tan solo un 2,4 % en su punto más alto, mientras que el uso de CPU llega a un máximo de 14 %. Se puede concluir que el algoritmo no es suficientemente exigido con una escena de este porte y que el tamaño de la misma permite validar rápidamente el algoritmo como se quiere pero no proporciona una idea clara de la potencia del mismo.

CPU en City

Ya con una escena con bastante más carga se nota como se empiezan a aprovechar más los recursos de la computadora. En primer lugar, se observa que la utilización de la GPU crece rápidamente, en tan solo 4 segundos se alcanza un 93 % de utilización. Paralelamente el CPU realiza su trabajo llegando a un 25 % de utilización, lo que equivale al 100 % de utilización de un procesador (recordar que la configuración de la PC de prueba es de 4 procesadores físicos)

Comparación con algoritmo previo

Empezando la comparación con la escena pequeña, es interesante notar cómo en una escena donde se realiza tan poco procesamiento, la variación en utilización de recursos es un poco grande. En cuanto a la GPU, con el algoritmo original se obtuvo una utilización máxima de 6 %, más del doble que lo registrado con el algoritmo optimizado. El CPU se encuentra más alineado con un máximo de 16 % de utilización, solamente un 2 % más alto.

En cuanto a la escena City, en comparación con el algoritmo optimizado se observa que el programa alternativo utiliza en menor medida la GPU en una ejecución estándar, llegando a su pico solamente al final de la ejecución, y siempre manteniéndose por debajo del 50 %. En cuanto al CPU ambos cuentan con una baja utilización, siendo el algoritmo optimizado un poco más intensivo en cuanto al uso de la misma.

4.3.2. Uso de memoria RAM

En esta sección se utilizaron las herramientas de análisis con el propósito de determinar la cantidad de memoria reservada por los procesos durante la ejecución del algoritmo sobre las escenas de prueba.

En primer lugar se ve en las imagenes 4.7 y 4.8 que las curvas de memoria del proceso son similares para ambas escenas de prueba, realizan una especie de cresta hacia arriba y otra hacia abajo y luego van creciendo hacia al final que es cuando se realiza la impresión de la matriz a un archivo, lo cual consume una cantidad considerable de memoria. También se observa que para la escena pequeña el máximo de memoria consumida por el proceso es de 39,5 MB y para la escena City es de 60,7MB.

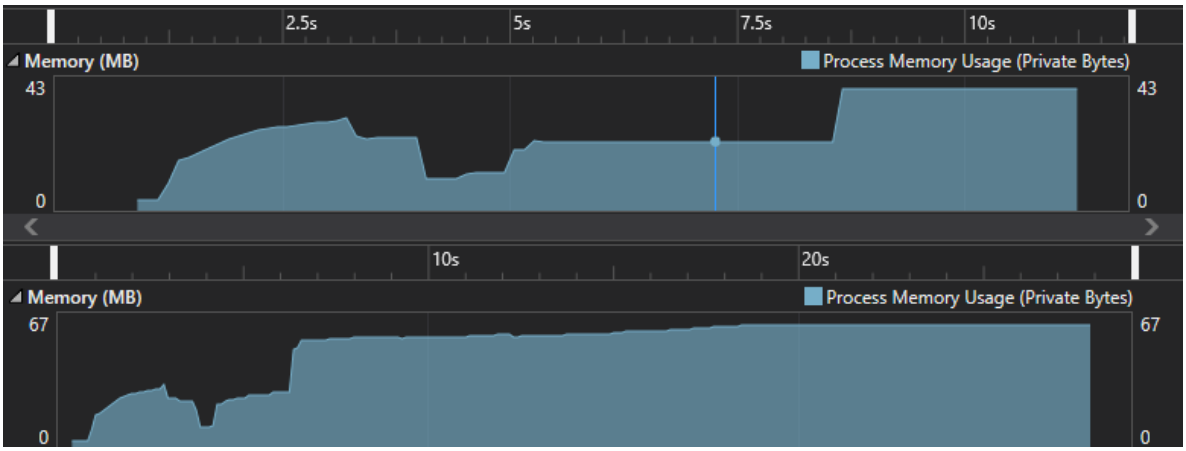


Figura 4.7: Uso de memoria RAM para Cornell box y City (algoritmo optimizado).

