UNIVERSITAT DE BARCELONA

Treball de Fi de Grau

GRAU D'ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

Renderizado Fotorrealista con Irradiance Caching

Alex Campos Aicart

Director: Ricardo Jorge Rodrigues Sepúlveda Marques
Realitzat a: Departament de Matemàtiques i Informàtica

Barcelona, 24 de enero de 2021

Índice

Tabla de ilustraciones

1.Abstract / Resumen / Resum

El renderizado fotorrealista es uno de los apartados más importantes en computación gráfica actual, aunque no lo pensemos usamos productos generados por estas técnicas cada día, desde catálogos de muebles, al cine o incluso videojuegos.

En este proyecto realizaremos la implementación de un algoritmo de renderizado fotorrealista llamado Irradiance Caching y lo compararemos con otro algoritmo de renderizado fotorrealista llamado Path Tracing, veremos las diferencias que hay entre uno y otro y los compararemos en el aspecto de resultados respecto al tiempo de ejecución de uno y otro.

La expectativa es poder realizar un algoritmo Irradiance Caching mucho más eficiente que el algoritmo Path tracing implementado en el ray tracer en el que vamos a desarrollar nuestro algoritmo, con una calidad de imagen igual o muy similar.

El renderitzat fotorealista es un dels temes mes importants en la computació gràfica actual, encara que no ho pensem, utilitzem milers de productes generats amb aquestes tècniques, des de catàlegs de mobles, cinema o videojocs.

En aquest projecte implementaren un algoritme de renderitzat fotorealista anomenat Irradiance Caching i el compararem amb un altre algoritme de renderitzat fotorealista anomenat Path Tracing, veurem les diferencies que hi ha entre un algoritme i un altre i els compararem en l'aspecte de resultats respecte al temps de execució.

L'expectativa es poder realitzar un algoritme Irradiance Caching molt mes eficient que l'algoritme Path Tracing proporcionat en el ray tracer en el qual desenvoluparem el nostre algoritme, amb una qualitat de imatge igual o molt similar.

Nowadays, we are sorrounded by computer generated images. Some examples may be: Forniture catalogues, video games, cinema... These images has evolved into what today is called Photorealistic Rendering, which is one of the most important topic in Computer Graphics.

This project implements one photorrealistic rendering algorithm called Irradiance Caching and compares it with another one called Path Tracing. Furthermore, it will showcase the differences between both of them and compare their execution times aswell.

The main goal is to program an Irradiance Caching algorithm in the ray tracer more efficient than the Path Tracing algorithm maintaining the image quality as much as possible.

2.Introducción

2.1 ¿Qué es el renderizado fotorrealista?

Llamaremos renderizado fotorrealista a una visualización 3D que es muy difícil de diferenciar de una fotografía, estas imágenes son útiles en la arquitectura y diseño de interiores, márquetin inmobiliario, catálogos etc.

El modelado 3D y el render fotorrealista esta desarrollado a partir de software, el cambio en los años de este último, junto al gran avance en hardware ha hecho que las técnicas y los resultados en renderizado fotorrealista haya avanzado mucho ya se en el aspecto de la rapidez de cálculo o en el aspecto visual del render.

2.2 Componentes de la mayoría de los renders

La mayoría de los renders tienen componentes que son necesarios para poder desarrollar la finalidad de estos, algunos de ellos son:

- Cámara: La cámara de un render simula los ojos de la persona e implica el punto de vista en el cual vamos a tener la imagen.
- Objetos de la escena: Estos objetos son los que saldrán en la imagen final si están a la vista de la cámara, son los encargados de representar objetos del mundo real ya sea en forma, color y textura.
- Algoritmo de renderizado: Una parte importantísima del renderizado fotorrealista es el algoritmo que usaremos para simular la luz dentro de nuestra escena.
- Fuentes de Luz: Las fuentes de luz son las que nos permitirán ver algo dentro de la escena como en el mundo real si estamos en una habitación sin ninguna fuente de luz no podemos ver nada en los algoritmos de rendering pasa exactamente lo mismo.

3.Iluminación Global

3.1 ¿Qué es la iluminación global?

En el mundo real nosotros podemos ver cosas gracias a los rayos de luz que salen de fuentes de luz, estos rayos de luz rebotan en las superficies de los objetos y llegan a nuestros ojos.

Esto es básicamente lo que nos deja ver cosas en el mundo real, y lo que tenemos que simular para recrear la iluminación global en una escena de una imagen en nuestro ordenador.

En el mundo real tendremos como fuentes de luz el sol, una bombilla, un fuego etc…, como objetos cualquier objeto real y la función de cámara la harán nuestros dos ojos.

La iluminación global se puede descomponer entre dos iluminaciones llamadas, iluminación directa e indirecta, podemos decir que la iluminación global es la suma de estas dos iluminaciones.

3.2 Tipos de Iluminación

En la iluminación distinguimos entre dos tipos, la diferencia básica entre los dos tipos que hay son el número de rebotes de la luz en los objetos.

Para explicar primero los tipos de iluminación tenemos que explicar qué son los rebotes que hace la luz en los objetos, como hemos dicho antes nosotros podemos ver los objetos gracias a que los rayos de luz rebotan en los objetos y nos llegan a nuestros ojos, estos rebotes pueden ser múltiples esta es la gran diferencia entre un tipo de iluminación y otra, el número de rebotes.

Los dos tipos de iluminación son los siguientes:

3.2.1 Iluminación Directa

Este tipo de iluminación incluye todos los rayos de luz que rebotan solo una vez en un objeto hasta llegar a nuestros ojos o a nuestra cámara, a partir de ahora hablaremos de cámaras en vez de ojos.

Como veremos más adelante esta iluminación es un tipo de iluminación que hace cambiar la imagen muy bruscamente por ello este tipo de iluminación es la que menos utilizaremos en nuestro algoritmo.

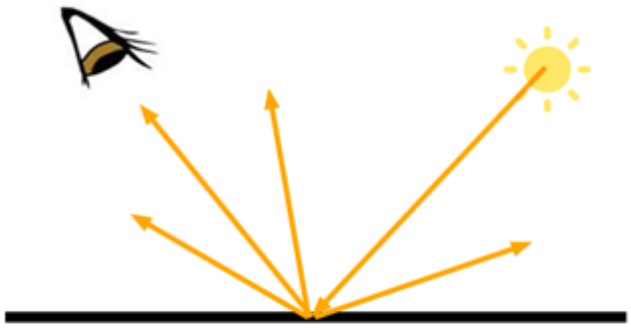

Ilustración Iluminación Directa

En esta imagen vemos la iluminación directa representada.

3.2.2 Iluminación Indirecta

También conocida como luz difusa este tipo de iluminación como la anterior se distingue por el número de rebotes del rayo antes de llegar a la cámara, en este caso contaremos como iluminación indirecta todos los rayos que rebotan más de una vez.

Por lo tanto, en esta iluminación tendremos todos los rayos que rebotan en varios objetos antes de llegar a la cámara.

Otra diferencia, que es resultado de tener múltiples rebotes en objetos, es que es un tipo de iluminación más suave, por lo tanto, el cambio de esta iluminación en una imagen es mucho menos perceptible que la iluminación directa.

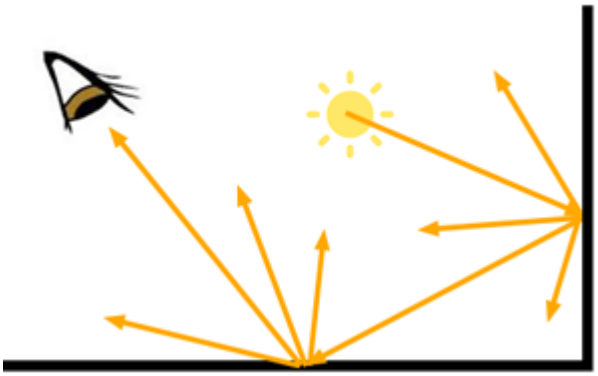

Ilustración Iluminación Indirecta

En esta imagen vemos la iluminación indirecta representada.

3.2.3 Diferencias de un render en luz indirecta y luz directa

Como hemos dicho una diferencia apreciable entre la luz directa y la luz indirecta, es la forma en que la imagen cambia de color.

Lo podemos ver en las siguientes imágenes:

Ilustración Iluminación Directa

Ilustración Iluminación Indirecta

La imagen de la derecha hace referencia a la iluminación directa y la de la izquierda a la iluminación indirecta, como podemos ver la imagen que solo muestra la iluminación directa el cambio de color es mucho más brusco que en la imagen que muestra la iluminación indirecta.

Ilustración Iluminación Global

En la imagen anterior mostramos la iluminación global que como hemos dicho antes es la suma de la iluminación indirecta y directa.

3.3 Solución a la Iluminación Global y Path Tracing

La solución a mostrar imágenes generadas como las mostradas anteriormente es desarrollar software capaz de simular estos caminos de luz explicados anteriormente, para esto hay algoritmos pensados explícitamente para simular los rayos de luz emitidos por fuentes de luz en una escena 3D que llegan a una cámara que será el punto de vista que nos formará la imagen final.

Uno de estos algoritmos es el algoritmo llamado Path Tracing, el cual trata de ofrecer solución al problema explicado anteriormente de la iluminación global.

Este tfg tratara de desarrollar otro algoritmo mejorando la eficiencia de este último.

3.4 Path Tracing

Path tracing es un algoritmo de renderizado de escenas, en el cual los rayos son lanzados desde una cámara virtual, y trazados a través de una escena 3D simulada.

Este algoritmo esta basado en un formula llamada *ecuación de render*, esta formula se basa en la premisa que el numero de luz que sale de un punto en una determinada dirección es igual al numero de luz que ese punto emite en esa dirección más el numero de luz que le llega de otros puntos.

Este algoritmo es un algoritmo que basa la obtención de las muestras en probabilidades, por lo tanto, podemos decir que se basa en el método de monte Carlo, esto significa que nuestro algoritmo lanzara muchas muestras para obtener un resultado que sea muy aproximado a la realidad.

Como hemos explicado path tracing es un algoritmo de rendering por lo tanto es un algoritmo que simula la luz de una escena a través de rayos simulados, podemos entender el numero de rayos simulados por el número de muestras.

El algoritmo de path tracing es simple, tenemos una imagen que es la que queremos rellenar, para eso tenemos que saber el color de cada pixel de la imagen en función de la escena 3D de la que estemos tomando la fotografía.
La forma de saber el color de cada pixel de la imagen es lanzando rayos a través de ellos, viendo donde chocan, y observar como actúa la luz en esa parte de la escena.
El número de rayos que lanzaremos por pixel es el numero de muestras del que hablábamos anteriormente.
La idea es simular rayos que inciden en algún punto de la escena 3D para saber la iluminación es ese punto, para eso cuando estemos en el primer punto volveremos a lanzar rayos, este paso lo repetiremos el número de veces que creamos necesario, esta característica es lo que nombramos la profundidad de los rayos, refiriéndonos a la profundidad como el numero de rebotes que harán los distintos rayos.

Hace falta detallar que como hemos explicado antes en el apartado de los tipos de iluminación para que exista algún tipo de iluminación los rayos que rebotan entre los objetos de la escena tienen que llegar a parar a alguna fuente de luz, esto hace que, al generar los rayos de manera aleatoria ya que es un algoritmo basado en el método de monte Carlo, haya muchos de ellos que nunca lleguen a un punto de luz y no sirvan para nada.

Este es uno de los motivos por el cual este algoritmo es tan lento, y es porque mucho de los recursos utilizados en el no llegan a intervenir en la imagen final.

Por esto que acabamos de explicar es el motivo por el cual necesitamos tener varias muestras por pixel ya que, si no podemos llegar a la situación de no tener ningún rayo que llegue a un punto de luz, por lo tanto, cuantas más muestras utilizamos en nuestro algoritmo de mayor calidad será la imagen porque esto hará que sea mas probable que alguno de los cientos de rayos que lanzamos en un pixel llegue a un punto de luz y nos aporte información a nuestro pixel.

4.Irradiance Caching

Irradiance Caching es el nombre del algoritmo que estaremos desarrollando durante todo este proyecto.

El objetivo de este algoritmo es tener un algoritmo de renderizado usando nuestro algoritmo path tracing, pero mucho más eficiente que este mismo.

4.1 Concepto Principal

El concepto de Irradiance Caching es bastante simple, para cada punto de la superficie en el cual queremos saber la irradiancia, miramos si nuestra cache de irradiancia tiene algún punto guardado valido, si tenemos valores interpolamos estos valores para sacar la irradiancia en nuestro punto, si no tenemos valores en nuestra cache calculamos la irradiancia explícitamente gracias al algoritmo path tracing y lo guardamos en esta.

4.2 Cache de Irradiancia

La cache de irradiancia es la estructura de datos que nos guarda toda la información sobre nuestros irradiance records, estos irradiance records son los puntos de la escena en los que hemos calculado la irradiancia en concreto.

En los irradiance records que guardamos en nuestra cache de irradiancia guardaremos todos los datos relevantes entorno al punto referenciado, estos datos pueden ser el mismo punto 3D, la irradiancia en ese punto calculada con el algoritmo path tracing, la normal del punto y por último todos los datos relacionados con la intersección del rayo en la superficie.

La estructura de datos más conocida aplicada en este algoritmo al ser una escena 3D es la llamada Octree, pero también podemos usar estructuras de datos más simples como un vector lineal, esto en el único aspecto que nos perjudicara es en el de la eficiencia del algoritmo, ya que como hemos explicado para cada punto en el cual queramos saber la irradiancia tendremos que buscar los irradiance records más cercanos recorriendo todo el vector una y otra vez.

4.1.2 Octree

Estructura de datos más aplicada en este algoritmo un octree es una estructura en forma de árbol donde cada nodo interno tiene exactamente 8 hijos.

Esto nos permite partir el espacio 3D de una escena en 8 partes iguales y a la vez partir estas partes en 8 más, así repetidamente hasta que tengamos las suficientes subdivisiones del espacio 3D.

