Práctica 2: Iluminación directa por Monte Carlo

Ana Márquez Moncada

Índice

En esta práctica se ha implementado el muestreo de luces y el MIS (Multi Importance Sampling) en el fichero direct.cpp. El muestreo de material ya se implementó en la práctica 1.

Para seleccionar cada caso he implementado un enumerador.

1. Muestreo de luces de área.

Aquí he programado el integrador con muestreo por importancia de la iluminación. Voy a comentar las principales funciones que he implementado, así como los resultados.

1.1. Función Li (direct.cpp)

El color resultante es el resultado de aplicar la fórmula:

$$Le + BSDF * Li * cos \tag{1}$$

Donde Le es el color de la componente emisiva de la luz, BSDF es el color del material, Li es el color del segundo rayo lanzado internamente que después de interseccionar con un material, y cos es el coseno de estos 2 últimos.

Se comienza lanzando un rayo desde la cámara y se comprueba si se ha producido una intersección. Si no se produce, no se calcula nada y se devuelve 0.

Si se produce, llamo al *sampleEmmiter()*, pasándole como parámetro un *EmitterQueryRecord* que he implementado con el punto de origen del rayo (*ray.p*), destino (*its.p*) y la normal. Dentro del *sampleEmmiter()* obtendré una radianza, a la que he llamado *colorMuestra*, que es el color del segundo rayo que se ha lanzado internamente. Equivale al Li de la fórmula.

Una vez que he hecho el sampleado del rayo, sampleo el material creando un *BSDFQueryRecord* y llamando a *getBSDF()- eval*. Este color lo guardo en *colorBSDF*. Aquí ya tengo en cuenta el segundo rayo.

Una vez que tengo esos colores, calculo el color final, para lo que se multiplica esta radianza por el valor de la muestra de luz y el coseno de ambos.

Por último compruebo si es emiter, y en caso afirmativo le sumo su componente emisiva al color final (Le). En este caso utilizo un *EmitterQueryRecord* con la información del primer rayo, ya que el que estaba usando la he modificado en la función *sampleEmitter()*

1.2. Función SampleEmitter (en scene.cpp)

Lanzo un segundo rayo desde el punto de intersección donde el primer rayo ha chocado en la escena. Este rayo se lanza hacia una luz de la escena.

Antes de lanzar el rayo hay que decidir qué luz de la escena contra la que hay que lanzar el rayo. Hay que seleccionar una de las luces, y de esas luces hay que seleccionar un triángulo (en la función *sample()*.

Primero Selecciono uno de los posibles emisores de la escena de forma aleatoria y lo guardo en *lRec.emitter*. Llamo a la función *sample()* que muestrea ese emisor. Así tengo el punto de destino del rayo. Estos valores se les asigna al *EmiterQueryRecord*.

Por último compruebo la visibilidad de la muestra del emisor, lanzando un rayo desde la muestra de luz al punto de la superficie. Si no es visible devuelvo 0.

1.3. Función sample (en area.cpp)

Primero Obtengo un punto aleatorio en la malla asociada al emisor. Para ello he creado la función *samplePoint()* (en *mesh.cpp*), que muestrea un triángulo, además de obtener sus vértices y normales en coordenadas baricéntricas. En esta función se usa s*quareToTent* para seleccionar un triángulo de

esa maya. También evalúo la probabilidad de la muestra desde el punto de vista del emisor (con probabilidad uniforme basada en el área).

Luego, esa probabilidad hay que transformarla al ángulo sólido del punto de destino de la luz.

Por último, evalúo la radiancia y ponderarla con la inversa de la probabilidad.

Además, hay que comprobar la orientación relativa entre el rayo incidente en la luz y la normal de la malla. Si el rayo incide por la parte interior de la malla, simplemente se devuelvo radiancia nula.

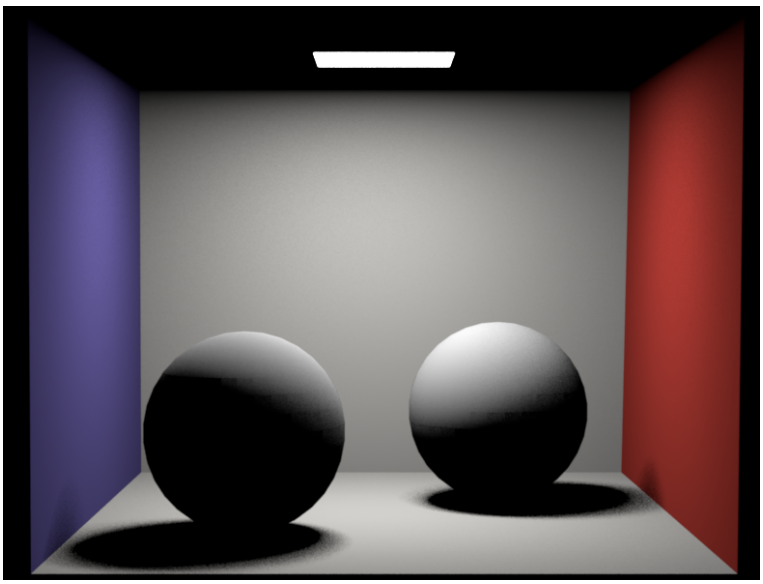

Figura 1: Muestreo de luces de área

2. Muestreo de emisores con probabilidad asociada a la radianza

En esta parte se modifica el *sampleEmitter()* para que el emisor se seleccione con una probabilidad proporcional a la radianza total. Para ello, utilizo *sampleReuse()* de discretePDF que hay que inicializar con el $areatotal * radianza$ en la función *scene::activate* para cada emisor.