Realizando la comparación con el algoritmo original, se distingue un uso mayor de memoria RAM en ambas escenas, presente sobre todo al final del algoritmo. Esto se debe a que existe una mayor ineficiencia al procesar la matriz desde la GPU al momento de imprimir. Fuera de esa diferencia, la forma de la curva y la cantidad de memoria consumida por el proceso a lo largo del algoritmo es similar para ambos.

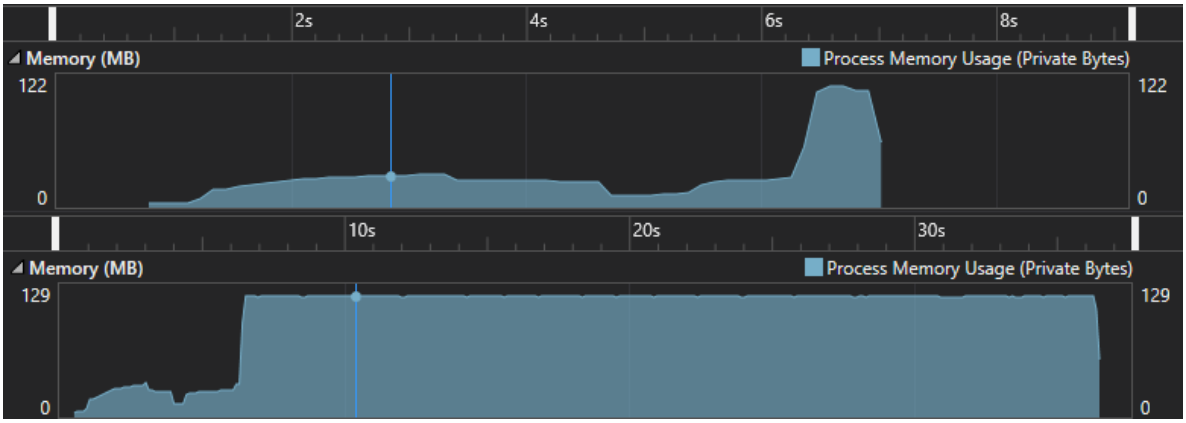


Figura 4.8: Uso de memoria RAM para escenas Cornell box y City (algoritmo original).

4.4. Estudio del error

A continuación se medirá el error tomando como referencia la matriz de factores de forma $\mathbf{F}$ del algoritmo original ($\mathbf{F}_{Orig}$) comparándola con la del algoritmo optimizado ($\mathbf{F}_{Opt}$) .

4.4.1. Norma de la distancia

La técnica utilizada para medir el error relativo fue la de calcular $\frac{\left\|\mathbf{F}_{Orig}-\mathbf{F}_{Opt}\right\|_F}{\left\|\mathbf{F}_{Orig}\right\|_F}$. Si el valor es relativamente pequeño, significa que la norma de la diferencia de las matrices es mucho menor que la norma de una de ellas, por lo que se puede considerar que las matrices tienen una gran similitud. Dada una matriz $\mathbf{A}$ cualquiera, la norma *Frobenius* se define como la raíz cuadrada de la sumatoria del cuadrado del módulo de todos sus elementos (Ecuación 4.1) [36]. En la Tabla 4.9 es posible ver los resultados obtenidos, que revelan la similitud en las matrices de salida para algunas de las escenas analizadas. No se incluyó la comparación con Venecia debido a que la cantidad de memoria RAM necesaria para realizar la prueba excede a la del hardware utilizado.

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2} \tag{4.1}$$

Tabla 4.9: Comparación de distancia de matrices.