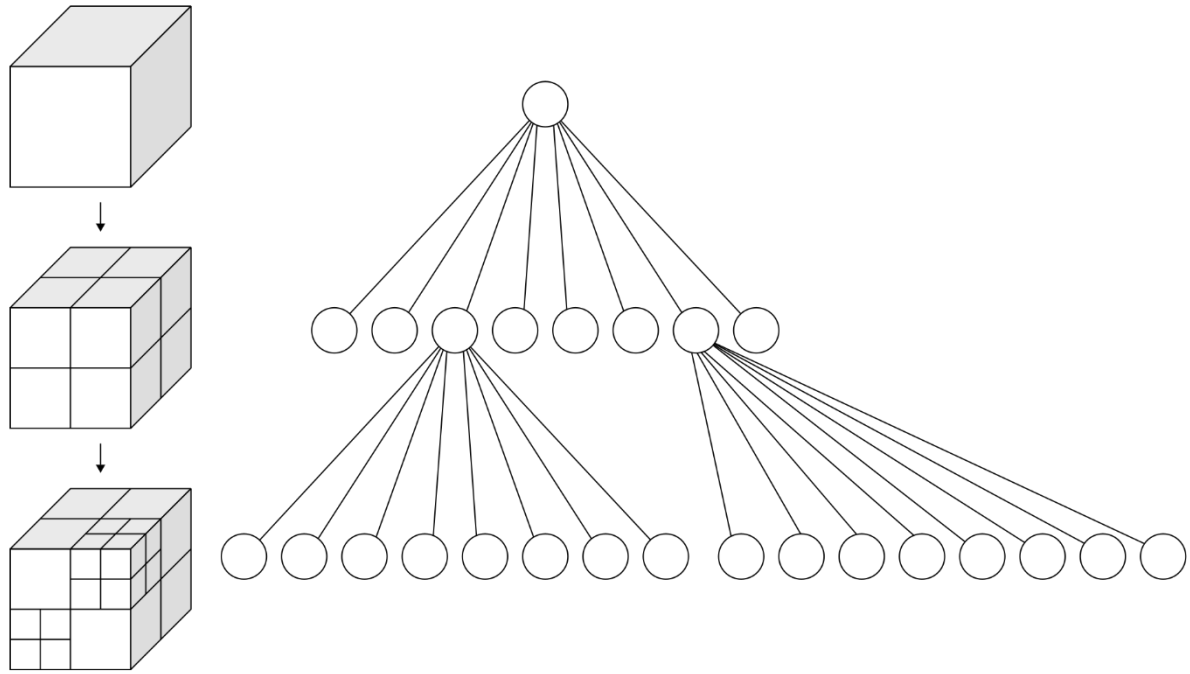

Ilustración Representación Grafica Octree

La imagen anterior nos muestra visualmente lo explicado anteriormente, podemos ver el cubo que sería nuestra escena 3D como se divide en cubos más pequeños y a la vez estos se subdividen en otros 8 cubos.

Como hemos dicho un octree es mucho más eficiente que por ejemplo un vector, esto es porque en cada hoja del arbol tendremos todos los irradiance records situados en esa parte de la escena, por lo tanto cuando queramos buscar los irradiance records más cercanos a un punto, solo tendremos que buscar en los irradiance records pertenecientes a la subdivision a la cual corresponda ese punto.
Esto hace que de tener que recorrer por ejemplo un vector de 1500 elementos para cada punto que queramos saber la irradiancia pasemos a recorrer vectores o grupos de elementos de 20 un cambio bastante significante.

4.2 Irradiancia/Iluminación

En los últimos párrafos hemos estado hablando de irradiancia, la irradiancia en un determinado punto es la suma de la iluminación que recibe de todas las direcciones.

En nuestro algoritmo solo usaremos la iluminación indirecta, por lo tanto, cuando estemos hablando de irradiancia será la suma de iluminación indirecta que recibe de varias direcciones.

Como hemos dicho utilizaremos la luz indirecta en nuestro algoritmo, nos puede surgir la duda de porque no utilizamos directamente el color o porque no usamos también la luz directa ya que esta es una luz igual de valida.
La respuesta a esta pregunta es la misma porque el color y la luz directa son factores que pueden cambiar muy bruscamente, mientras que la iluminación indirecta como explicamos más arriba es un tipo de iluminación que varía muy poco, su cambio es suave.

Esto nos facilitará las cosas ya que en este algoritmo como hemos dicho usaremos la irradiancia de distintos puntos para hacer una interpolación, esta interpolación será mucho más útil si lo que estamos interpolando tiene una tendencia de cambio suave, como la iluminación indirecta.

4.2.1 Calculo

En este punto ya sabemos que vamos a calcular, porque lo vamos a calcular y donde lo vamos a calcular, también sabemos el algoritmo principal que es bastante simple, lo que nos falta por explicar es como calcularemos cada irradiance record.

La principal diferencia entre irradiance caching y path tracing es que en path tracing lanzábamos un numero predeterminado de rayos por pixel, en irradiance lanzaremos solo uno.

En el punto de la superficie donde incida ese rayo es donde nosotros lanzaremos varios rayos y donde sacaremos la media de la iluminación en ese punto.

Cuando tengamos los rayos que vayamos a lanzar calculados, llamaremos a la función computeColor de el algoritmo Path Tracing, para calcular esta iluminación, por esto decíamos al principio que utilizábamos el algoritmo path tracing dentro del algoritmo de irradiance caching.

4.3 Muestras validas

Como hemos explicado para calcular la interpolación en un punto concreto de la escena, tendremos que decidir que irradiance records son válidos y cuales no son válidos.

Para esto nos fijaremos en 3 aspectos de nuestros puntos.

Primero de todo nos fijaremos si estos puntos están a la misma altura de la escena, ya que es tan importante que los puntos estén cerca como que los puntos estén a la misma altura o en otras palabras en el mismo plano de la imagen.

Segundo y la parte más lógica nos fijaremos en la distancia en la que están los puntos, contra más alta sea la distancia menos afectara este punto en la interpolación.

El tercero de estos aspectos será la normal de los puntos, en una superficie plana como una pared los dos puntos tendrán la misma normal, pero en una esfera por ejemplo la normal cambiara a cada punto, y contra mayor sea la diferencia entre la normal de los puntos menor será la incidencia del punto, esto es lógico ya que si en una esfera la normal es muy diferente querrá decir que el punto que estamos estudiando estará orientado probablemente a otros objetos en la escena y esto condicionara mucho la iluminación en ese punto.

El ultimo aspectos de todos es una idea que no hemos hablado aun y se llama distancia armónica, la distancia armónica nos dice a la distancia media en la que un punto tiene otros objetos de la escena, por lo tanto, si un punto tiene una distancia armónica más baja que otro, esto significara que ese punto tiene objetos más cercanos. Como hemos explicado la iluminación indirecta es la iluminación que nos mandan otros objetos de la escena, por lo tanto, si un punto tiene objetos más cercanos es más fácil que la iluminación en esa zona cambie más que en otra donde la distancia armónica sea más elevada.

Esto explicado anteriormente es solo las herramientas que utilizaremos para formar nuestro criterio a la hora de decidir qué puntos son válidos y cuáles no, más adelante en la parte de diseño explicaremos como utilizaremos estas herramientas para decidir.

5.Objetivos

En este punto tocaremos cuales son los objetivos principales de este trabajo, el cual es principalmente uno:

Conseguir tener un algoritmo que tenga un rendimiento superior al algoritmo de path tracing, con un resultado muy similar, en otras palabras, que visualmente nuestro render se vea con la misma claridad que el render generado por el algoritmo path tracer.

Para esto utilizaremos un ray tracer ya implementado con el path tracer, este ray tracer estará explicado en el siguiente punto de diseño, donde explicare como funciona y cómo está organizado.

Para conseguir nuestro objetivo utilizaremos un ray tracer proporcionado por el profesor y tutor del tfg Ricardo Jorge Rodrigues Sepúlveda Marques.

En este ray tracer tenemos casi todas las herramientas para implementar nuestro algoritmo Irradiance Caching.

Un objetivo extra de este Tfg es implementar un “octree”, ya que es la base de datos utilizada en este algoritmo, la idea principal de esta base de datos es la de un árbol binario, pero en vez de tener 2 hijos por nodo en este caso tendremos 8 hijos.

6.Diseño

6.1 Diagrama de clases del Ray Tracer

El siguiente diagrama representa las clases más importantes de nuestro ray tracing en azul están las clases que teníamos al empezar este tfg, en rosa son las clases añadidas para el desarrollo de este tfg.

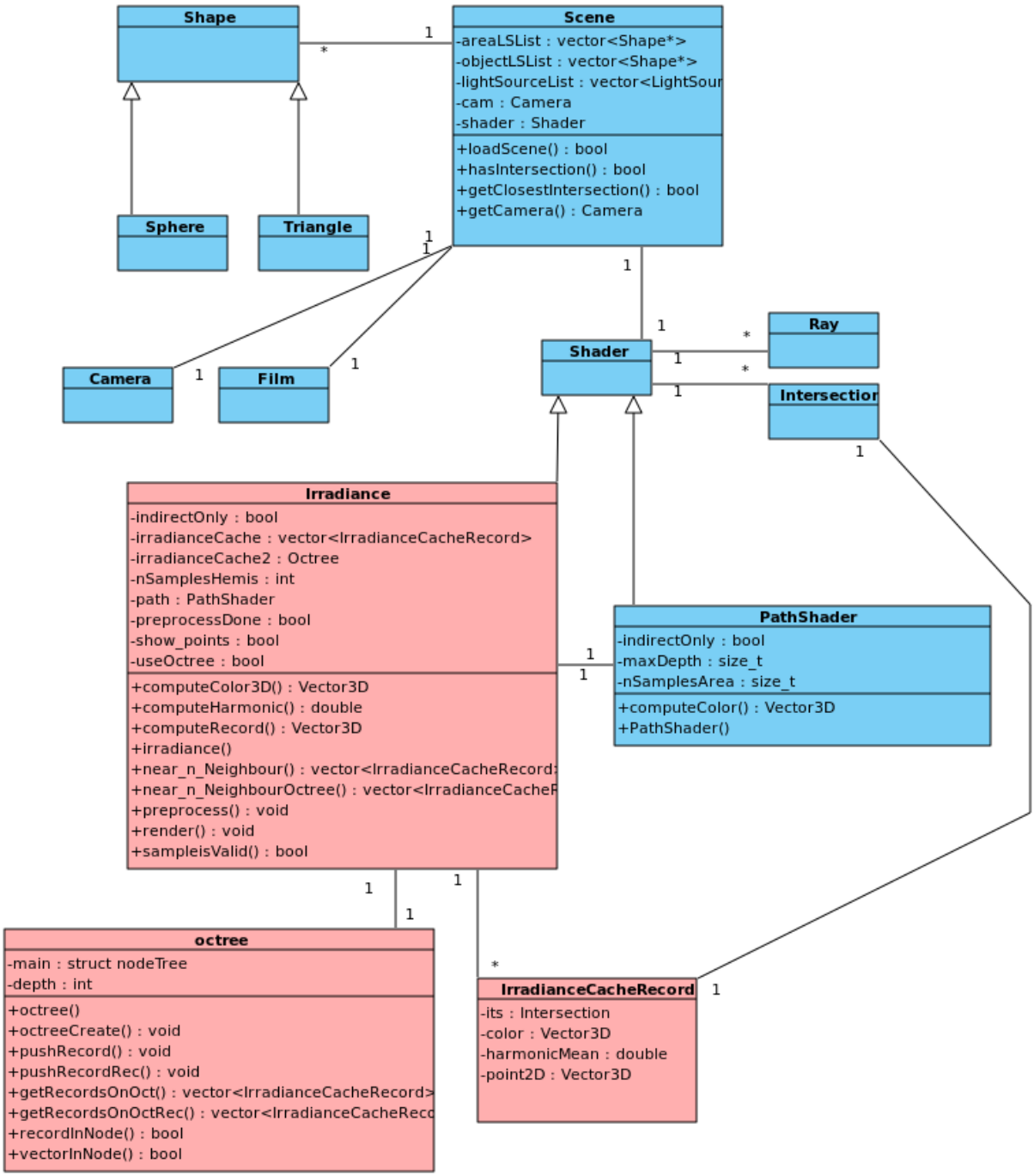

Ilustración Diagrama de clases Ray Tracer

Como podemos ver en el diagrama teníamos toda la estructura necesaria para implementar nuestro algoritmo, he incluido las clases que creo más importantes y las clases que más he utilizado durante el desarrollo del trabajo.

Explicaré ahora por encima cual era la funcionalidad de estas clases:

- **Shape** → Esta clase era una clase abstracta que nos daba las funciones y los parámetros necesarios para crear objetos 3D en nuestras escenas.
- **Sphere y Triangle**→ Eran los dos tipos de objetos que tenemos en nuestro ray tracer, estas dos clases heredan de Shape. Por ejemplo, todas las paredes que vemos en la escena que hemos utilizado durante todo el tfg está formada por dos triángulos diferentes.
- **Scene** → Esta clase era la encargada de contener toda la información relacionada con la escena, como podemos ver contiene los objetos, luces y cámaras que hay en nuestra escena, como es la encargada de contener toda la informacion tambien es la encargada de leer toda esta información desde el fichero donde especificamos estos datos.
- **Shader** → Clase abstracta que nos proporciona las funciones necesarias que necesitaremos para implementar algún tipo de shader, todos los shaders de nuestro ray tracing utilizan esta clase, en concreto la clase path shader y la clase irradiance.
- **Path Shader** → Clase que implementa el algoritmo path tracing, como hemos dicho antes utiliza la clase shader para implementar todas las funcionalidades de un shader. En particular este shader será utilizado por el shader irradiance para el cálculo de los irradiance records.
- **Irradiance** → Clase que implementa el algoritmo irradiance caching, también utiliza la clase shader como cualquier otro shader de este ray tracing. Esta es la primera clase de todas las que hemos comentado que será hecha por mí, será el centro de este tfg ya que como hemos explicado será la clase que desarrolle el algoritmo Irradiance Caching.
- **IrradianceCacheRecord** → Clase que utilizaremos como estructura de datos para nuestros irradiance records, esta clase guardará toda la información relacionada con cada uno de nuestros irradiance records calculados.
- **Ray** → Clase que utilizaremos durante todo el algoritmo ya que es la clase encargada de representar los rayos que calculamos para representar la luz indirecta en el caso de nuestro algoritmo Irradiance o luz directa en otros shaders, también es utilizado por ejemplo para saber a qué distancia media tenemos otros objetos respecto el punto que estamos estudiando, en general es una clase que simula rayos en nuestra escena.
- **Intersection** → Clase que utilizamos para guardar los datos de una intersección de un rayo con un objeto de la escena, estos datos pueden ser como el punto 3D donde se produce la intersección, a la normal del punto donde se produce la intersección etc…
- **Octree** → Clase que representará nuestra estructura de datos Octree, la cual es un árbol donde cada nodo tiene 8 hijos. Este octree como cualquier octree dividirá nuestro espacio 3D en 8 para que la búsqueda de puntos cercanos sea mucho más eficiente y rápida.

Esta es la explicación de la función de las clases más importantes de nuestro ray tracing, en este ray tracing hay muchas más clases que hemos utilizado, pero no creía importante incluirlas en esta explicación.

Más adelante entraremos más en detalle en las clases que hemos incluido en este ray tracing las clases Irradiance, IrradianceCachingRecord y la clase octree, sobretodo la clase irradiance que es la clase que incluye todo el algoritmo.

