Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación
Centro de Computación Gráfica

Iluminación global en tiempo real basada en la voxelización de escenas con superficies difusas

Trabajo Especial de Grado
presentado ante la Ilustre
Universidad Central de Venezuela
Por el Bachiller
Francisco Sans
para optar al título de
Licenciado en Computación

Tutor: Prof. Esmitt Ramírez

Caracas, 24 Septiembre de 2012

Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación
Centro de Computación Gráfica

ACTA DEL VEREDICTO

Quienes suscriben, Miembros del Jurado designado por el Consejo de la Escuela de Computación para examinar el Trabajo Especial de Grado, presentado por el Bachiller Francisco Sans C.I.: 18.244.557, con el título *Iluminación global en tiempo real basada en la voxelización de escenas con superficies difusas*, a los fines de cumplir con el requisito legal para optar al título de Licenciado en Computación, dejan constancia de lo siguiente:

Leído el trabajo por cada uno de los Miembros del Jurado, se fijó el día 24 de Septiembre de 2012, a las 2:00 pm, para que su autor lo defendiera en forma pública, en el *Centro de Computación Gráfica*, lo cual se realizó mediante una exposición oral de su contenido, y luego respondió satisfactoriamente a las preguntas que les fueron formuladas por el Jurado, todo ello conforme a lo dispuesto en la Ley de Universidades y demás normativas vigentes de la Universidad Central de Venezuela. Finalizada la defensa pública del Trabajo Especial de Grado, el jurado decidió aprobarlo.

En fe de lo cual se levanta la presente acta, en Caracas a los 24 días del mes de Septiembre del año dos mil doce, dejándose también constancia de que actuó como Coordinador del Jurado el Profesor Esmitt Ramírez.

_____
Prof. Esmitt Ramírez

_____ _____
Prof. Jaime Parada Prof. Walter Hernández

Resumen

El cálculo de la iluminación global en tiempo real sigue siendo un problema no resuelto en el área de la computación gráfica. Actualmente se tiende al uso de discretizaciones y simplificaciones que permiten una aproximación realista en un tiempo aceptable a la solución de este problema. Una de las técnicas actuales que ofrece buenos resultados es la denominada Voxel-based Global Illumination. Esta técnica se basa en la discretización de una escena empleando un vóxel como estructura de datos en superficies difusas y así poder calcular rápidamente intersecciones rayo/vóxel, permitiendo la iluminación en tiempo real cercana a un punto. En este trabajo se realizan modificaciones a la propuesta original, así como detalles de implementación para realizar pruebas y comparar resultados.

Palabras claves: iluminación global, voxelización de escenas, GPU, tiempo real, superficies difusas.

Tabla de Contenidos

Lista de figuras

Lista de tablas

Lista de códigos

Introducción

La computación gráfica es una rama de las ciencias de la computación que se encarga del estudio, diseño y modificación de imágenes en la pantalla de un computador utilizando herramientas proporcionadas por diferentes ciencias como la física, la térmica, la geometría, la óptica, etc. Una de sus mayores áreas de desarrollo es la computación gráfica tridimensional, donde se trata de recrear escenas virtuales en 3D y alcanzar un alto realismo. El objetivo es crear escenas que sean visualmente cercanas a si estas escenas existieran en realidad. Estas escenas son utilizadas en la arquitectura, el cine, en los juegos de video, en la publicidad, en simulaciones de carro y vuelos de aviones, recreaciones de objetos que no han sido producidos, entre otros.

Un factor fundamental para infundir realismo en una escena virtual es la simulación de la iluminación. Esta simulación es una tarea computacionalmente costosa, ya que la obtención de un buen resultado visual puede requerir mucho tiempo de cálculo. Debido a ello, la simulación de la iluminación sigue siendo un problema no resuelto completamente. Diversos estudios han procurado desarrollar técnicas para obtener un buen resultado visual en el menor tiempo posible.

En el área de computación gráfica, se comenzó calculando la iluminación tomando sólo la información de la fuente de luz y la superficie iluminada. Estos algoritmos sólo toman en cuenta la información local a la superficie y son conocidos como algoritmos de iluminación local. Por ello, éstos no consideran todos los posibles caminos de los rayos luminosos, por lo que se obvian efectos que son creados por la luz, restando realismo a la escena. También existen los algoritmos de iluminación global, los cuales tratan de simular la iluminación de una escena, considerando la mayor cantidad posibles de efectos creados por la interacción de la luz. El uso de algoritmos de iluminación global permite crear escenas con un alto nivel de realismo. Sin embargo, debido a su ámbito global y a lo complejo del modelo matemático que expresa el calculo de la iluminación, estos algoritmos tardan mucho tiempo en calcularla.

Al implementar un algoritmo de iluminación global, se busca lograr un buen resultado visual en un tiempo aceptable. Sin embargo, es difícil satisfacer ambos objetivos. A tal fin, es imprescindible conocer la gama de técnicas existentes para poder seleccionar aquella que más se adecúe a la solución esperada. De acuerdo a esto, se hace necesario desarrollar una aplicación que permita calcular la iluminación en un tiempo aceptable, aprovechando al máximo los avances tecnológicos más recientes.

El presente documento aborda los principales métodos creados para simular la iluminación, y se centra la implementación de la técnica propuesta por Thiedemann et al. [4], con

el fin de poder aplicar un algoritmo de iluminación global y realizar pruebas sobre éste. Para introducir esta técnica, en el capítulo 1 se presentan los conceptos básicos relacionados a la iluminación, así como los primeros trabajos elaborados en esta área. La explicación detallada de la solución propuesta por Thiedemann et al. [4] se expone en el capítulo 2. Luego, en el capítulo 3 se describe la implementación de esta solución realizada para este trabajo especial de grado. Después, en el capítulo 4 se describen y analizan las pruebas realizadas sobre esta implementación. Finalmente, se plantean las conclusiones en el capítulo 5.

Capítulo 1

Iluminación global

Uno de los campos más importantes en el área de la computación gráfica es la creación de imágenes realistas. Para crear este tipo de imágenes, se tiene una escena virtual vista desde una cámara virtual, desde la cual se despliega una imagen que sea lo más parecido posible a si esta escena existiera en la realidad. Para ello, es necesario representar los objetos presentes en la escena y todos los fenómenos que se producen por la interacción de estos con la luz.

La generación de imágenes realistas tiene diversas utilidades: en la arquitectura (recreando los diseños antes de que se construyan), en el cine (con efectos especiales), en los juegos de video, en la publicidad, en simulaciones de carros y vuelos de aviones, en la recreación de objetos que todavía no han sido producidos, entre otros. Sin embargo, debido a las limitaciones de hardware, la generación de estas imágenes no siempre ha sido posible, ya que representar la interacción de la luz en una escena es muy costoso desde el punto de vista computacional [2].

En tiempos pasados, en el campo de la computación gráfica, la iluminación se limitaba al trazado de rayos y puntos. Los primeros algoritmos de iluminación consistían en asignar un color determinado según el ángulo de incidencia de la luz sobre la superficie. Posteriormente Gouraud [5] y Phong [6] introdujeron sus modelos de iluminación a la computación gráfica, pero éstos solo consideraban la información local de la superficie, sin tomar en cuenta las demás entidades presentes en la escena. A este tipo de técnicas que solo usan información local, se les conoce como técnicas de iluminación local.

Por otra parte, existen las denominadas técnicas de iluminación global, las cuales toman en cuenta la interacción de la luz con todas las entidades de la escena. El objetivo de los algoritmos de la iluminación global es el cálculo del estado de la distribución de la energía luminosa en una escena en un momento determinado. Debido al ámbito global de estos algoritmos, son computacionalmente costosos y su implementación para tiempos interactivos es difícil de alcanzar. En la figura 1.1 se puede apreciar una escena iluminada con iluminación global (abajo) y local (arriba). A la derecha de la figura se puede observar los caminos luminosos que son tomados en cuenta para cada uno de los tipos de iluminación.

Para profundizar en los trabajos actuales en este campo de estudio, es necesario conocer sus bases. Primero se explicará los diferentes efectos que son importantes recrear por medio de la iluminación global. Luego se expondrá la ecuación de despliegue que se utiliza para

modelar la interacción de la luz en una escena. Posteriormente, se estudiarán los primeros métodos existentes para recrear la iluminación global, para luego estudiar los trabajos actuales que permiten su cálculo en tiempo real.

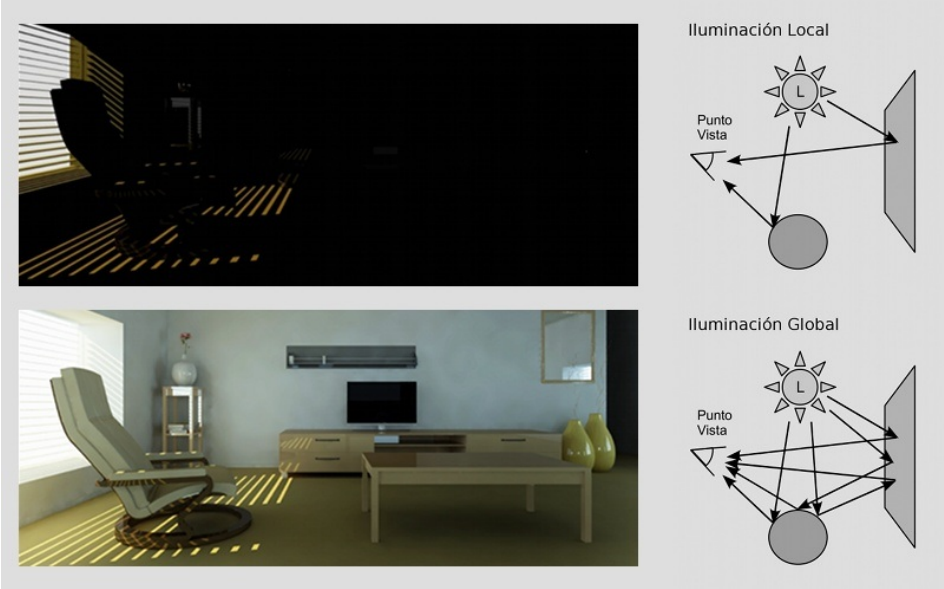


Figura 1.1: Imágenes desplegadas con iluminación local (arriba) y global (abajo).

1.1. Efectos creados por la interacción de la luz

Debido a que la iluminación global tiene como objetivo calcular la distribución de la energía de la luz en una escena, es importante aclarar los diversos efectos que produce la interacción de la luz con las superficies presentes en una escena. Los diferentes algoritmos propuestos en la actualidad tratan de recrear la mayoría de estos efectos, pero a mayor cantidad de estos, es mayor el procesamiento requerido para calcularlos.

En la figura 1.2 se muestran los efectos que generalmente se tratan de representar con un algoritmo de iluminación global. La luz directa se refiere al rayo de luz que viaja directamente desde una fuente de luz hasta una superficie, sin considerar otros posibles caminos. En cambio, la luz indirecta es la luz que ha sido rebotada, dispersada (*scattering*), reflejada o refractada desde una superficie antes de iluminar otro objeto. Esto se debe al cambio de la dirección de los rayos de la luz al incidir sobre una superficie. Los rebotes pueden causar efectos como el sangrado de color (*color bleeding*) o cáusticas.

Además, es importante recordar que la luz son partículas, por lo que un objeto podría obstaculizar su paso normal. Cuando un rayo luminoso es obstaculizado en gran medida se crean las sombras. Sin embargo, también se pueden encontrar elementos en el ambiente (llamados medios participantes), a través de los cuales el rayo luminoso es obstruido levemente y es distorsionado al pasar.

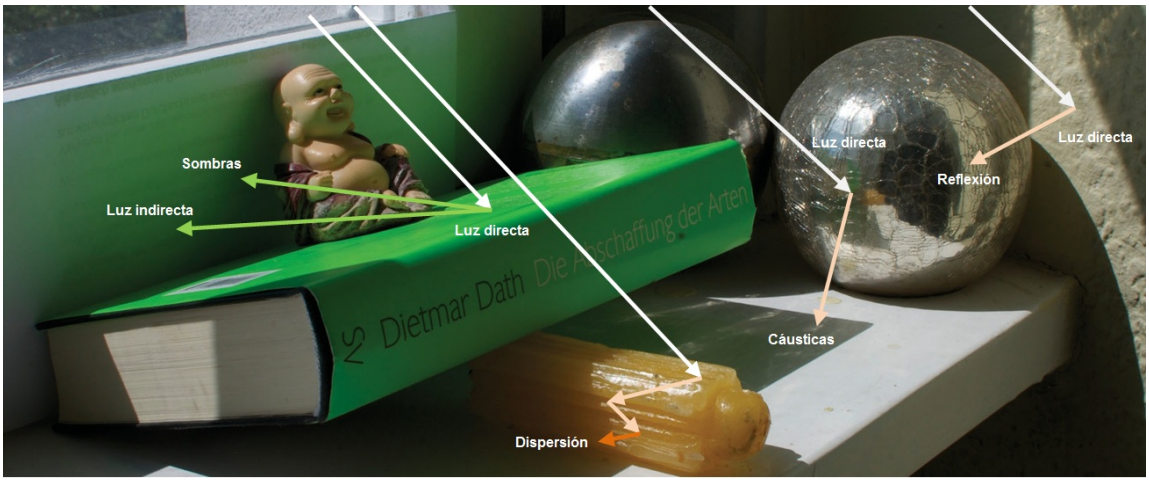


Figura 1.2: Fotografía de una escena con múltiples rebotes difusos y especulares, cáusticas y dispersión de la luz. Imagen tomada de [1].

1.2. Ecuación de despliegue

Como se mencionó anteriormente, el objetivo de la iluminación global es calcular la distribución de la energía de la luz en una escena para un momento dado. Debido a la alta complejidad de la interacción de la luz con los diferentes componentes de una escena, Kajiya [7] propone una ecuación que permite representar estas interacciones de tal forma que sea factible su despliegue. Esta ecuación describe el flujo de la radiación a través de un ambiente tridimensional. Philip Dutré et al. [2] plantean esta ecuación en términos radiométricos de la siguiente manera:

$$L(x \to \Theta) = L_e(x \to \Theta) + \int_{\Omega_x} fr(x, \Psi \to \Theta) L(x \leftarrow \Psi) \cos(N_x, \Psi) d_{\omega\Psi} \tag{1.1}$$

Esta ecuación esta expresada en términos de radiancia, la cual indica la cantidad de energía que es recibida en (o emitido desde) cierto punto en la superficie, por unidad de ángulo sólido y por unidad de área proyectada (watts/steradian $\times$ m^2). La radiancia saliente de un punto x en la dirección Θ ($L(x \to \Theta)$) es igual a la radiancia emitida en el punto x, saliendo en la dirección Θ ($L_e(x \to \Theta)$), más toda la radiancia proveniente del hemisferio iluminado que es incidente en el punto x y que es reflejada en la dirección Θ, ver figura 1.3. Esta es una integral sobre todo el hemisferio del punto x y la función fr representa la función de distribución de reflectancia bidireccional (*BRDF* por sus siglas en inglés). La *BRDF* se encarga de definir sobre la esfera visible la relación existente entre la radiación incidente en la dirección Ψ y la radiancia reflejada en la dirección Θ. Todas las luces incidentes en el ángulo Θ son sumadas en proporción al *BRDF* y al coseno del ángulo incidente, en [2] se explica con mayor detalle este proceso.

La ecuación de despliegue es una integral llamada la ecuación de Fredholm de segundo tipo, debido a su forma: la radiancia, que es el valor desconocido, aparece tanto en el lado

derecho como el izquierdo de la ecuación.

Sin embargo, esta función es expresada en términos más simples por Kajiya [7] de la siguiente manera:

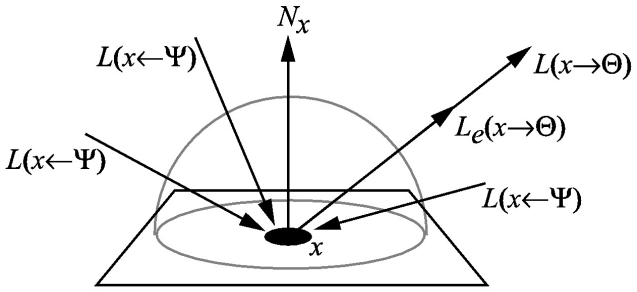


Figura 1.3: La radiancia emitida desde el punto x es igual a la radiancia emitida $L(x \rightarrow \Theta)$ más la radiancia reflejada $L(x \leftarrow \Psi)$. Imagen tomada de [2].

$$I(x, x') = g(x, x')[e(x, x') + \int_S p(x, x', x'')I(x', x'')d_{x''}] \tag{1.2}$$

Donde:
$I(x, x')$ es la intensidad de luz que pasa desde el punto x' al punto x.
$g(x, x')$ es un término geométrico.
$e(x, x')$ es la intensidad de luz emitida desde x' a x.
$p(x, x', x'')$ es la intensidad de luz dispersada desde x'' a x pasando por el punto x'.

Esta ecuación representa el balance de la energía transmitida de un punto de una superficie a otro punto. Así, se plantea que la intensidad de luz transportada desde un punto de una superficie a otro es la suma de la luz emitida y el total de luz dispersa hacia x desde todos los puntos de las superficies. Por lo mismo, pasa de tener una integral sobre el hemisferio a tener una integral sobre $S = \bigcup S_i$, la unión de todas las superficies.

A pesar de lo complejo de la ecuación, la misma no toma en cuenta todos los efectos creados por la luz. Entre otras cosas, asume que las superficies están en el vacío, donde carecen de un medio participante; no toma en cuenta el tiempo que tarda la luz en ser transportada, y no considera efectos como fosforescencia o fluorescencia. Sin embargo, en su trabajo, Kajiya [7] propone extensiones para abarcar estos efectos. A pesar de estas carencias, la ecuación representa una generalización para la mayor parte de los algoritmos existentes que se usan para el cálculo de la iluminación global.

La resolución de esta ecuación representa un gran reto, aún con los computadores actuales, especialmente en aplicaciones en tiempo real. Por lo tanto, ha sido un gran campo de estudio en los últimos años y se ha llegado a diversos métodos que intentan resolverla. Todos los algoritmos utilizan como base de sus cálculos la ecuación propuesta Kajiya [7].

1.3. Función de distribución de reflectancia bidireccional

La energía emitida hacia una escena interactúa con diferentes objetos, siendo transmitida o reflejada en su superficie. Parte de la energía también puede ser absorbida o disipada en forma de calor. La propiedad de reflectancia de la superficie de un objeto determina la manera en que la luz interactúa con el mismo y cambia su apariencia.

De forma general, un rayo luminoso puede incidir en un punto p con una dirección Ψ y ser reflejado en un punto q en una dirección Θ. La función que define la relación entre la radiancia incidente y la reflejada es llamada la función de distribución de reflectancia bidireccional con dispersión en la superficie (*BSSRDF*). Sin embargo, los algoritmos de iluminación global generalmente no consideran la dispersión de las superficies, ya que resulta muy compleja de modelar. Por lo tanto, asumen que el rayo reflejado siempre parte desde el mismo punto de incidencia del rayo luminoso.

Por ello, las propiedades de reflectancia de una superficie son descritas con la función de distribución de reflectancia bidireccional (*BRDF*). La *BRDF* en un punto x es definida como la proporción entre el diferencial de la radiancia reflejada en una dirección saliente Θ ($dL(x \rightarrow \Theta)$) y el diferencial de la irradiancia incidente a través del diferencial por unidad de ángulo sólido $d\omega_\Psi$ ($dE(x \leftarrow \Psi)$). La irradiancia es el total de energía incidente en una superficie, por unidad de área de la superficie. De esta manera, la *BRDF* se puede expresar como:

$$fr(x, \Psi \rightarrow \Theta) = \frac{dL(x \rightarrow \Theta)}{dE(x \leftarrow \Psi)} \tag{1.3}$$

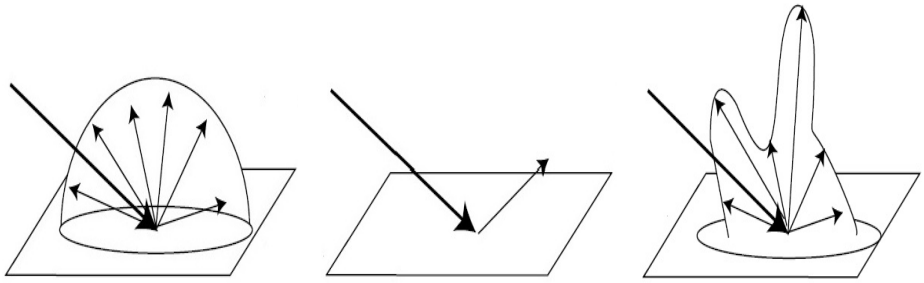
Los algoritmos de iluminación global generalmente usan modelos empíricos para representar la *BRDF*. Dependiendo de la naturaleza de la *BRDF*, el material de un objeto puede parecer una superficie difusa, especular o brillosa. La *BRDF* puede simplificarse dependiendo de la superficie a modelar.

Superficies difusas

Son superficies que reflejan la luz de manera uniforme en todas las direcciones posibles, como puede observarse en la figura 1.4(a). Por lo tanto, su valor de *BRDF* es constante para todos los valores de Θ y Ψ. Para un observador, un punto en una superficie difusa se ve igual desde todas las direcciones posibles. De esta manera, la *BRDF* de una superficie difusa ideal cumple la siguiente ecuación:

$$fr(x, \Psi \rightarrow \Theta) = \frac{\rho_d}{\pi} \tag{1.4}$$

La reflectancia ρ_d representa la cantidad de energía incidente que es reflejada de la superficie. Este valor varía entre 0 y 1.



(a) Difusas perfectas (b) Especulares perfectas (c) Brillosas

Figura 1.4: Tipos de superficie. Imagen tomada de [2].

Superficies especulares

Las superficies especulares perfectas son aquellas que reflejan y refractan la luz en una dirección específica. En la figura 1.4(b) se encuentra un ejemplo de estas superficies.

El rayo reflejado se puede conseguir de la siguiente manera:

$$R = 2(N \cdot \Psi)N - \Psi \tag{1.5}$$

Donde Ψ es la dirección de la luz incidente, N es la normal de la superficie y R es el rayo reflejado. Esta ecuación es derivada de la ley de reflexión, en la cual se establece que la normal de la superficie posee el mismo ángulo con la dirección del rayo incidente y la dirección del rayo reflejado.

Por otra parte, el rayo refractado es obtenido con la ley de Snell. En esta ley se considera la siguiente igualdad:

$$\eta_1 \sin\theta_1 = \eta_2 \sin\theta_2 \tag{1.6}$$

Donde θ_1 y θ_2 son los ángulos entre el rayo incidente y el transmitido con la normal de la superficie, y η_1 y η_2 son los índices refractivos del medio incidente y el saliente respectivamente. Con el uso de esta igualdad, el rayo refractado T es calculado de la siguiente manera:

$$T = -\frac{\eta_1}{\eta_2}\Psi + N(\frac{\eta_1}{\eta_2}(N \cdot \Psi) - \sqrt{1 - \frac{{\eta_1}^2}{\eta_2}(1 - (N \cdot \Psi)^2)}) \tag{1.7}$$

Estas ecuaciones para el cálculo de los rayos son solo válidas para superficies especulares perfectas, las cuales no son fáciles de encontrar en el mundo real.

Superficies brillosas

Generalmente las superficies en la naturaleza no son puramente especulares ni puramente difusas, sino que exhiben una combinación de los dos comportamientos, como se muestra en la figura 1.4(c). Estas superficies son llamadas superficies brillosas y son difíciles de modelar computacionalmente.

1.4. Algoritmos de iluminación global

Haciendo uso de la ecuación de despliegue presentada anteriormente, diferentes algoritmos se han propuesto para tratar de resolver el problema de la iluminación global. En esta sección se presentarán los primeros trabajos que buscan una solución a este problema.

Trazado de rayos

El trazado de rayos *(ray tracing)* fue uno de los primeros esfuerzos para lograr la iluminación global, introducida por Whitted [8]. En su trabajo se describe una extensión al algoritmo de lanzado de rayos (*ray casting*) para poder determinar la visibilidad de las superficies e incluir el efecto de refracción y reflección de superficies especulares perfectas.

La técnica original presenta una alta recursividad, por lo que termina siendo ineficiente. Sin embargo, se producen imágenes realistas, por lo que se han realizado grandes esfuerzos para acelerarlo. Entre las diversas variantes propuestas encontramos: el trazado de rayos distribuido propuesto por Cook et al. [9], el trabajo de Kajiya [7] sobre el trazado de caminos, el trazado de rayos inverso de Arvo [10], el trazado de rayos bidireccional [11] y el transporte de luz de metrópolis [12] propuestos por Veach y Guibas, entre otros.

Radiosidad

La técnica de radiosidad *(radiosity)* fue propuesta por Goral et al. [13] para resolver la interacción de la luz para superficies difusas. Por lo tanto, solo resuelve parcialmente el problema de la iluminación global. Su idea principal es calcular el promedio de radiosidad B_i en cada elemento de superficie o parche i para una escena tridimensional.

Esta técnica presenta la ventaja de que una vez realizado el cálculo de la iluminación, es posible el despliegue de manera interactiva, ya que no depende del punto de visión. Sin embargo, se limita a escenas y luces estáticas. Además, solo representa la interacción con superficies difusas y no da el aporte especular, restando así realismo a la escena.

Caché de irradiancia

Debido a que el *ray tracing* original es lento, Ward et al. [14] presentan una mejora para el cálculo de la iluminación difusa indirecta de las escenas, conocida como caché de irradiancia *(irradiance caching)*. La idea principal consiste en mantener almacenada la irradiación de diferentes puntos de la escena calculados anteriormente, basándose en que la radiación en superficies difusas varía de manera suave.

Mapeado de fotones

El mapeado de fotones *(photon mapping)* fue propuesto por Jensen [15–17]. Consiste en trazar rayos desde la luz y desde el punto de vista. Esta técnica hace uso del almacenamiento de la radiación calculada a través de los rayos emitidos desde la luz. En la primera pasada se disparan fotones desde la luz y la cantidad de radiancia que emiten es almacenada en una

estructura llamada el mapa de fotones [18]. En la segunda pasada se despliega la imagen utilizando la información almacenada.

En la figura 1.5 se observa el uso del mapeado de fotones para el cálculo de la iluminación. El procesos de toma de muestras en el mapa de fotones se realiza con el segundo rebote del rayo. Sin embargo, se puede crear un mapa de cáusticas tomando muestras en el primer rebote. Diferentes mapas conllevan diferentes formas de tomar muestras.

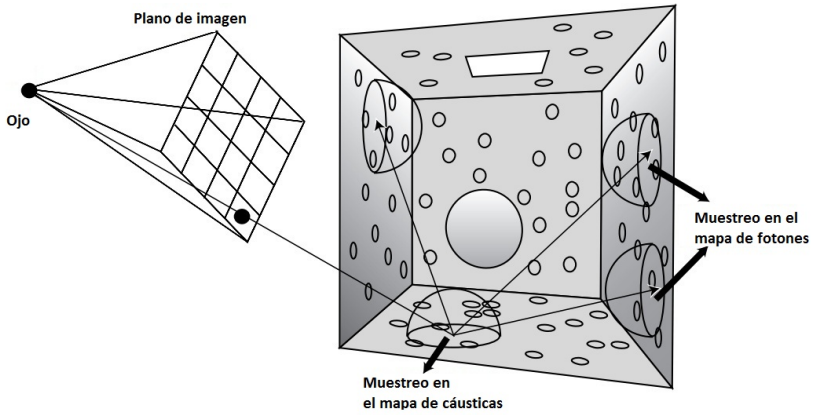


Figura 1.5: Cálculo de la iluminación usando el *photon mapping*. Imagen en [2].