Escena (parches)	$\frac{\left\|\mathbf{F}_{Orig}-\mathbf{F}_{Opt}\right\|_F}{\left\|\mathbf{F}_{Orig}\right\|_F}$
Cornell box	0,0033
City	0,0047
City×16	0,0056

Capítulo 5

Conclusiones y trabajo futuro

En este capítulo se concluye el trabajo sobre la técnica de aceleración del hemicubo, realizando un análisis de los resultados logrados y aportes para futuros trabajos que pueden continuar la línea de investigación abordada.

5.1. Conclusiones

Si bien el enfoque de este trabajo estuvo en la técnica del hemicubo, implícitamente se recorrió, investigó y validó un gran abanico de técnicas utilizadas en el campo de la computación gráfica, lo que demostró la flexibilidad que posee el campo por naturaleza, tanto en los problemas que se pueden resolver como en las soluciones que se pueden encontrar.

Luego de muchas iteraciones de desarrollo se logró llegar a una solución con una mejora significativa, con una aceleración de entre 2× y 3.5× con respecto a [9] y un error despreciable. Las *oclussion queries* en conjunto con el ordenamiento de la escena y el *frustum culling* permiten el aprovechamiento del alto factor de oclusión presente en la escena. Esto reduce los tiempos requeridos para el procesamiento de los hemicubos, al filtrar geometrías ocluidas. Estas técnicas resultan adecuadas para aplicarse sobre la estructura jerárquica en la que se divide la escena, el *octree*.

No solamente se logró un algoritmo más eficiente sino que también es más escalable, al trabajar siempre con matrices dispersas de una forma eficiente. Los requerimientos de memoria se redujeron drásticamente con respecto a [9], lo que permite entre otras cosas ejecutar el algoritmo sin problemas con la escena de 1,5 millones de parches, generando un archivo de salida de 9,3 GB. Si bien la mejora es considerable, depende de conocer los parámetros correctos previo a la ejecución del algoritmo para encontrar la máxima mejora. Esto se consigue empíricamente, teniendo en cuenta, la oclusión, densidad de parches y cantidad de parches, lo que permite obtener una idea de que valores son los que podrían funcionar mejor.

Otro aspecto a destacar fue el uso de *shaders*, que resultó muy adecuado para resolver el problema de cálculo de los factores de forma. En primer lugar se ahorró tiempo de transferencia entre CPU y GPU al reducir el número de veces que se transfieren

datos entre ellas con respecto al algoritmo original que utiliza CUDA. A su vez, permitió trabajar de una forma más uniforme con la tarjeta gráfica utilizando solamente OpenGL.

5.1.1. Aportes y resultados

1. Se validó una nueva forma de aplicar el método del hemicubo para el cálculo de radiosidad, proyectando los parches sobre cada hemicubo. Este proceso genera una textura, que puede ser procesada en forma paralela utilizando *shaders*. Todo esto solamente con un intercambio de memoria entre CPU y GPU por fila de la matriz de factores de forma (o sea por parche en la escena).

2. Se logró implementar un modelo para aprovechar las *occlusion queries* utilizando la estructura jerárquica del *octree*. Teniendo en cuenta que estas consultas son costosas, se podrá variar la configuración de acuerdo a la escena logrando mejores tiempos de ejecución. Cuanto mayor sea la cantidad de parches que haya en cada nodo del *octree*, más efectiva es esta técnica.

3. Se consiguió llevar a la práctica las ideas propuestas en [8] y se comprobó que las mismas siguen siendo efectivas (más de 20 años después de la publicación del articulo). Esto comprueba que si bien la tecnología sigue avanzando año tras año, incluso con cambios de arquitecturas de procesadores y tarjetas gráficas, muchos de los métodos ideados hace décadas pueden aplicarse a nuevas implementaciones con muy buenos resultados.

4. El manejo de matrices dispersas fue muy beneficioso para el proyecto, permitiendo procesar escenas con una gran cantidad de parches con la memoria disponible. No hubiese sido posible procesar la escena *Venecia* sin tener una representación eficiente de su matriz.