6.2 Diagrama de Flujo

En el siguiente diagrama vemos el diagrama de flujo del ray tracing en el caso del algoritmo de irradiance caching

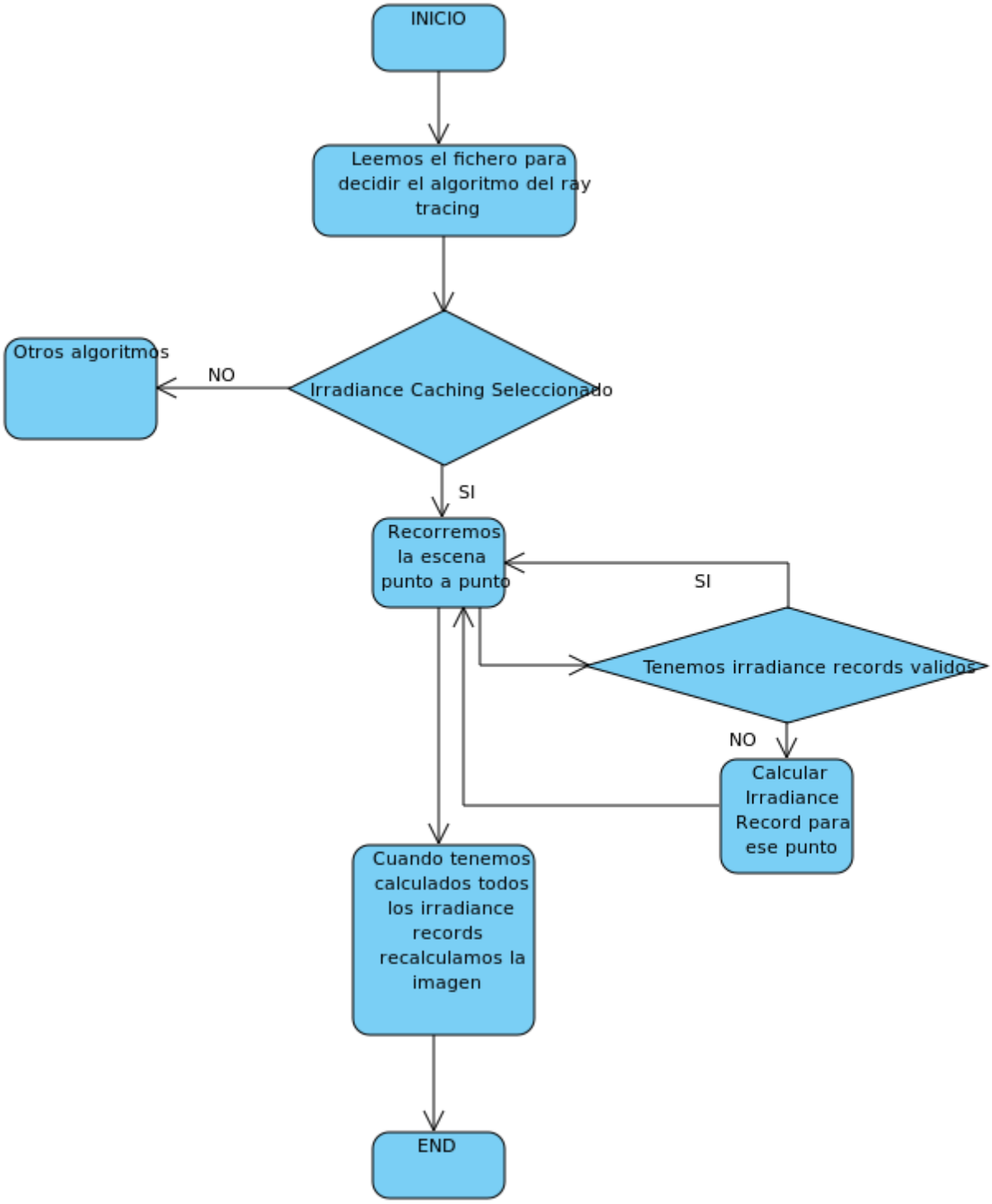

Ilustración Diagrama de flujo

Visto el anterior diagrama de flujo pasaremos a explicar más detalladamente el flujo de nuestro programa.

Como hemos visto primero cargamos el fichero en nuestro programa, esta funcionalidad nos la proporciona la clase scene, en este fichero se nos adjunta toda la información relacionada con nuestra escena como los objetos, materiales, fuentes de luz, cámaras y el algoritmo a utilizar y sus parámetros, en nuestro caso este algoritmo será Irradiance Caching.

Cuando tengamos el shader que queramos utilizar en este caso Irradiance Caching, se llamará a la función prerender de la clase irradiance, esta función tiene el objetivo de crear la caché de irradiancia y rellenarla.

Para esto iremos recorriendo la imagen pixel a pixel, y para cada pixel calcularemos el punto 3D de la escena visible correspondiente, todo esto lo haremos calculando el rayo de ese pixel y calculando su intersección, si para ese punto 3D tenemos puntos en la caché de irradiancia que no superen el criterio de error propuesto, pasamos al siguiente píxel, si no tenemos puntos en la caché de irradiancia calculamos la iluminación en ese punto explícitamente gracias al algoritmo de Path tracing y lo guardamos en la caché de irradiancia.

Cuando lleguemos al final de la imagen tendremos una caché de irradiancia capaz de dar valor a todos los puntos de la imagen, sin sobrepasar el criterio de error.

Por lo tanto, ahora pasaremos a calcular la imagen pixel por pixel, para cada pixel cogeremos los n puntos más cercanos guardados en la caché de irradiancia y de estos los que pasen la condición impuesta que explicaremos más abajo, son los irradiance records que utilizaremos para la interpolación para tener en la aproximación de la iluminación en ese punto.

6.3 Diseño de clase Irradiance

En este apartado explicaremos la función de cada una de las variables y funciones de nuestra clase irradiance, y más adelante entraremos en detalle en algunos aspectos técnicos del algoritmo.

Primero de todo explicaremos las variables de nuestro algoritmo y su utilidad.
Primero de todo tendremos los dos tipos de caché de irradiancia que utilizaremos como explicaremos en el apartado de implementación.

- **IrradianceCache** → Esta variable será una variable de tipo vector en la cual guardaremos irradianceCacheRecords que como hemos dicho antes es una clase que utilizaremos para guardar todas las variables necesarias para un irradiance record. Este será el primer tipo de caché de irradiancia de nuestro algoritmo.
- **IrradianceCacheOctree** → Como la anterior esta variable se usará como caché de irradiancia, será la segunda y última opción de caché de irradiancia. Esta variable será un objeto de la clase octree que hemos desarrollado para este tfg.
- **Path** → Aquí tendremos una instancia del shader PathShader que utilizaremos para calcular la iluminación explícita cuando sea necesario.
- **nSamplesHemis** → Variable que nos marcará el número de rayos que lanzaremos alrededor de una semiesfera, que crearemos alrededor del punto que queramos estudiar, esto nos servirá para decidir el número de muestras que queramos calcular para nuestros irradiance records.
- **show_points** → Variable booleana que nos permitira decidir si queremos que en nuestra imagen se marquen en negro los irradiance records calculados.
- **preprocessDone** → Este booleano nos dirá si el preprocess en nuestro algoritmo está hecho o no esto nos servirá para algunas decisiones a tomar en la función computeColor3D.
- **useOctree** → Esta variable será de tipo booleano y nos permitirá cambiar el tipo de caché de irradiancia que queramos usar.

Por otro lado, explicaremos la función de cada una de las funciones que tenemos implementada en esta clase:

- **Constructor** → Como todo constructor nos permitirá inicializar el objeto irradiance con todas sus características.

- **Preprocess** → Como hemos explicado antes esta función será la encargada de llenar la caché, para ello también usaremos la función computeColor3D.
  Ahora mostraremos un pseudocodigo de esta función:

Tabla Pseudocodigo Preprocess

Datos de Entrada: Scene Resultado: Irradiance Cache
1: for x < ancho de imagen *2: for y < alto de imagen* *3: transformamos x e y a coordenadas 3D* *4: si x e y = 0* *5: llamamos a computeRecord* *6: si no* *7: llamamos a computeColor* *8: end-for* *9: end-for* *10: preprocessDone = true*

Esta función recorrerá la imagen pixel a pixel, transformando las coordenadas 2D de nuestros pixeles de imagen a coordenadas 3D, cuando tengamos las coordenadas llamaremos a la función computeColor3D, si nos encontramos en el primer pixel de la imagen directamente llamaremos a la función computeRecord, ya que 100% necesitamos empezar con un compute record para no tener la caché de irradiancia vacía. Finalmente cambiaremos la variable preprocessDone a true para indicar que el preprocess ha acabado.

- **ComputeColor3D** → Función que se encarga de calcular el color para cada punto que necesitemos a partir de sus vecinos más próximos,pero si estamos en la fase de preprocess, es la función que decide cuando es necesario calcular un irradiance record y cuando no.

Tabla Pseudocodigo ComputeColor3D

Datos de Entrada: Scene, Ray del pixel al punto 3D Resultado: Color en el pixel
1: buscamos los n-vecinos más cercanos *2: for cada vecino* *3: calculamos wsub-i* *4: si el vecino es valido* *5: sumtop = irradiancia_vecino * wsubi* *6: sumbot = sumbot + wsub-i* *7: end-for* *8: si sumbot > 0:* *9: E = sumtop/sumbot* *10: si no* *11: E = computeRecord()* *12: return E*

- **near_n_neighbour** → Función que se encarga de dar los n vecinos más cercanos al punto que estamos estudiando, esta función es la encargada de trabajar con la caché de irradiancia en forma de vector.
- **near_n_neighbour_octree** → Función igual que la anterior nos devuelve los vecinos más cercanos, pero en este caso trabaja con la caché de irradiancia en forma de octree.
- **computeHarmonic**→ Función que nos devuelve la distancia entre el punto y la intersección del rayo lanzado, esta función será llamada por computeRecord para calcular la distancia armónica del irradiance record.
- **computeRecord** → Función que llamamos cuando hemos de calcular un irradiance record, se encarga de calcular este mismo y de añadirlo a la caché de irradiancia.

Tabla Pseudocodigo computeRecord

Datos de Entrada: Scene, Ray del pixel al punto 3D Resultado: Color en el pixel
1: transformamos x e y a coordenadas 3D *2: creamos semiesfera alrededor de las cordenadas 3D* *3: for nSamplesHemis* *4: generamos un rayo en una dirección aleatoria* *5: harmonicdistance += computeHarmonic* *6: E += cálculo de E con pathshader* *7: end-for* *8: harmonicdistance = harmonicdistance/nSamplesHemis* *9: E = E / nSamplesHemis* *10: guardamos el record en la cache*

- **Render** → Función que nos calcula la imagen final a partir de los puntos de irradiancia guardados en la caché de irradiancia, función muy parecida a prerender, ya que también tiene que recorrer la imagen pixel a pixel.

Tabla Pseudocodigo Render

Datos de Entrada: Scene, Irradiance Cache Resultado: Imagen final
1: for x < ancho de imagen *2: for y < alto de imagen* *3:* *4: transformamos x e y a coordenadas 3D* *5: llamamos a computeColor* *6: end-for* *7: end-for* *8: guardamos el color en la imagen*

- **sampleIsValid** → Función que nos permitirá decidir si un irradiance record cercano es válido para ese punto, esta decisión la tomaremos respecto a 3 variables que explicaremos más adelante.

6.3.1 Análisis de la interpolación

Ahora pasaremos a explicar la interpolación que hemos utilizado en nuestro algoritmo.

Hay muchas formas de interpolar diferentes puntos de una escena 3D, y este será uno de los puntos a añadir la sección de trabajo futuro, estudiar diferentes formas de interpolación para este algoritmo y como afectan al rendimiento y resultados de el mismo.

Nosotros en nuestro algoritmo hemos usado varios tipos de interpolación, estas están explicadas en siguiente sección de Implementación, per en este apartado explicaremos con mas detalle la interpolación que hemos utilizado para el resultado final del algoritmo.

Esta formula es generalmente la mas usada en este algoritmo y es la siguiente:

$$E = \frac{\sum_{i=1}^{n} w_i E_i}{\sum_{i=1}^{n} w_i}$$

$$w_i = \frac{1}{\frac{\left\|\vec{P} - \vec{P_i}\right\|}{Ri} + \sqrt{1 - \vec{N} \cdot \vec{N_i}}}$$

Estas dos formulas son las formulas que usaremos para nuestra interpolación, la primera de ellas se basa en la media de la suma del peso de cada irradiance record por su irradiancia.

La segunda formula es el cálculo del peso de cada irradiance record sobre el punto que estemos estudiando, este peso también se utilizará para el calculo de una de las condiciones para decidir si nuestra muestra es valida o no.

Cosas a tener en cuenta para esta formula es que la variable R_i es la distancia armónica del irradiance point que estemos calculando, por lo demás es una formula bastante sencilla, utilizamos la norma de los dos puntos para reflejar la distancia, y usamos las normales ya que la ponderación de un punto sobre otro esta depende mucho de la normal, ya que si la normal es muy diferente significara que la irradiancia en esa parte cambia mucho por lo tanto la ponderación de ese punto tiene que ser mas pequeña.

6.3.2 Irradiance Records válidos y porque

En este apartado explicaremos cómo seleccionamos los irradiance records que son válidos para el punto actual que estemos estudiando.

Para esto utilizamos la función sampleIsValid, la cual comprobará 3 condiciones para que nuestro irradiance record sea válido.

Estas 3 condiciones serán las siguientes:

La primera de estas condiciones es comprobar que el irradiance record que estamos estudiamos esta en el radio valido, para esto tenemos la variable “harmonicMean”.
Esta variable nos indica la distancia media a la que tenemos objetos en ese punto, esto se calcula lazando rayos aleatorios y viendo donde chocan y a que distancia chocan, esto quiere decir que los puntos que tengamos cerca de las esquinas tendrán una distancia armónica media menor que los puntos que tengamos por el centro de las paredes.

Por lo tanto, calcularemos esta condición de la siguiente forma:

$$HarmonicMean_i > Distancia\ entre\ HarmonicMean_i\ y\ el\ Punto\ a\ estudiar$$

La segunda condición la usaremos para comprobar que el irradiance record no está en un plano diferente al punto que queremos estudiar, esto lo haremos porque un punto puede estar cerca de otro punto, pero estar en dos planos diferentes, una repisa por encima etc...
En estos casos no queremos que estos vecinos ponderen en la interpolación ya que físicamente están cerca, pero no están en la misma sección o área.

La siguiente imagen nos explica muy bien el caso que estamos explicando:

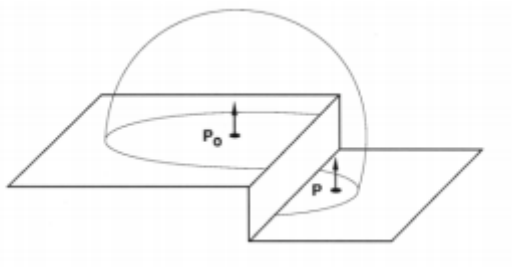

Ilustración Plano Diferenciado

La formula para aplicar lo explicado anteriormente es la siguiente:

$$\frac{\vec{P} - \vec{P_i} \cdot [\vec{N}(\vec{P}) + \vec{N}(\vec{P_i}]}{2} \geq 0$$

Como tercera y última condición es la que usaremos para decidir si un irradiance record puede ponderar para la interpolación de punto que estamos estudiando o no, esto lo decidiremos usando la variable $wSub_i$, esta variable como hemos explicado antes es el peso del irradiance record que tiene sobre el punto que estemos estudiando actualmente.
Esto lo haremos decidiendo un peso mínimo para esta variable, si la variable esta por encima de este valor no nos servirá, si la variable esta por debajo si nos servirá.