Radiosity **instantáneo**

En 1997, Keller [19] propone la técnica de *radiosity* instantáneo *(instant radiosity)*, como un algoritmo de dos pasadas. La idea consiste en reemplazar la luz difusa indirecta por luz difusa directa proveniente desde un punto de luz llamado punto de luz virtual *(virtual light points, VPL)*. Para ello, se trazan rayos desde la fuente de luz y se colocan luces virtuales en el punto de intersección. Eventualmente se puede permitir la dispersión especular para colocar más luces virtuales a partir de la primera luz virtual, como se observa en la figura 1.6.

(a) (b)

Figura 1.6: Despliegue del modelo de Marko Dabrovic de la Catedral de Sibenik [3], donde 1.6(a) es inyectada con un conjunto de luces virtuales y 1.6(b) la catedral es iluminada con *instant radiosity*.

Cortes-de-luz

Walter et. al [20, 21] proponen la técnica de Cortes-de-luz *(Lightcuts)*, para el despliegue de efectos complejos como los son: una gran cantidad de luces, desenfoque por movimiento, desenfoque por distancia y presencia de medios participantes.

En [20] se propone representar las luces a través de un árbol, de tal manera que estas puedan ser unidas en *clusters*. De esta forma, quedarán las fuentes de luz en las hojas y en los nodos internos quedarán representaciones promediadas de las mismas. Utilizando este árbol, no se necesita evaluar todas las luces, sino un corte del árbol tal que no se exceda de un error preestablecido.

Estas propuestas solo resuelven parcialmente el problema de la iluminación global. Si se realizan demasiadas simplificaciones, se obtiene una solución poco realista en un tiempo aceptable; si no se realizan simplificaciones, se obtiene una solución realista, pero en un tiempo excesivo. Actualmente, con los avances en computación, se han realizado propuestas para el cálculo de la iluminación tratando de mejorar los trabajos pasados y obteniendo mejores resultados. Generalmente se trata de obtener una solución visualmente aceptable en tiempos interactivos. En la siguiente sección se expondrán las investigaciones actuales en el cálculo de la iluminación global.

1.5. Iluminación global en tiempo real

A pesar de los grandes avances en computación en los últimos años, el cálculo de la iluminación global en tiempo real continua siendo un gran reto. Sin embargo, debido a su gran importancia, se han realizado diversas investigaciones que tratan de obtener una solución visualmente aceptable, en el menor tiempo posible. Esto se debe a que actualmente se reconoce que en la mayoría de los casos no es necesario calcular una solución exacta para crear un efecto convincente. Además, la mayor parte de las investigaciones actuales tienden al uso de la GPU.

La unidad de procesamiento gráfico o GPU (acrónimo del inglés *graphics processing unit*) es un coprocesador dedicado al procesamiento de gráficos u operaciones de coma flotante, para aligerar la carga de trabajo del procesador central como en los juegos de video o aplicaciones 3D interactivas. De esta forma, mientras gran parte de lo relacionado con los gráficos se procesa en la GPU, la unidad central de procesamiento (CPU) puede dedicarse a otro tipo de cálculos (como la inteligencia artificial o los cálculos mecánicos en el caso de los juegos de video).

Según Kaplayan y Dachsbacher [22] las técnicas actuales de iluminación global en tiempo real pueden clasificarse de la siguiente manera:

Métodos clásicos

Dadas las mejoras actuales de hardware, métodos clásicos como *raytracing* [23] y *radiosity* [24], han sido mejorados para poder alcanzar así desempeños interactivos. Sin embargo,

estas mejoras representan grandes restricciones o el uso de grandes cantidades de poder de computación, por lo que no son la mejor opción para aplicaciones en tiempo real.

Iluminación precalculada

Esta es una de las técnicas más usadas especialmente en video juegos, como *Halo 3* [25] o *Dragon Age II* [26]. Esta consiste en realizar el precálculo de la iluminación, incluyendo información de visibilidad, lo que conlleva la restricción del uso de escenas estáticas o semi-estáticas [27] [28].

Métodos en espacio de imagen

El espacio de imagen es un sistema de coordenadas 2D que corresponde a coordenadas físicas de la pantalla. Estas técnicas hacen uso de la GPU para el cálculo de la iluminación en el espacio de imagen [29] [30], y sólo trabajan en éste espacio, por lo que ignoran objetos o luces que se encuentran fuera de éste espacio. Así, crean una gran cantidad de artefactos y es necesario un post-procesamiento para disminuirlos.

Dachsbacher y Stamminger [31] proponen una técnica eficiente basada en el uso de *reflective shadow maps (RSM)*. Los *RSM* utilizan la idea de los mapas de sombras para capturar puntos iluminados de forma directa, y así convertir cada píxel en pequeños puntos de luz. Otra de las técnicas con más auge a sido el cálculo de oclusión ambiental en espacio de imagen [32] [33].

Métodos basados en *instant radiosity*

Haciendo uso del *instant radiosity* [19], es posible calcular la iluminación de una escena a través de *VPL*. Con el uso del *RSM* [31], se pueden calcular *VPL* para el primer rebote de luz indirecta, por lo que esta técnica ha cobrado gran interés en el cálculo de la iluminación global durante los últimos años [34] [35] [36] [37]. Una de las principales desventajas de estas técnicas es que es necesario un gran número de *VPL* para solventar los artefactos en la imagen final.

Métodos basado en modelos discretos

Consiste en la discretización de los términos de radiación de una escena, de tal manera que estos puedan ser guardados en un mallado tridimensional que la represente. La luz interactúa con elementos vecinos del volumen, reduciendo el cálculo por interacción a un ámbito local [38] [22] [39] [40]. Son principalmente empleados para modelar diversos medios participantes.

Métodos de aproximaciones geométricas

Uno de los mayores aspectos a tomar en cuenta cuando se quiere calcular la iluminación global, es la complejidad y el dinamismo de la escena. El uso de una descripción poligonal de

una escena generalmente consume mucho tiempo. Por lo tanto, su discretización en elementos más simples puede agilizar el cálculo del transporte de la luz.

Esta vertiente se basa en la idea de calcular la iluminación global en objetos atómicos precalculados, usualmente usando discos, esferas [41] o *surfels* [42] [43]. Debido al incremento del poder de procesamiento de la GPU, estás técnicas han cobrado importancia. Sin embargo, la representación de las escenas en estos elementos atómicos puede llegar a ser costosa y difícil de calcular. Una de las vertientes actuales es el uso de la discretización de escenas empleando vóxeles para el cálculo de la iluminación global, ya sea con el uso de vóxeles para la representación de la geometría de la escena [44] [45] [46] o de *grids* para la dispersión de la luz en el ambiente [38] [22] [45].

Uno de los trabajos más recientes en esta área es el realizado por Thiedemann et al. [4], donde se introduce una nueva técnica basada en el uso de texturas atlas para obtener la voxelización de la escena, además de proponerse pruebas de intersección rayo/vóxel. Estas pruebas permiten el cálculo de la iluminación de un solo rebote de la luz en un área cercana en tiempo real.

Capítulo 2

Iluminación global basada en vóxeles

En este capítulo, se expondrán las diferentes ideas expuestas en el trabajo de Thiedemann et al. [4] y como estas ideas son implementadas en este trabajo especial de grado. A continuación, se expondrá el planteamiento del problema y se presentará un esquema del algoritmo propuesto, resaltando cada uno de los pasos necesarios para obtener la iluminación global.

2.1. Planteamiento del problema

Como se mencionó en el capítulo anterior, el objetivo de los algoritmos de la iluminación global es el cálculo del estado de la distribución de la energía luminosa en una escena en un momento determinado. Dada la complejidad de la interacción de la luz con la escena, es necesario realizar simplificaciones para obtener una aproximación de la iluminación global en tiempos interactivos.

En este trabajo se hace necesario realizar una aplicación que permita el cálculo de la iluminación global en tiempos interactivos. Para ello, se plantea el uso de la GPU para la aceleración del proceso completo; y la técnica de iluminación basada en vóxeles. Esta técnica obtiene resultados visuales de alta calidad así como tiempos interactivos adecuados para aplicaciones en tiempo real.

Una de las técnicas más recientes para el cálculo de la iluminación global es la propuesta por Thiedemann et al. [4]. En esta propuesta, se utiliza la discretización de la escena por medio de la voxelización, pasando de una representación poligonal a vóxeles. Para ello, Thiedemann et al. proponen una nueva técnica de voxelizado a través de texturas atlas. Una textura atlas es una representación plana de un objeto, que generalmente es utilizada para su texturizado. El cálculo de la luz indirecta se realiza haciendo uso de los vóxeles, utilizando una técnica parecida a las técnicas del cálculo de la iluminación en espacio de imagen. Para simplificar los cálculos solo se considera un rebote de la luz indirecta, ya que según Tabellion et al. [47] un solo rebote es suficiente para introducir veracidad visual en la imagen. La escena es proyectada desde el punto de vista de la cámara y para cada uno de los píxeles de la imagen un rayo es lanzado hacia la escena. Para calcular la luz indirecta en el primer punto de intersección, se lanzan N rayos a partir de ese punto hacia la escena y se calcula su

intersección utilizando la representación voxelizada de la escena. De esta manera, la solución de la iluminación global carece de los defectos presentes en la técnicas de espacio de imagen, ya que tanto oclusores como emisores de luz que no son vistos en el espacio de imagen, son considerados a través de los vóxeles. Además, Thiedemann et al. [4] proponen una técnica para acelerar el cálculo de la intersección rayo/vóxel. En esta técnica, mientras más corto se mantenga el rayo, más rápido será el cálculo de la intersección. Por ello, manteniendo la longitud del rayo pequeña se puede obtener una solución a la iluminación global en tiempos interactivos.

Utilizando las ideas expuestas por Thiedemann et al. [4], se realizó una implementación de un algoritmo de iluminación global, en la cual se permite interactuar con la escena y modificar los diferentes parámetros del algoritmo. Con ello, se pretende realizar pruebas sobre el mismo para poder hacer mediciones sobre los resultados obtenidos.

2.1.1. Esquema propuesto

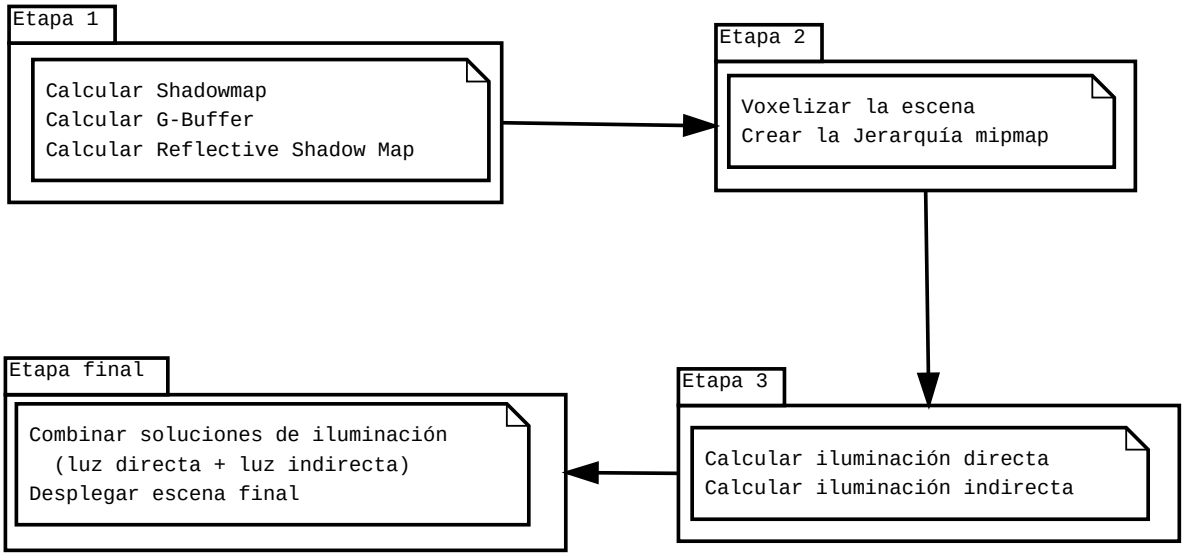


Figura 2.1: Esquema de las etapas propuestas para el cálculo de la iluminación global.

Para lograr el cálculo de la iluminación global basándose en el trabajo de Thiedemann et al. [4], se siguen las etapas mostradas en la imagen 2.1. La solución final se obtiene de combinar la contribución de la luz directa y la luz indirecta. Para la obtención de la luz directa es necesario calcular la contribución directa de cada una de las luces. Además, para crear un efecto más realista se agregan las sombras que cada una de las luces crea sobre la escena, por lo que es necesario el uso del *shadow maps*. En el caso de la luz indirecta es un proceso más complejo. Primeramente se necesita voxelizar la escena utilizando alguna de las técnicas propuestas: *slicemap* o texturas atlas. Una vez obtenida la escena voxelizada se puede crear una jerarquía *mipmap*, la cual es una estructura de datos con la que se aceleran los cálculos. Al mismo tiempo, se obtiene un *reflective shadow map* para cada una de las

luces y un *G-Buffer* para la escena, el cual contiene un conjunto de *buffers* intermedios que almacenan información sobre la escena para ser usada posteriormente. Una vez obtenida la escena voxelizada, el *G-Buffer* y los *RSM* se procede al cálculo de la iluminación indirecta de la manera propuesta por Thiedemann et al. [4] y combinandola con la iluminación directa se obtiene la iluminación global.

Cada uno de estos procesos son explicados con mayor detalle a lo largo de este capítulo. Primeramente se expondrá lo que es la voxelización, explicando las técnicas a utilizar en este trabajo, la forma de almacenar una escena voxelizada y como crear una jerarquía *mipmap* a partir de esta representación. Luego se detallará el método de intersección rayo/voxel propuesto por Thiedemann et al. [4] y se mostrará como se utiliza para obtener la iluminación indirecta. Por último, se explicará como se obtiene la iluminación directa y como se combina con la iluminación indirecta para crear la iluminación global.

2.2. Voxelización

Las escenas se diseñan con mayor complejidad geométrica cada vez. Mientras más complejidad exista, la interacción con una escena y la realización de cálculos sobre la misma, se vuelve un proceso más costoso. Por ello, es de gran interés encontrar una representación alternativa de la escena, como lo es el uso de vóxeles, lo cual requiere utilizar la voxelización.

La voxelización de escenas es el proceso mediante el cual se transforma la representación de una escena compuesta de entidades geométricas, en un *grid* tridimensional conformado por vóxeles. Cada celda del *grid* codifica información específica acerca de la escena. En la figura 2.2 podemos observar un objeto con su representación poligonal 2.2(a) y su representación con vóxeles 2.2(b).

(a) (b)

Figura 2.2: Objeto con 2.2(a) su representación poligonal y 2.2(b) vóxeles.

Dependiendo de la información que necesite ser almacenada, se tienen varios tipos de

voxelización. La binaria, que solo se encarga de codificar la presencia o la ausencia de geometría en el vóxel, y la multivalor, que puede almacenar más información como las normales o coeficientes de materiales. Por otro lado, la voxelización también puede dividirse en voxelización de superficies, si solo toma en cuenta las superficies de los objetos; o en voxelización sólida, si también toma en cuenta su interior.

Por mucho tiempo la voxelización fue una tarea costosa, generalmente realizada como un preprocesamiento. Sin embargo, la utilización del hardware gráfico moderno, ha permitido realizar la voxelización de escenas en tiempos interactivos. La idea es utilizar la funcionalidad de rasterización[1] presente en el hardware gráfico para acelerar la voxelización. Esto se debe a que la rasterización y la voxelización son procesos similares. En el proceso estándar de rasterización, los triángulos son convertidos en un *frame buffer* 2D. Solo los fragmentos más cercanos son almacenados en el *frame buffer*. Sin embargo, la voxelización es un proceso de rasterización 3D, por lo que es requerido un espacio discreto de vóxeles.

Haciendo uso de la GPU, existen diferentes maneras de voxelizar una escena. El primer enfoque fue propuesto por Fang et al. [48], con los algoritmos de cortes *(slicing)*. Su algoritmo consiste en usar diversas configuraciones del *z-near* y del *z-far* de una cámara ortogonal para capturar diferentes cortes de la geometría de una escena. Usando la idea de *slicing*, Crane et al. [49] proponen una técnica donde intersectan todos los triángulos de la escena con cada uno de los planos del volumen, para así llenar todas las capas. Posteriormente, Passalis et al. [50] y Li et al. [51] propusieron un método llamado pelado-por-capas (*Depth-peeling*). En esta técnica la escena se va pelando capa por capa, y se va almacenando cada una de las capas en el volumen. Por último, están las técnicas que utilizan rasterizadores personalizados. Fueron propuestas por Dong et al. [52] y Eisemann et al. [53], en donde se utiliza la profundidad de los fragmentos para codificar el volumen.

Actualmente, se poseen diversas técnicas que permiten voxelizar una escena en muy poco tiempo. En este trabajo, interesa el uso la voxelización de superficies, ya que solo se considera la interacción de la luz con la superficie de los objetos. Además, para simplificar los cálculos, se utiliza una voxelización binaria. Se implementan dos técnicas de voxelizado: mapa de cortes (*Slicemap*) [53] y la voxelización utilizando texturas atlas [4].

2.2.1. Basada en *Slicemap*

Es una técnica propuesta por Eiseman y Décoret [53], en donde se utilizan los *shaders* [2] para crear el volumen. Se basa en el uso del hardware gráfico, ya que la vista de despliegue puede definir el *grid* del volumen implícitamente, y porque para cada fragmento se encuentra la intersección del mismo con el *grid*.

[1] La rasterización es el proceso por el cual una imagen descrita en un formato gráfico vectorial se convierte en un conjunto de píxeles o puntos para ser desplegados en un medio de salida digital, como una pantalla de computadora, una impresora electrónica o una imagen de mapa de bits

[2] Los *shaders* son programas que reemplazan ciertas etapas del *pipeline* gráfico de la GPU. Los programas que más se utilizan son el de vértices (*vertex shader*), el de fragmentos (*fragment shader*) y el de geometrías (*geometry shader*).

Codificación del volumen La idea más básica para codificar un volumen con vóxeles binarios es el uso de una matriz tridimensional, donde cada posición indique la presencia de geometría o no. Sin embargo, es mucho más eficiente el uso de una textura 2D para el mismo propósito. El alto y el ancho de la textura definen el alto y el ancho del volumen, y la profundidad es codificada con los bits de los canales RGBA de la textura.

La manera más eficiente de almacenar un volumen en una textura 2D, es haciendo uso de *shaders* y colocando el volumen como objetivo de despliegue. Para ello, se define una cámara de la escena de tal forma que el *frustum* [3] envuelva el área a ser voxelizada. Se puede utilizar tanto una cámara perspectiva como una ortogonal. El *viewport* a utilizar debe corresponder con el alto y el ancho del volumen ($w \times h$). Para la profundidad es posible utilizar una textura de tipo $GL\_RGBA32UI$, las cuales utilizan 32 bits por canal, obteniendo en cada *texel* una columna de 128 celdas.

Una vez escogida la cámara y el *viewport*, se obtiene un volumen de dimensiones $w \times h \times 128$ donde cada bit de la textura 2D representa información, como se observa en la figura 2.3. Cada bit será utilizado para codificar si una primitiva intersecta esa celda o no.

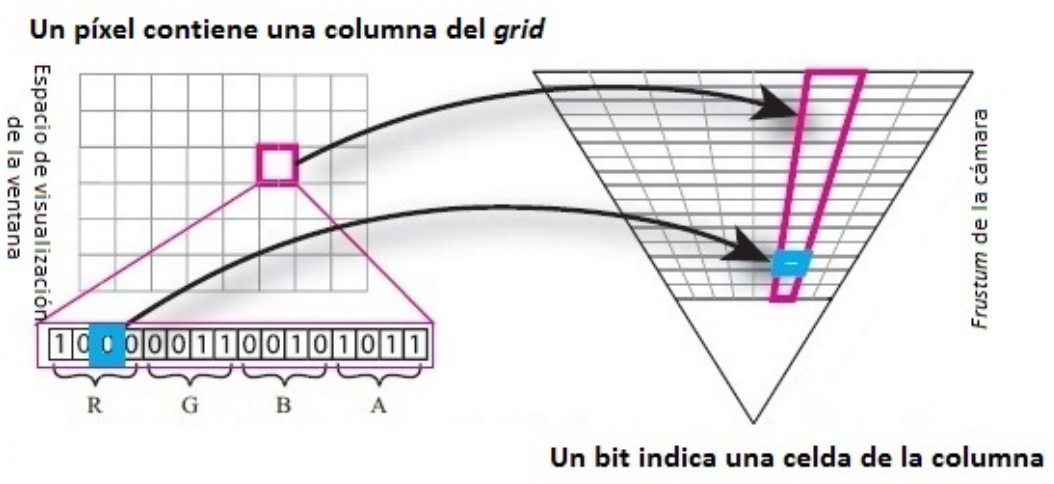


Figura 2.3: Representación de un volumen con una textura 2D.

Creación del volumen Para generar el volumen se despliega la escena en una textura utilizando un *fragment shader*. La textura necesita estar inicializada en 0 y se requiere colocar la cámara para que envuelva la escena. Además, se debe preparar el *pipeline* gráfico. Una textura debe ser colocada como objetivo de despliegue y se debe colocar la operación de OR lógico para la mezcla de colores:

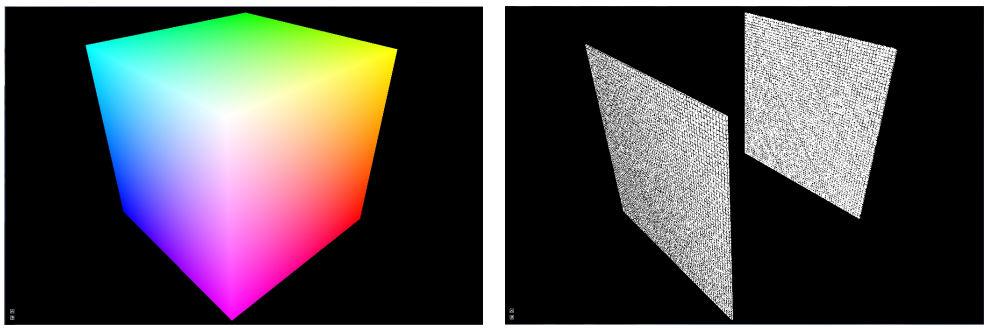
glLogicOp(GL_OR);
glEnable(GL_COLOR_LOGIC_OP);

[3] *Frustum* es una porción de una figura geométrica (e.g. pirámide) comprendida entre dos planos paralelos. Las aplicaciones de despliegue gráfico utilizan esta figura para calcular la parte del escenario virtual que ve la cámara virtual.

Para cada primitiva es necesario determinar con cuales vóxeles intersecta y colocar el bit de la textura correspondiente a esa celda en 1. El rasterizado de la primitiva creará un fragmento por cada columna que se intersecte. En el *fragment shader* se tiene la información de la profundidad y con el mismo se calcula una máscara de bits, en donde todos los bits son 0 excepto el correspondiente al corte intersectado, el cual se colocará en 1. El valor de profundidad está mapeado en los valores [0,1]. Este valor es usado para realizar una búsqueda en una textura 1D, la cual contendrá las máscaras de bits correspondientes para cada uno de los 128 valores de profundidad posible. Usar esta textura es útil, ya que los valores de cada una de las máscaras son independientes del volumen y pueden ser calculados una sola vez. Además, el acceso a una textura es más rápido que crear la máscara de bits.

El valor de profundidad del vértice que se encuentra disponible en el *fragment shader* no representa una distribución uniforme en coordenadas de mundo. Por lo tanto, se realiza una transformación que toma en consideración la distancia real de los vértices a la cámara. Estas distancias son transferidas al *fragment shader* como coordenadas de textura, las cuales serán interpoladas, obteniendo el valor correcto de profundidad z en el rango $[-zn, zf]$. Este valor es mapeado de manera uniforme usando:

$$z' = \frac{z + zn}{zn + zf} \tag{2.1}$$



(a) (b)

Figura 2.4: Ejemplo de una escena 2.4(a) mal voxelizada con *Slicemap* debido a superficies perpendiculares a la cámara de voxelizado 2.4(b).

Esta técnica solo necesita desplegar la escena una vez por ciclo, por lo que se realiza en tiempo despreciable. Sin embargo, como depende del proceso de rasterizado, superficies que son perpendiculares a la cámara de voxelizado solo crean un fragmento por columna, por lo que se podría perder mucha información. Eso se puede observar en la figura 2.4. Para solventar este problema Thiedemann et al. [4] proponen una técnica de voxelizado por medio de texturas atlas, la cual se explicará a continuación.

2.2.2. Basada en el uso de texturas atlas

Esta técnica fue propuesta por Thiedemann et al. [4] como un enfoque novedoso para la voxelización de escenas. Dicho enfoque se centra en el almacenamiento de superficies de objetos, pudiendo ser de forma binaria o multivalor. En la forma binaria se utilizan los bits de los canales RGBA de una textura 2D para la codificación de los vóxeles, de la misma manera en que se realiza en *slicemap*. Este método funciona para cualquier tipo de modelos, ya sean estáticos o dinámicos, siempre y cuando puedan ser almacenados en una textura atlas y que sea posible generar una correspondencia apropiada.

Una textura atlas es una imagen que contiene muchas imágenes más pequeñas, cada una de las cuales es una textura de una parte de un modelo, o bien texturas de modelos diferentes. Estas pequeñas texturas pueden ser desplegadas modificando las coordenadas UV del objeto en el atlas, básicamente indicándole que parte de la imagen corresponde a su textura. En la figura 2.5 se observa como una textura atlas (parte 2.5(b)) es mapeada a un objeto para su despliegue (parte 2.5(a)).

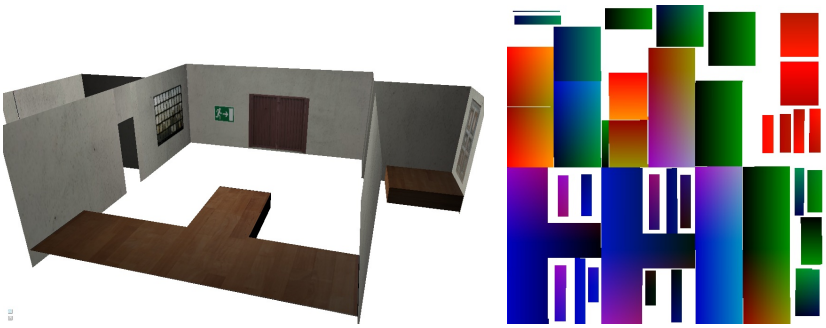
(a) (b)

Figura 2.5: Ejemplo de una textura atlas (2.5(b)) correspondiente a un modelo 3D (2.5(a)).

Para realizar la voxelización, primero se necesita desplegar la escena en una o varias texturas atlas por cada objeto. Para ello, se coloca la textura como objetivo de despliegue y se pasan las coordenadas de textura atlas como coordenadas de textura al *vertex shader*. Allí se normalizan estas coordenadas al espacio $[-1, 1]^2$ y se utilizan como la posición del vértice. Además, el *vertex shader* pasa la posición en el espacio de mundo al *fragment shader*, el cual solo se encargará de desplegar esta posición en la textura atlas como un color. En la figura 2.6 se observa como las coordenadas de mundo de un objeto (parte 2.6(a)) son almacenadas en una textura atlas (parte 2.6(b)).

Generalmente este mapeado dejará huecos, por lo que se necesita marcar esas entradas como inválidas. Una manera de hacerlo es crear un *vertex buffer object* con cada una de las posiciones válidas de la textura, el cual solo necesita ser creado una vez. Por cada *texel* válido en la textura atlas se crea un vértice.