5.2. Trabajo futuro

Mientras el proyecto se encontraba en desarrollo, sobre todo en etapas avanzadas, surgieron ideas que se dejaron fuera del alcance del proyecto. En esta sección se resumen las que serían las más viables e interesantes.

La solución propuesta requiere que se configuren ciertos parámetros, el nivel del octree y el mínimo de parches establecido para realizar las *occlusion queries*. Debido a que el tiempo de ejecución para una escena está directamente ligado a estos, es de interés poder estimar en base a las propiedades de la escena cuáles serían los parámetros ideales en cada caso para calcular la matriz de factores de forma de la misma.Se sabe que no es tarea sencilla dar con los valores exactos, pero hay información suficiente para poder hallar buenas aproximaciones. Una forma de hacer esto es a través de la búsqueda de relaciones entre las configuraciones posibles y las aceleraciones conseguidas. Por ejemplo, para escenas con una gran cantidad de parches, como Venecia (1.495.068), se obtienen

mejores resultados con niveles del octree más altos. Mediante un análisis exhaustivo de escenas y parámetros se pueden calcular estas relaciones con mejor precisión.

Otro de los aspectos que interesa conocer más a fondo es el comportamiento del algoritmo en otro tipo de escenas con alta oclusión, como por ejemplo laberintos. A su vez también se podría combinar con técnicas que ayuden al procesamiento de escenas de baja oclusión como la visibilidad precalculada [37].

Las técnicas de aceleración que se utilizaron en este algoritmo son perfectamente aplicables a otros métodos de cálculo de matrices de factores de forma. Independientemente de si se trata de métodos área diferencial a área o área a área, se pueden aplicar las técnicas de filtrado para reducir la cantidad de parches renderizados y de esa forma ahorrar tiempo de procesamiento. También podría estudiarse la viabilidad de aplicarse a métodos híbridos más modernos como el propuesto por Walton [38] en el que muestra como optimizar el cálculo de factores de forma para parches muy cercanos con un algoritmo de integración adaptativa.

En el algoritmo optimizado, todos los cálculos de visibilidad se realizan en tiempo real, luego de cargar la escena. Existen al día de hoy diversos métodos que, realizando un preprocesado de la escena ayudando a clasificar los distintos objetos, pueden resolver mucho más rápido estos cálculos. Algunos de estos estudios, como el de Wonka *et al.* [39], se centran en cómo precalcular esta visibilidad en ambientes de ciudades con mucha oclusión. En Wonka *et al.* se hace un preprocesamiento de la escena la cual se divide en celdas, para cada celda se precomputa la visibilidad celda-objeto y se combina con *occlusion culling* para una ganancia muy alta de rendimiento en el caso del renderizado de tiempo real. Sería muy interesante integrar la propuesta a la solución que se propuso en este proyecto.

Resulta de gran interés incluir superficies con propiedades especulares en la escena. Con esto se podría simular materiales como el vidrio en los edificios modernos. La inclusión de factores de forma extendidos puede ser de gran utilidad para agregar dicha funcionalidad [40]. Esto podría ser implementado utilizando la técnica de portales en OpenGL, convirtiéndolo en un método de dos pasos [41].