La comparación es la siguiente:

$$wSub_i > 1/A$$

Esa A será el valor que iremos cambiando para decidir si queremos mas irradiance records o menos puntos en nuestra imagen, este valor será el que modificaremos en las graficas de que mostraremos en la sección resultados, ya que, si es la variable que nos mide con mayor precisión el número de irradiance records que utilizaremos en nuestro algoritmo, también es la variable que nos modificara con mayor medida el tiempo que tardara nuestro algoritmo.

Por lo tanto, combinando estas tres condiciones decidiremos si un irradiance record es valido para el punto que estamos estudiando o no.

Remarcar que para que un irradiance record sea valido tienen que cumplirse las 3 condiciones, si una condición no se cumple automáticamente el irradiance record pasa a ser no valido.

7.Implementación

En esta parte de la memoria explicare como transcurrió la implementación de nuestro algoritmo y el porqué de las decisiones tomadas.

Primeramente, explicar que el planteamiento de este tfg no era el recrear el algoritmo de irradiance caching perfectamente desde el principio, este tfg se planteó desde el punto de vista de ir aprendiendo poco a poco las bases del algoritmo e ir mejorándolo semana a semana.

Esta decisión se tomó por el simple hecho de que implementar este algoritmo a la perfección con las bases con las que partía el alumno era muy difícil y lo sigue siendo a día de hoy, ya que actualmente tenemos una versión de este algoritmo bastante buena, pero ni mucho menos perfecta.

Esta decisión también se tomó porque entrar en según qué temas de este algoritmo sin tener ni una base programada de él mismo era bastante abrumador, también cabe decir que hacer un prototipo con un par de ideas básicas del algoritmo y ver resultados, desde mi punto de vista, es bastante motivante, mucho más que pasarse un par de semanas leyendo pdfs sobre qué tipo de interpolación es mejor o sobre qué forma de distribuir los puntos es la más optima etc…

Por lo tanto, separaremos la implementación de este algoritmo en varios prototipos desde el primero hasta el último, explicando en cada uno qué cambios se hicieron y porqué, qué resultados obtuvimos, y mostrando resultados visuales de estos mismos.

7.1 Primera implementación: Irradiance Caching Sencillo en 2D

Como su propio nombre indica esta fue la primera implementación de este algoritmo para ello nos planteamos los dos principios básicos del algoritmo irradiance caching:

- Interpolación de puntos similares y/o cercanos (pudiendo ser estos puntos 2D o 3D más adelante explicaremos porque).
- Cálculo de la “cache” de puntos de irradiancia.

Para el primer punto la decisión fue hacer una interpolación de puntos en 2D o visto de otra forma íbamos a interpolar los píxeles de la imagen, esta era la implementación más sencilla, esto nos desembocaba a la siguiente cuestión y por lo tanto el siguiente punto descrito, teníamos que pensar que puntos de irradiancia íbamos a calcular y porque, para rellenar nuestra “cache” de puntos.

Por ahora teníamos resuelto sobre qué hacer la interpolación, pero faltaba decidir cómo hacer esta interpolación, como veréis este primer prototipo tenía el objetivo de ser muy sencillo para poder moldearlo a placer, por lo tanto, no quería usar ninguna fórmula que tuviera en cuenta la distancia etc.…

Por lo tanto, usamos la siguiente:

$$E = \frac{\sum_{i=1}^{n} E_i}{n}$$

Como podemos ver la interpolación era la media de los valores que se elegían, en este caso tomamos los 4 irradiance points más cercanos por proximidad, esto como veremos en las imágenes nos dará lugar a que la imagen parezca que está hecha con pixeles enormes, esto se debe a que con esta interpolación todos los píxeles de la imagen que estén entre los mismos 4 irradiance points tendrán el mismo valor y por lo tanto el mismo color.

Como ya hemos visto el cálculo de estos puntos se da por algún criterio dado, la idea principal es que hay que calcular un nuevo irradiance point cuando el anterior ya no nos sirva, normalmente dado por una serie de consecuencias, en el punto que estaba del algoritmo (era la primera iteración) no quería complicar mucho las cosas por ahora, quería algo sencillo que funcionase y diese algún resultado probablemente malo, pero por algo hay que empezar.

Por lo tanto, al no querer calcular estos irradiance points mediante un error referente a cada punto, decidí usar un patrón constante para el cálculo de irradiance points, y dado a que estábamos haciendo la interpolación en 2D, o en otras palabras, interpolando los píxeles de la imagen, decidí calcular un irradiance points cada X píxeles de la imagen.

Este valor durante toda la implementación rondó entre tres números 20, 10 y 5.

Por último y no menos importante tenía que tener alguna estructura de datos para guardar estos irradiance points, más conocida como “cache de irradiancia”, en ese momento como todos los cálculos eran en 2D, decidí usar una matriz donde la posición del punto respecto a la imagen era la posición que ocuparía en esta matriz, por lo tanto, si calculamos el pixel 0,0 como un irradiance point este se guarda en la posición 0,0 de la matriz.

7.1.1 Imágenes Generadas

El balance de este primer prototipo fue bastante bueno, aunque la imagen no fuera ni por asomo buena, tenía un prototipo que hacía exactamente lo que quería y desde el cual se podía empezar a profundizar más en cualquier de los aspectos que necesitara el algoritmo.

Imágenes resultantes:

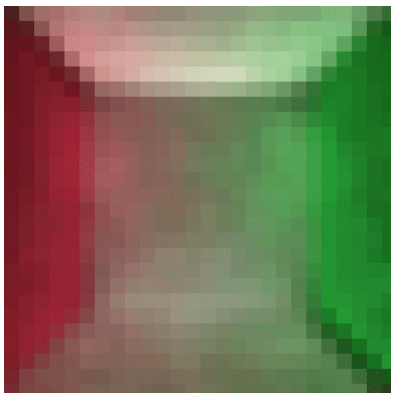

Ilustración 2D con salto 20

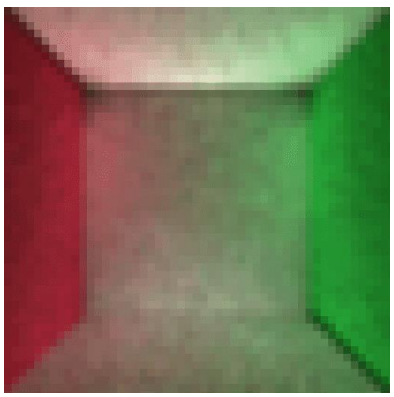

Ilustración 2D con salto 10

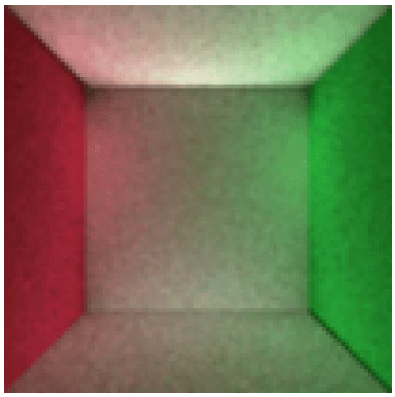

Ilustración 2D con salto 5

Estas tres imágenes están calculadas de la misma forma, están ordenadas en referencia al número de píxeles que hay entre irradiance point e irradiance point, podemos observar en la primera como habíamos dicho se da el efecto de pixeles grandes, ya que muchos pixeles están cogiendo el mismo valor por la forma de interpolar en este primer prototipo.

Uno de los principales problemas que tenemos en este tipo de integración es que no tenemos en cuenta la normal de los puntos por lo tanto hay transferencia de colores entre las paredes.

También observar que la imagen de la derecha es de una calidad tan elevada ya que hay muchos irradiance points calculados en ella, por lo tanto, también es mucho más lenta de calcular que las otras.

7.2 Irradiance Caching Sencillo en 2D con Interpolación Bilineal

En este punto de la implementación había una base estable y funcional, pero claramente era muy mejorable en todos los aspectos.

La elección de mejora fue cambiar el tipo de interpolación a una bilineal, esta interpolación nos permitiría eliminar el brusco cambio que había (cuadrados enormes con el mismo color) a un cambio más suave entre píxeles.

Como aún estábamos trabajando en el plano 2D esta interpolación era bastante buena para el caso que estaba tratando, esta interpolación se resume en una doble interpolación lineal, primero para unas coordenadas x y luego para unas coordenadas y, en nuestro caso pensamos en los pixeles como si fueran coordenadas de un plano 2D.

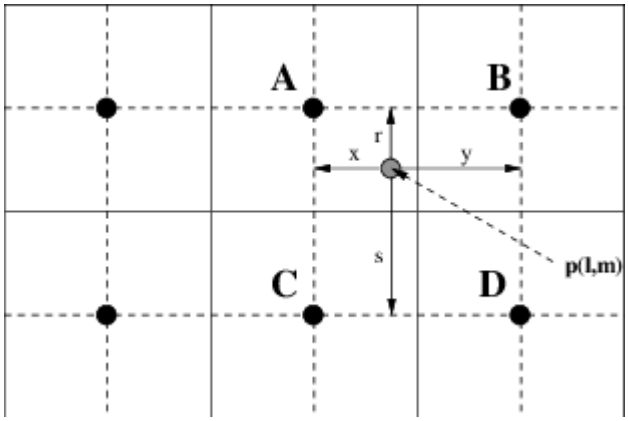

Ilustración Interpolación Bilineal

Como vemos en la imagen A, B, C y D serán los irradiance points y el punto central a estos serán los píxeles para los cuales queremos calcular el valor que les corresponde dependiendo de la distancia de la que estén de cada punto en concreto.

Las fórmulas que utilizamos son las siguientes:

$$Eab = \frac{Xp - Xa}{Xb - Xa} \cdot Eb + \frac{Xb - Xp}{Xb - Xa} \cdot Ea$$

$$Ecd = \frac{Xp - Xa}{Xd - Xa} \cdot Ed + \frac{Xd - Xp}{Xd - Xa} \cdot Ec$$

$$Ep = \frac{Yp - Yc}{Yd - Yc} \cdot Ecd + \frac{Yd - Yp}{Yd - Yc} \cdot Eab$$

Calcularemos primero la irradiancia en la coordenada X para los pares de puntos AB y CD, y después calcularemos la irradiancia final en la coordenada Y, como podemos ver una interpolación bilateral no es más que una interpolación lineal, pero con dos coordenadas en las que operar en vez de una.

7.2.1 Imágenes Generadas

Imágenes resultantes de este segundo prototipo:

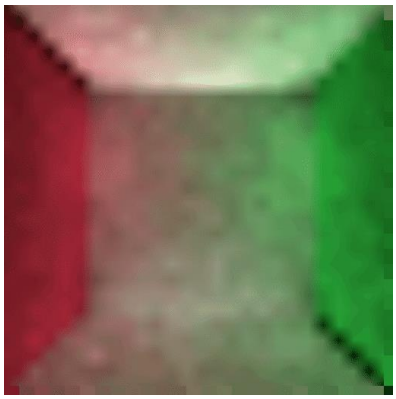

Ilustración Bilineal con salto 20

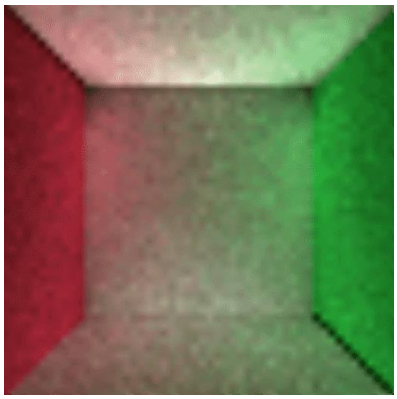

Ilustración Bilineal con salto 10

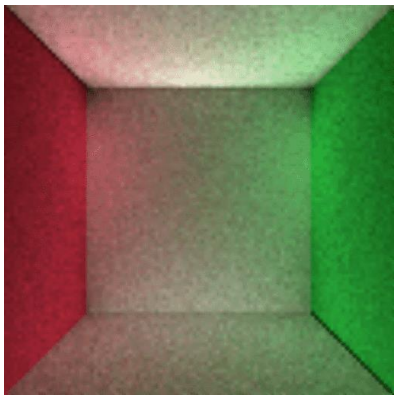

Ilustración Bilineal con salto 5

Como la vez anterior he mostrado 3 imágenes dependiendo del número de píxeles entre cada irradiance point (recuerdo que por ahora tenemos irradiance points puestos “a dedo” cada X píxeles de la imagen), este número de pixeles son 20,10 y 5 en el mismo orden en que se muestran las imágenes.

Comparando los resultados con los anteriores y matizando que solo hemos cambiado el tipo de interpolación, podemos ver un cambio bastante grande en las imagenes, sobretodo en la primera que es la que más nos interesa ya que es la menos irradiance points tiene y la que más rápido se calcula.

Podemos ver que las agrupaciones de muchos pixeles con el mismo color han desaparecido y ahora el cambio de colores es más sutil en la mayoría de la imagen.

En general un cambio bastante grande comparado al número de líneas de código que hemos cambiado que ha sido solamente añadir 3 líneas de código, las correspondientes a la implementación de las tres fórmulas mostradas antes.

7.3 Irradiance Caching: de 2D a 3D

En este punto del proyecto había que cambiar una cosa que no tenía sentido que estuviera como estaba actualmente.

Había sido útil para un primer esbozo del algoritmo, pero era hora de cambiarla, este era el punto de vista 2D que por ahora tenía nuestro algoritmo.
Cambiar el punto de vista a 3D nos permitía seguir avanzando hacia un prototipo más "correcto" de nuestro algoritmo irradiance caching.

Este cambio no era tan fácil como el anterior ya que para cambiar esto, tenía que también cambiar la estructura de datos en el cual se almacenaban los cache records o puntos de irradiancia y por otro lado teníamos que cambiar nuestra interpolación ya que esta estaba pensada para trabajar en vecinos 2D y no en vecinos 3D.

Así que vamos por pasos, lo primero que hice fue cambiar la estructura de datos, recuerdo que actualmente nuestra "cache" era simplemente una matriz, cambiaremos esta matriz a un vector de "IrradianceCacheRecords".

Un vector en este caso concreto era bastante ineficiente ya que cuando tenga muchos irradiance cache records, para cada pixel tendremos que recorrer todo el vector de principio a fin para encontrar sus vecinos más cercanos, pero como la estructura de datos perfecta para este tipo de algoritmos es bastante compleja por ahora nos servirá.

También tendremos una estructura de c++ nueva llamada IrradianceCacheRecord(en este punto era una simple estructura dentro de nuestra clase Irradiance, pero más adelante será una clase por un par de problemas que tuve durante el desarrollo, pero la función y la misión de esta es exactamente la misma), que lo que hará como su propio nombre indica es guardar toda la información relacionada con cada cache record.