Al crear el volumen, se despliega cada uno de los vértices almacenados en el *vertex buffer object*. La coordenada del vértice es utilizada para extraer de la textura atlas su correspondiente posición en el espacio de mundo. Es necesario transformar esta posición al sistema de



(a) (b)

Figura 2.6: Las coordenadas de mundo del objeto 2.6(a) son almacenados en una textura atlas 2.6(b).

coordenadas definido por la cámara de voxelización. Posteriormente, se realiza un proceso similar al propuesto en la técnica de *Slicemap* para introducir este vértice en el *grid*. En la figura 2.7 se observa cómo la escena es transformada en una textura atlas que contiene las coordenadas de mundo, para posteriormente ser insertada en el *grid*.

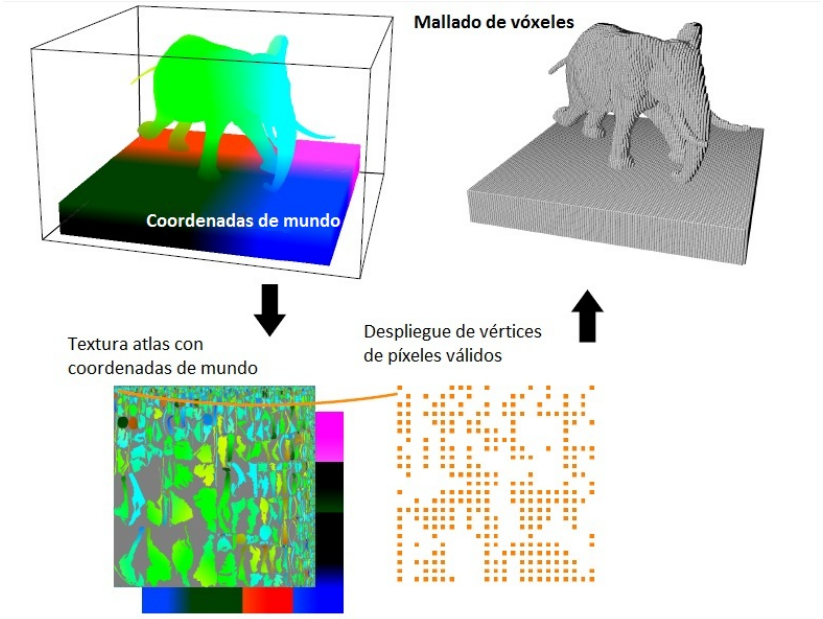


Figura 2.7: Proceso de voxelización de una escena.

Debido a que los objetos son desplegados en una textura atlas antes de ser voxelizados, está técnica carece de los problemas que presentaba *slicemap* con superficies perpendiculares a la cámara de voxelización. Sin embargo, se necesita desplegar cada uno de los objetos en una o varias texturas, y posteriormente procesar cada una de esas texturas para llevarse al *grid*.

Con el uso de cualquiera de las dos técnicas, si existe una escena con objetos dinámicos y estáticos, es posible aplicar una pre voxelización de todas la partes estáticas. De esta manera, en tiempo de ejecución solo será necesario voxelizar los objetos dinámicos y combinar sus vóxeles con los vóxeles precalculados para los objetos estáticos. Además, es posible crear una jerarquía de *mipmap* a partir del *grid* almacenado en una textura 2D, de tal forma que pueda acelerarse los cálculos realizados sobre éste.

2.2.3. Estructura de datos

Para poder acelerar los cálculos sobre el volumen, Thiedemann et al. [4] proponen el uso de una jerarquía, utilizando una idea similar a la propuesta por Forest et al. [54]. Se aprovecha la representación del volumen por medio de una textura 2D para emplear una jerarquía de *mipmaps* [55].

Los *mipmaps* son colecciones de imágenes más pequeñas, que acompañan a una textura principal para aumentar la velocidad de despliegue y reducir sus artefactos. Cada imagen del *mipmap* representa una versión reducida de la textura principal, cuyas dimensiones se van disminuyendo a la mitad hasta obtener una textura de 1×1 píxeles. Así, es posible obtener diversas resoluciones de una escena voxelizada en una textura 2D, donde el nuevo nivel es construido utilizando operaciones de OR lógico en los mapas de bits de niveles anteriores. Debido a que cada píxel siempre posee la misma resolución sin importar su nivel en la jerarquía, la profundidad del volumen siempre mantendrá la misma resolución (en la figura 2.8 se ejemplifica la construcción de esta jerarquía).

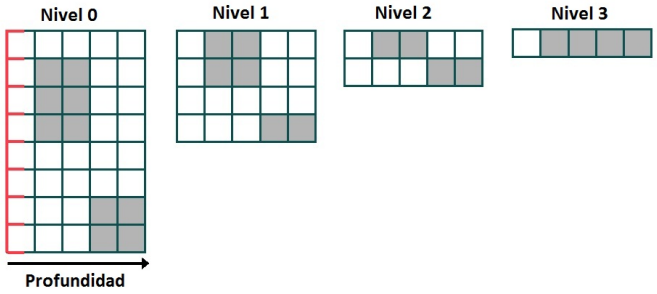


Figura 2.8: Construcción de la jerarquía *mipmap* en dos dimensiones.

En la siguiente sección se explicará el uso de la representación de la escena a través de vóxeles para el cálculo de la iluminación indirecta.

2.3. Cálculo de la Iluminación indirecta

Para obtener la iluminación indirecta es necesario poder lanzar rayos a través de la escena y calcular sus intersecciones con las mismas. Por ello, Thiedemann et al. [4] proponen una prueba de visibilidad entre dos puntos y una prueba para el calculo de la intersección,

haciendo uso de la representación de la escena en una textura 2D con una jearquía *mipmap* para acelerar los cálculos.

2.3.1. Intersección rayo/vóxel

Visibilidad entre dos puntos en el espacio

Con el objetivo de poder trazar un rayo a lo largo de la jerarquía *mipmap*, es necesario transformar el rayo en coordenadas de dispositivo normalizadas (*normal device coordinates, NDC*). Esto quiere decir que las dimensiones del volumen se colocarán en el espacio $[0, 1]^3$, y el rayo también se transformará para quedar dentro de estas coordenadas. Una vez obtenidas las *NDC* del rayo, se proyecta el origen del rayo en el nivel de la jerarquía correspondiente, seleccionando el *texel* apropiado. Una caja envolvente o *bounding box* es generada para ese *texel* en *NDC* y se calcula la intersección del rayo con esta caja. Debido a que un *texel* siempre mantiene la resolución máxima de profundidad, es posible crear una máscara de bits para el rayo que represente los vóxeles que son intersectados en la columna de bits correspondientes al *texel*. Para ello, se debe calcular el punto de entrada y el punto de salida del rayo con la caja envolvente. Una vez obtenidos estos dos valores de profundidad es posible obtener una máscara de bits del rayo, la cual tendrá todos los bits en 1 entre el punto de entrada y el punto de salida del rayo. Esta máscara puede ser obtenida a partir de una textura 2D precalculada.

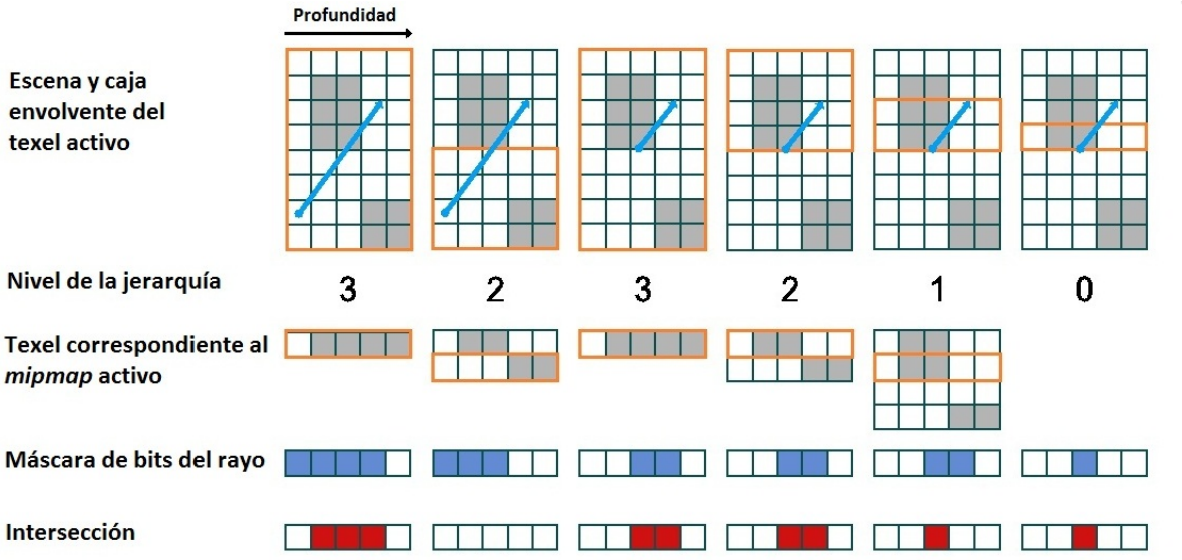


Figura 2.9: Recorrido de un rayo a través de la jerarquía mipmap.

Posteriormente se usa la máscara del rayo obtenida y la máscara de bits almacenadas en el *texel*, para determinar si hubo intersección. Para ello, se realiza una operación *AND* entre las dos máscaras. Si el resultado obtenido es 0, no hubo intersección, por lo que el origen del rayo es trasladado hasta el punto de salida del rayo de la caja envolvente y se incrementa un nivel en la jerarquía para realizar la prueba otra vez con el rayo trasladado. Si

ocurrió una intersección, es necesario decrementar un nivel en la jerarquía para comprobar si la intersección sigue presente a una mayor resolución del volumen. Es necesario alcanzar el nivel con mayor resolución de la jerarquía *mipmap* para asegurar que hubo una intersección. El algoritmo se detiene si hubo intersección o si la longitud del rayo es despreciable.

En la figura 2.9 se muestra un ejemplo del recorrido de un rayo en la jerarquía en dos dimensiones. En las columnas se muestra de arriba hacia abajo: el rayo en azul y la caja envolvente correspondiente al *texel* activo en naranja; el nivel actual de la jerarquía; las máscaras de bits del nivel de la jerarquía actual y el *texel* activo; la máscara de bits de la intersección del rayo con la caja envolvente en el nivel de la jerarquía actual; y los bits intersectados de la máscara de bits del rayo y la del *texel* actual.

Con esta prueba se determina la visibilidad entre dos puntos, pero es necesario realizar ciertos cálculos adicionales para poder conocer las coordenadas del primer punto de intersección del rayo.

Intersección del rayo

Con la prueba anterior se determina la visibilidad entre dos puntos, pero se obtienen todas las intersecciones del rayo para un *texel* en la máxima resolución del volumen. Para determinar las coordenadas de mundo del primer punto de intersección, es necesario calcular el primer bit encendido del *texel* y transformar sus coordenadas *NDC* a coordenadas de mundo.

Detallando en la implementación, se tiene como entrada una máscara de bits que indica todas las intersecciones a lo largo del rayo con los elementos de la escena. Si el producto punto de la dirección del rayo con la dirección de voxelización es mayor a 0, se debe buscar el bit más significativo. Para ello, se utiliza un logaritmo base 2 sobre la máscara de bits. En caso de que el producto punto sea menor a 0, se realiza la operación $X\ AND\ NOT(X-1)$, donde X es uno de los canales de la máscara de bits. Luego de esto, se aplica un logaritmo base 2 para obtener el bit menos significativo. Este último caso puede observarse en el código 2.1.

```
vec3 currentTNear; //vector que contiene la posición del texel
int x = 0;
int bitPosition = 0;

//Se busca el bit más lejano en la dirección de la voxelización
//Se itera sobre cada uno de los canales rgba de la máscara de bits de la
    intersección (Bitmask)
for(int v = 3; (x == 0) && (v >= 0); v--)
{
  x = int(Bitmask[v]);
  if(x != 0)
  {
        //Si x != 0 hay un bit encendido en el canal, por lo que es el
           primer punto de intersección
    int pos32 = int(log2(float(x & ~(x-1)))+0.5);
    bitPosition = (3-v)*32 + pos32;
  }
```

```
}

//Obtener la posición de la intersección en coordenadas NDC [0,1]^3
currentTNear.z = (float(127.0 - bitPosition))/128.0;

//Transformar de coordenadas NDC a coordenadas de mundo
HitPos = UnitToWorldCoordMatrix * vec4(currentTNear,1.0);
```

Código 2.1: Código para obtener el punto de intersección.

Una vez obtenido este bit es necesario transformarlo a coordenadas de mundo para conocer la posición de la intersección. El bit obtenido representará la profundidad y el *texel* de donde se obtuvo la máscara de bits representará las otras dos coordenadas. Primeramente el valor de profundidad está entre los valores $[0, 127]$, que representa la resolución de profundidad del volumen, por lo que es necesario transformar este valor a *NDC*. Se realiza un proceso similar con el alto y el ancho del volumen. Una vez obtenidas las coordenadas de la intersección en *NDC* se procede a transformarlas a coordenadas de mundo, por medio de la inversa de la matriz de transformación usada para voxelizar.

2.3.2. Iluminación indirecta

El cálculo de la iluminación global en tiempo real se logra utilizando una técnica parecida a las técnicas de espacio de imagen. Primero, se despliega la escena y para cada píxel de la imagen final se obtiene el valor de la iluminación directa e indirecta. Para lograr tiempos interactivos es necesario reducir el cálculo a un área cercana al punto de recepción. Esto se debe a que mientras más corto se mantenga el rayo, más rápido se realiza la prueba de intersección.

Para el desarrollo de esta técnica, se requiere obtener el *reflective shadow map (RSM)* [31] y un *G-buffer*. La idea de un *RSM* está basada en que un solo rebote de luz indirecta es causado por superficies que son visibles desde el punto de luz, por lo que son visibles en el *shadow map*. Para generar el *RSM* se usa el mismo procedimiento que genera un *shadow map*, donde la escena es desplegada desde la posición de la luz, pero se utilizan múltiples objetivos de despliegue. Además de guardar los valores de profundidad, se genera un *buffer* de vectores normales, de coordenadas de espacio de mundo, y de la energía reflejada. Se necesita crear un *RSM* para cada una de las luces presentes en la escena. El *G-buffer* sigue la misma idea del *RSM* pero se despliega desde el punto de vista de la cámara. De esta manera se obtienen *buffers* intermedios, donde se almacenará información que se utiliza para el despliegue final.

Luego de obtenido los *RSMs* y el *G-buffer*, se procede al cálculo de la luz indirecta en cada uno de los píxeles de la imagen final. Para ello, se hace uso de una versión reducida de la ecuación de despliegue propuesta por Philip Dutré et al. [2] (ecuación 1.1). Si se considera que las superficies de los objetos no emiten radiancia por ellas mismas, la ecuación quedaría de la siguiente manera:

$$L(x \to \Theta) = \int_{\Omega_x} fr(x, \Psi \to \Theta) L(x \leftarrow \Psi) \cos(N_x, \Psi) d_{\omega\Psi} \tag{2.2}$$

Thiedemann et al. [4] proponen usar la integración de Monte Carlo para simplificar la ecuación de la siguiente manera:

$$L(x \to \Theta) \approx \frac{1}{N} \sum_{i=1}^{N} \frac{fr(x, \Psi_i \to \Theta) \tilde{L}(x \leftarrow \Psi_i) \cos(N_x, \Psi_i)}{p(\Psi_i)} \tag{2.3}$$

donde $p(\Psi_i)$ es una función de densidad de probabilidad arbitraria y Ψ_i son las N direcciones de los rayos generadas por la función p. $\tilde{L}$ es la radiancia entrante aproximada, usando la escena voxelizada. Además, como se utilizan superficies difusas, la ***BRDF*** puede sustituirse por $\frac{\rho(x)}{\pi}$, por lo que la ecuación puede expresarse como:

$$L(x) \approx \frac{\frac{\rho(x)}{\pi}}{N} \sum_{i=1}^{N} \frac{\tilde{L}(x \leftarrow \Psi_i) \cos(N_x, \Psi_i)}{p(\Psi_i)} \tag{2.4}$$

Algoritmo 1 Primera intersección del rayo.

```
1: procedure ObtenerLuzIndirecta
2:     for cada rayo a lanzar do
3:         LanzarRayo()            ▷ Calcular intersecciones para cada píxel en la imagen
4:         for cada luz en la escena do
5:             CalculoIluminacion()        ▷ Calcular la iluminación en cada intersección
6:         end for
7:         CombinarSolucion()      ▷ Combinar la contribución con los rayos anteriores
8:     end for
9: end procedure
```

Con esta ecuación simplificada, la luz indirecta se calcula promediando el resultado del trazado de N rayos desde cada uno de los píxeles del ***G-buffer*** (el cual contiene la posición en coordenadas de mundo vista por la cámara) con una distancia máxima r. Para acelerar los cálculos se realizan múltiples pasadas, con una pasada por rayo como se expresa en el algoritmo 1. Para cada rayo trazado, se calcula el primer punto de intersección y se obtiene la radiancia directa en el vóxel intersectado. Para ello, se realiza una proyección de ese vóxel en el ***RSM***, el cual contendrá la información correspondiente al valor de radiancia directa en ese píxel. Esta radiancia solo será válida si la distancia de la posición de la intersección y la posición almacenada en el ***RSM*** es menor a un cierto umbral ϵ. Si esta condición no se cumple, ese punto se encuentra en sombra. Thiedemann et al. [4] proponen que el umbral ϵ sea ajustado a la discretización de la voxelización v, el tamaño del píxel s en el ***RSM***, la proyección perspectiva y la orientación de la normal α. Esto conlleva a que $\epsilon = max(v, \frac{s}{\cos\alpha} \cdot \frac{z}{Znear})$. El proceso de proyección del punto del vóxel en el ***RSM*** es necesario realizarlo para cada una de las luces en la escena, por lo que a mayor cantidad de luces más costosa es la solución. Esta técnica permite el cálculo de la luz indirecta de un solo rebote en un radio r. Manteniendo los valores de r pequeños, permite el cálculo en tiempo real.

En la figura 2.10 se muestra cómo se realiza el cálculo de la luz indirecta en el área cercana a un punto x. Para esto, se trazan diversos rayos en el hemisferio de x. Para cada

rayo es necesario obtener su punto de intersección y proyectarlo en el *RSM* para calcular la luz directa. En este caso, las intersecciones son A y B. En el caso de A, su distancia con la posición del píxel en el *RSM* es menor al umbral ϵ, por lo que su valor es tomado. En cambio, la posición de B difiere de la posición almacenada en el *RSM*, lo que significa que el punto se encuentra en sombra. Si ninguna intersección es encontrada, se puede obtener un valor opcional del ambiente (punto C).

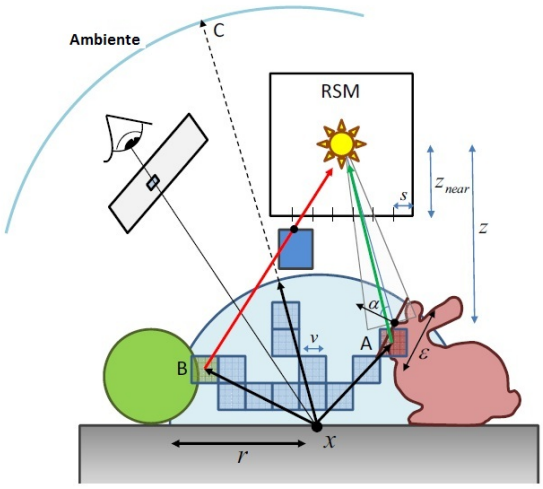


Figura 2.10: Cálculo de la luz indirecta en el área cercana. Imagen tomada de [4].

2.4. Despliegue

Con los pasos expuestos anteriormente, se logra obtener la contribución de la luz indirecta para la imagen final. Sin embargo, es necesario mezclarla con la luz directa. Para obtener la luz directa se debe considerar la contribución de cada una de las luces. Con el uso del mismo *G-buffer* que se utilizó para la luz indirecta, se calcula la luz directa en cada uno de los píxeles de la imagen, considerando la sombra por medio del *shadow map*. De esta manera, la obtención de la iluminación global es resumida en el algoritmo 2.

Algoritmo 2 Primera intersección del rayo.

```
1: procedure ILUMINACIONGLOBAL
2:     ObtenerGBuffer()
3:     ObtenerRSM()              ▷ En este paso también se obtiene el shadow map
4:     ObtenerLuzIndirecta()
5:     ObtenerLuzDirecta()
6:     CombinarSoluciones()      ▷ Luz directa más luz indirecta
7: end procedure
```

Capítulo 3

Implementación

En este capítulo se expondrá como se realizó la implementación de las ideas expuestas en el trabajo de Thiedemann et al. [4] sobre iluminación global en tiempo real. Primero se explicarán las herramientas de hardware y software utilizadas para la implementación. Luego se expondrá un diagrama de clases y se detallarán las clases más importantes. Posteriormente, se mostrará la importancia de los *shaders* en la implementación, para finalmente profundizar en el detalle de la implementación de los *shaders*.

3.1. Metodología

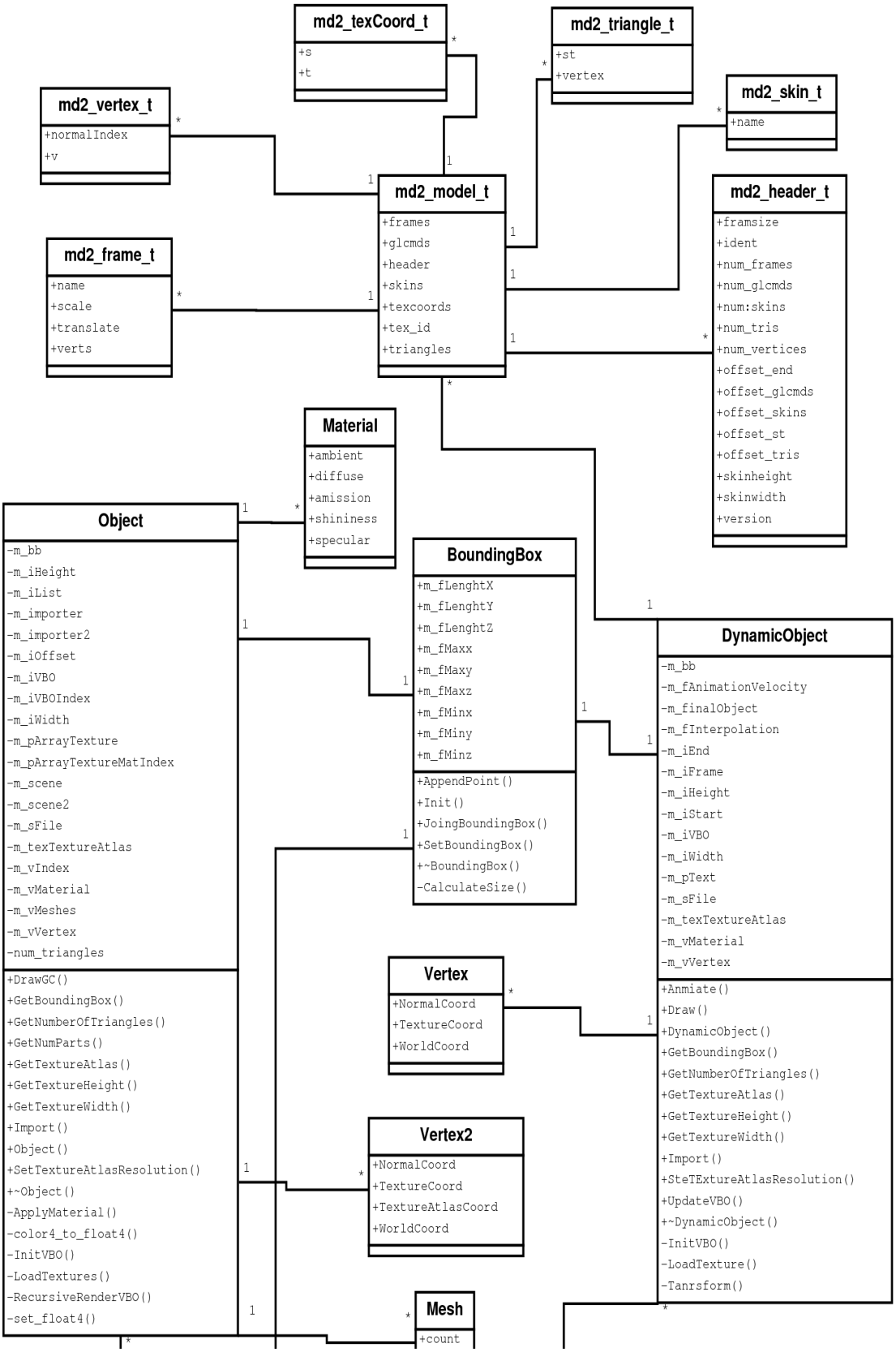
El desarrollo de la implementación se basa en el paradigma de la programación orientada a objetos. El hardware requerido para su ejecución consistió en una PC convencional con una tarjeta gráfica con soporte a *GLSL* con versión 3.30.

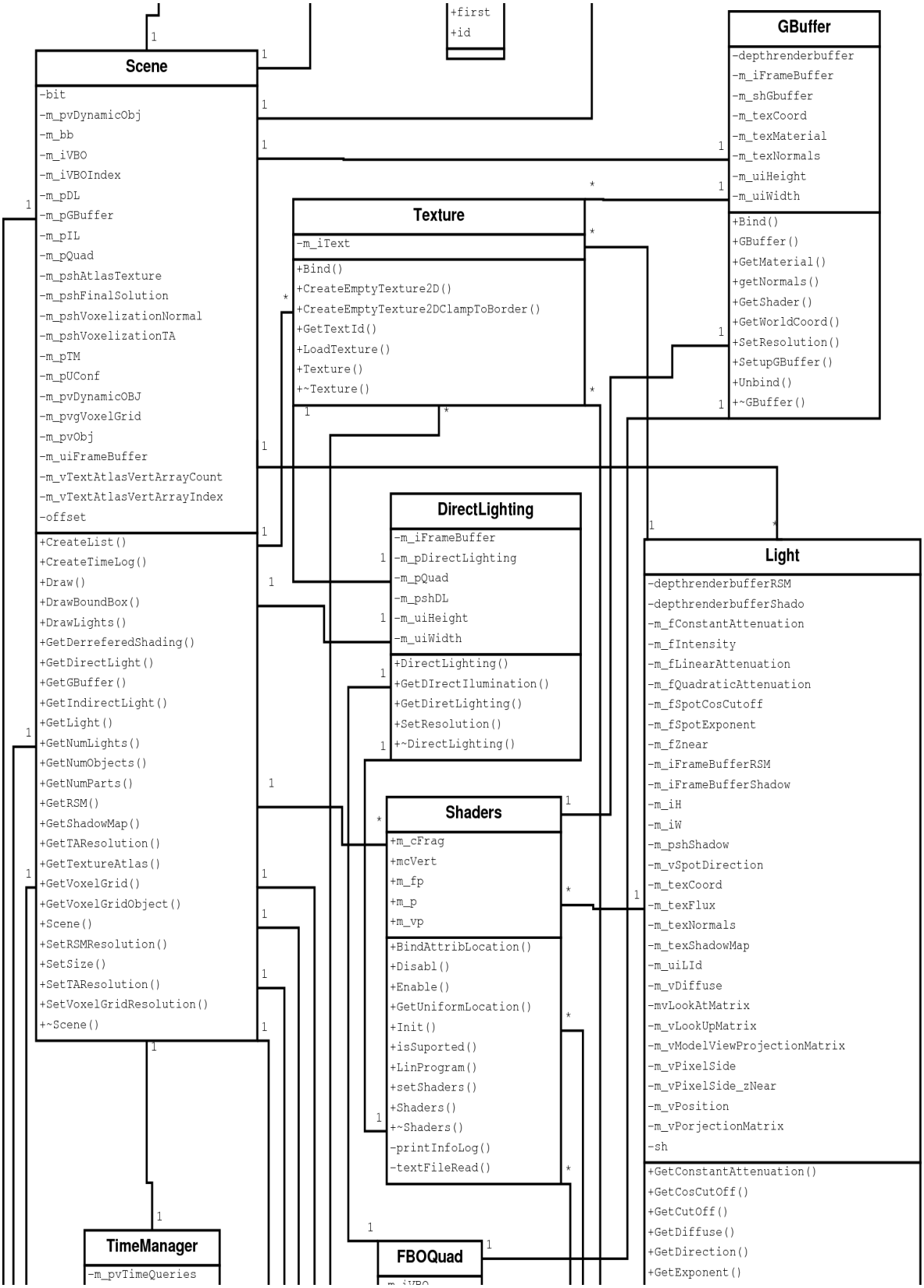
Por otro lado, se utilizó el siguiente software para el desarrollo:

- Windows 7 como sistema operativo.
- Microsoft Visual Studio 2010 como ambiente de trabajo.
- Ant Tweak Bar [56] en su versión 1.15 para la interfaz gráfica dentro de la aplicación.
- *OpenGL* en su versión 3.3.
- Open Asset Import Library [57] en su versión 3.0 para la carga de la escena y los modelos.