Capítulo 6

Glosario

API - *Application Programming Interface* (Interfaz de programación de aplicaciones) es un conjunto de subrutinas, funciones y procedimientos que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.
C++ - Lenguaje de programación orientada a objetos diseñado a mediados de los años 1980 por Bjarne Stroustrup.
Color Bleeding - Fenómeno en que los objetos y superficies son coloreados por superficies cercanas.
CPU - Del inglés *Central Processing Unit* (Unidad de procesamiento central) es el componente de hardware que interpreta las instrucciones de un programa informático mediante la realización de las operaciones básicas aritméticas, lógicas y de entrada/salida del sistema.
CUDA - Son las siglas de *Compute Unified Device Architecture* (Arquitectura Unificada de Dispositivos de Cómputo) que hace referencia tanto a un compilador como a un conjunto de herramientas de desarrollo creadas por NVIDIA que permiten a los programadores usar una variación del lenguaje de programación C para codificar algoritmos en GPU de NVIDIA.
Escena - Se le llama escena a un escenario en tres dimensiones construido a partir de uno o más archivos con el formato adecuado, conteniendo la información de vértices y/o texturas.
Form Factor - Es la cantidad de energía transferida de un parche a otro de la escena, utilizado en el método de radiosidad
Frostbite - Motor gráfico propiedad de *Electronic Arts*, utiliza una solución de radiosidad optimizada para tiempo real.
Frustum Culling - Técnicas de recorte del *Frustum*(cono de visión), utilizada para no gastar recursos en los objetos que no se ven en pantalla.
GB - Gigabyte, 10^9 Bytes
Git - Es un software de control de versiones diseñado por Linus Torvalds, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando estas tienen un gran número de archivos de código fuente.
GPU - *Graphics Processor Unit* (Unidad de procesamiento gráfico) es un coprocesador dedicado al procesamiento de gráficos u operaciones de punto flotante, para aligerar la

carga de trabajo del procesador central en aplicaciones como los videojuegos o aplicaciones 3D interactivas.
GLSL - *OpenGL Shading Language* (abreviado GLSL o GLslang) es un lenguaje de alto nivel de sombreado con una sintaxis basada en el lenguaje de programación C. Permite crear shaders dándole al desarrollador más posibilidades de controlar el *pipeline* de renderizado.
Hemicubo - Es un cubo dividido a la mitad, utilizado en la técnica del hemicubo para calcular la radiosidad, se posiciona el cubo de manera que el corte por la mitad quede sobre la superficie que se está procesando y se calculan los *form factors* para las 4 medias caras laterales y la cara superior del mismo.
Mental Ray - es un motor de renderizado, desarrollado por *Mental images* en Berlín (Alemania). Es uno de los pocos motores desarrollados en Europa que compiten con los desarrollados en EE. UU. y Canadá.
MVP - *Model View Projection* (Modelo, vista y proyección) es la matriz resultante del producto de las matrices Model, View y Projection, que se separan con el objetivo de mantener las transformaciones del mundo tridimensional separadas.
Occlusion Query - Técnica implementada sobre el hardware de la GPU para consultar si una geometría se vería en pantalla.
Octree - Técnica de volúmenes acotantes, ayuda a mantener una jerarquía de objetos en el mundo tridimensional.
OpenGL - *Open Graphics Library* (Biblioteca de gráficos abierta) es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D.
Parche - Nombre que se le da a cada elemento de las superficies de una escena en la técnica de radiosidad.
Píxel - es la menor unidad homogénea en color que forma parte de una imagen digital.
Radiosidad - Es una técnica para el cálculo de la iluminación global que trabaja con superficies de reflexión difusa y emisores difusos. Se basa en la discretización de las superficies de la escena en un conjunto finito de elementos discretos (parches), y en la resolución de un sistema lineal.
Shader - La tecnología *shaders* o coloreadores es cualquier unidad escrita en un lenguaje de coloreado que se puede compilar independientemente. Es una tecnología reciente y que ha experimentado una gran evolución destinada a proporcionar al programador una interacción con la unidad de procesamiento gráfico (GPU) hasta ahora imposible. Los *shaders* son utilizados para realizar transformaciones y crear efectos especiales, como por ejemplo iluminación, fuego o niebla. Para su programación los *shaders* utilizan lenguajes específicos de alto nivel que permitan la independencia del hardware.
Textura - es una imagen de mapa de bits que se usa para cubrir la superficie de un objeto virtual, ya sea tridimensional o bidimensional, con un programa de gráficos especial.
RAM - *Random Access Memory* (Memoria de acceso aleatorio) se utiliza como memoria de trabajo de computadoras para el sistema operativo, los programas y la mayor parte del software. En la RAM se cargan todas las instrucciones que ejecuta la unidad central de procesamiento (procesador) y otras unidades del computador. La GPU también