Una vez desarrollada la estructura de datos y todo lo relacionado con ella (por ejemplo, buscador de vecinos en 3D), pasamos a cambiar la interpolación.

Este es el tercer y último cambio de interpolación que haremos en este trabajo (más adelante modificaremos esta interpolación, pero será para añadirle valores de los que actualmente no disponemos).

Para calcular la irradiancia en cada punto de la escena (a partir de ahora hablaremos de escena y no de imagen ya que estamos empezando a trabajar en 3D) utilizaremos el peso de cada punto de irradiancia válido para el punto en cuestión que estemos estudiando, y para calcular este peso usaremos la fórmula propuesta por Greg Ward.

Las fórmulas serán las siguientes:

$$E = \frac{\sum_{i=1}^{n} w_i E_i}{\sum_{i=1}^{n} w_i}$$

$$w_i = \frac{1}{\frac{\left\|\vec{P} - \overrightarrow{P_i}\right\|}{Ri} + \sqrt{1 - \vec{N} \cdot \overrightarrow{N_i}}}$$

Esta será la fórmula que utilice al final de la implementación, pero como actualmente no tenemos r_j que es la distancia armónica de cada punto de irradiancia ese dato de la ecuación será negligido.

En este punto también cambiaremos un punto importante, que no hacía hasta ahora.
En las anteriores interpolaciones estábamos interpolando el color del pixel que teníamos, por lo tanto, la interpolación era de color y no de irradiancia, en este punto empezaremos a hacer la interpolación de irradiancia y después de esto aplicaremos esta irradiancia en la superficie que queramos saber el color.

Esta es la forma correcta de interpolar ya que el color puede cambiar bruscamente, pero la irradiancia cambia suavemente conforme nos movemos por la escena.

Por último, incluir que en este momento se implementó una opción para poder ver los irradiance records en la imagen de color negro, estos son los puntos que explícitamente se están calculando.
Esto nos ayudará más adelante a ver como cambiamos el reparto de puntos y en este caso en concreto podremos ver con mayor claridad la densidad de irradiance records de las imágenes mostradas en la parte de resultados más abajo.

7.3.1 Imágenes Generadas

En estos resultados mostraremos todas las imágenes juntas debajo ya que será más fácil visualmente ver las diferencias entre ellas, en estos resultados tendremos 6 imágenes.
Las imágenes de la fila de la izquierda serán las imágenes que tengan la opción nueva que hemos comentado anteriormente de mostrar los irradiance records de color negro, las imágenes de la derecha serán las mismas, pero sin esta opción activada.

De abajo a arriba tendremos los resultados que siempre mostramos con los tres tipos de distancias entre irradiance records cada 5,10 y 20 píxeles.

Los resultados a este nivel ya eran bastante buenos como se puede ver en la imagen excepto por algunos artefactos en la parte de abajo a la derecha la imagen tiene muy buena calidad, y esta es la más rápida de calcular.
Las siguientes dos imágenes siguen teniendo algún error en la esquina de abajo a la derecha, pero al haber muchos más irradiance points estos se ven menos.

En este punto podríamos empezar a hablar de velocidad ya que visualmente las imágenes ya se están acercando mucho a la imagen objetivo del algoritmo path tracing, pero la parte de la eficiencia la hablaremos en la sección de resultados finales ya que allí haremos varias comparaciones en calidad y eficiencia con el algoritmo Path tracing.

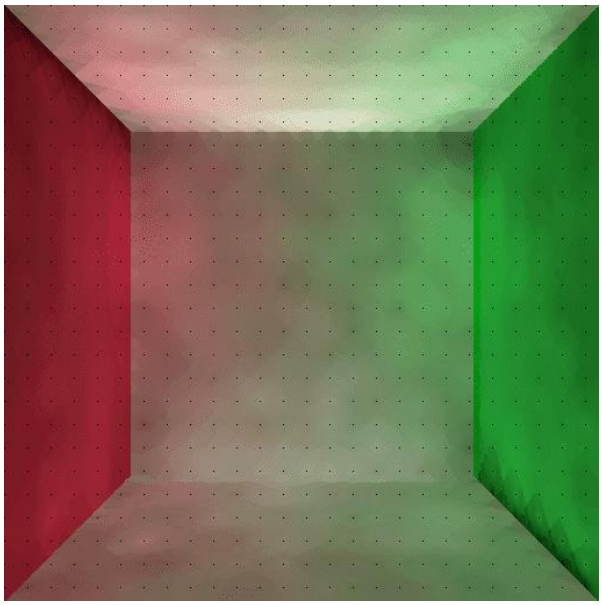

Ilustración con salto 20 punteada

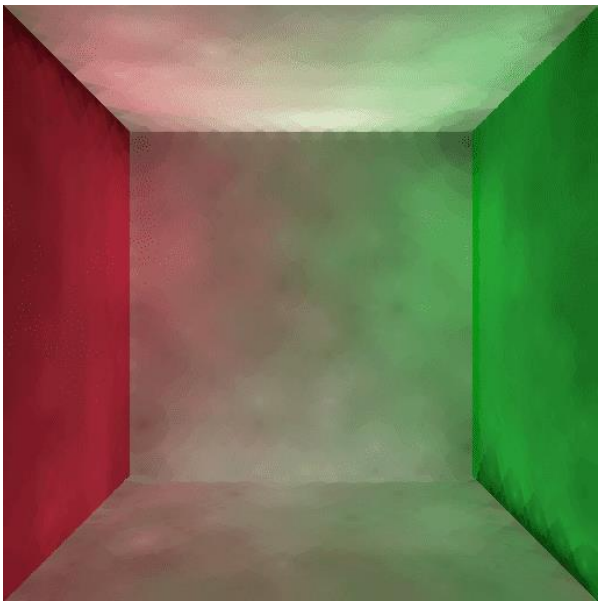

Ilustración con salto 20

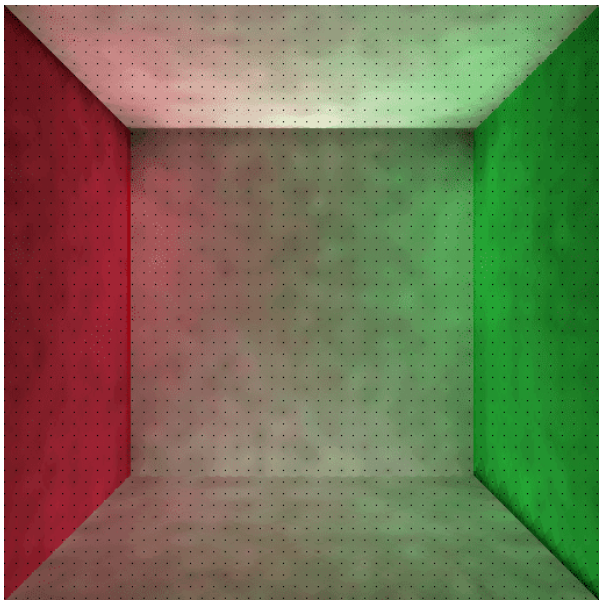

Ilustración con salto 10 punteada

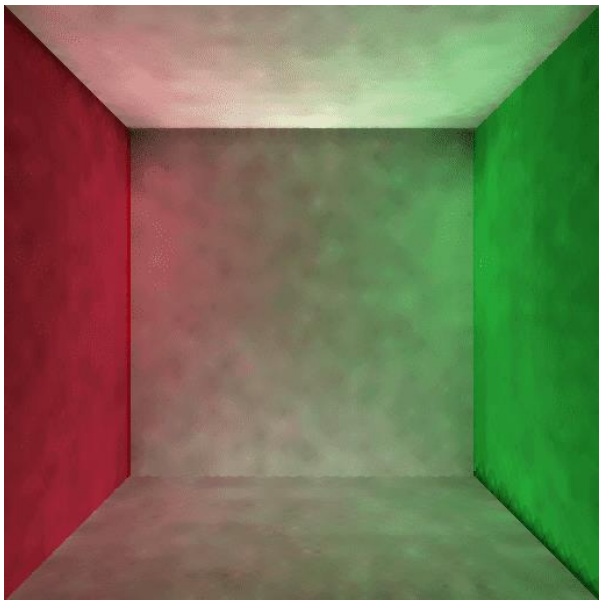

Ilustración con salto 10

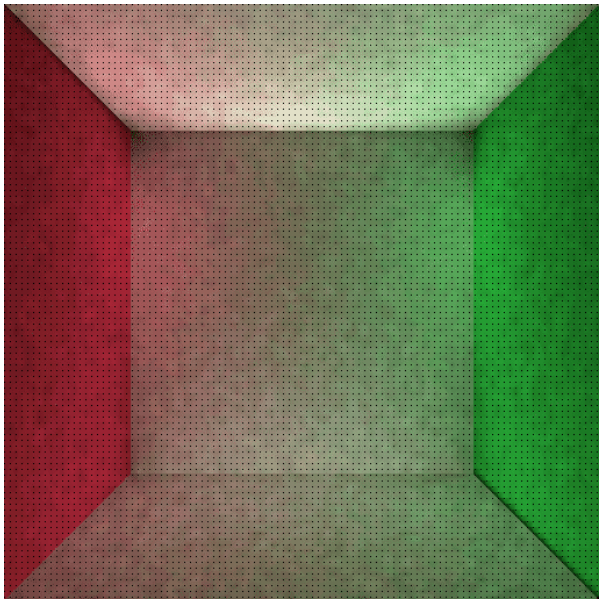

Ilustración con salto 5 punteada

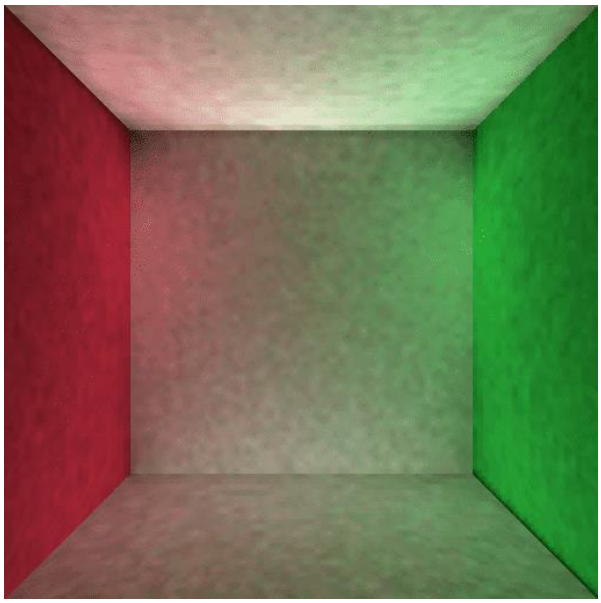

Ilustración con salto 5

Finalmente, también os quiero mostrar la diferencia de una imagen con la interpolación actual pero con el sistema antiguo de búsqueda de irradiance records cercanos en 2D para apreciar, que no todo el buen resultado de estas imágenes se debe a la interpolación, que el cambio a 3D de todo el algoritmo también ha ayudado en calidad.

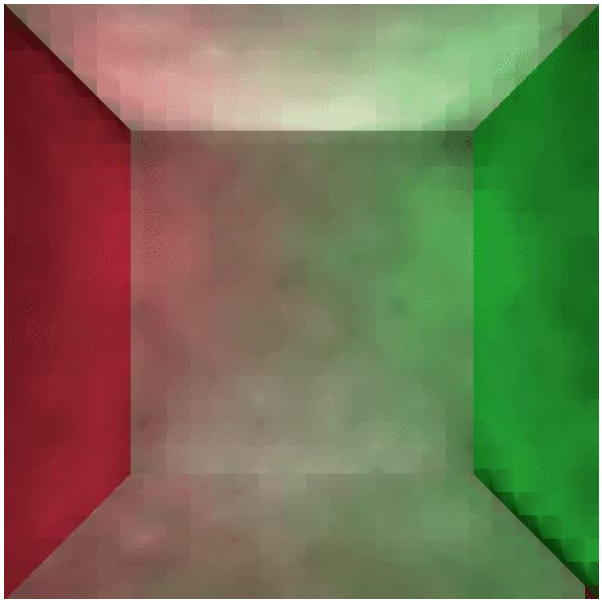

Ilustración 2D

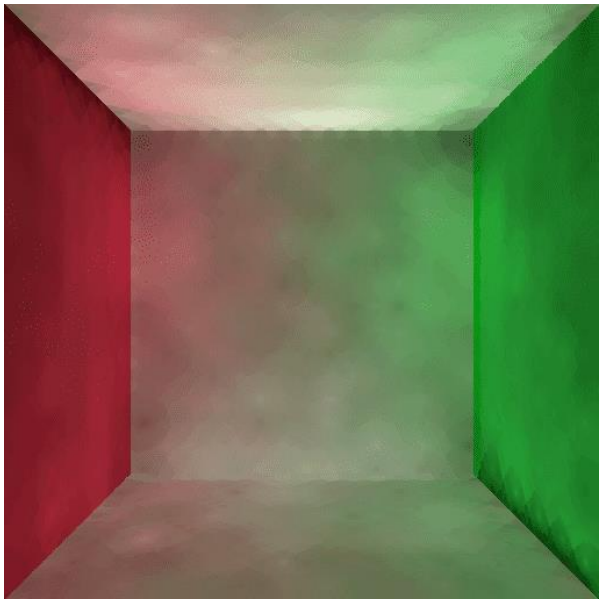

Ilustración 3D

En la imagen de la izquierda se pueden apreciar artefactos cuadrados en las esquinas y algún que otro cambio de color rectangular, mientras que en la imagen de la derecha artefactos y sombras tienen forma circular.

7.4 Irradiance Caching en 3D: Distribución adaptativa de irradiance cache records

A esta altura solo faltaban dos cosas para tener una primera versión del algoritmo de Irradiance Caching funcional, tener la interpolación 100% terminada y repartir los puntos a través de la escena con un criterio de error que hiciera que los irradiance records se concentren en más en las esquinas.
Para hacer posible esta parte del algoritmo nos faltaba calcular una característica de cada irradiance record, esta característica es la distancia armónica.

Como estas tres características del algoritmo van ligadas de la mano por el hecho de que dos dependen de la distancia armónica, decidimos implementarlas a la vez.

El primer elemento de la lista claramente es el cálculo de la distancia armónica ya que los dos elementos siguientes dependen de ella.

La distancia armónica gracias a las herramientas proporcionadas por el ray tracer en el cual estamos trabajando es muy simple de calcular, usando la clase Ray.

La modificación de la fórmula está explicada en el anterior apartado por lo tanto no entraremos en detalle ni la mostraremos.

Por último y una de las partes más importantes de este algoritmo, la colocación de irradiance records según un error determinado para conseguir que estos se concentren con mayor densidad en las esquinas.

La parte teórica está explicada más arriba, por lo tanto, vamos a explicar que tuve que hacer para implementar esta teoría en mi algoritmo.
Primero de todo, claramente el planteamiento actual que tenía para repartir los irradiance records tenía que ir fuera, actualmente tenía que calcular los irradiance records a medida que se calculará la imagen.