3.2. Diagrama de clases y estructuras de datos

En la figura 3.1 se muestra un diagrama de clases del programa, con sus respectivos métodos y atributos. En esta sección se explicará en detalle las principales clases.


md2_texCoord_t
+s
+t
md2_triangle_t
+st
+vertex
md2_vertex_t
+normalIndex
+v
md2_skin_t
+name
md2_model_t
+frames
+glcmds
+header
+skins
+texcoords
+tex_id
+triangles
md2_header_t
+framsize
+ident
+num_frames
+num_glcmds
+num:skins
+num_tris
+num_vertices
+offset_end
+offset_glcmds
+offset_skins
+offset_st
+offset_tris
+skinheight
+skinwidth
+version
md2_frame_t
+name
+scale
+translate
+verts
Material
+ambient
+diffuse
+amission
+shininess
+specular
Object
-m_bb
-m_iHeight
-m_iList
-m_importer
-m_importer2
-m_iOffset
-m_iVBO
-m_iVBOIndex
-m_iWidth
-m_pArrayTexture
-m_pArrayTextureMatIndex
-m_scene
-m_scene2
-m_sFile
-m_texTextureAtlas
-m_vIndex
-m_vMaterial
-m_vMeshes
-m_vVertex
-num_triangles
+DrawGC()
+GetBoundingBox()
+GetNumberOfTriangles()
+GetNumParts()
+GetTextureAtlas()
+GetTextureHeight()
+GetTextureWidth()
+Import()
+Object()
+SetTextureAtlasResolution()
+~Object()
-ApplyMaterial()
-color4_to_float4()
-InitVBO()
-LoadTextures()
-RecursiveRenderVBO()
-set_float4()
BoundingBox
+m_fLenghtX
+m_fLenghtY
+m_fLenghtZ
+m_fMaxx
+m_fMaxy
+m_fMaxz
+m_fMinx
+m_fMiny
+m_fMinz
+AppendPoint()
+Init()
+JoingBoundingBox()
+SetBoundingBox()
+~BoundingBox()
-CalculateSize()
DynamicObject
-m_bb
-m_fAnimationVelocity
-m_finalObject
-m_fInterpolation
-m_iEnd
-m_iFrame
-m_iHeight
-m_iStart
-m_iVBO
-m_iWidth
-m_pText
-m_sFile
-m_texTextureAtlas
-m_vMaterial
-m_vVertex
+Anmiate()
+Draw()
+DynamicObject()
+GetBoundingBox()
+GetNumberOfTriangles()
+GetTextureAtlas()
+GetTextureHeight()
+GetTextureWidth()
+Import()
+SteTExtureAtlasResolution()
+UpdateVBO()
+~DynamicObject()
-InitVBO()
-LoadTexture()
-Tanrsform()
Vertex
+NormalCoord
+TextureCoord
+WorldCoord
Vertex2
+NormalCoord
+TextureCoord
+TextureAtlasCoord
+WorldCoord
Mesh
+count


+first
+id
GBuffer
-depthrenderbuffer
-m_iFrameBuffer
-m_shGbuffer
-m_texCoord
-m_texMaterial
-m_texNormals
-m_uiHeight
-m_uiWidth
+Bind()
+GBuffer()
+GetMaterial()
+getNormals()
+GetShader()
+GetWorldCoord()
+SetResolution()
+SetupGBuffer()
+Unbind()
+~GBuffer()
Scene
-bit
-m_pvDynamicObj
-m_bb
-m_iVBO
-m_iVBOIndex
-m_pDL
-m_pGBuffer
-m_pIL
-m_pQuad
-m_pshAtlasTexture
-m_pshFinalSolution
-m_pshVoxelizationNormal
-m_pshVoxelizationTA
-m_pTM
-m_pUConf
-m_pvDynamicOBJ
-m_pvgVoxelGrid
-m_pvObj
-m_uiFrameBuffer
-m_vTextAtlasVertArrayCount
-m_vTextAtlasVertArrayIndex
-offset
+CreateList()
+CreateTimeLog()
+Draw()
+DrawBoundBox()
+DrawLights()
+GetDerreferedShading()
+GetDirectLight()
+GetGBuffer()
+GetIndirectLight()
+GetLight()
+GetNumLights()
+GetNumObjects()
+GetNumParts()
+GetRSM()
+GetShadowMap()
+GetTAResolution()
+GetTextureAtlas()
+GetVoxelGrid()
+GetVoxelGridObject()
+Scene()
+SetRSMResolution()
+SetSize()
+SetTAResolution()
+SetVoxelGridResolution()
+~Scene()
Texture
-m_iText
+Bind()
+CreateEmptyTexture2D()
+CreateEmptyTexture2DClampToBorder()
+GetTextId()
+LoadTexture()
+Texture()
+~Texture()
DirectLighting
-m_iFrameBuffer
-m_pDirectLighting
-m_pQuad
-m_pshDL
-m_uiHeight
-m_uiWidth
+DirectLighting()
+GetDIrectIlumination()
+GetDiretLighting()
+SetResolution()
+~DirectLighting()
Light
-depthrenderbufferRSM
-depthrenderbufferShado
-m_fConstantAttenuation
-m_fIntensity
-m_fLinearAttenuation
-m_fQuadraticAttenuation
-m_fSpotCosCutoff
-m_fSpotExponent
-m_fZnear
-m_iFrameBufferRSM
-m_iFrameBufferShadow
-m_iH
-m_iW
-m_pshShadow
-m_vSpotDirection
-m_texCoord
-m_texFlux
-m_texNormals
-m_texShadowMap
-m_uiLId
-m_vDiffuse
-mvLookAtMatrix
-m_vLookUpMatrix
-m_vModelViewProjectionMatrix
-m_vPixelSide
-m_vPixelSide_zNear
-m_vPosition
-m_vPorjectionMatrix
-sh
+GetConstantAttenuation()
+GetCosCutOff()
+GetCutOff()
+GetDiffuse()
+GetDirection()
+GetExponent()
Shaders
+m_cFrag
+mcVert
+m_fp
+m_p
+m_vp
+BindAttribLocation()
+Disabl()
+Enable()
+GetUniformLocation()
+Init()
+isSuported()
+LinProgram()
+setShaders()
+Shaders()
+~Shaders()
-printInfoLog()
-textFileRead()
TimeManager
-m_pvTimeQueries
FBOQuad

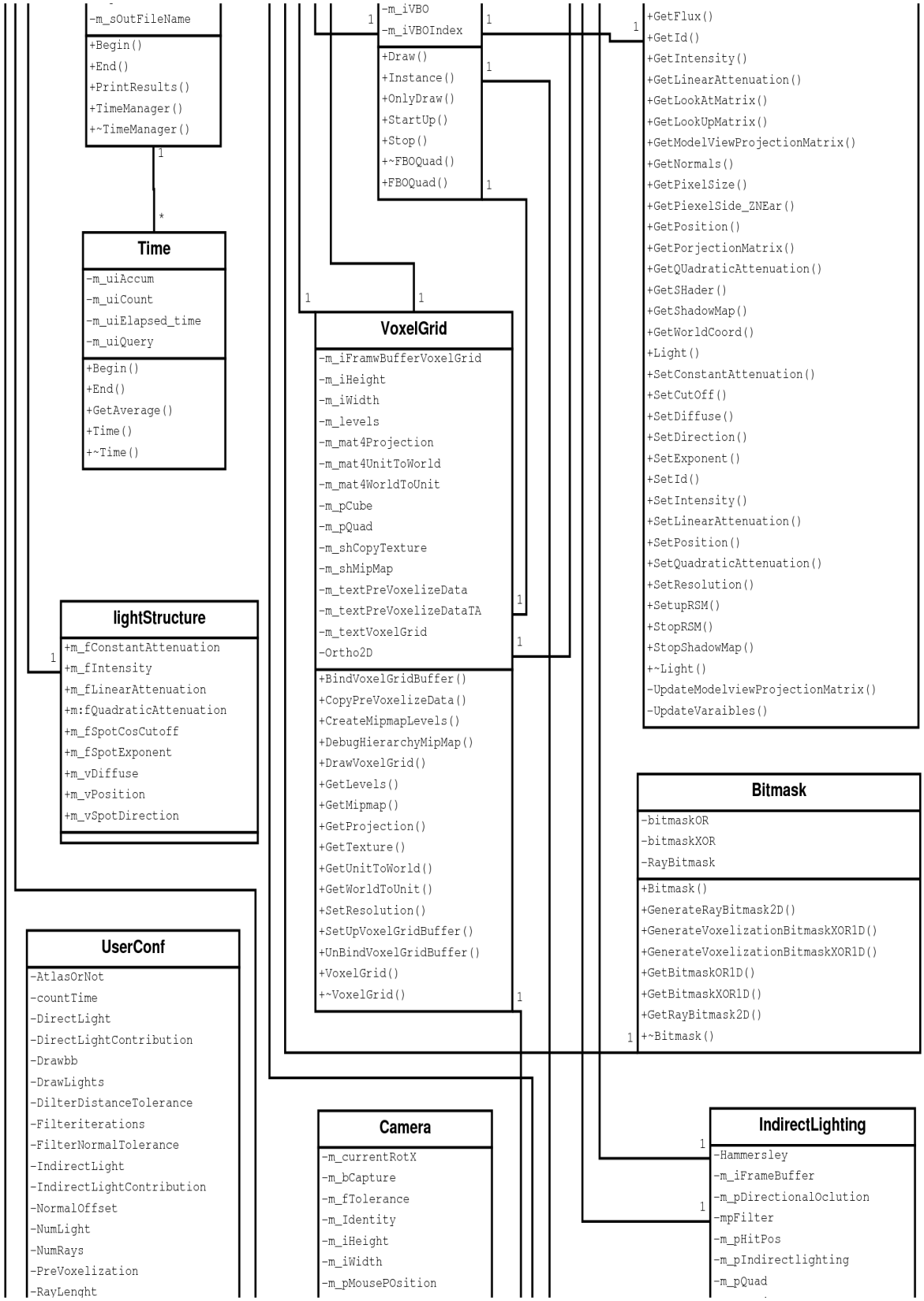

-m_sOutFileName
+Begin()
+End()
+PrintResults()
+TimeManager()
+~TimeManager()
Time
-m_uiAccum
-m_uiCount
-m_uiElapsed_time
-m_uiQuery
+Begin()
+End()
+GetAverage()
+Time()
+~Time()
lightStructure
+m_fConstantAttenuation
+m_fIntensity
+m_fLinearAttenuation
+m:fQuadraticAttenuation
+m_fSpotCosCutoff
+m_fSpotExponent
+m_vDiffuse
+m_vPosition
+m_vSpotDirection
UserConf
-AtlasOrNot
-countTime
-DirectLight
-DirectLightContribution
-Drawbb
-DrawLights
-DilterDistanceTolerance
-Filteriterations
-FilterNormalTolerance
-IndirectLight
-IndirectLightContribution
-NormalOffset
-NumLight
-NumRays
-PreVoxelization
-RayLenght
-m_iVBO
-m_iVBOIndex
+Draw()
+Instance()
+OnlyDraw()
+StartUp()
+Stop()
+~FBOQuad()
+FBOQuad()
VoxelGrid
-m_iFramwBufferVoxelGrid
-m_iHeight
-m_iWidth
-m_levels
-m_mat4Projection
-m_mat4UnitToWorld
-m_mat4WorldToUnit
-m_pCube
-m_pQuad
-m_shCopyTexture
-m_shMipMap
-m_textPreVoxelizeData
-m_textPreVoxelizeDataTA
-m_textVoxelGrid
-Ortho2D
+BindVoxelGridBuffer()
+CopyPreVoxelizeData()
+CreateMipmapLevels()
+DebugHierarchyMipMap()
+DrawVoxelGrid()
+GetLevels()
+GetMipmap()
+GetProjection()
+GetTexture()
+GetUnitToWorld()
+GetWorldToUnit()
+SetResolution()
+SetUpVoxelGridBuffer()
+UnBindVoxelGridBuffer()
+VoxelGrid()
+~VoxelGrid()
+GetFlux()
+GetId()
+GetIntensity()
+GetLinearAttenuation()
+GetLookAtMatrix()
+GetLookUpMatrix()
+GetModelViewProjectionMatrix()
+GetNormals()
+GetPixelSize()
+GetPiexelSide_ZNEar()
+GetPosition()
+GetPorjectionMatrix()
+GetQUadraticAttenuation()
+GetSHader()
+GetShadowMap()
+GetWorldCoord()
+Light()
+SetConstantAttenuation()
+SetCutOff()
+SetDiffuse()
+SetDirection()
+SetExponent()
+SetId()
+SetIntensity()
+SetLinearAttenuation()
+SetPosition()
+SetQuadraticAttenuation()
+SetResolution()
+SetupRSM()
+StopRSM()
+StopShadowMap()
+~Light()
-UpdateModelviewProjectionMatrix()
-UpdateVaraibles()
Bitmask
-bitmaskOR
-bitmaskXOR
-RayBitmask
+Bitmask()
+GenerateRayBitmask2D()
+GenerateVoxelizationBitmaskXOR1D()
+GenerateVoxelizationBitmaskXOR1D()
+GetBitmaskOR1D()
+GetBitmaskXOR1D()
+GetRayBitmask2D()
+~Bitmask()
Camera
-m_currentRotX
-m_bCapture
-m_fTolerance
-m_Identity
-m_iHeight
-m_iWidth
-m_pMousePOsition
IndirectLighting
-Hammersley
-m_iFrameBuffer
-m_pDirectionalOclution
-mpFilter
-m_pHitPos
-m_pIndirectlighting
-m_pQuad

Figura 3.1: Diagrama de clases de la implementación realizada.

3.2.1. Clase *FBOQuad*

FBOQuad es una clase que sigue el patrón *singleton*, lo cual solo permite la creación de una sola instancia para la ejecución del programa. En ella se define un cuadrado de dimensiones $[-1, 1]^2$ almacenado en un *vertex buffer object*. Tiene los métodos necesarios para preparar la tarjeta de video para su preparación (*StartUp*), su despliegue (*OnlyDraw*) y su culminación (*Stop*). La idea de esta clase es usarse en combinación con un *shader* para desplegar un cuadrado que ocupe toda la ventana y así poder crear un fragmento por cada píxel de la misma.

3.2.2. Clase *Texture*

Esta clase tiene los métodos necesario para la creación y carga de una textura 2D. Los métodos *CreateEmptyTexture2DClampToBorder* y *CreateEmptyTexture2D* permiten la creación de una textura en blanco, o pasándoles información, con la posibilidad de cambiar la mayoría de los valores de entrada en la declaración de una textura en OpenGL. Además, el método *LoadTexture* carga una textura desde un archivo de entrada. La clase elimina toda la información de la texturas excepto el identificador creado por OpenGL, por lo que la textura sólo queda almacenada en la tarjeta de video. Las clases que contengan una instancia de *Texture* pueden acceder al identificador de la textura por medio de *GetTextId*.

3.2.3. Clase *Shaders*

La clase *Shaders* permite la creación, carga y compilación de un *fragment* y un *vertex shader*. Además, permite la activación y desactivación del *shader* por medio de los métodos *Enable* y *Disable*, y la modificación de variables uniformes al poder tener acceso al programa

por el atributo *m_p*. También permite vincular ubicaciones para atributos de los *shaders*. Tanto *Shaders* como *Textures* son las clases más utilizadas en el programa.

3.2.4. Mediciones de tiempo

Se define la clase *Time* para medir el tiempo y *TimeManager* como un coordinador. En la clase *Time* los métodos *Begin* y *End* definen el fragmento de código en el cual se va a medir el tiempo. El tiempo se va acumulando para posteriormente ser promediado con el método *GetAverage*. Debido a que la mayor parte del algoritmo esta pensado para funcionar en la GPU, las mediciones de tiempo se realizan con consultas OpenGL, a través de la variables *GL_TIME_ELAPSED*. Por otro lado, la clase *TimeManager* se encargará de coordinar varias instancias de la clase *Time*, de manera que puedan medirse diferentes partes del algoritmo. Con el método *PrintResults()* se totalizan los tiempos y se escriben en disco en un archivo de salida.

3.2.5. Manejo de modelos

Estas clases sirven para la carga y el despliegue de modelos. Las clases principales son *Object* y *DynamicObject*, aunque se utilizan otras estructuras auxiliares.

DynamicObject

La clase *DynamicObject* sirve para la carga y despliegue de modelos en formato MD2. Debido a su peculiar formato de almacenamiento, es necesario la creación de unas subclases para su importación. Se utiliza *md2_model_t* para la carga del modelo, clase que a su vez usa instancias de *md2_header_t* para describir las cabecera, *md2_vertex_t* para los vértices, *md2_skin_t* para los nombres de las texturas, *md2_texCoord_t* para las coordenadas de texturas, *md2_triangle_t* para los triángulos y *md2_frame_t* para los frames. Una vez importado el modelo con el método *Import*, se cambia la estructura de datos por una matriz de *Vertex* (*finalObject*), la cual se encargará de almacenar los frames. *Vertex* almacena por cada vértice las coordenadas de mundo, el vector normal y las coordenadas de textura. Además, mientras se transforma la estructura del modelo, se realizan las transformaciones sobre los vértices. Una caja envolvente del model es creada haciendo uso de la clase *BoundingBox*.

El objeto se despliega mediante el uso de un *vertex buffer object*, con el método *Draw*. Debido a que es un objeto dinámico se necesita del método *Animate* para llevar el tiempo entre frames y *UpdateVBO* para interpolar los vértices y actualizar el *vertex buffer object*. La clase *DynamicObject* también posee dos instancias de *Texture*: *pText* para el almacenamiento de la textura del modelo y *m_texTextureAtlas* para almacenar la textura atlas. Es posible modificar la resolución de la textura atlas a través del método *SetTextureAtlasResolution*.

Object

La clase *Object* se utiliza para cargar y desplegar modelos en formato OBJ. Se utiliza la librería Open Asset Import Library para importar el modelo y posteriormente su estructura de datos es transformada. El modelo es almacenado en el arreglo *m_vVertex*, que contiene instancias de la clase *Vertex2*. Debido a que el modelo puede dividirse en varios grupos categorizados por material, se utiliza el vector *m_vMeshes* con instancias de la clase *Mesh* para dividir los grupos, el vector *m_vMaterial* para almacenar los materiales y el vector *m_pArrayTexture* para almacenar las texturas de cada grupo. También posee una textura atlas (*m_texTextureAtlas*), a la cual se le puede cambiar la resolución. Se utiliza el método *DrawGC* para poder dibujar el modelo por medio de un *vertex buffer object*.

3.2.6. Manejo de luces

En la clase *Light* se poseen las propiedades y los métodos para el manejo de una luz focal. Por ello, se definen las propiedades posición (*m_vPosition*) , dirección (*m_svSpotDIrection*), color difuso (*m_vDiffuse*), ángulo de apertura (*m_ fSpotCosCutoff*) y las variables de atenuación (*m_ fSpotExponent*, *m_ fConstantAttenuation*, *m_ fLinearAttenuation* y *m_ fQuadraticAttenuation*). Para poder modificar los atributos se crean los métodos *get* y *set* para cada uno de ellos.

Dado la necesidad de crear un *reflective shadow map* y un *shadow map* por luz, la clase *Light* contiene los métodos necesarios para generarlos. Para ello, es necesario un *FrameBuffer Object*, las instancias de la clase *Shaders* (*sh* y *m_pshShadow*) y las texturas para almacenar los resultados (*m_texCoord*, *m_texFlux*, *m_texNormals* y *m_texShadowMap*). Las texturas son accesibles a través de métodos *get*. Además, es posible cambiar la resolución de estos *buffer* con el método *SetResolution*.

3.2.7. Clase *GBuffer*

Esta clase está diseñada para crear los *buffers* temporales necesarios por el *G-Buffer*. De manera similar a la clase *Light*, se tiene un *FrameBuffer Object*, la instancia de la clase *Shaders* (*m_shGbuffer*) y las texturas para almacenar los resultados (*m_texCoord*, *m_texMaterial* y *m_texNormals*). Estas texturas son accedidas a través de métodos *gets*. Por otro lado, la resolución de los *buffers* depende del tamaño de la ventana del programa, por lo que es posible modificarlos con el método *SetResolution*.

3.2.8. Voxelización de la escena

Se define la clase *VoxelGrid* con el fin de contener un *FrameBuffer Object* y la textura *m_ TextVoxelGrid*, con lo cual se almacena la escena voxelizada. También permite hacer una prevoxelización, la cual almacena la escena prevoxelizada en la textura auxiliar *m_ TextPreVoxelizeDataTA* si se utiliza el método de texturas atlas, o la textura *m_ TextPreVoxelizeData*

si se utiliza *slicemap*. Para poder copiar la textura prevoxelizada en la textura a utilizar, se usa la instancia de la clase *Shaders m_shCopyTexture*.

También, se define el método *GetMipmap* para obtener la jerarquía *mipmap* de la textura que contiene la escena voxelizada. Para ello, necesita la instancia de la clase *Shaders m_shMipMap* y una instancia de la clase *FBOQuad* para desplegar un cuadrado que abarque toda el área de despliegue.

Además, con el método *DrawVoxelGrid* se permite el despliegue de la escena por medio de vóxeles. Dado que los vóxeles pueden representarse como cubos, se tiene una clase *m_pCube* que define un cubo para desplegar por medio de *vertex buffer objects*. El método recorre la textura que contiene la escena y despliega un cubo por cada bit encendido.

El tamaño de las texturas presentes en esta clase puede ser modificado a través del método *SetResolution*, y es independiente del tamaño de la ventana.

3.2.9. Obtención de la luz directa

La clase *DirectLighting* se encarga de obtener la iluminación directa de la escena por medio del método *GetDirectLighting*. Utiliza la información almacenada por el *G-Buffer* en vez de desplegar la escena otra vez. Por ello, tiene la misma resolución que la ventana del programa y utiliza una instancia de *FBOQuad* para desplegar un cuadrado que ocupe toda la ventana. El cálculo de la iluminación se realiza con un *shader* definido en la variable *m_pshDL*. Una vez calculada la iluminación es posible obtener la textura resultado a través del método *GetDirectIlumination*.

3.2.10. Obtención de la luz indirecta

La estructuras necesarias para la obtención de la luz indirecta están definidas en la clase *IndirectLighting*. Se definen las texturas *m_ pHitPos*, *m_ pIndirectlighting* y *m_ pDirectionalOclution* para la obtención de la iluminación. Posteriormente, la clase permite aplicar un filtro a la solución, por lo que se utiliza la textura auxiliar *m_pFilter*. Todas las texturas tienen la misma resolución que la ventana principal del programa. Debido a que se necesita realizar cálculos sobre todos los píxeles de la imagen, se utiliza una instancia de la clase *FBOQuad*. Todos los cálculo son realizados a través de la GPU, por lo que es necesario el uso de instancias de la clase *Shaders* y el uso de un *FrameBuffer Object*.

3.2.11. Clase *Scene*

Esta es la clase principal, ya que se encarga de manejar todas las demás clases del programa. Contiene instancias de las clases *UserConf*, *Camera*, *lightStructure*, *TimeManager*, *Shaders*, *Object*, *DynamicObject*, *GBuffer*, *Light*, *VoxelGrid*, *DirectLighting* e *Indirectlighting*. También posee una instancia de la clase *Bitmask*, la cual se encarga de crear unas texturas auxiliares que servirán de entrada a otras clases. La idea de la clase *Scene* es enmascarar todos los métodos de las clases anteriores por los métodos presentes en *Scene*, así el método principal del programa sólo contendrá una instancia de *Scene*.

La clase define arreglos que contienen a los objetos (*m_pvDynamicObj* y *m_pvObj*) y esta encargado de crear las texturas atlas de los mismos. Para ello, necesita el uso de *shaders*. Posteriormente se encarga de crear un *FrameBuffer Object* que contiene todos los vértices válidos de una textura atlas.

Además, a través del método *GetDerreferedShading*, la clase *Scene* se encarga de combinar y desplegar la solución final.

3.2.12. Navegación en la escena

La navegación dentro de la escena es manejada por la clase *Camera*. En ella se definen los 3 vectores principales para poder definir una cámara con OpenGL: el vector posición (*m_pPosition*), el vector *up* (*m_pUpVector*) y el vector de dirección (*m_pView*). Con estos 3 vectores es posible construir una matriz de transformaciones que es necesaria para varias etapas del algoritmo, por lo que es posible obtenerla con el método *GetLookAt*.

Para mover la cámara en la escena con el teclado se utilizan los métodos *MoveCamera*, *MoveCamera2* y *MoveCameraVertical*. Para el mouse se usan los métodos *RotateView* y *SetViewByMouse*.

3.2.13. Manejo de los menús

Estas clases sirven como estructuras de datos para manejar las diferentes entradas de usuario presentes en los menús de Ant Tweak Bar. Las dos clases que son utilizadas con este propósito son *lightStructure* y *UserConf*. *UserConf* contiene los campos necesario para permitirle manejar al usuario diferentes parámetros del programa, como la resolución de los *buffers* intermedios, parámetros del rayo, parámetros del filtro, activación de diferentes características, entre otros. Por otro lado, *lightStructure* contiene todas las propiedades de la luz, de tal forma que el usuario pueda cambiarlas y estas sean transmitidas a una instancia de la clase *Light*.

3.3. Algoritmo

Los procesos más importantes del programa fueron realizados para trabajar en la GPU mediante el uso de *shaders*. Por ello, para estudiar con mayor detalle cada uno de los pasos es necesario analizar cada unos los programas del GPU. En la imagen 3.2 podemos observar los *fragment shaders* (con extensión .frag) y los *vertex shaders* (con extensión.vert) utilizados en el programa.