posee su propia memoria RAM.
Render - Renderización en español, es un término usado en jerga informática para referirse al proceso de generar una imagen o vídeo mediante el cálculo de iluminación partiendo de un modelo en 3D.
RGB - sigla en inglés de *red*, *green* , *blue*, en español rojo, verde y azul o RVA (sigla preferida por la ASALE y la RAE) es la composición del color en términos de la intensidad de los colores primarios de la luz.
Visual Studio - es un entorno de desarrollo integrado (IDE, por sus siglas en inglés) para sistemas operativos Windows. Permite el desarrollo en C++ entre otros lenguajes.
Z-Curva - es una función que mapea datos multidimensionales a una sola dimensión, preservando la localidad de los puntos.
Z-Buffer - es la parte de la memoria de un adaptador de vídeo encargada de gestionar las coordenadas de profundidad de las imágenes en los gráficos en tres dimensiones (3D), normalmente calculados por hardware y algunas veces por software.

Bibliografía

[1] D. Halliday, R. Resnick, and K. S. Krane, *Física: Volumen 2*. Compañía Editorial Continental, 1994.

[2] J. F. Hughes and J. D. Foley, *Computer graphics: principles and practice*. Pearson Education, 2014.

[3] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile, "Modeling the interaction of light between diffuse surfaces," in *ACM SIGGRAPH Computer Graphics*, vol. 18, no. 3. ACM, 1984, pp. 213–222.

[4] E. M. Sparrow and R. D. Cess, "Radiation heat transfer," *Series in Thermal and Fluids Engineering, New York: McGraw-Hill, 1978, Augmented ed.*, 1978.

[5] "NVIDIA® mental ray," http://www.nvidia.es/object/nvidia-mental-ray-es.html, accedido por última vez: 2017-11-20.

[6] "Electronic arts - a real-time radiosity architecture," https://www.ea.com/frostbite/news/a-real-time-radiosity-architecture, accedido por última vez: 2017-10-01.

[7] M. F. Cohen and D. P. Greenberg, "The hemi-cube: A radiosity solution for complex environments," in *ACM SIGGRAPH Computer Graphics*, vol. 19, no. 3. ACM, 1985, pp. 31–40.

[8] N. Greene, M. Kass, and G. Miller, "Hierarchical z-buffer visibility," in *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. ACM, 1993, pp. 231–238.

[9] J. P. Aguerre, "Efficient representations of large radiosity matrices," 2016.

[10] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-time rendering*. CRC Press, 2008.

[11] J. T. Kajiya, "The rendering equation," in *ACM Siggraph Computer Graphics*, vol. 20, no. 4. ACM, 1986, pp. 143–150.

[12] "Lamberts Law," http://scienceworld.wolfram.com/physics/LambertsLaw.html, accedido por última vez: 2017-12-26.

[13] P. Bak, "Real time ray tracing," M.S. thesis, Technical University of Denmark, DTU, DK-2800 Kgs. Lyngby, Denmark, 2010.

[14] E. P. Lafortune and Y. D. Willems, "Bi-directional path tracing," 1993.

[15] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan, "Photon mapping on programmable graphics hardware," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. Eurographics Association, 2003, pp. 41–50.

[16] J. R. Howell, M. P. Menguc, and R. Siegel, *Thermal radiation heat transfer*. CRC press, 2010.

[17] M. F. Cohen and J. R. Wallace, *Radiosity and realistic image synthesis*. Elsevier, 2012.

[18] E. Fernández and G. Besuievsky, "A sample-based method for computing the radiosity inverse matrix," *Computers and Graphics*, vol. 41, no. Supplement C, pp. 1 – 12, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0097849314000259

[19] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, "A characterization of ten hidden-surface algorithms," *ACM Computing Surveys (CSUR)*, vol. 6, no. 1, pp. 1–55, 1974.