Al calcular la imagen a medida que se iban calculando los irradiance records, llegue a la conclusión (gracias a la imagen siguiente) que, aunque calculara los irradiance records a medida que se iba creando la imagen, necesitaba luego volver a crear la imagen de nuevo, ya que muchos pixeles de la nueva imagen utilizarían muchos más irradiance records de los que tenían a su alcance en su primer render.

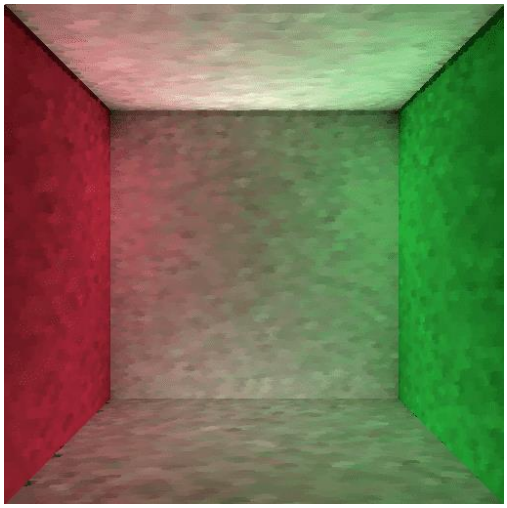

Ilustración curiosa

Como vemos en esta imagen, aunque en mi opinión es bastante bonita por los cambios de color que parecen pinceladas, es errónea esto se debe a como explique antes que muchos de esos puntos solo tenían un posible irradiance record al alcance.
Como veremos en la parte de resultados esto se arregla haciendo la fase de pre render para calcular los irradiance records y la fase de render para poder usar todos los irradiance records disponibles en cualquier punto de la imagen.

Así que finalmente el algoritmo que hicimos en el primer prototipo no distaba mucho del algoritmo final, el ciclo de programa al fin y al cabo es el mismo, una fase de pre render para calcular los irradiance records necesarios y guardarlos en nuestra irradiance cache y una fase de render para calcular la imagen a través de estos irradiance records calculados.

7.4.1 Imágenes Generadas

En esta parte primero enseñaré una imagen curiosa y la prueba de que la distancia armónica calculada en este algoritmo es correcta.

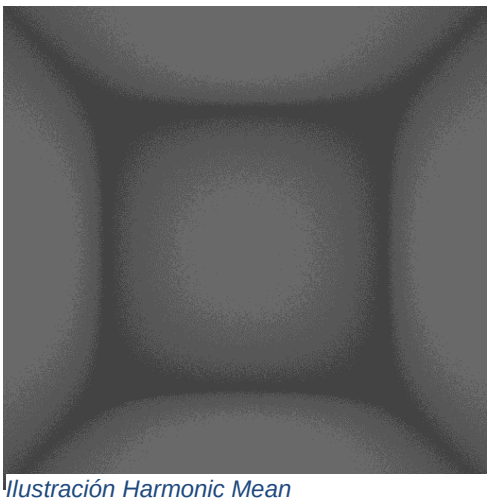

Ilustración Harmonic Mean

En esta imagen podemos ver la idea básica de la distancia armónica, la cual es la distancia media que hay hacia los otros elementos de la escena, esta teoría hace que en nuestra imagen se muestran las esquinas más oscuras y respecto nos vamos acercando al centro de las paredes, techo o suelo de la imagen va apareciendo un color más claro.

Ahora mostrare una imagen del resultado de la distribución de los irradiance records ya que en resultados finales tendremos más ejemplos ya que dependiendo del error que pongamos tendremos una distribución u otra.

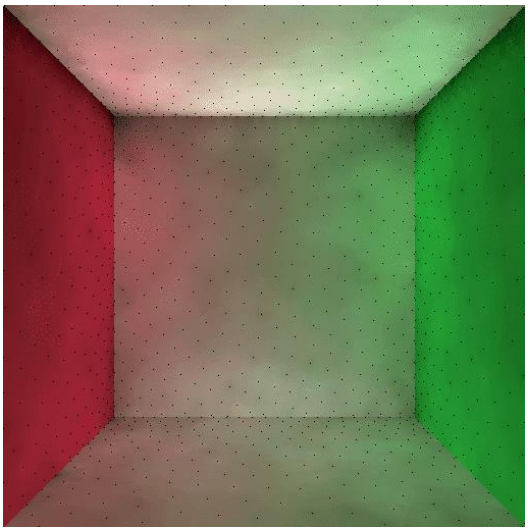

Ilustración Irradiance Records distribucion

Como podemos observar en la imagen los irradiance records se reparten por la imagen sin ningún problema, también podemos apreciar que en las esquinas es donde se concentran mayor número de puntos, el cual era el objetivo.

7.5 Irradiance Caching en 3D: Mejorando la estructura de datos

Por último, no nos tenemos que olvidar uno de los dos objetivos de este algoritmo, el primero está superado, que es el conseguir imágenes de calidad parecida al algoritmo path tracing con un menor número de puntos calculados, el siguiente objetivo e igual de importante es que todo esto último se haga de la forma más rápida y eficiente que podamos.

Si no nos hemos olvidado sabremos que actualmente nuestra irradiance cache es un vector el cual para buscar los irradiance records más cercanos a un pixel tiene que recorrer todos y cada uno de los elementos del vector.

El vector de la imagen anterior tenía 1270 elementos en él, cada uno de los pixeles de nuestra imagen tiene que recorrer este vector una vez mínimo buscando los irradiance records más cercanos a él, nos podemos hacer a la idea de que tiene que haber una forma más rápida de organizar estos irradiance records, para que el acceso a ellos sea mucho más rápido.

Ahí es donde entra una estructura de datos basada en árboles llamada octree, esta estructura de datos como hemos visto divide es espacio 3D en 8 subespacios, y estos 8 subespacios en otros 8 subespacios etc… las veces que creamos necesarias.

El objetivo era implementar un octree funcional, que acelerará la búsqueda de los irradiance records pero que no afectará al resultado final de la imagen.

7.5.1 Imágenes Generadas

Actualmente tenemos un octree en funcionamiento que cumple uno de los dos objetivos para que sea un octree óptimo, y este es el de la aceleración de los irradiance records hablaremos de números en el siguiente apartado, pero podemos decir con seguridad que este octree acelera nuestro algoritmo.

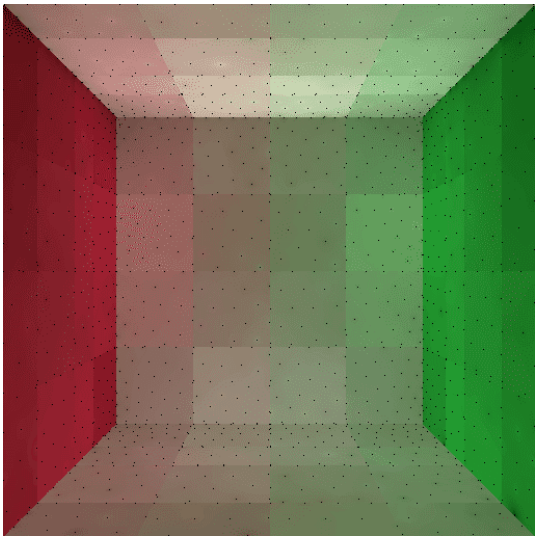

Ilustración Octree Resultado

Por otro lado, no podemos afirmar que este octree no afecta al resultado final de la imagen.

Como podemos observar cada “sección” del octree tiene un tono ligeramente diferente al vecino.
Este error es bastante simple y es simplemente que cada sección de nuestro octree no comparte ningún irradiance récords con sus vecinos más próximos, este error es bastante simple, y al ser bastante simple tiene fácil solución, hacer que los vecinos próximos puedan compartir irradiance records, cuando accedan a ellos de alguna forma.

Por lo tanto, podríamos decir que el octree está medio implementado, pero que tiene muy fácil solución ya que tenemos localizado el problema y sabemos cómo arreglarlo.

8.Resultados Finales

En este apartado compararé los resultados obtenidos con nuestro algoritmo Irradiance Caching y la implementación del algoritmo Path Tracing proporcionado por el tutor.

Dividiremos los resultados en dos apartados diferentes, comparando dos apartados totalmente diferentes, comparando la eficiencia del algoritmo o en otras palabras el tiempo de ejecución (teniendo en cuenta siempre la calidad de la imagen) y por otro lado teniendo en cuenta la calidad de la imagen sin dar importancia al tiempo de generación de esta, en otras palabras, compararé la imagen de mayor calidad de los dos algoritmos.

Finalmente haré una pequeña comparación de imágenes de los dos algoritmos en un tiempo de ejecución similar.

Para las comparaciones de calidad de imagen quiero explicar que usaré mi opinión totalmente subjetiva y sin ningún tipo de fundamento en un método.

8.1 Eficiencia

En este apartado mostraremos la eficiencia de Path Tracing y Irradiance Caching, este último con las dos estructuras de datos que tenemos implementada, vector y octree.

8.1.1 Path Tracing

En el siguiente gráfico vemos el tiempo que tarda el algoritmo Path Tracing respecto al número de muestras por pixel del algoritmo.

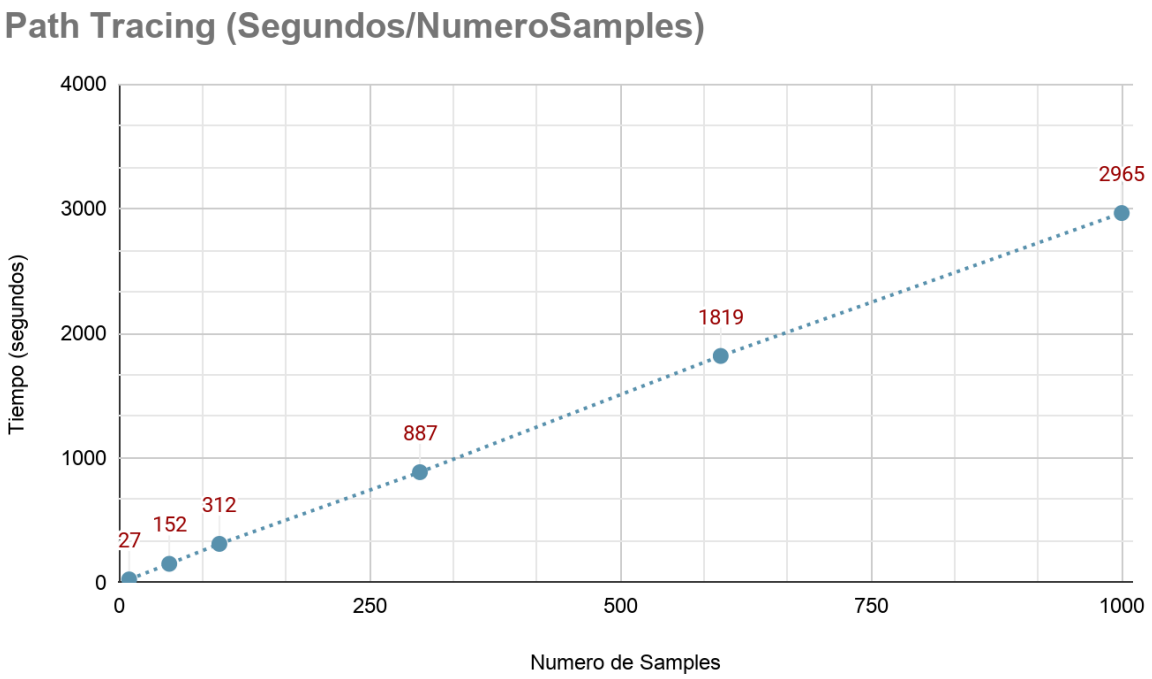

Ilustración Path Tracing Grafica

A continuación, mostraremos las imágenes resultantes a esta gráfica.

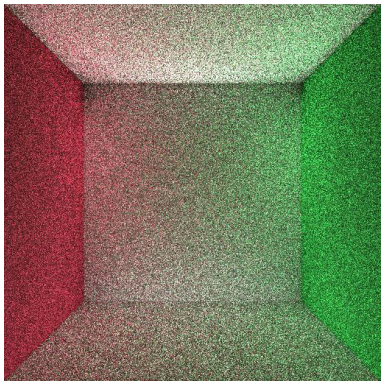

Ilustración 10 Samples

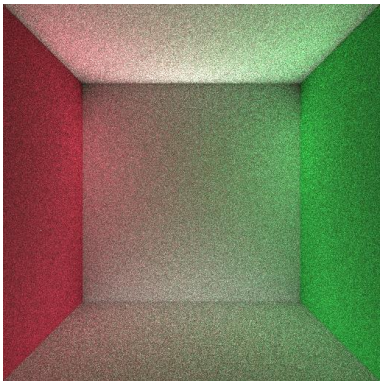

Ilustración 50 samples

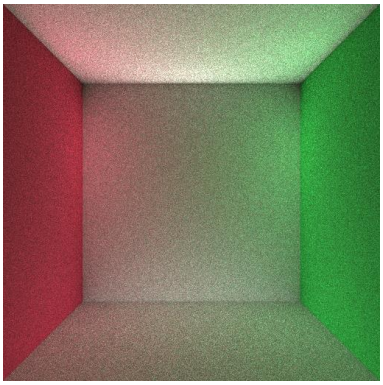

Ilustración 100 samples

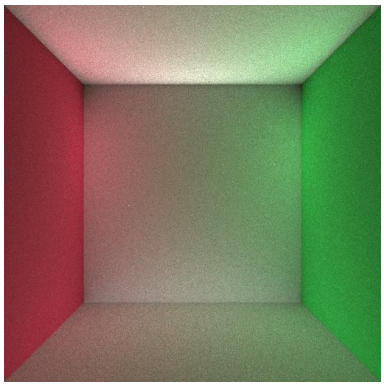

Ilustración 300 samples

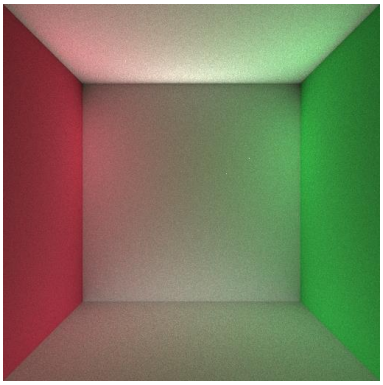

Ilustración 600 samples

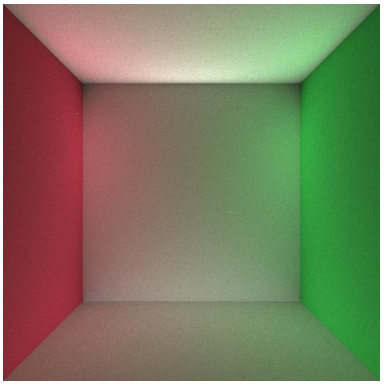

Ilustración 1000 samples

Como podemos ver la mejoría de la primera imagen a la última es bastante notable, pero entre ellas hay 45 minutos de ejecución de diferencia.