Para poder almacenar los resultados del uso de los *shaders* y evitar el despliegue de estos por pantalla, se hizo uso de los *FrameBuffer Objects*. Estos son mecanismos que permiten el despliegue de imágenes a *FrameBuffers* diferentes al *FrameBuffer* por defecto de OpenGL. Además, estos permiten desplegar directamente a una textura e incluso utilizar múltiples objetivos de despliegue, para ser mostrados en varias texturas simultáneamente. Para simplificar su uso, se crean varios *FrameBuffer Object*, donde para cada uno se configuran las texturas

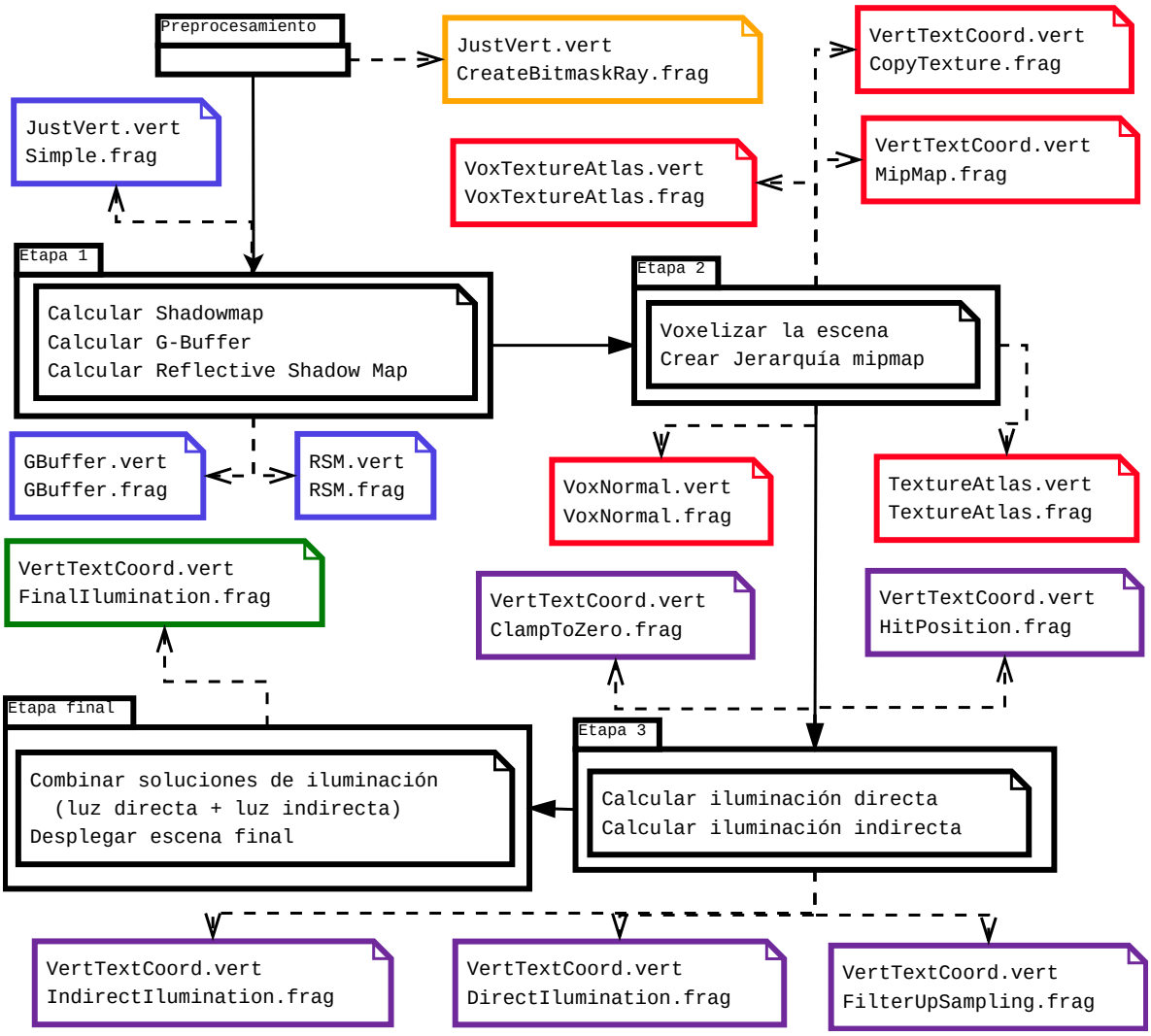


Figura 3.2: Distribución de los *shaders* (bordes de colores) en las diferentes etapas del algoritmo (bordes negros).

a utilizar como blancos de despliegue. Además, es necesario indicar a OpenGL cual es el tamaño de la ventana que se va a desplegar, por lo que es necesario cambiar el tamaño del *viewport* cada vez que se va a utilizar un *FrameBuffer Object* diferente.

Cada *FrameBuffer Object* creado en el programa esta ligado con alguna etapa del mismo, existiendo uno o más programas de *shaders* asociado a cada uno. Para cada *shader* en esta aplicación se tiene un *vertex shader* y un *fragment shader*. Como se utiliza GLSL en su versión 3.30, es necesario enviar las matrices de transformación a los *shaders* por medio de variables uniformes, ya que OpenGL no las envía de manera directa. Es necesario realizar lo mismo para las propiedades de las luces y para las propiedades de los vértices.

En la siguiente sección se explicará detalladamente la función de cada uno de los *shaders* en las diferentes etapas del programa.

3.4. Implementación detallada en la GPU

En estas sección se detallará en los *shaders* realizados para esta implementación.

3.4.1. *Shaders* multifuncionales

Pase de vértices

```
uniform mat4 ModelViewProjectionMatrix;

in vec4 glVertex;

out vec2 pos;

void main(){
  gl_Position = ModelViewProjectionMatrix * glVertex;
  pos = glVertex.xy;
}
```

Código 3.1: Pase de vértices del *vertex shader* al *fragment shader*.

El *vertex shader* mostrado en el código 3.1 solo tiene como entrada la posición de un vértice en espacio de objeto y lo envía al *fragment shader*. Además, multiplica la posición del vértice por la matriz de transformaciones para colocarlo en la posición correcta en el espacio de imagen.

Pase de vértices y de coordenadas de textura

```
uniform mat4 ModelViewProjectionMatrix;

in vec4 glVertex;
in vec2 glTexCoord;

out vec2 TextCoord;
out vec4 pos;

void main(){
  gl_Position = ModelViewProjectionMatrix * glVertex;
  TextCoord = glTexCoord;
  pos = gl_Position;
}
```

Código 3.2: Pase de vértices y coordenadas de textura del *vertex shader* al *fragment shader*.

En el código 3.2 se recibe como entrada la posición de un vértice en espacio de objeto y su correspondiente coordenada de textura. La posición es multiplicada por la matriz de transformaciones para luego ser enviada al *fragment shader*, al igual que la coordenada de textura. A pesar de su simplicidad, este es uno de los *vertex shaders* más utilizados en el programa.

Copia de una textura

```
uniform usampler2D PreVoxelizedTexture;

in vec2 TextCoord;
in vec4 pos;

/*El fragment shader escribe información en uno o más framebuffers.
  Cuando se tiene varias salidas, el valor de "location" indica cual es
  el framebuffer de  salida correspondiente a esa variable*/
layout(location = 0) out uvec4 FragColor;
void main(){
   FragColor = texture2D(PreVoxelizedTexture, TextCoord);
}
```

Código 3.3: Copia de la textura de prevoxelización a la textura de voxelizado (*fragment shader*).

El *fragment shader* mostrado en el código 3.3 sirve para copiar una textura de entrada (*PreVoxelizedTexture*) a una textura colocada como blanco de despliegue (*FragColor*). Está asociada al *vertex shader* que se muestra en el código 3.2. Como entrada se recibe las coordenadas de textura que estará asociado a un cuadrado de dimensiones $[-1, 1]^2$, para poder obtener cada una de las coordenadas de la textura de entrada. Se utiliza para copiar la información de una escena prevoxelizada almacenada en una textura, a la textura a utilizar en la voxelización.

Creación de la textura de máscara de rayos

```
uniform usampler1D BitmaskXOR;

in vec2 pos;

layout(location = 0) out uvec4 FragColor;
void main(){
  FragColor = texelFetch1D(BitmaskXOR,int((pos.x + 1.0) / 2.0 * 128),0)
        ^
        texelFetch1D(BitmaskXOR,int((pos.y + 1.0) / 2.0 * 128 + 1),0);
}
```

Código 3.4: Creación de la máscara de bits (*fragment shader*).

Para la creación de las texturas que contienen las máscaras de bits de una dimensión se puede utiliza la CPU. Sin embargo, para acelerar el cálculo en la creación de la textura 2D se utiliza el *fragment shader* mostrado en el código 3.4. Se utiliza en conjunto con el *vertex shader* presentado en el código 3.1. Para poder tomar muestras sobre la textura 1D se despliega un cuadrado cuyas coordenadas están entre [-1,1] y posteriormente son transformadas en el espacio [0,127]. Se utiliza la función *texelFetch1D* para tomar las muestras en la textura con valores enteros.

3.4.2. *G-Buffer*

Para crear el *G-Buffer* es necesario utilizar múltiples objetivos de despliegue con el fin de almacenar las coordenadas de mundo de los objetos, los vectores normales y el valor de la *BRDF*, para cada uno de los fragmentos que son visibles desde el punto de vista de la cámara. Por lo tanto, es necesario enviar al *vertex shader* mostrado en el código 3.5 las coordenadas de objeto, los vectores normales y las coordenadas de textura de cada uno de los vértices. Además, es necesario poder manejar las diferentes matrices de transformaciones.

Cada una de estas coordenadas debe ser transformada antes de ser enviadas al *fragment shader*. Los vectores normales son multiplicados por la matriz *glNormalMatrix*, la cual es equivalente a la traspuesta de la inversa de la matriz de modelos (*glModelMatrix*); las coordenadas de texturas no son transformadas y la posición es multiplicada por la matriz de modelos para modificar las coordenadas a espacio de mundo.

```
in vec4 glVertex;
in vec2 glTexCoord;
in vec3 glNormal;

uniform mat3 glNormalMatrix;
uniform mat4 glModelViewProjectionMatrix;
uniform mat4 glModelMatrix;

out vec3 vNormal;
out vec4 vecPos;
out vec2 TextureCoord;

void main(){
  vNormal = glNormalMatrix * glNormal;
  vecPos = glModelMatrix * glVertex;
  TextureCoord = glTexCoord;
  gl_Position = glModelViewProjectionMatrix * glVertex;

}
```

Código 3.5: Obtención del *G-Buffer* (*vertex shader*).

Una vez que los datos son enviados al *fragment shader* que se muestra en el código 3.6, cada uno de los valores son almacenados en la textura correspondiente. Las coordenadas de mundo son desplegadas sin ningún procesamiento. Para los vectores normales es necesario normalizarlos antes de poder almacenarlos. El material de los objetos son enviados a través de una variable uniforme (*mat_diffuse*) y se divide entre π para obtener la *BRDF* en ese fragmento. Además, se puede agregar el valor de la textura del objeto si se posee.

```
#define PI 3.14159265359

uniform vec4 mat_diffuse;   //Propiedad difusa del material
uniform bool useTexture;    //Indica si se usa textura o no
uniform sampler2D text;     //Textura del objeto

in vec3 vNormal;
```

```
in vec4 vecPos;
in vec2 TextureCoord;

layout(location = 0) out vec4 WorldPosition;
layout(location = 1) out vec4 Normals;
layout(location = 2) out vec4 Material;
void main(){
  vec4 n = vec4(normalize(vNormal),0.0);

  WorldPosition = vecPos; //Buffer de coordenadas de mundo

  Normals = n; //Buffer de vectores normales

  vec3 texval = (useTexture)? pow(texture2D(text,TextureCoord.xy).rgb,
     vec3(2.2)) : vec3(1.0);

  //Buffer del material
  Material.rgb =  mat_diffuse.rgb
          / 3.14159265359
          * texval;
}
```

Código 3.6: Obtención del *G-Buffer* (*fragment shader*).

3.4.3. *Reflective Shadow Map*

El *RSM* mantiene la misma idea del *G-Buffer*, salvo que la escena es desplegada desde la posición de la luz en la dirección de la luz, por medio de la matriz *glModelViewProjectionMatrix*. Aquí se almacenarán las coordenadas de mundo, los vectores normales y la energía recibida en cada uno de los puntos visibles desde la luz. Cada luz tiene asociado un *RSM*. El *viewport* de la escena debe ser configurado según el tamaño de la resolución del *RSM*.

En el *vertex shader*, mostrado en el código 3.7, se recibe como entrada las coordenadas de objeto, las coordenadas de textura y los vectores normales de cada vértice. Además, se debe poseer la posición de la luz para poder realizar los cálculos de iluminación posteriores. Al *fragment shader* se le envían los vectores normales, las coordenadas de mundo, las coordenadas de textura del objeto y el vector de dirección de la luz en coordenadas de mundo. La posición del vértice se transforma para verse desde el punto de vista de la luz.

```
in vec4 glVertex;
in vec2 glTexCoord;
in vec3 glNormal;

uniform mat3 glNormalMatrix;
uniform mat4 glModelViewProjectionMatrix;
uniform mat4 glModelMatrix;
//Posición de la luz
uniform vec4 light_pos;

out vec3 vNormal;
```

```
out vec4 vecPos;
out vec3 light_dir;
out vec2 TextureCoord;

void main(){
  vNormal = glNormalMatrix * glNormal;
  vecPos = glModelMatrix * glVertex;
  //Dirección de la luz en espacio de mundo
  light_dir = light_pos.xyz - vecPos.xyz;
  TextureCoord = glTexCoord;
  gl_Position = glModelViewProjectionMatrix * glVertex;
}
```

Código 3.7: Obtención del *RSM* (*vertex shader*).

El *fragment shader* que se muestra en el código 3.8 se encarga de desplegar los vectores normales, las coordenadas de mundo y lo valores de iluminación. Para poder calcular los valores de iluminación, es necesario tener los valores de las diferentes propiedades de la luz, por lo que son enviados al *shader* como variables uniformes.

```
#define PI 3.14159265359

uniform vec4 mat_diffuse; //Propiedad difusa del material
uniform vec4 light_diffuse; //Color de la luz
uniform vec3 light_spotDirection; //Dirección
uniform float light_spotCosCutoff;//Ángulo de corte
uniform float light_spotExponent; //Exponente
uniform float light_spotConstant; //Atenuación constante
uniform float light_spotLinear;   //Atenuación linear
uniform float light_spotQuadratic;//Atenuación cuadrática
uniform sampler2D text;
uniform bool useTexture;

in vec3 vNormal;
in vec4 vecPos;
in vec2 TextureCoord;
in vec3 light_dir;

layout(location = 0) out vec4 WorldPosition;
layout(location = 1) out vec3 Normals;
layout(location = 2) out vec4 Flux;
void main(){
  vec3 n = normalize(vNormal);
  //Buffer de coordenadas de mundo
  WorldPosition = vec4(vecPos.xyz,abs(vecPos.z));
  Normals = n; //Buffer de vectores normales

  float dist = length(light_dir);
  vec3 lightVec = light_dir / dist;

  float NdotL = max(dot(n,lightVec),0.0);
```

```
  if(NdotL > 0.0){
    float SpotEffect = dot(normalize(light_spotDirection), -lightVec);

    if (SpotEffect > light_spotCosCutoff){

      SpotEffect = pow(SpotEffect, light_spotExponent);
      float att = SpotEffect / (light_spotConstant +  light_spotLinear *
          dist + light_spotQuadratic * dist * dist);

      vec4 texval = (useTexture)? pow(texture2D(text,TextureCoord.xy).
          rgba,vec4(2.2)) : vec4(1.0);
      vec4 material = mat_diffuse / PI * texval;

      Flux = att * ( material * light_diffuse * NdotL);
    }
  }
}
```

Código 3.8: Obtención del *RSM* (*fragment shader*).

La iluminación directa en la escena proviene de luces focales. Estas luces tienen la propiedad de que los rayos luminosos se encuentran restringidos a un cono. Las luces focales poseen una posición, una dirección (*light_ spotDirection*) y un ángulo de apertura (*light_ spotCosCutoff*). Además, posee un porcentaje de decadencia de la intensidad de la luz desde el centro hasta los bordes del cono. La atenuación de la luz es calculada utilizando la siguiente ecuación:

$$SpotEffect = (-lightVec \cdot light\_spotDirection) \tag{3.1}$$

$$att = \frac{SpotEffect}{light\_spotConstant + light\_spotLinear \times dist + light\_spotQuadratic \times dist^2} \tag{3.2}$$

Debido a que solo se utilizan superficies difusas, la contribución de la luz se calcula tomando en cuenta la propiedad difusa de la superficie (*material*), la componente difusa de la luz (*light_ diffuse*) y el producto punto de la normal de la superficie con la dirección de la luz (*NdotL*). Lo anterior puede resumirse en la siguiente ecuación:

$$Flux = att \times (material \times light\_diffuse \times NdotL) \tag{3.3}$$

3.4.4. *Shadow map*

La creación del *shadow map* es semejante al del *RSM*. La escena es proyectada desde la posición de la luz, pero solo los valores de profundidad son almacenados en una textura de profundidad, cuya componente es $GL\_DEPTH\_COMPONENT$. El *FrameBuffer Object* asociado a este *shader* no desplegará color en ninguna textura. Para cada luz en la escena está asociado un *shadow map*.

3.4.5. Despliegue de texturas atlas

Este shader es utilizado para desplegar las coordenadas de mundo de un objeto en su correspondiente coordenada de textura atlas, para posteriormente ser voxelizado. Para la creación de las texturas atlas es necesario colocar el *viewport* del tamaño de la resolución de las texturas atlas. Este valor es colocado dependiendo de la resolución necesaria para el objeto que se está procesando. El proceso de creación de texturas atlas se realiza para cada uno de los objetos de la escena.

Para el despliegue de la textura atlas, el *vertex shader* que se muestra en el código 3.9 debe tener como entrada las coordenadas de textura atlas del objeto y las coordenadas de espacio de objeto del objeto. Para poder hacer el despliegue de manera correcta, se envía al *fragment shader* las coordenadas de mundo del vértice y se coloca como posición del mismo las coordenadas de textura transformada al espacio [-1,1].

```
uniform mat4 glModelViewMatrix;

in vec4 glVertex;
in vec2 glTexCoord;

out vec3 vecPos;

void main(){
  vecPos = (glModelViewMatrix * glVertex).xyz;
  gl_Position = vec4((glTexCoord * 2.0) - vec2(1.0), 1.0 ,1.0);
}
```

Código 3.9: Despliegue de las coordenadas de mundo a la textura atlas (*vertex shader*).

Una vez en el *frament shader* presentado en el código 3.10, la posición en coordenadas de mundo del vértice recibidas como entrada, se colocan como color del fragmento.

```
in vec3 vecPos;

layout(location = 0) out vec4 FragColor;
void main(){
  FragColor = vec4(vecPos,1.0);
}
```

Código 3.10: Despliegue de las coordenadas de mundo a la textura atlas (*fragment shader*).

De esta manera, se obtendrá una imagen en donde cada píxel tendrá como color las coordenadas de mundo del objeto asignadas según la textura atlas.

3.4.6. Proceso de voxelización

Mediante *slicemap*

Para realizar la voxelización por medio de *slicemap* los objetos son desplegados normalmente y posteriormente los fragmentos son codificados en el volumen. El *viewport* debe ser

configurado para que contenga todo el volumen, esto es, del tamaño de la resolución del volumen. En el *vertex shader* mostrado en el código 3.11 se tiene como entrada solamente la coordenada de objeto del vértice y por medio de la matriz *viewProjMatrixVoxelCam* la coordenada es transformada a espacio del volumen. El valor de profundidad Z es transformado para distribuirlo uniformemente en el volumen.

```
uniform mat4 viewProjMatrixVoxelCam;

in vec4 glVertex;

out float Z;

void main(){
  gl_Position = viewProjMatrixVoxelCam * glVertex;

  //Se transforma la coordenada Z al espacio [0,1]
  Z = gl_Position.z * 0.5 + 0.5;
}
```

Código 3.11: Voxelización por medio del uso de *slicemap* (*vertex shader*).

Posteriormente, en el *fragment shader* presentado en el código 3.12 cada fragmento creado activa un bit del volumen en la posición correspondiente. Para ello, se busca la máscara de bits correspondiente en la textura *bitmask*, utilizando el valor de profundidad adecuado. En esta textura estarán almacenadas todas las máscaras que contienen todos los valores en 0 excepto el correspondiente al valor de profundidad.

```
uniform usampler1D bitmask;

in float Z;

layout(location = 0) out uvec4 FragColor;
void main(){
  FragColor = texture(bitmask,Z);
}
```

Código 3.12: Codificación del volumen (*fragment shader*).

Mediante texturas atlas

```
uniform mat4 viewProjMatrixVoxelCam;
uniform int res;

uniform sampler2D TextureAtlas;

in vec2 glVertex;

out float Z;

void main(){
```

```
  //Se obtiene la coordenada de mundo de la textura
  vec3 WorldCoord = texelFetch2D(TextureAtlas, ivec2(glVertex.xy * res)
     ,0).rgb;

  //Se transforma la coordenada de mundo al espacio del volumen
  gl_Position = viewProjMatrixVoxelCam * vec4(WorldCoord,1.0);

  //Se transforma la coordenada Z al espacio [0,1]
  Z = gl_Position.z * 0.5 + 0.5;
}
```

Código 3.13: Voxelización a través del uso de texturas atlas (*vertex shader*).

Esta voxelización funciona de manera similar al *slicemap*, la única diferencia es que en el *fragment shader* mostrado en el código 3.13 las coordenadas de mundo son obtenidas a través de la textura atlas *TextureAtlas*. Para ello, se debe desplegarse un vértice por cada píxel válido de la textura, cuya coordenada sirve para obtener muestras sobre la textura usando *texelFetch2D*. Posteriormente se realizan los mismos pasos utilizados en *slicemap*.

Creación de la jerarquía *mipmap*

```
uniform usampler2D voxelTexture; //Textura con el volumen
uniform int level; //Del nivel de la jerarquía que se va a leer
uniform float inverseTexSize; // (1.0 / Resolución de la jerarquía)

in vec2 TextCoord;
in vec4 pos;

layout(location = 0) out uvec4 FragColor;
void main(){
  //Se calculan las coordenadas de los vecinos
  vec2 offset1 = vec2(inverseTexSize, 0.0);   //Derecha
  vec2 offset2 = vec2(0.0, inverseTexSize);   //Arriba
  vec2 offset3 = vec2(inverseTexSize, inverseTexSize); //Arriba y derecha
  vec2 coord = TextCoord - vec2(inverseTexSize/2.0, inverseTexSize/2.0);

  /*Se obtiene el valor de textura de los vecinos en el nivel
     correspondiente*/
  uvec4 val1 = texture2DLod(voxelTexture, coord, level);
  uvec4 val2 = texture2DLod(voxelTexture, coord+offset1, level);
  uvec4 val3 = texture2DLod(voxelTexture, coord+offset2, level);
  uvec4 val4 = texture2DLod(voxelTexture, coord+offset3, level);

  //Se realiza la operación lógica Or entre los 4 valores
  FragColor = val1 | val2 | val3 | val4;
}
```

Código 3.14: Creación de la jerarquía *mipmap* (*fragment shader*).

Cuando la escena voxelizada es obtenida por medio de *slicemap* o de las texturas atlas, el volumen es almacenado en una textura 2D (*voxelTexture*). Para crear la jerarquía *mipmap* a

partir de esta textura se utiliza el *fragment shader* que se muestra en código 3.14. Se utiliza en conjunto con el *vertex shader* del código 3.2. La textura original representará el nivel 0 de la jerarquía y dada una resolución X del volumen se tendrán $Y = \log_2(X)$ niveles en la jerarquía. En cada nivel de la jerarquía es necesario utilizar el *shader* para crear el nivel siguiente. Por ejemplo, para calcular el nivel *i* de la jerarquía, a la variable *level* se le asigna *i-1* para acceder al nivel anterior, y el *shader* despliega información en el nivel *i* de la jerarquía. Para ello, se obtiene el valor de 4 píxeles en el nivel anterior utilizando *texture2DLod*, y por medio de la operación OR lógico se obtiene el valor del píxel en el nivel actual.

3.4.7. Iluminación directa

Para el cálculo de la luz directa se utiliza el *fragment shader* mostrado en el código 3.15, en conjunto con el *vertex shader* presente en el código 3.2. La iluminación directa se calcula desde el punto de vista de la cámara, por lo que puede utilizarse la información almacenada en el *G-Buffer*, en vez de desplegar toda la escena otra vez. Por ello, se utilizarán las texturas que almacenan las coordenadas de mundo (*position*), los vectores normales (*normal*) y la *BRDF* de las superficies (*material*). Como se va a calcular la iluminación, se deben enviar las propiedades de la luz a través de variables de tipo uniformes al *shader*, además de enviar el *shadow map* como una textura (*ShadowMap*).

Con la coordenada de textura enviada al *fragment shader* se obtienen los valores correctos a través de las texturas del *G-Buffer*, almacenando las coordenadas de mundo en *vposition*, el vector normal en *vnormal* y el valor de la *BRDF* de la superficie en *vmaterial*. Con estos valores es posible calcular la contribución de la luz directa en este píxel de manera similar a como se calcula la iluminación para el *RSM*.

Para el cálculo de las sombras es necesario proyectar la coordenada de mundo al espacio del RSM, ya que el *shadow map* se encuentra en este espacio. Por ello, la matriz de transformaciones del *RSM* es enviada al *shader* en la variable *glRSMReprojectMatrix*. Debido a la baja resolución de las sombras en el *RSM*, se hace uso del filtro propuesto por Bunnell y Pellacini [58]. En este filtro se obtienen los valores de sombra de 16 valores alrededor del píxel y se promedian sus contribuciones, logrando el efecto de una sombra suave.

Debido a que la escena puede poseer varias luces, se hace uso de este *shader* para sumar las contribuciones de las luces en la textura final. Para cada uno de los despliegues se modifican los valores uniformes de las propiedades de la luz.

```
in vec4 vecPos;        //Coordenadas de mundo
in vec2 TextCoord;       //Coordenadas de textura

//G-buffers
uniform sampler2D position;    //Posición
uniform sampler2D normal;    //Normal
uniform sampler2D material;    //BRDF
uniform sampler2DShadow ShadowMap;

uniform vec2 texmapscale;
```

```
uniform mat4 glRSMReprojectMatrix;

//Propiedades de la luz
uniform vec3 lightPos;      //Posición
uniform vec3 lightColor;    //Color
uniform vec3 lightDirection;  //Dirección
uniform float lightCosCutoff; //Ángulo de corte
uniform float lightExponent;  //Exponente
uniform float lightConstant;  //Atenuación constante
uniform float lightLinear;    //Atenuación linear
uniform float lightQuadratic; //Atenuación cuadrática

uniform bool softShadow;    //Calidad de la sombra

layout(location = 0) out vec4 DirectLight;

float offset_lookup(vec4 loc, vec2 offset){
  return textureProj(ShadowMap, vec4(loc.xy + offset * texmapscale *
                        loc.w, loc.z - 0.01, loc.w));
}

void main(){
  DirectLight = vec4(0.0);

  vec3 vposition = texture2D(position,TextCoord).rgb;
  vec3 vnormal = texture2D(normal,TextCoord).rgb;
  vec3 vmaterial = texture2D(material,TextCoord).rgb;

  if(vposition.z < 100.0){
    DirectLight = vec4(0.0,0.0,0.0,0.0);

    vec3 light_dir = lightPos - vposition;
    float dist = length(light_dir);
    light_dir = light_dir/dist;

    float NdotL = max(dot(vnormal,light_dir),0.0);

    if(NdotL > 0.0){
      float SpotEffect = dot(normalize(lightDirection), -light_dir);

      if (SpotEffect >= lightCosCutoff){
        SpotEffect = pow(SpotEffect, lightExponent);
        float att = SpotEffect / (lightConstant + lightLinear * dist +
            lightQuadratic * dist * dist);

        DirectLight = vec4(att * (vmaterial * lightColor * NdotL),1.0);
      }
    }

    vec4 shadowProjectedCoord = glRSMReprojectMatrix * vec4(vposition
        ,1.0);
    float shadow = 1.0;
```

```
    if(shadowProjectedCoord.w > 0.0){
      //Se obtienen muestras de la sombra
      shadow = 0.0;
      float x, y;
      float limit = (softShadow)?1.5:0.0;
      int num = 0;

      for (y = -limit; y <= limit; y += 1.0){
        for (x = -limit; x <= limit; x += 1.0){
          shadow += offset_lookup(shadowProjectedCoord, vec2(x, y));
          ++num;
        }
      }
       shadow /= num;
     }
     DirectLight.rgb *= shadow;
  }
}
```

Código 3.15: Cálculo de la iluminación directa (*fragment shader*).