[20] T. Theoharis, *Algorithms for parallel polygon rendering*. Springer Science & Business Media, 1989, vol. 373.

[21] "Cámaras 3D y cámaras en 3a persona," https://geeks.ms/jbosch/2010/05/25/xna-cmaras-3d-y-cmaras-en-3a-persona/, accedido por última vez: 2017-11-26.

[22] J. R. Gilbert, C. Moler, and R. Schreiber, "Sparse matrices in matlab: Design and implementation," *SIAM Journal on Matrix Analysis and Applications*, vol. 13, no. 1, pp. 333–356, 1992.

[23] V. Laurmaa, M. Picasso, and G. Steiner, "An octree-based adaptive semi-lagrangian vof approach for simulating the displacement of free surfaces," *Computers and Fluids*, vol. 131, no. Supplement C, pp. 190 – 204, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0045793016300500

[24] G. M. Morton, *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company New York, 1966.

[25] "Curba de Lebesgue ," https://www.mathcurve.com/fractals/lebesgue/lbesgue.shtml, accedido por última vez: 2017-12-26.

[26] "NVIDIA® GPU Gems 2, Capítulo 6," https://developer.nvidia.com/gpugems/GPUGems2/gpugems2_chapter06.html, accedido por última vez: 2017-11-30.

[27] J. Elseberg, D. Borrmann, and A. Nüchter, "One billion points in the cloud – an octree for efficient processing of 3d laser scans," *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 76, no. Supplement C, pp. 76 – 88, 2013, terrestrial 3D modelling. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0924271612001888

[28] S. Laine and T. Karras, "Efficient sparse voxel octrees–analysis, extensions, and implementation," *NVIDIA Corporation*, vol. 2, 2010.

[29] "OpenGL tutorial - color picking," http://www.opengl-tutorial.org/miscellaneous/clicking-on-objects/picking-with-an-opengl-hack, accedido por última vez: 2017-10-01.

[30] M. Pharr and R. Fernando, *Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation*. Addison-Wesley Professional, 2005.

[31] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore gpus," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–10.

[32] R. S. Wright and M. Sweet, *OpenGL SuperBible with Cdrom*. Sams, 1999.

[33] "Shader Atomic," https://www.khronos.org/registry/OpenGL/extensions/NV/NV_shader_atomic_float.txt, accedido por última vez: 2017-12-26.

[34] "Mathworks® MATLAB ," https://www.mathworks.com/products/matlab.html, accedido por última vez: 2017-11-26.

[35] "History of the Cornell Box, Cornell University Program of Computer Graphics ," http://www.graphics.cornell.edu/online/box/history.html, accedido por última vez: 2017-11-25.

[36] J. E. Gentle, *Matrix algebra: theory, computations, and applications in statistics*. Springer Science & Business Media, 2007.

[37] S.-e. Yoon, E. Gobbetti, D. Kasik, and D. Manocha, "Real-time massive model rendering," *Synthesis Lectures on Computer Graphics and Animation*, vol. 2, no. 1, pp. 1–122, 2008.

[38] G. N. Walton, *Calculation of obstructed view factors by adaptive integration*. US Department of Commerce, Technology Administration, National Institute of Standards and Technology USA, 2002.

[39] P. Wonka, M. Wimmer, and D. Schmalstieg, "Visibility preprocessing with occluder fusion for urban walkthroughs," in *Rendering Techniques 2000*. Springer, 2000, pp. 71–82.

[40] F. Sillion and C. Puech, "A general two-pass method integrating specular and diffuse reflection," in *ACM SIGGRAPH Computer Graphics*, vol. 23, no. 3. ACM, 1989, pp. 335–344.

[41] "Rendering recursive portals with OpenGL." https://th0mas.nl/2013/05/19/rendering-recursive-portals-with-opengl/, accedido por última vez: 2017-12-07.