8.1.2 Irradiance Caching

Este apartado lo dividiremos en dos, ya que tenemos dos implementaciones, una implementación con un vector y otra con un octree.

En la siguiente gráfica vemos las dos implementaciones del algoritmo en función a la variable A, que nos afecta a la ponderación máxima que un irradiance record puede tener sobre un punto:

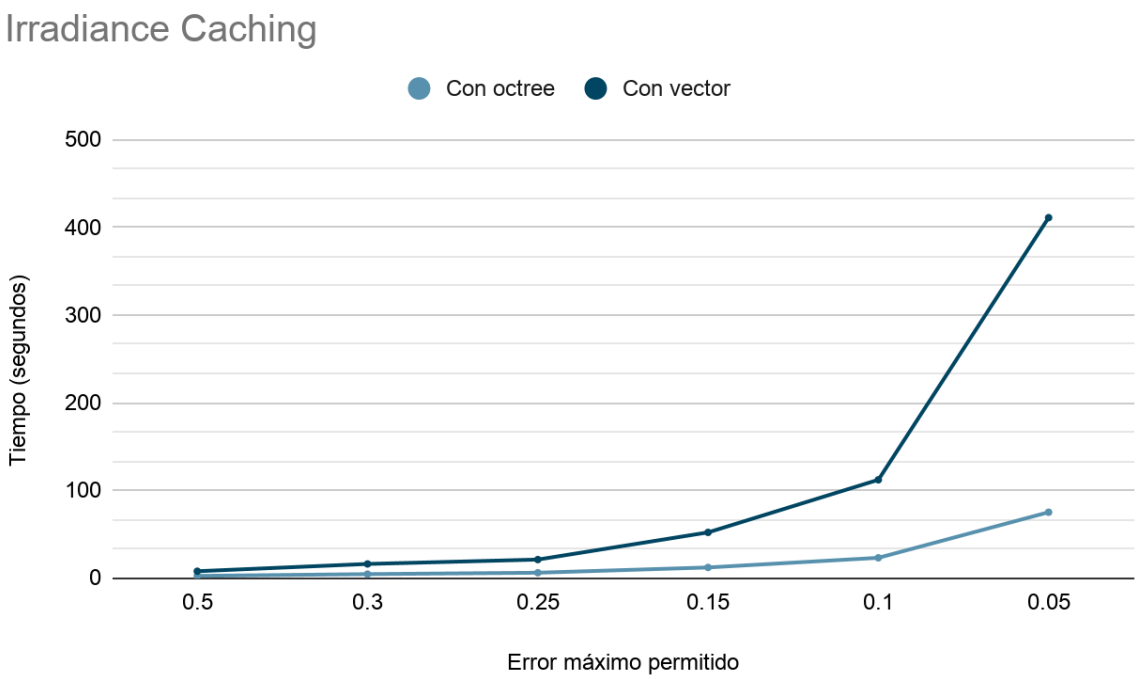

Ilustración Irradiance Caching Grafica

Como podemos ver la comparación entre las dos estructuras de datos se acentúa cuanto más, menor sea la variable, esto es debido a que esta variable afecta al número de irradiance records que tendremos en la imagen, cuanto menor sea la variable mayor número de irradiance records tendremos.
Esto hará que en el caso del vector tengamos que recorrer todo el vector para cada punto en el que queramos calcular la irradiancia, ya que tendremos que buscar los n irradiance records más cercanos, en cambio en el octree esta búsqueda se reduce muchísimo ya que por cada nivel de nuestro árbol tendremos 8 divisiones del espacio 3D, lo que significa que tenemos nuestros irradiance records divididos, por lo tanto, su búsqueda es mucho más rápida.
Imágenes resultantes de irradiance Caching con vector:

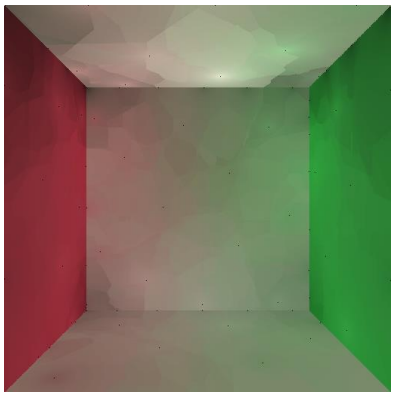

Ilustración A = 0.5

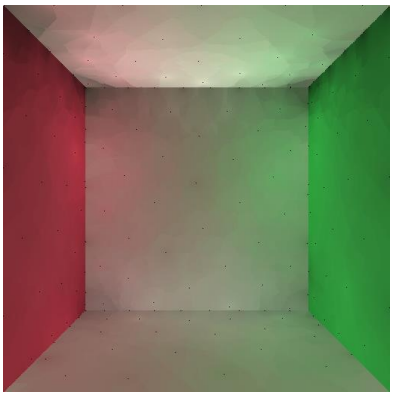

Ilustración A = 0.3

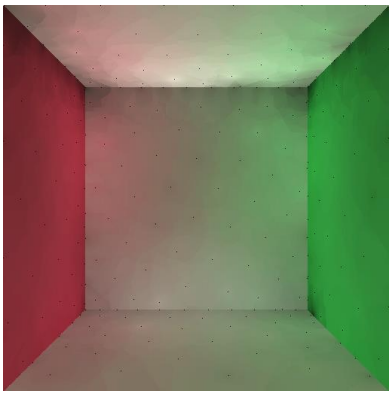

Ilustración A = 0.25

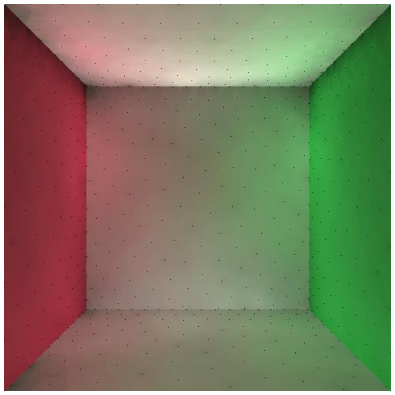

Ilustración A = 0.15

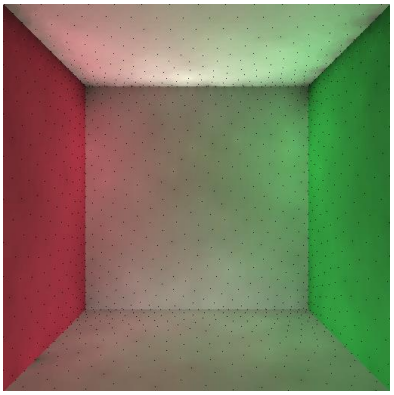

Ilustración A = 0.1

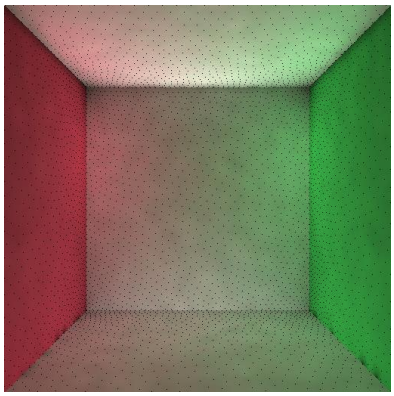

Ilustración A = 0.05

En estas imágenes se puede apreciar que la distribución de irradiance records (puntos en negro) aumenta según hemos explicado anteriormente.
No mostraremos las imágenes generadas con el octree ya que como hemos enseñado antes los resultados no son correctos porque el octree aún no está terminado.

8.1.3 Comparación de los dos algoritmos

Como podemos ver claramente y como era de esperar el algoritmo Irradiance Caching en cualquiera de las dos versiones es mucho más rápido, que el algoritmo Path Tracing.

Esto se debe a que el algoritmo irradiance caching no calcula explícitamente todos los valores que se muestran sino los necesarios.

Podemos llegar a la conclusión observando los tiempos y las imágenes resultantes que el objetivo de tener un algoritmo mucho más rápido que el algoritmo path tracing ha sido cumplido con éxito.

8.2 Calidad de imagen

A continuación, compararemos nuestros algoritmos por la calidad de imagen resultante, el objetivo de este tfg era tener una imagen de calidad similar a la generada por el algoritmo path tracing, por lo tanto, vamos a comparar la calidad de imagen de estos dos algoritmos.

Primero mostraremos la imagen generada con el algoritmo path tracing, esta imagen es una imagen generada con un número de muestras de 10000, esto lo hemos hecho para tener una imagen con la mayor calidad posible esta imagen ha estado alrededor de 15 horas compilando.

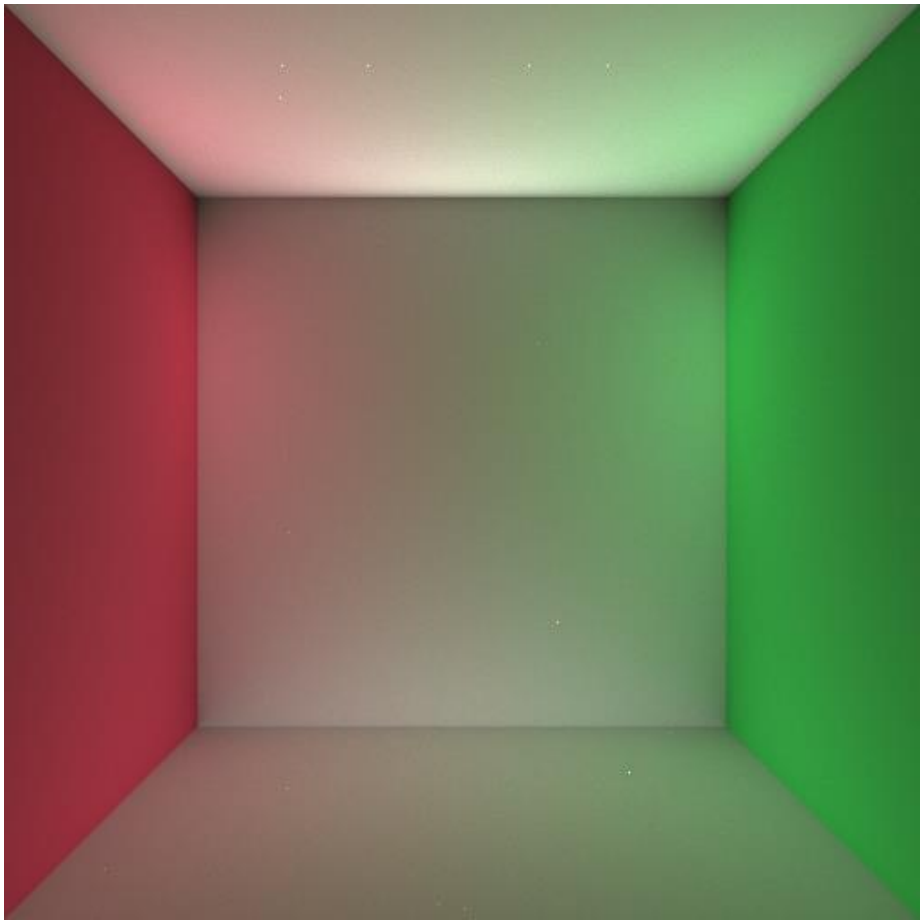

Ilustración 10000 samples

Para comparar con el algoritmo irradiance caching mostraremos 4 imágenes. Las siguientes:

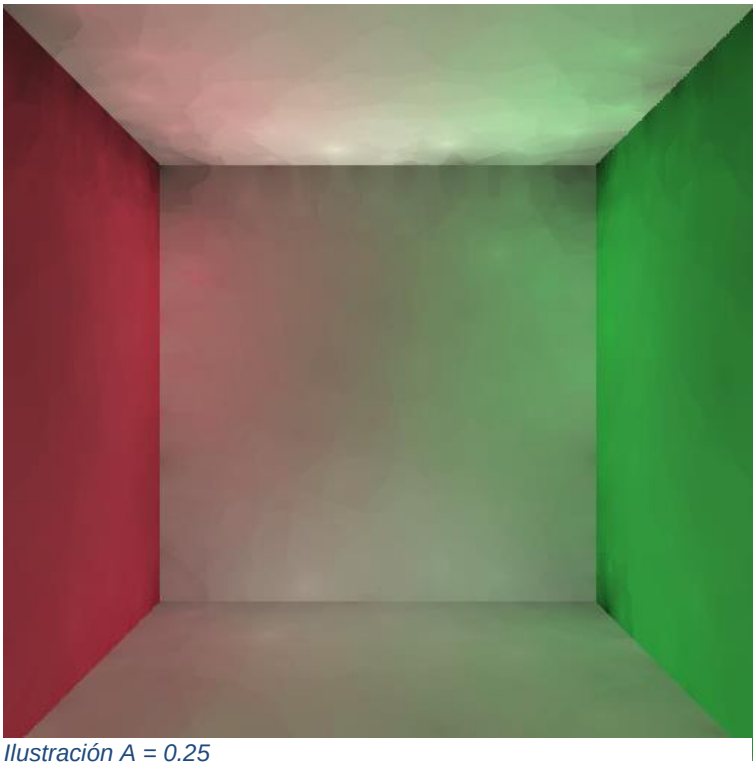

Ilustración A = 0.25

Ilustración A= 0.15

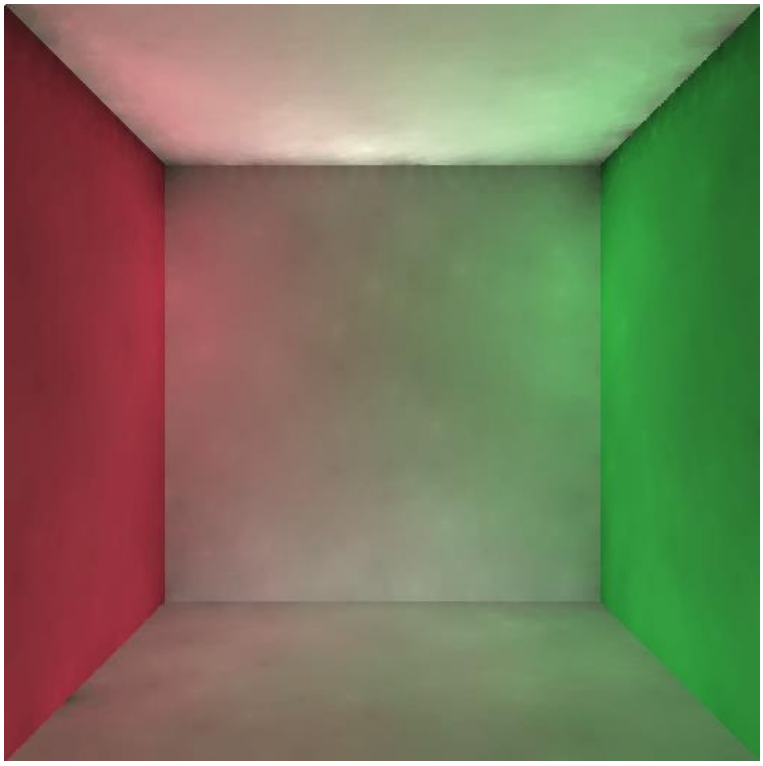

Ilustración A = 0.1

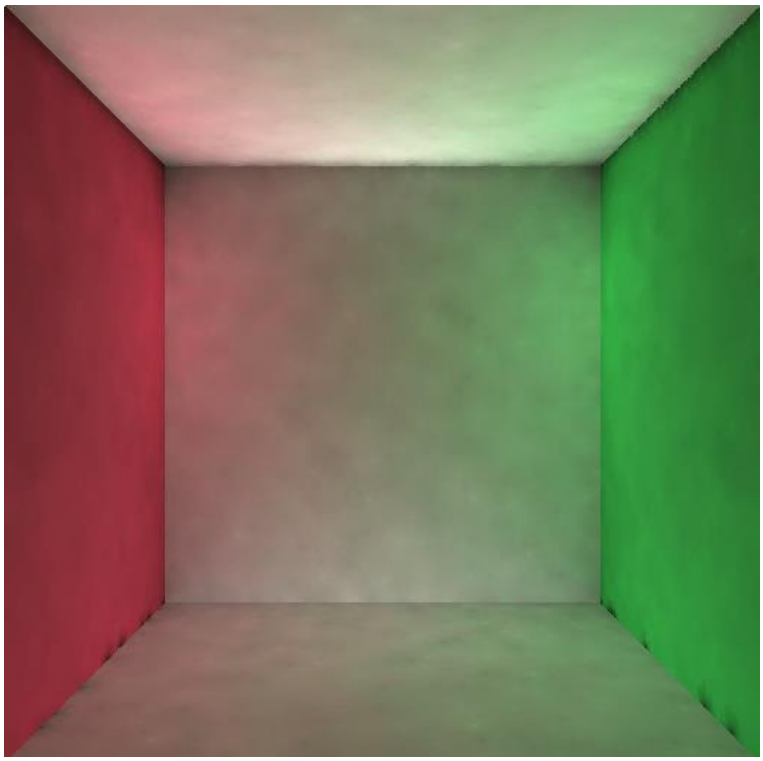

Ilustración A=0.05

Todas estas imágenes son calculadas con un error diferente, por lo tanto, la distribución de puntos es muy diferente, el resultado obtenido en la mayoría de ellas es muy satisfactorio comparada con la imagen de Path tracing, el único problema es que tenemos artefactos oscuros en la imagen número 1 y 4, esto es un problema que tenemos actualmente en nuestro algoritmo y no hemos solucionado.
Este problema se produce entorno a la distribución de puntos y tiene que ver con los puntos que se generan justo en las esquinas de las imágenes.
Fuera de este error el resultado es muy satisfactorio, tenemos una imagen muy nítida sin transferencia de colores entre paredes.

Las mejores imágenes resultantes son las imágenes 2 y 3 las cuales tienen una distribución más elevada que la imagen número 1, pero mucho menos que la imagen 4.

Con estos resultados podemos llegar a la conclusión que el segundo objetivo, que era conseguir una calidad de imagen similar al algoritmo path tracing también está superada con éxito.

8.3 Experimentos finales

En esta sección compararemos nuestros dos algoritmos haciendo que los dos hagan ejecuciones que tarden el mismo tiempo o aproximadamente el mismo tiempo, con esto veremos realmente la diferencia entre algoritmos usando el mismo tiempo.

Haremos 3 experimentos cada uno con un tiempo diferente.

La hipótesis que planteamos es que el algoritmo Irradiance Caching ganara en calidad a todas las imágenes generadas por el algoritmo Path Tracing, ya que como hemos visto en los anteriores apartados de resultados, este algoritmo es muy lento.

8.3.1 Primer Experimento: 15-20 segundos

Para este primer experimento haremos una prueba de 15 – 20 segundos, dejaremos correr las aplicaciones este rango de tiempo.

Resultados:

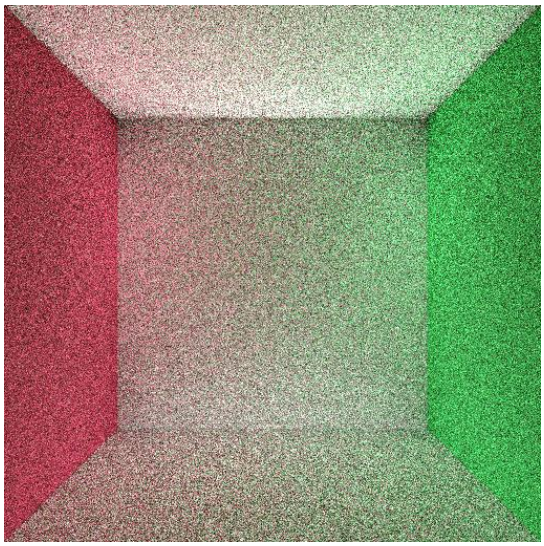

Ilustración Path Tracing 15-20 sec

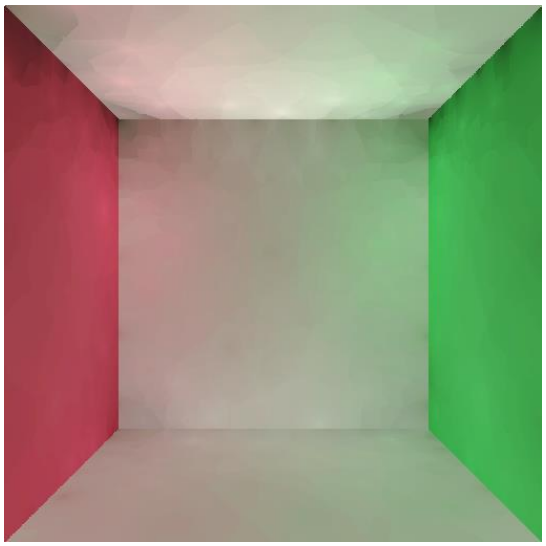

Ilustración Irradiance 15-20 sec

Como podemos ver en las imágenes la de la izquierda corresponde al algoritmo path tracing y la de la derecha corresponde al algoritmo de irradiance caching, como podemos ver el resultado es tenemos mucho mejor resultado en la imagen generada por el algoritmo de irradiance caching.

8.3.2 Segundo Experimento: 50 – 60 segundos

En este experimento dejaremos los dos algoritmos un tiempo aproximado de 50 a 60 segundos.

Resultados:

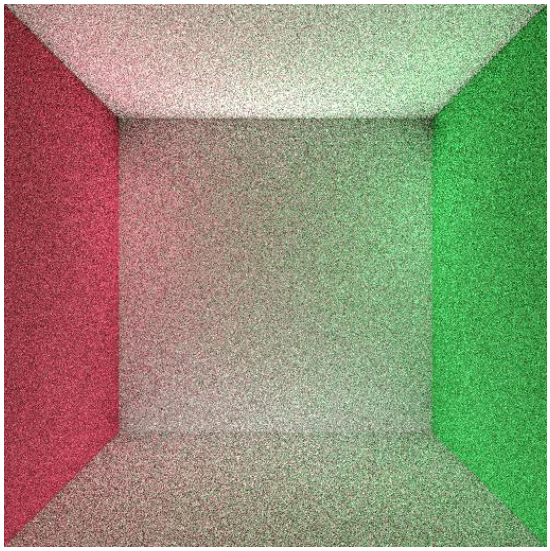

Ilustración Path Tracing 50-60 sec

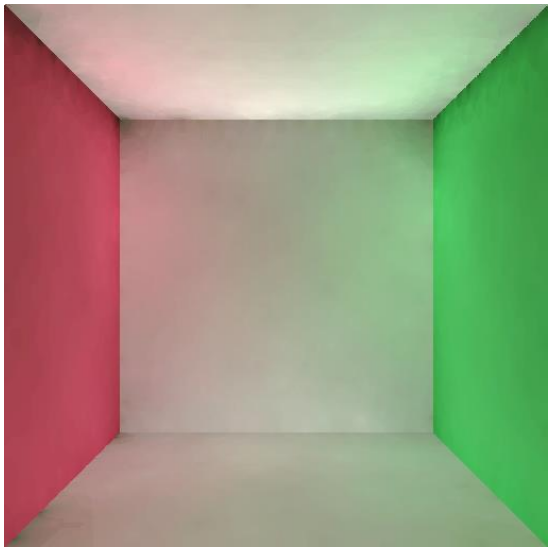

Ilustración Irradiance 50-60 sec

Como vemos en las imágenes, la imagen en este caso de irradiance caching tiene muchísima más calidad que la imagen de path tracing, exactamente igual que el caso anterior, pero en este caso vemos una calidad mucho más alta del algoritmo irradiance, mientras que path tracing no se nota apenas la mejoría.

8.3.3 Tercer experimento: 5 Minutos

Para el tercer experimento haremos unas pruebas que duraran aproximadamente 5 minutos.

Resultados:

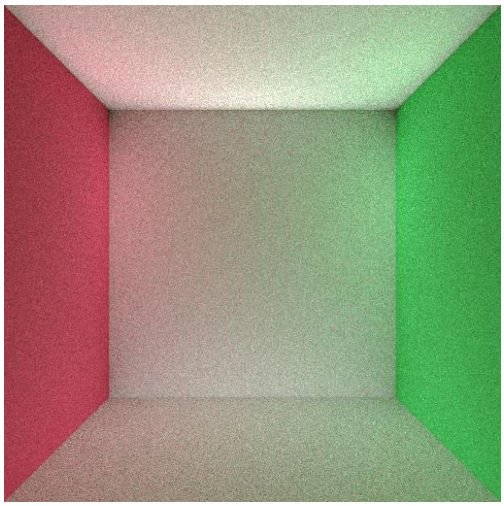

Ilustración Path Tracing 5 min

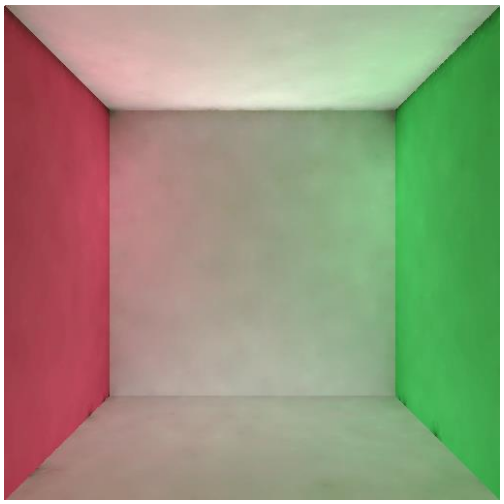

Ilustración Irradiance 5 min

En estas imágenes podemos ver que la imagen de path tracing tiene más calidad que las anteriores, pero aun podemos ver muchísimo ruido comparado con la imagen generada con irradiance caching.

Como podemos ver en los 3 experimentos en ningún caso la calidad de el algoritmo Path tracing supera la calidad del algoritmo irradiance caching, en estos tiempos, ya que como hemos visto en las imágenes y las gráficas de las anteriores secciones el algoritmo de Path Tracing.

9.Conclusiones

El objetivo principal de este tfg era realizar un algoritmo Irradiance Caching capaz de sintetizar imágenes de calidad similar al algoritmo Path Tracing con un tiempo de ejecución mucho mas rápido o en otras palabras teniendo un algoritmo mucho más eficiente.

El resultado final en la calidad de las imágenes es muy satisfactorio ya que tenemos imágenes muy similares a las imágenes generadas por el algoritmo Path tracing, tenemos algunos fallos de desarrollo como los zonales negros mostrados en algunas imágenes del apartado resultados, pero a parte de estos bugs que no he sido capaz de arreglar aun, la calidad de imagen final es muy elevada.

Por otro lado, en el aspecto de eficiencia del programa tenemos una gran mejoría respecto al algoritmo path tracing, el algoritmo de path tracing para tener una imagen con un resultado decente hemos tardado unos 30 minutos mientras que con el algoritmo irradiance caching desarrollado con 2 minutos tenemos una imagen de calidad parecida.

Por lo tanto, podemos decir que los objetivos de nuestro tfg han sido resueltos con éxito.

10. Trabajo futuro

En este proyecto podríamos continuar avanzando en 3 aspectos muy diferentes cada uno del otro.

Primero de todo y creo que primordial seria tener nuestro octree 100% funcional, esto implicaría como hemos visto unos resultados muchos más rápidos que los que tenemos actualmente, creo que este cambio es muy necesario porque como ya hemos visto en las gráficas anteriores la mejora en tiempo de nuestro algoritmo con una estructura de datos en forma de vector a una estructura de datos en forma de octree es demasiado grande como para no finalizar este octree por delante de todo.
La solución a este octree como ya he explicado antes es compartiendo u número determinado de irradiance records entre nodos del octree vecinos.

Otro posible trabajo, seria probar distintas formas de interpolación, buscar otras fórmulas para la interpolación o pensar en una propia, ya sea para mejorar el resultado visual de la imagen o para mejorar la performance del algoritmo, creo que esta posible ventana muy grande ya que se podría probar todo tipo de interpolación y estudiar las mejoras o desmejoras de estas mismas ya sean en el sentido de la eficiencia del algoritmo o en el resultado visual de la imagen.

Como última opción propondría pasar este algoritmo a GPU, con esto obtendríamos una mejora increíble en términos de eficiencia, también podríamos introducirnos en alguna librería de nvidia para realizar esto.

11. Bibliografía

Rory Driscoll,Blog irradiance-caching-part-1, 18 de enero de 2009, http://www.rorydriscoll.com/2009/01/18/irradiance-caching-part-1/

Jaroslav Křivánek (Czech Technical University in Prague) y Pascal Gautron (France Telecom R&D) y Greg Ward (Anyhere Software) y Okan Arikan (University of Texas at Austin) y Henrik Wann Jensen (University of California, San Diego), Practical Global Illumination with Irradiance Caching, Siggraph 2007, https://sgvr.kaist.ac.kr/~sungeui/SGA08/course_note/IC.pdf

Wojciech Jarosz, University of California, San Diego, Efficient Monte Carlo Methods For Light Transport in Scattering Media, Septiembre de 2008, https://cs.dartmouth.edu/~wjarosz/publications/dissertation/frontmatter.pdf

Greg Ward, Presentation Irradiance Caching Algorithm, Siggraph 2008, Agosto de 2008 https://cgg.mff.cuni.cz/~jaroslav/papers/2008-irradiance_caching_class/03-greg-ic.pdf

Pascal Gautron, Jaroslav Krivánek, Kadi Bouatouch, Sumanta Pattanaik, Eurographics Symposium on Rendering, Radiance Cache Splatting: A GPU-Friendly Global Illumination Algorithm, 2005 http://www.cs.ucf.edu/~ceh/Publications/Papers/Rendering/EGSR05GautronEtAl.pdf

Jaroslav Krivanek, Pascal Gautron Libro Practical Global Illumination with Irradiance Caching, 2009

Philip Dutre, Libro Advanced Global Illumination, Septiembre de 2006

Matt Pharr, Wenzel Jakob, and Greg Humphreys, Libro Physically Based Rendering: From Theory To Implementation, 11 de Agosto de 2004