3.4.8. Iluminación indirecta

Cálculo de la intersección del rayo

Para realizar el cálculo de la iluminación indirecta es necesario lanzar un rayo por cada píxel de la imagen final y calcular su intersección con la escena voxelizada. En el código 3.16 se muestra el método principal del *fragment shader* que se encarga de calcular la intersección. Primero, se extrae el origen del rayo de las coordenadas de mundo almacenadas en el *G-Buffer*. Luego, la dirección del rayo se calcula tomando muestras en el hemisferio de la superficie. El origen es desplazado en dirección de la normal para evitar que intersecte con la misma geometría. Una vez calculado el origen y la dirección del rayo, se calcula el punto final del rayo, el cual dependerá de la longitud establecida.

Para poder recorrer el volumen es necesario transformar las coordenadas del rayo a *NDC*, lo cual se realiza por medio de la matriz *WordlToNDCMatrix*. Para poder ir avanzando en la dirección del rayo se utlizan las variables *TNear* y a *TFar*, y cuando *TNear* sea mayor *TFar* la longitud restante del rayo será 0.

Una vez que se tiene preparado el rayo se procede a realizar el recorrido a través del volumen. Las condiciones de parada del recorrido son que la longitud del rayo se acabe, se encuentre una intersección o se haga un número determinado de pruebas de intersección (*steps*). Para cada iteración el rayo es proyectado sobre el volumen, calculando el píxel del volumen en el que se encuentra el origen. En este píxel se puede crear una caja envolvente que contenga la columna de bits de ese píxel. Con esta caja se puede calcular un valor de salida y uno de entrada del rayo en la caja. Posteriormente con estos valores es posible calcular la intersección del rayo con los valores que están contenidos en el píxel. Si se encuentra

una intersección, se necesita verificar si también la hay en un nivel menor de la jerarquía. Solamente se considera que hay intersección si esta se encuentra en el nivel 0 de la jerarquía. En caso que no haya intersección, se debe avanzar el rayo con el punto de salida de la caja envolvente del píxel calculado anteriormente, actualizando el *TNear* y se incrementa un nivel en la jerarquía. Este avance del rayo se realiza con un desplazamiento para evitar errores de punto flotante.

```
#define PI 3.14159265359

in vec4 vecPos;
in vec2 TextCoord;

uniform usampler2D allBitMasks; //Textura con todas las máscaras de bit
uniform usampler2D Voxel;   //Textura que almacena el volumen

//G-Buffers
uniform sampler2D positionGBuffer;  //Posición
uniform sampler2D normalGBuffer;  //Normal
uniform sampler2D random;     //Una textura aleatoria

uniform mat4 WordlToNDCMatrix;    //Coordenadas de mundo a NDC
uniform mat4 UnitToWorldCoordMatrix;//NDC a coordenadas de mundo

uniform int max_lvl;          //Nivel máximo de la jerarquía
uniform int steps;          //Número de pasos
uniform float ray_lenght;     //Longitud del rayo
uniform vec3 VoxelizeDirection;   //Dirección de la voxelización
uniform float VoxelDiagonal;    //Tamaño de la diagonal de un vóxel
uniform vec2 SamplingSequence[128]; //Arreglo con muestras
uniform int ray_number;       //El número del rayo
uniform float contribution;     //Contribución del rayo
uniform float NormalOffset;     //Desplazamiento del rayo con la normal
uniform float RayOffsetStart;   //Desplazamiento del comienzo del rayo

//Definición del rayo
vec3 dir;
vec3 origin;

layout(location = 0) out vec3 Hit;
layout(location = 1) out vec3 directionalOclusion;
void main(){
  vec4 Position = texture2D(positionGBuffer,TextCoord);
  Hit = vec3(0.0,0.0,100.0);
  directionalOclusion = vec3(0);

  if(Position.z < 100.0){
    vec3 normal = texture2D(normalGBuffer,TextCoord).xyz;
    normal = normalize(normal);

    //Se generan 2 vectores ortogonales con la normal
    vec3 R = vec3(0.0,0.0,1.0);
```

```
if(abs(normal.x) < 0.001 && abs(normal.y) < 0.001)
  R = vec3(0.0,1.0,0.0);
vec3 T = normalize(cross(normal,R));
vec3 B = cross(T,normal);

//Se obtiene un valor aleatorio de la textura
vec2 ran = texelFetch2D(random, ivec2(mod(ivec2((TextCoord.st)*1024),
    ivec2(16))), 0).rg;
float r1 = fract(SamplingSequence[ray_number].x + ran.x);
float r2 = fract(SamplingSequence[ray_number].y + ran.y);

//Se obtiene un valor aletorio en el hemisferio
float r = 2 * PI * r2;
float sq2 = sqrt(r1);
float x = cos(r) * sq2;
float y = sin(r) * sq2;
float z = sqrt(max(0.0, 1.0 - x*x - y*y));

//Se proyecta ese punto en la dirección de la normal
vec3 ray = normalize(x*B + y*T + z*normal);

//Desplazamiento para prevenir el auto sombreado
//Se mueve el rayo al menos el tamaño de la diagonal del vóxel
float soffset = RayOffsetStart * VoxelDiagonal / dot(ray, normal);

if(soffset <= ray_lenght){
  //Se construye el origen y el final del rayo
  origin = Position.xyz + NormalOffset * normal + ray * soffset;
  vec3 end = Position.xyz + NormalOffset * normal + ray * ray_lenght;

  //Solo se procede si el rayo tiene una longitud aceptable
  if(distance(origin,end) > VoxelDiagonal*1.5){
    //Transformar las coordenadas a NDC
    origin  = (WordlToNDCMatrix * origin).xyz;
    dir   = (WordlToNDCMatrix * end).xyz - origin;

    //Un pequeño desplazamiento para el recorrido de la jerarquía
    float offset = (0.25 / (1 << (max_lvl -1))) / length(dir);
    float TNear = 0.0, TFar;

    //Se calcula la intersección del rayo en el máximo nivel
    if(!IntersectBox(vec3(0.0),vec3(1.0),TFar))
      TFar = 1.0;

    bool intersection = false;
    uvec4 Bitmask = uvec4(0.0);

    //Colocar el origen actual con el origen del rayo
    vec3 currentTNear = origin;
    float currentTFar;
    int lvl = min(3 , max_lvl - 2);
```

```
        //Se calcula la intersección del rayo con el volumen
        for(int i=0; (i < steps) && (TNear <= TFar) && (!intersection);
            ++i ){
          currentTFar = 1.0;

          if(IntersectWithVolume(lvl, currentTNear, currentTFar, Bitmask)
              ){
            //Si hay intersección se debe probar si hay intersección un
            //nivel menor de la jerarquía. Solo ocurre intersección si
            //hay intersección en el nivel 0 de la jerarquía
            intersection = (lvl == 0);
            -- lvl;
          }else{
            //Si no hay intersección el rayo es trasladado y se avanza
            //al nivel anterior
            TNear = currentTFar + offset;
            currentTNear = origin + TNear * dir;
            ++ lvl;
          }
        }

        //Si hay intersección se calcula cual fue el bit intersectado
        if(intersection){
          CalculatePosition(bitPosition)

          //Se obtiene la posición del hit en NDC, para luego
          //transformarla a coordenadas de mundo y almacenar
          //el resultado
          currentTNear.z = (float(127.0 - bitPosition) + 0.5)/128.0;
          Hit = (UnitToWorldCoordMatrix * vec4(currentTNear,1.0)).rgb;
        }

        //Se calcula la oclusión direccional
        vec3 senderRadiance = vec3(0.2);
        float occlusionStrength = 1.1;
        float receiverGeometricTerm = max(0.0, dot(ray, normal));
        float visibility = 1.0 - (intersection ? occlusionStrength: 0.0);
        directionalOclusion = contribution * visibility *
            receiverGeometricTerm * senderRadiance;
      }
    }
  }
}
```

Código 3.16: Método principal para el cálculo de la intersección del rayo (*fragment shader*).

En el código 3.17 se muestra el método *IntersectWithVolume*, el cual se encarga de proyectar el rayo en la jerarquía y calcular la caja envolvente del píxel. Primeramente se debe calcular el píxel activo de la jerarquía. Como el rayo se encuentra en coordenadas *NDC*, esto se realiza multiplicando su posición actual (*posTNear*) por el alto y el ancho del volumen en el nivel actual de la jerarquía. Posteriormente, se obtiene la caja envolvente dividiendo la

coordenada del píxel entre el alto y el ancho de volumen en el nivel actual. El valor de entrada del rayo con la caja envolvente está definido por *posTNear.z* y el valor de salida es almacenado en *zFar*, después de ser calculado el valor de *tFar* con el método *IntersectBoxOnlyTFar*. Con el valor de salida y entrada del rayo en la caja, se obtiene la máscara de bits del rayo, tomando una muestra de la textura 2D *allBitMasks*. Esta textura tiene ya precalculado todas las máscaras posibles dado un valor de entrada y un valor de salida. Esta máscara es pasada junto con la coordenada del píxel y el nivel de la jerarquía actual, al método *intersectBits*.

```
bool IntersectWithVolume(in int level, in vec3 posTNear, inout float tFar
    , out uvec4 intersectionBitmask){
  //Calcula las coordenadas del rayo en píxeles
   float res = float(1 << (max_lvl - 1 - level));
   ivec2 pixelCoord = ivec2(posTNear.xy * res);
   vec2 voxelWH = vec2(1.0) / res;

  //Se calcula el AABB perteneciente a la posición del píxel
  vec2 box_min = pixelCoord * voxelWH;

  //Se computa la intersección con la caja envolvente
  tFar = IntersectBoxOnlyTFar(
      vec3(box_min, 0.0),
      vec3(box_min + voxelWH, 1.0));

  //Se calcula la salida del rayo con la caja envolvente
  float zFar = tFar*dir.z + origin.z ;

  /*Con la entrada y la salida de la caja envolvente
      se calcula una máscara de bits con 1's entre
      estos dos valores y 0's los demás bits, y se verifica
      si intersecta con la columna de bits del volumen*/
  return intersectBits(
      texture2D(allBitMasks,
      vec2(min(posTNear.z, zFar), max(posTNear.z, zFar))),
      pixelCoord, level, intersectionBitmask);
}
```

Código 3.17: Intersección del rayo con la jerarquía.

El método *intersectBits* mostrado en el código 3.18 toma como entrada la máscara de bits del rayo y procede a calcular la máscara de bits del píxel, obteniendo el valor del volumen (*Voxel*) y en el píxel requerido (*texel*), en el nivel actual de la jerarquía (*level*). Luego procede a realizar la operación lógica AND entre estas dos máscaras y procederá a almacenar el valor en la variable *intersectionBitmask*. Si el valor de *intersectionBitmask* es diferente de 0 significa que hubo intersección. El método *IntersectWithVolume* posteriormente se encargará de notificar al método principal si hubo intersección o no.

El valor presente en la variable *intersectionBitmask* es almacenado en el método principal en la variable *bitPosition*, y con él se procede a calcular el primer bit encendido de la manera mostrada en el código 2.1 en el capítulo 2. De esta manera, se obtiene el valor de profundidad de la intersección. Transformando el valor almacenado en *currentTNear* de coordenadas *NDC*

a coordenadas de mundo, se obtiene el punto de la intersección del rayo en coordenadas de mundo y se despliega en la textura *Hit*.

```
bool intersectBits(in uvec4 bitRay, in ivec2 texel,
          in int level, out uvec4 intersectionBitmask){

  intersectionBitmask = (bitRay & texelFetch2D(Voxel, texel, level));

  return (intersectionBitmask != uvec4(0));
}
```

Código 3.18: Cálculo de la máscara de bits de la intersección.

Adicionalmente, se calcula la oclusión direccional. La oclusión direccional fue propuesta por Ritschel et al. [59] y representa un complemento a la solución de la iluminación global. Para el cálculo de este valor se lanzan rayos alrededor del hemisferio de una superficie, si el rayo no intersecta con geometría se toma un valor proveniente del ambiente (*senderRadiance*). En caso de que haya intersección, el valor tenderá a oscurecer el píxel. La prueba de intersección realizada anteriormente se puede aprovechar para calcular este valor. De esta manera, superficies que no reciban luz indirecta de ningun rayo, podrán ser iluminadas por la luz ambiental.

Obtención del valor de iluminación

Una vez obtenido el punto de intersección del rayo con el *shader* anterior, se procede a proyectar ese punto en el *RSM*, mediante el uso de la matriz *RSMMatrix*, para calcular la iluminación indirecta. Este proceso se realiza con el *fragment shader* mostrado en el código 3.19. La textura *HitBuffer* almacena todas las intersecciones de los rayos para cada uno de los píxeles de la imagen. Con ella es posible obtener el punto proyectado en el *RSM* y es almacenado en la variable *coord*. Con esta variable se obtiene el vector normal, las coordenadas de mundo y el color de los *buffers* creados por el *RSM*. Por otro lado, *RayOriginBuffer* es la textura obtenida con las coordenadas de mundo del *G-Buffer*, las cuales representan el origen del rayo. Una vez obtenida esta información, se determina si el punto de intersección realmente es visible desde la luz, utilizando el umbral que se explicó en el capítulo 2, en la sección 2.3.2. Si el punto de intersección es visible desde la luz, se coloca el valor de iluminación del *RSM* como la iluminación directa para este punto.

```
in vec4 vecPos;   //Coordenadas de mundo del fragmento
in vec2 TextCoord;  //Coordenadas de textura del fragmento

//Textura que almacena el final del rayo
uniform sampler2D HitBuffer;

//Textura que almacena el origen del rayo
uniform sampler2D RayOriginBuffer;

//Buffers del RSM
uniform sampler2D positionRSM;  //Posición
uniform sampler2D normalRSM;  //Normal
```

```
uniform sampler2D colorRSM;   //Color

//Matriz para reproyectar el punto hacia el espacio visto por el RSM
uniform mat4 RSMMatrix;

uniform float voxelDiagonal;
uniform float pixel_sideDivzNear;
uniform float contribution;   //Contribución de la luz indirecta
uniform vec3 lightDir;      //Dirección de la luz
uniform float thresholdFactor;

layout(location = 0) out vec4 FragColor;
void main(){
  //Obtención de origen del rayo
  vec3 rayOrigin = texture2D(RayOriginBuffer,TextCoord).xyz;
  //Obtención del final del rayo
  vec3 rayEnd =  texture2D(HitBuffer,TextCoord).xyz;
  //Obtención de la dirección del rayo
  vec3 raydir = normalize(rayOrigin - rayEnd);
  //Proyectar el final del rayo en el espacio del RSM
  vec4 coord = RSMMatrix * vec4(rayEnd,1.0);

  //Coordenadas de mundo del final del rayo en el RSM
  vec4 position = texture2DProj(positionRSM, coord);

  //Vector normal en la posición final del rayo en el RSM
  vec3 normal = texture2DProj(normalRSM, coord).xyz;

  //Iluminación directa en la posición final del rayo en el RSM
  vec4 color = texture2DProj(colorRSM, coord);


  float lightZ = position.w;
  float pixelSide = pixel_sideDivzNear * lightZ;
  float front_face = dot(normal, raydir);
  float cosAlpha = max(dot(normal, lightDir),0.001);


    /*El umbral de comparación de la distancia entre el final del rayo y
        la posición del RSM debe ser ajustada a la discretización de la
        voxelización (definida por el tamaño de la diagonal del voxel),
        el tamaño de un píxel en el RSM y la orientación de la superficie
           */


  if( rayEnd.z < 100.0  &&    /*Si hay una intersección*/
    color.z < 100.0   &&    /*Si el material es válido en esa posición*/
    coord.w > 0.0   &&    /*Si el final del rayo es proyectado al RSM*/
    front_face >= 0.0 &&    /*Si la superficie es una cara frontal*/
    distance(rayEnd,position.xyz) < min( 4.0 * voxelDiagonal, max(
       voxelDiagonal,pixelSide/cosAlpha)) * thresholdFactor){
```

```
        FragColor = contribution * color;

  }else{

        FragColor = vec4(0.0);

  }
}
```

Código 3.19: Cálculo de la luz indirecta (*fragment shader*).

Este *shader*, en conjunto con el *shader* para el cálculo de la intersección del rayo mostrado en el código 3.16, deben utilizarse para cada uno de los rayos que deban lanzarse en una pasada hacia la escena. Por ello, el número de rayos y el *viewport* de la imagen final son factores que inciden en el desempeño del algoritmo.

Combinación de la oclusión direccional

Cuando se obtuvo el valor de iluminación indirecta se almacenó los valores de la oclusión direccional en una textura diferente. Por lo tanto, se necesitan sumar estos valores con los valores de la iluminación indirecta. El *fragment shader* mostrado en el código 3.20, en conjunto con el *vertex shader* presentado en el código 3.2, se encargan de realizar esta tarea. Para ello, se coloca como objetivo de despliegue la textura que contiene la iluminación indirecta y se debe preparar el *pipeline* gráfico para poder mezclar los valores de oclusión direccional (*directionalOclution*), con los ya presentes en la textura objetivo *IndirectIlumination*. Debido a que los valores de oclusión direccional pueden ser negativos en presencia de una fuerte oclusión, es necesario transformar los valores negativos a 0.

```
//Textura con los valores de oclusión direccional
uniform sampler2D directionalOclution;

in vec4 vecPos;   //Coordenadas de mundo del fragmento
in vec2 TextCoord;  //Coordenadas de textura del fragmento

layout(location = 0) out vec3 IndirectIlumination;
void main(){
  IndirectIlumination = max(vec3(0.0),texture(directionalOclution,
     TextCoord).rgb);
}
```

Código 3.20: Combinación de la oclusión direccional con la iluminación indirecta (*fragment shader*).

3.4.9. Filtro para la iluminación indirecta

Debido a que la iluminación indirecta se calcula en espacio de imagen y con un número limitado de rayos, la solución resultante posee mucho ruido. Para poder eliminar este ruido es necesario la aplicación de un filtro. En el *fragment shader* mostrado en el código 3.21, se

aplica el filtro espacial presentado por Herzog et al. [60]. Este filtro es usado para interpolar valores de píxeles cercanos, tomando en cuenta las coordenadas de mundo y los vectores normales que están almacenados en el *G-Buffer*. De esta manera, la contribución de los valores es dependiente de la cercanía de los píxeles en espacio de mundo y de la orientación de las superficies. El valor filtrado $h(i)$ para un píxel i puede ser calculado de la siguiente manera:

$$h(i) = \frac{1}{\sum \omega_s} \sum_{j \epsilon N\{i\}} \omega_s(i,j) \cdot l(j) \tag{3.4}$$

Donde $N\{i\}$ son los vecinos alrededor de i, j es el índice del píxel vecino y $\omega_s(i,j)$ es la contribución del píxel i al píxel j dependiente de la geometría. Esta contribución puede ser calculada de la siguiente manera:

$$\omega_s(i,j) = n(max(0, 1 - (\overrightarrow{n_i} \cdot \overrightarrow{n_j}))^2, \sigma_n^2) \times d(|z_i - z_j|^2, \sigma_z^2) \times k(i,j) \tag{3.5}$$

Este peso tiene un factor dependiente de la orientación n, uno dependiente de la distancia d y uno dependiente de el espacio de imagen k. Los valores de σ_n^2 y de σ_z^2 son dependientes de la escena. Para n, d y k se utiliza el siguiente kernel:

$$g(x, \sigma) = max(\epsilon, 1 - (\frac{x^2}{\sigma}))^3 \tag{3.6}$$

En este *fragment shader* se aplica esta ecuación con el fin de poder eliminar el ruido en la textura de la iluminación indirecta (*inputText*), con lo que puede utilizarse una menor cantidad de rayos obteniendo un resultado aceptable.

```
uniform sampler2D inputText;  //Textura con iluminación indirecta
uniform sampler2D normal;   //Vectores normales G-Buffer
uniform sampler2D position;   //Coordenadas de mundo G-Buffer

uniform float normalTolerance;  //Tolerancia de la normal al cuadrado
uniform float distanceTolerance;//Tolerancia de la distancia al cuadrado
uniform int radius;       //Radio del filtro
uniform float pixelDiagonal;
uniform vec2 sampleDisplacement;

in vec4 vecPos;         //Coordenadas de mundo del vértice
in vec2 TextCoord;        //Coordenadas de textura del vértice

layout(location = 0) out vec3 indirLight;

float kernel(in float x, in float rho){
  float aux = max(0.01,1.0 - (x * x / rho));
  return aux * aux * aux;
}

float k(in float dist){
  float aux = max(0.01,1.0 - (dist / pixelDiagonal));
  return aux * aux * aux;
}
```

```
void main(){
  vec2 textcoord = TextCoord;
  vec3 actualposition = texture(position,textcoord).rgb;

  if(actualposition.z < 100.0){
    vec3 actualnormal = texture(normal,textcoord).rgb;
    float weight;
    vec3 totalWeight = vec3(0.0);
    float sumWeight = 0.0;
    vec2 sampleCoord;

    for(int i = -radius;i <= radius; ++i){
      sampleCoord =  textcoord + i * sampleDisplacement;
      weight =  kernel(max(0.0,1.0 - dot(actualnormal, texture(normal,
          sampleCoord).rgb)),normalTolerance) *
             kernel(distance(actualposition, texture(position, sampleCoord
                 ).rgb),distanceTolerance) *
             k(i);
      sumWeight += weight;
      totalWeight += weight * texture(inputText,sampleCoord).rgb;
    }

    if(sumWeight > 0.0)
      indirLight = totalWeight / sumWeight;
  }
}
```

Código 3.21: Filtro para eliminar el ruido en la iluminación indirecta (*fragment shader*).

3.4.10. Combinación de soluciones

Una vez obtenidas tanto la iluminación directa como la iluminación indirecta, sus correspondientes texturas son utilizadas para obtener la solución final. La combinación de las soluciones se realiza con el *fragment shader* mostrado en el código 3.22, en combinación con el *vertex shader* que se muestra en el código 3.2. El *shader* recibe como entradas las texturas de coordenadas de mundo y de la *BRDF* del *G-Buffer*, además de las texturas con contienen la iluminación directa y la iluminación indirecta. Se busca la coordenada de mundo, vista desde el píxel haciendo uso del *G-Buffer*. Si la posición es válida, se procede a sumar la luz indirecta y la luz directa. Ambos valores son multiplicados por un factor de contribución y la luz indirecta es multiplicada por la *BRDF* de la superficie vista desde el píxel. Finalmente, el resultado es codificado con un operador de corrección gama, el cual resaltará más las superficies iluminadas de la escena.

```
in vec4 vecPos;   //Coordenadas de mundo
in vec2 TextCoord;  //Coordenadas de textura

uniform sampler2D position; //Coordenadas de mundo del G-Buffer
uniform sampler2D material; //BRDF del G-Buffer
```

```
uniform sampler2D IIlu;   //Texturas con iluminación indirecta
uniform sampler2D DIlu;   //Texturas con iluminación directa

uniform float dlcont;
uniform float idlcont;

uniform bool addDirect;
uniform bool addIndirect;

layout(location = 0) out vec3 FinalResult;
void main()
{
  vec3 vposition = texture(position,TextCoord).rgb;
  vec3 vmaterial = texture(material,TextCoord).rgb;

  if(vposition.z < 100.0){

    FinalResult = vec3(0.0);

    if(addDirect){
      //Se suma la luz directa
      FinalResult +=  dlcont * texture(DIlu,TextCoord).rgb;
    }

    if(addIndirect){
      //Se coloca la luz indirecta
      //Luz indirecta = contribución indirecta * BRDF difusa
      FinalResult +=  idlcont *  max(vec3(0.00001), texture(IIlu,
          TextCoord).rgb * vmaterial);
    }

    //Corrección gama para colocar la imagen más brillante
    FinalResult = pow(FinalResult, vec3(1.0/2.0));
  }
}
```

Código 3.22: Combinación de la luz indirecta y la luz directa (*fragment shader*).

En la figura 3.3 se muestran algunas de las texturas resultado más importantes en las diferentes etapas del algoritmo. Para la etapa 1 se muestra el G-Buffer con sus coordenadas de mundo, vectores normales y las *BRDF* de la superficie. Además, se encuentra el *shadow map* y el *reflective shadow map* con sus coordenadas de mundo, vectores normales y la iluminación directa. En la etapa 2 se observan las texturas atlas de la escena (izquierda) y la textura con la escena voxelizada (derecha). Las texturas creadas para el cálculo de la luz directa (derecha) y la luz indirecta (izquierda) son mostradas en la etapa 3. Para la luz indirecta se muestra el *buffer* de intersección de los últimos rayos lanzados, la oclusión direccional, y la luz indirecta ya combinada con la oclusión direccional y filtrada. Por último, en la etapa 4 se observa el resultado final al combinar la contribución de la luz directa y la luz indirecta.

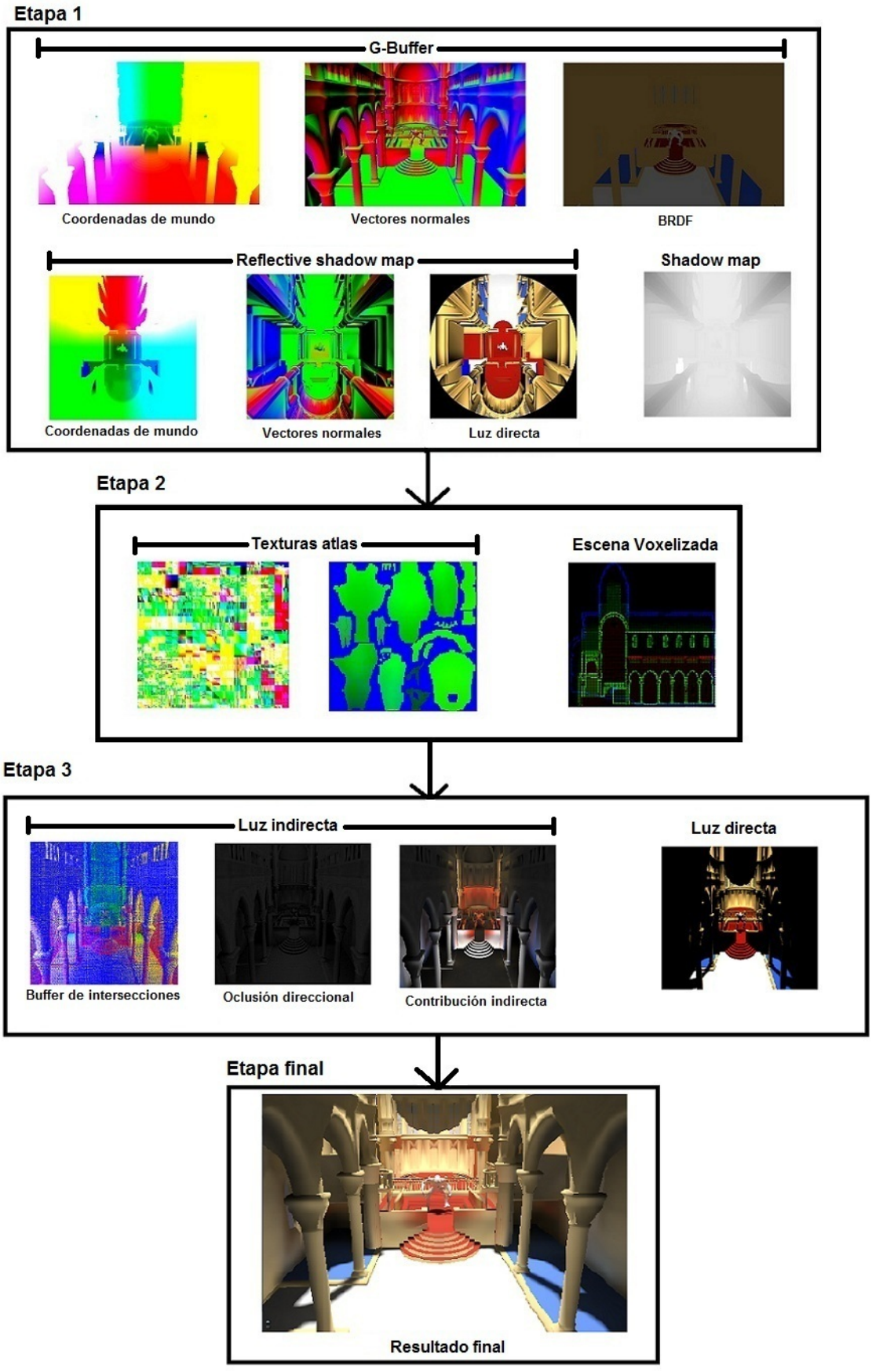


Figura 3.3: Resultado de las diferentes etapas del algoritmo.