Para coger la radianza de un emisor, añadí una función para obtener la radianza, y la declaro en la clase *area.cpp*, que es la clase hija de emiter. El emiter es un polimorfismo de *area.cpp*. En *area.cpp* tengo el *m_radiance*, y lo que hace la nueva función es devolver esa variable. Teniendo esta variable, llamo a *getLuminance()*, que devuelve la radianza.

El área total está en la clase mesh. Necesito una variable que vaya calculando y almacenando el valor del área de cada triángulo de la malla para tener el área total de la malla. La almaceno en la variable *m_area*, y el área la obtengo mediante la función *surfaceArea()*.

También he declarado otras funciones que necesitaba, get y set para el área de la malla, declaradas en la clase *area.cpp*. Así cuando vaya a rellenar los valores, a ese emiter le asigno su área y una vez hecho esto ya se puede aplicar la fórmula llamando también a *getArea()*.

En la siguiente figura se puede ver la diferencia con el muestreo uniforme. Se nota como el emmisor de la derecha genera más luz (más radianza).

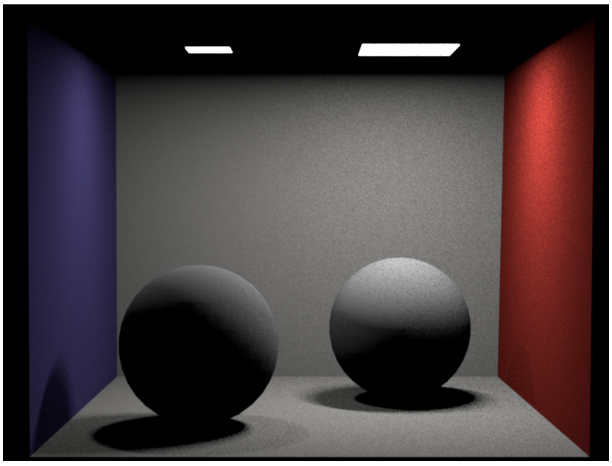
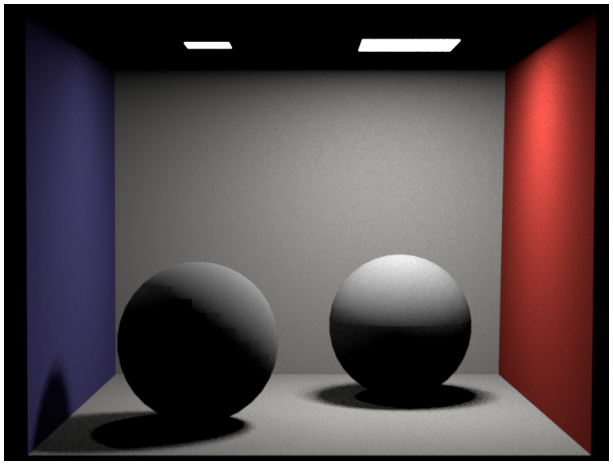

Figura 2: Misma escena con 2 emisores de distinto tamaño. A la izquierda con muestreo uniforme de emisores, a la derecha con muestreo basado en radianza

3. Multi Importance Sampling

El MIS hace un sampleo de material y un sampleo por luz, y pondera ambos mediante *balance heuristic*. Hay que aplicar la fórmula

$$\frac{(pdfMat * colorMat + pdfLuz * colorLuz)}{(pdfMat + pdfLuz)} \tag{2}$$

Donde pdfMat y pdfLuz son las probabilidades del muestreo de material y del de luces, y colorMat y colorLuz son los colores obtenidos tras realizar sampleo de material y de luces.

Para ello hay que quedarse con el color de cada parte y la pdf de cada parte. También hay que sumarle la Le.

A continuación se compara el muestreo de luces con MIS. Se puede apreciar claramente la mejora al realizar MIS.

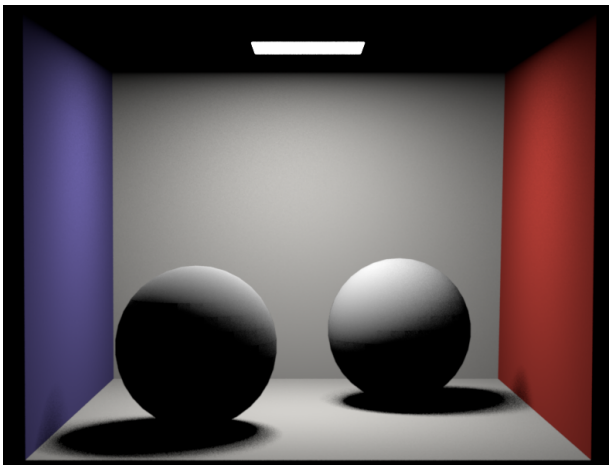
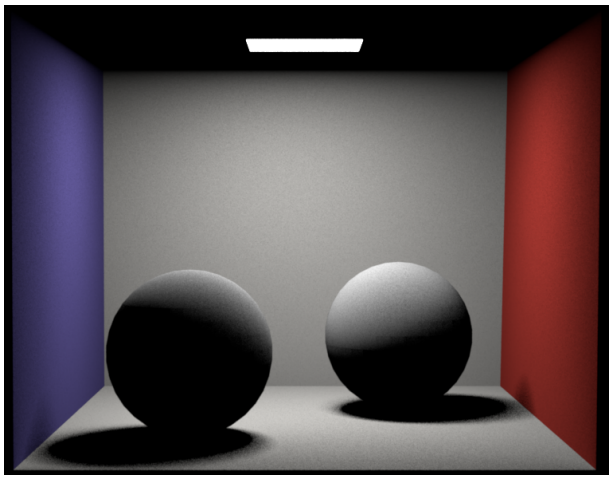

Figura 3: Comparación de la misma escena con muestreo de luces (izquierda) y MIS (derecha)