Capítulo 4

Pruebas y resultados

En este capítulo se analizarán los resultados obtenidos al aplicar diferentes pruebas en la implementación del algoritmo realizada para este trabajo especial de grado. Debido a que la implementación esta orientada mayormente al uso de la GPU para realizar los cálculos, las mediciones de tiempo se efectúan en tiempos de la GPU y no de la CPU. Primero se detallarán las escenas y los ambientes utilizados para realizar las pruebas. Luego se estudiarán diferentes factores que afectan la voxelización, como lo son el tamaño del volumen, el tamaño de las texturas atlas y la comparación del uso de las texturas atlas y *slicemap*. Posteriormente, se estudiarán diferentes factores que afectan el rendimiento del cálculo de la iluminación indirecta. Finalmente, se analizará el desempeño de la implementación para diferentes tamaños de la ventana.

4.1. Ambiente de pruebas

Una escena contiene dos tipos de objetos: los dinámicos y los estáticos. Los estáticos son cargado en la aplicación como modelos en formato OBJ, representando la estructura básica de la escena. Los dinámicos son cargados como modelos en formato MD2. Para la prueba de la implementación, se diseñó 3 escenas de diferentes tamaño: pequeña (figura 4.1(a)), mediana (figura 4.1(b)) y grande (figura 4.1(c)).

(a) Caja de colores (b) Catedral de Sibenik (c) Sponza de Crytek

Figura 4.1: Escenarios de prueba.

En la tabla 4.1 se muestra el nombre, los modelos contenidos, la cantidad de vértices y la cantidad de triángulos de la escenas diseñadas. El sponza de Crytek [61], debido a su gran cantidad de geometría, fue divido en dos partes: la estructura del edificio y los objetos dentro del mismo. En la tabla solo se toma en cuenta los modelos en formato OBJ, ya que poseen mayor cantidad de geometría y un nivel de detalle más alto que los objetos dinámicos. Los objetos dinámicos pueden ser agregados a cualquiera de la escenas.

Nombre	Modelos	Nro. de Vértices	Nro. deTriángulos
Escena pequeña (E1)	Caja de colores	38	32
Escena mediana (E2)	Catedral de Sibenik	79.116	79.851
Escena grande (E3)	Sponza de Crytek: Estructura Objetos	 60.094 83.438	 112.257 147.333

Tabla 4.1: Escenas para la realización de pruebas.

Las pruebas se realizaron en 2 ambientes diferentes. En ambos se utilizó una PC convencional con las siguientes especificaciones: Windows 7 de 64 bits, procesador Intel Core i3 de 3 GHz y OpenGL versión 3.3. La diferencia entre los ambientes es la tarjeta de video. En el ambiente 1 (A1) se utilizó una Nvidia GeForce GTS 450, mientras que en el ambiente 2 (A2) se usó una Nvidia GeForce GTX 470. En la tabla 4.2, se pueden observar las especificaciones de algunas características importantes de ambas tarjetas.

	Nvidia GeForce GTS 450	Nvidia GeForce GTX 470
Reloj del núcleo	783 MHz	607 MHz
Reloj de la memoria	1804 MHz	1674 MHz
Ancho de banda de la memoria	57.728 GB/sec	133.92 GB/sec
Taza de relleno de píxeles	12528 MPixels/sec	24280 MPixels/sec
Taza de relleno de texturas	25056 MTexels/sec	33992 MTexels/sec

Tabla 4.2: Especificaciones de las tarjetas de video.

4.2. Voxelización

La voxelización es una de las etapas principales para la obtención de la iluminación global. Aunque este proceso puede ser realizado en tiempos interactivos, hay varios factores a tomar en cuenta para un buen resultado visual en el menor tiempo posible. En esta sección se estudiará primero la importancia del tamaño del volumen. Luego, se expondrá la relación entre el tamaño de la textura atlas y el tamaño del volumen. Finalmente, se comparará el uso de la voxelización por medio de texturas atlas y por medio de *slicemap*.

4.2.1. Tamaño del volumen

El tamaño del volumen afectará a la resolución de la escena voxelizada. Un mayor tamaño implica una mayor resolución, pero es más costoso de crear la jerarquía *mipmap* y crea mayor cantidad de niveles con los cuales intersectar el rayo. Por ello, hay que mantener un tamaño del volumen que posea una representación aceptable de la escena y no afecte en mayor medida la calidad y el tiempo necesario para el cálculo de la iluminación indirecta.

Debido a que el volumen es almacenado mediante una textura 2D, hay ciertas restricciones en su tamaño. Para su creación es necesario utilizar la tarjeta de video, por lo que el alto y el ancho del volumen están limitados por el máximo tamaño del *viewport*. En cuanto a la profundidad, esta limitada al uso de una textura con componentes $GL\_RGBA32UI$, la cual permite 32 bits por canal para un total de 128 bits. Por ello, la resolución del volumen depende es del alto y del ancho.

El tiempo de voxelización no depende del tamaño del volumen. El único proceso que es afectado por el tamaño del volumen es la creación de la jerarquía *mipmap*. En la figura 4.2 se observa un gráfico del tiempo en microsegundos que tarda en realizarse la creación de la jerarquía para diferentes resoluciones del volumen. Como puede observarse, a medida que se aumenta la resolución, el tiempo tiende a aumentar. Al llegar a la resolución $1024 \times 1024 \times 128$ píxeles el incremento de tiempo es considerable. Para todos los casos, en el A1 se presentan mejores tiempos. Esto se debe a que para crear la jerarquía *mipmap* se realizan cuatro operaciones lógicas por cada píxel en cada uno de los niveles de la jerarquía. Estas operaciones son atómicas y se realizan en menor tiempo en el A1 debido principalmente a que tiene un reloj de núcleo más rápido que el A2.

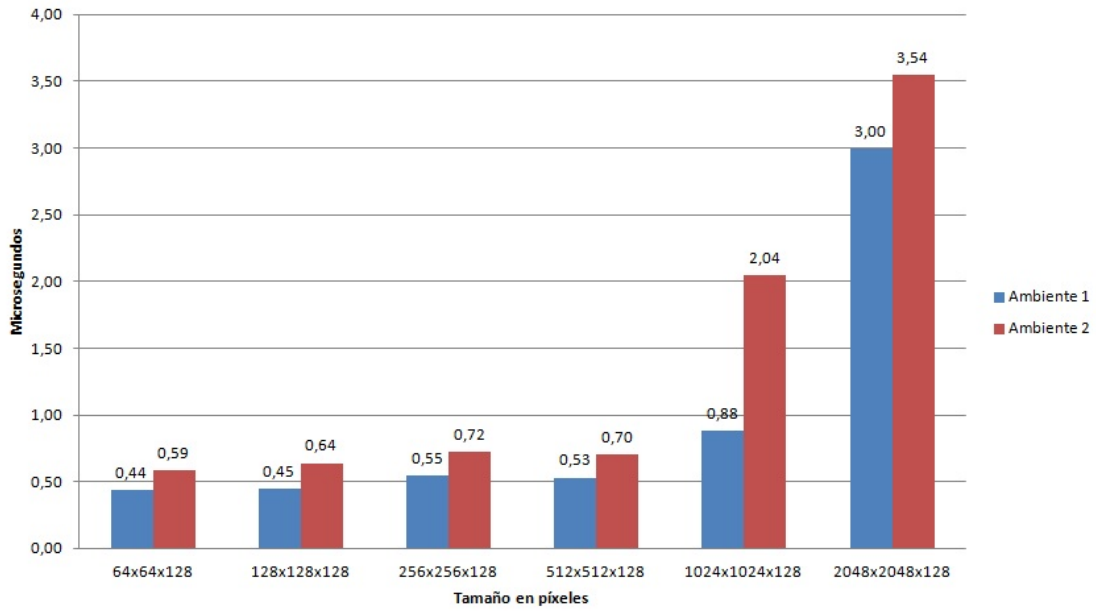


Figura 4.2: Tiempo en microsegundos de la creación de la jerarquía *mipmap* para diferentes resoluciones del volumen.

Con estos tiempos podemos observar que es recomendable utilizar una resolución de los volúmenes entre $128 \times 128 \times 128$ píxeles y $512 \times 512 \times 128$ píxeles. Aunque la resolución de $64 \times 64 \times 128$ píxeles también presenta buenos tiempos, contiene una representación de la escena muy discretizada y esto no es conveniente para el cálculo de la iluminación. Sin embargo, si se utiliza la voxelización por texturas atlas, una alta resolución del volumen implica la utilización de una alta resolución de la textura atlas, lo cual es costoso computacionalmente.

Debido a lo anterior expuesto, a partir de ahora se utilizará un volumen de dimensiones $128 \times 128 \times 128$ píxeles en las pruebas, para mantener una representación de la escena cuadrada y que utilize poco tiempo para la creación de la jerarquía.

4.2.2. Tamaño de la textura atlas

El tamaño del volumen representará la resolución de la escena voxelizada. Sin embargo, a mayor cantidad de vóxeles, se necesita un mayor tamaño de textura atlas para poder hacer una buena correspondencia entre la geometría y el volumen. Debe existir una correspondencia de uno a uno desde el *texel* de la textura atlas a un vóxel.

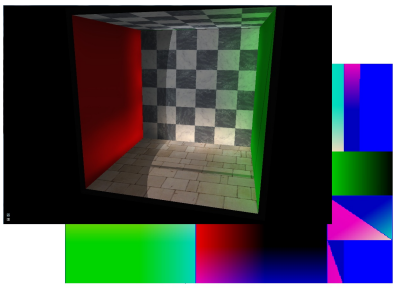
En la tabla 4.3 se muestra la cantidad de vértices creados por diferentes resoluciones de textura atlas. La cantidad de vértices varía poco entre una escena y otra, ya que no depende de la complejidad de la escena sino de la correspondencia con la textura atlas. Sin embargo, el número de vértices de una resolución es aproximadamente cuatro veces mayor a la cantidad de vértices de la resolución anterior.

	Resolución en píxeles		
Escena	128^2	256^2	512^2
E1	15.956	63.831	253.917
E2	16.346	65.346	260.847
E3	15.808	64.127	256.573

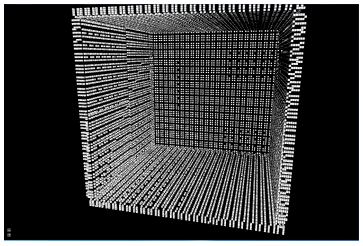
Tabla 4.3: Vértices creados por diferentes resoluciones de texturas atlas.

A pesar de la cantidad de vértices creados, es necesario tener una buena resolución que permita representar la escena de forma correcta. En la figura 4.3 se observa como la E1 (figura 4.3(a)) es voxelizada mediante el uso de diferentes resoluciones de textura atlas. En la figura 4.3(b) se observa que una textura atlas de dimensiones 128^2 píxeles no es suficiente para representar correctamente la escena. Con una resolución de 512^2 píxeles, en la figura 4.3(d) se observa una buena correspondencia. Colocar una mayor resolución a la textura atlas solo lograría que varios *texels* correspondan con un mismo vóxel, desplegando así vértices innecesariamente y afectando el desempeño de la aplicación.

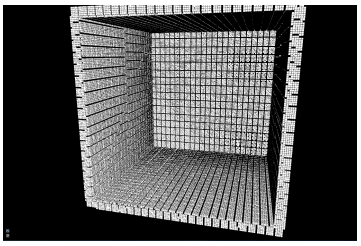
Una resolución de 512^2 píxeles para la textura atlas se considera adecuada para la mayoría de las escenas. Sin embargo, dependiendo del área ocupada por las superficies en la textura atlas, este valor puede cambiar. Superficies que ocupen mucho espacio en la escena y estén



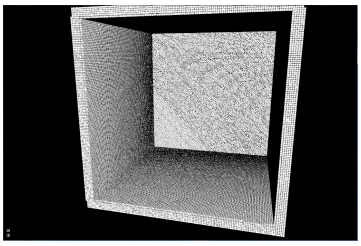
(a) Escena con textura atlas



(b) Resolución 128^2



(c) Resolución 256^2



(d) Resolución 512^2

Figura 4.3: Voxelización de una escena con diferentes tamaños de textura atlas.

representadas en espacios muy pequeños en la textura atlas, necesitan de una textura atlas con una resolución mayor. En el caso de objetos que ocupen muy poco espacio dentro del volumen, se puede utilizar una resolución mucho menor, ya que la cantidad de vóxeles que ocupa es muy pequeña. No obstante, esto ocasionará que la superficie del objeto se represente de una manera muy discretizada en el volumen.

4.2.3. Comparación del uso de texturas atlas y *slicemap*

Tanto el uso de las texturas atlas como el *slicemap* logran la voxelización de la escena en tiempo real. Sin embargo, difieren en calidad visual y en tiempo. En la figura 4.4 se observa un gráfico con mediciones de tiempo en microsegundos, en las que se compara el uso de las texturas atlas y el *slicemap*. Todas las pruebas se realizaron con volúmenes de resolución 128^3 píxeles y texturas atlas de resolución 512^2 píxeles.

En ambos métodos, se observa que mientras sea mayor la complejidad de la geometría, mayor es el tiempo de voxelización. La única excepción a esto es las texturas atlas del A2 en la E2, comparado con la texturas atlas del A2 en la E1. Por otro lado, para todas las escenas y todos los ambientes es posible observar que voxelizar por medio de las texturas atlas consume más tiempo que por medio del *slicemap*. Esto se debe a que el *slicemap* despliega directamente la geometría al volumen, mientras que las texturas atlas necesitan desplegar la geometría a las texturas atlas y luego desplegar estas al volumen. Sin embargo, para la mayoría de las escenas el tiempo adicional que se necesita para voxelizar con esta técnica no es excesivo. En el peor de los casos (E3) la diferencia es de 5 microsegundos, pero esto se debe a que en esta escena hay dos objetos, cada uno con sus respectivas texturas atlas. Para

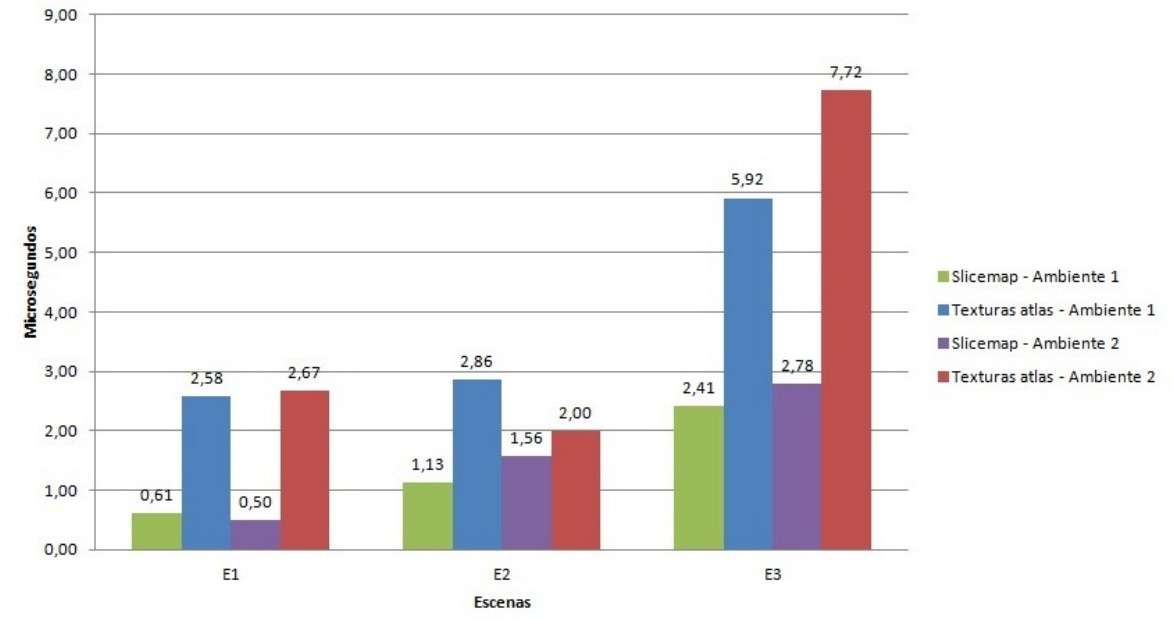


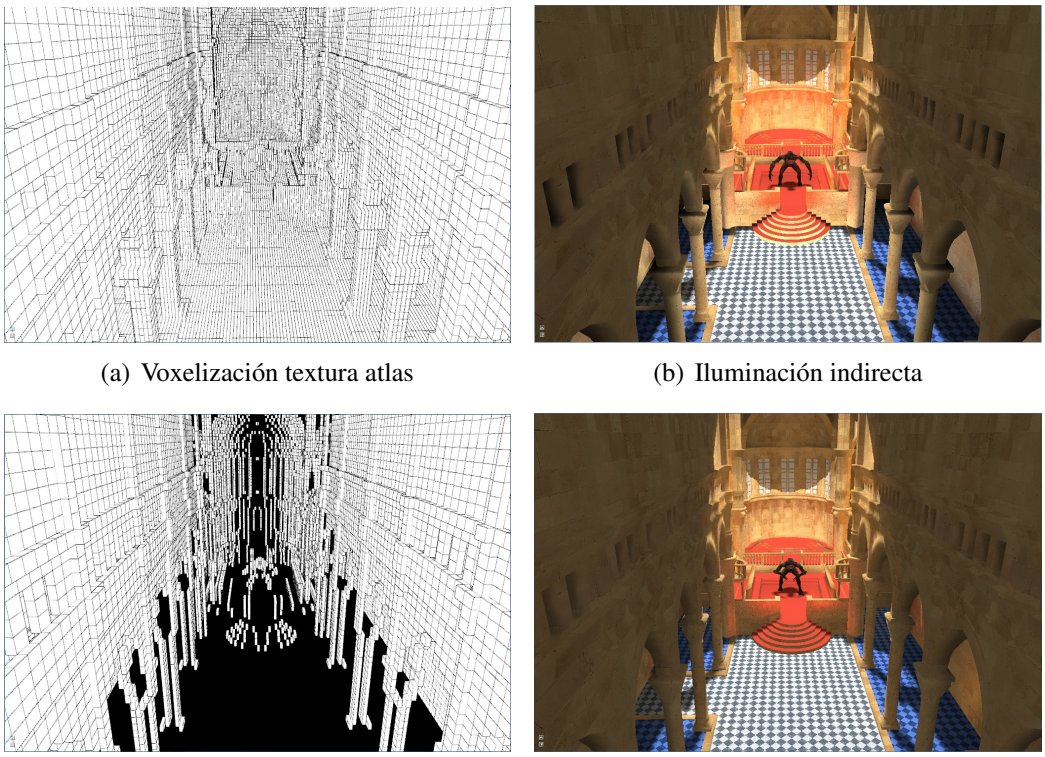
Figura 4.4: Tiempo de voxelización de escenas en microsegundos.

la mayoría de las mediciones se observa que el A1 presenta mejores tiempos que el A2.

A partir del análisis anterior se puede apreciar que hay poca diferencia en cuanto al tiempo de voxelización entre el *slicemap* y la *textura atlas*. Sin embargo, en la figura 4.5 se muestra la E2 voxelizada con texturas atlas 4.5(a) y con *slicemap* 4.5(c). Utilizando *slicemap* se pierde mucha información en la voxelización, lo que ocasiona errores al calcular la iluminación indirecta 4.5(d), a diferencia de la voxelización mediante texturas atlas 4.5(b). Por ello, es recomendable utilizar siempre texturas atlas.

Todas los análisis sobre los tiempos de voxelización se realizaron utilizando solo modelos en formato OBJ, los cuales representan la parte estática de la escena. En la tabla 4.4 se observa la variación de tiempo de voxelización presente en la E1 al agregar un solo modelo en formato MD2, utilizando texturas atlas. Debido a que este modelo ocupa un espacio pequeño dentro del volumen, se utilizó una textura atlas de resolución 128^2, la cual es suficiente para representar al modelo de forma voxelizada. Por ello, el agregar un objeto dinámico a la escena, se genera poco retraso en el tiempo de voxelización. Como se observa en la tabla, un objeto dinámico en el A1 genera un aumento de tiempo cerca del $5\,\%$, mientras que agregarlo al A2 solo representa un aumento del $1\,\%$. En ambos casos se observa que el A1 voxeliza más rápido que el A2.

El uso de la voxelización de la escena para hacer el cálculo de la iluminación, no representa un gran costo de tiempo en el desarrollo del algoritmo. Como se observa en el gráfico de la figura 4.4, la escena más grande (E3) consume menos de 10 microsegundos para ser voxelizada. Esto representa un tiempo aceptable. Sin embargo, como se mencionó en los capítulos 2 y 3, se puede utilizar una prevoxelización de la escena, en la cual los objetos estáticos son voxelizados una sola vez y posteriormente solo se voxelizan los objetos dinámicos.



(a) Voxelización textura atlas (b) Iluminación indirecta

(c) Voxelización *slicemap* (d) Iluminación indirecta

Figura 4.5: Voxelización de una escena mediante *slicemap* y texturas atlas, y su correspondiente contribución al cálculo de la iluminación indirecta.

Ambiente	Sin MD2	Con MD2
A1	2,58 μs	2,70 μs
A2	2,67 μs	2,71 μs

Tabla 4.4: Comparación de tiempos en microsegundos al agregar un objeto dinámico a la escena.

De esta manera, los tiempos de voxelización disminuyen, permitiendo realizar el cálculo de la iluminación inidrecta de manera más rápida.

4.3. Iluminación indirecta

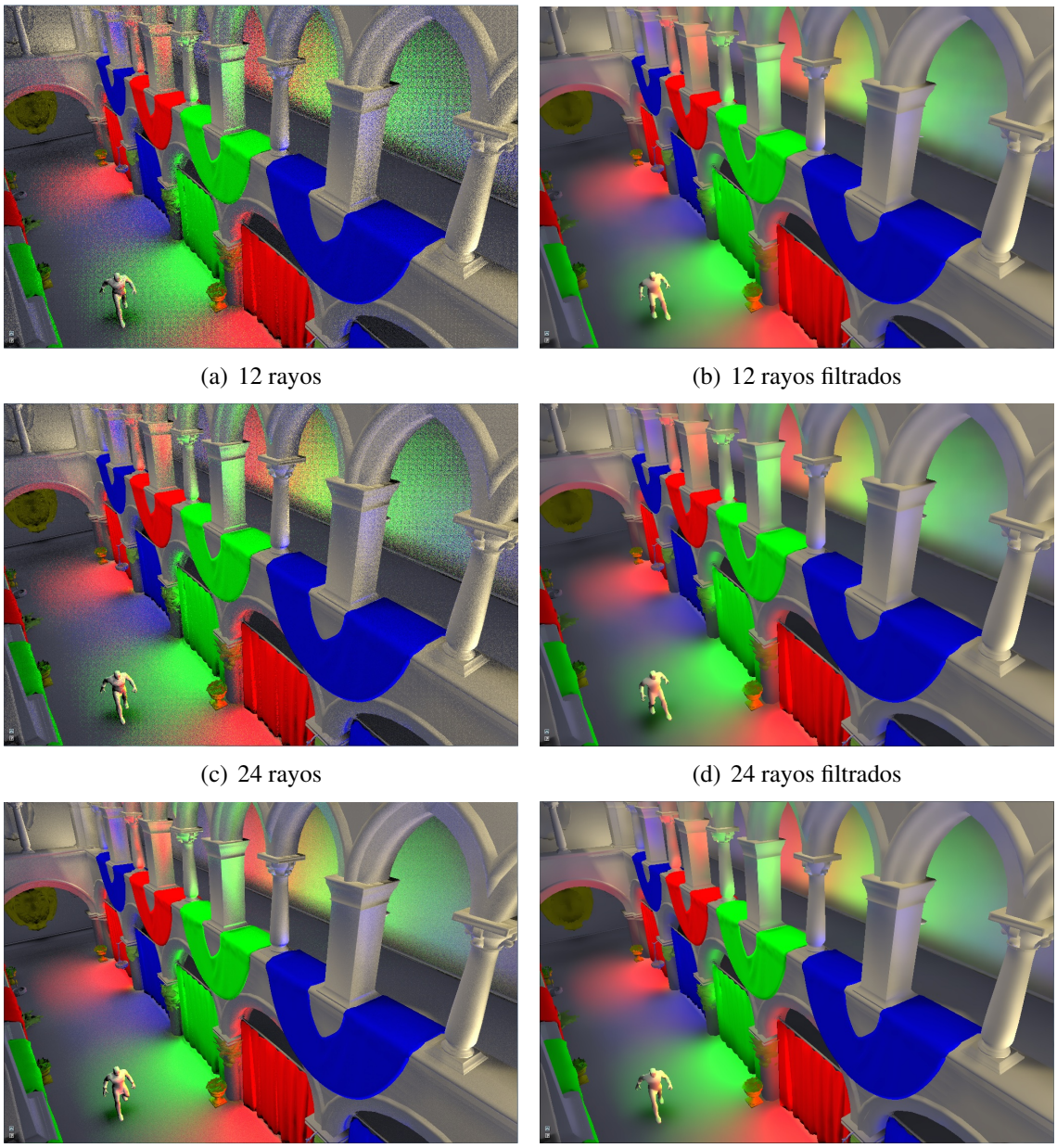
El cálculo de la iluminación indirecta es la etapa del algoritmo que consume más tiempo. Esto depende de muchos factores como el número de rayos, el número de pasos permitidos para encontrar una intersección, el tamaño de la ventana, la longitud del rayo, entre otros. La importancia del tamaño de la ventana se estudiará en la siguiente sección y todas las pruebas en este capítulo serán realizada con una ventada de 1024×680 píxeles.

En cuanto al número de pasos (variable *step* mencionada en la sección 3.4.8 del capítulo 3) para encontrar una intersección, a través de pruebas visuales se determinó que 16 pasos es un valor que crea una solución aceptable para las escenas de prueba. Un aumento de este valor conlleva un aumento en el tiempo necesario para calcular la intersección y también es dependiente del número de rayos. Como los rayos son lanzados para cada uno de los píxeles de la imagen, al aumentar la cantidad de rayos se disminuye el desempeño de la implementación.

En la figura 4.6 puede observarse como el uso de diferentes cantidades de rayos y del filtro afectan el resultado visual de la iluminación indirecta. La cantidad de rayos utilizados en esta prueba fueron escogidos empíricamente. Se puede observar que el uso de 12 (4.6(a), 13 fps), 24 (4.6(c), 7,4 fps) y 128 (4.6(e), 1,5 fps) rayos es insuficiente para crear una solución de iluminación indirecta sin ruido. Por ello, se hace el uso del filtro mostrado en la sección 3.4.9 del capítulo 3, con el cual se elimina de manera eficiente el ruido en las imágenes de 12 (4.6(b), 10 fps), 24 (4.6(d), 6,3 fps) y 128 (4.6(f), 1,46 fps) rayos. La cantidad de *frames* por segundo[1] mostrados fueron medidos en el A1. Se puede apreciar que hay poca diferencia visual entre las imágenes, por lo que utilizar 12 rayos con filtro es suficiente para crear una imagen visualmente aceptable en un tiempo aceptable. Por lo tanto, las pruebas en esta sección se realizarán con 12 rayos y con 16 pasos para encontrar la intersección.

Una vez fijado el número de rayos y el número de pasos para encontrar la intersección, es importante estudiar como afecta la longitud del rayo en el desempeño de la aplicación. En la figura 4.7 podemos observar dos gráficos, en los que se muestra el tiempo en microsegundos de utilizar diferentes longitudes del rayo en cada una de las escenas de prueba. La longitud del rayo está medida en porcentaje de la longitud del lado más grande de la caja envolvente de la escena. A medida que el porcentaje es más grande, el tiempo para el cálculo de la iluminación indirecta es mayor para todas las escenas y para todos los ambientes. Sin embargo, solo en la E1 hay aumentos considerables del tiempo al aumentar la longitud. Esto se debe a que en la E2 y E3, con una longitud corta del rayo basta para calcular la mayor parte de las intersecciones desde el punto de vista en el cual se realizaron las pruebas, por lo que un aumento de la longitud del rayo permite encontrar pocas intersecciones más. Para todos los casos, los tiempos obtenidos por el A2 (figura 4.7(b)) son notoriamente menores a los tiempos obtenidos por el A1 (figura 4.7(a)), aunque la velocidad del reloj del núcleo y de la memoria sean mayores en el A1. Esto se debe a que el cálculo de la iluminación se realiza en espacio de imagen accediendo a varias texturas, por lo que el A2 con un ancho de banda de la memoria, una taza de relleno de píxeles y una taza de relleno de texturas más rápido, realiza este cálculo más rápido. Además, en el A2 la mayoría de los tiempos obtenidos en la E3 son menores a los tiempos obtenidos en la E2, a pesar de que la E3 es más compleja. Esto se debe a que el cálculo de la iluminación indirecta no es dependiente de la complejidad geométrica de la escena, sino de la representación de la misma en el volumen. En una escena abierta con objetos lejanos unos de otros, el recorrido del rayo consume mayor tiempo que en una escena donde los objetos se encuentran cercanos, como es el caso de la E3.

[1] *Frames* por segundo o *fps* es la medida de frecuencia a la cual un reproductor de imágenes genera distintos frames. En una aplicación gráfica es ideal producir imágenes a 60 fps.



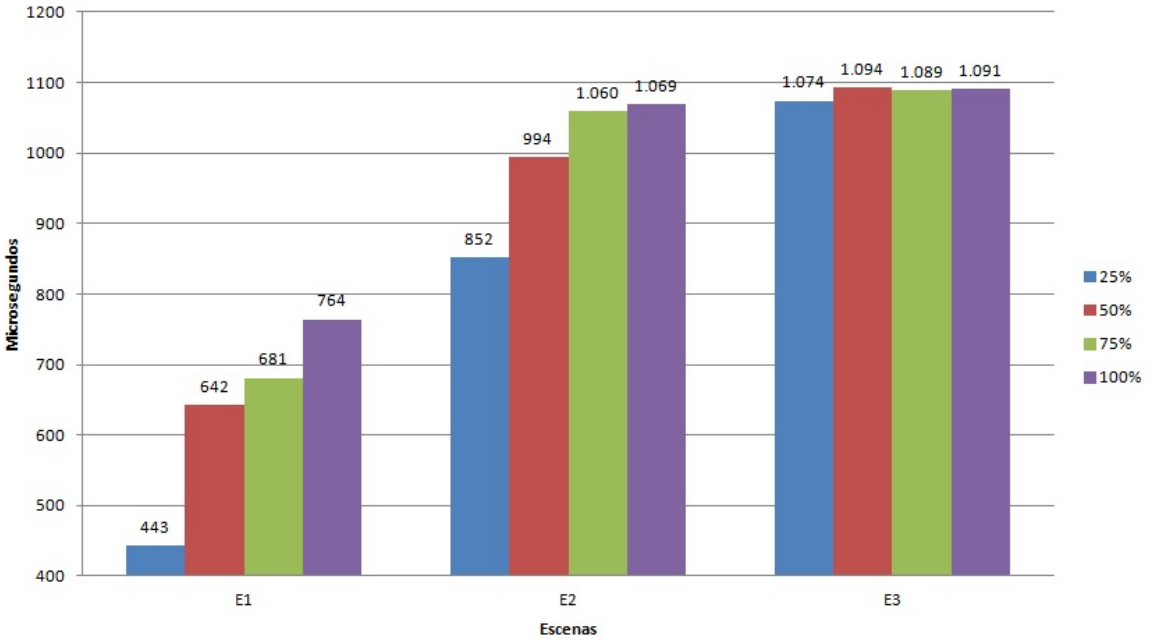
(a) 12 rayos (b) 12 rayos filtrados

(c) 24 rayos (d) 24 rayos filtrados

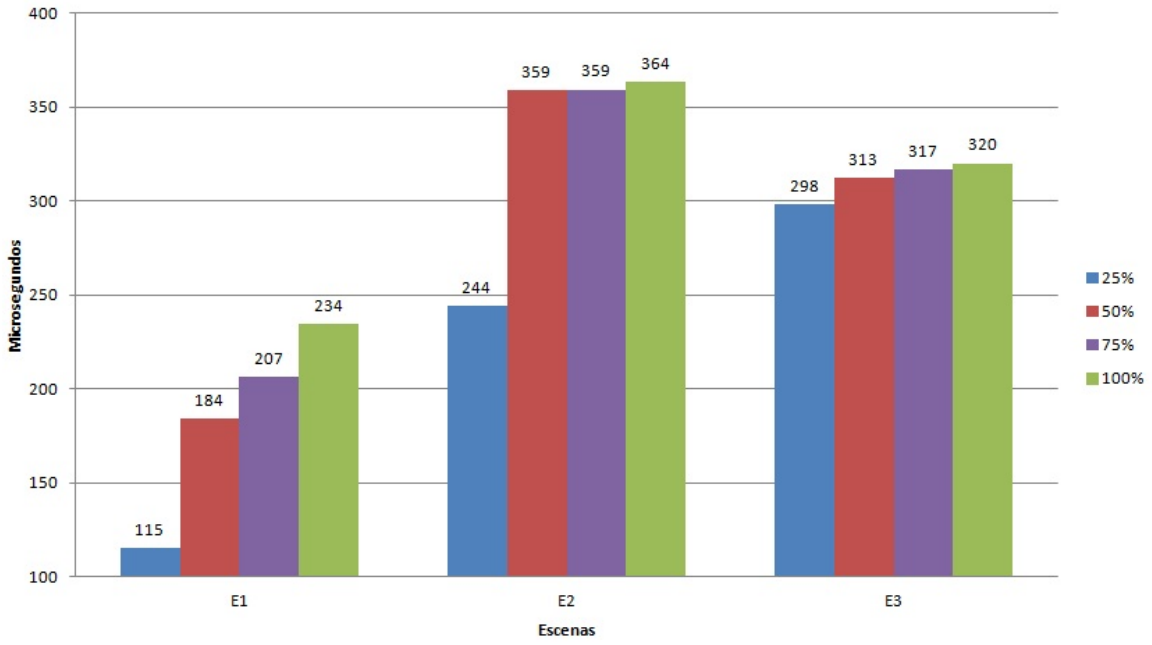
(e) 128 rayos (f) 128 rayos filtrados

Figura 4.6: Comparación del uso del filtro para diferentes cantidades de rayos.

A pesar que el filtro elimina de manera considerable el ruido de la iluminación indirecta, consume una gran cantidad de tiempo. En la tabla 4.5 se puede apreciar el tiempo en microsegundos necesario para realizar el filtro de una imagen. El tiempo necesario por el A2 es mucho menor al tiempo necesario por el A1. Para ambos ambientes y según los gráficos mostrados en la figura 4.7, el tiempo del filtrado de la imagen representa entre aproximadamente $40\,\%$ (los tiempos más cortos) y $15\,\%$ (los tiempos más grandes) del tiempo total del cálculo



(a) Ambiente 1



(b) Ambiente 2

Figura 4.7: Comparación del uso de diferentes longitudes del rayo en el cálculo de la iluminación indirecta.

de la iluminación indirecta. A pesar de que el uso del filtro es costoso, su uso ofrece mejores resultados que aumentar el número de rayos.

Ambiente	Tiempo
A1	180,72 μs
A2	47,28 μs

Tabla 4.5: Tiempos en microsegundos de la aplicación del filtro en la iluminación indirecta.

4.4. Tamaño de la ventana

Dado que la implementación del algoritmo trabaja mayormente en el espacio de imagen, cambios en el tamaño de la ventana de despliegue afectan el desempeño de la aplicación. Procesos como el cálculo del *G-Buffer*, la iluminación directa y en especial la iluminación indirecta dependen de la cantidad de píxeles de la imagen. En la figura 4.8, se observa un gráfico que indica la cantidad de *frames* por segundo para diferentes tamaños del *viewport*. A medida que el tamaño de la ventana va aumentando, la cantidad de *frames* por segundo va disminuyendo gradualmente. Para todos los tamaños el A2 presenta un mejor rendimiento al mostrado por el A1, incluso la menor cantidad de *frames* por segundos mostrado en A2 (mayor tamaño de ventana), es igual a la mayor cantidad de *frames* por segundo mostrado por A1 (menor tamaño de la ventana).

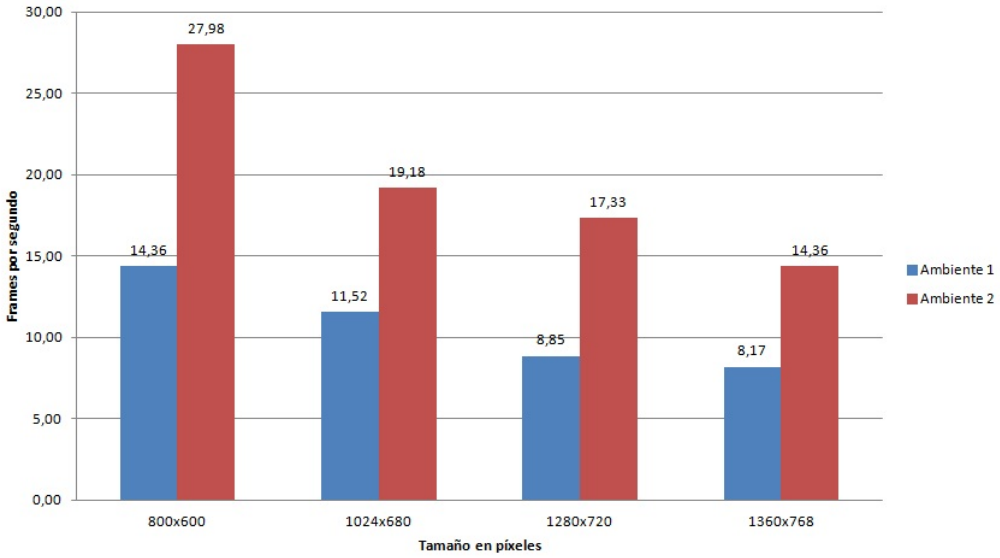


Figura 4.8: Cantidad de fps para diferentes tamaños del *viewport*.

Capítulo 5

Conclusiones y trabajos futuros

En este trabajo se implementó un algoritmo de iluminación global basándose en las ideas propuestas por Thiedemann et al. [4]. Esta implementación fue desarrollada para hacer la mayor parte de sus cálculos con el uso de la GPU. Además, para permitir la interactividad y la prueba de la misma, se realizó de manera que los parámetros puedan ser modificado.

A través de pruebas sobre esta implementación se logró determinar que la voxelización por medio de texturas atlas obtiene una mejor representación de la escena y es una mejor opción que voxelizar por medio de *slicemap*, a pesar de que consuma mayor tiempo de procesamiento y espacio en memoria. La elección de un tamaño adecuado para el volumen y las texturas atlas es dependiente de la escena y el hardware. Es dependiente de la escena, ya que una escena con mucha geometría y detalle puede requerir de una mayor resolución del volumen y de texturas atlas para ser representada correctamente. También dependen del hardware, ya que un tamaño considerable puede saturar la memoria de la GPU. Por otro lado, se logró determinar a través de las pruebas que el número de rayos, la longitud de los mismos y el tamaño de la ventana son factores que influyen de manera significativa en el desempeño del algoritmo, así como el uso de un filtro es crucial para poder obtener una solución de iluminación indirecta sin ruidos en tiempo real. Debido a que las pruebas fueron realizadas con diferentes tarjetas de video, se pudo apreciar que una tarjeta de video más novedosa tiene la capacidad de acelerar de manera significativa este algoritmo.

Aunque esta técnica de iluminación global permite el cálculo en tiempos interactivos, presenta algunas simplificaciones que pueden restar realismo a la imagen final. Primero, solo toma en cuenta superficies difusas, obviando la contribución especular. Además, solo considera un solo rebote de la luz, por lo que simplifica caminos de la luz que pueden añadir iluminación a la escena. Aunque se hace uso del volumen para el cálculo de la luz indirecta, muchos otros cálculos se realizan en espacio de imagen y este algoritmo posee algunos de los defectos que otros algoritmos de espacio de imagen también poseen. A pesar de ello, es posible lograr buenos resultados visuales en un tiempo aceptable y se puede integrar con alguna otra herramienta.

Como trabajo futuro se plantea el uso de una voxelización más novedosa como la propuesta por Fei et al. [62] y un filtro para la iluminación indirecta más novedoso como el propuesto por Chen et al [63]. También, es importante estudiar la influencia en tiempo y ca-

lidad visual, de la variación de la cantidad de pasos para buscar una intersección. Además, se recomienda comparar el uso de tarjetas de videos más novedosas (actualmente disponibles en el mercado), esperando así un aumento de la velocidad de procesamiento de la iluminación.

Bibliografía

[1] T. Ritschel, C. Dachsbacher, T. Grosch, y J. Kautz, "The state of the art in interactive global illumination," *Computer Graphics Forum*, vol. 31, pp. 160–188, Febrero 2012.

[2] P. Dutré, K. Bala, P. Bekaert, y P. Shirley, *Advance Global Illumination*. A K Peters, Ltd., 2006.

[3] M. Dabrovic. (2001) Sibenik cathedral. [En línea]. Disponible en: http://hdri.cgtechniques.com/ sibenik2/

[4] S. Thiedemann, N. Henrich, T. Grosch, y S. Müller, "Voxel-based global illumination," en *I3D '11 Symposium on Interactive 3D Graphics and Games*. ACM, 2011, pp. 103–110.

[5] Gouraud, "Continuous shading of curved surfaces," *IEEE Transactions on Computers*, vol. 20, pp. 623–629, Junio 1971.

[6] B. T. Phong, "Illumination for computer generated pictures," *Communications of the ACM*, vol. 18, pp. 311–317, Junio 1975.

[7] J. Kajiya, "The rendering equation," *ACM SIGGRAPH Computer Graphics*, vol. 20, pp. 143–150, 1986.

[8] T. Whitted, "An improved illumination model for shaded display," *ACM SIGGRAPH Computer Graphics*, vol. 13, pp. 343–349, 1979.

[9] R. Cook, T. Porter, y L. Carpenter, "Distributed ray tracing," *ACM SIGGRAPH Computer Graphics*, vol. 18, pp. 137–145, Julio 1984.

[10] J. Arvo, "Backward ray tracing," en *ACM SIGGRAPH '86 Course Notes - Developments in Ray Tracing*, 1986, pp. 259–263.

[11] E. Veach y L. Guibas, "Bidirectional estimators for light transport," en *Fifth Eurographics Workshop on Rendering*. Darmstadt, 1994, pp. 147–162.

[12] ——, "Metropolis light transport," en *SIGGRAPH '97 Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. ACM, 1997, pp. 65–76.

[13] C. Goral, K. Torrance, D. Greenberg, y B. Battaile, "Modeling the interaction of light between diffuse surfaces," *ACM SIGGRAPH Computer Graphics*, vol. 18, pp. 213–222, Julio 1984.

[14] G. Ward, F. Rubinstein, y R. D. Clear, "A ray tracing solution for diffuse interreflection," *ACM SIGGRAPH Computer Graphics*, vol. 22, pp. 85–92, Julio 1988.

[15] H. W. Jensen y N. J. Christensen, "Photon maps in bidirectional monte carlo ray tracing of complex objects," *Computers and Graphics*, vol. 19, pp. 215–224, 1995.

[16] H. W. Jensen, "Global illumination using photon maps," en *Proceedings of the eurographics workshop on Rendering techniques '96*. Springer-Verlag, 1996, pp. 21–30.

[17] ——, "Rendering caustics on non-lambertian surfaces," en *GI '96 Proceedings of the conference on Graphics interface '96*. Canadian Information Processing Society, 1996, pp. 116–121.

[18] ——, *Realistic Image Synthesis Using Photon Mapping*. A. K. Peters, Ltd., 2009.

[19] A. Keller, "Instant radiosity," en *SIGGRAPH '97 Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. ACM Press/Addison-Wesley, 1997, pp. 49–56.

[20] B. Walter, S. Fernandez, A. Arbree, K. Bala, M. Donikian, y D. P. Greenberg, "Lightcuts: A scalable approach to illumination," *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2005*, vol. 24, pp. 1098–1107, Julio 2005.

[21] B. Walter, A. Arbree, K. Bala, y D. P. Greenberg, "Multidimensional lightcuts," *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2006*, vol. 25, pp. 1081–1088, Julio 2006.

[22] A. Kaplanyan y C. Dachsbacher, "Cascaded light propagation volumes for real-time indirect illumination," en *I3D '10 Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*. ACM, 2010, pp. 99–107.

[23] R. Wang, R. Wang, K. Zhou, M. Pan, y H. Bao, "An efficient gpu-based approach for interactive global illumination," *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2009*, vol. 28, pp. 1–8, Agosto 2009.

[24] Z. Dong, J. Kautz, C. Theobalt, y H.-P. Seidel, "Interactive global illumination using implicit visibility," en *PG '07 Proceedings of the 15th Pacific Conference on Computer Graphics and Applications*. IEEE Computer Society, 2007, pp. 77–86.

[25] Bungie. (2007) Halo 3. [En línea]. Disponible en: http://halo.xbox.com/en-us/intel /titles/halo3

[26] BioWare. (2011) Dragon Age II. [En línea]. Disponible en: http://dragonage.bioware .com

[27] P.-P. Sloan, J. Kautz, y J. Snyder, "Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments," *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2002*, vol. 21, pp. 527–536, 2002.

[28] H. Chen y X. Liu, "Lighting and material in halo 3," en *SIGGRAPH '08 ACM SIGGRAPH 2008 classes*. ACM, 2008, pp. 1–22.

[29] T. Ritschel, T. Engelhardt, T. Grosch, H.-P. Seidel, J. Kautz, y C. Dachsbacher, "Micro-rendering for scalable, parallel final gathering," *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH Asia 2009*, vol. 28, Diciembre 2009.

[30] M. McGuire y D. Luebke, "Hardware-accelerated global illumination by image space photon mapping," en *HPG '09 Proceedings of the Conference on High Performance Graphics 2009*. ACM, 2009, pp. 77–89.

[31] C. Dachsbacher y M. Stamminger, "Reflective shadow maps," en *I3D '05 Proceedings of the 2005 symposium on Interactive 3D graphics and games*. ACM, 2005, pp. 203–231.

[32] M. Mittring, "Finding next gen: Cryengine 2," en *SIGGRAPH '07 ACM SIGGRAPH 2007 courses*. ACM, 2007, pp. 97–121.

[33] T. Ritschel, T. Grosch, y H.-P. Seidel, "Approximating dynamic global illumination in image space," en *I3D '09 Proceedings of the 2009 symposium on Interactive 3D graphics and games*. ACM, 2009, pp. 75–82.

[34] C. Dachsbacher y M. Stamminger, "Splatting indirect illumination," en *I3D '06 Proceedings of the 2006 symposium on Interactive 3D graphics and games*. ACM, 2006, pp. 93–100.

[35] G. Nichols y C. Wyman, "Multiresolution splatting for indirect illumination," en *I3D '09 Proceedings of the 2009 symposium on Interactive 3D graphics and games*. ACM, 2009, pp. 83–90.

[36] T. Ritschel, T. Grosch, M. H. Kim, H.-P. Seidel, C. Dachsbacher, y J. Kautz, "Imperfect shadow maps for efficient computation of indirect illumination," *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH Asia 2008*, vol. 27, Diciembre 2008.

[37] G. Nichols, J. Shopf, y C. Wyman, "Hierarchical image-space radiosity for interactive global illumination," *Computer Graphics Forum*, vol. 28, pp. 1141–1149, 2009.

[38] A. Kaplanyan, "Light propagation volumes in cryengine 3," en *ACM SIGGRAPH 2009 Courses - Advances in Real-Time Rendering in 3D Graphics and Games Course*, 2009.

[39] R. Fattal, "Participating media illumination using light propagation maps," *ACM Transactions on Graphics (TOG)*, vol. 28, Enero 2009.

[40] R. Geist, K. Rasche, J. Westall, y R. Schalkoff, "Lattice-boltzmann lighting," en *Rendering Techniques 2004 Proceedings of the Eurographics Symposium on Rendering*, 2004, pp. 355–362.

[41] P.-P. Sloan, N. Govindaraju, D. Nowrouzezahrai, y J. Snyder, "Image-based proxy accumulation for real-time soft global illumination," en *PG '07 Proceedings of the 15th Pacific Conference on Computer Graphics and Applications*. IEEE Computer Society, 2007, pp. 97–105.

[42] C. Dachsbacher, M. Stamminger, G. Drettakis, y F. Durand, "Implicit visibility and antiradiance for interactive global illumination," *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2007*, vol. 26, Julio 2007.

[43] A. Evans, "Fast approximations for global illumination on dynamic scenes," en *SIGGRAPH '06 ACM SIGGRAPH 2006 Courses*. ACM, 2006, pp. 153–171.

[44] P. Mavridis y G. Papaioannou, "Global illumination using imperfect volumes," en *GRAPP '11: Proceedings of the international conference on computer graphics theory and applications*, 2011.

[45] G. Greger, P. Shirley, P. Hubbard, y D. Greenberg, "The irradiance volume," *IEEE Computer Graphics and Applications*, vol. 18, pp. 32–43, Marzo 1998.

[46] C. Crassin, F. Neyret, M. Sainz, S. Green, y E. Eisemann, "Interactive indirect illumination using voxel cone tracing," en *Pacific Graphics 2011*, vol. 30. The Eurographics Association and Blackwell Publishing Ltd., 2011.

[47] E. Tabellion y A. Lamorlette, "An approximate global illumination system for computer generated films," *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2004*, vol. 23, pp. 469–476, Agosto 2004.

[48] S. Fang y H. Chen, "Hardware accelerated voxelization," *Computers and Graphics 24*, 2000.

[49] K. Crane, I. Llamas, y S. Tariq, "Real-time simulation and rendering of 3d fluids," *GPU Gems 3: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pp. 633–675, 2007.

[50] G. Passalis, T. Theoharis, G. Toderici, y I. Kakadiaris, "General voxelization algorithm with scalable gpu implementation," *Journal of Graphics, GPU and Game Tools*, vol. 12, pp. 61–71, Enero 2007.

[51] W. Li, Z. Fan, X. Wei, y A. Kaufman, "Flow simulation with complex boundaries," *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, 2005.

[52] Z. Dong, W. Chen, H. Bao, H. Zhang, y Q. Peng, "Real-time voxelization for complex polygonal models," en *PG '04 Proceedings of the Computer Graphics and Applications, 12th Pacific Conference*. IEEE Computer Society, 2004, pp. 43–50.

[53] E. Eisemann y X. Décoret, "Fast scene voxelization and applications," en *I3D '06 Proceedings of the 2006 symposium on Interactive 3D graphics and games*. ACM, 2006, pp. 71–78.

[54] V. Forest, L. Barthe, y M. Paulin, "Real-time hierarchical binary-scene voxelization," *Journal of Graphics, Gpu, and Game Tools*, vol. 14, pp. 21–34, 2009.

[55] D. Shreiner, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. Addison-Wesley Professional, 2009.

[56] Anttweakbar. [En línea]. Disponible en: http://www.antisphere.com/Wiki /tools:anttweakbar

[57] Open asset import library. [En línea]. Disponible en: http://assimp.sourceforge.net/

[58] M. Bunnell y F. Pellacini, "Shadow map antialiasing," *GPU Gems*, p. Capítulo 11, 2003.

[59] T. Ritschel, T. Grosch, y H.-P. Seidel, "Approximating dynamic global illumination in image space," en *I3D '09 Proceedings of the 2009 symposium on Interactive 3D graphics and games*. ACM, 2009, pp. 75–82.

[60] R. Herzog, E. Eisemann, K. Myszkowski, y H.-P. Seidel, "Spatio-temporal upsampling on the gpu," en *I3D '10 Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*. ACM, 2010, pp. 91–98.

[61] Crytek sponza. [En línea]. Disponible en: http://www.crytek.com/cryengine/cryengine3 /downloads

[62] Y. Fei, B. Wang, y J. Chen, "Point-tessellated voxelization," en *GI '12 Graphics Interace Conference*. Canadian Information Processing Society Toronto, Ont., 2012, pp. 9–18.

[63] Y.-C. Chen, S. I. E. Lei, y C.-F. Chang, "Spatio-temporal filtering of indirect lighting for interactive global illumination," *Computer Graphics Forum*, vol. 31, pp. 189–201, Febrero 2012.

[64] M. Cohen y D. Greenberg, "The hemi-cube: a radiosity solution for complex environments," *ACM SIGGRAPH Computer Graphics*, vol. 19, pp. 31–40, Julio 1985.

[65] M. Cohen, S. E. Chen, J. Wallace, y D. Greenberg, "A progressive refinement approach to fast radiosity image generation," *ACM SIGGRAPH Computer Graphics*, vol. 22, pp. 75–84, Agosto 1988.

[66] E. Eisemann y X. Décoret, "Single-pass gpu solid voxelization and applications," *In GI '08: Proceedings of Graphics Interface 2008*, vol. 322, pp. 73–80, 2008.

[67] C. Everitt, "Interactive order-independent transparency," NVIDIA Corporation, Rep. Tec., 2001.

[68] E. Lafortune y Y. Willems, "A theoretical framework for physically based rendering," *Computer Graphics Forum*, vol. 13, pp. 97–107, Mayo 1994.

[69] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, H. Teller, y E. Teller, "Equations of state calculations by fast computing machines," *Journal of Chemical Physics*, vol. 21, pp. 1087–1092, 1953.

[70] B.-T. Phong y F. C. Crow, "Improved rendition of polygonal models of curved surfaces," en *Proceedings of the 2nd USA-Japan Computer Conference*, 1975.

[71] E. Sparrow, "A new and simpler formulation for radiative angle factors," *Transactions of the ASME, Journal of Heat Transfer*, vol. 85, pp. 81–88, 1963.

[72] E. Sparrow y R. Cess, "Radiation heat transfer," en *Thermal and Fluids Engineering*. Hemisphere Publishing Corporation, 1978.

[73] M. Zwicker y M. Pauly, "Editorial: Point-based computer graphics," *ACM Computer Graphics*, vol. 28, pp. 799–800, Diciembre 2004.